www.uni-frankfurt.de

Modul: B-PRG1
Grundlagen der Programmierung 1 und Einführung in die Programmierung EPR

V04 Elementare Datentypen - Teil 3

Numerischer Datentyp: Float **und**

Typing allgemein

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Rückblick auf Teil 1 und 2 der "Elementaren Datentypen"

- Gab es Schwierigkeiten mit den Übungsblättern?

- Schon behandelte Datentypen:
  - Numerische Datentypen: **Integer / int**
  - Boolesche Datentyp: **Bool / bool**
  - Nonetype
  - Datentyp **String str**

studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Unser heutiges Lernziele heute

- *Elementare Datentypen sind solche, die von der Programmiersprache direkt unterstützt werden und meist auch von der Hardware direkt unterstützt werden.*
- *Das Wandeln von Dezimalzahlen < == > Float*
- *The "The Perils of Floating Point"*
- *Besonders wichtig und sehr unterschiedlich in verschiedenen Programmiersprachen ist das sogenannte **Typsystem**, starke versus schwache Typisierung; statische versus dynamische Typisierung.*

studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Übersicht

- ***Numerische Datentypen***
  - *Gleitpunktzahlen (floating point number)*
  - *Erste Programmiererfahrungen mit float, Literale und Operatoren*
  - Wandlung von Dezimalzahlen
  - "*The Perils of Floating Point" , also Gefahren!*
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*
- *Abschluss*

Gleitpunktzahlen allgemein

Eine **Gleitpunktzahl** - engl: *floating point number*
(auch: Gleitkommazahl, manchmal auch Fließkommazahl) ist eine **halblogarithmische Darstellung**)

Wird oft durch Hardware (floating point unit) unterstützt.

ist eine (**meist** approximierte) Kodierung einer rationalen oder reellen Zahl in einer **festgelegten Anzahl von Bits** (meist 32, 64, seltener 16, 128 oder gar 256 Bits).

Die Menge der Gleitkommazahlen ist eine endliche **Teilmenge** der rationalen Zahlen, meist erweitert um einige Spezialelemente (+Unendlich, –Unendlich, NaN (="Not A Number"), –0, usw.

Gleitpunktzahlen (allgemein)

- Zur Kodierung einer Zahl wird eine **Mantisse m** und ein **Exponent e** zu einer bestimmten, festen Basis b benutzt
  $8{,}432 \cdot 10^{23}$ (wissenschaftliche Zahlendarstellung)
- Eine Zahl $a \neq 0$ wird durch zwei Zahlen ***m*** und ***e*** solcherart dargestellt, dass $a = m \cdot b^e$ gilt.
  Dabei ist die **Basis b** (auch: Radix) eine beliebige natürliche Zahl ≥ 2.
- Die Zahl ***m*** wird **Mantisse** genannt und ist eine Zahl mit ***p*** Stellen (der so genannten Präzision, engl. ***precision***) der Form $\pm z0,z_{-1}z_{-2} \ldots z_{m-1}$.
- Hierbei steht z für eine Ziffer zwischen 0 und *b* – 1.

Normalisierte und normierte **Mantisse, Wertebereich**

- Liegt die Mantisse im Wertebereich $1 \leq m < b-1$ (im Fall b=2 ist die Vorkommazahl 1), so spricht man von einer **normalisierten Mantisse**.
- Liegt die Mantisse im Wertebereich $1/b \leq m < 1$ (also im Fall b=2, ist die Vorkommazahl 0 und die erste Nachkommastelle ist ungleich 0), so spricht man von einer **normierten Mantisse** (0.xxxx-Form).
- Gegenüber einer Integerdarstellung kann mit Gleitkommazahlen bei gleichem Speicherplatzbedarf ein viel größerer Wertebereich abgedeckt werden.
- Beispiel: 32 Bit Zweierkomplement: $-2{,}147 \cdot 10^9 \leq z \leq 2{,}147 \cdot 10^9$
  32 Bit Gleitpunktzahl (IEEE 754): $-3{,}403 \cdot 10^{38} \leq z \leq 3{,}403 \cdot 10^{38}$

IEEE 754 und IEEE 754-2008

Das gebräuchliche und häufig auch durch Hardware unterstützte Format ist in der Norm **IEEE 754** (ANSI/IEEE Std 754-1985; IEC-60559 - International version) festgelegt.

IEEE 754-2008 ist eine Revision des IEEE 754 und die heute gültige Form.

Diese Norm legt die Standarddarstellungen für **binäre** Gleitkommazahlen fest und definiert Verfahren für die Durchführung mathematischer Operationen, insbesondere für Rundungen und für

Beinahe alle modernen Prozessoren folgen diesem Standard.
Ausnahmen:

- **Java Virtual Machine** mit den Java Typen float und double, die nur einen Subset der IEEE 754 Funktionalität unterstützt.

IEEE 754-2008 definiert folgende Formate

- **Gleitkommazahlen** mit
  halber (16 Bit) (Miniformat),
  einfacher (32 Bit),
  doppelter (64 Bit) und
  vierfacher (128 Bit) Genauigkeit
- Es gibt „analoge“ weitere Zahlendarstellung mit einem ganzzahligen Vielfachen von 32 Bits und größer 128 Bits
- NaN, z.B. irrationale Zahlen
- Eine Signaling NaN ist eine NaN mit gesetztem Bit 7.
- Darstellungen von $\pm\infty$ existieren und sind leicht erkennbar

IEEE 754-2008

benutzt **normalisierte** Gleitkommazahlen (NZ)

auf der Basis b = 2.

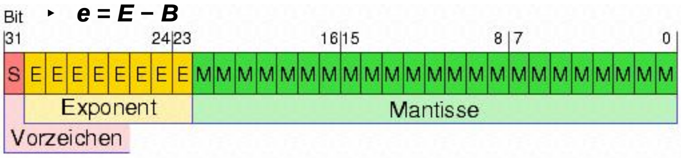
Das **Vorzeichen** $\boldsymbol{s} = (-1)^S$ wird in einem Bit S
(S = 0 positive Zahlen und S = 1 negative Zahlen)

Der **Exponent e** ergibt sich aus der in den Exponentenbits gespeicherten nichtnegativen Binärzahl E durch Subtraktion eines festen **Biaswertes B**:

$e = E - B.$

IEEE 754 – hier 32 Bit

- Die Mantisse ist ein Wert, der sich aus den p Mantissenbits M berechnet, als $m = 1 + M / 2p$. Einfacher ausgedrückt, denkt man sich an das Mantissenbitmuster M links eine 1. angehängt:
- Da die Mantisse immer mit 1. beginnt, braucht dieses Bit nicht mehr gespeichert zu werden (**hidden Bit**). Damit gewinnt man ein zusätzliches Bit „Genauigkeit“. Also:
  - **m = 1.M**
  - $\boldsymbol{s = (-1)S}$
  - $\boldsymbol{e = E - B}$



single precission „float“

Prinzip der IEEE 754-2008 Floatingpoint-Typen

Vorzeichen **1 Bit**	S	Exponent mit Vorzeichen (Versatz)	Nachkommaanteil M der normalisierten Mantisse (1.M)

mini	1	5 Bits	10 Bits
float	1	8 Bits	23 Bits
double	**1**	**11 Bits**	**52 Bits**
Extended	1	15 Bits	112 Bits

Python

Python

Typ	Größe	Mantisse m	p Mant. bei NZ	Exponent e	e_{min}	e_{max}	Werte der Ch. bei NZ	Bias	Informationsgehalt in Bit
B16 (Mini)	16 Bit	10 Bit	11 Bit	5 Bit	–14	15	1 ≤ E ≤ 30	15	16
b32 (single)	32 Bit	23 Bit	24 Bit	8 Bit	–126	127	1 ≤ E ≤ 254	127	32
b64 (double)	**64 Bit**	**52 Bit**	**53 Bit**	**11 Bit**	**–1022**	**1023**	**1 ≤ E ≤ 2046**	**1023**	**64**
b128	128 Bit	112 Bit	113 Bit	15 Bit	–16382	16383	1 ≤ E ≤ 32766	16383	128
k = 32j, j >= 4	k Bit	k - rnd (4 ld (k)) + 12 Bit	k - rnd (4 ld (k)) + 13 Bit	rnd (4 ld (k)) - 13 Bit	1-emax	2^(k-p-1) - 1		emax	k
d32	32 Bit	20+ Bit	7 Ziffern	6 Bit	–95	96		101	31,83
d64	64 Bit	50+ Bit	16 Ziffern	8 Bit	–383	384		398	63,73
d128	128 Bit	110+ Bit	34 Ziffern	12 Bit	–6143	6144		6176	127,53
k = 32j, j >= 1	k Bit	15 k/16 - 10 Bit	9 k/32 - 2 Ziffern	k/16+4 Bit	1–emax	3 2^(k/16 +3)		emax + p - 2	
d64	64 Bit	50+ Bit	16 Ziffern	8 Bit	–383	384		398	63,73

Floating Point Literale

- `123.4,   -42.0,   0.,   .1` einfache Float: die Kennzeichnung erfolgt durch den **Dezimalpunkt**
- `3.11e-8` oder `-4E11` (ein kleines e oder ein großes E kennzeichnet eine Zahl in Exponentialschreibweise (wissenschaftliche) zur Basis 10)
- Die Zahlen **vor und nach** dem e/E werden immer als Dezimalzahlen interpretiert.
(Oktalzahlen oder Hexadezimalzahlen sind hier nicht zugelassen!)

studiumdigitale megadigitale GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Floating Point Operatoren

- Es funktionieren alle Operatoren wie bei Integer bis auf die „Bit Manipulatioren“, also

  x << y, x >> y, x & y; x ^ y und x | y funktionieren **NICHT**

Auch mit X, Y jeweils Float funktionieren:

- X // Y — Liefert das „Ganzzahlige“ Ergebnis einer Division (Vorkommateil - Ergebnis der Form xxx**.0**). Das Ergebnis ist von Typ float,
- X % Y — Liefert den „Nachkommateil“ einer Division - Ergebnis der Form **0.xxx** als float.

Beispiele:

- 13.0 // 3.2 — liefert den Wert 4.0
- 13.0 % 3.2 — liefert den Wert 0.2 (0.1999999999999993)

studiumdigitale megadigitale GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Wichtige Funktionen für Float:

- divmod(X,Y) — Ergebnis ist ein Tupel (X // Y, X % Y)
- pow(X,Y[,Z]) — X zur Potenz Y [modulo Z]
- round (X [, N]) Liefert ein float, gerundet auf N Dezimal-Ziffern nach dem Komma. Default N = 0.

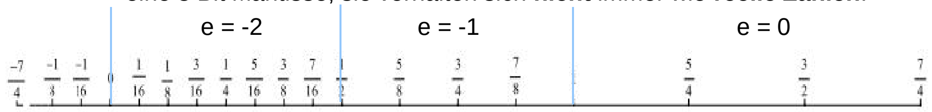
studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

„*The Perils* (Risiken) *of Floating Point*“ (1)

nach dem Paper von Bruce M. Bush (1996)

- Gleitpunktzahlen sind nicht gleich dicht (im gleichen absoluten Abstand) auf dem Zahlenstrahl:
- Bei ***single* Precission (float)** liegen z.B. **zwischen 1 und 2** 8.388.607 verschiedene Gleitkommazahlen, zwischen **1023 und 1024** dagegen nur **8191 (weniger als 1%)**. (Weil eben nicht die absolute Genauigkeit konstant ist, sondern die relative, also die Anzahl der signifikanten (=korrekten) Stellen (**Precision**).)
- Die darstellbaren Zahlen liegen **nicht dicht** auf dem Zahlenstrahl, hier für eine 3 Bit Mantisse, sie verhalten sich **nicht** immer wie **reelle Zahlen**.



studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Übersicht

- ***Numerische Datentypen***
  - *Gleitpunktzahlen (floating point number)*
  - *Erste Programmiererfahrungen mit float, Literale und Operatoren*
  - Wandlung von Dezimalzahlen
  - "*The Perils of Floating Point*"
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*
- *Abschluss*

Algorithmus (Dezimalzahl ➔ binäre float)

Es gibt natürlich viele Möglichkeiten. Wir führen hier nur eine vor.

0. Vorzeichenbit:
Negative Zahlen eine Eins als Vorzeichenbit, positive Zahlen eine Null.

1. Mantisse:
Wir führen die Rechnung für den Vorkomma-Anteil und Nachkomma-Anteil getrennt aus:

- Für **Vorkommaanteil** durch **"Kettendivision"**
- Für **Nachkommaanteil** durch **"Kettenmultiplikation"**

2. Mantissenteile konkatenieren und in eine 1.xyxyxy-Form verschieben
➔ **wahrer Expnent e** Links: negativer Exponent -- Rechts: positiver Exponent

3. Gespeicherter Exponent E = e + Bias

Beispiel

Wie lautet die 32-Bit-IEEE-Codierung (float) für -42.234375 als Binärvektor?

0. Vorzeichenbit: Negaive Zahl ➔ Bit 31 = 1
1. Mantisse

42 / 2 = 21 Rest 0	0,234375 * 2 = 0,468750
21 / 2 = 10 Rest 1	0,46875 * 2 = 0,93750
10 / 2 = 5 Rest 0	0,9375 * 2 = 1,8750
5 / 2 = 2 Rest 1	0,875 * 2 = 1,750
2 / 2 = 1 Rest 0	0.75 * 2 = 1.5
1 /2 = 0 Rest 1	0,5 * 2= 1

➔ **101010.001111**

2. Verschieben nach rechts: e = 5 mal: ➔ **1.**01010001111
3. Exponent: E = ´101+1111111 = **10000100**

Beispiel (Ergebnis)

also:

- 1 **10000100 010100011110 … 0**

reduzierte **Mantisse**

Exponent E

Vorzeichen(Sign):

Wandlung:
Binäre floating-point Repräsentation → Dezimalzahl

0. Vorzeichen

1. wahren Exponent bestimmen

e = E-Bias (=127 für float)

2. Mantisse M verschieben:→ m

3. Aufaddieren gemäß Stellenwert :

Jetzt mit dem Rechner / dem Python Interpreter wandeln – int!

- Bei **Integer** ist das doch ganz toll: die Dezimalzahl wird als string ausgegeben!

```
>>> hex(254)
'0xfe'
>>> oct(254)
'0o376'
>>> bin(254)
'0b11111110'
>>> bin(-254)
'-0b11111110'
```

ACHTUNG
Dies sind nicht die exakten Repräsentationen im Hauptspeicher, sondern **lesbare Vorzeichenbehaftete Reps (nicht** Zweierkomplement**!)**

Jetzt mit dem Rechner / dem Python Interpreter wandeln – float!

- Hier gibt es leider nur die hex() Funktion und die Ausgabe ist etwas gewöhnungsbedürftig!

```
>>> float.hex(3.14159)
'0x1.921f9f01b866ep+1'

>>> 3.14159.hex()
'0x1.921f9f01b866ep+1'

>>> float.fromhex('0x1.921f9f01b866ep+1')
3.14159
```

Interpretieren der hex()-Ausgabe für floats

```
>>> (-1/3.14159).hex()
'-0x1.45f318e7adaf5p-2'
```

Python 3.5 benutzt als float-Repräsentation die IEEE 457 **b64 (double) … also**
1 VZ,
52 Bit Mantisse (+ hidden Bit)
11 Bit Exponent mit Bias von 1023

VZ	E = e+B	1.<13*4 Bit Mantisse>
-	E = 1021 = -2+1023	4 5 f … 5
1	01111111101	0100 0101 1111 … 0101

➔ Etwas einfacher als mit Hand, insbesondere natürlich für kompliziertere Zahlen.

… und noch eine interessante Repräsentation

Alle floats sind rationale Zahlen, also können sie als Bruch, mit ganzzahligen Zähler und Nenner dargestellt werden.

```
>>> 3.125.as_integer_ratio()
(25, 8)

>>> 3.14159.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Zusammenfassung

Wir können jetzt beliebige reelle Zahlen meist als Näherung in eine float-Darstellung (rationale Zahl zur Basis 2) wandeln.

Die Dichte der Zahlen nimmt bei großen Werten stark ab.

Dies hat diverse Konsequenzen und darum müssen wir uns noch kümmern.

Die Präzision von "double" Gleitpunktzahlen Was braucht man realistischerweise?

Die "Perils"

1. Entfernung Erde-Mond oder Erde-Mars auf den Meter genau?

   Erde-Mond: 384.400 km = 384.400.000 m, also 9 Dezimalstellen in m
   Erde-Mars: 225.300.000 km = 225.300.000.000 m also 12 Dez-Stellen
   Erde-Pluto: 4.275-7.525*10^6 km ≈ 13 Dezimalstellen.

Die Präzision von "double" Gleitpunktzahlen
Was braucht man realistischerweise?

2. Umsatz eines Weltkonzerns auf den Cent genau?

   Facebook 12.466 Mio. US-Dollar (2014) in US-Cent:
   1.246.600.000.000,
   also 13 Dezimalstellen.

   Walmart 476.294 Mio US-Dollar (2013)
   in US-Cent: 47.629.400.000.000
   Also 14 Dezimalstellen (weltweit umsatzstärkstes Unternehmen)

Welche Präzision haben "double" Gleitpunktzahlen?

- Entscheidend ist die Anzahl der Mantissenbits: bei 'double' sind das:
- 52+1 Bit (wg. Normalisierung, d.h. die Mantisse ist **1.**xyxyx)
- in Dezimalstellen:
- 53 (Bit) / $\log_2 10$ Dezimalstellen = 53/3.322=15,955 Dezimalstellen, also fast 16 Dezimalstellen
- Also: es ist doch alles gut:

  … also, was wollen wir mehr?

Wir können Abstände zum Mars auf einen Meter genau bestimmen, die Controller können den Umsatz von Walmart auf den Cent genau bestimmen, … was ist das Problem?

Aber: merkwürdige Ergebnisse

```
>>> (1 + 1e16 - 1e16)
0.0
>>> (1 + 1e15 + 1 - 1e15)
2.0

>>> 0.1 + 0.2
0.30000000000000004

>>> .1 + .1 + .1 == .3
False
```

Das müssen wir uns etwa genauer anschauen!

Zahlen verschiedener Größenordnung (*absorption*)

```
>>> (1 + 1e16 + 1 - 1e16) # Die Eins ist zu klein!
0.0
>>> (1 + 1e15 + 1 - 1e15) # Präzisionsgrenze!
2.0
>>> 1.0 - 1e-17
1.0
>>> 1.0 - 1e-16
0.9999999999999999
>>> 2.0 - 1e-16
2.0
```

Vorsicht: Die Addition bzw. Subtraktion einer betragsmäßig viel kleineren Zahl ändert die größere Zahl nicht. Dies nennt man **Absorption**.

Auslöschung (cancellation)

```
>>> 1000.2 - 1000
0.20000000000004547
```

Bemerkenswerte falsche Ziffern schon an der **14. Dezimalstelle.**

Vorsicht: Unter Auslöschung (cancellation) versteht man den Effekt, dass bei der **Subtraktion** fast gleich großer Zahlen das Ergebnis viel ungenauer wird.

Vergleiche auf gleichen Wert sind fast immer falsch!

```
>>> .1 + .1 + .1 == .3
False
>>> abs((.1 + .1 + .1) - .3) < 1e-16
True
>>> abs((.1 + .1 + .1) - .3) < 1e-17 # nicht optimal
False

>>> import math
>>> math.isclose(.1 + .1 + .1, 0.3) # ist richtig!
True
```

- Anstelle des == benutzen Sie bitte (nach import math)
- **math.isclose()**

Sicheres vergleichen in Python (1)

```
Help(math.isclose
isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool
```

Determine whether two floating point numbers are close in value.
rel_tol
maximum difference for being considered "close", relative to the magnitude of the input values
abs_tol
maximum difference for being considered "close", regardless of the magnitude of the input values

Return True if a is close in value to b, and False otherwise.

For the values to be considered close, the difference between them must be smaller than **at least one of the tolerances**.

Sicheres vergleichen in Python (2)

```
is_close(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool
```

-inf, inf and NaN behave similarly to the IEEE 754 Standard. That is, NaN is not close to anything, even itself. inf and -inf are only close to themselves.

… aber das geht jetzt zu weit ➔ Numerik (3. oder 4. Semester)

Zusammenfassung:
VORSICHT beim Programmieren mit float

1. In Python sind floats **maximal** 16 Dezimalstellen genau – Es kann aber je nach Rechenweg deutlich weniger sein!
2. Wandlungsfehler treten **bei jeder Wandlung** vom Dezimalsystem ins Binärsystem und vice versa auf.
3. Nur **sichere** Vergleiche (mit Epsilon) oder **math.isclose()** nutzen!
4. Vorsicht bei Subtraktionen (und ggf. auch Additionen), sie können sehr schnell die Präzision des Ergebnisses reduzieren.
5. Python benutzt auf fast allen Plattformen 'double' – die sogenannte Maschinenpräzision sind fast 16 Bit.

Alternativen zu float

Python besitzt einen

- **`decimal` — Decimal fixed point and floating point arithmetic Modul**

und einen

- **`fractions` — Rational numbers Modul**.

- Beide können im Prinzip die floats ersetzen, haben aber durchweg eine höhere Laufzeit.

Übersicht

- ***Numerische Datentypen***
  - *Gleitpunktzahlen (floating point number)*
  - *Erste Programmiererfahrungen mit float, Literale und Operatoren*
  - Wandlung von Dezimalzahlen
  - "*The Perils of Floating Point*"
- *Typisierung*
  - *Starke und Schwache Typisierung*
  - *Statische und Dynamische Typisierung*
  - *Typwandlung*
- *Abschluss*

Typisierung

Beispiel:

X = ‚ANTON'
Y = ‚ & '
Z = ‚BERTA'.
U = X + Y + Z

arithmetisch interpretiert macht das keinen Sinn, erst recht nicht, dieses auf Datenebene plump zu errechnen.

Vielmehr bedeutet der Operator + bei Zeichenketten eine Aneinanderreihung (Konkatenation), also

U = ‚ANTON & BERTA'

Datentypen

- Die Bedeutung von Operatoren in Ausdrücken hängt von der Art (=dem **Typ**, engl. *type*) der Daten ab

  Gleiche Operatorenzeichen (z.B, +) können abhängig vom Datentyp durchaus Verschiedenes bedeuten
- Ein **Datentyp** in der Informatik ist die **Zusammenfassung von Objektmengen mit den darauf definierten Operationen**.
- **Grundsätzlich** dürfen nur **gleiche Datentypen** miteinander verknüpft werden! (➔ starke Typisierung).
- In Python ist dann auch das Ergebnis vom selben Typ (➔ dynamische Typisierung).

Beispiel (1):

- 2 + 3.5 + ’0001’
- Wir als Mensch interpretieren vermutlich:

  2.0 + 3.5 + 1.0 = 6.5 (als Gleitpunktzahl)
- Aber warum nicht:
- ’2’ + ’3.5’ + ’0001’ = ’23.50001’ (als String) interpretieren?
- Also: Es ist entscheidend wichtig, den Typ der Operanden zu kennen!

Beispiel (2)

```
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 00000000 64-Bit Float-Kodierung für 3.5

a als Float interpretiert, so hätte a den Wert $4.643426084 \cdot 10^{-314}$,
c als Float interpretiert, so hätte c den Wert $1.39804468550329 \cdot 10^{-76}$,
b als Integer interpretiert, so hätte es den Wert 1,074,528,256 ,

- kurz, ein totales Tohuwabohu.

Beispiel (3)

Achtung: Bei der Division von Integer-Zahlen können Brüche (➔ float) entstehen: z.B.

7 / 2 = 3 ½ oder 3.5 (float) aber 6 / 2 = 3 (integer)

In der Mathematik führen wir diese „Coercion“ automatisch durch, oder?
In Python (ab Version 3.X) auch!

Aber es gibt auch die ganzzahlige Division:
7 // 2 = 3
Und die Modulo-Divison (Rest):
7 % 2 = 1

werden float, wenn einer der Operanden float ist

Also:

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:

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

Starke 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**.

- stark typisierte Sprachen: Java, **Python**, Pascal
- schwach typisierte Sprachen: C / C++, PHP, Perl, JavaScript
- Starke Typisierung schützt vor vielen Programmierfehlern!

Dynamische versus statische Typisierung

- **dynamischen Typisierung** (engl. *dynamic typing*) erfolgt die Typzuweisung der Variablen zur Laufzeit eines Programms (Bindung), z.B. durch eine Zuweisung
- Dies erspart es dem Programmierer, die Typisierung „von Hand" durch eine Deklaration durchführen zu müssen.
- 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 und weiteren Aspekten einer Variablen.

Casting und Coercion

- **implizite** Typkonvertierung oder **coercion** (*engl. Nötigung, Zwang*)
- **explizite** Typkonvertierung oder **cast(ing)** (engl eingießen, formen, werfen, …)
- **Coercion** finden wir sehr häufig bei Zahlen, also:

  Integer ➔ Float ➔ Complex

  Wie in der Mathematik, macht Sinn ... Trotzdem Vorsicht!

„Echte“ Casting-Funktionen in Python

Ziel-Typ (Kürzel)	Konvertierungsfunktionen (Quelle)
Integer (**int**)	`int(O)`
Float (**float**)	`float(O)`
Complex (**complex**)	`complex(O)`
Boolean (**bool**)	`bool(O)`

String (**str**)	`str(O)`
	`repr(O)`
	`ascii(o)`
	`B.decode(encoding='utf-8')`

Ganz einfach:
Ziel-Typ ist der Funktionsname!

O: (beliebiges) Objekt

S: String

Übrigens: Was macht Python bei
`int(3.1)`
`int('3.1')` und bei
`int(False)`

Spezielle Castings Integer ←→ String

Ziel-Typ (Kürzel)	Konvertierungsfunktionen (Quelle)
Integer (**int**)	`int(str[,basis=10])` `ord(C)`

String (**str**)	`hex(int)`
	`oct(int)`
	`bin(int)`
	`chr(int)`

C: „Character“ = ein-elementiger „String“

int: Integer

Also sind chr(int) und ord(C) echte „Gegenspieler“ (Umkehrfunktionen).
Der Wert des Integer entspricht dem Unicode-Codepoint.

„Gegenspieler“ sind auch hex(int), oct(int),bin(int) und int(str[,basis=10])

studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Weitere „echte“ Casting-Funktionen für Typen, die wir noch nicht kennen. (Nur zur Vollständigkeit.)

Ziel-Typ (Kürzel)	Konvertierungsfunktionen (Quelle)
String (str)	`B.decode(encoding='utf-8')`
Bytes (**bytes**) 1)	`bytes(S or iterable)` `S.encode(encoding='utf-8')`
Bytearray 1)	`bytearray(S or iterable)`
Frozenset	`frozenset(iterable)`
Set	`set(iterable)`
Tupel	`tupel(iterable)`
List	`list(iterable)`

Gilt auch für zusammen-gesetzte Datentypen, die wir noch nicht besprochen haben.

Ganz einfach: **Ziel-Typ ist der Funktionsname!**

iterable: Ein Objekt, dass die Methode __iter__ hat:

1) Wenn der Typ des Argumentes = S ist, dann muss auch ein encoding angegeben werden.

studiumdigitale megadigitale

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Abschluss - Zusammenfassung

Puuuuu……hhhh, da war echt zügig

Viele, viele Details, ich weiß!

NICHT VERGESSEN: **PRG04 und EPR03 (Pair Programming) fürs nächste WE**

Ausblick ... Nächsten Montag

Erste Schritte im Software-Engineering

Teile davon brauchen Sie schon fürs EPR 3

... Und übrigens: Danke für Ihre Aufmerksamkeit!