Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-05 Dozent: Karsten Flügge

Themen des Tages

- Funktionen
  - Argumente ≈ Parameter
  - named args : Schlüsselwortargumente
  - default Standardwerte
  - call by value / call by reference

Syntactic Sugar for Functions

nützlich:

- **Default Argumente**
- **pass Keyword**
- **docstrings # nützlich!**

komplex:

- **Typsicherheit**
- **var**iadic **arg**uments
- **k**ey**w**ord **arg**uments
- **global**

Syntactic Sugar for Functions

Mehrzeilige Ausdrücke in python via **Backslash ** sind erlaubt aber verpönt!
Stattdessen: Ausdrücke aufteilen in kleiner Teile (Variablen / Funktionen)

```
if True \
   and True:
  print("OK")

istWochenende = wochentagende in ["Samstag","Sonntag"]
istHeiss = aussentemperatur > minimalerSchwitzGrad
if istWochenende and istHeiss and arbeitslaune == False:
   print("Arbeitsverweigerung")
```

Schlüsselwort Argumente

In python muss man sich die Reihenfolge von Argumenten nicht merken, sondern kann sie per Name übergeben:

```
def drucke_info(titel, alter, beruf):
  print(f"Titel: {titel}")
  print(f"Alter: {alter}")
  print(f"Beruf: {beruf}")
```

```
drucke_info("Frau Muster", 30, "Ingenieurin") # sicher? oder kam doch der Beruf zuerst?
drucke_info(titel="Frau Muster", alter=30, beruf="Ingenieurin")
```

```
drucke_info(alter=30, beruf="Ingenieurin", titel="Frau Muster") # Aha!
```

💡 Selbst wenn man die original Reihenfolge beibehält sind diese **benamten Parameter** sehr hilfreich für die Lesbarkeit!

💡 Besonders da Python (erstmal) keine Typsicherheit hat kann man so auch vermeiden das die **falsche Information an der falschen Stelle** landet! apropos…

Schlüsselwort Argumente

Achtung PyCharm

PyCharm zeigt die Namen 'hints' der Parameter grau mit ":" an

```
drucke_info( titel: "Frau Muster",  alter: 30,  beruf: "Ingenieurin")
drucke_info(titel="Frau Muster", alter=30, beruf= "Ingenieurin")
^^^^
```

die richtige Syntax ist aber mit "="

kann beim Abtippen / Screenshots zu Verwirrung führen

Default Parameter

In python kann man Funktionen Standard Werte für Argumente mitgeben:

```
def connect(server="localhost", port=8080): pass
```

Diese können dann beim Aufruf weggelassen oder überschrieben werden:

```
connect() # ok nutze vorgegebene Werte
connect(port=80)
connect(server="myhost.com")
connect(port=80, server="myhost.com")
connect("myhost.com") # erstes explizit, zweites default
connect("myhost.com", 80) # reguläre Argumentübergabe
```

⚠️ Standardwerte müssen bei der Funktionen Deklaration **am Ende stehen:**

```
def default_ambiguity(x=42, y, z=44): # SyntaxError: non-default argument follows default argument
  print(y)

default_ambiguity(43, 43) # x,y or y,z ?? Mehrdeutigkeit!
```

Typsicherheit von Funktionen

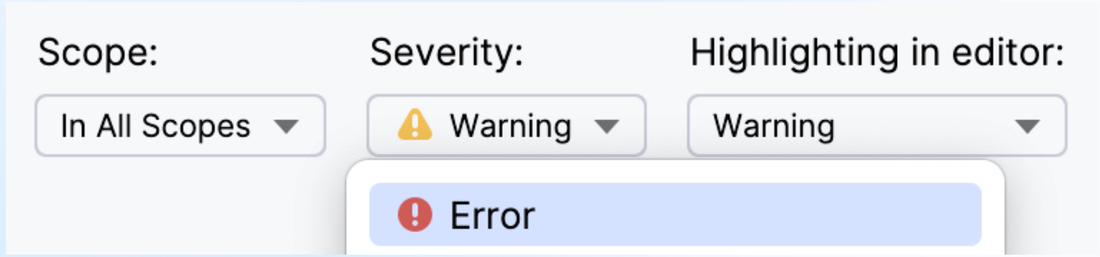
mit dem **mypy** plugin kann man sich in Pycharm automatisch den Typ prüfen lassen!

```
def function(param: type) -> return_type: body
```

```
def addiere(zahl1: int, zahl2: int) -> int:
  return zahl1 + zahl2
```

```
addiere( zahl1: "a", zahl2: "b")
```

wird rot unterstrichen. Damit es tatsächlich als Fehler behandelt wird:




```
x:int
x = "a"
```

geht auch für Variablen!

```
zahl : int = "b"
```

Typsicherheit von Funktionen

```
def addiere(zahl1, zahl2):
  return zahl1 + zahl2
```

```
# : int der Typ hinter Doppelpunkt heisst, dass Funktion als Argumente hier nur Zahlen erwartet
# -> float der Typ hinter dem Pfeil sagt, was die Funktion zurückgibt, hier eine Zahl mit Nachkommastellen
def addiere(zahl1: int, zahl2: int) -> float:
  return zahl1 + zahl2
```

⚠️ Achtung, bis python 3.14 sind dies nur **Kommentare, man muss sich beim Aufruf nicht dran halten:**

```
print(addiere("5", "3")) # geht noch 😡 und kann zu semantischen Fehlern führen
```

Typsicherheit von Variablen

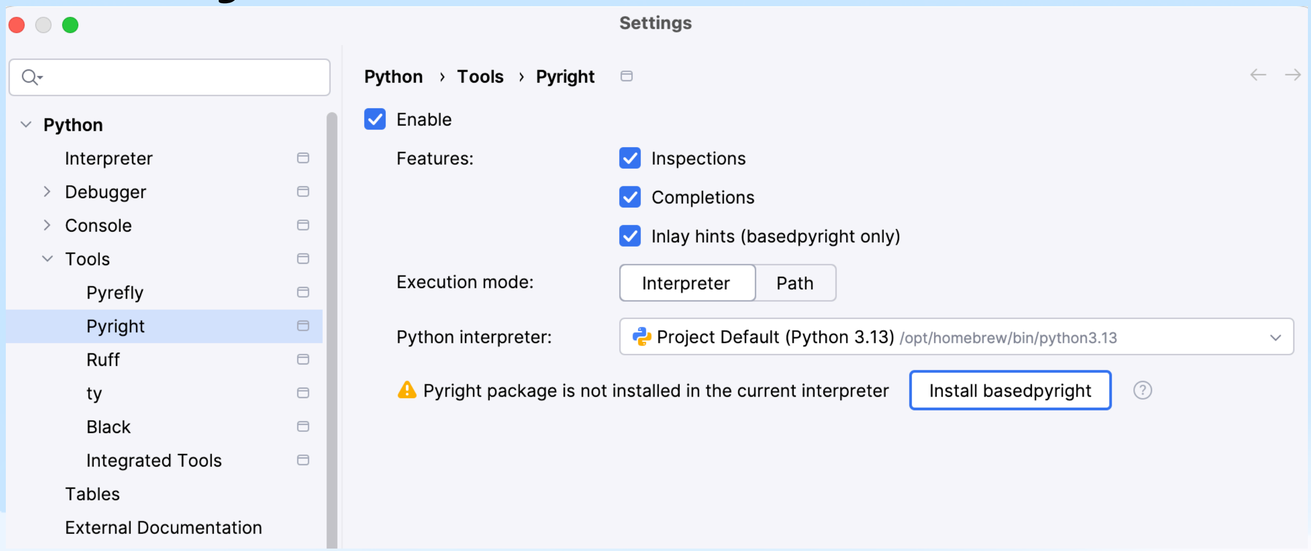
```
zahl1: int = 7
```

⚠️ Achtung, bis python 3.14 sind dies nur **Kommentare, man muss sich beim Aufruf nicht dran halten:**

```
zahl1: int = "7" # geht noch 😠 und kann zu Fehlern führen
print(type(zahl1)) # <class 'str'> 😠😠😠
```

Typsicherheit beim Starten

Settings:




```
# pyright: strict
oben in Datei
```

Beliebige Parameterzahl

“**var**iadic **arg**uments”. variierende Anzahl von Parametern

- ***vararg / list deconstruction**

Statt einer festen Anzahl von Parameter kann eine Funktion auch eine beliebige Anzahl von Parametern erhalten z.B. print(1,2,3)

Dazu schreibt man ein Sternchen vor das Argument:

```
def polyparams(*varargs):
  for arg in varargs:
    print("Wie in Liste: ", arg)


polyparams(1, 2, 3, 4, 5) # einzelne Argumente werden wie Liste übergeben
polyparams([1, 2, 3, 4, 5]) # Achtung, Liste zählt als ein Argument
polyparams(*[1, 2, 3, 4, 5]) # "unpacking deconstruction" von Liste zählt als viele Argumente

def polyparams(*varargs:int) # kann auch Typ festgelegt werden
```

Beliebige Parameterzahl

Achtung, Listen werden bei **var**iadic **arg**uments nicht in Elemente aufgelöst, sondern als ganzes Objekt betrachtet:

```
>>> print([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
>>> print(1, 2, 3, 4, 5)
1 2 3 4 5
```

```
polyparams(1, 2, 3, 4, 5) # einzelne Argumente werden wie Liste übergeben
polyparams([1, 2, 3, 4, 5]) # Achtung, Liste zählt als ein Argument
polyparams(*[1, 2, 3, 4, 5]) # "unpacking deconstruction" von Liste zählt als viele Argumente

def polyparams(*varargs:int) # kann auch Typ festgelegt werden
```

Beliebige Schlüsselwortargumente

Beliebige Parameterzahl mit Namen
Syntactic suggar

- ****kwargs (keyword arguments) dict deconstruction**

Statt einer festen Anzahl von Parametern kann eine Funktion auch eine beliebige Anzahl von Parametern mit Namen erhalten

Dazu schreibt man zwei Sternchen vor das Argument:

```
def polyKeywords(**kwargs): # kwargs ist jetzt ein dict!
  for key, value in kwargs.items():
    print(f"{key}: {value}")

def polyKeywords2(**kwargs):
  for key in kwargs:
    value = kwargs[key]
    print(f"{key}: {value}")

polyKeywords(a=1, b=2, c=3) # pseudo dict
polyKeywords(**{"a":1, "b":2, "c":3}) # decomposing dict
```

Diese spezielle Syntax wandelt eine Liste von key-value Argumenten in ein echtes dict um: 'kwargs'

Docstring

- **Man kann Funktionen mit sogenannten Docstrings dokumentieren:**

```
# Docstrings
def function_name(param1, param2):
    """
    Description of what the function does.

    Args:
        param1 (type): Description of param1.
        param2 (type): Description of param2.

    Returns:
        type: Description of return value.

    Raises:
        ExceptionType: Why and when this exception is raised.

    """
    # function body
    pass
```

Dies erfolgt über speziellen mehrzeilen Kommentar String """ welche durch **help(function_name)** auslesbar ist

Docstring Beispiel

```
# Docstrings Kommentare direkt hinter der Definition, welche intern ausgelesen werden
# Steht dann per help zur Verfügung

def division(x, y):
    """
    zwei Zahlen division

    Args:
        x (int): erste Zahl
        y (int): zweite Zahl

    Returns:
        int: die Division von x und y

    Raises:
        DivisionByZero: Man darf nicht durch 0 teilen

    """
    # if y == 0:
    # raise DivisionByZero("")
    return x / y

help(division)
```

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*100 + 2*10 + 3*__1__

Folge von Ziffern
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   # binary bit
0b0001 = 1 = 2⁰  letzte Ziffer immer kleinster Wert
0b0010 = 2 = 2¹
0b0100 = 4 = 2²
0b1000 = 8 = 2³
```

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

byte 0b11111111 8 bit 256 Zahlen (2**8)
```

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}

0b0011 = 0*2³ + 0*2² + 1*2¹ + 1*2⁰ =
0b0011 = 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 =
0b1101 = 8 + 4 + 0 + 1 = **13**

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 = 0x**A** = 10
0b1011 = 0xB = 11
0b1100 = 0xC = 12
0b1101 = 0xD = 13
0b1110 = 0xE = 14
0b1111 = 0x**F** = 15 !
0…15 0x0…xF Ziffer nennt man "**Nibble**" "hex"

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)

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 = 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 >>> 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

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
```

```
Malen mit bits (heute noch oft)

00000000000000000000000000
00000000011111100000000000
00000000010000100000000000
00000000011111100000000000
00000000010000100000000000
00000000010000100000000000
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:

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

$$0b0.100 = 2 = 2^{-1}$$
$$0b0.010 = 4 = 2^{-2}$$
$$0b0.001 = 8 = 2^{-3}$$

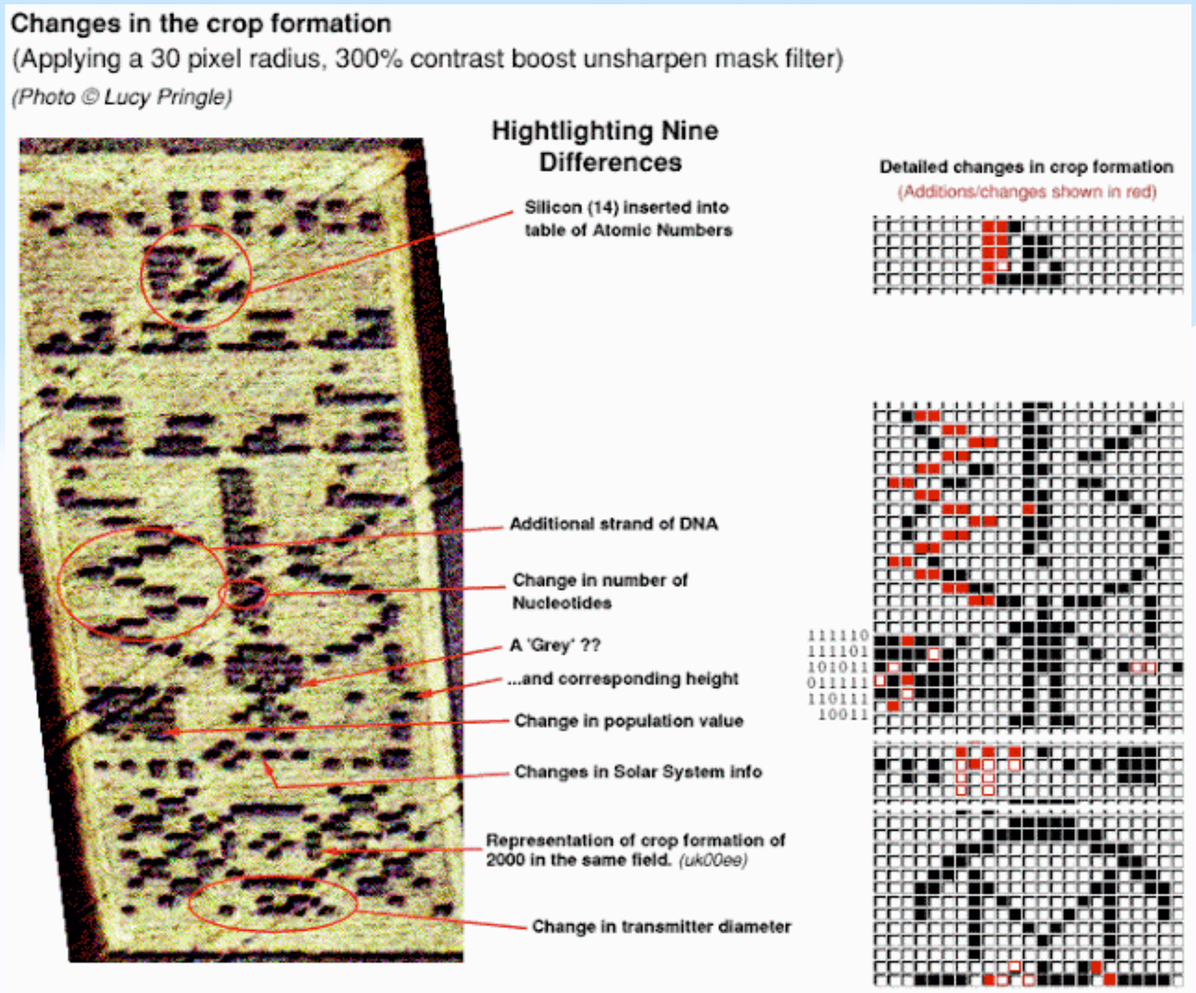
$3.125 = 3 + 1*10^{-1} + 2*10^{-2} + 3*10^{-3}$

z.B.: 3.125 = 3⅛ = 2+1+1/8 = 11.001

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

```
π = 11.00100100001111111011010101000100010000101101000
```

```
π = 11.001001000011111110110101010001000100001011010
```



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)

Vokabeln

Vokabeln

Funktion
Methode
Prozedur
pur
Block
Funktionskörper / **body**
Funktionsvariable(n)
Parameter(liste)
Argument
Rekursion
Breakpoint
Thread (gibt erst man nur das Hauptprogramm)
Stack **Trace** / Back Trace / Traceback