```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("CTWS2021-1-Template.ipynb")
```

1 1. Variablen

Variablen können wir uns intuitiv zunächst als Namen oder Bezeichnung eines Wertes vorstellen. Mit diesem Namen können wir einen Wert im *Arbeitsspeicherspeicher* des Computers identifizieren. So können wir uns Zwischenergebnisse merken und damit weiter rechnen.

1.1 1.1 Initialisierung und Zuweisung

Durch das `=` Zeichen weisen wir einer *Variablen* auf der linken Seite den Wert des *Ausdrucks* auf der rechten Seite zu: `[Variable] = [Ausdruck]`. Zum Beispiel, weist

```
In [70]: x = 3 + 10
```

den ausgewerteten Wert `3 + 10` also `13` der Variablen `x` zu. Der Ausdruck `3 + 10` wird vor der Zuweisung ausgewertet / berechnet. Es ist äußerst wichtig, dass die den zwischen dem `=` und dem mathematischen $=$ unterscheiden.

$$x = 13$$

bedeutet, dass x gleich 13 ist, wohingegen

```
In [71]: x = 13
```

den Wert der Variablen `x` auf `13` setzt oder genaue die Variable auf einen Speicherbereich verweisen lässt, welcher den Wert `13` enthält. Um das mathematisch auszurücken verwendet man oft $\leftarrow$, also

$$x \leftarrow 13.$$

Dies verdeutlicht, dass es sich um eine *Zuweisung* handelt. Mit

```
In [72]: x = None
```

weisen wir `x` den Wert `None` d.h. 'Nichts' zu. Doch ist dieses 'Nichts' nicht nichts ;). Versuchen wir eine *Variable* zu verarbeiten, die noch nicht initialisiert wurde, so erhalten wir einen Fehler:

```
In [133]: v + 20
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-133-4f5acf813e7b> in <module>
----> 1 v + 20


NameError: name 'v' is not defined
```

Hierbei kommt es zu dem Fehler `name 'v' is not defined`, da die *Variable* `v` noch nicht *initialisiert* wurde.

In `Python` reicht es wenn Sie der *Variablen* einen Wert zuweisen. Sie wird automatisch erzeugt, d.h., *initialisiert*. Besitzt Sie noch keinen Wert so existiert sie auch nicht bzw. ist noch nicht *initialisiert*.

1.2 1.2 Variablen und der Arbeitsspeicher

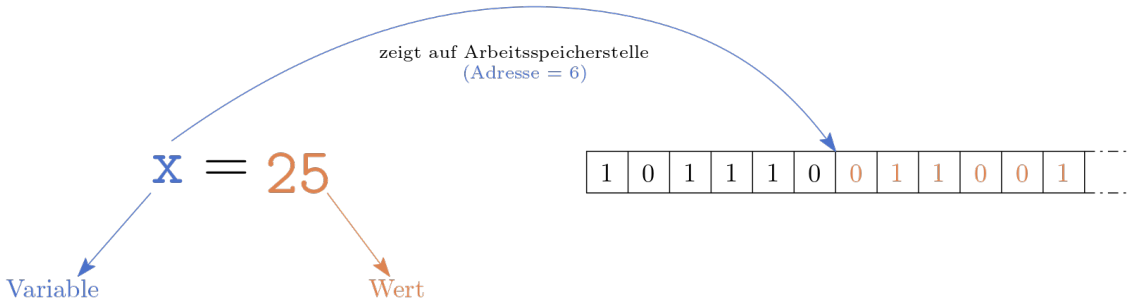
Eine *Variable* können wir als Paar von **Wert** und **Arbeitsspeicheradresse** verstehen. Der **Wert** der *Variablen* steht im *Arbeitspeicher* an einer bestimmten **Arbeitsspeicheradresse**. *Variablen* abstrahieren diesen Zusammenhang, sodass Sie uns die Arbeit mit dem *Arbeitsspeicher* erleichtern.

Mit

```
In [134]: x = 25
```

Wird der **Wert** `25` in den Arbeitsspeicher an eine freie **Speicheradresse** geschrieben. Diese **Adresse** erhält die *Variable* `x`. `x` *zeigt* auf den Speicherbereich in dem der **Wert** `25` steht!

Folgende Abbildung verdeutlicht die Situation:

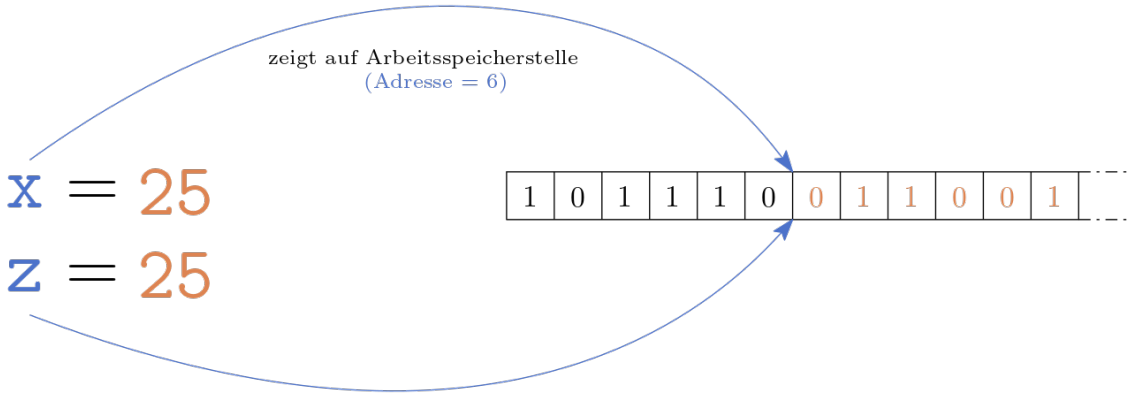


`id`: Mit der *built-in*-Funktion `id` (*Identität)* können Sie sich eine Identifikationsnummer einer Variablen ausgeben lassen. Für zwei *Variablen* ist diese genau dann gleich, wenn deren **Arbeitsspeicheradressen** gleich sind.

```
In [135]: x = 25
          z = 25
          print(id(x))
          print(id(z))
```

```
4450884656
4450884656
```

Sie sehen dass die `id` der Variablen `x` und `z` identisch sind. Ebenso ist ihr Wert identisch. Diese Situation sieht demnach wie folgt aus:



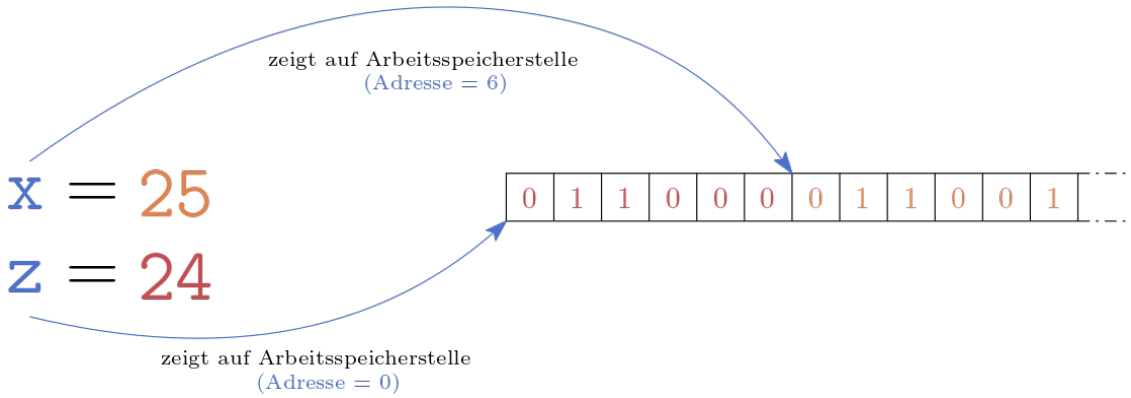
`Python`-erkennt, dass es ausreicht, wenn beide *Variablen* auf den gleichen Speicherbereich zeigen. Wir als Programmierer*innen bekommen davon gar nichts mit. Verändern wir den Wert von `z` dann verändert sich auch deren `id`:

```
In [136]: x = 25
          z = 24
```

```
print(id(x))
print(id(z))
```

```
4450884656
4450884624
```

Die Situation könnte in etwa wie folgt aussehen:



Achtung: Ist die `id` zweier Variablen identisch, so ist auch deren Wert identisch. Jedoch kann der Wert der Variablen identisch sein und deren `id` nicht. Beispiel:

```
In [137]: x = 2131313
          z = 2131313
          print(id(x))
          print(id(z))
```

```
4525046832
4525046448
```

Hmm?? Warum war die `id` beim Wert `25` identisch aber beim Wert `2131313` nicht? Hier kommen wir in tiefen Details von `Python`, welche fürs erste nicht so wichtig sind. Zur Optimierung der Laufzeit legt `Python` alle kleinen ganze Zahlen bei Start der Ausführung in den Speicher, sodass Speicherplatz gespart wird. Das geht jedoch nur für eine endliche Anzahl an Zahlen (deshalb für die ersten k kleinen Zahlen). `2131313` zählt nicht dazu und somit wird der Wert jedesmal neu in den Speicher geschrieben.

Folgender Code berechnet die erste Zahl die nicht bereits bei der Ausführung im Speicher liegt:

```
In [138]: x = 0
          z = 0
```

```
while(id(x) == id(z)):
    x = x + 1
    z = z + 1
x
```

```
Out[138]: 257
```

Wie sieht es mit negativen Zahlen aus?:

```
In [139]: x = 0
          z = 0
          while(id(x) == id(z)):
              x = x - 1
              z = z - 1
          x
```

```
Out[139]: -6
```

1.3 1.3 Veränderung

Wie der Name bereits betont, sind *Variablen* **variabel** und können somit verändert werden. Wir müssen jedoch zwischen zwei Veränderungen einer Variablen `x` unterscheiden:

1. der Veränderung ihrer Wertes `x`
2. der Veränderung ihrer Speicheradresse `id(x)` (die auf den Wert zeigt)

1.3.1 1.3.1 Zuweisung eines neuen Werts

Eine *Variable* kann immer nur einen **Wert** bzw. auf einen bestimmten Speicherbereich *zeigen*. Weisen wir einer *Variablen* erneut einen **Wert** zu, wird dieser **Wert** in den Speicher an eine freie **Adresse** geschrieben und die **Adresse** der Variablen auf jene neue **Adresse** gesetzt.

```
In [140]: half = 1/2
          print(f'value of half = {half}')
          print(f'id of half = {id(half)}')

          x = 25
          print(f'value of x = {x}')
          print(f'id of x = {id(x)}')
```

```
x = 24
print(f'value of x = {x}')
print(f'id of x = {id(x)}')
```

```
value of half = 0.5
id of half = 4525046192
value of x = 25
id of x = 4450884656
value of x = 24
id of x = 4450884624
```

Veränderungen der einen *Variablen* haben keinen Effekt auf die **Adresse** bzw. *Identität* `id` *anderer* Variablen*.

```
In [141]: print(f'value of half = {half}')
          print(f'id of half = {id(half)}')
```

```
value of half = 0.5
id of half = 4525046192
```

Verändern wir *Variablen* nicht so behalten ihre **Adresse** über das gesamte Notebook hinweg.

1.3.2 1.3.2 Zuweisung einer neuen Adresse

Weisen wir einer Variblen `x` eine andere Variable `y` zu, so ändern wir die **Adresse** von `x` auf jene von `y`. Das heißt, nach der *Zuweisung* zeigen beide Variablen auf den gleichen Speicherbereich und damit auf den gleichen **Wert**.

```
In [142]: x = 2131313
          y = 10
          z = 2131313

          print(f'value of x = {x}')
          print(f'id of x = {id(x)}')

          print(f'value of y = {y}')
          print(f'id of y = {id(y)}')

          print(f'value of z = {z}')
          print(f'id of z = {id(z)}')
```

```
value of x = 2131313
id of x = 4525046480
value of y = 10
id of y = 4450884176
value of z = 2131313
id of z = 4525047280
```

```
In [143]: y = x

          print(f'value of x = {x}')
          print(f'id of x = {id(x)}')

          print(f'value of y = {y}')
          print(f'id of y = {id(y)}')

          print(f'value of z = {z}')
          print(f'id of z = {id(z)}')
```

```
value of x = 2131313
id of x = 4525046480
value of y = 2131313
id of y = 4525046480
value of z = 2131313
id of z = 4525047280
```

Aufgabe 1. Betrachten Sie folgenden Code. Geben Sie an welche *Variablen* die gleiche `id` besitzten und welche nicht. Welche Variablen besitzten den gleichen Wert und welche nicht?

```
In [144]: x = 1.1231
          y = 1.1231
          z = x
          k = z + 1
```

Type your answer here, replacing this text.

Die Variablen `x` und `z` haben die gleiche `id` und somit auch den gleichen Wert. `y` besitzt den gleichen Wert wie `x` und `z` jedoch eine andere `id`. Der Wert wie auch die `id` der Variable `k` ist verschienden von allen anderen.

```
In [145]: print(id(x)) # SOLUTION
          print(id(y)) # SOLUTION
```

```
print(id(z)) # SOLUTION
print(id(k)) # SOLUTION
```

```
4525046576
4525046544
4525046576
4525045552
```

Aufgabe 2. Hat ein Wert wie zum Beispiel `25` eine `id`? Wenn ja warum und wenn nein warum nicht?

Type your answer here, replacing this text.

Auch ein Wert der keiner Variablen zugewiesen wurde hat eine `id` denn auch dieser Wert steht irgendwo im Arbeitspeicher.

```
In [146]: id(25)
```

```
Out[146]: 4450884656
```

Aufgabe 3. Weshalb ist der Wert der Variablen `x` nach der Ausführung des folgenden Codes noch immer `42`. Beschreiben Sie was im **Arbeitsspeicher** passiert und auf was die Variablen zeigen.

```
In [147]: x = 24
          z = 42
          x = z
          z = 33
          x
```

```
Out[147]: 42
```

Type your answer here, replacing this text.

1. `x` erhält den Wert `24` durch die Adresse die auf `24` im Speicher zeigt.
2. `z` erhält den Wert `42` durch die Adresse die auf `42` im Speicher zeigt.
3. Die Speicheraddresse von `x` wird auf die von `z` geändert, damit zeigt `x` auf `42`.
4. Die Speicheraddresse von `z` wird auf jene geändert die `33` den Wert `33` enthält, diese Adresse ist von der die `42` enthält unterschiedlich. `x` bleibt unverändert.

1.3.3 1.3.3 Seiteneffekte

Wie bereits erwähnt: Verändern wir eine *Variable* so können wir dadurch nicht die **Adresse** / *Identität* `id` einer anderen Variablen ändern! Wie verhält es sich jedoch mit dem **Wert** einer *Variablen*?

Die Antwort ist etwas komplizierter und ist erst dann begreiflich wenn wir das Thema Datentypen besprechen. Dennoch versuchen wir unser Glück:

Eine Variable kann nicht nur einen einzelnen atomaren **Wert** wie eine Zahl enthalten, sondern auch einen **Wert** der sich aus anderen **Werten** zusammensetzt. Zum Beispiel:

```
In [148]: x = [1,2,3,4,5]
          x
```

```
Out[148]: [1, 2, 3, 4, 5]
```

Der Variablen `x` weisen wir hierbei eine sog. *Liste* `list` zu, also eine geordnete Menge an Zahlen. In unserem Fall besteht die Liste und somit `x` aus den Werten `1, 2, 3, 4` und `5`.

Um auf einen bestimmten **Wert** der Liste zuzugreifen brauchen wir seinen Index. Zum Beispiel liefert uns der Index `1` den Wert `2`:

```
In [149]: x[1]
```

```
Out[149]: 2
```

Wie sieht das nun im Speicher aus??? Welche Adresse hat `x` und wie sieht der Speicher an der Adresse von `x` aus? Der Aufruf

```
In [150]: id(x)
```

```
Out[150]: 4518658688
```

Liefert uns eine `id`, allerdings lieft uns der Aufruf

```
In [151]: id(x[1])
```

```
Out[151]: 4450883920
```

ebenfalls eine (andere) `id`.

```
In [152]: id(x[2])
```

```
Out[152]: 4450883952
```

Eine Liste `list` mit n Elementen besteht in `Python` aus n Adressen. Jede dieser Adressen zeigt auf den Wert des Listenelements. Das heißt, eigentlich sieht unsere Liste `x` wie folgt aus:

```
[id(x[0]), id(x[1]), id(x[2]), id(x[3]), id(x[4])]
```

Doch `Python` vereinfacht uns den Umgang mit Listen und verbirgt diese Tatsache geschickt.

Was aber passiert mit `x` wenn wir eines seiner Listenelemente verändern? Hier wird es spannend:

```
In [153]: print(f'value of x = {x}')
          print(f'id of x = {id(x)}')

          print(f'value of x[1] = {x[1]}')
          print(f'id of x[1] = {id(x[1])}')

          print()

          x[1] = -10
          print(f'value of x = {x}')
          print(f'id of x = {id(x)}')

          print(f'value of x[1] = {x[1]}')
          print(f'id of x[1] = {id(x[1])}')
```

```
value of x = [1, 2, 3, 4, 5]
id of x = 4518658688
value of x[1] = 2
id of x[1] = 4450883920

value of x = [1, -10, 3, 4, 5]
id of x = 4518658688
value of x[1] = -10
id of x[1] = 4525047696
```

Die Adresse von `x` ändert sich nicht!!! Es ändert sich nur die Adresse von `x[1]`!!! Mit anderen Worten durch die *Zuweisung* von `x[1] = -10` wird keine neue Liste im Speicher angelegt sondern nur ein neues Element!

Warum? Listen können groß werden und würden wir bei jeder Änderung eines Listenelements die gesamte Liste im Speicher kopieren, wäre das zu teuer was die Laufzeit angeht.

Dieses Verhalten hat jedoch Konsequenzen!

Aufgabe 4. Blicken Sie auf folgenden Code. Weisen Sie `z` eine Liste zu die eine andere `id` aber den gleichen **Wert** wie `y` hat. Lassen Sie sich zunächst `y` **nicht** ausgeben!

```
In [198]: x = [1, 2, 3]
          y = [x, x, [1,2,3]]
          x[0] = -10
```

```
In [199]: z = [[-10, 2, 3], [-10, 2, 3], [1,2,3]] # SOLUTION
```

```
In [ ]: grader.check("q4")
```

Solche Veränderungen eines Wertes einer Variablen durch die Veränderung eines Werts einer anderen Variablen, nennen wir *Seiteneffekt*.

2 2. Ausdrücke

Jedes Programm, bzw. jeder Algorithmus besteht aus vielen *Ausdrücken*. Ein *Ausdruck* beschreibt wie Daten (die Eingabe) verarbeitet werden sollen. Durch eine *Abfolge von Ausdrücken* werden Daten immer und immer weiter verarbeitet. Die Initialisierung und Zuweisung einer *Variablen*

```
In [158]: x = 42
```

ist ein Ausdruck (engl. Expression). Die Multiplikation

```
In [159]: 3 * 5
```

```
Out[159]: 15
```

ist beispielsweise ein Ausdruck der zwei Dezimalzahlen multipliziert. Der Ausdruck besteht aus dem Symbol `*` und zwei nummerischen Werten (Zahlen).

```
[Zahl] * [Zahl]
```

Die Multiplikation wird durch den Computer, genauer die CPU berechnet. `3 * 5` ergibt `15`. Die Zeichenfolge muss *syntatisch* korrekt sein, damit diese auch als Ausdruck vom Computer (bzw. Interpreter) verstanden wird. Die Zeichenfolge

```
In [160]: 3 * * 5
```

```
  File "<ipython-input-160-a03b9e7879f4>", line 1
    3 * * 5
        ^
SyntaxError: invalid syntax
```

ist kein *syntaktisch* korrekter `Python`-Ausdruck, was Ihnen die Fehlermeldung auch zu verstehen gibt. Die gewählte Programmiersprache bestimmt welche Zeichenfolge *syntaktisch* korrekt ist. Bereits kleiner Änderungen an der *Syntax* können zu einer neuen *Bedeutung (Semantik)* führen. In Python die Zeichenkette

```
In [161]: 3 ** 5
```

```
Out[161]: 243
```

ein *syntaktisch* valider Ausdruck und *bedeutet* 3^5 was 243 ergibt.

Ausdrücke können sich aus weiteren *Ausdrücken* zusammensetzten.

```
In [162]: 3 + 5
```

```
Out[162]: 8
```

ist ein Ausdruck

```
In [163]: 4 * 6
```

```
Out[163]: 24
```

ebenfalls und

```
In [164]: 3 + 5 - 4 * 6
```

```
Out[164]: -16
```

auch.

2.0.1 2.1 Arithmetische Operatoren

Die Multiplikation `*` wie auch die Potenz `**` bezeichnen wir als *arithmetische Operatoren*, da sie numerische Werte (Zahlen) verarbeiten. Es gibt jedoch noch eine ganze Reihe von weiteren *arithmetische Operatoren*:

Operator	Beschreibung	Beispiel	Bedeutung
`+`	Addition	`3 + 4`	$3+4$
`-`	Subtraktion	`3 - 4`	$3-4$
`*`	Multiplikation	`3 * 4`	$3 \cdot 4$
`/`	Division	`3 / 4`	$3/4$
`**`	Potenzierung	`3**4`	3^4
`//`	ganzzahlige Division	`3 // 4`	$\lfloor 3/4 \rfloor$
`%`	Modulo	`10 % 4`	$10 - (4 \cdot \lfloor 10/4 \rfloor)$

Jeder dieser Operatoren `op` erwartet zwei Zahlen, eine links und eine rechts vom jeweiligen Operator. Die Bedeutung der Modulo-Operation sieht kompliziert aus doch bedeutet dies schlicht, dass der Rest `10 % 4` der ganzzahlige Rest der *Restwertdivision* ist. Die ganzzahlige Division rundet das Ergebnis der Division auf die nächst kleinere ganze Zahl (Integer).

```
In [165]: -2 // 3
```

```
Out[165]: -1
```

```
In [166]: -2 / 3
```

```
Out[166]: -0.6666666666666666
```

```
In [167]: 2 // 3
```

```
Out[167]: 0
```

```
In [168]: 2 / 3
```

```
Out[168]: 0.6666666666666666
```

```
In [4]: 14 % 5
```

```
Out[4]: 4
```

```
In [7]: 14 - (14 // 5)*5
```

```
Out[7]: 4
```

```
In [15]: x = 12314
         y = 7

         x % y == x - (x // y) * y
```

```
Out[15]: True
```

Die Addition und Subtraktion von Fließkommazahlen kann zu merkwürdigen Ergebnissen führen. Dazu später mehr:

```
In [36]: 0.1 + 0.2
```

```
Out[36]: 0.30000000000000004
```

```
In [37]: 0.1 + 0.2 == 0.3 # False!?
```

```
Out[37]: False
```

```
In [39]: 0.1 + 0.4
```

```
Out[39]: 0.5
```

```
In [40]: 0.1 + 0.4 == 0.5 # True
```

```
Out[40]: True
```

Aufgabe 5. Finden Sie heraus wie `Python` arithmetische Operationen priorisiert. Zum Beispiel ist

```
3 + 3 * 5**4
```

gleich

$$3 + 3 \cdot 5^4$$

oder, zum Beispiel,

$$((3 + 3) \cdot 5)^4?$$

Notieren sie die Prioritäten von der höchsten zur niedrigsten Priorität.

Type your answer here, replacing this text.

Der obige Ausdruck entspricht $3+3\cdot 5^4 = 3+(3\cdot(5^4))$ Prioritäten der arithmetischen Operationen in `Python` sind so wie wir sie gewohnt sind:

`**`, `*` bzw. `/`, `+` bzw. `-`.

Aufgabe 6. Eine Person dreht sich um `x` Grad im Kreis. Bestimmen Sie wie viele abgeschlossene Umdrehungen `cycles` sie gemacht hat. Wie viel Grad `rest` haben für die letzte volle Umdrehung gefehlt?

```
In [41]: x = 3141 # Grad
         cycles = x // 360 # SOLUTION
         rest = x % 360 # SOLUTION
```

```
In [ ]: grader.check("q6")
```

Aufgabe 7. Berechnen Sie folgenden Ausdruck

$$(5 - 9^2 \cdot 3)^3$$

und weisen Sie das Ergebnis der Variable `x` zu.

```
In [169]: x = (5 - 9**2 * 3)**3 # SOLUTION
```

```
In [ ]: grader.check("q7")
```

Aufgabe 8. Berechnen Sie folgenden Ausdruck

$$10^6 - 10^{-10}$$

und

$$10^6 - 10^{-11}$$

und weisen Sie das erste Ergebnis der Variable `x` und das zweite Ergebnis der Variable `y` zu. Lassen Sie sich `x` und `y` ausgeben. Was beobachten Sie?

```
In [171]: x = 10**6 - 10**(-10) # SOLUTION
          x
```

```
Out[171]: 999999.9999999999
```

```
In [172]: y = 10**6 - 10**(-11) # SOLUTION
          y
```

```
Out[172]: 1000000.0
```

```
In [ ]: grader.check("q8")
```

Aufgabe 9. Berechnen Sie die Wurzel aus 2 und weisen Sie das Ergebnis der Variable `z` zu. Repräsentiert der Wert der Variable wirklich $\sqrt{2}$? Warum bzw. warum nicht?

```
In [175]: z = 2**(0.5) # SOLUTION
```

```
In [ ]: grader.check("q9")
```

2.0.2 2.2 Vergleichsoperatoren

Objekte können über Vergleichsoperatoren miteinander verglichen werden. Das Ergebnis ist ein Boolscher Wert.

Operator	Beschreibung
`x == y`	ist `x` gleich `y`?
`x != y`	ist `x` ungleich `y`?
`x > y`	ist `x` größer als `y`?
`x >= y`	ist `x` größer oder gleich `y`?
`x < y`	ist `x` kleiner `y`?
`x <= y`	ist `x` kleiner gleich `y`?
`x is y`	ist `x` das selbe Objekt wie `y`?

Erneut ist `Python` hier ein wenig speziell indem es die mathematische Schreibweise $0 < x < 5$ anstatt $0 < x \ \wedge \ x < 5$ zulässt. Dies erhöht die Lesbarkeit, da wir solche Verkettungen von Vergleichsoperatoren gewohnt sind.

```
In [177]: 5 > 7
```

```
Out[177]: False
```

```
In [178]: 5 < 7
```

```
Out[178]: True
```

```
In [179]: 5 <= 7
```

```
Out[179]: True
```

```
In [180]: 5 == 5
```

```
Out[180]: True
```

```
In [181]: 5 == -5
```

```
Out[181]: False
```

```
In [46]: 0.1 + 0.2 == 0.3 # Achtung!!! Gleichheit auf Fließkommazahlen ist ungeeignet.
```

```
Out[46]: False
```

```
In [47]: epsilon = 1e-10
         0.3 - epsilon < 0.1 + 0.2 < 0.3 + epsilon # Besser einen Bereich prüfen
```

```
Out[47]: True
```

Der Ausdruck

```
In [182]: 5 < 7 < 10
```

```
Out[182]: True
```

hat die gleiche *Bedeutung* wie der Ausdruck

```
In [183]: 5 < 7 and 7 < 10
```

```
Out[183]: True
```

Vergleichsoperatoren können auch auf anderen Datentypen als numerische Werte (ganze Zahlen `int`, Fließkommazahlen `float`) definiert sein. So können wir in `Python` auch Zeichenketten `str` mit den Vergleichsoperatoren lexikographisch vergleichen:

```
In [184]: 'Anna' < 'Emma' # True
```

```
Out[184]: True
```

Aufgabe 10. In der folgenden Zelle berechnen wir

$$\sqrt{2} + 2 - \sqrt{2} - 2 = 0$$

jedoch erhalten wir nicht 0 sondern eine sehr kleine negative Zahl. Schreiben Sie einen Ausdruck der prüft ob `x` fast gleich `0` ist.

```
In [48]: x = 2**(0.5) + 2 - 2**(0.5) - 2
         x
```

```
Out[48]: -2.220446049250313e-16
```

```
In [186]: epsilon = 0.00001 # SOLUTION
          -epsilon < x < epsilon # SOLUTION
```

```
Out[186]: True
```

Was ist der Unterschied zwischen `is` und `==`? Die Funktion `id(x)` gibt eine eindeutige *Id* des Objekts `x` zurück. Diese ist vergleichbar mit der Arbeitsspeicheradresse an der das Objekt `x` im Speicher beginnt. `x`

`is y` ist genau dann `True` wenn diese *Id* für beide Objekte gleich ist, wenn also `id(x) == id(y)`. Ist diese *Id* nicht gleich aber an beiden unterschiedlichen Stellen im Speicher liegt der gleiche Wert dann ist `x is y` gleich `False` aber `x == y` ist `True`.

Aufgabe 11. Führen Sie folgenden Code aus. Was folgt aus der Ausgabe die der Code generiert?

```
In [187]: x = 1
          y = 1
```

```
In [188]: print(x is y)
          print(id(x) == id(y))
          print(x == y)
```

```
True
True
True
```

Type your answer here, replacing this text.

Die Variablen `x` und `y` zeigen auf den gleichen Speicherbereich.

Aufgabe 12. Führen Sie folgenden Code aus. Was folgt aus der Ausgabe die der Code generiert?

```
In [189]: x = 1
          y = 1.0
          print(x is y)
          print(id(x) == id(y))
          print(x == y)
```

```
False
False
True
```

Type your answer here, replacing this text.

Die **Werte** der Variablen `x` und `y` stehen an verschiedenen Stellen im Speicher. Sie werden von der Vergleichsoperation `==` als gleich angesehen doch sind `x` und `y` nicht identisch.

2.0.3 2.3 Logische (Boolsche) Operatoren

Der obige *Ausdruck*

```
In [190]: 5 < 7 and 7 < 10
```

```
Out[190]: True
```

besteht aus den *Ausdrücken* `5 < 7` und `7 < 10` die dem logischen `and`-Operator verknüpft werden. Dieser erwartet auf der linken und rechte Seite jeweils einen Wahrheitswert (*boolschen Ausdruck*). *Vergleichsoperatoren* liefern sind *boolsche Ausdrücke*.

```
In [191]: exp1 = True
          exp2 = 5 < 6
          exp1 and exp2
```

```
Out[191]: True
```

ergibt genau dann `True` wenn die Auswertung von `exp1` und `exp2` jeweils `True` ergeben. Es gibt drei logische Operatoren:

Operator	Beschreibung
`not x`	ist `True` genau dann wenn `x == False`.
`x and y`	ist `True` genau dann wenn `x == True` und `y == True`.
`x or y`	ist `True` genau dann wenn `x == True` oder `y == True`.

Aufgabe 13. Angenommen Sie haben zwei logische Ausdrücke `exp1` und `exp2`. Schreiben Sie einen logischen Ausdruck der genau dann `True` ergibt wenn entweder `exp1` oder `exp2` `True` sind jedoch nicht beide (exklusives Oder).

```
In [51]: expr1 = True
         expr2 = False
         expr1 and not expr2 or not expr1 and expr2 # SOLUTION
```

```
Out[51]: True
```

2.0.4 2.4 Bitoperatoren

Der Vollständigkeit listen wir auch noch die *Bitoperatoren* auf. Diese sind dazu vorgesehen um den Wert in *Binärdarstellung* zu manipulieren.

Jeder Wert egal ob Zahl, Zeichen, Bild, Ton wir als Binärcode im Speicher abgelegt. *Bitoperatoren* nehmen diesen Binärwert und verarbeiten bzw. kombinieren ihn genau wie die Addition zwei Zahlen in der Dezimalschreibweise verarbeitet.

Dabei wird jedes Bit des einen Werts mit dem *Bit* des anderen Werts kombiniert. Zum Beispiel `5 & 4` führt eine führ jedes Bit die `and` Operation aus. Das nennen wir Verundung.

```
In [197]: 5 & 4
```

```
Out[197]: 4
```

führt eine Verundung der Binärzahlen $5_{10} = 101_2$ mit $4_{10} = 100_2$ durch und ergibt demnach 100_2, was wiederum gleich 4_{10} ist.

Operator	Beschreibung	Beispiel	Ergebnis
`x & y`	Verundung von `x` mit `y`	`10 & 3`	`2`
`x | y`	Veroderung von `x` mit `y`	`10 | 3`	`11`
`x ^ y`	exklusive Veroderung von `x` mit `y`	`10 ^ 3`	`9`
`x << y`	Bitverschiebung von `x` um `y` Stellen nach links	`8 << 3`	`64`
`x >> y`	Bitverschiebung von `x` um `y` Stellen nach rechts	`8 >> 2`	`2`

Weshalb ist `10 ^ 3` gleich `9`? Nun $10_{10} = 01010_2$ und $3_{10} = 00011_2$. Das exklusive oder bedeutet gesprochen **entweder oder**, d.h. ein Bit wird zur 1 genau dann wenn das Bit der einen Zahl gleich 1 und das Bit der anderen Zahl gleich 0 ist oder genau anders herum. Dies ergibt $01001_2 = 9_{10}$.

Für ganze Zahlen entspricht die Bitverschiebung nach rechts um ein Bit der Multiplikation mit 2. Die Verschiebung nach rechts um ein Bit hingegen der *ganzzahligen Division* durch 2. Deshalb ist `8 << 3` gleich $8 \cdot 2 \cdot 2 \cdot 2 = 8 \cdot 2^3 = 64$ und `8 >> 2` gleich $\lfloor 8 \cdot 2^{-2} \rfloor = 2$.

Aufgabe 14. Berechnen Sie $x \leftarrow 2^{10}$ unter Verwendung der Bitverschiebung.

```
In [53]: x = 2 << 9 # SOLUTION
```

```
In [ ]: grader.check("q14")
```

Aufgabe 15. Angenommen die Variablen `x` und `y` haben jeweils eine ganze Zahl als **Wert**, sodass beide dieser Zahlen in der Binärdarstellung an keiner Stelle ein gleiches Bit besitzten. Zum Beispiel:

$$x \leftarrow 5_{10} = 0101_2 \text{ und } y \leftarrow 10_{10} = 1010_2.$$

Mit welcher Bitoperation könnten Sie die Addition `x + y` berechnen? Testen Sie diesen Sachverhalt.

Type your answer here, replacing this text.

Die Veroderung würde die Addition realisieren. Zum Beispiel:

```
In [56]: x = 5  # SOLUTION
         y = 10 # SOLUTION
         x | y # SOLUTION
```

```
Out[56]: 15
```

To double-check your work, the cell below will rerun all of the autograder tests.

```
In [ ]: grader.check_all()
```

2.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

In [ ]:
```
grader.export(pdf=False, force_save=True)
```