

Inhalt

3 Elementare Datentypen	2
3.1 Numerische Datentypen	3
3.2 Darstellung Ganzer Zahlen (auch negative Dualzahlen)	6
Ganze Zahlen in Python	8
3.3 Gleitpunktzahlen (floating point number)	10
Das IEEE 754- und IEEE 754r-Format	11
Umwandlungen: Dezimalzahl ↔ Gleitpunktzahl	13
Eigenschaften von Gleitpunktzahlen	14
Gleitpunktzahlen in Python	16
3.4 Zusammenfassung und Ergänzungen zu numerischen Datentypen	17
3.5 Boolescher Datentyp	17
Repräsentation	17
Operationen auf Booleschen Daten	18
Negation (logisches NICHT, <i>not</i>)	18
Disjunktion (logisches ODER, <i>or</i>)	19
Vergleichsoperatoren in Python	22
3.6 Datentypen für Zeichen und Zeichenketten (Text)	22
Zeichensätze – Text-Repräsentationen	25
ASCII (American Standard Code for Information Interchange)	25
ASCII – Nationale Varianten, firmenspezifische Varianten, ISO/IEC 8859	27
Unicode	27
Zeichenketten (Strings) in Python	31
Vergleichsoperatoren	33
3.7 Typisierung	34
Starke und Schwache Typisierung	35
Statische und Dynamische Typisierung	36
Typwandlung	37
Funktionen zur Typwandlung in Python	38

3 Elementare Datentypen

Dieses Kapitel behandelt elementare Datentypen. Elementare Datentypen sind solche, die von der Programmiersprache direkt unterstützt werden und meist direkt auf einfache Speicherzellen abgebildet sowie durch entsprechende Hardwarekomponenten effizient behandelt werden können. Neben den Numerischen Datentypen Ganzzahlen (Integer) und Gleitkommazahlen (Floating Point) sind dies Boolesche Wahrheitswerte und Zeichenketten (Strings). Zum Inhalt gehört auch die Wandlungen zwischen verschiedenen Repräsentationen. Ziel ist es, diese Datentypen beim Programmieren beherrschen zu lernen und ihre Eigenarten zu kennen. Besonders wichtig und sehr unterschiedlich in verschiedenen Programmiersprachen ist das sogenannte Typsystem, starke versus schwache Typisierung; statische versus dynamische Typisierung. Die Unterschiede werden klar gestellt und eine Einordnung von Python erfolgt.

Für einen Computer (*Rechner*) ist die interne Repräsentation von **Zahlen** und (*Schrift*)-**Zeichen** sicherlich elementar – dies ist ja geradezu sein Zweck. Jede übliche Programmiersprache stellt hierfür elementare Datentypen (*in Python Built-in Types genannt*) zur Verfügung.

Prinzipiell würde es ausreichen, diese Datentypen abstrakt zu beschreiben, um mit ihnen umzugehen. In einigen Fällen ist es aber wichtig, die konkrete Repräsentation (*Rechnerintern*) eines solchen Datentyps zu kennen – zumindest die Prinzipien, denn aus dieser Kenntnis lassen sich Eigentümlichkeiten dieser konkreten Datentypen verstehen und ableiten, die abstrakt beschrieben viel zu kompliziert würden. Bei Formulierungen in Programmiersprachen versucht man dabei, die üblichen Notationen der Mathematik weitgehend beizubehalten, in der Regel die angloamerikanischen Konventionen. Es lohnt sich daher, die mathematischen Grundlagen kurz zu rekapitulieren, bevor wir die Rechnerrepräsentationen betrachten. Aus der Mathematik sind uns insbesondere folgende Zahlenmengen geläufig:

Die Menge der Natürlichen Zahlen	N oder \mathbb{N}
Die Menge der Ganzen Zahlen	Z oder \mathbb{Z}
Die Menge der Rationalen Zahlen	Q oder \mathbb{Q}
Die Menge der Reellen Zahlen	R oder \mathbb{R}
Die Menge der Komplexen Zahlen	C oder \mathbb{C}
Die Menge der Quaternionen	H oder \mathbb{H}

Allgemein gilt: $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C} \subset \mathbb{H}$ (*echte Teilmengen*)

Dabei schreiben wir diese Zahlen gewöhnlich als vorzeichenbehaftete Dezimalzahl, also in einem Stellenwertsystem zur Basis 10. Beispiele mit einer eigenen Stelle für das Vorzeichen (+ oder –) sind:

0 – 42 1,414 3,141 – 1,59 1,43 · 10⁶

oder

0 – 42 1.141 3.141 – 1.59 1.43 · 10⁶

Anstelle des Kommas als Dezimaltrenner, verwendet man im angloamerikanischen Raum den Punkt – diese Punkt-Schreibweise findet sich in allen Programmiersprachen wieder! Insbesondere für die Zahlenmengen \mathbb{N} , \mathbb{Z} , \mathbb{Q} und \mathbb{R} benötigen wir geeignete rechnerinterne Kodierungen (Darstellungen), und zwar solche, auf denen die üblichen Operationen effizient und möglichst so wie gewohnt ausgeführt werden können. Das geht leider nur eingeschränkt.

3.1 Numerische Datentypen

Ganze Zahlen (Integer)

Im Sprachgebrauch der Informatik werden Ganze Zahlen häufig als Ganzzahlen bezeichnet. Gemeint ist hiermit die Möglichkeit Ganze Zahlen in bestimmten Wertebereichen zu repräsentieren. Dies erfolgt in Dualzahlendarstellung, als eine Folge von *Nullen* und *Einsen*.

Dualzahlendarstellung für Natürliche Zahlen

In einem (*Digitalrechner*) werden alle Daten als Bits repräsentiert, also letztlich als Folgen von Nullen und Einsen. Für Natürliche Zahlen \mathbb{N} bietet sich hierzu die Darstellung als Dualzahl (*Binärzahl*) an anstelle der Basis 10 (wie gewöhnlich) benutzen wir die Basis 2.

Die Dyadik (*dyo, griech. = Zwei*), also die Darstellung von Zahlen im Dualsystem, wurde schon Ende des 17. Jahrhunderts von Leibniz entwickelt.¹ Das Dualsystem ist ein vollständiges Zahlensystem mit der geringstmöglichen Anzahl an verschiedenen Ziffern. Es wird durch die Ziffern $z_i \in \{0,1\}$ repräsentiert. Die Ziffern werden wie im Dezimalsystem ohne Trennzeichen hintereinander geschrieben, ihr Stellenwert entspricht allerdings der zur Stelle passenden Zweierpotenz und **nicht** der Zehnerpotenz. Es wird also die höchstwertige Stelle mit dem Wert z_m ganz links und die niederwertigeren Stellen mit den Werten z_{m-1} bis z_0 in absteigender Reihenfolge rechts davon aufgeschrieben:

$$z_m z_{m-1} \cdots z_1 z_0$$

Der Wert Z der Dualzahl ergibt sich aus der Summe der mit ihrem Stellenwert 2^i multiplizierten Einzelziffern, für eine Dualzahl mit $m + 1$ Stellen also:

$$Z = \sum_{i=0}^m z_i \cdot 2^i$$

Beispiel: Die Ziffernfolge **1** , stellt nicht (*wie im Dezimalsystem*) die „Tausendeinhundertundeins“ dar, sondern die Dreizehn, denn im **Dualsystem** berechnet sich der Wert durch:

$$\begin{aligned} [1101]_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= [13]_1 \end{aligned}$$

Die Klammerung der Resultate mit der tiefgestellten 2 beziehungsweise der 10 gibt die Basis des verwendeten Stellenwertsystems an. In der Literatur werden die eckigen Klammern oft weggelassen und die tiefer gestellte Zahl wird dann oft in runde Klammern gesetzt, also:

$$1101_{(2)} = 13_{(1)}$$

¹ Leibnitz sah das Dualsystem ein besonders überzeugendes Sinnbild des christlichen Glaubens an. So schrieb er an den chinesischen Kaiser Kangxi: "Zu Beginn des ersten Tages war die 1, das heißt Gott. Zu Beginn des zweiten Tages die 2, denn Himmel und Erde wurden während des ersten geschaffen. Schließlich zu Beginn des siebenten Tages war schon alles da; deshalb ist der letzte Tag der vollkommenste und der Sabbat, denn an ihm ist alles geschaffen und erfüllt, und deshalb schreibt sich die 7 111, also ohne Null. Und nur wenn man die Zahlen bloß mit 0 und 1 schreibt, erkennt man die Vollkommenheit des siebenten Tages, der als heilig gilt, und von dem noch bemerkenswert ist, dass seine Charaktere einen Bezug zur Dreifaltigkeit haben."

Angenommen, wir haben statt einer Dualzahl eine Dezimalzahl gegeben, wie funktioniert nun die Umrechnung der Dezimalzahl $Z_{(1)}$ in eine Dualzahl $Z_{(2)}$? Aus der obigen Darstellung können wir uns leicht das erste Verfahren herleiten.

Verfahren 1:

Schritt 1: Man berechne alle Potenzen der Basis $b = 2$ bis die größte Potenz größer als die umzuwandelnde Zahl ist, beispielsweise die $23_{(1)}$.

Also:

$$2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \quad 2^3 = 8, \quad 2^4 = 16, \quad 2^5 = 32$$

Schritt 2: Man zerlegt die umzuwandelnde Zahl schrittweise in Potenzen der Basis b

$$\begin{array}{rcl} 23_1 & = & 1 \cdot 2^4 + R_7 \\ 7_1 & = & 0 \cdot 2^3 + R_7 \quad (8 \text{ i } z \text{ g } \beta) \\ 7_1 & = & 1 \cdot 2^2 + R_3 \\ 3_1 & = & 1 \cdot 2^1 + R_1 \\ 1_1 & = & 1 \cdot 2^0 + R_0 \end{array}$$

Aus dieser Zerlegung können wir von oben nach unten die Dualzahl $10111_{(2)}$ einfach ablesen. Interessant ist, dass sich jede Zahl auch folgendermaßen ausdrücken lässt (*eine spezielle Faktorisierung*):

$$\begin{aligned} 23_1 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 2 \cdot (2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) + 1 \cdot 2^0 \\ &= 2 \cdot (2 \cdot (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) + 1 \cdot 2^0) + 1 \cdot 2^0 \\ &= 2 \cdot (2 \cdot (2 \cdot (1 \cdot 2^1 + 0 \cdot 2^0) + 1 \cdot 2^0) + 1 \cdot 2^0) + 1 \cdot 2^0 \\ &= 2 \cdot (2 \cdot (2 \cdot (2 \cdot (1 \cdot 2^0) + 0 \cdot 2^0) + 1 \cdot 2^0) + 1 \cdot 2^0) + 1 \cdot 2^0 \\ &= 2 \cdot (2 \cdot (2 \cdot (2 \cdot (\mathbf{1}) + \mathbf{0}) + \mathbf{1}) + \mathbf{1}) + \mathbf{1} \end{aligned}$$

Wie wir sehen, kann man die dezimal Zahl „23“ auch in mehreren Schritten durch Addition mit „0“ bzw. „1“ sowie durch Multiplikation mit „2“ (*der Basis des Zahlensystems*) erhalten. Die gesuchte Dualzahl $10111_{(2)}$ ist auch in dieser Darstellung offensichtlich (*siehe letzte Zeile*). Wir können den Ausdruck auch umgekehrt formulieren. Die Stellen der Binärzahl erhalten wir, indem wir die Dezimalzahl fortgesetzt durch 2 teilen und die Reste „0“ bzw. „1“ als Binärstellen interpretieren. Dies führt uns zum zweiten Verfahren.

Verfahren 2:

Man errechnet die gesuchte Darstellung durch Fortgesetzte Division mit Rest durch die Basis b . Der erste Rest ergibt die Stelle z_0 der zweite z_1 usw.

$$\begin{array}{rcl} \frac{23}{2} & = & 11 \text{ R } \mathbf{1} \text{ (Least Significant Bit (LSB))} \\ \frac{11}{2} & = & 5 \text{ R } \mathbf{1} \\ \frac{5}{2} & = & 2 \text{ R } \mathbf{1} \\ \frac{2}{2} & = & 1 \text{ R } \mathbf{0} \\ \frac{1}{2} & = & 0 \text{ R } \mathbf{1} \text{ (Most Significant Bit (MSB))} \\ & \Rightarrow & 23_{(1)} = 10111_{(2)} \end{array}$$

Dieses zweite Verfahren lässt sich sehr effizient als Programm implementieren. Beachten Sie: Bei jedem Stellenwertsystem lässt sich die **Multiplikation mit der Basis** durch ein Verschieben (*shiften*) der Repräsentation um eine Stelle nach links und dem Nachziehen einer 0 realisieren:

$$\begin{aligned} 10 \cdot 0013_{(1)} &= 0130_{(1)} \\ 2 \cdot 0110_{(2)} &= 1100_{(2)} \end{aligned}$$

Genauso einfach ist die **ganzzahlige Division (ohne Rest) durch die Basis**, dies ist einfach ein Verschieben der Zahl nach rechts:

$$\begin{aligned} 135_{(1)} // 10 &= 013_{(1)} \\ 1011_{(2)} // 2 &= 0101_{(2)} \end{aligned}$$

Bei allen Stellenwertsystemen lassen sich die uns bekannten Verfahren für die Addition und Multiplikation ganz analog anwenden:

$$\begin{aligned} 0110_{(2)} &= 6_{(1)} \\ + 0011_{(2)} &= 3_{(1)} \\ = 1001_{(2)} &= 9_{(1)} \end{aligned}$$

nach den sehr einfachen Rechenregeln:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0 \text{ und } 1 \text{ im Übertrag} \end{aligned}$$

Somit errechnet sich

$$0110_{(2)} \cdot 0101_{(2)} = 6_{(1)} \cdot 5_{(1)}$$

als

$$\begin{aligned} &011000_2 \\ + &000110_2 \\ = &011110_2 = 30_1 \end{aligned}$$

nach dem sehr einfachen „*kleinen Einmaleins*“ für Dualzahlen

$$\begin{aligned} 0 \cdot 0 &= 0 \\ 0 \cdot 1 &= 0 \\ 1 \cdot 0 &= 0 \\ 1 \cdot 1 &= 1 \end{aligned}$$

Das Rechnen mit Dualzahlen ist also sehr einfach. Neben den Dualzahlen ist es in der Informatik auch üblich, mit Zahlen zu anderen Basen umzugehen, nämlich Oktanzahlen (*zur Basis 8*) mit den Ziffern $(0, \dots, 7)$ und Hexadezimalzahlen (*zur Basis 16*) mit den Ziffern $(0, \dots, 9, A, B, C, D, E, F)$: $A_1 = 10_1 \dots F_1 = 15_1$.

Die Verfahren zur Umrechnung zwischen der Basis 10 und anderen Basen ist ganz analog zu dem oben gezeigten für Dualzahlen. Da 8 und 16 jeweils ganzzahlige Potenzen von 2 sind ergibt sich zur Umrechnung zwischen *Hexadezimal* oder *Oktanzahlen* und Dualzahlen ein sehr einfaches Verfahren: Es werden jeweils 4 Dualziffern zu einer Hexadezimalziffer zusammengefasst. Für das Oktalsystem entsprechend 3 Dualziffern.

$$010111010010_2 = \underbrace{0101}_5 \underbrace{1101}_D \underbrace{0010}_2 = 5D2_1$$

3.2 Darstellung Ganzer Zahlen (auch negative Dualzahlen)

Hier bietet sich zunächst ein analoges Vorgehen an, wie das, welches wir vom Dezimalsystem her gewöhnt sind. Wir spendieren eine Vorzeichenstelle und kodieren diese mit 0 für positive Zahlen und 1 für negative Zahlen, siehe **Fehler! Verweisquelle konnte nicht gefunden werden.** für ein 8-Bit langes Speicherwort.

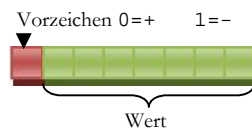


Abbildung 1 Darstellung einer Ganzen Zahl in einem Byte (8 Bit) mit Vorzeichenbit

Wir haben damit für m Stellen nur noch $m-1$ Stellen für die Darstellung des Wertes, problematischer aber ist, dass die Darstellung der Null nicht eindeutig ist: es gibt $+0$ und -0 . Zudem müssen wir neben einen Algorithmus für die Addition auch einen anderen für die Subtraktion bereitstellen (und ggf. entsprechende Hardware). Wir würden gern die Rechenregel $a - b = a + (-b)$ nutzen, um nur die Addition und die Invertierung implementieren zu müssen. Dies geht, wenn wir ein sogenanntes „Komplement“ benutzen.

Das Einer-Komplement zur Darstellung einer negativen Zahl $-x$ mit der Stellenzahl m entsteht dadurch, dass wir die positive Zahl x in m Stellen dual kodieren und dann einfach alle Stellen invertieren:

-3 im 4-Bit-Einerkomplement wird also:

$$\begin{aligned} 3_1 &= 0011_2 \\ -3_1 &= 1100_2 \end{aligned}$$

Leider hat diese Kodierung nicht all die Eigenschaften, die wir erwarten:

1. wir haben eine Doppelkodierung der 0 ($+0$ und -0)
2. Tritt ein Überlauf bei der Addition auf, d.h. wird die $(m+1)$ -te Stelle bei der Addition 1, so muss das Additionsergebnis korrigiert werden, indem $+1$ zusätzlich addiert wird.

Dies ist in Hardware unangenehm und aufwändig zu implementieren und lässt sich leicht korrigieren. Praktisch alle Implementierungen nutzen heute daher das **Zweier-Komplement** welches den geforderten Ansprüchen genügt.

Tabelle 1: Kodierung Ganzer Zahlen im Einer- und Zweier-Komplement für 4 Bit Wortlänge

Kodierung	Dezimalwert des		Anmerkungen
	Einerkomplement	Zweierkomplement	
...	Overflow
1010	-5	-6	
1001	-6	-7	
1000	-7	-8	
0111	7	7	$2^{n-1}-1$
0110	6	6	
0101	5	5	
0100	4	4	
0011	3	3	
0010	2	2	

0010	2	2	
0001	1	1	
0000	0	0	
1111	-0	1	
1110	-1	2	
1101	-2	3	
1100	-3	4	
1011	-4	5	
1010	-5	6	
1001	-6	7	
1000	-7	8	-2^{n-1}
0111	7	+7	Underflow
0110	6	+6	
0101	5	+5	
...	

Konzeptionell bilden Komplement-Zahlen einen Zahlenzylinder: Beim Übergang von 0111_2 nach 1000_2 stoßen die maximale positive Zahl und die größte negative Zahl beim „Hochzählen“ aneinander. Damit kann man anstelle einer Subtraktion von 3 auch $16 - 3 = 13$ addieren.

Mit n Bits lassen sich im Zweier-Komplement negative und positive Zahlen von -2^{n-1} bis $2^{n-1} - 1$ darstellen, also z.B. ein Wertebereich von $-2.147.483.648_1$ bis $+2.147.483.647_1$ für $n = 32$ Bit. Ein potentiell Problem, das dabei auftreten kann, ist der sogenannte Überlauf bzw. Unterlauf (Overflow, Underflow). Er tritt immer dann auf, wenn zwei Zahlen verrechnet werden, mit einem Ergebnis, das außerhalb des Wertebereiches liegt.

Wie berechnen wir eine Zweier-Komplement Darstellung? Voraussetzung ist eine feste Stellenzahl, sagen wir etwa $n = 8$. Die Umrechnung erfolgt dann in drei Schritten.

1. Die Zahl wird ungeachtet des Vorzeichens (unter Beachtung des Wertebereichs) in das Dualzahlensystem übertragen. Für positive Zahlen **und** die kleinste negative Zahl (-2^{n-1}) sind wir fertig.
2. Für alle anderen negativen Zahlen bilden wir das Einer-Komplement durch Invertieren aller Stellen der Dualzahldarstellung.
3. Zu der so errechneten Zahl addieren wir im dritten und letzten Schritt eine 1 (Addition von 1; oder um eins hochzählen).

Beispiel: zur Umwandlung einer Dezimalzahl ins Zweierkomplement, hier -4_1 mit einer Wortlänge von 8 Bit:

1. Vorzeichen ignorieren und ins Binärsystem umrechnen: $4_1 = 00000100_2$
2. Invertieren der Darstellung: 11111011_2
3. Eins addieren: $11111011_2 + 00000001_2 = 11111100_2$

Wir bemerken, dass das höchstwertige Bit (Most Significant Bit) eine „1“ ist und damit eine negative Zahl darstellt. Zu diesem Algorithmus „Invertieren und Addieren“ (Punkte 2 und 3) gibt es eine Alternative, nämlich:

Beginnen Sie mit der niederwertigsten Stelle z_0 : Ist dies eine Null, gehen Sie weiter zur nächsten Stelle z_1 und so fort, bis Sie eine 1 erreichen. Diese lassen Sie so, aber alle weiteren Stellen bis zum höchstwertigen Bit z_n invertieren Sie, ungeachtet ob es Einsen oder Nullen sind. Damit haben Sie Punkt 2 und 3 ebenfalls ausgeführt.

Wenn man eine Zahl vom Zweierkomplement ins Dezimalsystem umrechnen will, kehrt man das oben dargelegte Verfahren um. Falls das erste Bit 1 ist (negative Zahl), subtrahieren wir Eins und invertieren die resultierende Bit Folge. Eine Umrechnung ins Dezimalsystem liefert die gewünschte Darstellung.

Mit der hier vorgestellten Kodierung lassen sich alle Grundrechenarten effizient realisieren.

Wir sind hier ausnahmsweise einmal in Hardware-Realisierungsdetails „abgestiegen“. Algorithmen auf diesem Niveau sind „*nicklich*“, leider oft auch trickreich. In aller Regel versuchen höhere Programmiersprachen von diesen Details zu abstrahieren (d.h. die Programmiererin muss sie gar nicht kennen), das Zweier-Komplement ist allerdings fundamental für die Integer-Darstellung auch in höheren Programmiersprachen.

Auf ein weiteres Problem sei noch kurz verwiesen: **Little und Big Endian**. Leider ist die Darstellung von Zahlen im Computerspeicher nicht einheitlich: Sie hängt vom Rechnertyp ab. Grund dafür ist die Organisation der Speicherelemente. Üblicherweise werden jeweils 8 Bits zu einer Einheit, dem Byte, zusammengefasst und als kleinste Einheit angesprochen („*adressiert*“). Mehrere Bytes bilden dann zusammen einen Speicherbereich, das Speicherwort, das heutzutage meist 32 Bit (4 Byte) oder 64 Bit (8 Byte) umfasst. Allerdings unterscheiden sich die Prozessoren in den Rechnern durch die Reihenfolge, in der sie die Bytes eines Speicherwortes belegen. Wenn zum Beispiel eine 32 Bit-Binärzahl, hier in Hexadezimal-Schreibweise $A4B3C2D1_{(16)} = -615.760.594_{(10)}$, im Hauptspeicher abgelegt wird, so gibt es zwei verschiedene Haupt-Varianten, sie in 4 Byte abzulegen:

Tabelle 2: Gebräuchliche Byte-Reihenfolgen (Byte order)

Adresse	little endian	big endian
x...x00	D1	A4
x...x01	C2	B3
x...x10	B3	C2
x...x11	A4	D1

Wenn wir die 4 Bytes in der Reihenfolge **D1C2B3A4**, also das niederwertigste Byte an der niedrigsten Speicheradresse ablegen, spricht man von **little endian**. Die Speicherung in der Reihenfolge **A4B3C2D1**, also das höchstwertige Byte an der niedrigsten Speicheradresse, wird als **big endian** bezeichnet.

Auf Programmiersprachenebene in einem Programm hat diese Unterscheidung keine Bedeutung, weil der Interpreter und die Hardware dies immer gleich handhaben werden. Relevant wird die Unterscheidung aber beim Datenaustausch über Netzwerke oder über Speichermedien, wenn man Binär-Daten (Zahlen) von einem Rechner auf den anderen übertragen will.

Im Internet ist das Schema des **big-endian** als *Network Byte Order* festgelegt. Die sogenannte *Host Byte Order*, das Schema auf dem Rechner selbst, ist aber bei heute gebräuchlichen Systemen verschieden und hängt vom Prozessortyp ab:

- Little Endian Intel-x86-Prozessoren und auch das Betriebssystem Windows
- Big Endian Power PC (umschaltbar), Motorola-68000-Familie, MIPS Prozessoren, HP-UX, Internet Protokoll (IP)

Je nach Rechnertyp muss also dafür gesorgt werden, dass alle Zahlen vor einer Übertragung im Internet auf **big endian** konvertiert werden – oder auch nicht.

Ganze Zahlen in Python

In allen höheren Programmiersprachen versucht man von diesen Hardware-Nicklichkeiten zu abstrahieren und den Programmierern ein einfaches Modell anzubieten.

Manche Programmiersprachen unterscheiden je nach Wortlänge verschiedene Integer-Datentypen, z.B. für eine systemnahe Sprache wie C die Typen

`char`, `short int`, `int`, `long int` und (seit C99) `long long int`

Zu all diesen Typen existieren noch vorzeichenlose Typen, die durch ein vorangestelltes `unsigned` notiert werden. Also insgesamt 10 verschiedene Integer-Typen. Dabei ist nicht festgelegt, wie viele Bits der Basistyp `int` hat, heute meist 32 Bit oder 64 Bit. Festgelegt sind aber Mindestgrößen. Auf einem 32-Bit Rechner wäre folgendes typisch:

<code>char</code>	8 Bit
<code>short int</code>	16 Bit
<code>int</code>	32 Bit
<code>long int</code>	64 Bit
<code>long long int</code>	128 Bit

Wirklich Bedeutung haben diese Unterschiede eigentlich nur, wenn man gezwungen ist, möglichst optimalen Code (Speicherverbrauch und Laufzeit) zu schreiben. Viele andere Sprachen abstrahieren davon und haben nur ein oder zwei verschiedene Integer-Typen, so auch Python.

In Python werden ab Version 3.x (von uns genutzt) nur noch **ein Integer-Format** unterstützt.

- Integer (mindestens 32 Bit) `<type 'int'>`
kann „beliebig“ groß werden (Grenze ist der Speicher)

Integer Typen werden immer als vorzeichenbehaftet (signed) im Zweier-Komplement kodiert betrachtet. Bei größeren Integer als das Maschinenwort steht die kennzeichnende 1 konzeptionell beliebig weit „links“.

Literale: Integer-Zahlen können im Programm auf folgende Art notiert werden:

1234, -42, 0	ohne Kennzeichnung werden Dezimalzahlen angenommen
1234L oder (1234l)	mit dem Postfix L oder l wird ein Long Integer erzeugt, bevorzugt großes L, um das kleine l nicht mit einer 1 (Eins) zu verwechseln
017	mit dem Präfix 0 (Null) wird eine Oktalzahl angenommen
0x170D oder 0x170d	mit dem Präfix 0x wird eine Hexadezimalzahl angenommen, als Buchstabenziffern sind A,B,C,D,E,F oder a,b,c,d,e,f zugelassen

Operatoren auf Integer: Auf dem Integer-Datentyp sind fast alle Operatoren definiert, mit Ausnahme der Teilbereichsoperatoren für Sequenztypen.

Speziell nur für den Datentyp Integer sind die Befehle zur Bitmanipulation, also speziell:

<code>~X</code>	Bitweises Invertieren von X
<code>X & Y</code> <code>X Y</code> <code>X ^ Y</code>	Bitweises logisches UND, ODER, XOR zwischen X und Y; siehe auch Datentyp Boolean.

$X \ll Y$	Bitweises Verschieben von X um Y Binärstellen nach links, zieht 0 nach, Vorzeichen bleibt erhalten!
$X \gg Y$	Bitweises Verschieben von X um Y Binärstellen nach rechts, zieht Vorzeichenbit nach, Vorzeichen bleibt erhalten!

Bitmanipulationen sind prinzipiell sehr diffizil und fehleranfällig. Sie werden genutzt, um mit minimalen Speicheraufwand viele Informationen in einem Speicherwort hochkomprimiert zu halten, z.B. in der Systemprogrammierung oder bei Statuswörtern externer Geräte (Drucker, etc.). Wenn es geht, sollte man diese „Bitfrimelei“ in der Programmierung vermeiden.

Eigentlich auch speziell für den Integer-Datentyp sind die Funktionen der Modulo-Arithmetik (Ganzzahligen Divisionen) obwohl diese in Python auch für die anderen numerischen Datentypen definiert sind.

$X // Y$	Liefert das Ganzzahlige Ergebnis einer Division (ohne Rest)
$X \% Y$	Liefert den Rest einer Ganzzahligen Division: X modulo Y

Beispiel:

$13 // 3$ liefert den Wert 4

$13 \% 3$ liefert den Wert 1

Diese Funktionen haben zum Beispiel in der Kryptographie eine besondere Bedeutung. Für Integer sind wenige spezielle Funktionen definiert, für alle numerischen Datentypen gibt es:

<code>abs(x)</code>	Absolutwert von x (Positiven Integerwert)
<code>divmod(X, Y)</code>	Ergebnis ist ein Tupel $(X//Y, X\%Y)$
<code>pow(X, Y[, Z])</code>	X zur Potenz Y [modulo Z]

3.3 Gleitpunktzahlen (floating point number)

Neben ganzen Zahlen, müssen wir natürlich auch reelle Zahlen repräsentieren können. Mit einer festgelegten Anzahl von Bits (meist 32, 64, seltener 128 oder gar 256 Bits) geht dies für die meisten Zahlen nur approximativ. Die Menge der *Gleitkommazahlen* ist sogar nur eine endliche Teilmenge der rationalen Zahlen, meist erweitert um einige Spezialelemente ($+\infty$, $-\infty$, NaN (= "Not A Number"), $+0$, -0 , usw., siehe unten).

Im Englischen wird statt des Dezimalkommata der Dezimalpunkt genutzt. Für das Programmieren ist es daher wichtig, sich diese Schreibweise anzugewöhnen, also anstelle 3,14 dann 3.14 zu schreiben. Wir nutzen im Weiteren die Punktschreibweise für Dezimalzahlen.

Die Repräsentation einer Gleitpunktzahl ist sehr ähnlich zu der in Mathematik und Naturwissenschaft üblichen Exponentialschreibweise. Betrachten wir beispielsweise die Zahl

$$x = -1245.75.$$

Sie lässt sich auch schreiben als

$$x = -1.24575 \cdot 10^3$$

In dieser Schreibweise besteht diese Zahlen also aus einem Vorzeichen, üblicherweise einer einzigen Ziffer $1 \leq m \leq 9$ vor dem Komma (normalisierte Darstellung) (oder 0 nur bei dem Wert 0) und mehreren Dezimalstellen hinter dem Komma und einer Zehnerpotenz,

d.h. wir benutzen die Basis 10. Diese Notation können wir leicht auf die Dualzahlen übertragen. Eine Zahl x schreiben wir dann als:

$$x = (-1)^s \cdot m \cdot 2^e$$

- s ist das Vorzeichenbit (1 für Minus, 0 für Plus),
- m als sogenannte Mantisse² (normalisiert dann im halboffenen Intervall $[1,2)$ und
- e als Exponent zur Basis 2

Die Normalisierung kann durch Erhöhung oder Erniedrigung des (ganzzahligen) Exponenten e immer erreicht werden. Da eine normalisierte Mantisse immer mit 1 beginnt, genügt es, nur die Nachpunktstellen zu speichern, also die reduzierte Mantisse $m' = m - 1$.

Das IEEE 754- und IEEE 754-2008-Format

Das gebräuchliche und häufig auch durch Hardware unterstützte Format ist in der Norm **IEEE 754** (ANSI/IEEE Std 754-1985; IEC-60559 - International version) von 1987 festgelegt. Eine Neufassung **IEEE 754-2008** wurde im Juni 2008 verabschiedet. Die Unterschiede sind gering: Reduktion von Implementierungsalternativen und Mehrdeutigkeiten aus IEEE 754, halbe und vierfache Genauigkeit (also Formate für 16, 32, 64 und 128 Bit) und die von der Finanzwirtschaft als notwendig erachteten Dezimalformate (früher in IEEE 854 genormt). Die Konzepte sind identisch. Viele Implementierungen beziehen sich heute noch auf IEEE 754 (1987).

Beide Normen legen die Standarddarstellungen für **binäre Gleitkommazahlen** fest und definieren Verfahren für die Durchführung mathematischer Operationen, insbesondere für Rundungen. Beinahe alle modernen Prozessoren folgen diesem Standard. Ausnahmen sind u.a. die **Java Virtual Machine** mit den Java Typen *float* und *double*, die nur einen Subset der IEEE 754 Funktionalität unterstützt.³

In der **IEEE 754-2008** sind vier Grunddatenformate für binäre Gleitkommazahlen mit 16, 32, 64 und 128 Bit definiert. (b32 und b64 waren auch schon in IEEE 754).

Tabelle 3: Kennzahlen der IEEE 754r Gleitpunktdarstellungen

Typ	Größe	Anzahl der Mantissenbits für M	Anzahl der Exponenten-bits für E	e_{\min}	e_{\max}	Bias B
b16	16 Bit	10 Bit	5 Bit	-14	15	15
b32 (single)	32 Bit	23 Bit	8 Bit	-126	127	127
b64 (double)	64 Bit	52 Bit	11 Bit	-1022	1023	1023
b128	128 Bit	112 Bit	15 Bit	-16382	16383	16383

² Der aus dem lateinischen kommende Begriff **Mantisse** ist die Bezeichnung für Nachkommastellen. Bekannt ist der Begriff vor allem durch das Logarithmieren. In der Informatik sind die **Mantissen** für die Darstellung von Gleitkommazahlen von herausragender Bedeutung.

³ Hierzu gab es hitzige Fachdiskussionen, vergl. z.B. „How Java’s Floating-Point Hurts Everyone Everywhere“ by W. Kahan and D. Darcy, 1998, siehe <http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf> und auch Lösungsansätze, siehe <http://www.sonic.net/~jddarcy/Borneo/>. Eines der Probleme dreht sich um NaN. Es gibt eine Vielzahl von Ursachen, warum ein Rechenergebnis NaN ist und entsprechend in IEEE 754 eine Vielzahl verschiedener NaN-Darstellungen, die in Java einfach auf einen einzigen NaN-Wert abgebildet werden.

Bei normalisierten Gleitpunktzahlen nach IEEE 754r ist die Basis $b = 2$:

Das **Vorzeichen** $s = -1^s$ wird in einem Bit s gespeichert, so dass $s = 0$ positive Zahlen und $s = 1$ negative Zahlen markiert.

Der **Exponent** e ergibt sich aus der in den Exponentenbits gespeicherten nichtnegativen Binärzahl E durch Subtraktion eines festen **Biaswertes** B : $e = E - B$. Achtung: Der Exponent ist also **nicht** im Zweier-Komplement gespeichert!

Schließlich ist die **Mantisse** $1 \leq m < 2$ ein Wert, der sich aus den p Mantissenbits M berechnet als $m = 1 + M/2^p$. Einfacher ausgedrückt, denkt man sich an das Mantissenbitmuster M links eine 1. angehängt: $m = 1.M$.

Dieses sogenannte hidden-Bit-Verfahren ist möglich, weil durch Normalisierung die Bedingung $1 \leq m < 2$ für alle (darstellbaren) Zahlen immer eingehalten wird. Da also die Mantisse immer links mit 1. beginnt, braucht dieses Bit nicht mehr gespeichert zu werden. Damit gewinnt man ein zusätzliches Bit Genauigkeit (Präzision).

Der Wert einer normalisierten Gleitpunktzahl bestimmt sich also zu:

$$W = \pm 1.m \cdot 2^e = \pm 1.m \cdot 2^{E-B}$$

Die Anordnung der Bits einer Gleitpunktzahl in 32 bit (*float*) zeigt die nachfolgende Abbildung.⁴

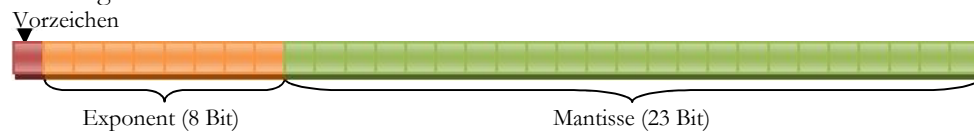


Abbildung 2 Typische Repräsentation einer Single-precision Gleitpunktzahl

Auch wenn wir hier hauptsächlich die Zahlenformate erörtern, liegt die Bedeutung der IEEE 754-Normen insbesondere darin, dass für Gleitpunktzahlen genaue Vorschriften für Rundung, arithmetische Operationen, Wurzelberechnung, Konversionen und Ausnahmebehandlung (engl. *exception handling*) festgelegt werden.

Sind im Exponenten einer Zahl alle Bits gesetzt ($= 1$) oder alle gelöscht ($= 0$), so hat diese Darstellung eine besondere Interpretation gemäß folgender Tabelle:

Tabelle 4: Bedeutung besonderer Codewörter in IEEE 754

Exponent	Mantisse	Bedeutung
111...111binär	000...000binär	+/- Unendlich, je nach Vorzeichen
111...111binär	\neq 000...000binär	"Keine Zahl" (NaN = Not a Number)
000...000binär	000...000binär	+/- 0. (Null)
000...000binär	\neq 000...000binär	Denormalisierte Gleitpunktzahl als „subnormal“ nach IEEE 754r bezeichnet

"+/-Unendlich": Repräsentiert Zahlen, deren Betrag zu groß ist, um überhaupt dargestellt zu werden. Es wird zwischen +"Unendlich" und -"Unendlich" unterschieden. Die Berechnung von $+1.0/0.0$ ergibt *per Definition* ebenfalls +"Unendlich".

"Keine Zahl" (NaN): Damit werden ungültige (oder nicht definierte) Ergebnisse dargestellt, z. B. wenn versucht wurde, die Quadratwurzel aus einer negativen Zahl zu berechnen. Einige "unbestimmte Ausdrücke" haben als Ergebnis "keine Zahl", zum

⁴ In einer Implementierung darf die konkrete Anordnung der Bits im Speicher von der im Bild dargestellten abweichen und hängt insbesondere von der jeweiligen Bytereihenfolge (little/big endian) und ggf. weiteren Rechnereigenheiten ab.

Beispiel 0.0/0.0 oder "Unendlich". Außerdem werden NaNs in verschiedenen Anwendungsbereichen benutzt, um "Kein Wert" oder "Unbekannter Wert" darzustellen. **+/- Null:** repräsentiert die absolute Null. Auch Zahlen, deren Betrag zu klein ist, um dargestellt zu werden (Unterlauf) werden auf null gerundet. Ihr Vorzeichen bleibt dabei erhalten. Negative kleine Zahlen werden so zu -0.0 gerundet, positive Zahlen zu +0.0. Beim direkten Vergleich werden jedoch +0.0 und -0.0 als gleich angesehen. (Dieses ist übrigens der Grund dafür, warum auf Gleitpunktzahlen streng genommen keine Ordnung definiert ist, sie sind nicht ordinal)

Denormalisierte Zahl: Ist eine Zahl zu klein, um in normalisierter Form mit dem kleinsten, von Null verschiedenen Exponenten gespeichert zu werden, so werden sie als "Denormalisierte Zahl" gespeichert. Ihre Interpretation ist nicht mehr

$$W = \pm 1.m \cdot 2^e = \pm 1.m \cdot 2^{E-B}$$

sondern

$$W = \pm 0.m \cdot 2^{e_m} .$$

e_m ist dabei der Wert des kleinsten "normalen" Exponenten (siehe Tabelle 3). Damit lässt sich die Lücke zwischen der kleinsten normalisierten Zahl und Null ausfüllen. **Denormalisierte Zahlen haben jedoch eine geringere Genauigkeit (Präzision) als normalisierte Zahlen**, die Anzahl der signifikanten Stellen in der Mantisse nimmt zur Null hin ab.

Ist der Betrag des Ergebnisses (oder des Zwischenergebnisses) einer Rechnung kleiner als die kleinste darstellbare Zahl in normalisierter Form, so wird das Ergebnis im Allgemeinen auf Null gerundet; das nennt man **Unterlauf (Underflow)**. Da dabei Information verloren geht, versucht man, Unterlauf nach Möglichkeit zu vermeiden. Die denormalisierten Zahlen in IEEE 754 bewirken einen **allmählichen Unterlauf**, indem "um die 0 herum" 2^{24} (für 'single') bzw. 2^{53} (für 'double') Werte eingefügt werden, die alle denselben absoluten Abstand voneinander haben und sonst zu 0 gerundet werden müssten.

Umwandlungen: Dezimalzahl Gleitpunktzahl

Für die Umwandlung einer Dezimalzahl in die IEEE 754-Maschinendarstellung geht man folgendermaßen vor. Als Beispiel soll -166,125 soll in eine float-Repräsentation (single precision) umgewandelt werden.

Schritt 1: Man wandle die Zahl Vorkomma- und Nachkommastellen getrennt in Dualzahlen

Beispiel: $-166.125_{(10)} = -10100110.001_{(2)}$

Schritt 2: Normalisieren (Verschieben des „Punkts“) also in eine Darstellung 1.xxx.... Die Anzahl der Verschiebungen ergeben e.

Die Mantisse M erhält wir, indem wir die Vorkomma 1 und das Komma weglassen. Um auf 23 Bit zu kommen wird (falls nötig) hinten mit Nullen aufgefüllt:

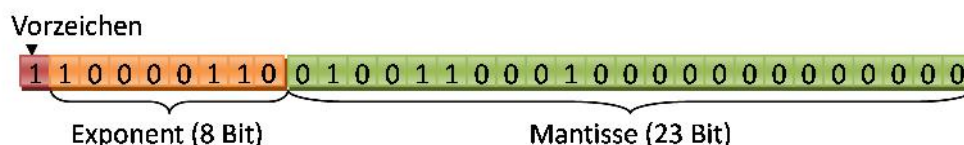
Beispiel: $-10100110.001_{(2)} = -1.0100110001_{(2)} \cdot 2^{7(1)}$

M = 0100110 00100000 00000000

Schritt 3: Bestimmung der Exponentenrepräsentation: $E = e + \text{Bias}$

Beispiel: $7_1 + 127_1 = 134_1 = 10000110_2$ (127 ist der Bias-Wert, da Darstellungsgenauigkeit vom Typ "Single")

-166,125 wird als Single-Gleitpunktzahl also folgendermaßen repräsentiert:



Die Umwandlung einer Zahl aus der normalisierten Gleitpunktdarstellung in eine Dezimalschreibweise verläuft entsprechend umgekehrt. Wir wissen, der Wert einer Gleitpunktdarstellung ist

$$W = \pm 1 \cdot m \cdot 2^e = \pm 1 \cdot m \cdot 2^{E-B}$$

Beispiel: Versuchen wir also folgendes Bitmuster in Dezimalschreibweise zu überführen, von dem wir wissen, dass dieses eine single-precision-(32 Bit)-Kodierung ist:

0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 1 0 1 0 1 1 0 0

Da das erste Bit auf 0 gesetzt ist, sehen wir sofort dass es sich um eine positive Zahl handelt. Den mit einem Bias behaftete Exponenten e dekodieren wir als

$$e = 10000110_2 = 134_1 - 127_1 = 7_1$$

Für die Mantisse m ergibt sich schließlich

$$m = 1000100011110010101100_2 = 0.5350547_1 + 1_1 = -1.5350547_1 .$$

Insgesamt erhalten wir also

$$\begin{aligned} x &= (-1)^s m \cdot 2^e \\ &= (-1)^s \cdot 1.5350547 \cdot 2^7 \\ &= 1 \cdot 1.5350547 \cdot 128 \\ &= 196.4870016_1 \end{aligned}$$

Eigenschaften von Gleitpunktzahlen

Die Anzahl der Exponentenbits bestimmt den Wertebereich der darstellbaren Zahlen (größter und kleinster Absolutwert einer Zahl). Die Anzahl der Mantissenbits legt die Genauigkeit (→ „Anzahl der korrekten Dezimalstellen“) dieser Zahlen fest.

Mit Gleitpunktzahlen lassen sich trotz begrenzter Länge (Bitstellen) relativ große **Wertebereiche** abdecken, nämlich wie in Tabelle 5 angegeben von $[-MAX, MAX]$. Dabei liegt gleichzeitig die betragsmäßig kleinste Zahl relativ nahe bei Null.

Tabelle 5: Wertebereiche der IEEE 754-2008 Gleitpunktzahlen

Typ Bitzahl	betragsmäßig kleinste Zahl $\neq 0$	Größte Zahl (MAX)
b16	$2^{-10} \times 2^{-14} = 2^{-24} \approx 5960 \cdot 10^{-8}$	$(2-2^{-11}) \times 2^{15} \approx 65520$
b32 (single)	$2^{-23} \times 2^{-126} = 2^{-149} \approx 1,401 \cdot 10^{-45}$	$(2-2^{-23}) \times 2^{127} \approx 3,403 \cdot 10^{38}$
b64 (double)	$2^{-52} \times 2^{-1022} = 2^{-1074} \approx 4,941 \cdot 10^{-324}$	$(2-2^{-52}) \times 2^{1023} \approx 1,798 \cdot 10^{308}$
b128	$2^{-112} \times 2^{-16382} = 2^{-16494} \approx 10^{-4965}$	$(2-2^{-112}) \times 2^{16383} \approx 10^{4239}$

Deutlich werden diese Eigenschaften, wenn man Sie mit Ganzzahlen im Zweierkomplement vergleicht: Sowohl die betragsmäßig kleinste Zahl $\neq 0$ ist bei einer Gleitpunktzahl viel kleiner als 1 (Integer).

Tabelle 6: Vergleich der Wertebereiche der IEEE 754-2008 Gleitpunktzahlen und einer Ganzzahldarstellung im Zweierkomplement

Bitzahl	Gleitpunktzahl		Ganzzahl Zweierkomplement	
	betragsmäßig kleinste Zahl $\neq 0$	Größte Zahl (MAX)	betragsmäßig kleinste Zahl $\neq 0$	Größte Zahl (MAX)

b16	$\approx 5960 \cdot 10^{-8}$	≈ 65520	1	32767
b32 (float)	$\approx 1,401 \cdot 10^{-45}$	$\approx 3,403 \cdot 10^{38}$	1	$\approx 2,147 \cdot 10^7$
b64 (double)	$\approx 4,941 \cdot 10^{-324}$	$\approx 1,798 \cdot 10^{308}$	1	$\approx 9,223 \cdot 10^{18}$
b128	$\approx 10^{-4965}$	$\approx 10^{4239}$	1	$\approx 1,701 \cdot 10^{38}$

Betrachten wir jetzt die Genauigkeit einer Gleitpunkt-Repräsentation: Für jeden Wert des Exponenten $e \neq 0$ können 2^p verschiedene Gleitpunktzahlen repräsentiert werden. Bei Gleitpunktzahlen ist also nicht die absolute Anzahl von Dezimalstellen konstant, sondern die Anzahl **wesentlicher oder signifikanter** Stellen, die durch die Präzision (Anzahl der Bits zur Darstellung der Mantisse) bestimmt wird.

Also zum Beispiel für eine float-Gleitpunktzahl mit 23 Bit Mantisse: $2^2 = 8388608$, also mehr als 8 Millionen verschiedene Zahlen für jeden gültigen Exponent: Tabelle 7 zeigt dies für verschiedene Intervalle.

Tabelle 7 Absolute Genauigkeit einer IEEE 754r b32 Gleitpunktzahl (float)

e	Intervall für Z	Größen- ordnung	Delta zwischen der größten und zweitgrößten Zahl Z = absolute Genauigkeit
1	$1 \leq Z < 2$	1	0,000000238
2	$2 \leq Z < 4$		0,000000477
10	$512 \leq Z < 1024$	10^3 (Kilo)	0,000122
20	$524,288 \leq Z < 1,048,576$	10^6 (Mega)	0,125
30	$0.537 \cdot 10^9 \leq Z < 1.074 \cdot 10^9$	10^9 (Giga)	128
50	$0.563 \cdot 10^{15} \leq Z < 1.126 \cdot 10^{15}$	10^{15} (Peta)	134.217.728

Die absolute Genauigkeit einer Gleitpunktzahl hängt direkt von der Größe der Zahl ab: Im Bereich um 1 Million für Z beträgt sie Größenordnungsmäßig 0,1. Die **relative Genauigkeit** ist über den gesamten Wertebereich **konstant bei etwa 6-7 Dezimalstellen** und hängt natürlich direkt von der Länge der Mantisse ab. Für die anderen Formate des IEEE 754r können wir die Anzahl der gültigen Dezimalstellen auch leicht berechnen, indem wir die Länge des Exponenten in Bit (p) betrachten. Diese entspricht der Anzahl der Binärstellen und lässt sich leicht in Dezimalstellen umwandeln.

$$2^p = 10^x \Rightarrow x = p \cdot \lg(2) = p \cdot 0.3010$$

Tabelle 8: Relative Genauigkeit der IEEE Gleitpunktzahlen

Typ Bitzahl	p	Relative Genauigkeit (Präzision, precision) in Anzahl der Dezimalstellen
b16	10 Bit	≈ 3
b32 (single)	23 Bit	≈ 6
b64 (double)	52 Bit	≈ 15
b128	112 Bit	≈ 33

Wichtig ist also, dass alle Gleitpunktzahlen nur eine begrenzte Genauigkeit bieten und es zwangsweise zu Rundungsfehlern kommen muss. Dies kommt zum Beispiel dann zum Tragen, wenn viele einzelne Fließpunktzahlen aufsummiert werden. Oft addieren sich hier die Rundungsfehler. Zum Beispiel kann die Zahl $0.1_{(10)}$ in keiner Gleitpunktdarstellung exakt dargestellt werden, immer nur als Approximation: $0.000110011001100110011..._{(2)}$, also eine Zahl mit periodischen, und somit unendlich langen Nachpunktstellen.

Für die meisten aller technisch-wissenschaftlichen Berechnungen reicht die Genauigkeit der Gleitpunktzahlen trotzdem aus. Genau deswegen aber sollten jedem Programmierer spezifische Eigenarten der Gleitpunktkodierung bewusst sein. Durch die binäre Darstellung der Zahlen kommt es zu „Gleitpunkt-Artefakten“. Das bedeutet, dass Zahlen, die zwar im Dezimalsystem präzise darstellbar sind, mit Gleitkomazahlen aber nicht exakt abgebildet werden können.

Das Programmieren mit Gleitpunktzahlen erfordert stets Aufmerksamkeit und Umsicht. Als Grundregeln seien genannt:

- Vorsicht bei der Addition und Subtraktion betragsmäßig kleiner und großer Zahlen (Sie verlieren sehr schnell die theoretische Genauigkeit einer einzelnen Zahl!)
- Führen Sie nur sichere Vergleiche durch (**nie** auf Null testen)!
- Erwarten Sie nicht, dass sich verschiedene Implementierungen von Gleitpunkt-Arithmetik genau gleich verhalten.

Gleitpunktzahlen in Python

In Python wird nur ein Floatingpoint-Format unterstützt und zwar **b64**, also double (`<type 'float'>`)

Literale: Gleitkommazahlen (Float) können im Programm auf folgende Art notiert werden:

<code>123.4, -42.0, 0., .1</code>	einfache Float: die Kennzeichnung erfolgt durch den Dezimalpunkt)
<code>3.11e-10 oder -4E11.1</code>	(ein kleines e oder ein großes E kennzeichnet eine Zahl in Exponentialschreibweise: $aEb = a \cdot 10^b$ für a,b sind einfache Float oder Integerschreibweisen im Dezimalsystem zugelassen) Die Zahlen vor und nach dem e/E werden immer als Dezimalzahlen interpretiert. (Oktalzahlen oder Hexadezimalzahlen sind nicht zugelassen!)

Operatoren auf Float: Auf dem Float-Datentyp sind viele Operatoren definiert. Interessant ist, dass für Float auch die Modulo-Operation existiert, allerdings mit einer anderen Bedeutung

<code>X // Y</code>	Liefert das „Ganzzahlige“ Ergebnis einer Division, Ergebnis der Form $\cdot 0$. Das Ergebnis ist von Typ float, wenn X oder Y vom Typ float sind.
---------------------	--

<code>X % Y</code>	Liefert den Rest einer „Division“ bei ganzzahligem Ergebnis in Float repräsentiert.
--------------------	---

Beispiel:

<code>13.2 // 3.2</code>	liefert den Wert 4.0
<code>13.0 % 3.2</code>	liefert den Wert 0.2

Funktionen für Float: Python stellt insbesondere die folgenden Funktionen für Float zur Verfügung (auch wieder für alle numerischen Datentypen), hier nur zur Vollständigkeit.

<code>divmod(X, Y)</code>	Ergebnis ist ein Tupel $(X//Y, X\%Y)$
<code>pow(X, Y[, Z])</code>	X zur Potenz Y [modulo Z]

round (X [, N])

Liefert ein float, gerundet auf N Dezimal-Ziffern nach dem Komma. Default N = 0.

3.4 Zusammenfassung und Ergänzungen zu numerischen Datentypen

Aus der Mathematik kennen wir die Teilmengenbeziehung zwischen verschiedenen Zahlenmengen: Für die *Natürlichen*-, *Ganzen*-, *Rationalen*-, *Reellen*- und *Komplexen Zahlen* gilt:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

In Programmiersprachen sind diese Mengen repräsentierbar:

$\mathbb{N} \subset \mathbb{Z}$ als Integer (Ganzzahlen)

$\mathbb{Q} \subset \mathbb{R}$ als Float (Gleitpunktzahlen)

Alle Datentypen haben ihre charakteristischen Eigenarten. Die verschiedenen Programmiersprachen unterscheiden sich dadurch, wie hardwarenah diese Numerischen Datentypen angeboten werden – oder man kann auch sagen, wie sehr in Richtung der üblichen mathematischen Schreibweisen und Zusammenhänge von der Hardware (und den Computerinterna) abstrahiert wird.

Mit den beiden numerischen Datentypen Integer und Float, kann man die Anforderungen der allermeisten Anwendungen abdecken. Für Anwendungen im kaufmännischen Bereich wird oft noch eine Dezimalgleitkommaarithmetik angeboten, da insbesondere bei größeren Beträgen die Gleitpunktfehler nicht hingenommen werden können. Es ist ja „undenkbar“, dass ab 10.000.000 € nicht mehr auf den Cent genau gerechnet werden kann, oder? (Beachte, auch mit sehr vielen kleinen „Rundungsfehlern“ kann man ggf. viel Geld „verdienen“ oder „verlieren“!)

Bis heute muss eine Dezimalarithmetik (z.B. Python `Decimal`) in der Regel in Software realisiert werden, was sie um Größenordnungen langsamer macht als die Floating-Point-Arithmetik, die heute fast immer durch Hardware unterstützt wird. Auf der anderen Seite, sind für sehr große Datenmengen, z.B. im wissenschaftlichen Rechnen, die speicheraufwendigen Repräsentationen in Python nicht optimal.

Für diese Anwendungsfälle bieten alle Programmiersprachen Erweiterungsmöglichkeiten an, die dann diese spezifischen Datentypen (Datenstrukturen und Operationen oder Funktionen) anbieten, aber verschieden in die Programmiersprachen eingebunden werden:

Am Beispiel Python:

Komplexe Zahlen \mathbb{C}	im Sprachkern	complex
Dezimalzahlen	im builtin-Modul	decimal
Weitere mathematische Funktionen	im builtin-Modul	math (float) und cmath (complex)
Zufallszahlen	im builtin-Modul	random
Wissenschaftliches (numerisches) Rechnen	Paket (Sammlung von Modulen)	von SciPy und NumPy

All diese und viele, viele andere Module und Pakete stehen der Programmiererin nicht direkt zur Verfügung, sondern müssen „importiert“ werden.

3.5 Boolescher Datentyp

Repräsentation

Das kleinste Speicherelement eines Computers ist eine Speicherstelle, ein „Bit“, deren Wertebereich durch zwei Zustände 0 und 1 gegeben ist. Diese zwei Werte können unterschiedlich interpretiert werden, beispielsweise als Zahlen {0,1}, oder als Wahrheitswerte {*F*, *T*}. Im Folgenden bezeichnen wir den Wert 0 als **False** und 1 als **True**. Oder auch in ihrer deutschen Übersetzung mit *Falsch* und *Wahr*.

Der zugehörige Datentyp wird als „Boolesch“ (nach George Boole), oder Englisch „Boolean“ bezeichnet.

Eine einzelne Variable vom Typ Boolean, kann in einem Rechner verschieden realisiert werden. Da die kleinste normal adressierbare Einheit ein Byte (8 Bit) oder gar ein Wort (z.B. 32 Bit) ist, muss man hier Vereinbarungen treffen:

In Python: Jede Variable kann boolesch „interpretiert“ werden und zwar:

False	für numerische Datentypen, wenn der Wert 0 vorliegt bei Strings die Länge 0 besitzt
True	für numerische Datentypen, wenn ein Wert $\neq 0$ vorliegt bei Strings eine Länge $\neq 0$ hat, also auch <code>'\x00'</code>

Der Typ Boolean ist in Python ein Subtyp von Integer.

Alternativ kann aber auch nur eine Stelle in einem Integer als Bit interpretieren. Das Bit ist dann durch seine Position (Stelle) in einem Integer identifizierbar. Ansprechbar wäre dies dann durch den Variablennamen und die Position.

Operationen auf Booleschen Daten

Auf Booleschen Daten gibt es überhaupt nur 16 verschiedene Operationen. Ein vollständiger Satz (d.h. aus diesen lassen sich alle anderen durch entsprechende Kombination erzeugen) sind die Negation (**not**), das UND (**and**) und das ODER (**or**).

Negation (logisches NICHT, not)

Die beiden Werte **True** und **False** gehen durch Verneinung („Negation“) auseinander hervor:

Die Negation wird, in Python, durch das Operator **not** (als Schlüsselwort) symbolisiert und negiert den Wert einer Booleschen Variable oder eines Booleschen Ausdrucks. Das folgende Codefragment legt eine Variable vom Typ **bool** an und weist ihr zunächst den Wert **False** zu. In der darauffolgenden Zeile wird der mit **not** negierte Wert einer weiteren Variablen zugewiesen.

```
black_equals_white = False
black_notequals_white = not black_equals_white
```

Das **not** bezieht sich nur auf einen einzigen Operanden und wird deshalb als *unärer Operator* bezeichnet.

Konjunktion (logisches UND, and)

Möchte man ausdrücken, dass von zwei Dingen beide Operatoren notwendig (**True**) sind, so verwendet man dazu eine „und“-Operation: Zum Schreiben benötigt man einen Stift UND ein Blatt Papier. Der Operator „UND“ (der sogenannten Konjunktion „**and**“) gehorcht der einfachen Regel, dass er als **True** ausgewertet wird, genau dann wenn beide Werte seiner Operanden **True** sind. In Python könnte das z.B. folgendermaßen aussehen:

```
isEmpty = False
hasMoreWork = True
continueWork = not isEmpty and hasMoreWork
```

In diesem konkreten Fall würde **continueWork** als **True** ausgewertet werden, da **not** früher ausgewertet wird („*stärker bindet*“) als **and**, siehe Tabelle 9. In der nachstehenden Tabelle („*Wahrheitstafel*“) sind alle Kombinationen der Argumente **a** und **b** für den **and**-Operator und sein jeweiliges Ergebnis aufgeführt.

Tabelle 9: Wahrheitstafel der Konjunktion

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Disjunktion (logisches ODER, or)

Möchte man dagegen ausdrücken, dass von zwei Dingen mindestens eins vorhanden sein muss, also zum Schreiben entweder eine Kugelschreiber ODER ein Filzstift oder beide, so verwendet man die „ODER“-Operation („Disjunktion“). Die Disjunktion wird also immer dann als **True** ausgewertet, wenn mindestens einer der beiden Ausdrücke **True** ist, oder beidem auf die er angewandt wird. In Python verwenden wir das Schlüsselwort **or** zur Darstellung.

Im nachfolgenden Codefragment würde **processExercise** den Wert **True** erhalten, solange mindestens eine der beiden Variablen **isInterestingExercise** und **needMoreScores** den Wert **True** besitzen.

```
isInterestingExercise = True
needMoreScores = False
processExercise = isInterestingExercise or needMoreScores
```

Die Wahrheitstafel der Disjunktion sieht folgendermaßen aus.

Tabelle 10: Wahrheitstafel der Disjunktion a or b und des Ausschließlichen Oders (XOR)

a	b	a or b	a xor b
False	False	False	False
False	True	True	True
True	False	True	True
True	True	True	False

Entsprechend der zweiten Interpretationsmöglichkeit (Bits als Teil eines Integer) sind für Integer auch entsprechende Bit-Operatoren definiert, die diese Verknüpfungen für jede einzelne Stelle im Integer-Wort durchführen:

X	Y	Oder
X & Y		Und
X ^ Y		Exclusives Oder, entweder oder, aber nicht beide
~ X		Nicht, invertieren

Aber bitte Aufpassen, beachten Sie die Unterschiede.

```

>>> X = 5
>>> Y = 2
>>> X or Y # Integer werden als Boolean aufgefasst
5
>>> X | Y
7

```

Im Sinn einer starken Typisierung ist dies unsauber. Der `or` Operator ist eigentlich nur für Boolean definiert, NICHT für Integer. Wir werden aber später sehen, dass dies zum Teil sehr elegante, kurze Schreibweisen zulässt, aber eben auch unübersichtlich sein kann. Kleine Unsauberkeiten, entweder mit Performance oder Bequemlichkeit begründet hat fast jede Programmiersprache, leider!

Bei „Dive into Python“ finden wir hierzu eine ausführliche Diskussion, die hier direkt wiedergegeben werden soll. Der folgende Teil in englischer Sprache ist von der Website: http://www.diveintopython.net/power_of_introspection/and_or.html.

“The Peculiar Nature of and and or

In Python, `and` and `or` perform boolean logic as you would expect, but they do not return boolean values; instead, they return one of the actual values they are comparing.

Example 4.15. Introducing and

```

>>> 'a' and 'b' ❶
'b'
>>> '' and 'b' ❷
''
>>> 'a' and 'b' and 'c' ❸
'c'

```

- ❶ When using `and`, values are evaluated in a boolean context from left to right. `0`, `''`, `[]`, `()`, `{}`, and `None` are false in a boolean context; everything else is true. Well, almost everything. By default, instances of classes are true in a boolean context, but you can define special [methods](#) in your class to make an instance evaluate to false. You'll learn all about classes and special methods in [Chapter 5](#). If all values are true in a boolean context, `and` returns the last [value](#). In this case, `and` evaluates `'a'`, which is true, then `'b'`, which is true, and returns `'b'`.
- ❷ If any value is false in a boolean context, `and` returns the first false value. In this case, `''` is the first false value.
- ❸ All values are true, so `and` returns the last value, `'c'`.

Example 4.1.6. Introducing or

```

>>> 'a' or 'b' ❶
'a'
>>> '' or 'b' ❷
'b'
>>> '' or [] or {} ❸
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx() ❹
'a'

```

- ❶ When using `or`, values are evaluated in a boolean context from left to right, just like `and`. If any value is true, `or` returns that value immediately. In this case, `'a'` is the first true value.
- ❷ `or` evaluates `' '`, which is false, then `'b'`, which is true, and returns `'b'`.
- ❸ If all values are false, `or` returns the last value. `or` evaluates `' '`, which is false, then `[]`, which is false, then `{}`, which is false, and returns `{}`.
- ❹ Note that `or` evaluates values only until it finds one that is true in a boolean context, and then it ignores the rest. This distinction is important if some values can have side effects. Here, the function `sidefx` is never called, because `or` evaluates `'a'`, which is true, and returns `'a'` immediately.


If you're a C hacker, you are certainly familiar with the `bool ? a : b` expression, which evaluates to `a` if `bool` is true, and `b` otherwise. Because of the way `and` and `or` work in Python, you can accomplish the same thing.

4.6.1. Using the and-or Trick

Example 4.17. Introducing the and-or Trick

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

- ❶ This syntax looks similar to the `bool ? a : b` expression in C. The entire expression is evaluated from left to right, so the `and` is evaluated first. `1 and 'first'` evaluates to `'first'`, then `'first' or 'second'` evaluates to `'first'`.
- ❷ `0 and 'first'` evaluates to `False`, and then `0 or 'second'` evaluates to `'second'`.

However, since this Python expression is [simply](#)  boolean logic, and not a special construct of the language, there is one extremely important difference between this `and-or` trick in Python and the `bool ? a : b` syntax in C. If the value of `a` is false, the expression will not work as you would expect it to. (Can you tell I was bitten by this? More than once?)

Example 4.18. When the and-or Trick Fails

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b ❶
'second'
```

- ❶ Since `a` is an empty string, which Python considers false in a boolean context, `1 and ''` evaluates to `' '`, and then `' ' or 'second'` evaluates to `'second'`. Oops! That's not what you wanted.

The `and-or` trick, `bool and a or b`, will not work like the C expression `bool ? a : b` when `a` is false in a boolean context.

The real trick behind the `and-or` trick, then, is to make sure that the value of `a` is never false. One common way of doing this is to turn `a` into `[a]` and `b` into `[b]`, then taking the first element of the returned list, which will be either `a` or `b`.

Example 4.19. Using the and-or Trick Safely

```
>>> a = ""
```

```
>>> b = "second"
>>> (1 and [a] or [b])[0] ❶
''
```

❶ Since `[a]` is a non-empty list, it is never false. Even if `a` is `0` or `''` or some other false value, the list `[a]` is true because it has one element.

By now, this trick may seem like more trouble than it's worth. You could, after all, accomplish the same thing with an `if` statement, so why go through all this fuss? Well, in many [cases](#), you are choosing between two constant values, so you can use the simpler syntax and not worry, because you know that the `a` value will always be true. And even if you need to use the more complicated safe form, there are good reasons to do so. For example, there are some cases in Python where `if` statements are not allowed, such as in `lambda` functions.”

Das ist tatsächlich schon ein recht merkwürdiges Verhalten, das allerdings „wohlüberlegt“ entschieden wurde. Über das “WARUM ist das so?” findet man in den [PEPs](#) (Python Enhancement Proposals) Aufklärung: Zu diesem Fall in der PEP 0285: <https://www.python.org/dev/peps/pep-0285/>.

Vergleichsoperatoren in Python

Vergleichsoperatoren liefern als Ergebnis den Datentyp Boolean, also diese Operatoren liefern **True** oder **False**. Sie sind auf allen eingebauten Typen definiert. Für numerische Datentypen ist die Semantik dieselbe wie in der Mathematik üblich. Tabelle 11 zeigt eine Übersicht.

Tabelle 11 Vergleichsoperatoren

Operator	Beschreibung
<code>X < Y</code>	echt kleiner als
<code>X <= Y</code>	kleiner oder gleich
<code>X > Y</code>	echt größer als
<code>X >= Y</code>	größer oder gleich
<code>X == Y</code>	gleicher Wert
<code>X != Y</code>	Ungleicher Wert (
<code>(X <> Y)</code>	Ungleicher Wert (Schreibweise nicht empfohlen)
<code>X is Y</code>	Gleiches Objekt (Variable)
<code>X is not Y</code>	Negierte Objektgleichheit

Beachten Sie den Unterschied zwischen

`X == Y`

und

`X is Y`

Im ersten Fall wird verglichen ob `X` und `Y` den gleichen Wert haben, im zweiten Fall, ob sie dasselbe Objekt sind (also, die Identität/der Name gleich ist).

3.6 Datentypen für Zeichen und Zeichenketten (Text)

Als **Text** bezeichnen wir im engeren Sinne geschriebene Sprache. **Sprache** ist die wichtigste Kommunikationsform des Menschen. Sie wird originär akustisch durch

Schallwellen (Lautketten) oder visuell-räumlich durch Gebärden (vgl. Gebärdensprache) oder eben durch **Schrift** repräsentiert.

Text benötigt zu seiner Darstellung eine Schrift, deren Zeichen Phoneme, Silben oder Wörter bzw. Begriffe kodieren. Im westlichen Kulturkreis werden üblicherweise auf dem lateinischen Alphabet basierende Alphabete zur Darstellung von Text verwendet, in Osteuropa oft das kyrillische Alphabet, im Vorderen Orient das arabische. Ostasiatische Sprachen verwenden komplexe Schriftzeichen, um Text darzustellen.

Die Einführung der Schrift ist zweifellos eine der größten Kulturleistungen des Menschen. Durch die Einführung der „geschriebenen Sprache“ wurde eine Möglichkeit geschaffen, Texte, wie zum Beispiel Geschichtsschreibung, Erzählungen, aber auch Rezepte und Verfahren (durch Programmiersprachen Algorithmen) für die Nachwelt zu archivieren. Ein großer Teil unseres Wissens über vergangene Epochen stammt aus schriftlichen Aufzeichnungen, die archiviert wurden oder zufällig erhalten blieben.

Die ägyptische Hochkultur entwickelte schon ca. 2000 v. Chr. eine Schrift, die so genannten Hieroglyphen, die zunächst als Bilderschrift entstand und sich zu einer Silbenschrift entwickelte. Alle weit verbreiteten heutigen Alphabetschriften stammen vermutlich von der gleichen phönizischen Schriftform ab, oder sind zumindest durch Schriften phönizischer Abstammung inspiriert geschaffen worden. Die einzigen größeren noch bestehenden Schrifttraditionen nichtphönizischer Herkunft sind die chinesische und die koreanische.

Man unterscheidet allgemein:

- Alphabetschrift – die grundlegende Beziehung hier ist: ein Zeichen entspricht einem Laut (bzw. Phonem)
- Silbenschrift – hier entspricht weitgehend ein Zeichen einer Silbe
- Logogramme – ein Zeichen steht hier in der Regel für ein Wort bzw. für eine Aussage oder Anweisung

Alphabet- und Silbenschriften sind mehr oder weniger phonologisch, also lautbasiert. Logogramm- oder Ideogramm-Schriften sind hingegen eher bedeutungsbasiert: das Schriftzeichen entspricht eher einer bestimmten Bedeutung als einem bestimmten Laut. Beispiele für eine solche Schrift sind die arabischen Ziffern (1,2,3 ...), mathematische Symbole, Verkehrszeichen, Piktogramme, Gefahrensymbole. Diese sind oft international, also auch über Sprachgrenzen hinweg, verständlich.

Unter einem **Zeichensatz** versteht man einen Vorrat an Elementen zur Darstellung von Sachverhalten.

Unter anderem sind dies die Buchstaben eines Alphabetes, sowie Ziffern. Es können aber auch andere grafische Symbole sein, z. B. die Zeichen einer Lautschrift (z.B. des IPA-Codes), die Zeichen der Brailleschrift oder Bildzeichen (Ikonen engl. *Icons*).

Wir fokussieren uns im Folgenden auf Alphabetschriften. Ein **Alphabet** ist eine Menge von Zeichen (ein Zeichensatz) zur Abbildung von Lauten einer Sprache. Der Name **Alphabet** geht auf die ersten beiden Buchstaben des griechischen Schriftsystems zurück (Alpha – α , Beta – β). Analog dazu sagt man im Deutschen A-B-C. Die festgelegte Reihenfolge erlaubt *alphabetische* (oder lexikographische) Anordnungen, z.B. in Wörterbüchern.

In der Informatik und Mathematik wird der Begriff Alphabet oft etwas weiter gefasst: Unter **Alphabet** versteht man (z.B. nach DIN 44300) eine total geordnete endliche Menge von unterscheidbaren Symbolen (Zeichen). Für Alphabete wird häufig Σ (Sigma) als Formelzeichen verwendet.

Ein Alphabet kann ein „normales“ Alphabet sein, beispielsweise das Alphabet der lateinischen Buchstaben. In der Informatik kommen jedoch häufig auch Alphabete vor, deren Zeichen bereits aus mehreren Buchstaben bestehen, beispielsweise: $\Sigma = \{\text{oma, mutter, tochter}\}$: Welches Zeichen für die Elemente des Alphabets verwendet werden ist belanglos, **solange sie voneinander unterscheidbar sind.**

Unter Kleenescher Hülle Σ^* des Alphabets versteht man die Menge aller Wörter (=Zeichenreihen, also endliche lineare Folgen von Zeichen eines Alphabets) über dem Alphabet Σ . Auch die Zeichenreihe, die keine Zeichen enthält, ist ein Wort - das leere Wort. Es wird mit ϵ bezeichnet. Alphabete bilden somit die Grundlage für formale Sprachen, für die sie das **Zeicheninventar für Wörter** zur Verfügung stellen.

Beispiele: Dezimalzahlensystem: Es ist $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ und alle Wörter über Σ können als Dezimalzahlen aufgefasst werden (zumindest wenn führende Nullen erlaubt sein sollen). Elemente von Σ nennen wir dann Ziffern, ein Wort ist eine Zahl.

Römisches Zahlensystem: $\Sigma = \{I, V, X, L, C, D, M\}$, wobei hier die Regeln, wie die Zeichenfolge beschaffen sein muss um als "Wort" des römischen Zahlensystems zu gelten komplexer sind (IV anstatt IIII, größere Einheiten weiter links als kleinere, etc.).

Spezifischer versteht man im Kontext von Programmiersprachen unter einem **Zeichensatz** die Zuordnung von alphanumerischen Zeichen (Buchstaben und Ziffern) sowie Sonderzeichen zu einem Zahlencode. Traditionell ist der US-amerikanische **ASCII**-Code besonders weit verbreitet. Man hat versucht, ein Zeichen mit 7 oder 8 Bit (also in einem Byte) zu kodieren. Dies führte dazu, dass nahezu unzählige **nationale Varianten** des weithin akzeptierten ASCII-Codes entstanden um die benötigten Sonderzeichen (im Deutschen öÖüÜäÄß) zu repräsentieren (siehe unten).

Zunehmend wichtiger werden deshalb Zeichensätze, die diese vielen Code-Varianten überflüssig machen, dadurch, dass sie es prinzipiell zulassen, z.B. 16 Bit für ein Zeichen zu benutzen, insbesondere der international anerkannte Standard **Unicode**.

Der Begriff Zeichensatz **muss dabei streng unterschieden** werden von den Möglichkeiten der typografischer Gestaltung einer Schrift durch unterschiedlicher Schriftarten, Schriftgrößen und Auszeichnungsarten, die Wahl der Satzbreite (Zeilenlänge), des Zeilenfalls, des Satzspiegels innerhalb des Darstellungsraumes u. v. m. Ein Zeichensatz enthält „abstrakte Zeichen“, nicht deren graphische Ausprägung.

Als **Schriftart** (*font*, *typeface*) bezeichnet man **die grafische Gestaltung eines Zeichensatzes**, also eine Menge von Glyphen. Es gibt unterschiedliche Schriftarten um gestalterische Unterschiede zu ermöglichen, aber auch um die bestmögliche Lesbarkeit auf verschiedenen Medien wie Papier oder Monitoren zu erreichen. Wichtige Unterscheidungsmerkmale sind:

Die **Breite der Zeichen**: Im Normalfall sind die einzelnen Zeichen einer Schriftart unterschiedlich breit, ein „w“ nimmt also mehr Platz ein als ein „i“. Solche Schriftarten werden **proportional** genannt. Weit verbreitete Mitglieder dieser Gruppe sind Times oder Arial. Um die Konstruktion der ersten Schreibmaschinen nicht unnötig zu verkomplizieren, kamen **nichtproportionale**, so genannte dicktengleiche Schriftarten, zum Einsatz, bei denen alle Zeichen eine identische Breite aufweisen. Die wohl bekannteste dieser Schriften ist die Courier. Auch heute noch werden solche Schriften verwendet, z.B. um Programmtexte zu schreiben.

Zweitens unterscheidet man zwischen **Schriften mit Serifen (Antiqua) und solchen ohne Serifen (Grotesk)**, also serifenlosen Schriften. Serifen sind kleine Endstriche eines Buchstabens, umgangssprachlich auch „Füßchen“ genannt. Sie bilden eine horizontale Linie, an der sich das Auge des Lesers orientieren kann. Daher eignen sich Serifenschriften besonders für gedruckten Fließtext (Bücher, Artikel). Bei Postern, Plakaten, Schildern, etc. kommt es dagegen darauf an, auch auf größere Distanz einzelne Worte zu entziffern. Hier werden wegen ihrer größeren Klarheit Schriften ohne Serifen eingesetzt.

Courier:	Informatik	nichtproportionale	(dicktengleiche)
		Serifenschrift	
Times:	Informatik	proportionale	Serifenschrift
Arial:	Informatik	proportionale	Serifenlose (Grotesk-) Schrift

Angeführt sind nur die ganz groben Unterscheidungen der Typografie, die natürlich zum Beispiel in einem Textverarbeitungsprogramm alle unterschieden werden müssen.

In der Typographie bemüht man sich, Regeln für die gute Gestaltung von Druckwerken oder auch Bildschirmtexten aufzustellen. Regeln für gute Typografie sind allerdings immer an den historischen Kontext und die technischen Möglichkeiten gebunden. Im Laufe der

Zeiten haben sich viele Konventionen gebildet und verändert, die bei der Wahl einer Schrift und der Gestaltung zu berücksichtigen sind. Diese sind nach Zeit und Ort verschieden. So werden Anführungszeichen, Gedankenstriche, Satzzeichen und Überschriften in verschiedenen Ländern der Welt und selbst innerhalb Europas mitunter sehr unterschiedlich dargestellt und behandelt.

Zeichensätze – Text-Repräsentationen

Wir fokussieren unsere Betrachtung auf den **Zeichensatz**, also der Zuordnung von abstrakten alphanumerischen Zeichen (Buchstaben und Ziffern) sowie Sonderzeichen zu einem Zahlencode. Schon hier gibt es eine Unzahl von Möglichkeiten:

Wichtige internationale Zeichensätze			
ASCII	Einer der ältesten Computer-Zeichensätze – 7 Bit	1963 (1968)	Sehr weit verbreitet, in der Regel der Basis-Zeichensatz für Programmiersprachen, Internet-Adressen, etc.
Unicode ISO/IEC 10646 – 1991	Ein internationale Standard – (7) 8, 16 oder 32 Bit	1991	Universell, mit verschiedenen Code-längen. nimmt an Bedeutung stark zu.

Tabelle 12 Wichtige internationale Zeichensätze

ASCII (American Standard Code for Information Interchange)

ASCII wurde 1963 definiert und als ANSI-Standard X3.4 im Jahr 1968 genormt. Er basiert auf dem lateinischen Alphabet, wie es im modernen Englisch benutzt wird.

ASCII beschreibt als Code die Zuordnung von digital dargestellten natürlichen Zahlen zu den in der normalen Schriftsprache geschriebenen Zeichen. Wie der Name schon sagt, diente ASCII ursprünglich zur Darstellung von Schriftzeichen der englischen Sprache. Die erste Version als Sechs-Bit Code, noch ohne Kleinbuchstaben und mit kleinen Abweichungen vom heutigen ASCII, entstand im Jahr 1963. Im Jahr 1968 wurde dann der bis heute gültige Sieben-Bit ASCII-Code festgelegt.

ASCII																
Co de	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NU L	SO H	ST X	ET X	EO T	EN Q	AC K	BE L	BS	H T	LF	V T	FF	CR	SO	SI
1...	DL E	DC I	DC 2	DC 3	DC 4	NA K	SY N	ET B	CA N	E M	SU B	ES C	FS	GS	RS	US
2...	SP !	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6...	`	a	b	c	d	e	F	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	V	w	x	y	z	{		}	~	DE L

Tabelle 13 Die ASCII Code-Tabelle (in Hexadezimalcodierung)

Die Codes **0x00 bis 0x1F**, also die ersten 32 ASCII-Zeichencodes (Code 0x00 bis 0x1F) sind für Steuerzeichen (*control character*) reserviert. Dies sind Zeichen, die keine Schriftzeichen (Glyphen) darstellen, sondern die zur Steuerung von Geräten dienen (oder dienten) etwa Drucker oder Fernschreiber. Viele dieser Codes haben heute keine Bedeutung mehr, einige aber sehr wohl, siehe Tabelle 13.

Code **0x20 (SP)** ist das Leerzeichen (engl. *space* oder *blank*), welches in einem Text als Leer- und Trennzeichen zwischen Wörtern verwendet und auf der Tastatur durch die große breite Leertaste erzeugt wird.

Die Codes **0x21 bis 0x7E** sind alle *druckbaren* Zeichen, die sowohl Buchstaben, Ziffern und Satzzeichen enthalten.

Code **0x7F** (alle sieben Bits auf eins gesetzt) ist ein Sonderzeichen, welches auch als "Löschzeichen" bezeichnet wird (DEL). Dieser Code wurde früher wie ein Steuerzeichen verwendet, um auf Lochstreifen oder Lochkarten ein bereits gelochtes Zeichen nachträglich durch das Setzen aller Bits, d.h. durch Auslöchen aller sieben Markierungen, löschen zu können.

Heute haben nur noch wenige Steuerzeichen eine praktische Bedeutung (z.B. Line Feed, Form Feed, Carriage Return, Escape), die meisten Steuerzeichen werden praktisch nicht mehr verwendet. Manchmal werden sie auch missbraucht, um Zeichen zu übertragen, die im verwendeten Zeichensatz sonst nicht definiert sind.

Hex	Escape Codes in Python	Abk.	Name	Deutsch	(ursprüngliche) Bedeutung
00	\0	NUL	Null	Nullzeichen	Füllzeichen ohne Informationsgehalt. Kann nach Belieben in eine Zeichenkette eingefügt werden und wird vom Empfänger verworfen. Markiert das Ende eines Strings in C (NICHT in Python!)
07	\a	BEL	Bell	Tonsignal, Glocke	Erzeugt ein Tonsignal (Glocke oder Beep) am empfangenen Terminal. Benutzt als Alarmzeichen oder um auf Fehlersituationen aufmerksam zu machen.
08	\b	BS	Backspace	Rückschritt	Bewegt den Druckkopf/Cursor eine Position zurück. (Und löscht ggf. das dort gedruckte Zeichen)
09	\t	HT	Horizontal Tabulator	Horizontaler Tabulator	Bewegt den Druckkopf/Cursor zur nächsten vordefinierten Position (Tab-Stop) in der aktuellen Zeile.
0A	\n	LF	Line Feed	Zeilenvorschub	Bewegt den Druckkopf/Cursor in die nächste Zeile. Wenn zwischen Sender und Empfänger abgesprochen, bedeutet es "New Line", wobei die erste Druckposition der nächsten Zeile angefahren wird. Wird unter Unix als "Zeilenendezeichen" benutzt. Unter Microsoft Windows wird mit der Kombination "Carriage Return" + "Newline" eine Zeile beendet.
0B	\v	VT	Vertical Tab	Vertikaler Tabulator	<i>Bewegt den Druckkopf/Cursor zur nächsten vordefinierten Zeile.</i>
0C	\f	FF	Form Feed	Seitenvorschub	Bewegt den Druckkopf/Cursor zur ersten Druckposition auf der nächsten Seite. (Wirft die aktuelle Seite aus, löscht den Bildschirm)
0D	\r	CR	Carriage Return	Wagenrücklauf/Druckkopfrücklauf	Bewegt den Druckkopf/Cursor zurück in die erste Druckposition der aktuellen Zeile. Wird von Mac OS als Zeilenendezeichen ("New line") benutzt. Unter Microsoft Windows wird mit der Kombination "Carriage Return" + "Newline" eine Zeile beendet.
1B		ESC	Escape	Escape (wörtlich "Entkommen")	Steuerzeichen, das die Erweiterung des Zeichensatzes anzeigen soll. Es ist selbst der Anfang einer Sequenz von

					unmittelbar folgenden Zeichen, die eine besondere Bedeutung tragen.
--	--	--	--	--	---

Tabelle 14 Gebräuchliche ASCII Steuercodes. Die "\x"-Zeichen geben die Schreibweise als Literal für dieses Zeichen in der Programmiersprache Python an (und auch vielen anderen Sprachen wie C, C++, Java).

ASCII – Nationale Varianten, firmenspezifische Varianten, ISO/IEC 8859

Das **klassische ASCII**-Alphabet enthält insbesondere **keine diakritischen Zeichen** (zu Buchstaben gehörige kleine Zeichen wie Punkte, Striche, Häkchen oder Kringel, die eine besondere Aussprache oder Betonung markieren), die in vielen Sprachen auf der Basis des lateinischen Alphabets verwendet werden (wie Ö, ö, Ü, ü, Ä, ä) und z.B. auch kein „ß“.

Der internationale Standard ISO 646 (1972) war der erste Versuch, dieses Problem anzugehen. Er ist immer noch ein Sieben-Bit-Code und weil keine anderen Codes verfügbar waren, wurden einige Codewerte für spezifische Zeichen verwendet und ausgetauscht. So ist etwa die ASCII-Position 5D für die rechte eckige Klammer (]) in der deutschen Zeichensatz-Variante ISO 646-DE durch das große U mit Trema (Umlaut) (Ü) und in der dänischen Variante ISO 646-NO durch das große A mit Ring (Krouzek) (Å) ersetzt. Benutzte man z.B. eckigen Klammern, so führte dies z.B. mit dem Deutschen Alphabet ausgedruckt häufig zu ungewollten komischen Ergebnissen, indem etwa die Einschaltmeldung des Apple II von "APPLE][" zu "APPLE ÜÅ" mutierte.

Verschiedene Hersteller entwickelten eigene Acht-Bit-Codes. Der (Microsoft-) **Codepage 437** genannte Code war lange Zeit der am weitesten verbreitete, er kam auf dem IBM-PC unter MS-DOS, und heute noch in DOS- oder Eingabeaufforderungs-Fenstern von MS-Windows, zur Anwendung.

Bei späteren Standards wie **ISO/IEC 8859** werden **acht** Bits verwendet. Dabei existieren wieder mehrere Varianten, zum Beispiel *ISO 8859 definiert* 15 verschiedene 8-Bit-Zeichensätze weil wieder nicht alle gebräuchlichen sprachspezifische Zeichen unterzubringen waren.

Alle diese Ansätze sind heute obsolet, obwohl man sie in der Praxis noch häufiger vorfindet.

Unicode

Um den Anforderungen möglichst vieler Sprachen und Schriftsystemen gerecht zu werden, viele mathematische Zeichen repräsentieren zu können, etc. wurde der Unicode entwickelt. Er verwendet bis zu 32 Bit pro Zeichen. **Unicode** ist ein internationaler Standard, in dem langfristig für jedes sinntragende Zeichen bzw. Textelement **aller** bekannten Schriftkulturen und Zeichensysteme ein digitaler Code festgelegt werden soll. Das gemeinnützige „Unicode Consortium“ zeichnet sich für den Industriestandard Unicode verantwortlich. Von der ISO (International Organization for Standardization) wird die internationale Norm ISO 10646 herausgegeben. Beide Institutionen arbeiten eng zusammen. Seit 1993 sind Unicode und ISO 10646 bezüglich der Zeichenkodierung identisch. In der ISO heißt Unicode offiziell Universal Character Set (UCS). Diese Bezeichnung ist allerdings allgemein ungebräuchlich.

Während ISO 10646 lediglich die eigentliche Zeichenkodierung festlegt, gehört zum Unicode ein umfassendes Regelwerk, das u. A. für alle Zeichen weitere zur konkreten Anwendung wichtige Eigenschaften eindeutig festlegt wie Sortierreihenfolge, Schreibrichtung und Regeln für das Kombinieren von Zeichen.

Ziel ist es, ein Zeichen-Kodierungssystem zu etablieren, das den weltweiten Austausch, die Bearbeitung und Anzeige für geschriebene Texte in verschiedenen Sprachen und für verschiedene Disziplinen (Technik, Musik, etc.) unterstützt. Zusätzlich sollen auch klassische und historische Schriften unterstützt werden.

Die Codes von Unicode-Zeichen werden hexadezimal mit vorangestelltem U+ notiert. Der Coderaum von Unicode umfasst 17 Ebenen (*planes*) zu jeweils 65532 Zeichen (16 Bit) = 1.114.112 Zeichen bzw. sogenannte **Codepoints** (= eindeutige Zeichennummer) im

Codebereich von U+00000 bis U+10FFFF: Die ersten beiden Hex-Ziffern identifizieren die Ebene, die letzten 4 Ziffern identifizieren einen Codepoint in einer Ebene.

Zur besseren Übersicht sind die Ebenen weiter unterteilt in sogenannte Blöcke (*blocks*). Ein Block enthält häufig 128 verschiedene Codepoints, manche, z.B. für ostasiatische Schriften, aber auch einige Tausend.

Bislang, in Unicode 5.1, sind etwas mehr als 100.000 Code-Points individuellen Zeichen zugeordnet. Das entspricht nur etwa 10% des Coderaumes – viel Raum für zukünftige Erweiterungen.

In der Ebene 0: BMP (*Basic Multilingual Plane*) sind hauptsächlich Schriftsysteme, die aktuell in Gebrauch sind sowie Satzzeichen und Symbole, Kontrollzeichen, etc angeordnet. Der Block ist größtenteils belegt, so dass umfangreichere Schriftsysteme nicht mehr Platz finden.

Darauf folgt die SMP (*Supplementary Multilingual Plane*) als Plane 1. Sie enthält vor allem historische Schriftsysteme, aber auch größere Ansammlungen an Zeichen, die selten in Gebrauch sind, wie z. B. Domino- und Mahjonggsteine. Die SIP (*Supplementary Ideographic Plane*) als Plane 2 enthält ausschließlich selten genutzte CJK-Schriftzeichen (Chinesische, Japanische und Koreanische Schriftzeichen).

Die Planes 3 bis 13 sind noch nicht belegt, Plane 14 enthält spezielle Kontrollzeichen zur Sprachmarkierung, Plane 15 und 16 sind für eine „private“ Nutzungen reserviert.

Eine vollständige Übersicht zu den bisher festgelegten Unicode-Codepoints findet man unter <http://unicode.org/charts/>.

Die Zuordnung von Zeichen zu Blöcken ist nicht willkürlich, sondern orientiert sich an den bei der Einführung von Unicode in Benutzung befindlichen Zeichensätzen.

Unicode-Codepoint		Offizielle Bezeichnung	Kommentar
U+000000 U+00007F	–	Controls and Basic Latin	das klassische ASCII
U+000080 U+0000FF	–	Controls and Latin 1 Supplement	die zweite Hälfte des klassischen ISO/IEC 8859-Teil 1
U+000100 U+0000FF	–	Latin Extended A	
...			
U+000300 U+00036F	–	Combining Diacritical Marks	
U+002200 U+0022FF	–	Mathematical Operators	
...			

Zusammengefasst: Unicode-Codepoints kodieren jeweils **ein abstraktes Zeichen**. Vielen Unicode-Zeichen ist keine Glyphe zugeordnet. Aber auch sie gelten als „characters“. Neben den Steuerzeichen wie Zeilenvorschub (U+000A), Tabulator (U+0009) usw. sind allein 19 Zeichen explizit als Leerzeichen definiert, sogar solche ohne Breite, die u. a. als Worttrenner gebraucht werden für Sprachen wie Thai oder Tibetisch, die ohne Wortzwischenraum geschrieben werden. Die graphische Erscheinungsform eines abstrakten Zeichens (Glyph) ist durch den Unicode nicht festgelegt, hierzu muss eine Information über den zu benutzenden Fonts, z.B. Arial, Times Roman, etc. hinzukommen. Kein Font deckt alle Unicode Codepoints ab; in diesen Fällen wird von der Software meist ein spezielles Symbol ausgegeben, das für „undruckbares Zeichen“ steht, z.B: oder | .

Auf eine Besonderheit sei noch hingewiesen: Jedem Unicode-Codepoint ist eindeutig ein abstraktes Zeichen zugeordnet. Diese Eindeutigkeit gilt umgekehrt dies aber nicht: Ein Zeichen hat ggf. mehrere zulässige Codepoints, z.B. der Umlaut ä: U+00E4 kann in einem Zeichen oder aus zwei Zeichen, nämlich U+0061 (a) und U+0308 (¨), also dem Basiszeichen a und dem diakritische Zeichen (Pünktchen) getrennt repräsentiert

werden. Braucht man hier Eindeutigkeit, so muss man einen Unicode-String normalisieren. Dies ist allgemein nicht trivial, aber hierfür sind im Unicode Regeln festgelegt, siehe ggf. <http://www.unicode.org/reports/tr15/>.

Um ein Unicode-Zeichen zu speichern oder zu übertragen bräuchte man bei einer direkten Kodierung der Codepoints pro Zeichen minimal 3 Bytes; der Codebereich reicht ja von U+00 00 00 bis U+10 FF FF. Verglichen mit ASCII (7 Bit / 1 Byte) ist dies genau das Dreifache – große „Verschwendung“? -

Man benutzt sogenannte Unicode Transformation Formats, abgekürzt UTF. Üblich sind insbesondere 3 Varianten in denen sich jeweils alle 1.114.112 im Unicode-Standard enthaltenen Zeichen (Codepoints) darstellen lassen. Die verschiedenen Formate unterscheiden sich hinsichtlich ihres Platzbedarfs (Speichereffizienz), dem Kodierungs- und Dekodierungsaufwand (Laufzeitverhalten) sowie in ihrer Kompatibilität zu anderen (älteren) Kodierungsarten, zum Beispiel ASCII. Während beispielsweise einige Formate einen sehr effizienten Zugriff (wahlfreier Zugriff) auf einzelne Zeichen innerhalb der Zeichenkette erlauben, gehen andere sparsam mit Speicherplatz um. Daher ist bei der Auswahl eines bestimmten Unicode-Transformationsformats das für das vorgesehene Anwendungsgebiet geeignetste zu bestimmen. Häufig genutzte Kodierungen sind:

- UTF-32 kodiert ein Zeichen in 32 Bit und ist damit am einfachsten, da auf 32 Bit Maschinen Zeichen am schnellsten gehandhabt werden können, allerdings auf Kosten der Speichergröße – werden nur Zeichen des ASCII-Zeichensatzes verwendet, wird viermal so viel Speicherplatz benötigt wie bei einer Kodierung in ASCII.
- UTF-16 ist das älteste Kodierungsverfahren, bei dem ein oder zwei 16-Bit-Einheiten (2 oder 4 Bytes) zur Kodierung eines Zeichens verwendet werden. Diese Kodierung findet sich z.B. in Windows und .NET und Programmiersprachen wie Java.
- **UTF-8** kodiert Zeichen in 8-Bit-Einheiten (Bytes). Dabei wird ein Unicodezeichen in 1 bis 4 Bytes kodiert. Die Codepoints 0 bis 127, die dem ASCII-Zeichensatz entsprechen, werden in einem Byte kodiert, wobei das höchstwertige Bit stets 0 ist. Mithilfe des 8. Bits kann ein längeres Unicode-Zeichen eingeleitet werden, was sich auf 2, 3 oder 4 Byte erstreckt. Damit wird bei auf dem lateinischen Alphabet basierenden Schriften am effizientesten mit dem Speicherplatz umgegangen. Verwendet wird diese Kodierung in Betriebssystemen (GNU/Linux, Unix) und in verschiedenen Internetdiensten (E-Mail, WWW), sowie in Python (generell ab Version 3.0). Im Internet wird immer häufiger die UTF-8-Kodierung verwendet.

UTF-8 (Abk. für *8-bit Unicode Transformation Format*) ist die populärste Kodierung für Unicode-Zeichen; dabei wird jedem Unicode-Zeichen eine speziell kodierte **Bytekette von variabler Länge** zugeordnet. Das Kodierungsschema ist in Tabelle 15 angegeben.

Unicode-Bereich	UTF-8-Kodierung	Bemerkungen	Anzahl der erlaubten Codewörter	
00 0000 - 00 007F „ASCII-Bereich“	0x ₆ ... x ₀ (1 Byte)	In diesem Bereich (128 Zeichen) entspricht UTF-8 genau dem ASCII-Code: Das erste Bit ist 0, die darauf folgende 7-Bitkombination ist das ASCII-Zeichen.	2 ⁷	128

00 0080 - 00 0 7FF	110x ₁₀ ... x ₆ 10x ₅ ... x ₀ (2 Byte)	Das erste Byte beginnt mit binär 11, die folgenden Bytes beginnen mit binär 10; die <i>x</i> stehen für die fortlaufende Bitkombination des Unicode-Codepoints. Die Anzahl der Einsen bis zur ersten 0 im ersten Byte ist die Anzahl der Bytes für das Zeichen.	2 ¹¹ (- 2 ⁷)	1.920
00 0800 - 00 F FFF	1110x ₁₅ ... x ₁₂ 10x ₁₁ ... x ₆ 10x ₅ ... x ₀ (3 Byte)		2 ¹⁶ – (2 ¹¹)	63.488
01 0000 - 10 F FFF	11110x ₂₀ x ₁₉ x ₁₈ 10x ₁₇ ... x ₁₂ 10x ₁₁ ... x ₆ 10x ₅ ... x ₀ (4 Byte)		2 ²⁰	1.048.576
Summe				1.114.112

Tabelle 15 Kodierungsschema der UTF-8 Kodierung des Unico des

Betrachtet man die Bitfolgen etwas genauer, erkennt man die große Sinnfälligkeit von UTF-8:

1. Ist das höchste Bit des ersten Byte gleich 0, handelt es sich um ein gewöhnliches ASCII-Zeichen, da ASCII eine 7-Bit-Kodierung ist und die ersten 128 Zeichen des Unicode (der erste Block) genau diese sind. Damit sind alle ASCII-Dokumente (1 ASCII-Zeichen/Byte) automatisch aufwärtskompatibel zu UTF-8.

2. Ist das höchste Bit des ersten Byte gleich 1, handelt es sich um ein Mehrbytezeichen, also ein Unicode-Zeichen mit einer Zeichennummer größer als 127.

- Sind die höchsten beiden Bits des ersten Byte = 11, handelt es sich um das Start-Byte eines Mehrbytezeichens, sind sie = 10, um ein Folge-Byte.

Zu beachten: Jeder Unicode-Codepoint ließe sich in vier Varianten kodieren, erlaubt ist nur die jeweils kürzest mögliche Kodierung.

Für alle auf dem Lateinischen Alphabet basierenden Schriften ist UTF-8 die kompakteste Methode zur Abbildung von Unicode-Zeichen.

UTF-8, UTF-16 und UTF-32 kodieren alle den vollen Wertebereich von Unicode. Es gibt weitere Kodierungen für Spezialanwendungen, z.B. UTF-EBCDIC für Mainframe-Anwendungen, Punycode (für URLs, email-Adressen) etc. .

Leider ist die Welt in der Praxis nicht so heil, wie man es vielleicht aus obiger Darstellung ableiten könnte:

1. Viele „alte“ Kodierungen sind noch im Gebrauch, obwohl der Unicode mehr und mehr genutzt wird, aber oft ist die Unicode-Unterstützung noch unvollständig.

2. Die meisten Programmiersprachen unterstützen deshalb auch noch die klassische Zeichenkodierung mit 7/8 Bit Kodierungen für z.B. ASCII, ISO/IEC 8859 oder proprietären Codierungen. Weil „scheinbar“ einfacher, wird dieses von vielen Programmierern bevorzugt.

3. MS Windows (wie auch schon DOS, aber auch andere Betriebssysteme) benutzen so genannte Codepages, die proprietär sind und zum Teil gravierend von den Standards abweichen.

Wozu das führt, zeigt ein Beispiel: Das Schriftzeichen € (Euro) wurde 1997 von der Europäischen Kommission als Symbol für die europäische Gemeinschaftswährung eingeführt. Alle vorher entwickelten Codes hatten dieses Zeichen nicht: ASCII, ISO/IEC 646, ... Es kam zu hektischen „Integrationsmaßnahmen“ an verschiedenen Stellen und führte zum heutigen „Chaos“:

Das Euro-Währungssymbol wird im

- Unicode durch den Codepoint U+20AC repräsentiert.

- Im ISO 8859-15 (ISO-Latin-9) und ISO 8859-16 (ISO-Latin-10) findet man das Eurosymbol unter dem hexadezimalen Code 0xA4 repräsentiert.
- Unter Microsoft Windows kommt häufig die proprietäre Zeichenkodierung Windows-CP-1252 zum Einsatz, in der das Euro-Symbol durch 0x80 (hexadezimal) repräsentiert wird.
- Auf Apple Macintosh wird häufig MacRoman benutzt. Hier findet man das Euro-Symbol bei 0xDB.
- In HTML kann es durch € kodiert werden.

Universell verwendbar sind in der Praxis bis heute eigentlich nur die originären ASCII Zeichen (und natürlich der Unicode). Bei allen anderen 8-Bit Kodierungen ist man in der Praxis vor Überraschungen nie gefeit.

Zeichenketten (Strings) in Python

Python bis zur Version 2.6 (die von uns genutzte Version) implementiert noch zwei verschiedene Basis-Typen für Zeichenketten:

- Strings `<type 'str'>` (8-Bit, ein Oktett, ein Byte) und
- Unicode-Strings `<type 'unicode'>` (variable Codelänge von 8-32 Bits, in UTF-8 Kodierung)

Ab Version 3.0 gibt es in Python nur noch Unicode-Strings, also: str ist in Unicode kodiert.

In der Version 2.6 haben wir noch die „klassische“ Situation in der Zeichenbearbeitung: In der Python Definition ist der Zeichensatz für Strings nicht festgelegt. Die Default-Zeichenkodierung wird bei der Installation auf ihrem Rechner durch Auslesen einer „Systemvariablen zur Ländereinstellung“ festgelegt, kann also bei verschiedenen Installationen von Python verschieden sein. Folge ist: Programme, die eine konkrete Zeichenkodierung unterstellen, sind nicht einfach portabel und auf Rechnern mit anderen Zeichensätzen i.a. nicht fehlerfrei lauffähig.

Dies ist ein typisches Problem der sogenannten „Internationalisierung von Software“: Ein Programm soll so gestaltet sein, dass es leicht (ohne den Quellcode ändern zu müssen) an andere (natürliche) Sprachen und Kulturen angepasst werden kann. Im englischen Sprachraum wird Internationalisierung (internationalization) gerne mit I18N abgekürzt (im engl. Wort befinden sich 18 Buchstaben zwischen i und n). Dies erforderte diverse Vorkehrungen bei der Softwareentwicklung: Neben dem Zeichensatz (hier ist Unicode eine gute Lösung) müssen auch Datumsformatierungen, Dezimaltrennzeichen, Feiertage, Währungssymbole, unterschiedlich lange Bezeichnungen und kulturelle Aspekte (Farben, Symbole, etc.) zur Laufzeit des Programms angepasst werden: Ein sehr anspruchsvolles Unterfangen, dass wir hier nicht detailliert behandeln können.

Eine Übersicht zu den sehr unterschiedlichen Zeichenkodierungen auf verschiedenen Systemen findet man unter <http://www.manderby.com/mandalex/a/ascii.php>.

Um String-Literale zu erzeugen, schreibt man den Text innerhalb von

- einfachen `'TE'XT'` (Apostrophe),
- doppelten `"TE'XT"` (Gänsefüßchen)
- dreifachen `'''TEXT'''` oder `"""TEXT"""`

Anführungszeichen. Nebeneinander stehende String-Literale werden zusammengesetzt:

`'Dies ' 'ist ' 'ein ' 'TEXT-LITERAL'`

Es müssen jeweils die gleiche Anfangsmarkierung und Endmarkierung für einen String verwendet werden. Ansonsten ist die Wirkung gleich, nur können die jeweils nicht genutzten Zeichen im String vorkommen; bei dreifachen Anführungszeichen kann sich der String über mehrere Zeilen erstrecken, also ein Zeilenwechselzeichen enthalten.

Für einzelne Zeichen (character) gibt es keinen speziellen Typ. Diese sind Strings der Länge 1.

Je nach landessprachlicher Einstellung können Buchstaben mit diakritischen Zeichen (z.B. Umlaute) verwendet werden. **Alles was Sie auf der Tastatur tippen können, kann Teil des Strings sein, also nicht nur ASCII-Zeichen!** Zugelassen sind auch diverse Steuerzeichen zur einfachen Textformatierung oder Eingabe von Sonderzeichen.

<code>\</code>	Zeilenende; innerhalb eines Strings kein Effekt, am Ende einer Zeile „Zeilenfortsetzungszeichen“
<code>\\</code>	Schrägstrich rückwärts
<code>\a</code>	Steuerzeichen BEL (Tonsignal)
<code>\b</code>	Rückwärtsschritt (1 Zeichen)
<code>\f</code>	Seitenvorschub
<code>\n</code>	Zeilenvorschub
<code>\r</code>	Rücklauf zum Zeilenanfang
<code>\t</code>	(Horizontaler) Tabulator
<code>\v</code>	Vertikaler Tabulator
<code>[\' oder \"]</code>	funktionieren wie in C, braucht man in Python meist nicht, siehe oben
<code>\xhh</code>	hh Hexadezimaler Wert (Code) des Zeichens (genau 2 Hex-Ziffern)
<code>\ooo</code>	ooo octaler Wert (Code) des Zeichens (drei Oktalziffern)

Der Backslash (`\`) wird hier als sogenanntes „Fluchtsymbol“ (*escape*) genutzt, also als spezielles Steuerzeichen, das die Erweiterung des Zeichensatzes anzeigen soll. Es ist selbst der Anfang einer Folge von unmittelbar folgenden Zeichen, die eine besondere Bedeutung tragen.

Unicode Strings werden als Literal mit vorangestellten u gekennzeichnet (mit dem Präfix u), also z.B. als

```
u'Dies ist ein Unicode Text.'
```

Ansonsten gelten dieselben Schreibweisen und Möglichkeiten wie bei Strings. In einem Unicode-String können zusätzlich folgende Fluchtsymbole angegeben werden, um einen beliebigen Unicode-Codepoint einzugeben.

```
\uhhhh oder \Uhhhhhhh (genau 4 bzw. 8 Hexziffern)
```

z.B. für das €-Zeichen: `\u20AC`

```
u'\u20AC'
```

Die gleichen Fluchtsymbole finden sich übrigens in C und verwandten Programmiersprachen wie C++, Java, Perl und JavaScript Verwendung.

Operatoren auf String und Unicode: Auf dem String-Datentypen sind einige Operatoren definiert insbesondere

a + b	Konkateniert (fügt aneinander) die Strings a und b
a * b	Wiederholung: Entweder muss a oder b ein Integer sein. Der andere Operand ein String. Vervielfacht den String um die angegebene Zahl

Weitere Verküpfungsoperatoren, wie `/`, `**`, etc. sind nicht definiert. Eine Besonderheit ist noch der `%`-Operator (normalerweise der Modulo-Operator). Bei Strings wird dieser als Formatierungsoperator benutzt, siehe später bei der `print`-Funktion.

Strings sind Zeichen-Sequenzen (Zeichenketten) und gehören somit in Python zu den sogenannten **Sequenztypen**. Für Sequenztypen sind insbesondere Indizierungs- und Teilbereichsoperatoren definiert. Strings sind mit ganzen Zahlen, **beginnend bei Null**, indiziert. Um auf ein einzelnes Zeichen zuzugreifen, verwendet man den Index-Operator `s[i]` (mit Eckigen Klammern) wie folgt:


```
a = "Hello World"
b = a[4] # b = 'o'
```

Um einen Teilstring zu erhalten, benutzt man den **Teilbereichsoperator** (*slice*) `s[i:j]`. Dieser extrahiert alle Elemente von `s`, deren Index `k` im Intervall $[i, j)$ liegt. Falls einer der beiden Indizes weggelassen wird, so wird entweder der Anfang oder das Ende des Strings angenommen:

```
c = a[0:5] # c = "Hello"
d = a[6:] # d = "World"
e = a[3:8] # e = "lo Wo"
```

Anmerkungen zu Indexierungs- und Teilbereichsoperatoren:
Indexierung `S[i]`

Indexierung (anstelle der Integer-Literale können beliebige Ausdrücke stehen, die als Typ einen Integer liefern).

negative Indizes zählen vom Ende des Strings nach vorn.

Index 0 selektiert das erste Element.

Index -2 selektiert das zweitletzte Element.

Teilbereichsbildung (Slicing) `S[i:j]`

extrahiert zusammenhängende Bereiche des Strings

Die Bereichsgrenzen sind mit 0 und Stringlänge vorbelegt, können also ggf. weggelassen werden.

`S[1:3]` geht von Index 1 bis 2 (also ausschließlich der 3).

`S[1:]` geht von Index 1 bis zum Ende

`S[1:-1]` enthält alles, bis auf das letzte Element

`S[:]` ist eine Kopie des String `S`

Erweiterte Teilbereichsbildung `S[i:j:k]` mit drittem Parameter `k`

Das dritte Element ist eine Schrittweite

`S[::2]` ergibt jedes zweite Element der Sequenz `S`

`S[::-1]` ergibt die Umkehrung von `S`

Achtung: Teilbereichsoperatoren für Strings dürfen nur auf der rechten Seite des Zuweisungsoperators genutzt werden. (Strings sind sogenannte unveränderliche (*immutable*) Datentypen, wie alle bisher benutzten Typen. Wird einer Variablen ein neuer Wert zugewiesen, so wird in jedem Fall ein neues Objekt erzeugt!)

Vergleichsoperatoren

Sequenzen werden mit den Operatoren `<`, `>`, `<=`, `>=`, `==` und `!=` verglichen. Beim Vergleich zweier Sequenzen werden zunächst die ersten Elemente beider Sequenzen miteinander verglichen. Falls sie verschieden sind, bestimmt dies das Ergebnis. Sind sie gleich, setzt sich der Vergleich mit den zweiten Elementen beider Sequenzen fort. Dieser Vorgang wiederholt sich, bis entweder zwei verschiedene Elemente angetroffen werden oder kein weiteres Element in einer der beiden Sequenzen existiert. In diesem Fall ist die kürzere Sequenz kleiner, also falls `a` eine Untersequenz von `b` ist, so gilt `a < b`. Strings werden mit Hilfe einer lexikalischen Ordnung verglichen: Jedem Zeichen ist ein Code zugeordnet, je nach Zeichensatz des Rechners. Ein Zeichen ist kleiner als ein anderes, wenn seine Kodierung (als Integer) kleiner ist.

Speziell für Strings sind noch folgende Vergleichsoperatoren definiert

`x in s` liefert `TRUE`, falls `x` als Teilstring in `s` enthalten ist.

x not in s	liefert TRUE, falls x als Teilstring nicht enthalten ist.
Funktionen für Strings:	
len(s)	Liefert die Länge eines Strings als Integer.
min(s[,T]*)	Mit nur einem Argument liefert min einen String der Länge 1 (Character) mit dem kleinsten Wert (Element) in S. Ein Zeichen ist kleiner als ein anderes, wenn seine Kodierung (als Integer) kleiner ist. Mit mehr als einem Argument liefert min den kleinsten String zurück. Siehe bei den Vergleichsoperatoren, oben.
max(s[,T]*)	Mit nur einem Argument liefert max () einen String der Länge 1 (Character) mit dem größten Wert (Element) in S. Ein Zeichen ist größer als ein anderes, wenn seine Kodierung (als Integer) größer ist. Mit mehr als einem Argument liefert max () den größten String zurück. Siehe bei den Vergleichsoperatoren, oben.
ord(c)	Liefert den Kodewert (ein Integer im Intervall [0,255] eines Ein-Zeichen-Strings (eines Characters als str)) oder den Unicode-Codepoint (ein Integer eines unicode Characters). Achtung Python 2.6 unterstützt nur die Ebene 0: BMP (Basic Multilingual Plane) des Unicode. ord () auf ein Unicode-Zeichen angewendet liefert also ein Integer im Intervall [0,65535].
chr(i)	Liefert einen String der Länge 1 für ein Integer i im Intervall [0,255] in str-Kodierung (gemäß der genutzten Codetabelle).
unichr(i)	Liefert einen Unicode-String der Länge 1 für ein Integer i im Intervall [0, 65535] in unicode-Kodierung.

Weitergehende Möglichkeiten der String-Verarbeitung (Suchen, Aufteilen und Zusammenführen, Formatieren, Konstanten, usw.) findet man im String-Modul. Wie für die numerischen Datentypen müsste dieser Modul importiert werden. Die allermeisten Funktionen des string-Moduls stehen ohne Import auch als sogenannten String-Methoden bereit. Diese werden formal anders genutzt als Funktionen und würden den Rahmen unserer Betrachtungen jetzt sprengen.

3.7 Typisierung

Wir hatten oben schon festgehalten: Typ (*type*) oder Datentyp bezeichnet die Zusammenfassung konkreter Wertebereiche von Variablen und darauf definierten Operationen zu einer Einheit. Wir haben nun verschiedene elementare Datentypen kennengelernt, einschließlich ihrer Operationen und möglicher Funktionen. Die meisten Operatoren waren von der Art, dass die Verknüpfung zweier Objekte eines Typs wieder genau denselben Typ ergeben, z.B:

$$S_1 = V_1 + V_2$$

Wenn V_1 und V_2 vom Typ Integer sind, so ist auch Summe vom Typ Integer. Gleiches gilt für float oder str. Dies ist der Regelfall. Sind V_1 und V_2 nicht vom selben Typ, so ist das Ergebnis (Summe), zumindest beim Operator + (bisher) nicht definiert: Weder der Wert noch der Typ.

Betrachten wir ein Beispiel:

$$2 + 3.5 + '0001'$$

Obwohl der Wert dieses Ausdruckes für uns „eigentlich“ klar ist, nämlich 6.5, ist dieses für einen Compiler oder Interpreter keineswegs so: Wir unterstellen nämlich, dass man die Operanden wandeln kann, also:

$2.0 + 3.5 + 1.0 = 6.5$ (als Gleitpunktzahl)

rechnen wird. Aber warum nicht

`'2'+'3.5'+'0001' = '23.50001'` (als String)

interpretieren? Also: Es ist entscheidend wichtig, den Typ der Operationen zu kennen und diese nicht einfach zu interpretieren!

Ein weiteres Problem sei noch kurz angesprochen: Nehmen wir an, die Basis für die obere Rechnung wäre folgendes kleines Programm:

```
a = 2      # Integer 32 bit
c = '0001' # String 4 byte
b = 3.5    # Float 64 bit
```

Diese drei Variablen könnten im Speicher etwa wie folgt repräsentiert sein:

a	→	00000002	32-Bit Integer-Kodierung für 2
c	→	30303031	ASCII-Kodierung für den String '0001'
b	→	400C0000	64-Bit Float-Kodierung für 3.5
		00000000	

Würde man jetzt (versehentlich) a als Float interpretieren, so hätte a den Wert $4.643426084 \cdot 10^{-314}$, würde man jetzt (versehentlich) c als Float interpretieren, so hätte c den Wert $1.39804468550329 \cdot 10^{-76}$,

würde man jetzt (versehentlich) b als Integer interpretieren, so hätte es den Wert 1,074,528,256 ,

kurz, ein totales Tohuwabohu. Dieses „Experiment“ zeigt folgendes: Eine „Verwechslung“ des Typs einer Variablen führt schnell zum Program(mier)-Gau, zur Katastrophe.

Es ist daher offensichtlich, dass es gilt, solche Situationen in jedem Fall zu vermeiden und mögliche Programmierfehler so früh wie möglich zu entdecken. Hierzu unterscheiden wir:

- starke Typisierung (*strong typing*) - schwache Typisierung (*weak typing*)
- dynamische Typisierung (*dynamic typing*) - statische Typisierung (*static typing*)

Starke und Schwache Typisierung

Bei der **starken Typisierung** (*strong typing* oder strengen, strikten Typisierung) bleibt eine einmal durchgeführte Bindung zwischen Variable und Datentyp in jedem Fall bestehen. Eine nicht stark typisierte Sprache bezeichnet man als **schwach typisiert**.

Leider ist das Konzept des **strong typing** alles andere als eindeutig. In der Literatur finden sich diverse Regeln, die sich teilweise sogar widersprechen. Versucht man alle bisher in der Literatur aufgestellten Regeln für *strong typing* auf bekannte Programmiersprachen anzuwenden, hält keine Sprache dieser Überprüfung stand.

Eine Sprache ist **stark typisiert**, wenn

- Typkonvertierungen explizit durchgeführt werden müssen; es gibt keine implizite Typkonvertierung oder gar Typkonvertierungen generell verboten sind;
- es existiert ein komplexes, fein abgestuftes System an Typen mit Sub-Typen gibt;
- das Typ-System garantieren kann, dass aufgrund von Typ-Fehlern keine Fehlberechnungen stattfinden können;

- [sie Typüberprüfungen zur Compile-Zeit enthält? – Vermischt ggf. die Argumentation mit dynamischer – statischer Bindung;]

Vorteile der starken Typisierung: Der Compiler oder Interpreter kennt zu jeder Zeit den Typ einer Variablen im Speicher, d.h.

(1) Typfehler können entweder zur Compilezeit, spätestens beim Binden erkannt werden oder werden zur Laufzeit durch einen Interpreter geeignet abgefangen.

Folge: Ein Compiler erzeugt performanteren Code, weil Typprüfungen zur Laufzeit nicht mehr nötig sind!

Nachteile der starken Typisierung:

(1) **Variablen bleiben während der Laufzeit bezüglich Typ und Größe unveränderlich** (insbesondere auch die Größe eines zusammengesetzten Datentyps),

(2) die Übersetzer sind aufwendiger, weil dort mehr Aufwand für die Analyse anfällt, und

(3) viele „effiziente“ Programmiertricks auf Datenebene (z.B. Änderung eines kleinen Buchstaben „a“ in ein großes „A“ durch Subtraktion von 20₍₁₆₎) sind nicht möglich.

Beispiele für stark typisierte Sprachen (*keine der genannten Sprachen genügt allerdings allen Definitionsversuchen*):

- Java, Python, Pascal

schwach typisierte Sprachen:

- C / C++, PHP, Perl, JavaScript

Statische und Dynamische Typisierung

Bei der **dynamischen Typisierung** (engl. *dynamic typing*) erfolgt die Typzuweisung der Variablen zur Laufzeit eines Programms. Dies erspart es dem Programmierer, die Typisierung „von Hand“ vorzunehmen, bringt aber gewisse **Nachteile für die Performance und bei der Fehlersuche** mit sich.

Bei der **statischen Typisierung** muss zur Übersetzungszeit der Datentyp von Variablen bekannt sein. Dies erfolgt in der Regel durch Deklaration. Unter **Deklaration** versteht man die Festlegung von Bezeichner, Datentyp, Dimension (für zusammengesetzte Datentypen, siehe später) und weiteren Aspekten einer Variablen (oder eines Unterprogramms). Durch die Deklaration wird dem Compiler oder Interpreter diese Variable (bzw. dieses Unterprogramm) bekannt gemacht; es ist damit zulässig, diese an anderen Stellen im selben Quelltext zu verwenden.

Häufig werden die Begriffe Deklaration und Definition gleichgesetzt. Streng genommen ist **Definition** allerdings ein Sonderfall der Deklaration. Bei Variablen spricht man von Definition, wenn der Compiler Code erzeugt, der entweder statisch (im Datensegment) oder dynamisch (zur Laufzeit) Speicherplatz für diese Variable reserviert.

Das folgende Beispiel deklariert und definiert die Variable x mit dem Datentyp Integer.

```
int x;           // in C, C++, Java
INTEGER x;      ! in FORTRAN
```

Im folgenden Beispiel bewirkt das Schlüsselwort `extern`, dass die Variable x nur deklariert, nicht definiert wird. Die Definition muss an einer anderen Stelle in derselben oder einer anderen Quelltext-Datei erfolgen.

```
extern int x;    in C, C++, Java
```

Bei einer Deklaration ohne Definition überprüft erst der Binder (*Linker*), dass die Variable bzw. das Unterprogramm an anderer Stelle definiert wurde und verknüpft die Deklarationen miteinander.

Neben der **expliziten Deklaration** gibt es (nur noch historisch bedeutsam) in einigen Programmiersprachen (z.B. Fortran, Basic, PL/1) auch die Möglichkeit einer **impliziten Deklaration** von Variablen, z.B. alle Variablen, deren Bezeichner mit i, j, k, l, m, n beginnen sind vom Typ Integer. In diesem Fall führt das erste Auftreten einer Variablen zu einer automatischen Typzuordnung.

Vorteile der dynamischen Typisierung:

- (1) Auswahl des Operators wird zur Laufzeit entschieden, einfaches „Operator Overloading“ und einfacheres → „Generic Programming“.
- (2) Wesentlich kürzere Übersetzungs-Zeiten, weil viele Überprüfungen entfallen.
- (3) Variablen müssen nicht deklariert werden = Bequemlichkeit für den Programmierer, auch Geschwindigkeit beim Programmieren.
- (4) Variablen müssen nicht an einen festen Speicherbereich gebunden werden und damit eine feste Größe haben.

Nachteile der dynamischen Typisierung:

- (1) Typ von Variable/Wert wird zur Laufzeit (bei jedem Zugriff, jeder Zuweisung) überprüft → Werte im Speicher müssen (unveränderlichen) Type-Tags haben → geringere Performance (langsamer!) und mehr Speicherplatzbedarf für Variablen.
- (2) Typfehler sind erst zur Laufzeit erkennbar → mehr Laufzeitfehler.
- (3) Debugger benötigt wesentlich höhere Funktionalität.

Zusammenfassend ist festzuhalten:

static vs. dynamic typing und **strong vs. weak** typing sind orthogonal zueinander!

Die folgende Abbildung gibt die prinzipiellen Zusammenhänge wieder.

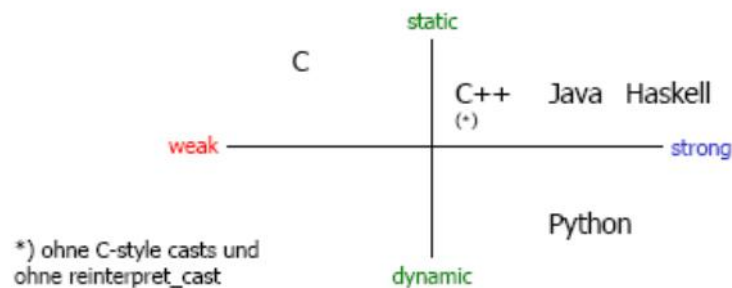


Abbildung 3 Einordnung einiger Programmiersprachen bzgl. ihres Typings

Typwandlung

Man kann darüber streiten, ob Typkonvertierungen (-wandlungen) überhaupt sinnvoll sind. In den meisten Programmiersprachen werden sie in „hoffentlich“ wohlkontrollierter Form aus folgenden Gründen zugelassen: aus der Mathematik sind wir es gewohnt, dass wir ganze Zahlen zu reellen Zahlen addieren können (Das ist auch sinnvoll, da ganze Zahlen ja eine Teilmenge der reellen Zahlen sind). Implizit unterstellen wir dann, dass das Ergebnis eine reelle Zahl ist. Diese „Selbstverständlichkeit“ sollte auch in Programmiersprachen abgebildet sein.

Es gibt aber diverse Fälle, bei denen wir den Typ einer Variablen verändern wollen, um zum Beispiel bestimmte Operationen ausführen zu können: Naheliegender ist dies bei Numerischen Daten, d.h. Umwandlungen zwischen Integer und Float. Entsprechendes gilt für die Strings, also zwischen str und unicode. Durch die Tastatur eingegeben werden grundsätzlich Strings; diese wollen wir (bestimmte Konventionen vorausgesetzt) aber auch als Zahlen interpretieren, usw. Kurz: Wir werden Funktionen brauchen, um zwischen verschiedenen Typen zu wandeln.

Wir unterscheiden in Programmiersprachen grundsätzlich zwei Arten von Typkonvertierungen:

- (1) implizite Typkonvertierung oder **coercion** (engl. *Nötigung, Zwang*)
- (2) explizite Typkonvertierung oder **cast(ing)** (engl. *eingießen, formen, werfen, ...*)

Coercion finden wir sehr häufig bei Zahlen, wie das einführende Beispiel es schon nahe legt. Dabei ist eine Regel unterlegt, dass wenn zwei nichtgleiche Zahlentypen miteinander verknüpft werden sollen, zunächst zum allgemeineren (höheren) Typ gewandelt wird, also z.B. eine

$$\mathbb{N} \rightarrow \mathbb{Z} \rightarrow \mathbb{Q} \rightarrow \mathbb{R} \rightarrow \mathbb{C}$$

Dies vollkommen gerechtfertigt und problemlos möglich, weil

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}.$$

Bei Integer und Float stimmt dies nur eingeschränkt, weil sich nicht alle Integerzahlen (z.B. sehr große) als Float exakt darstellen lassen – trotzdem ist dies üblich. Bei String und Unicode ist diese Beziehung „ist Teilmenge“ auch gegeben, ein Coercion bietet sich an. Achtung: Jede Programmiersprache hat sowohl für die explizite Typwandlung (Casting) als auch für die implizite Typwandlung (Coercion) ihren eigenen Regelsatz.

Typkonvertierung ist ein mächtiges – aber durchaus nicht unproblematisches – Verfahren.

Funktionen zur Typwandlung in Python

In Python haben wir bisher diverse elementare Datentypen kennengelernt. Diese lassen sich mit den in Tabelle 16 angegebenen Konvertierungsfunktionen in andere Typen wandeln. Der Name der Konvertierungsfunktion ist schlicht das Kürzel des Zieltyps. Die Wandlung ist **immer überwacht**, d.h. bei Verletzung der Nutzungsbedingungen wird ein Fehler generiert. (Solang dieser Fehler nicht „abgefangen“ wird (betrachten wir später), führt dieses zum Abbruch des Programms!)

Typ (Kürzel)	Konvertierungsfunktionen	Anmerkungen
Integer (int)	<code>int(O[,basis])</code> <code>ord(C)</code>	Ist O ein String, muss O ein gültiges Literal für Integer sein, optional kann eine Basis (beliebiger Integer) angegeben werden. C ist ein String der Länge 1.
Long Integer (long)	<code>long(O[,basis])</code>	obsolete in Version 3
Float (float)	<code>float(O)</code>	Ist O ein String, muss O ein gültiges Literal für Float sein.
Complex (complex)	<code>complex(O)</code>	Ist O ein String, so muss O ein gültiges Literal für complex, float oder integer sein.
Boolean (bool)	<code>bool(O)</code>	Jeder Wert $\neq 0$ bei numerischen Datentypen oder dem „leeren String“ liefert 'TRUE'
String (str)	<code>str(O)</code> <code>repr(O)</code> oder <code>`I`</code> (Backticks) <code>chr(I)</code> <code>hex(I)</code> <code>oct(I)</code>	liefert ein pretty-print (String) von O sehr ähnlich wie <code>str(O)</code> , erzeugt ein von <code>eval()</code> auswertbaren String, wird später behandelt Parameter muss ein Integer im zulässigen Wertebereich sein, liefern alle Strings
Unicode String (unicode)	<code>unicode(O)</code>	obsolete in Version 3; wie <code>str()</code> , nur liefert dies einen Unicode-String

Tabelle 16 Funktionen zur Typkonvertierung in Python (`ord()`, `chr()`, `hex()`, `oct()` sind streng genommen keine Funktionen zur Typwandlung, sondern Funktionen zur Zeichenverarbeitung)

Eine implizite Typwandlung (**coercion**) findet man in Python nur an 2 Stellen:

- in numerischen Ausdrücken und zwar nach folgendem Schema:
`bool` \rightarrow `int` \rightarrow `long` \rightarrow `float` \rightarrow `complex`
Wenn in einem Ausdruck verschiedene Numerische Datentypen vorkommen, dann wird zum umfassenderen Typ gewandelt.
- bei String-Ausdrücken nach folgendem Schema:
`str` \rightarrow `unicode`
- Jede Variable in Python kann boolesch interpretiert werden, wenn ein Wahrheitswert erwartet wird:
False für numerische Datentypen, wenn sie den Wert 0 hat
bei Strings die Länge 0 hat
True für numerische Datentypen, wenn sie den Wert $\neq 0$ hat
bei Strings eine Länge $\neq 0$ hat, also auch `'\x00'`

Ansonsten ist **Python stark typisiert**, d.h. jede Vermischung von Typen erzeugt einen Typfehler. Durch die dynamische Typisierung kann eine Variable jedoch jederzeit den Typ wechseln, hier muss man sehr aufpassen!

Als Programmiererin kann man zu jeder Zeit den Typ einer Variablen erfragen, durch die Funktion

`type(O)`

Diese Funktion liefert ein Type-Objekt,

Diese Funktion kann für jedes Objekt (jede Variable) angewendet werden liefert ein Type-Objekt, dass ausgedruckt wie folgt erscheint:

`<type 'Typ-Kuerzel'>`

Man kann auf diese Art die Typen verschiedener Variablen vergleichen, z.B.

```
type(I) == type(F)
```