Elementare Datentypen

In diesem Abschnitt geht es um Elementare Datentypen und wie diese von Computern gespeichert bzw. codiert werden.

Datenstruktur vs Datentyp: Bei einer Datenstruktur liegt der Fokus auf der Repräsentation der Daten wo-hingehend beim Datentyp der Fokus auf den Operationen liegt.

Byteorder Ist die Frage, ob man im Speicher eines Computers zuerst das höherwertige Byte - MSB(Most significant Byte) ($A4$) zuerst oder das niederwertige Byte - LSB(Least significant Byte) ($D1$) zuerst:

$$A4\ B3\ C2\ D1_{(16)}$$

Man spricht von Little Endian wenn man das LSB zuerst speichert. Diese Art hat den Vorteil, dass Addition, Subtraktion und Multiplikation (beginnt mit der kleinsten
Stelle) ist hier einfach ist.
Man spricht von Big Endian wenn man das MSB zuerst speichert. Diese Art hat den Vorteil, dass Dividieren & Vergleichen einfacher ist und Vergleichsoperationen für Zahlen und Texte gleich sind. Heutzutage sind beide Technologien im Einsatz. Beim Austausch von Daten im Internet wird aber immer Big Endian verwendet. Verwendet der Computer selber das andere System, muss die Byteorder umgedreht werden.

Ganze Zahlen

Codierung: Grundsätzlich: Codieren einer Dezimalzahl durch Darstellung als Binärzahl. Problem: Wie wird das Vorzeichen codiert? Vorraussetzung ist hierbei immer dass wir eine feste Stellenzahl gegeben haben. Also dass die Anzahl der Bits aus denen eine Zahl besteht fixiert ist!

- Erste Ziffer als Vorzeichen interpretieren. Also $0\ldots$ positiv, $1\ldots$ negativ. Problem: Es gibt eine positive und negative 0. Außerdem ist die Arithmetik kompliziert
- Einerkomplement: Für eine negative Zahl wird jedes Bit der entsprechenden positiven Zahl invertiert, also $-3_{10} = 1100_{(2)}$. Also wie viele Bit die Zahl haben soll. Problem: Es gibt eine positive und negative 0. Außerdem ist die Arithmetik kompliziert
- Zweierkomplement: Für eine negative Zahl wird jedes Bit der entsprechenden positiven Zahl invertiert, also $-3_{10} = 1100_{(2)}$, dann eins zu der Zahl addiert. Also wie viele Bit die Zahl haben soll. Damit kann man dann wie gewohnt addieren. Man kann dabei eine Negative Zahl mehr als Positive speichern, also ist der Wertebereich von -2^{n-1} bis $2^{n-1} - 1$ darstellbar.
- Excess-x-Code: Hierbei wählt man einen Bias B. Der Wert einer Zahl e ergibt sich aus der nicht negativen Binärzahl E durch Subtraktion eines festen Biaswertes B, also $e = E - B$. Wobei E hier die zu speichernde Zahl ist. B ist bei n Bit meist 2^{n-1}, also $100\ldots_{(2)}$.

Python - Syntax:

- Dezimalzahlen wie erwartet z.B. `100, 143, 0, -3`
- Binärzahlen mit führender 0b bzw. 0B z.B, `0b101, -0B01, 0b111`
- Oktalzahlen mit führender 0o bzw. 0O z.B. `0o777. -0O632`
- Hexzaheln mit führender 0x bzw. 0X z.B. `0X12, -0xaaaf, 0XAAFF`

Python - Besonderheiten:

- Ints in Python können nahezu beliebig lang werden.
- Speicherbedarf sind mindestens 24-Byte (8 für den Wert, 8 für den Referenzzähler, 8 für Typ)

Python - Operationen: Die Operatoren sind von oben nach unten nach Priorität geordnet!

Operation	Beschreibung
`x ** y`	Exponential-Bildung x^y (Achtung: rechts-assoziativ)
`+x, -x,` `~x`	Einstellige Operatoren Invertiere x bitweise (nur Integer)

`x * y`
`x / y`
`x % y`
`x // y`
	Multiplikation (Wiederholung)
	Division Modulo (-Division) = Division mit Rest
	Restlose Division [2])

`x + y` `x - y`	Addition (Konkatenation) Subtraktion
`x << y, x >> y`	Bitweises „Schieben“ **(nur bei Integer)**
`x & y`	Bitweises Und **(nur bei Integer)**
`x ^ y`	Bitweises exklusives Oder **(nur bei Integer)**
`x | y`	Bitweises Oder **(nur bei Integer)**

Wichtig: Für Bitoperationen also den blauen, stellt Python negative zahlen im Zweierkomplement zur Verfügung.
Wichtig 2: Der Operator `~` flippt alle Bits einer Zahl und macht eine Negative Zahl daraus. Diese Zahl ist im Einerkomplement, wird aber bei der Ausgabe im Zweierkomplement interpretiert. Also z.B.:

```
print(~4) -# gibt -5 aus, da 4=0b100 also ~4=0b011 also im Zweierkomlement 0b011-1 ->
0b010 -> 0b101 = 5 also -5
```

Boolescher Datentyp

Codierung: Einfache Darstellung als Bit.

Python Syntax: `True = 1`, `False = 0`.

Python - Besonderheiten: In Python kann jede Variable als Boolean genutzt werden. Die Umwandlung funktioniert dabei wie folgt:

- False: für numerische Datentypen, wenn Wert 0 ist. Bei Strings(sequentiellen) Datentypen bei Länge 0
- True: für numerische Datentypen, wenn Wert ungleich 0 ist. Bei Strings(sequentiellen) Datentypen eine Länge ungleich 0.
  Um eine Variable in eine Boolean umzuwandeln kann man die Funktion `bool()` nutzen.

Eine weitere Besonderheit ist, das Booleans in Python auch Integers sind. Somit kann man mit Booleans wie mit Integers rechnen. True ist dabei die Zahl 1 und False die Zahl 0.

Python - Operationen: Python stellt folgende Operatoren zum Arbeiten mit Booleans zur Verfügung.

- <x> and <y>: Ist True wenn sowohl x als auch y True ist.
- <x> or <y>: Ist True wenn mindestens eins von x und y True ist.
- not <x>: Ist True wenn x False ist.
  Es gibt auch noch sogenannte Vergleichsoperatoren. Das sind Operatoren, die Booleans als Ergebnis liefern. Es gibt die folgenden: `<, >, <=, >=, ==, !=, in, is`
  Wichtig zu erwähnen ist aber der Unterschied zwischen `is` und `==`. Während `==` prüft ob zwei Elemente den gleichen Wert haben, prüft `is` ob es sich wirklich um das selbe Element handelt. Also der Wert auch unter der selben Adresse gespeichert ist.

None Type

Der Wert None (Datentyp = NoneType) in Python sagt aus, dass eine Variable keinen Wert hat. Dies ist vorallem nützlich wenn man einer Variable erst später einen Wert zuweisen will oder ausdrücken will, dass eine Suche Ergebnislos war.

Python Syntax: `None`

Python - Besonderheiten:
None ist dabei ein sogenanntes Singleton Objekt. Das bedeutet es gibt nur genau eine Instanz vom Datentyp NoneType nämlich der Wert None.
In Python geben Funktionen, die kein Ergebnis liefern immer None zurück.

Python - Operationen: None verglichen mit allem außer sich selbst ist immer False. Es wird dabei empfohlen vergleiche mit None immer nur mit `is` oder `is not` zu machen!

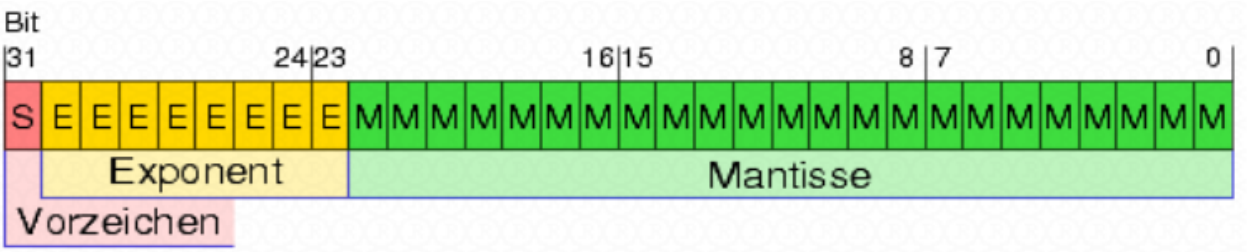
Gleitpunktzahlen

Gleitpunktzahlen sind eine endliche Teilmenge der Rationalen Zahlen erweitert um die Werte "+Unendlich", "-Unendlich", "NaN" und "-0".

Codierung: Es wird die sogenannte: IEEE 754-2008 Codierung verwendet.
Dabei ist die Anzahl der Bits von entscheidender Rolle. Bei 32 Bit spricht man von float und bei 64 Bit von double.
Man stellt nun eine Kommazahl in sogenannter Wissenschaftlicher Darstellung dar. Also also $z_0, z_{-1}z_{-2}\dots z_{-(p-1)} \cdot b^n$ wobei b hierbei die Basis ist und $z_0, z_{-1}z_{-2}\dots z_{-(p-1)}$ die Mantisse, also eine Zahl mit p (=Präzision) Stellen ist. Die Mantisse kann dabei normalisiert, also $1 \leq m < b$ sein oder normiert also $1/b \leq m < 1$.
Für IEEE ist die Basis nun 2 und die Mantisse normalisiert, also $m := 1.M$, die Hochzahl E Excess-x-Code codiert und das Vorzeichen der Mantisse einfach als Bit gespeichert. Es ergeben sich also die folgenden Werte

$$\begin{aligned} \text{Mantisse} &: m = 1.M \\ \text{Vorzeichen} &: s = (-1)S \\ \text{Exponent} &: e = E - B \end{aligned}$$

die wie folgt im Speicher des Computers liegen



Hier noch eine Tabelle, die die Anzahl an Bits für die jeweiligen Teile einer Floationpoint Zahl angiebt.

NAME	S	EXPONENT MIT VORZEICHEN $B = 2^{N-1}$	NACHKOMMAANTEIL M DER MANTISSE $M = 1.M$
mini - 16Bit	1	5	10
float - 32Bit	1	8	23
double - 64Bit	1	11	52
extended - 128Bit	1	15	112

Händische Codierung:
In Gleitkommadarstellung Codieren:

- Schritt 1: Vorzeichenbit S bestimmen.
- Schritt 2: Zahl in Binärdarstellung bringen. Dafür Kettendivision/Kettenmultiplikation verwenden

- Schritt 3: Zahl in 1.xxxx Form verschieben (Siehe Multiplikation/Division mit der Basis). Je nachdem um wie viele Stellen man das Komma verschieben muss ergibt sich e.
- Schritt 4: M durch weglassen der 1 bei 1.xxxx bestimmen.
- Schritt 5: e muss nun nach E Codiert werden. Wobei $E = e + B$
- Schritt 6: Alle Anteile durch hinzufügen von 0 vor der Zahl auf die benötigte Bitanzahl bringen.
- Schritt 7: Anteile wie folgt anordnen: SEM
  Von Gleitkommadarstellung Codieren:
- Schritt 1: Vorzeichen des Ergebnisses bestimmen
- Schritt 2: E ablesen und durch rechnen von $e = E - B$ den wahren Wert des Exponenten bestimmen## Gleitpunktzahlen
- Schritt 3: Die Mantisse durch hinzufügen von 1 zu M bestimmen
- Schritt 4: Das Komma in der Mantisse um e Stellen nach Rechts verschieben um die Zahl zu erhalten
- Schritt 5: Vorzeichen mit Zahl kombinieren.

Probleme mit Floats:

- *Dichte der Zahlen*: Ein Grund wieso es oft zu Problemen kommt ist, dass die Dichte der durch Gleitkommadarstellung codierten Zahlen auf dem Zahlenstrahl mit der Größe der Zahlen stark abnimmt. Das bedeutet, dass man zwischen 0 und 0.5, wesentlich mehr Dezimalzahlen darstellen kann als zwischen 0.5 und 1. Im vergleich dazu sind in den Reellenzahlen zwischen 0 und 0.5 natürlich genau gleich viele Zahlen wie zwischen 0.5 und 1. Zustande kommt diese Phänomen durch die Wissenschaftliche Darstellung welche für die Codierung verwendet wird. Die Mantisse hat dabei ja eine fixe Größe wodurch auch nur eine bestimmte Anzahl an Zahlen A_p dargestellt werden können. Je nach Exponenten werden diese A_p Zahlen dann auf ein Interval von z.B. 0 bis 0.5, 0 bis 1, 0 bis 1.5 usw aufgeteilt.
- *Nicht alle Zahlen darstellbar*: Wie wir beim Umwandeln von Dezimalzahlen in ein anderes Zahlensystem bereits gesehen haben, kann es beim Umwandeln zu unerwarteten Ergebnissen kommen. So ist z.B. die Zahl $2.675_{(10)}$ im Dualsystem eine Periodische Zahl $10.1010\overline{1100}_{(2)}$. Wird sie nun Gleitkomma-codiert, so werden aber nicht alle unendlichen Nachkommastellen gespeichert. Somit geht aber Information verloren.
- *Rundung*: Der Gleitkomma Standard IEEE definiert für das Runden von Floats, dass diese immer bevorzugt auf die nächste gerade Zahl gerundet werden sollen. Also 2.7 wie gewohnt auf 3 aber 2.5 auf 2. Für die Zahl $2.675_{(10)}$ gerundet auf 2 Dezimalstellen würde man nun $2.68_{(10)}$ erwarten. Tatsächlich wird aber auf $2.67_{(10)}$ gerundet. Dies ist, da die unendlichen Nachkommastellen, die ja benötigt werden um $2.675_{(10)}$ binär dazustellen, bei der kodierung als Gleitkomma Zahl verloren gehen. In Wirklichkeit ist also nicht die Zahl $2.675_{(10)}$ gespeichert sondern eine etwas kleinere weshalb auf $2.67_{(10)}$ gerundet wird.
- *Absorption*: Dieses Problem tritt auf wenn man mit Zahlen unterschiedlicher Größenordnung rechnet. Addiert man z.B. zu einer Zahl $1e16$ die Zahl 1 so reicht die Auflösung der Floationpoint codierung nicht dazu aus, das Ergebnis dazustellen. Dasselbe Problem tritt auch auf wenn man eine sehr kleine Zahl von einer wesentlich größeren Abzieht. Also zum Beispiel $2.0 - 10^{-16}$.

- *Auslöschung*: Dieses Problem tritt auf wenn man fast gleich große Zahlen voneinander abzieht. So kann es passieren, dass $2.2 - 2.0 = 0.20000000000000018$ Die Genauigkeit der Differenz ist also wesentlich geringer als die Präzision.
- *Vergleiche auf Gleichheit*: Diese problem tritt auf wenn man zwei Zahlen in Gleitkommadarstellung auf Gleichheit vergleicht. So kann es passieren, dass $0.1 + 0.1 + 0.1! = 0.3$ ist. Deshalb sollte man in Programmiersprachen in denen Gleitkommadarstellung verwendet wird, immer spezielle Vergleichsfunktionen verwenden.

Python Syntax: Wie erwartet. Besonderheit ist, dass man durch `e` bzw `E` Zahlen in Wissenschaftlicher Darstellung schreiben kann. Also z.B. `10E-2`.

Python - Besonderheiten: Wie bei der Floatingpoint Codierung erklärt können durch diese Codierung neben Dezimalzahlen auch noch die Werte "+0.0", "-0.0", "+Infinity", "-Infinity" und "NaN" codiert werden.
Besonderheiten sind nun:

- dass $x \cdot -0.0 = -0.0$ ist.
- division durch 0 egal ob plus oder minus liefert einen ZeroDivisionError
- ist eine Zahl klein genug wird sie zu +/- 0.0
- liefert eine Rechnung ein Ergebnis größer als darstellbar, so wird dieses zu +/-infinity
- mittels math.isfinite(...) kann man auf endlichkeit prüfen
- Arithmetische Operationen mit infinity geben +/-infinity oder NaN
- Arithmetische Operationen mit NaN geben NaN
- NaN ist NIE gleich zu irgendetwas, nicht einmal zu sich selbst#
- +/-infinity ist nur sich selbst ähnlich.
- will man auf NaN prüfen so gibt es die Funktion math.isnan(...)

Python - Operationen:

- Gleitkommadarstellung einer Zahl in Hex: `float.hex(...)`. Die Aussage ist dabei +/-, dann die Mantisse in Hex mit dem Hiddenbit dann ein "p" gefolgt vom Exponenten.
- Gleitkommadarstellung in eine Zahl wandeln: `float.fromhex(...)`
- Alle Operationen auf Ints bis auf Bitoperationen.
- Vergleich von Floats mit `math.isclose(a, b, rel_tol=1e-09, abs_tol=0.0)` wobei `rel_tol`: Die Maximaldifferenz relativ zur Größenordnung der Inputwerte ist und `abs_tol`: Die Maximaldifferenz unabhängig von den Inputwerten ist.
  Außerdem:

```python
print(divmod(3.14,1)) # Ergebnis ist ein Tupel (3.14//1, 3.14 % 1)
print(pow(3.14, 2.0)) # 3.14^2
print(round(2.25,1)) # Rundet 2.25 auf 1 Dezimalziffer. Default = 0
                     # Achtung! Python rundet dann immer auf die nächste
                     # ganze Zahl. Also 2.5 auf 2 und 2.25 auf 2.2
print(int(3.14)) # Streicht alle Nachkommastellen von 3.14
print(float(3)) # Erzeugt einen Floatwert aus Integern
print(float("3.14")) # Erzeugt einen Floatwert aus Text
print(float("inf")) # Kann auch andere erlaubte Floatwerte aus Text erzeugen
print(float("+Nan"))
print(float("INFInitY")) # Groß Klein Schreibung egal
```

Text

Zeichensatz vs. Zeichencode: Ein Zeichensatz ist nach der definition oben das gleiche wie ein Alphabet also, eine Sammlung an Symbolen. Ein Zeichensatz kann dabei auch aus den Symbolen von zwei Alphabeten bestehen(z.B.: aus dem lateinischen und griechischen Alphabet).
Ein Zeichencode oder eine Zeichencodierung hingegen ist *mehr* als nur ein Zeichensatz. In einer Zeichencodierung wird jedem Symbol in einem Zeichensatz eine eindeutige Zahl(meist in Binär) zugeordnet.

Codierung - ASCII: Im ASCII Code (American Standart Code for Information Interchange) werden Symbole als 7 Bit Zahlen codiert.

Codierung - ISO/IEC 8859: Diese Codierung gibt es in 15 verschiedenen Ausführungen um so europäische, arabische, ... Zeichen abdeken zu können. Die Symbole werden hier als 8 Bit Zahlen codiert.

Codierung - Unicode: Hierbei werden immer nur abstrakte Zeichen(characters) kodiert. Die graphische Darstellung (Glyphen) wird nicht kodiert.
Der Code Funktioniert so, dass im ersten Schritt jedem Unicode-Zeichen ein eindeutiger sogenannter Codepunkt zugeordnet wird. Codepoints sind Zahlen zwischen 0x0 und 0x10FFFF. Dieser Bereicht ist dann noch in 17 Ebenen eingeteilt. Spricht man von einem Codepoint wird meist die Hex Darstellung der Zahl mit dem Präfix U+ angegeben. Also zum Beispiel U+0041. Zusätzlich zum Codepoint hat jedes Zeichen dann auch noch einen eindeutigen Namen.
Im zweiten Schritt werden die Unicode-Zeichen dann mittels eines sogenannten Unicode Transformation Formats (UTF) auf eine Binärzahl abgebildet. Dies hat den Vorteil, dass man einfach unterschiedliche UTF's für unterschiedliche Zwecke verwenden kann. Übliche Encodings sind:

- UTF-8: Codiert in 1 bis 4 Bytes(variable Länge!), wobei die Codepoints die dem ASCII-Zeichensatz entsprechen in einem Byte codiert werden.
- UTF-16: Codiert in 2 oder 4 Bytes. Hierbei ist darauf zu achten, dass es je nach Reihenfolge der Bytes einen Code UTF-16BE oder UTF-16LE gibt.
- UTF-32 Codiert Zeichen immer in genau 32 Bit. Achtung auch hier auf die Bytereihenfolge.

Python Syntax: String Ausdrücke müssen mit " oder ' beginnen und enden. Innerhalb der Anführungszeichen sind alle UTF-8 Zeichen erlaubt. Will man mehrzeilige String schreiben, so kann man anstelle von einem " oder ' auch einfach """ oder ''' verwenden. Es gibt dabei sogenannte Escapecharacters. Das sind spezielle Zeichen, denen ein \ vorgehängt werden muss. Hier eine Tabelle von all diesen Zeichen:

ESCAPE SEQUENCE	MEANING
\\	Backslash ()
\'	Single quote (')
\"	Double quote (")
\n	ASCII Linefeed (LF)

ESCAPE SEQUENCE	MEANING
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh
\N{name}	Character name in the Unicode database

Python - Besonderheiten: Strings sind Sequences und unterstützen somit Indexing, Slicing und erlauben es über die Einträge zu iterieren.

Python - Operationen:

- Konkatenation mit + z.B.: `"abc"+"def"`
- Wiederholung mit * z.B.: `"abc"*2`
- Alle Vergleichsoperatoren - vergleichen nach Zahlenwert des Codepoints - kürzeres Wort ist kleiner
- `char(i)` liefert die String Repräsentation für den Unicode code point i
- `ord(c)` liefert den Unicode code point des Zeichens c
- ...

Python - Byte: Ein Byte Objekt ist ein codierter String. Mittels `"abc".encode(): String -> Byte` und `b"abc".decode: Byte -> String` kann man zwischen Strings und Bytes hin und her wandeln. Beide Methoden akzeptieren eine Codierung z.B.: "utf-8".