1. Floating-Point-Zahlen

b) 255,25 = 255 + 0,25 = 11111111,01 $_{(2)}$ = 1,111111101 $_{(2)}$ * 2^7

Vorzeichen: 0 (positiv)

Exponent (8 Bit): 7, plus Exzess (127) -> 134 = 10000110 $_{(2)}$

Mantisse (23 Bit): 111111101 (Rest Nullen)

Zusammen: 01000011011111110100000000000000 $_{(2)}$ = 0x437F 4000

Probe:
```
[esser@quad:Gleitkomma]$ ./gleitkomma.py 255.25 | tail -2
      DualToIEEE(32):  0 10000110 11111110100000000000000
      Hexadezimal:     437F 4000
```

0,2578125 = 1/4 + 1/128 = 0,0100001 $_{(2)}$ = 1,00001 $_{(2)}$ * 2^–2

Vorzeichen: 0 (positiv)

Exponent: –2, plus Exzess (127) -> 125 = 01111101 $_{(2)}$

Mantisse: 00001 (Rest Nullen)

Zusammen: 00111110100001000000000000000000 $_{(2)}$ = 0x3E84 0000

Probe:
```
[esser@quad:Gleitkomma]$ ./gleitkomma.py 0.2578125 | tail -2
      DualToIEEE(32):  0 01111101 00001000000000000000000
      Hexadezimal:     3E84 0000
```

2. 32-Bit-Arithmetik

a) Perioden

0,1 = 0,0001100110011001100110011... $_{(2)}$ = 0,0 $\overline{0011}$ $_{(2)}$

0,2 = 2 * 0,1
= 0,0011001100110011001100110011... $_{(2)}$ = 0, $\overline{0011}$ $_{(2)}$

0,3 = 0,0100110011001100110011... $_{(2)}$ = 0,01 $\overline{0011}$ $_{(2)}$

0,3333... = 0,010101010101010101... $_{(2)}$ = 0,$\overline{01}$ $_{(2)}$

0,4444... = 0,011100 011100 011100... = 0,$\overline{011100}$ $_{(2)}$

0,5555... = 0,100011 100011 100011... = 0,$\overline{100011}$ $_{(2)}$

Proben: 0,3333... = 1/3; mit 3 multiplizieren: 2 * 0,$\overline{01}$ $_{(2)}$ = 0,$\overline{10}$ $_{(2)}$; Summe: 0,$\overline{11}$ $_{(2)}$ = 1,

1 = 4/9 + 5/9 = 0,444... + 0,555... = 0,$\overline{011100}$ $_{(2)}$ + 0,$\overline{100011}$ $_{(2)}$ = 0,$\overline{111111}$ $_{(2)}$ = 1

b) Genauigkeit – Zahlen, die „groß“ sind und gleichzeitig einen „kleinen“ Anteil haben, benötigen in (exakter) Dualdarstellung mehr Stellen als die Mantisse Platz bietet. Die letzten Nachkommastellen fallen also weg (Rundung).

9999,0001 = **1001 1100 0011 11,00 0000 000** 00001101000110110111000101110 1 $_{(2)}$ hat diese Eigenschaft.

3. Erweiterung: 16- und 128-bittig

a)

[...]

```
  # Ausgabe: IEEE-754-Darstellung (32 oder 64 Bit)
  if laenge == 64:
    explen = 11    # Laenge des Exponenten
    manlen = 52    # Laenge der Mantisse; 1 + 11 + 52 = 64
    exzess = 1023  # Exzess 1023 (2^(11-1)-1)
  elif laenge == 32:
    explen = 8     # Laenge des Exponenten
    manlen = 23    # Laenge der Mantisse; 1 + 8 + 23 = 32
    exzess = 127   # Exzess 127 (2^(8-1)-1)
  elif laenge == 16:
    explen = 5     # Laenge des Exponenten
    manlen = 10    # Laenge der Mantisse; 1 + 5 + 10 = 16
    exzess = 15    # Exzess 15 (2^(5-1)-1)
  elif laenge == 128:
    explen = 15    # Laenge des Exponenten
    manlen = 112   # Laenge der Mantisse; 1 + 15 + 112 = 128
    exzess = 16383 # Exzess 16383 (2^(15-1)-1)
```

[...]

Außerdem ist die Zeile `counter = 52` (in der Berechnung der Nachkommastellen) anzupassen, und am Ende des Programms muss man passende Ausgabe-Anweisungen für 16 und 128 Bit einbauen.

Das Ganze funktioniert trotzdem nicht gut: Beim Versuch, z.B. 0,1 in eine 128-bittige Darstellung zu bringen, kommt es zu Rechenfehlern.

```
[esser@quad:Gleitkomma]$ ./gleitkomma-16-128.py 0.1

Zahl:            0.1
Vorzeichen:      0 (+)
Vorkomma:        0
Vorkomma(b):
Nachkomma:       0.1
Nachkomma(b):    (0.)0001100110011001100110011001100110011001100110011001100110011001
Dualdarstellung: +.0001100110011001100110011001100110011001100110011001100110011001
Fast normiert:   +1,1001100110011001100110011001100110011001100110011001100110011001 * 2^-4

IEEE 754: (Vorzeichen, Exponent, Mantisse)
DualToIEEE(128): 0 011111111111011 10011001100110011001100110011001100110011001100110011001100110100000
00000000000000000000000000000000000000000000000000000000000000000000
Hexadezimal:     3FFB 9999 9999 9999 A000 0000 0000 0000
DualToIEEE(64):  0 01111111011 1001100110011001100110011001100110011001100110011001100110011010
Hexadezimal:     3FB9 9999 9999 999A
DualToIEEE(32):  0 01111011 10011001100110011001100
Hexadezimal:     3DCC CCCC
DualToIEEE(16):  0 01011 1001100110
Hexadezimal:     2E66
```

b) Vorschlag für 512-bit-Format:

Bit	Exp.	Mant.	Exzess
16	5	10	15
32	8	23	127
64	11	52	1023
128	15	112	16383
256	**19**	**236**	**2^(19-1)-1 = 262.143**
512	**24**	**487**	**2^(24-1)-1 = 8.388.607**

(nur Vorschläge...)