Beim Rechnen mit Kommazahlen am Computer können stets Fehler passieren!
auch in Excel etc.

Warum?

- Rundungsfehler: Wenn man nur auf 4 Nachkommastellen genau rechnet und etwa $\frac{1}{3} = 0.3333... = 0.\overline{3}$ auf 0.3333 rundet, so gilt exakt
$$0.3333 + 0.3333 + 0.3333 = 0.9999 \neq 1$$
aber genau natürlich $\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1$.
Fazit: Die Summe der Rundungen ist nicht die Rundung der Summe!
Dasselbe kann bei anderen Rechenoperationen, also Multiplikation, Subtraktion oder Division passieren. Das passiert genauso in anderen Stellenwertsystemen, etwa im Binärsystem, weshalb der Computer Rundungsfehler macht.
- subtile Rundungsfehler: Der Befehl `print((0.3-0.1) == 0.2)` in Python liefert als Ausgabe `FALSE`! Dies ist im Zehnersystem nicht mit Rundungsfehlern zu erklären.
Aber: Im Binärsystem, als binäre Kommazahl, sind 0.1 und 0.2 und 0.3 periodisch und brechen nie ab! Der Computer rechnet mit gerundeten binären Kommazahlen und macht deswegen manchmal Rundungsfehler, auch wenn man diese vom Dezimalsystem her keineswegs erwartet!
Mit den Befehlen

```
print("%.20f" % (0.3-0.1))
print("%.20f" % 0.2)
```

kann man sehen, dass 0.3-0.1 und 0.2 in Python verschieden sind! Die Ausgabe ist

```
0.19999999999999998335
0.20000000000000001110
```

1. Codierung von Texten

1.1. **ASCII- und Unicode-Codierung.** Erklärungen im Video, vgl. auch Wikipedia: ASCII.

Zu Unicode und UTF-8 (Unicode Transformation Format 8bit): Unicode ist wie ASCII eine Zuordnung zwischen gewissen Zeichen und Zahlen. Diese Zahlen schreibt man meist hexadezimal mit `U+` als Präfix und nennt so etwas *Unicode code point*.

Zum Beispiel ist dem Eurozeichen € der Unicode code point U+20AC zugeordnet. Diesem ordnet die Kodierung UTF-8 die Folge 11100010 10000010 10101100 aus drei Bytes oder hexadezimal E2 82 AC zu (Beispiel aus Wikipedia UTF-8; Examples):

Zeichen: € $\xleftrightarrow{\text{Unicode}}$ Unicode code point: U+20AC $\xleftrightarrow{\text{UTF-8}}$ E2 82 AC

Warum speichert der Computer nicht direkt den Unicode code point? Wenn ich es recht verstehe, bräuchte er dann pro Zeichen 4 Bytes; mit weiterer Umwandlung per UTF-8 geht es oft sehr viel platzsparender, insbesondere bei englischen oder westlichen Texten, wo die meisten Zeichen mit einem oder zwei Byte codiert werden.

Aufgabe 1.1.1. (a) (ähnlich wie im Video) Was ergeben die folgenden hexadezimalen Zahlen, wenn man sie per ASCII (= American Standard Code for Information Interchange) als Zeichen interpretiert? Verwende eine ASCII-Tabelle aus dem Internet!

46 61 6c 73 63 68 3f

(b) Was ergeben die folgenden Dezimalzahlen, wenn man sie mit ASCII entschlüsselt?

71 117 116 32 103 101 109 97 99 104 116 33

(c) Für welche Zeichen stehen die folgenden Unicode code points?

U+0061 U+0430 U+1f99c U+1fbf9 U+1f939 U+1f3bc U+4e23

2. Codierung von Bildern

Die Teile über RGB und Pixelgrafik-Teil folgen stark einer Vorlage von Barbara Pampel.

2.0.1. Die wesentlichen Verfahren zum Codieren von Bildern sind

- Rastergrafik (auch Pixelgrafik, *raster graphics*, *bitmap*, *pixmap*)
- Vektorgrafik (*vector graphics*)

Zuerst muss man aber wissen, wie man Farben kodiert.

2.1. **RGB-Farbmodell.**

2.1.1. Farben werden häufig mit dem RGB-Modell kodiert (RGB = red green blue = rot grün blau). Für jede der drei Farben (in der Reihenfolge RGB) speichert man den Farbanteil als eine Zahl zwischen 0 und 255, meist als zweistellige Hexadezimalzahl. Die Farbe ergibt sich dann durch additive Farbmischung (so wie sich die Lichtkegel von roten, grünen und blauen Scheinwerfern auf einer weissen Wand„addieren“).

Unterscheide dies von der subtraktiven Farbmischung, die etwa beim Drucken verwendet wird (CMYK-Farbmodell).

Aufgabe 2.1.2. (a) Welche Farbe kodiert 1E53DF?
(b) Gib die Codierung (in Hexadezimalziffern) eines Lila-Tons an, ohne die Farbwerte von Grün und Blau zu verändern.
(c) Wie viele Farben kann man mit dem RGB-Modell kodieren?

2.2. **Rastergrafiken.**

2.2.1. Wenn man ein Bild als Rastergraphik speichern will, legt man ein quadratisches Raster über das Bild, bestimmt für jedes Rechteck in diesem Raster eine Durchschnittsfarbe und speichert diese.

Aufgabe 2.2.2. Speichere den folgenden Code mit einem Text-Editor deiner Wahl (etwa mit Tigerjython) als ppm-Datei unter einem geeigneten Namen, etwa `bild.ppm`.

```
P3
# "P3" means this is a RGB color image in ASCII
5 6
255
# "255" is the maximum value for each color
# end of header, data below
255 255 255   255 255 255   255 0 0   255 255 255   255 255 255
255 255 255   255 0 0       255 0 0   255 0 0       255 255 255
255 0 0       255 0 0       255 0 0   255 0 0       255 0 0
0 0 255       0 0 255       0 0 255   0 0 255       0 0 255
0 0 255       255 255 255   0 0 255   255 255 255   0 0 255
0 0 255       255 255 255   0 0 255   0 0 255       0 0 255
```

Die Endung `ppm` steht für *portable pixmap* und bezeichnet ein sehr einfaches Grafikformat.

Öffne die Datei mit einem Bildanzeigeprogramm oder einem Bildbearbeitungsprogramm, beispielsweise GIMP (*Gnu Image Manipulation Program*.

Das Bild ist sehr klein, so dass du es stark vergrössern musst.

Schalte das Licht im Fenster auf Gelb und färbe die Tür braun oder manipuliere das Bild in einer anderen Weise.

Aufgabe 2.2.3. Schreibe eine ppm-Datei mit kleinen Abmessungen, so dass das entsprechende Bild alle 2^3 RGB-Farben anzeigt, bei denen jeder der drei RGB-Werte entweder 0 oder 255 ist, vgl. Farbbalkentestbild.

Wenn du willst, kannst du auch alle 3^3 Farben mit RGB-Werten entweder 0 oder 127 oder 255 anzeigen.

2.3. **Vektorgrafiken.**

2.3.1. In der Vektorgrafik stellt man Bilder dar, indem man sie aus relativ einfachen Formen wie Linien, Kreisen, Polygonen, Ellipsen, Splines, Bezierkurven etc. zusammensetzt.

Ein gebräuchliches Format für Vektorgrafiken ist das SVG-Format (*scalable vector graphics*).

Aufgabe 2.3.2. Hier ist eine SVG-Datei.

```
<svg height="200" width="200">
  <polyline points="100,100 100,80
                    120,80  120,120  80,120  80,60
                    140,60 140,140  60,140"
            style="fill:green;stroke:black;stroke-width:2"/>
  <circle cx="80" cy="50" r="20" stroke="blue" stroke-width="5" fill="yellow" />
  <text x="30" y="130" fill="red" transform="rotate(-45 30,110)">SVG is great</text>
</svg>
```

Speichere sie etwa unter dem Namen `beispiel.svg` und zeige sie mit einem Grafikprogramm an, etwa mit Inkscape.

Erstelle selbst eine SVG-Datei mit einer netten Grafik kleinen Grafik (etwa einer Kinderzeichnung). Wenn du zum Beispiel sehen willst, ob du den Befehl `polyline` verstehst: Zeichne damit das Haus vom Nikolaus.

Wenn du kreativ sein willst, lerne etwa von W3Schools SVG-Tutorial: SVG <ellipse>, wie man ein Ellipse zeichnet und stöbere im Tutorial dort herum.

2.4. **Vergleich: Rastergrafik versus Vektorgrafik.**

Aufgabe 2.4.1. Überlege dir Vor- und Nachteile von Rastergrafik versus Vektorgrafik.

Gerne darfst du im Internet stöbern, etwa auf den entsprechenden (englischen oder deutschen) Wikipedia-Seiten; dort gibt es nette Vergleiche mit Bildern.

2.4.2. Abschliessend einige Links:

- SVG-Tutorial Intro
  Am Ende der Seite sind einige Vorteile von SVG aufgelistet.
- Wikipedia: Vector graphics; File formats
  Hier werden Standardformate für Rastergrafiken und Vektorgrafiken aufgezählt. Rechts ist auch ein nettes Bild, das Vorteile von SVG illustriert.

Aufgabe 2.4.3. (freiwillig) Wähle ein Foto in einem komprimierten Format wie beispielsweise JPEG (*Joint Photographic Experts Group*). Öffne es mit einem Bildbearbeitungsprogramm und speichere es wenn möglich als ppm-Datei (mit GIMP geht das jedenfalls).

Um welchen Faktor unterscheiden sich die Grössen der beiden Dateien?

Aufgabe 2.4.4. (freiwillig) Speichere eine SVG-Datei als ppm-Datei oder jpeg und führe einen Grössenvergleich durch.