Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-05 Dozent: Karsten Flügge

Themen des Tages

Syntax
Lexis/Semantik
PEP-8

- **Interpreter** vs. Compiler

Zahlensysteme (binär, oktal, hex)
- Int, Float
- **Binäre Zahlen**
- **Hexadezimale Zahlen**
- **Scientific Notation**

```
1E2
100.0
```

Syntax

Lexis/Semantik
PEP-8

- **Interpreter** vs. Compiler

Zahlensysteme (binär, oktal, hex)

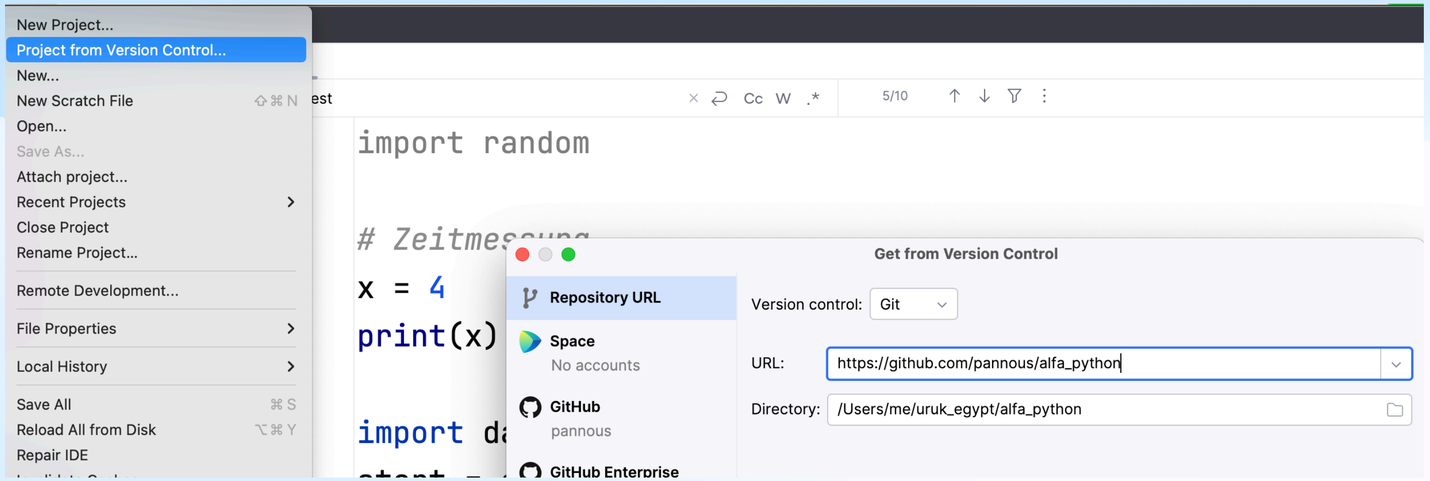
- Int, Float
- **Binäre Zahlen**
- **Hexadezimale Zahlen**
- **Scientific Notation**

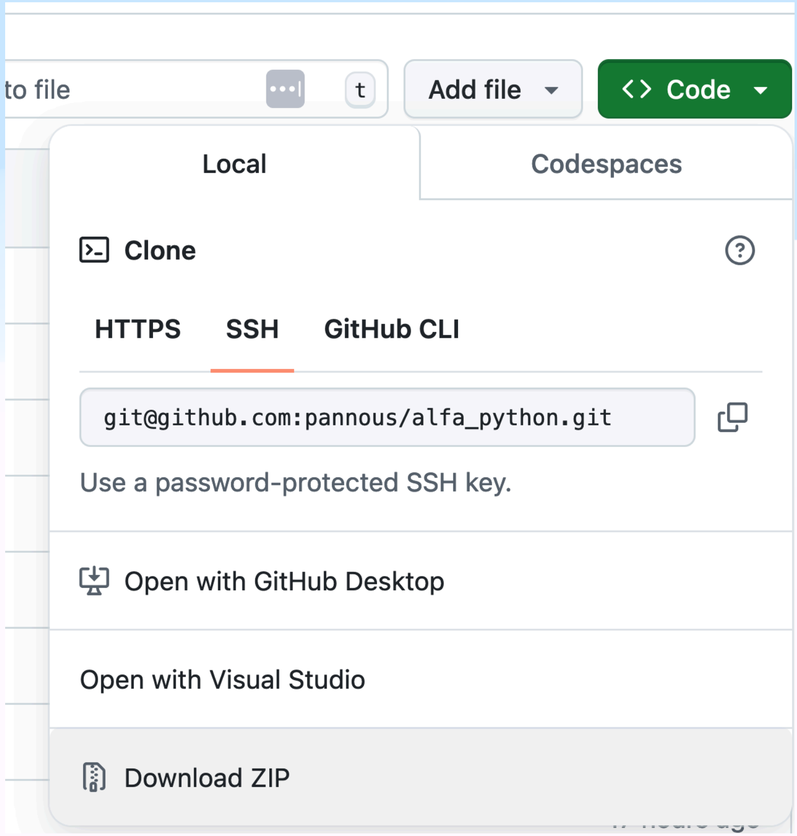
```
1E2
100.0
```

GitHub

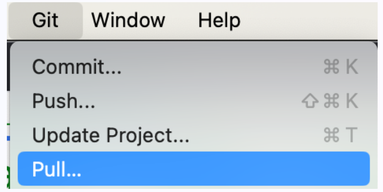
Freiwillig GitHub Account anlegen
zum Herunterladen meiner Dateien und Lösung

https://github.com/pannous/alfa_python





aktualisieren über git pull



Begriff Syntax

Syntax ≈ "Erlaubter Code"
Syntax ≈ Grammatik ≈ "Zulässige Ausdrücke"

Die **Syntax** in der Programmierung bezieht sich auf die **Regeln**, die festlegen, wie Code geschrieben werden muss, damit der Computer ihn **verstehen und ausführen** kann. Man kann sich das ähnlich wie die Grammatik in einer Sprache vorstellen. Wenn man in der deutschen Sprache einen Satz bildet, muss man sich an bestimmte Regeln halten, wie die richtige Reihenfolge von Subjekt, Prädikat und Objekt, damit der Satz verständlich ist. In ähnlicher Weise gibt es in jeder Programmiersprache spezifische Regeln, wie Befehle, Funktionen und andere Anweisungen formuliert werden müssen.

"drei minu 5 nie gut wie"

- Beenden von 'Sätzen' am Ende des Dokumentes.
- ... genaue **Syntax** lernen wir in den nächsten Wochen Schritt für Schritt

Lexis/Semantik

Lexis (Lexikalische Struktur)**, Teil der Syntax**

Die Lexis einer Sprache beschreibt ihren "**Wortschatz**"

— die kleinsten bedeutungstragenden Einheiten, die der Parser erkennt.

In Python unterscheidet man:

Keywords (reservierte Wörter): `if`, `else`, `for`, `while`, `def`, `class`, `return`, `import`, `True`, `False`, `None`, ... — insgesamt **35 Stück** (Python 3.12). Können nicht als Variablennamen verwendet werden.

Identifier (Bezeichner): Selbstgewählte Namen für Variablen, Funktionen, Klassen. Regeln: beginnen mit Buchstabe oder _, keine Sonderzeichen, case-sensitive (`Alter ≠ alter`).

Literale: Feste Werte im Quellcode — z.B. `42`, `3.14`, `"Hallo"`, `True`, `[1,2,3]`.

Operatoren: `+`, `-`, `*`, `=`, `==`, `!=`, `and`, `or`, ...

Delimiter/Trennzeichen: `(`, `)`, `:`, `,`, `.`, `=`, ...

Kommentare: `# einzeilig` — werden vom Interpreter ignoriert.

Lexis/Semantik

Keywords (reservierte Wörter): `if`, `else`, `for`, `while`, `def`, `class`, `return`, `import`, `True`, `False`, `None`, ... — insgesamt **35 Stück** (Python 3.12). Können nicht als Variablennamen verwendet werden.

```
False    await    else      import     pass
None     break    except    in         raise
True     class    finally   is         return
and      continue for       lambda     try
as       def      from      nonlocal   while
assert   del      global    not        with
async    elif     if        or         yield
```

Lexis/Semantik

Semantik (Bedeutung)

Lexis und Syntax sagen, *was man schreiben darf*. Semantik sagt, *was es bedeutet*.

print("Hallo") # Die **Bedeutung** sollte jedem **klar** sein. Und ich behaupte mal, das ist auch **syntaktisch korrekt.**

Syntaktisch falsch, semantisch 'korrekt':

print "Hallo" # Auch wenn uns die Bedeutung klar ist, versteht der Computer das so nicht (mehr).

Syntaktisch korrekt, semantisch falsch:

```
x = "Hallo" + 5   # TypeError zur Laufzeit
```

Python "versteht" die Einzelteile, aber die Operation ergibt keinen Sinn.

Statische vs. dynamische Semantik: Python ist dynamisch typisiert — viele semantische Fehler fallen erst zur Laufzeit auf, nicht beim Kompilieren.

```
def addiere(a, b):
    return a + b

addiere(3, 4)       # → 7
addiere("Py", "thon") # → "Python"
addiere("Py", 3)    # → TypeError
```

Dieselbe Syntax, drei verschiedene Bedeutungen — das ist Semantik in Aktion.

Kernbotschaft: Lexis = Vokabeln, Syntax = Grammatik, Semantik = Bedeutung. Python prüft Syntax beim Parsen, aber Semantik oft erst bei der Ausführung.

Lexis/Semantik

Semantik (Bedeutung)

Lexis und Syntax sagen, *was man schreiben darf*. Semantik sagt, *was es bedeutet*.

print("Hallo") # Die **Bedeutung** sollte jedem **klar** sein. Und ich behaupte mal, das ist auch **syntaktisch korrekt.**

Syntaktisch falsch, semantisch 'korrekt':

print "Hallo" # Auch wenn uns die Bedeutung klar ist, versteht der Computer das so nicht (mehr).

Syntaktisch korrekt, semantisch falsch:

```
x = "Hallo" + 5   # TypeError zur Laufzeit
```

Python "versteht" die Einzelteile, aber die Operation ergibt keinen Sinn.

Lexis/Semantik

Syntaktisch korrekt aber falsches Wort:

```
alter = input("wie alt bist du") # Zuweisung
print(Alter , "ist viel zu alt")
```

NameError: name 'Alter' is not defined. Did you mean: 'alter'?

Semantik ≈ Bedeutung

Statische vs. dynamische Semantik:

Python ist dynamisch typisiert — viele semantische Fehler fallen erst zur Laufzeit auf, nicht beim kompilieren.

```
def addiere(a, b): # Genaue Semantik ergibt sich erst beim Aufruf  ...
    return a + b

addiere(3, 4)        # → 7
addiere("Py", "thon") # → "Python"
addiere("Py", 3)     # → TypeError
```

Dieselbe Syntax, drei verschiedene Bedeutungen — das ist Semantik in Aktion.

Kernbotschaft:

Syntax = Grammatik,

Lexis = Vokabeln,

Semantik = Bedeutung. "Der Hund fraß das Karussell"

Python prüft **Syntax** beim Parsen, aber Semantik oft erst bei der Ausführung.

Probieren!

Python ist eine hervorragende Sprache um Dinge auszuprobieren.

Mal schauen was wir für Ergebnisse herauslockern können bevor wir uns mit der Theorie beschäftigen.

Aufgabe Python als Taschenrechner

Berechne

3 + 3

3 mal 3

2 + zwei halbe

3 milliarde mal 3 milliarde

Versuche:

3 hoch 3 ? **

Python Bytecode

Vor dem Aufruf von python wird das Programm umgewandelt in **Bytecode.** Das wird dann interpretiert. (Intern zum cachen)

Bei Java wird Bytecode oft vorher in Maschinencode umgewandelt, das macht python nicht.

Andere Sprachen wandeln Programm direkt in **Maschinencode** um.

PEP 8

PEP 8 — Der Style Guide für Python-Code

PEP = **P**ython **E**nhancement **P**roposal. PEP 8 ist *das* 'verbindliche' Regelwerk für lesbaren, einheitlichen Python-Code.

Warum PEP 8? Code wird häufiger gelesen als geschrieben. Einheitlicher Stil reduziert kognitive Last im Team. Quasi die "Rechtschreibregeln" von Python. Vereinfacht Kooperation.

Die wichtigsten Regeln:

Einrückung: 4 Leerzeichen pro Ebene (keine Tabs).

```
if x > 0:
    print("positiv")
```

Zeilenlänge: Maximal 79 Zeichen.

Leerzeilen: 2 Leerzeilen vor Funktions-/Klassendefinitionen auf Modulebene, 1 Leerzeile zwischen Methoden innerhalb einer Klasse.

Imports: Immer am Dateianfang, jeder Import auf eigener Zeile.

```
import os
import sys
# NICHT: import os, sys
```

PEP 8

Namenskonventionen:

- Variablen/Funktionen: **`snake_case`** → `mein_wert`, `berechne_summe()`
- Klassen: `CamelCase` → `MeinObjekt`
- Konstanten: `UPPER_SNAKE_CASE` → `MAX_GROESSE`
- **Privat**: `_vorangestellter_unterstrich` `__doppelter_unterstrich_noch_privater`

Leerzeichen:

```
# Ja:
x = 1
liste = [1, 2, 3]
ergebnis = funktion(a, b)

# Nein:
x=1
liste = [1 , 2 , 3]
ergebnis = funktion( a,b )
```

Vergleiche:

```
# Ja:
if x is None:
if not liste:

# Nein:
if x == None:
if len(liste) == 0:
```

Kommentare: Auf Englisch (in internationalen Projekten). Inline-Kommentare sparsam, mindestens 2 Leerzeichen Abstand.

PEP 8

Don't worry, IDE does it!

Praxistipp für Teilnehmer:

In PyCharm ist PEP-8-Prüfung eingebaut (gelbe Warnungen). Auf der Kommandozeile:

```
pip install pycodestyle
pycodestyle mein_script.py
```

Oder automatisch formatieren lassen:

```
pip install black
black mein_script.py
```

Pause

Zahlen in Python

Python-Zahl**Typen:**

Typ	Beispiel	Bemerkung
Natürliche Zahlen : Integer		
int	42	**beliebig große** Ganzzahlen
Fließkomma Zahlen		
float	3.14	IEEE-754 **double** precision
complex	2 + 3**j**	komplexe Zahlen
Wahrheit Boolean		
bool	True/False	Unterklasse von int (True == 1)

Pause

Zusätzliche numerische Typen aus Bibliotheken:

Modul	Typ	Zweck
`decimal`	`Decimal`	exakte Dezimalarithmetik
`fractions`	`Fraction`	rationale Zahlen
`numpy`	`int8`, `float32`, `uint64`, …	feste Bitbreiten, SIMD/NumPy
`mpmath`	`mpf`	beliebige Präzision
`sympy`	`Integer`, `Rational`, `Real`	symbolische Mathematik

`tensorflow / pytorch oder` **`numpy`**

Zahlensysteme

Zahlensysteme (binär, oktal, hex)

- **Int, Float**
- **Binäre Zahlen**
- **Hexadezimale Zahlen**
- **Scientific Notation**

```
1E2
100.0
```

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

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

Großer Durchbruch:

Positionsabhängiges Zählen:

77 = 10*__7__ + 7

123 = 1*100 + 2*10 + 3*__1__

abcd = a*10^3 + b*10^2 + c*10^1 +d*__10^0__ "__basis__ 10"

Binäre Darstellung … Basis 2 = {0,1}

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 …

Morsen: Basis 3;)

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

1111 decimal = 1000 + 100 + 10 + 1
0b1111 = 2^3 + 2^2 + 2^1 + 2^0 =
0b1111 = 8 + 4 + 2 + 1 = **15**

byte 0b11111111 8 bit 256 Zahlen
unsigned byte 256 Zahlen
signed byte KONVENTION: alles über 128 ist negativ

0b0111 = 0 + 4 + 2 + 1 = 7
0b0101 = 0 + 4 + 0 + 1 = 5

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

1111 decimal = 1000 + 100 + 10 + 1
0b1111 = 2^3 + 2^2 + 2^1 + 2^0 =
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}

```
Negative Zahlen

byte 0b11111111 8 bit 256 Zahlen
unsigned byte 256 Zahlen
signed byte KONVENTION: alles über 128 ist negativ

signed byte x = -1 // 0b11111111
signed byte x = -2 // 0b11111110

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 (Hexadezim griechisch: **16** decim+hexa)

0b0000 = **0x**0 = 0
0b0001 = 0x1 = 1
0b1001 = 0x9 = 9
0b1010 = 0xA = 10
0b1011 = 0xB = 11
0b1100 = 0xC = 12
0b1101 = 0xD = 13
0b1110 = 0xE = 14
0b1111 = 0xF = 15 !
0…15 0x0…xF Ziffer nennt man "**Nibble**"

Motivation byte = 8 bit 0b1111_1111

0x55 == 5*16 + 5*1
0xB12 = B***16**^{2} + 1*16^1 + 2*16^0
0xB12 = 11*256 + 1*16 + 2*1 (rechts: Dezimalzahlen)

byte 255 Zahlen = 2 Zeichen 00 … FF

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

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
0xFFFF = größtes (unsigned) short
0xFFFFFFFF = größtes (unsigned) int (32 bit = 8 nibbles)
0xFFFFFFFFFFFFFFFF = größtes (unsigned) long long (64 bit = 16 nibbles)

Binäre bitwise Operatoren

```
0 = False
1 = True

and, or, not

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

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  # invalid syntax, nur theoretisch
0b0.010 = 1/4 = 2⁻²
0b0.001 = 1/8 = 2⁻³
```

z.B.: 3.125 = 3⅛ = 2+1+1/8 = 11.001

```
print(decimal_to_binary(math.pi, 100))
```

```
π = 11.00100100001111110110101010001000100001011010O
```

```
π = 11.001001000011111101101010001000100001011010
```

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 binary32 format:

π ≈ + 2^1 * 1.00100100001111110110110 (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

π as float32 : 00100100001111110110110 (implicitly with a leading 1)