Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-09 Dozent: Karsten Flügge

Themen des Tages

- Variable vs Objekte
- list comprehension

- Übergabe von Argumenten an Programm
- Module

- Ausdruck versus Anweisung
- Expression / Statement

- Binär / Hex
- Speichern / Laden II

Themen des Tages

Variable vs Objekte

Beim (um)setzen einer Variable auf ein (neues) Objekt wird kein Objekt verändert!

Das ist unabhängig davon ob das Objekt an sich veränderlich ist oder nicht

```
bob  = {"name": "Bob", "alter": 25} # dictionary
fred = {"name": "Fred", "alter": 35}
mein_bester_freund = bob # Referenz aufs selbe 'Objekt' Bob
annas_bester_freund = bob # selber Bob in drei Variablen
print(mein_bester_freund)
mein_bester_freund["alter"]=26  #  Annas Freund Bob ist der selbe, auch 26

mein_bester_freund = fred # ich setze eine Variable neu, der Rest bleibt gleich
print(annas_bester_freund) # immer noch Bob, 26
```

Themen des Tages

Variable vs Objekte

Beim (um)setzen einer Variable auf ein (neues) Objekt wird das Objekt nicht verändert!

Das ist unabhängig davon ob das Objekt an sich veränderlich ist oder nicht

```
bob   = {"name": "Bob", "alter": 25}
fred = {"name": "Fred", "alter": 35}
mein_bester_freund = bob # Referenz aufs 'Objekt' Bob
annas_bester_freund = bob

# Bob hat Geburtstag gehabt
# bob["alter"] = bob["alter"] + 1
mein_bester_freund["alter"] = mein_bester_freund["alter"] + 1  # selber Effekt: Bob ist ein Jahr älter


print(mein_bester_freund)
mein_bester_freund = fred
print(annas_bester_freund) # immer noch Bob
```

Themen des Tages

Variable vs Objekte

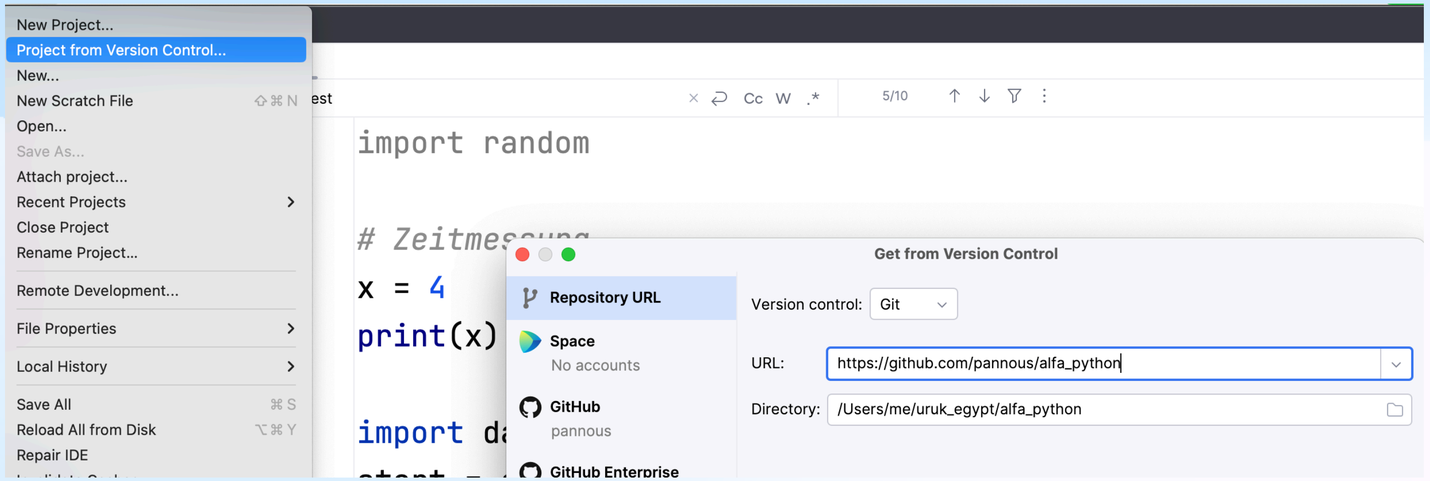
Beim (um)setzen einer Variable auf ein (neues) Objekt wird das Objekt nicht verändert!

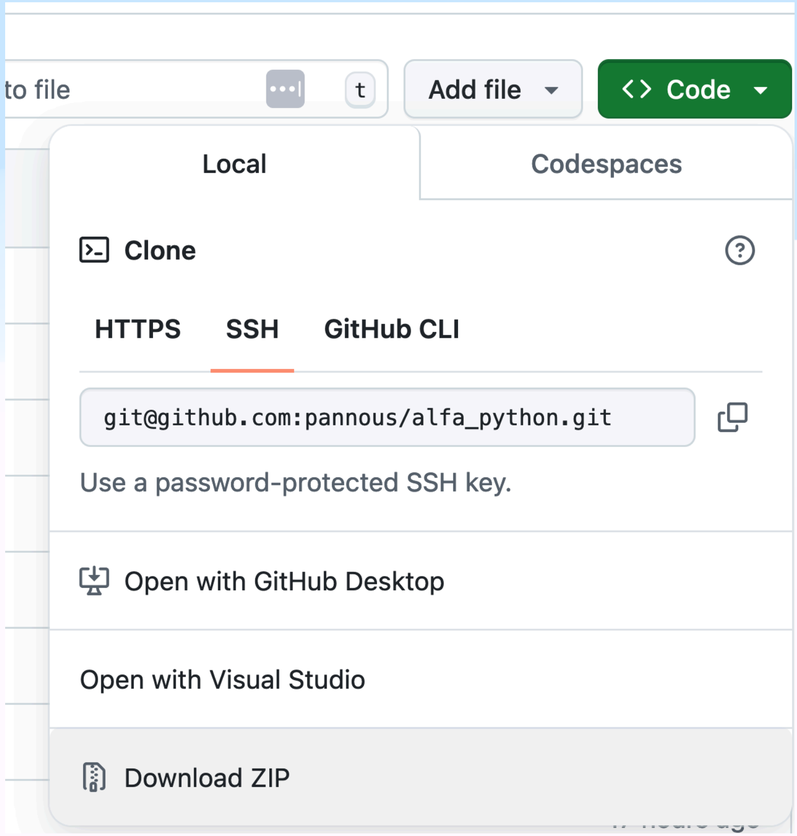
```
tupel1 = (1, 2, 3)
tupel2 = (4, 5, 6)
mein_tupel = tupel1
print(mein_tupel)
mein_tupel = tupel2 # altes Objekt wird nicht geändert, wird aber nicht mehr 'referenziert'

Tupel bleiben unveränderlich, auch wenn die Variable sich ändern kann

mein_tupel.insert() etc gibts nicht Tupel selbst ist unveränderlich,
wenn dann würde eine neue Kopie zurückkommen
```

GitHub

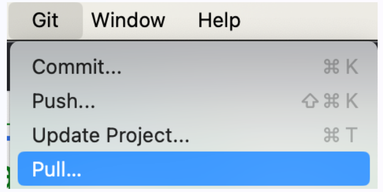
Freiwillig GitHub Account anlegen
zum Herunterladen meiner Dateien und Lösung

https://github.com/pannous/alfa_python





aktualisieren über git pull



list comprehension

In Python bietet eine List Comprehension eine prägnante Möglichkeit, **Listen zu erstellen.**
Sie besteht aus **eckigen Klammern**, die einen **Ausdruck** enthalten, gefolgt von einer **for**-Klausel, und danach null oder mehr **for**- oder **if**-Klauseln.
Die Ausdrücke können alles Mögliche sein, was bedeutet, dass man alle Arten von Objekten in Listen einfügen kann.

```
[expression(of item) for item in iterable]  Grundsyntax
```

```
quadrate = [(x**2) for x in range(10)]
```

```
in Mathe   {x^2 | x ∈ [0,10)}
```

```
[expression(of item) for item in iterable if condition]  erweitert mit Bedingung
```

```
gerade_zahlen = [x for x in range(20) if x % 2 == 0] # Filter
```

set comprehension

List comprehension ≈ set comprehension
Erzeugen von Listen ≈ Erzeugen von Mengen

```
[x**2 for x in range(-10,10)]
[100, 81, 64, 49, 36, 25, 16, 9, 4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
{x**2 for x in range(-10,10)}  # wie Menge in Mathe   {x^2 | x ∈ [-10,10)}
{64, 1, 0, 100, 36, 4, 9, 16, 81, 49, 25}
```

⚠️ Wiederholungen und Ordnung fallen weg!

```
Selbe Syntax auch für Tupel nur mit anderen Klammern: [] {} ()
```

```
(x**2 for x in range(-10,10))  # Optimiertes objekt für for schleifen
<generator object <genexpr> at 0x11e1dd630>

tuple(x**2 for x in range(0,10))
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)   unveränderlich
```

list comprehension

```
[expression for item in iterable]
```

expression ist beliebiger Ausdruck ≈ Berechnung, z.B. auch Paare als Tupel, strings, ...

Beispiele:

```
quadrate_paare = [(x, x**2) for x in range(5)] # als Tupel
quadrate_strings = [f"Hallo {x}" for x in range(5)]
quadrate_kombiniert = [(x,f"quadriert: {x*x}") for x in range(5)]

großbuchstaben = [buchstabe for buchstabe in "Hallo Welt!" if buchstabe.isupper()]
# [H,W]
```

list comprehension

Erweiterbar mit Bedingung pro Element (item)

```
[expression for item in iterable if condition]
```

```
in Mathe   {x^2 | x ∈ [0,20) ∧ x%2=0 }   # ∧ and / if
```

```
gerade_zahlen = [x for x in range(20) if x % 2 == 0] # Filter
ungerade_zahlen = [x for x in range(20) if x % 2] # die mit Rest
```

Conditionsvariable (hier x) muss natürlich mit Schleifenvariable (hier x) zusammenpassen.

```
großbuchstaben = [buchstabe for buchstabe in "Hallo Welt!" if buchstabe.isupper()]  # [H,W]
alle_kleinbuchstaben = {buchstabe for buchstabe in "Hallo Welt!" if buchstabe.islower()} #ohne Duplikate wegen set
```

dict comprehension

Wie list comprehension nur mit {key:value … } syntax:

```
gerade_quadrate = {x: x**2 for x in range(10) if x % 2 == 0}
```

dict comprehension wie list comprehension:

```
letters_to_count = ["A","B","C"]

letter_counts = {letter: 0 for letter in letters_to_count} # initialisiere Zählungen mit 0

for letter in "HALLO":
  letter_counts[letter]+=1
```

Listenerzeugung

Spezialsyntax list comprehension

ähnlich Mengen Definition früher in Mathe: $\mathbb{R}_+ := \{x \mid x \in \mathbb{R} \text{ und } x>0\}$

Mit Listenerzeugung kann man Elemente einer anderen Liste filtern und umwandeln.

```
[x*x for x in range(10)] # Quadrate [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[x for x in range(10) if x > 5]  # [6,7,8,9]
```

Generelle Form:
[Ausdruck **for** item **in** iterable/list (**if** Bedingung)]

Ausdruck kann auch beliebige Berechnungen durchführen:

```
def quadrat(x): return x*x
[quadrat(x) for x in range(10)] # Quadrate [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

weiteres Beispiel

```
[x*x for x in range(10) if x % 2 == 0] # alle geraden Zahlen von 0 bis 8 quadriert
```

Listenerzeugung

Spezialsyntax list comprehension

ähnlich Mengen Definition früher in Mathe: $\mathbb{R}_+ := \{x \mid x \in \mathbb{R} \land x>0\}$

Mit Listenerzeugung kann man Elemente einer anderen Liste filtern und umwandeln.

- *UMWANDELN und ERZEUGEN*

```
[x*x for x in range(10)] # Quadrate [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

•Filtern: filter(lambda x:x>5, range(10))

```
[x for x in range(10) if x > 5]  # [6,7,8,9]
```

Generelle Form:

```
[Ausdruck for item in iterable/list (if Bedingung)]
```

- Kombiniert:

```
[x*x for x in range(10) if x % 2 == 0] # alle geraden Zahlen von 0 bis 8 quadriert
```

Listenerzeugung

Spezialsyntax list comprehension

ähnlich Mengen Definition früher in Mathe: $\mathbb{R}_+ := \{x \mid x \in \mathbb{R} \text{ und } x>0\}$

Ausdruck kann auch **beliebige Berechnungen** durchführen:

```
def quadrat(x):
  return x*x

liste = [1, 2, 3, 4, 5]

quadrate = [quadrat(x) for x in liste] # [1, 4, 9, 16, 25]
```

list comprehensions werden schnell unübersichtlich, dafür ist es dann elegant die Berechnungsformel (hier quadrat) in einer eigenen Funktion zu definieren.

Listenerzeugung ≈ map

List comprehension (Einbeziehung/Einschliessung) ist ähnlich zu map Funktionen und Schleifen

```
def f(x): return x*x

[f(x) for x in liste]

≈

map(f, liste)

≈

results = []
for x in liste:
  results.append(f(x))
```

Schleifen sind offensichtlich weniger elegant, map und [for … in] ist *Geschmacksfrage*

Listenerzeugung

Erzeugung komplizierterer Elemente:

```
# List comprehension kann auch verwendet werden, um eine Liste von Tupeln zu erstellen:
[(x, x*x) for x in range(10)] # 10
```

Der Ausdruck kann **beliebig komplex** sein, unter anderem wieder neue Comprehension:

```
[ [ x-i for i in (1,2,3)] for x in [1, 2, 3, 4]] # 4

[ [0,-1,-2], [1,0,-1], [2,1,0], [3,2,1] ]
```

Tipp: einfach *überspringen*, erst am ENDE zu den schweren zurück!!

Verschachtelte Listenerzeugung

```
# Verschachtelte List Comprehension

listen = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

flache_liste = [element   for unterliste in listen  for element in unterliste]

print(flache_liste) # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Achtung die erste Schleife ist die Äussere,
# Achtung die innere Schleife ist die letzte Schleife
```

Verschachtelte Listenerzeugung ist *für meinen Geschmack* weniger elegant als:

```
alle = []
for unterliste in listen:
   for element in unterliste:
      alle.append(element)
```

Dict comprehension

Dictionary Erzeugung

Analog Listenerzeugung nur mit geschweiften Klammern und unserem Key:Value Doppelpunkt:

```
{str(x):x*x for x in range(10)}
```

Generelle Syntax:

```
{ key_ausdruck:value_ausdruck for item in list }
```

z.B.

```
[(x,x**2) for x in range(-1,10)]
{x:x**2 for x in range(-1,10)}
{-1: 1, 0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Auch wieder mit optionaler **Filter Bedingung:**

```
{ key_ausdruck:value_ausdruck for item in list if Bedingung}
{x:x**2 for x in range(-1,10) if x%2}  # nur ungerade
{-1: 1, 1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Beispiele Anwendungen

Dictionary Erzeugung

```
{x:x**2 for x in range(-1,10)}
{-1: 1, 0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

satz = "Die Zahlentheorie ist die Königin der Mathematik"
{wort: len(wort) for wort in satz.split()} #
{'Die': 3, 'Zahlentheorie': 13, 'ist': 3, 'die': 3, 'Königin': 7, 'der': 3, 'Mathematik': 10}


# Filtern (ohne Einträge zu bearbeiten)
noten = {"Anna": 1.3, "Ben": 4.7, "Clara": 2.0, "David": 5.0}
{name: note for name, note in noten.items() if note <= 4.0}
{name: noten[name] for name in noten if noten[name] <= 4.0}
# {'Anna': 1.3, 'Clara': 2.0}

def name(unicode): return unicodedata.name(chr(unicode),'')
{nr:(chr(nr),name(nr)) for nr in range(170000) if "horse" in name(nr).lower()}
12002 马 CJK RADICAL C-SIMPLIFIED HORSE
128014 🐎 HORSE
128052 🐴 HORSE FACE
```

Beispiele Anwendungen

Dictionary Erzeugung, häufige Anwendung

"CACHE!"
Oft ist die Berechnung von einem Wert teuer, dann kann man den in einem Dictionary zwischen**speichern,** um es nie wieder neu berechnen/abrufen zu müssen.

```
WOERTER = ["horse", "dog", "cat", "house", "tree"]
woerterbuch = {word: translate(word) for word in WOERTER} # 🥰
```

translate kann hier ein zeitlich oder finanziell teurer Aufruf zu einer Webseite sein...

⇒ woerterbuch lokal speichern!

Aufgaben

list comprehension:
strings = ["hallo","welt","ok"]
a) erzeuge eine Liste mit der Länge von allen strings `[len(s) for s in strings]`
b) erzeuge eine Liste mit dem letzten Buchstaben von allen strings `[s[-1] for s in strings]`
c) erzeuge eine Liste mit der Anzahl Vokale von allen strings [len(vokale(s)) for s in strings]
d) erzeuge eine Liste mit der Anzahl Konsonanten von allen strings mit Länge>2

dict comprehension
e) erzeuge ein dict mit der Länge von allen strings pro Wort `{wort:len(wort) for wort in strings}`
f) erzeuge ein dict mit dem letzten Buchstaben von allen strings
g) erzeuge ein dict mit der Anzahl Vokale von allen strings
h) erzeuge ein dict mit der Anzahl Konsonanten von allen strings mit Länge>2

set comprehension
i) kurzer Einzeiler zum sehen, welche **Buchstaben**/Zeichen in einem Text vorkommen `{c for c in text}`
`{c for c in text if c.isascii()}` `isalpha()`

Tests:

Wähle eine Funktion der letzten Tage und prüfe mit einem Test, ob diese tut was sie soll.

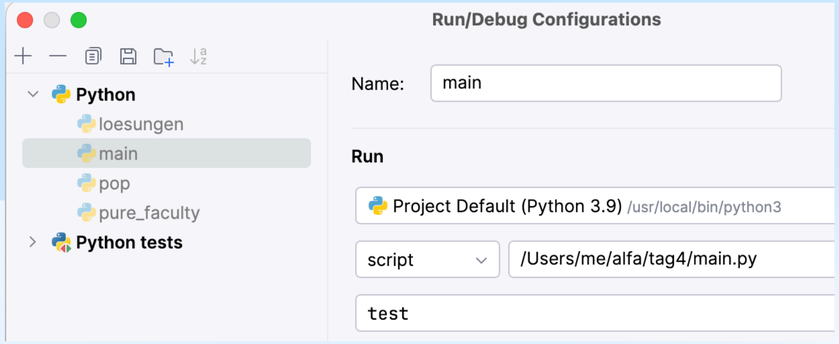
Gehe alle Tage durch,
dort wo die Dateien syntaktisch ok sind:
Erstelle Tests für alle Funktionen welche Werte zurück liefern.
Prüfe ob die Werte für bestimmte Argumente mit den Erwartungen übereinstimmen

Übergabe von Argumenten an Programm

```
import os # system commando ausführen
cmd = "ls -l"
ok=os.system(cmd) # führt ein Kommando aus und gibt den Rückgabewert zurück
```

Übergabe von Argumenten an Programm

```
Speichern unseres kleinen Programmes main.py :
#!/usr/bin/env python3
import sys  # system modul

# argV argumente-vector = parameter list
print("alle Argumente:", sys.argv) # index 0 ist der Name des Skripts
if len(sys.argv) > 1:
  print("Zusatzargument 1", sys.argv[1])
```



oder über System Kommandozeile
cmd.exe powershell.exe Terminal.app bash

```
~/alfa/tag4/ python3 main.py ok
['main.py', 'ok']
Zusatzargument 1 ok
```

Vertiefung Module

```
Zugriff auf eigenen oder externen Erweiterungscode

# Datei: mein_modul.py
print("Modul wird geladen!") # ⚠️ Seiteneffekte DRINGEND MEIDEN!
xxx = 42
def meine_modul_funktion():
  print("Modul Funktion wurde aufgerufen!")

# andere Datei:
import mein_modul
mein_modul.meine_modul_funktion()
print(mein_modul.xxx)
# Variablen aus mein_modul.py stehen per modul-Präfix zur Verfügung
```

Vertiefung Module

```
Komfortabler Zugriff

# man kann sich dieses modul-Präfix auch sparen (bei häufiger Verwendung…)

# spezielle Funktionen und Variablen eines Moduls direkt zugänglich machen:
from mein_modul import meine_modul_funktion, xxx
print(xxx)
meine_modul_funktion()

# generelle Syntax:
# from modul_name import funktion_names, variable_names


from modul_name import *  # ALLES importieren schlechter Stil
# cluttered namespace ≈ vollgestopfter Namensraum

# Stichwort Encapsulation # Abkapselung von Funktionalitäten
```

Vertiefung Module

Verzeichnis als modul

/modul_name/__init__.py

steht dann als Modul zur Verfügung:

import modul_name

Binäre Darstellung

Frage: wer weiss was **0x01A** als Zahl ist?

Binäre Darstellung

Vorüberlegungen:

Welche Zahl ist universell? 1, 2 , paar, viele?

Erste Zählweise der Menschheit = **unary (uno = 1)**
7 = || ||||| Striche zählen

Zweite Zählweise der Menschheit:
17 = X|||||||| Sonderzeichen für 10, 60, 100, 600…

MDCXXI = 1621 MMMCCXXXXII * MMMCCXXXXII abacus

Dezimale Darstellung : Basis 10 = {0,1,2,3,4,5,6,7,8,9};

Großer Durchbruch:

Positionsabhängiges Zählen:

707 = 100***7** + 10*0 + 7

123 = 1***10*10** + 2***10** + 3***1**

Folge von Ziffern
abcd = a*10³ + b*10² + c*10¹ +d***10⁰** "**basis** 10"

Binäre Darstellung … Basis 2 = {0,1}

Kleinste Recheneinheit (im Computer)

Lichtschalter

1 Lichtschalter: an + aus ☐ , ■ 2 "Zustände"

2 Lichtschalter: ☐ ☐ , ☐ ■ , ■ ☐ , ■ ■ 4 "Zustände"

3 Lichtschalter:

☐ ☐ ☐ , ☐ ☐ ■ , ☐ ■ ☐ , ☐ ■ ■

■ ☐ ☐ , ■ ☐ ■ , ■ ■ ☐ , ■ ■ ■ 8 Zustände

4 Lichtschalter: 16 Zustände …

n Lichtschalter: **2**n** 2 hoch n Zustände …

32 bit ≈ 4 Milliarde / 64 bit ≈ Fantastilliarde 10^20

Morsen: Basis 3;)

Binäre Darstellung … Basis 4 = {0,1,2,3}={A,G,C,T}

Code mit 4 'Einträgen' **DNA/RNA**
A,G,C,T 4 Basenpaare

Human DNA: 3,1 Billion

AGCTCT (4*4*4* ... Auswahlmöglichkeiten)
4^(3*10^9) verschiedene Kombinationen bei Mensch

Virus DNA: 1.700 - 2 Mio. Venter: ~5.386 bp
Phagen Antibiotika

Binäre Darstellung … Basis 2 = {0,1}

Ganze Zahlen lassen sich als folge von 0 und 1 darstellen. Grundidee: jede 1 steht für eine Potenz von 2:

```
0b000 0 = 0   # binary bit
0b000 1 = 1 = 2⁰  letzte Ziffer immer kleinster Wert
0b00 1 0 = 2 = 2¹
0b0 1 00 = 4 = 2²
0b1000 = 8 = 2³
0b1001 = 9 = 2³ + 2⁰ = 8 + 1
0b1010 = A = 2³ + 2¹ = 8 + 2 = 10
inklusive 0 hat man mit 4 bit 16 Zahlen > 10 Zahlen

1111 decimal = 1000 + 100 + 10 + 1
0b1111 = 2³ + 2² + 2¹ + 2⁰  =
0b1111 = 8  + 4  + 2  + 1 = 15      4 bits 'nibble'

byte 0b11111111 8 bit 256 Zahlen (2**8) > 10*10 alle Zeichen

32 bit ≈ 4 Milliarde / 64 bit ≈ Fantastilliarde 10^20
```

Binäre Darstellung … Basis 2 = {0,1}

Ganze Zahlen lassen sich als folge von 0 und 1 darstellen. Grundidee: jede 1 steht für eine Potenz von 2:

```
0b0000 = 0   # 0b binär
0b0001 = 1 = 2⁰
0b0010 = 2 = 2¹
0b0100 = 4 = 2²
0b1000 = 8 = 2³
0b1010 = 10

1111 decimal = 1000 + 100 + 10 + 1
0b1111 = 1*2³ + 1*2² + 1*2¹ + 1*2⁰  =
0b1111 =   8  +   4  +   2  +   1 = 15

0b0011 =          2 + 1  = 3
0b0110 =      4 + 2 + 0  = 6
0b0111 = 0 + 4 + 2 + 1  = 7
0b0101 = 0 + 4 + 0 + 1  = 5
```

Binäre Darstellung … Basis 2 = {0,1} => neue Basis 4 bit = *hex*

```
0b0000 = 0   # binary bit
0b0001 = 1 = 2⁰  letzte Ziffer immer kleinster Wert
0b0010 = 2 = 2¹
0b0011 = 3 = 2¹ + 2⁰
0b0100 = 4 = 2²
0b0101 = 5 = 2² + 2⁰ = 4 + 1
0b0110 = 6 = 2² + 2¹ = 4 + 2
0b0111 = 7 = 2² + 2¹ = 4 + 2 + 1
0b1000 = 8 = 2³
0b1001 = 9 = 2³ + 2⁰ = 8 + 1
0b1010 = A = 2³ + 2¹ = 8 + 2 = 10
0b1011 = B = 2³ + 2¹ + 2⁰ = 8 + 2 + 1 = 11
0b1100 = C = 12
0b1101 = D = 13
0b1110 = E = 14
0b1111 = F = ... = 15
```

Binäre Darstellung … Basis 2 = {0,1}

0b0011 = 0*2³ + 0*2² + 1*2¹ + 1*2⁰ =
0b011 = 0 + 0 + 2 + 1 = 3

0b1011 = 1*2³ + 0*2² + 1*2¹ + 1*2⁰ =
0b1011 = 8 + 0 + 2 + 1 = **11**

0b1101 = 1*2³ + 1*2² + 0*2¹ + 1*2⁰ =
0b1011 = 8 + 4 + 0 + 1 = **13**

Binäre Darstellung … Basis 2 = {0,1}

Negative Zahlen

byte 0**b1**1111111 **8 bit** 256 Zahlen
unsigned **byte** 2^^8 = 256 Zahlen
signed byte KONVENTION**:**
alles was mit 0b0 anfängt ist **positiv**
die andere Hälfte wird negativ (Hälfte **je nach bit zahl)**
bei byte**:** 256 Zahlen alles über 128 ist negativ definiert

Aber wir zählen **rückwärts**
signed byte x = -1 // 0b**1**1111111 per Definition
signed byte x = -2 // 0b**1**1111110
signed byte x = -3 // 0b**1**1111101
... Formel: -256 + x je nach Bitzahl
... Formel: -2^Bitzahl + x je nach Bitzahl
Python: beliebig lange Zahlen, daher - OHNE Konvention Also -1 = -0b1

hexadezimal identisch
signed byte x = -1 // 0xFF

Hexadezimale Darstellung

In der Hexadezimalen Darstellung werden vier bit (der binären Darstellung) zusammengefasst:

0b0001 = 1 = 2^0
0b0010 = 2 = 2^1
0b0100 = 4 = 2^2
0b1000 = 8 = 2^3

16 Verschiedene Zahlen die man mit **4 bit** darstellen kann
0,1,2,3,4,5,6,7,8,9,**A**,B,C,D,E,F **Hexadezimale** Basis (**Hex**adezim griechisch: **16** decim+**hexa Dezimal plus 6:abcdef**)

0b0000 = **0x0**0 = 0
0b0001 = 0x1 = 1

0b1001 = 0x9 = 9
0b1010 = 0x**A** = 10 # präfix 0x heisst in python/überall: HEXADEZIMALZAHL
0b1011 = 0xB = 11
0b1100 = 0xC = 12
0b1101 = 0xD = 13
0b1110 = 0xE = 14
0b1111 = 0x**F** = 15 ! 16 - 1 für die Null
0b0001.0000 = 0x10 = 16
0b0001.0001 = 0x11 = 17 = 16+1
0b0001.0001 = 0x12 = 18 = 16+2
0b1111.1111 = 0x**FF** = 2^8-1 = **255** = 256 - 1 = 16*16 -1 ! das größte was wir mit 8 bit darstellen können
0…15 0x0…xF Ziffer nennt man "**Nibble**" "hex"

Motivation **byte** = 8 bit 0b1111_1111 zwei hex digits 00...FF

0x55 == 5*16 + 5*1

Hexadezimale Darstellung

In der Hexadezimalen Darstellung werden vier bit (der binären Darstellung) zusammengefasst:

16 Verschiedene Zahlen die man mit **4 bit** darstellen kann
0,1,2,3,4,5,6,7,8,9,**A**,B,C,D,E,F **Hexadezimale** Basis (**Hex**adezim griechisch: **16** decim+**hexa Dezimal plus 6:abcdef**)

0b0000 = **0x0**0 = 0
0b0001 = 0x1 = 1
0b1001 = 0x9 = 9
0b1010 = 0x**A** = **10** # präfix 0x heisst in python/überall: HEXADEZIMALZAHL
0b1011 = 0xB = 11
0b1100 = 0xC = 12
0b1101 = 0xD = 13
0b1110 = 0xE = 14
0b1111 = 0x**F** = 15 ! 16 - 1 für die Null

22 = 2*10^1 + 2*10^0 = 20 + 2 Achtung: Bedeutung ändert sich durch 0x hex Präfix:
0x**2**2 = 2***16**^1 + 2*16^0 = 32 + 2 = **34**
0x**AF = 10*16**^1 + 15*16^0 = 160 + 15

Anwendungen Hexadezimale

Anwendungen:

Grundverständnis Computer, insbesondere Speicher(Adressen, layout: Blöcke)

Universell

Dateien (binär) special unicode

ASCII Unicode '\U0001F63C' == '😼'

Bilder Farben (binary bitmaps gruppiert) #RRGGBB z.B. #FF5733 (rot=255, grün=87, blau=51) #FF5733

Hacken (moderne Bedeutung: umbiegen / anpassen)

Assembly

```
~/alfa/alfa_python/tag10/ hd ./people.db
00000000  53 51 4c 69 74 65 20 66  6f 72 6d 61 74 20 33 00  |SQLite format 3.|
00000010  10 00 01 01 00 40 20 20  00 00 00 02 00 00 00 03  |.....@  ........|
00000020  00 00 00 00 00 00 00 00  00 00 00 02 00 00 00 04  |................|
```

Anwendungen Hexadezimale

ASCII / Unicode '\U0001F63C' == '😼'

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	**16**	**10**	DLE	32	**20**		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	**0A**	**LF**	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	**0D**	**CR**	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Oktale Darstellung

In der Oktale Darstellung werden drei bit (der binären Darstellung) zusammengefasst:

0b0001 = 1 = 2^0
0b0010 = 2 = 2^1
0b0100 = 4 = 2^2

0b0000 = **0o**0 = 0
0b0001 = 0o1 = 1
0b0111 = 0o7 = 7
0b1000 = 0o**10** = 8

Folge von Ziffern
abcd = a*$\mathbf{8^3}$ + b*$\mathbf{8^2}$ + c*$\mathbf{8^1}$ +d*$\mathbf{8^0}$ "**basis** 10"

0,1,2,3,4,5,6,7 damit ist '7' die größte Ziffer

0b1001 = 0o11 = 9

Binäre Darstellung … Basis 2 = {0,1}

Ganze Zahlen lassen sich als folge von 0 und 1 darstellen. Grundidee: jede 1 steht für eine Potenz von 2:

0b0001 = 1 = 2^0
0b0010 = 2 = 2^1
0b0100 = 4 = 2^2
0b1000 = 8 = 2^3

0b1111 = $2^0 + 2^1 + 2^2 + 2^3$ =
0b1111 = 1 + 2 + 4 + 8 = 15

0b0111 = 4 + 2 + 1 = 7
0b0101 = 4 + 0 + 1 = 5
0b0110 = 4 + 2 + 0 = 6
0b1001 = 9
0b1100 = 2^3 + 2^2 = 8 + 4 = 12
Anzahl der darstellbaren Zahlen 2^(Anzahl der Ziffern)

Hexadezimale Darstellung

In der Hexadezimalen Darstellung werden vier bit (der binären Darstellung) zusammengefasst:

```
0b0001 = 1 = 2⁰    # 0b = binär, basis 2, nur 0 oder 1 als 'Ziffern'
0b0010 = 2 = 2¹
0b0100 = 4 = 2²
0b1000 = 8 = 2³

0b0001 = 0x1 = 1
0b1001 = 0x9 = 9
0b1010 = 0xA = 10 !
0b1111 = 0xF = 15 !
```

Warum? So dass lange Zahlen übersichtlich als Gruppen von bytes (2 hex) dargestellt werden können!

```
0xFF = größtes (unsigned) byte >>> 0xFF # 255 plus die 0 => 256 Zahlen
0xFFFF = größtes (unsigned) short
0xFFFFFFFF = größtes (unsigned) int  ( 32 bit = 8 nibbles ) 4 Mrd SQL
0xFFFFFFFFFFFFFFFF = größtes (unsigned) long long int ( 64 bit = 16 nibbles )
```

Hexadezimale Darstellung

Anwendungen:

Dateisystem:

hexedit kann binäre Dateien kompakt darstellen

```
00000000  4a 41 56 41 20 50 52 4f  46 49 4c 45 20 31 2e 30  |JAVA PROFILE 1.0|
00000010  2e 32 00 00 00 00 08 00  00 01 9e 16 81 12 6d 01  |.2............m.|
```

Unicode

```
>>> "\U0001F63C"
'😼'
>>> "\u08aa"
'ઍ'
```

Überall

Computer ist halt so aufgebaut (alles binär, wird zu bytes gruppiert)

Binäre bitwise Operatoren

0 = False
1 = True

and, or, not

True and False <==> 1 & 0

0,1 bit
Verallgemeinerung von logischen Operatoren
auf alle **bits** in einer Zahl

Binäre bitwise Operatoren

Verallgemeinerung von logischen Operatoren
auf alle bits in einer Zahl

and => & gemeinsame 1 in beiden Positionen
or => | **eine 1 in einer der beiden Positionen**

- & **Bitweise** UND <> Logisches und &&
- | Bitweise ODER
- ^ Bitweise XOR
- ~ Bitweise NOT flip 0=>1 1=>0
- << (Linksschieben) verdoppeln "füge **0** rechts an" 0b00101 << 1 => 0b01010
- **>>** (Rechtsschieben) halbieren "schneide letztes **bit** ab" a >> n == a/2**n

mit **bin()** kann ich Zahlen als binär anzeigen lassen

```
>>> bin(0b111001101 >> 1 )  '0b11100110'
```

Binäre bitwise Operatoren

```
Logische Operatoren op(bool a, bool b) sind das selbe wie bit
Operatoren

False and False => false
False and True  => false
True  and False => false
True  and True  => true

false ≈ 0 , true ≈ 1

0 and 0 => 0
0 and 1 => 0
1 and 0 => 0
1 and 1 => 1
```

Binäre bitwise Operatoren

```
Inklusives oder ( das eine, das andere, oder beide )

false or false => false
false or true  => true
true  or false => true
true  or true  => true

false ≈ 0 , true ≈ 1

0 or 0 => 0   r00 a
0 or 1 => 1   r01 b
1 or 0 => 1   r10 c
1 or 1 => 1   r11 d
or operator in diesem System ist dann [0 1 1 1]
```

Binäre bitwise Operatoren

Exklusives oder (eins von beiden, aber nicht beide)

```
false xor false => false
false xor true  => true
true  xor false => true
true  xor true  => false
```

```
false ≈ 0 , true ≈ 1
```

```
0 xor 0 => 0
0 xor 1 => 1
1 xor 0 => 1
1 xor 1 => 0
```

Binäre bitwise Operatoren

Inklusives oder (das eine, das andere, oder beide)

```
false or false => false
false or true  => true
true  or false => true
true  or true  => true
```

false ≈ 0 , true ≈ 1

```
0 or 0 => 0
0 or 1 => 1
1 or 0 => 1
1 or 1 => 1
```

Exklusives oder (definitiv eins von beiden, aber nicht beide)

```
false xor false => false
false xor true  => true
true  xor false => true
true  xor true  => false
```

false ≈ 0 , true ≈ 1

```
0 xor 0 => 0
0 xor 1 => 1
1 xor 0 => 1
1 xor 1 => 0
```

xor == ≠ 'xor prüft auf ungleichheit' x or y == x ≠ y

x xor y = x or y and not x and y

Bitweise Operatoren

- & Bitweise UND <> Logisches und &&
- | Bitweise ODER
- ^ Bitweise XOR
- ~ Bitweise NOT
- << (Linksschieben) verdoppeln 0b00101 << 1 => 0b01010
- >> (Rechtsschieben) halbieren

Abhängig vom Datentyp!!

unsigned byte x= 3; // 0b00000011; // 2+1
unsigned byte y=~x; // 0b11111100; // 255-x bitweise Negierung
0b11111100 = 4 + 8 + 16 + 32 + 64 + 128 = 252
unsigned byte z=!x; // 0 absolute Negierung

Bitweise Operatoren

```
Bitweises 'und'
0b1110 &
0b1101 ==
0b1100
```

```
Anwendung Masken (Bild-) … …
```

```
Bitweises 'und'
0b1110 &
0b1101 ==
0b1100
```

Bitweise Operatoren

```
bits umkehren über xor 111...

Bitweises 'xor'
0b1110101 ^
0b1111111 ==
0b0001010
```

Bitweise Operatoren

```
Bitweise 'negation'
>>> bin(~0b1010101111000111000)
       '-0b1010101111000111001'
```

```
bin(x ^ ((1 << n) - 1))
```

```
Anwendung Masken (Bild-) … …
```

```
Bitweises 'und'
0b1110 &
0b1101 ==
0b1100
```

Malen mit bits (heute noch oft)

```
00000000000000000000000000
00000000111111000000000000
00000000100001000000000000
00000000111111000000000000
00000000100001000000000000
00000000100001000000000000
00000000000000000000000000
```

Gib mir ein "A"! bitmap bmp

- Übungen:

Berechne auf Typ byte (unsigned char):

0b10010011 & 0b10011100 = 0b10010000 = 128 + 16 = 144

0b10010011 | 0b10011100 = 0b10011111 = 144 + 15 = 159

0b10010011 ^ 0b10011100 = 0b00001111 = 15

Unterschied zu logischen Operationen

0b10010011 && 0b10011100 == 0b00000001 == 1

0b10010011 || 0b10011100 == 0b00000001 == 1

0b10010011 ^ 0b10011100 == 0b00000001 == 1

^^ gibt es in c++ nicht

Binäre bitwise Operatoren

```
Logische Operatoren op(bool x, bool y) zB and
def op(bool x, bool y): return …

x    y => Resultat
0 op 0 => r00 (0 oder 1)
0 op 1 => r01 (0 oder 1)
1 op 0 => r10 (0 oder 1)
1 op 1 => r11 (0 oder 1)

# x, y ein bit/bool Rückgabe ein bit/bool
def op(x, y):
 if(x==0 and y==0): return r00;
 if(x==0 and y==1): return r01;
 if(x==1 and y==0): return r10;
 if(x==1 and y==1): return r11;


4 Ausgaben => 2^4 Kombinationen = 16 Operatoren
zB op="and" r00,r01,r10=0 r11=1
0 and 0 => 0
0 and 1 => 0
1 and 0 => 0
1 and 1 => 1
```

Binäre bitwise Operatoren

```
16 Operatoren

x    y => Resultat
0 op 0 => r00 = a (0 oder 1)
0 op 1 => r01 = b (0 oder 1)
1 op 0 => r10 = c (0 oder 1)
1 op 1 => r11 = d (0 oder 1)

op(x,y)=const 0,  a,b,c,d=0 trivialer Operator "Widerspruch, false!"
op(x,y)=not(y)    a=1 b=0 c=1 d=0
op(x,y)=not(x)    a=1 b=1 c=0 d=0
op(x,y)=and(x,y)  a=0 b=0 c=0 d=1
op(x,y)=or(x,y)   a=0 b=1 c=1 d=1
op(x,y)=nand(x,y) a=1 b=1 c=1 d=0 // not and  "Funktional vollständig"
op(x,y)=nor(x,y)  a=1 b=0 c=0 d=0 // -''-
op(x,y)=x         a=0 b=0 c=1 d=1  x-Identität
op(x,y)=y         a=0 b=1 c=0 d=1  y-Identität
op(x,y)=const 1   a=1 b=1 c=1 d=1  Tautologie "Wahr"
op(x,y)=xor       a=0 b=1 c=1 d=0  exklusives Oder "entweder oder"  ^  / Nichtgleich neq != ≠
op(x,y)=eq        a=1 b=0 c=0 d=1  equality = Gleichheit, Äquivalenz negiertes xor == xnor '=='
op(x,y)=imp       a=1 b=0 c=1 d=1  Implikation  =>  aus x folgt y, aber aus y folgt nicht automatisch x
op(x,y)=¬imp      a=0 b=1 c=0 d=0  Reverse-Implikation Nicht-Implikation
op(x,y)=xxx    zwei fehlen noch
```

Binäre bitwise Operatoren

Die wichtigsten dieser **16** binäre bitwise Operatoren können auf Integrale typen übertragen werden?
Warum? Weil int intern als Folge von 0^{en} und 1^{en} angesehen werden können:

```
int i = 256 + 8; // 0b0100001000
int j = 256 + 4; // 0b0100000100
int k = i & j;   // 0b0100000000 => 256 !
7 & 5 0b111 & 0b101 =0b101
Als bool 1≈True 0≈False  1&0 vs logisches and
```

Binary Nachkommastellen

Floats (Fließpunktzahlen) kann man auch binär darstellen.

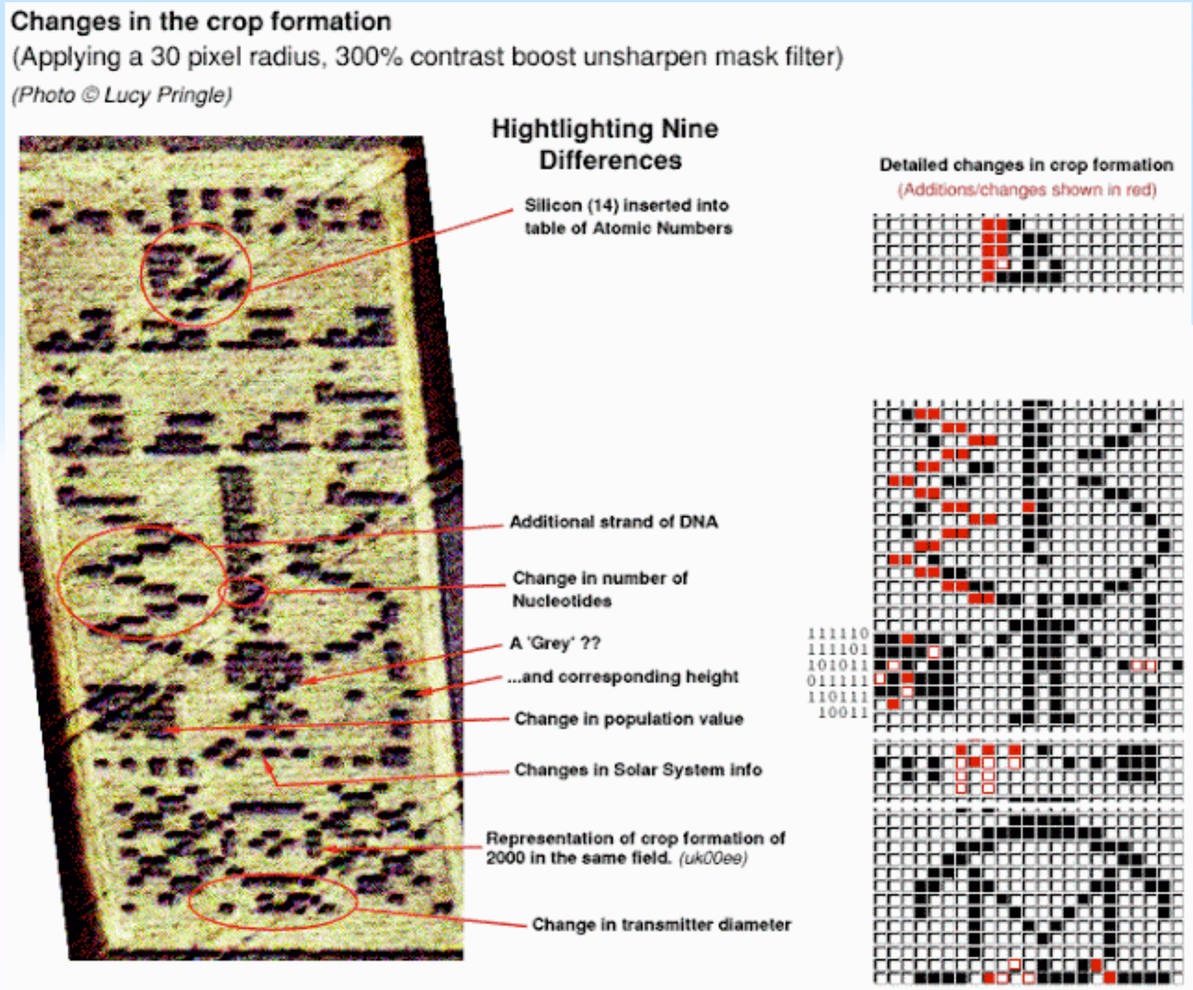
Dazu kann man die Nachkommastellen als Belegungen von 1/2 1/4 1/8 … ansehen:

```
0b0001 = 1 = 2⁰
0b0010 = 2 = 2¹
0b0100 = 4 = 2²
0b1000 = 8 = 2³

0b0.100 = 1/2 = 2⁻¹
0b0.010 = 1/4 = 2⁻²
0b0.001 = 1/8 = 2⁻³
```

$3.125 = 3 + 1*10^{-1} + 2*10^{-2} + 3*10^{-3}$

z.B.: 3.125 = 3⅛ = 2+1+1/8 = 11.001

```
print(decimal_to_binary(math.pi, 100))
Alian_Signal = 10110011100011110000111110000000
π = 11001001000011111101101010100010001000010110100

π = 11.001001000011111101101010100010001000010110100
Vielleicht Signale mit 4 Stärken?
```



Binary Floats

Da Komputer keinen natürlichen "Punkt" für Nachkommastellen haben, muss die echte Darstellung (ähnlich negativer Zahlen) einer **Konvention** folgen:

π = 11.00100100001111110110101010001000100010110100

IEEE 754 binary**32** format:

π ≈ + 2^1 * 1.00100100001111111011011 (rounded)

π as float32 : **0**1000000 01001001 00001111 11011011

π as float32 : **0** : positive Sign bit

π as float32 : 10000000 : **Exponent** (**8 bits**) => **128** - 127 = 1 (IEEE 754 uses a bias of 127) => 2^1 2^128

π as float32 : 10010010000111111011011 (implicitly with a leading **1**) Mantisse?

0 = 000000000000

Ausdruck versus Anweisung

Ausdruck (Expression) : Wert oder Code Fragment oder Berechnung welches Wert Erzeugt
Anweisung (Statement) : Komplette Code Zeile welche eine Aktion Ausführt

```
sum = 3 + 1; # '3 + 1' ist ein Ausdruck, das ganze ist ein Statement ( Zuweisung )
```

Arithmetic Expression:

```
sum = a + b; // sum is 8
wert = ( 1 + 2 ) * 3
```

Relational Expression:

```
isEqual = (a == b); // isEqual is false
kleiner10 = ( x < 10 )
```

Logical Expression:

```
jaOderNein = true or false;
brotODERkaese = wants_bread xor wants_cheese; // xor normale entweder oder exklusives oder
ungleichZehn = x < 10 and x > 10
isEitherEven = (a % 2 == 0) || (b % 2 == 0); // isEitherEven is false
```

Function Call (Expression):

```
maxVal = max(a, b); // maxVal is 5
```

Ausdruck versus Anweisung

Anweisung (statement)

Deklaration:
x:int;
def test():pass

Definition:
Kontroll Ausdruck:
if, if:else:, for
while, for(:), for(;;)
return Statement (in a function):

Compound Statement (block / body / Körper / Anweisungsliste)
if x: (…)

Bytecode

In Python wird der Quellcode zuerst in eine niedrigere Form namens **Bytecode** übersetzt, bevor er von der Python Virtual Machine (PVM) ausgeführt wird. Dieser Bytecode ist eine **abstraktere Darstellung** des Quellcodes und ist plattformunabhängig. Hier einige grundlegende Informationen dazu, wie man den Bytecode ansehen und aufbewahren kann:

```
import dis

def beispiel():
   return "Hallo Welt"

dis.dis(beispiel)
#   4           0 LOAD_CONST               1 ('Hallo Welt')
#               2 RETURN_VALUE

def beispiel2(x):
   return x * x + 42

dis.dis(beispiel2)
# 11           0 LOAD_FAST                0 (x)
# 2 LOAD_FAST                0 (x)
# 4 BINARY_MULTIPLY
# 6 LOAD_CONST               1 (42)
# 8 BINARY_ADD
# 10 RETURN_VALUE
```

Bytecode

~/alfa/tag4/ hex-dump __pycache__/simple.cpython-312.pyc

```
00000000  cb 0d 0d 0a 00 00 00 00  95 90 33 66 09 00 00 00  |..........3f....|
00000010  e3 00 00 00 00 00 00 00  00 00 00 00 00 02 00 00  |................|
00000020  00 00 00 00 00 f3 12 00  00 00 97 00 64 00 5a 00  |............d.Z.|
00000030  65 00 65 00 7a 05 00 00  01 00 79 01 29 02 e9 03  |e.e.z.....y.)...|
00000040  00 00 00 4e 29 01 da 01  78 a9 00 f3 00 00 00 00  |...N)...x.......|
00000050  fa 09 73 69 6d 70 6c 65  2e 70 79 fa 08 3c 6d 6f  |..simple.py..<mo|
00000060  64 75 6c 65 3e 72 07 00  00 00 01 00 00 00 73 11  |dule>r.......s.|
00000070  00 00 00 f0 03 01 01 01  d8 04 05 80 01 d8 00 01  |................|
00000080  80 21 83 03 72 05 00 00  00                       |.!..r....|
00000089
```

Warum byte/Binärformate? Speichereffizient, optimiertes lesen

Memory Map: Speichere Speicher direkt auf Festplatte (extremst effizient)

Ausdruck versus Anweisung

R-Value versus L-Value

not explicitly applicable because Python handles variables and memory management differently.

Ausdrücke (Expressions) teilen sich in R-Value ≠ L-Value
Namensgebung von right/left, aber Fehlbezeichnung “misnomer”
i = 3; # 3 ist ein r value 'rechts', i ist ein l value 'left'
j = i; # ⚠️ i ist immer noch ein l value!
j = i + 1; # ist wieder ein R-Value

r-value:
Ein temporärer oder nicht-adressierbarer Wert, der nicht über den Ausdruck, der ihn verwendet, hinaus bestehen bleibt.

Merke: "Ein R value hat keinen Namen"

l-value:
Ein Ausdruck, der auf ein Objekt verweist, das einen dauerhaften Speicherort im Speicher hat und daher adressierbar ist.

Merke: "Ein L value hat einen Namen"

Eine Funktion bezeichnet man als **pure** (**rein**), wenn sie keine **Seiteneffekte** hat und bei gleichen Eingaben immer das gleiche Ergebnis liefert.

Keine **Nebeneffekte** heisst: keine globalen Variablen ändern, keine Dateien schreiben oder andere Aktivitäten ausführen, die den Programmzustand ändern könnten. Nicht modifizierend.

```python
# Beispiel reine Funktion
def fakultät(n):
  if n==0 : return 1
  return n * fakultät(n - 1)
```

Pure (reine) Funktionen, welche nicht überflüssig sind, liefen **immer** einen Rückgabewert

Aufgaben

Lesen Tag 9/10& 4:

- **Kapitel 3: Listen**
- **Kapitel 4: Arbeiten mit Listen**
- **Kapitel 6: Dictionary**

Freiwillig:

- was ist 0b0101001 (erst per **Papier**, dann mit python bestätigen)
- was ist 0b1010010 (was fällt auf? ^^)
- was ist 0b1011011
- was ist 0x0101 (!)
- was ist 0x0abcdef
- was ist 0x01A
- was ist 0x10A

- was ist 0b11011011 (negativ)

Aufgaben dictionary

- **Zusammenführen von zwei Dictionaries:** Schreibe eine Funktion, die zwei Dictionaries zusammenführt und dabei die Werte von übereinstimmenden Schlüsseln addiert
- **Umkehren der Schlüssel-Werte-Paare in einem Dictionary:** Erstelle eine Funktion, die die Schlüssel und Werte eines Dictionaries umkehrt.
- darf eine Datei (f = open(file_name)) als Schlüssel verwendet werden? warum(nicht)?
-

Ausdruck versus Anweisung

Ausdruck (Expression) : Wert oder Code Fragment oder Berechnung welches Wert/Ergebnis Erzeugt
Anweisung (Statement) : Komplette Code Zeile welche eine Aktion Ausführt, z.B. Zuweisung oder Aufruf()

```
sum = 3 + 1; # '3 + 1' ist ein Ausdruck, das ganze ist ein Statement ( Zuweisung )
```

Ausdruck (Expression) "das was auf der rechten Seite vom `=` steht"
Ausdruck (Expression) "das was in runden Klammer steht" print(9*9)

Arithmetic Expression:
```
sum = a + b; // sum is 8
wert = ( 1 + 2 ) * 3
```

Relational Expression:
```
isEqual = (a == b); // isEqual is false
kleiner10 = ( x < 10 )
```

Logical Expression:
```
jaOderNein = True or False;
ungleichZehn = x < 10 and x > 10
isEitherEven = (a % 2 == 0) || (b % 2 == 0); // isEitherEven is false
```

Function Call (Expression):
laenge = **len**("abc")
```
maxVal = max(a, b); // maxVal is 5
```
print("abc") **# Statement !**

Ausdruck versus Anweisung

Anweisung (statement)

Deklaration:
x:int;
def test():pass

Definition:
Kontroll Ausdruck:
if, if:else:, while, for(:), for(;;)
return Statement (in a function):

Compound Statement (block / body / Körper / Anweisungsliste)
if x: (…)

Ausdruck versus Anweisung

R-Value versus L-Value

not explicitly applicable because Python handles variables and memory management differently.

Ausdrücke (Expressions) teilen sich in R-Value ≠ L-Value
Namensgebung von right/left, aber Fehlbezeichnung “misnomer”
i = 3; # 3 ist ein r value 'rechts', i ist ein l value 'left'
j = i; # ⚠️ i ist immer noch ein l value!
j = i + 1; # ist wieder ein R-Value

r-value:
Ein temporärer oder nicht-adressierbarer Wert, der nicht über den Ausdruck, der ihn verwendet, hinaus bestehen bleibt.

Merke: "Ein R value hat keinen Namen"

l-value:
Ein Ausdruck, der auf ein Objekt verweist, das einen dauerhaften Speicherort im Speicher hat und daher adressierbar ist.

Merke: "Ein L value hat einen Namen"

Speichern / Laden

Texte (strings) und allgemeine Daten lassen sich in Dateien schreiben und lesen
Dafür können wir die open() Funktion auf folgende Weise nutzen:

```
file_name = "test"

file_out = open(file_name, "w") # "w" ≈ "write"
file_out.write("Hallo Welt")
file_out.close() # ⚠️ nicht vergessen, nach dem Schreiben die Datei schliessen!

file_in = open(file_name)  # "r" ≈ "read" optional
zeilen = file_in.readlines()

print(zeilen)


Aufgabe: was passiert wenn ich ein dict in eine Datei speichern möchte?



# pro tip / später : "wb" für write binary
```

Serialisieren / Speichern von dict als json

Die Ähnlichkeit zu json lässt sich zunutze machen, um dictionaries leicht zu speichern und zu laden:

```
import json

obj = {"name": "Max", "alter": 25}
file_name = "data.json"

s = json.dumps(obj) # object als String serialisieren
obj = json.loads(s) # object aus String laden


json.dump(obj, open(file_name, "w"))  # object als Datei serialisieren
obj = json.load(open(file_name))      # object aus Datei laden

print(obj)
```

json ok für einfache Daten, für komplexere nimm **pickle**

Syntaktischer Zucker

Syntaktischer Zucker

Bezeichnet Syntax, welche nicht notwendig ist, d.h. welche man auch mit anderen Mitteln umsetzen kann, aber welche das Leben schöner oder leichter macht.

z.B.

```
mein_bester_freund["alter"] = mein_bester_freund["alter"] + 1
mein_bester_freund["alter"] += 1    "in-place addition operator"
```

"read-only syntax" reicht meisst, die Syntax zu erkennen, muss man nicht selber aktiv tippen.