

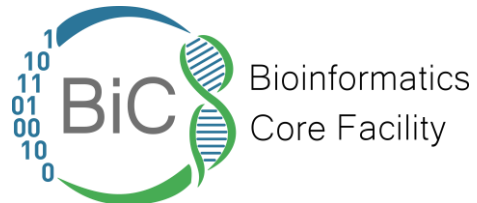
# Programmieren für Einsteiger

- Arbeiten mit numerischen Daten in Python -

Tag 2

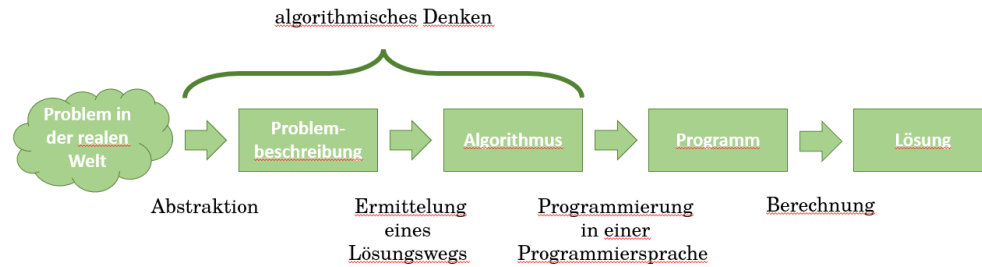
27.07.2022

Emanuel Barth, Daria Meyer



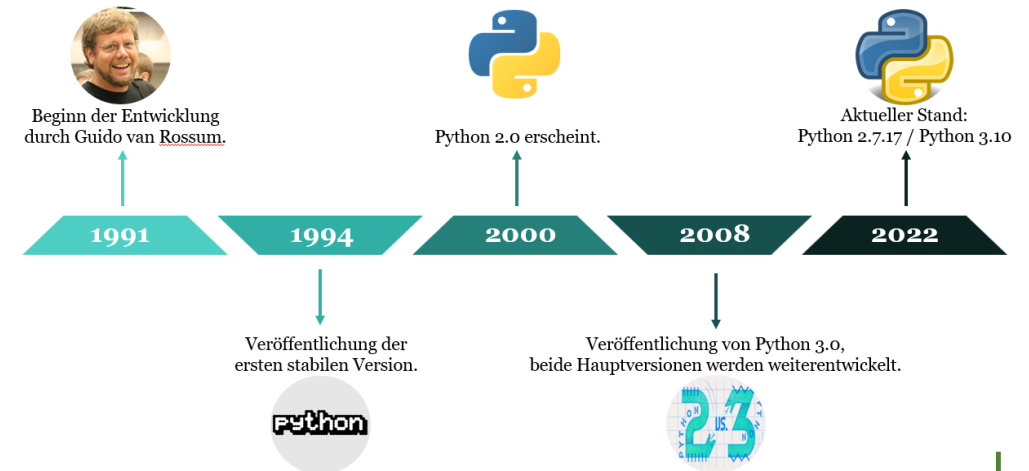
# Kurze Wiederholung Tag 1

## Was ist Programmieren?



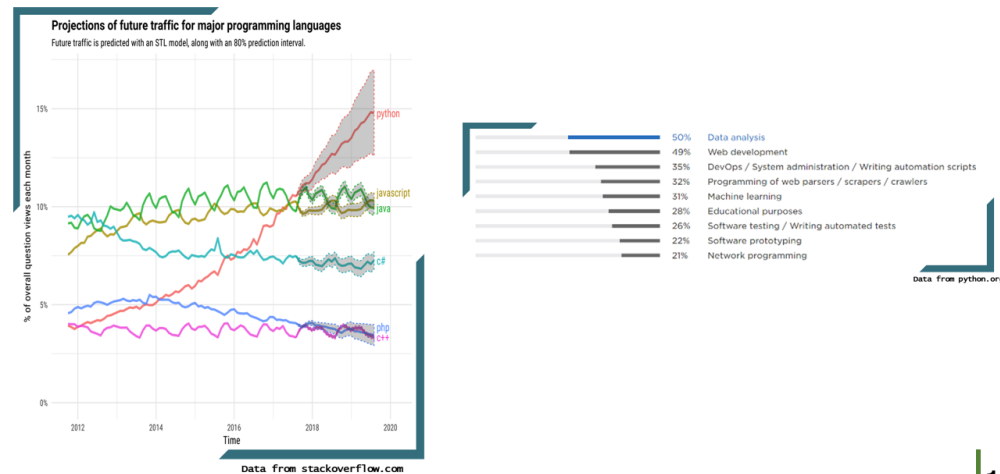
10

## Entwicklungsgeschichte



14

## Warum Python?



18

## Python Code Beispiel

```
import os
import sys

#This Scripts reads a Fasta file and parses it into a dictionary.

fastaDict = {}
fastaFile = open('/home/emanuel/test.fna')
header = ''
sequence = ''

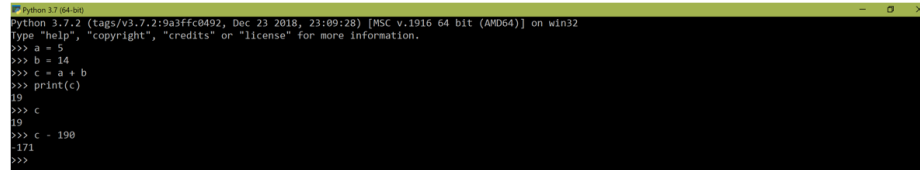
for line in fastaFile:
    if line[0] == '>':
        if len(header) > 0:
            fastaDict[header] = sequence
            sequence = ''
            header = line
        else:
            sequence += line

    fastaDict[header] = sequence
fastaFile.close()
```

20

# Kurze Wiederholung Tag 1

## Die Pythonshell

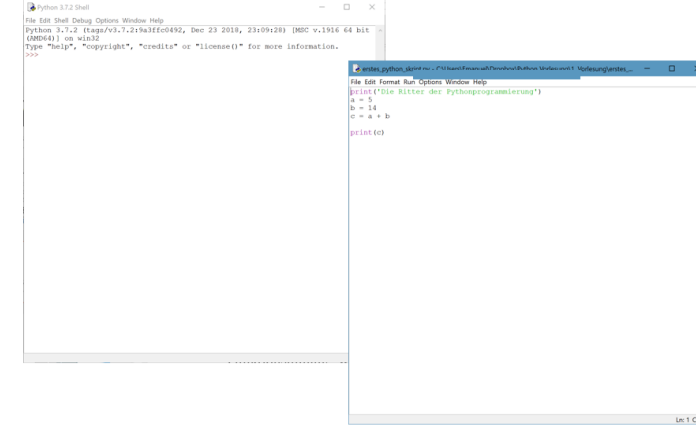


```
Python 3.7.2 Shell
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 5
>>> b = 14
>>> c = a + b
>>> print(c)
19
>>> c
19
>>> c - 190
-171
>>>
```

- Durch die Pythonshell kann man interaktiv mit dem Pythoninterpreter arbeiten.
- Die Pythonshell eignet sich um Codeabschnitte oder einzelne Funktionen ausgiebig zu testen.
- Neben der Standardshell gibt es auch Varianten mit einem größerem Funktionsumfang, wie z. B. IPython (<http://ipython.org/>) → Dazu später mehr.

24

## Die IDLE

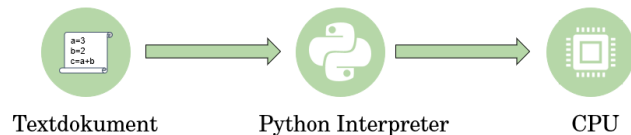


- Die IDLE ist der mitgelieferte Standardeditor von Python mit integrierter Pythonshell.
- Über den Editor lassen sich Pythonskripte erstellen, modifizieren und direkt ausführen.

25

## Pythonskripte

- Ein Pythonskript ist letztendlich nur eine einfache Textdatei, welche (hoffentlich) ausführbaren Pythoncode enthält.
- Neben Python Code können auch Kommentare, markiert durch ein #-Symbol, im Skript stehen.
- Pythonskripte können mit einem beliebigen, einfachen Texteditor erstellt und verändert werden und sind üblicherweise mit der Dateiendung **.py** gekennzeichnet.
- Übergibt man einem Pythoninterpreter eine Datei, so versucht er diese auszuführen, in dem er den darin enthaltenen Code (Text) interpretiert.



26

# Kurze Wiederholung Tag 1

## Variablen in Python

- Grundlegend haben wir einfach einen (Variablen)Namen, welchem wir einen Wert (oder Daten) zuweisen:

Variablenname = Variablenwert

- Jeder Variablenwert besitzt auch einen bestimmten Variablentyp, z. B.:
  - 5 ist eine Ganzzahl (*Integer*)
  - 'Python' ist ein Text (*String*)
- Daher legt der Wert einer Variable auch deren Typ in Python fest:

Variablenname = Variablenwert  
↑  
Variablentyp

- Man kann auch sagen:

Variablenname = Daten  
↑  
Datentyp

31

## Typfunktionen

- Jeder Variablentyp hat eigene Typfunktionen, welche zur Verarbeitung seiner Variablenwerte nützlich sind, für andere Variablentypen aber nicht sinnvoll wären.
- Ändert sich der Typ einer existierenden Variable, weil sich ihr Variablenwert verändert, dann ändern sich auch automatisch die Typfunktionen, welche die Variable kennt.

```
zahlen = [1, 2, 3, 4]
zahlen = zahlen - 5
TypeError: unsupported operand type(s) for -: 'list' and 'int'

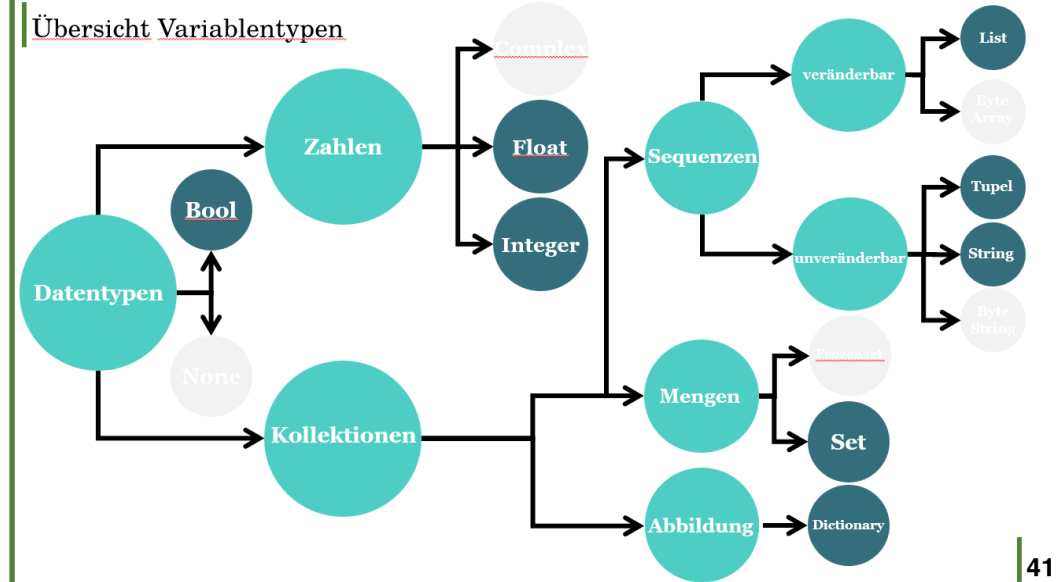
zahlen = 19
zahlen = zahlen - 5

zahlen.append(5)
AttributeError: 'int' object has no attribute 'append'
```

Variablenname = Variablenwert  
↑  
Typfunktionen

62

## Übersicht Variablentypen



41

## Typumwandlung

- Da Variablen **dynamisch** Typisiert sind, werden Ausdrücke **stark** Typisiert, d. h. bei Uneindeutigkeiten gibt der Pythoninterpreter einen Fehler aus, weil er nicht wissen kann, welchen Datentyp ihr als Ergebnis erwartet.  
→ „Explicit is better than implicit.“

```
5 + '14'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Durch Typumwandlungsfunktionen (auch Casting Funktionen genannt) lassen sich Variablentypen ineinander umwandeln:  
`int()`, `float()`, `bool()`, `set()`, `tuple()`, `list()`

```
5 + int('14')
19
str(5) + '14'
'514'
```

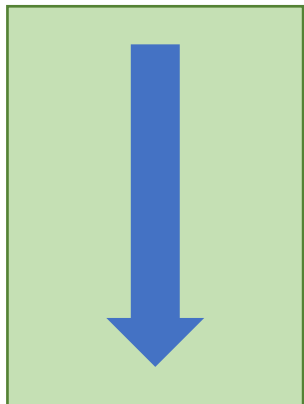
76

# Lernziele für Heute

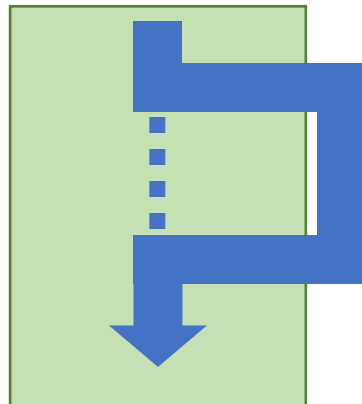
- Du bist in der Lage (komplexe) logische Vergleiche und Bedingungen zu formulieren und anzuwenden.
- Du kannst nicht-lineare Programme mithilfe von Kontrollstrukturen verstehen und selbst entwickeln.
- Du weißt warum bestimmte Arten von Fehlern (Exceptions) auftreten können und wie man diese abfangen kann.
- Du kannst erklären wofür selbstdefinierte Funktionen gut sind und wie man diese implementieren kann.

# Lineare vs. nicht-lineare Programme

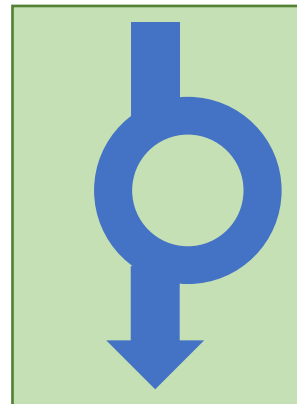
- Bis jetzt wurden alle unsere Programme Zeile für Zeile (von "oben nach unten") abgearbeitet, so dass jede Anweisung genau einmal ausgeführt wurde (wenn unser Code fehlerfrei war).
- Oft möchte man aber Programme schreiben, welche flexible auf eine Eingabe reagieren können und unter bestimmten Voraussetzungen einige Anweisungen überspringen.
- Um uns Schreibarbeit zu sparen wäre es praktisch bestimmte Programmabschnitte wiederholt ausführen zu lassen.
- Im Idealfall sollte unser Programm auch auf mögliche Fehler reagieren können die z. B. durch fehlerhafte Dateneingaben auftreten können.



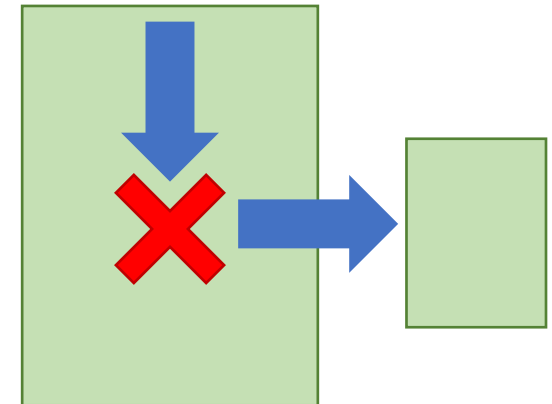
lineares Programm



Programmverzweigung



Programmschleife



Fehlerbehandlung

I'LL BE IN YOUR CITY TOMORROW  
IF YOU WANT TO HANG OUT.

BUT WHERE WILL YOU BE IF  
I DON'T WANT TO HANG OUT?!

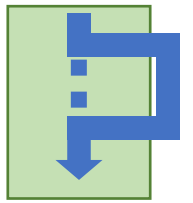
YOU KNOW, I JUST  
REMEMBERED I'M BUSY.



WHY I TRY NOT TO BE  
PEDANTIC ABOUT CONDITIONALS.

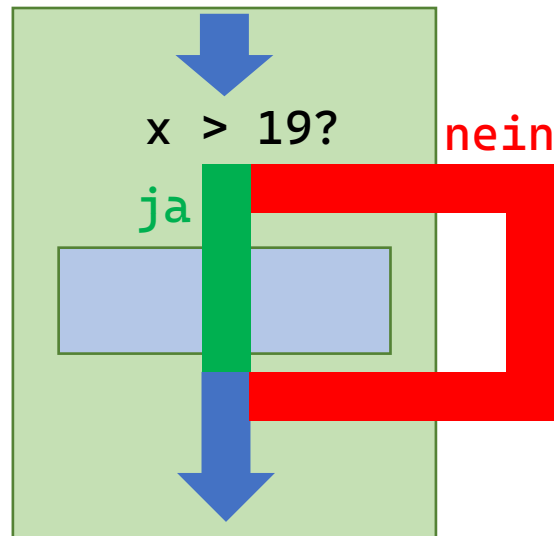
# Kontrollstrukturen

# Programmverzweigung mittels Bedingungen definieren



Programmverzweigung

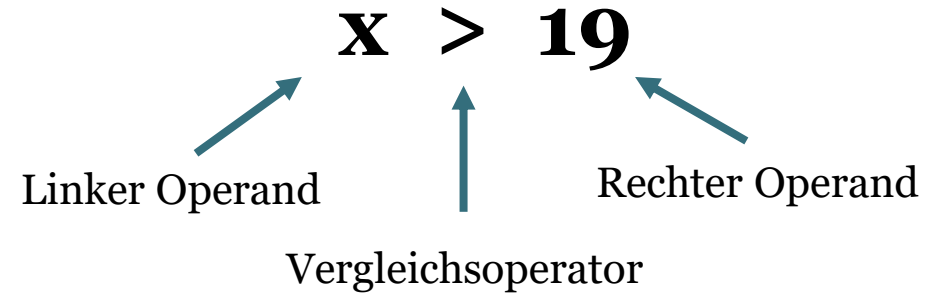
- Um eine Programmverzweigung zu ermöglichen müssen wir im Prinzip zwei Dinge definieren:
  - Wo soll die Verzweigung auftreten?
  - Wann soll das Programm die Verzweigung nutzen?
- Verzweigungen im Programm werden durch Bedingungen formuliert, deren Grundlage wiederum *aussagenlogische Vergleiche* sind.  
Bspw.: "Wenn der Wert der Variable  $x$  größer als 19 ist, dann führe die folgenden Code Zeilen aus, ansonsten überspringe sie."





# Vergleiche

- Die einfachsten Vergleiche bestehen immer aus mindestens drei Elementen:



- Übersicht der geläufigsten Vergleichsoperatoren:

Operator	Erklärung
<	Kleiner
<=	Kleiner gleich
>	Größer
>=	Größer gleich
==	Gleich
!=	Ungleich
in	Zugehörigkeit
not in	Nicht-Zugehörigkeit

# Vergleiche

- Das Ergebnis eines Vergleichs ist immer einer der beiden Wahrheitswerte `True` oder `False` (auch bool'sche Werte genannt).
- Zahlen gleichen und unterschiedlichen Typs, können beliebig miteinander verglichen werden:

```
5 < 19
True
36 != 36.0
False
```

- Daten unterschiedlichen Typs können nur auf (Un)gleichheit überprüft werden, auf andere Weise lassen sie sich nicht sinnvoll miteinander vergleichen:

```
119 < 'Monty'
TypeError: '<' not supported between instances of 'int' and 'str'
119 == 'Monty'
False
```

# Vergleiche

- Sequenzen gleichen Typs werden elementweise miteinander verglichen:

```
[1,2,3] == [1,2,1]  
False  
[1,2,3] >= [1,2,1]  
True
```

- Bei Strings werden die einzelnen Buchstaben elementweise verglichen, entsprechend der Reihenfolge im Alphabet:

```
'perl' < 'python'  
True  
'a' > 'B'  
True
```

- Vergleichsoperatoren können beliebig lang miteinander verkettet werden, wobei jeder Vergleich von links nach rechts abgearbeitet wird:

```
1 < 2 >= 5  
False
```

# Zugehörigkeit und intrinsische Wahrheitswerte

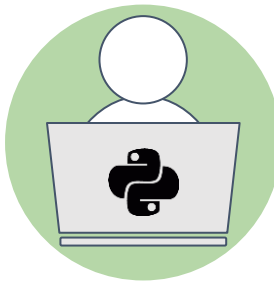
- Mit den bereits bekannten Vergleichsoperatoren `in` und `not in` kann erfragt werden, ob ein bestimmter Wert innerhalb einer Kollektion vorkommt:

```
cooleSkriptsprachen = ['Python' , 'Ruby' , 'Julia' , 'PHP']
'Python' in cooleSkriptsprachen
True
'Perl' not in cooleSkriptsprachen
True

'mist' in 'Pessimist'
True
'Mist' in 'Pessimist'           #Auf Groß-/Kleinschreibung achten!
False
```

- Alle Datentypen und Ausdrücke haben in Python einen intrinsischen Wahrheitswert:

```
bool(119)           #Numerische werte ungleich Null sind True
True
bool(0.0)           #Numerische werte gleich Null sind False
False
bool('Nichts')      #Alle nicht-leeren kollektionen sind True
True
bool([])            #Alle leeren kollektionen sind False
False
bool(1-1)
False
```



Aufgabe: Welche Wahrheitswerte haben die folgenden Ausdrücke und warum?  
Verwendet zum Überprüfen eurer Vermutung die interaktive Pythonshell.

```
'a' == 'A'  
'a' == 'a '  
'Python' == 1  
123 > 2  
'123' > '2'  
'Monty' != 'Monty'  
12.2 > 12  
12 + 3 > 4  
3 > 4 + 12  
'adenin' in 'Trisulfatmonoadeninpentose'  
19 not in (5, 19, 36, 69, 119, 149)
```

# Komplexe Vergleiche

- Einfache Vergleiche lassen sich mit Hilfe logischer Operatoren zu so genannten aussagenlogischen Ausdrücken zusammen bauen.
- Es gibt in Python drei dieser logischen Operatoren:
  - Konjunktion, das logische *und* (**and**)
  - Disjunktion, das logische *oder* (**or**)
  - Negation, das logische *nicht* (**not**)
- Aussagenlogische Ausdrücke folgen immer dieser Form, wobei ein Operand ein einfacher Vergleich oder direkt einer der beiden bool'schen Werte sein kann:

**(Negation)** <Operand 1> **Konjunktion/Disjunktion** **(Negation)** <Operand 2> ...

- Das heißt, vor einem Operand kann eine Negation stehen (muss aber nicht) und zwei Operanden werden entweder durch eine Konjunktion oder eine Disjunktion verbunden.
- Die Auswertung solcher komplexen Vergleiche erfolgt auch immer von links nach rechts.

# Logische Operatoren

- Der **and** Operator liefert genau dann **True**, wenn beide verbundenen Operanden auch **True** sind. Ist mindestens einer, oder beide Operanden **False**, so wird auch **False** zurückgeliefert:
- Der **or** Operator liefert genau dann **True**, wenn mindestens einer der beiden verbundenen Operanden auch **True** ist. Sind beide Operanden **False**, so wird **False** zurückgeliefert:
- Der **not** Operator kehrt einen Wahrheitswert jeweils um:

Operand A	Operand B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Operand A	Operand B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Operand A	not A
True	False
False	True

# Strukturierte logische Ausdrücke

- Vergleiche lassen sich weiter strukturieren, in dem mit Klammerpaaren () die Auswertungsreihenfolge festgelegt werden kann:

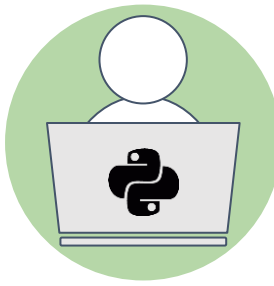
```
not 2 > 3 and 1 != 1  
False
```

```
not (2 > 3 and 1 != 1)  
True
```

- Zudem verbessern Klammerpaare die Lesbarkeit einfacher und komplexer logische Ausdrücke und erleichtern somit das Finden von Logikfehlern im Code:

```
not ((2 > 3) and (1 != 1))  
True
```





Aufgabe: Welche Wahrheitswerte haben die folgenden Ausdrücke und warum?  
Verwendet zum Überprüfen eurer Vermutung die interaktive Pythonshell.

```
not (True or not False and not True) or False and not True or not False
```

```
4 > (9-(1+1)**2)
```

```
('a' <= 5) and (not 'e' > 'bc')
```

```
(not 'e' > 'bc') and ('a' <= 5)
```

```
len([(1, 2), ([True], [False, True])]) > len((1,2,True,False,True))
```

```
[[]] != False and not '' == True
```

I'LL BE IN YOUR CITY TOMORROW  
IF YOU WANT TO HANG OUT.

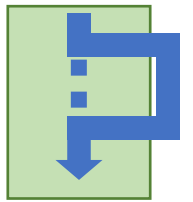
BUT WHERE WILL YOU BE IF  
I *DON'T* WANT TO HANG OUT?!

YOU KNOW, I JUST  
REMEMBERED I'M BUSY.



WHY I TRY NOT TO BE  
PEDANTIC ABOUT CONDITIONALS.

# Bedingte Anweisungen



- Mit Hilfe der gerade gelernten Vergleiche und logischen Operatoren, lassen sich umgangssprachliche Bedingungen in logischen Ausdrücken wiedergeben:

Umgangssprachliche Bedingung	logischer Ausdruck
Bedingung für einen ermäßigten Eintritt im Theater: Die Person ist höchstens 12 oder mindestens 65 Jahre alt oder sie ist studierend.	<code>(alter &lt;= 12) or (alter &gt;= 65) or (thoska == True)</code> <code>not (12 &lt; alter &lt; 65) and not (thoska != True)</code>
Der Stoff is nicht brennbar und geht im Wasser unter.	<code>not brennbar and (dichte &gt; 1)</code> <code>(brennbar == False) and (dichte &gt; 1)</code>

- Meistens kann man mehrere äquivalente logische Ausdrücke für eine Bedingung finden, die geeignet sind.

# Bedingte Anweisungen

- Bei einer Programmverzweigung wird eine einzelne oder eine Folge von Anweisungen in Abhängigkeit einer Bedingung ausgeführt.
- Solche Verzweigungen werden durch die `if` Anweisung realisiert:

**if Bedingung:**  
(Einrückung →) **Anweisungsblock**

- Einrückungen definieren in Python zusammenhängende Anweisungsfolgen, man spricht auch von Anweisungs- oder Codeblöcken:

```
x = 1
y = 2
z = int(input())

if z > x + y:
    x = z // 2
    y = 3
    z = x - y

print(z)
```

← äußerer Anweisungsblock

← innerer Anweisungsblock

# Einseitige Verzweigung

- Ist die Bedingung einer `if` Anweisung nicht erfüllt (d. h. sie ist `False`), dann wird der dazugehörige Anweisungsblock übersprungen:

```
a = 5
b = 19
if a > b:
    print('a ist größer als b.')
```

- Zusätzlich zur einseitigen `if` Verzweigung gibt es noch die zweiseitige `if else` und die mehrseitige `if elif` Verzweigung.

# Zweiseitige Verzweigung

- Bei einer **if else** Verzweigung gibt es zwei Anweisungsblöcke, einen für die **if** Anweisung und einen für die **else** Anweisung.
- Ist die Bedingung der **if** Anweisung **True**, dann wird der **if** Anweisungsblock ausgeführt und der **else** Anweisungsblock übersprungen.
- Ist die Bedingung der **if** Anweisung **False**, dann wird der **if** Anweisungsblock übersprungen und der **else** Anweisungsblock ausgeführt:

**if Bedingung:**

(Einrückung →) Anweisungsblock

**else:**

(Einrückung →) Anweisungsblock

```
a = 5
b = 19
if a > b:
    print('a ist größer als b.')
else:
    print('a ist kleiner als b.')
```

# Mehrseitige Verzweigung

- Bei einer `if elif` Verzweigung gibt es mindestens zwei Anweisungsblöcke, einen für die `if` Anweisung und für jede `elif` Anweisung.
- Es wird immer nur genau derjenige Anweisungsblock ausgeführt, dessen Bedingung `True` ist, d. h. so bald ein Anweisungsblock ausgeführt wurde, werden alle anderen übersprungen.
- Zusätzlich kann (muss aber nicht) eine `else` Anweisung am Ende folgen, deren Anweisungsblock nur dann ausgeführt wird, wenn alle `if` und `elif` Bedingungen `False` waren.

**if Bedingung:**

(Einrückung →) Anweisungsblock

**elif Bedingung:**

(Einrückung →) Anweisungsblock

**else:**

(Einrückung →) Anweisungsblock

```
a = 5
b = 19
if a > b:
    print('a ist größer als b.')
elif a < b:
    print('a ist kleiner als b.')
else:
    print('a und b sind gleich groß.')
```

# Mehrseitige Verzweigung – Beispiel

```
a = 150

if a > 5:
    print('a ist größer als 5')
elif a > 19:
    print('a ist größer als 19')
elif a > 36:
    print('a ist größer als 36')
elif a > 69:
    print('a ist größer als 69')
elif a > 119:
    print('a ist größer als 119')
elif a > 149:
    print('a ist größer als 149')
else:
    print('a ist echt riesig')

a ist größer als 5
```

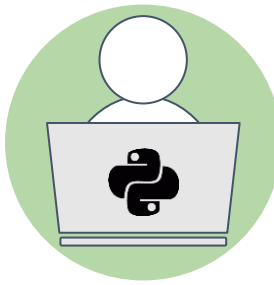


# Mehrseitige Verzweigung – Beispiel

```
a = 150

if a > 5 and <= 19:
    print('a ist größer als 5')
elif a > 19 and <= 36:
    print('a ist größer als 19')
elif a > 36 and <= 69:
    print('a ist größer als 36')
elif a > 69 and <= 119:
    print('a ist größer als 69')
elif a > 119 and <= 149:
    print('a ist größer als 119')
elif a > 149 and <= 9000:
    print('a ist größer als 149')
else:
    print('a ist größer als 9000')

a ist größer als 149
```



## Aufgabe:

Schreibt ein Programm, welches auf Nachfrage entscheidet, ob eine Person ermäßigten Eintritt ins Theater erhält. Ermäßigte Karten gibt es für Kinder unter 13 und Personen über 64 Jahre.

## Hinweis:

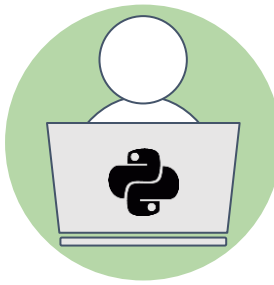
Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

# Verschachtelte Verzweigungen

- Verzweigungen lassen sich auch beliebig verschachteln, so dass sich sehr flexible (aber auch komplexe) Programme schreiben lassen:

```
zahl = int(input('Bitte gib eine Zahl zwischen 0-10 an: '))
sonnig = input('Scheint heute die Sonne? (Ja/Nein): ')

if zahl > 10:
    if sonnig == 'Ja' or sonnig == 'ja':
        if zahl > 3:
            if zahl < 7:
                print('Es ist ein sonniger Tag und deine Zahl ist 4, 5, oder 6.')
            elif zahl <= 3:
                print('Es ist ein sonniger Tag und deine Zahl 3 oder kleiner.')
        else:
            print('Ohne Sonne mag ich keine Zahlen raten...')
    else:
        print('Zahlen größer 10 sind nicht erlaubt!')
```



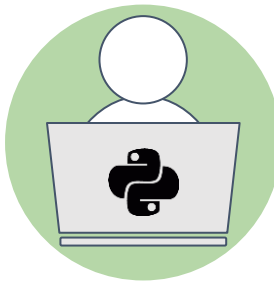
## Aufgabe:

Erweitert euer Programm zur Abfrage nach dem ermäßigten Theatereintritt, so dass auch Studierende mit einer gültigen Thoska, unabhängig ihres Alters, günstigere Karten bekommen.

Was ist die minimale und was die maximale Anzahl an `if` bzw. `elif` Anweisungen um dieses Programm zu realisieren?

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



## Aufgabe:

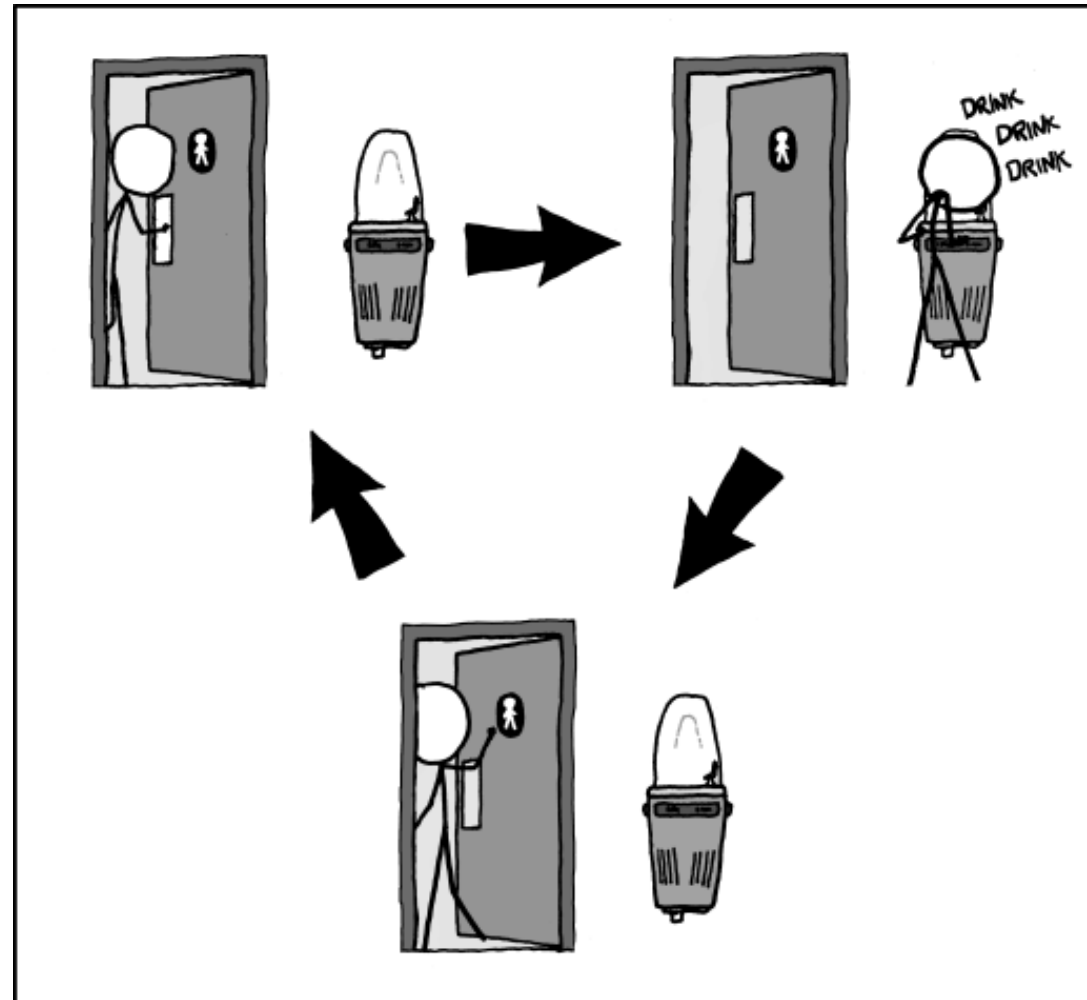
Ein Schaltjahr hat 366 statt 365 Tage und dessen Einführung ist auf die päpstliche Kalenderreform im Jahre 1582 zurückzuführen, d. h. vor diesem Jahr gab es keine Schaltjahre.

Mit folgender Regel lässt sich bestimmen ob ein Jahr ein Schaltjahr ist: Wenn die Jahreszahl mit Rest 0 durch 400 teilbar ist, oder wenn die Jahreszahl ohne Rest durch 4 und nicht durch 100 teilbar ist.

Schreibt ein Programm das nach Eingabe einer Jahreszahl feststellt, ob es sich um ein Schaltjahr handelt und anschließend das Ergebnis ausgibt.

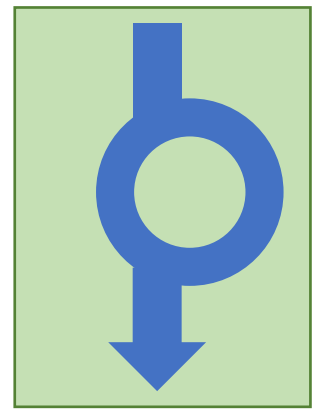
## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



I AVOID DRINKING FOUNTAINS OUTSIDE BATHROOMS  
BECAUSE I'M AFRAID OF GETTING TRAPPED IN A LOOP.

# Wiederholungen mit Schleifen



- Programmschleifen werden verwendet um eine einzelne Anweisung oder einen ganzen Anweisungsblock eine bestimmte Anzahl zu wiederholen, bevor das Programm normal weiterläuft.
- Python gibt es zwei Arten von Schleifen:
  - *Bedingte Schleifen*, die so lange einen Anweisungsblock wiederholen bis eine Bedingung nicht mehr erfüllt (**False**) ist.
  - *Iterationsschleifen*, die für jedes Element innerhalb einer Kollektion (String, Tupel, Liste, Set, Dictionary) einen Anweisungsblock wiederholen.

# Bedingte Schleifen

- Bedingte Schleifen werden durch eine **while** Anweisung erzeugt und der dazugehörige Anweisungsblock so lange wiederholt, bis die Schleifenbedingung nicht mehr erfüllt ist:

**while Bedingung:**  
(Einrückung →) Anweisungsblock

```
a = 2
while a <= 256:    #solange a kleiner als 256 verdopple a
    a = a * 2
```

- Mit bedingten Schleifen lassen sich schnell (un)beabsichtigt Endlosschleifen erzeugen, aus denen das Programm nicht mehr rauskommt, da die Schleifenbedingung niemals **False** werden kann.

```
a = 2
while a > 1:        #Endlosschleife
    a = a * 2
```

#Mit Strg+C löst man ein KeyboardInterrupt aus und bricht das Programm ab



# Iterationsschleifen

- Iterationsschleifen werden durch eine **for in** Anweisung erzeugt und der dazugehörige Anweisungsblock so lange wiederholt, bis jedes Element einer gegebenen Kollektion "abgearbeitet" ist:

**for Element in Kollektion:**  
(Einrückung →) Anweisungsblock

```
farben = ('grün', 'hellgrün', 'dunkelgrün')
for farbe in farben:
    print(farbe)
grün
hellgrün
dunkelgrün
```

- Die Variable einer Iterationsschleife nennt man auch Iterations- oder Laufvariable und sie nimmt nach jedem Schleifendurchgang den Wert des nächsten Elements der Kollektion an.

```
for c in 'ABC':
    print(c)
A
B
C
```

# Iterationsschleifen

- Auch wenn jede Iterationsschleife eine Laufvariable braucht, so muss die Laufvariable selbst gar nicht innerhalb des Anweisungsblocks der Schleife verarbeitet werden:

```
a = 1
b = 2
for i in [1, 2, 3, 4, 5]:
    a = a + b
    b = b - a
print('a =', a, '; b =', b)
a = -2 ; b = 3
```

- Iterationsschleifen eignen sich perfekt als Zählschleifen, da sie es erlauben einen Anweisungsblock eine exakte Anzahl an Wiederholungen zu durchlaufen.

# Die range Funktion

- Anstatt für eine Zählschleife manuell eine Liste oder Tupel von Zahlen schreiben zu müssen, kann man stattdessen die **range** Funktion nutzen:

Stopp  
`range(6)`



0	1	2	3	4	5
---	---	---	---	---	---

Start Stopp  
`range(2, 6)`



2	3	4	5
---	---	---	---

Start Stopp Schritt  
`range(1, 6, 2)`



1	3	5
---	---	---

# Iterationsschleifen als Zählschleifen

- Mittels der `range` Funktion erhält man eine Folge von Ganzzahlen (Integern):

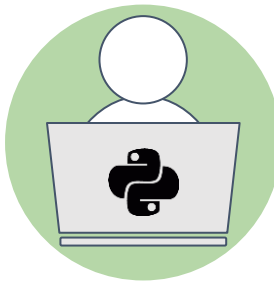
```
zahlenfolge = range(3)
for i in zahlenfolge:
    print(i)
0
1
2
```

- Die `range` Funktion läßt sich auch mit einem Start- und einem Stoppwert aufrufen:

```
for i in range(5, 8):
    print(i)
5
6
7
```

- Die `range` Funktion läßt sich auch mit einem Start-, Stopp- und Schrittwert aufrufen:

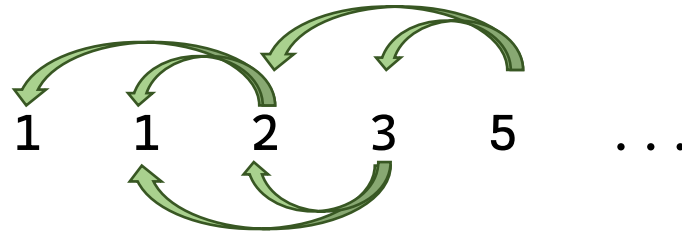
```
for i in range(1, 5, 2):
    print(i)
1
3
```



## Aufgabe:

Die Fibonacci-Folge ist eine berühmte Zahlenfolge, welche sich wie folgt berechnet:

Die ersten beiden Fibonacci-Zahlen haben beide den Wert 1, jede weitere Zahl der Folge ergibt sich aus der Summe seiner beiden Vorgänger.



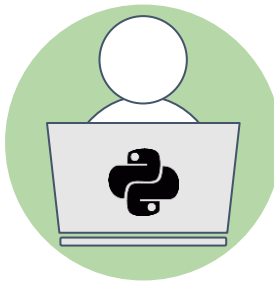
Schreibt ein Programm, welches mit Hilfe einer Zählschleife alle Fibonacci-Zahlen bis zur 20ten Zahl der Folge berechnet und ausgibt.

Lässt sich diese Aufgabe auch mit einer `while` Schleife lösen?

Wenn ja, wie? Wenn nein, warum nicht?

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



## Aufgabe:

Eine Primzahl ist eine Zahl die nur geteilt durch sich selbst oder 1 wieder eine ganze Zahl (Integer) ergibt. Man sagt auch: eine Primzahl hat nur sich selbst und die 1 als echten Teiler.

Primzahlen sind z. B. 2, 3, 5, 7, 11, 13, 17, 19, 23, ...

Schreibt ein Programm, mit Hilfe einer Iterationsschleife, welche überprüft ob eine eingegebene Zahl eine Primzahl ist.

Tipp: Nutzt den Modulo-Operator (%) um zu überprüfen ob eine Zahl  $i$  echter Teiler einer Zahl  $n$  ist.

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

# Primzahlen und Schleifen

```
n = int(input('Gib eine ganze Zahl ein: '))

echten_Teiler_gefunden = False

for i in range(2, n):
    if n%i == 0:
        echten_Teiler_gefunden = True
        print(n, 'geteilt durch', i, 'ist', n/i)

if echten_Teiler_gefunden == False:
    print(n, 'ist eine Primzahl')
```

- Dieses Programm arbeitet zwar korrekt, macht aber viele unnötige Berechnungen:

```
Gib eine ganze Zahl ein: 100
100 geteilt durch 2 ist 50.0
100 geteilt durch 4 ist 25.0
100 geteilt durch 5 ist 20.0
100 geteilt durch 10 ist 10.0
100 geteilt durch 20 ist 5.0
100 geteilt durch 25 ist 4.0
100 geteilt durch 50 ist 2.0
```

# Abbruch einer Schleife

- Mit der Anweisung **break** wird die Ausführung einer Schleife sofort beendet und das Programm fährt bei der ersten Anweisung nach der Schleife fort.

```
n = int(input('Gib eine ganze Zahl ein: '))

echten_Teiler_gefunden = False

for i in range(2, n):
    if n%i == 0:
        echten_Teiler_gefunden = True
        print(n, 'geteilt durch', i, 'ist', n/i)
        break

if echten_Teiler_gefunden == False:
    print(n, 'ist eine Primzahl')
```

```
Gib eine ganze Zahl ein: 100
100 geteilt durch 2 ist 50.0
```



# Überspringen eines Schleifendurchlaufs

- Mit der Anweisung `continue` wird die aktuelle Wiederholung der Schleife abgebrochen und die nächste Wiederholung gestartet, wenn die Schleife noch nicht zuende ist, also wenn:
  - Bei einer `while` Schleife die Schleifenbedingung noch `True` ist.
  - Bei einer `for` Schleife noch nicht alle Elemente der aktuellen Kollektion bearbeitet wurden.

```
#Zählen der Konsonanten in einem Wort

wort = input('Bitte gib ein Wort ein: ')
anzahl = 0
for buchstabe in wort:
    if buchstabe in 'aeiouAEIOU':
        continue
    anzahl = anzahl + 1
print('Das Wort enthält', anzahl, 'Konsonanten.')
```

```
Bitte gib ein Wort ein: Iterationsschleife
Das Wort hat 10 Konsonanten
```

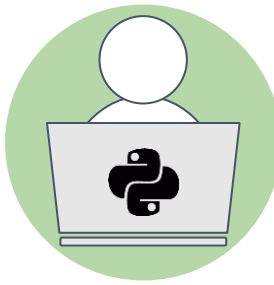
# Geschachtelte Schleifen

- Auch Schleifen lassen sich beliebig verschachteln:

```
for zahl in range(4):  
    for buchstabe in 'abc':  
        print(zahl, buchstabe)
```

```
0 a  
0 b  
0 c  
1 a  
1 b  
1 c  
2 a  
2 b  
2 c  
3 a  
3 b  
3 c
```

- Dabei werden innere Schleifen so oft durchlaufen, bis die jeweils äußeren Schleifen auch einmal komplett durchgelaufen sind.



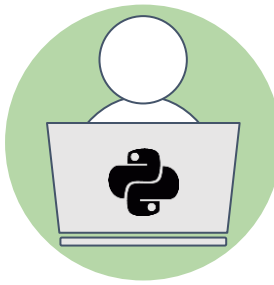
## Aufgabe:

Schreibt ein Programm, dass mittels Schleifen die folgenden drei Sternchen-Muster auf dem Bildschirm ausgibt:

```
* * * *  
* * * *  
* * * *
```

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
  *  
 * *  
* * *  
* * * *  
* * * * *
```



## Aufgabe:

Schreibt ein Programm, welches mit euch Zahlenraten spielt. Am Anfang generiert das Programm eine unbekannte Zufallszahl zwischen 0 und 100. Danach erwartet das Programm eine Zahl als Eingabe und überprüft ob diese mit der Zufallszahl übereinstimmt. Ist eure geratene Zahl zu groß oder zu klein, dann teilt euch das Programm dies mit, so dass ihr wisst ob ihr eine größere oder kleinere Zahl raten müsst. Das Spiel läuft so lange, bis ihr die Zufallszahl richtig erraten habt.

Tipp: Mit den folgenden zwei Anweisungen erzeugt ihr eine Zufallszahl zwischen 0 und 100 und weist sie der Variable `zufallszahl` zu:

```
import random
zufallszahl = random.randint(0,100)
```

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



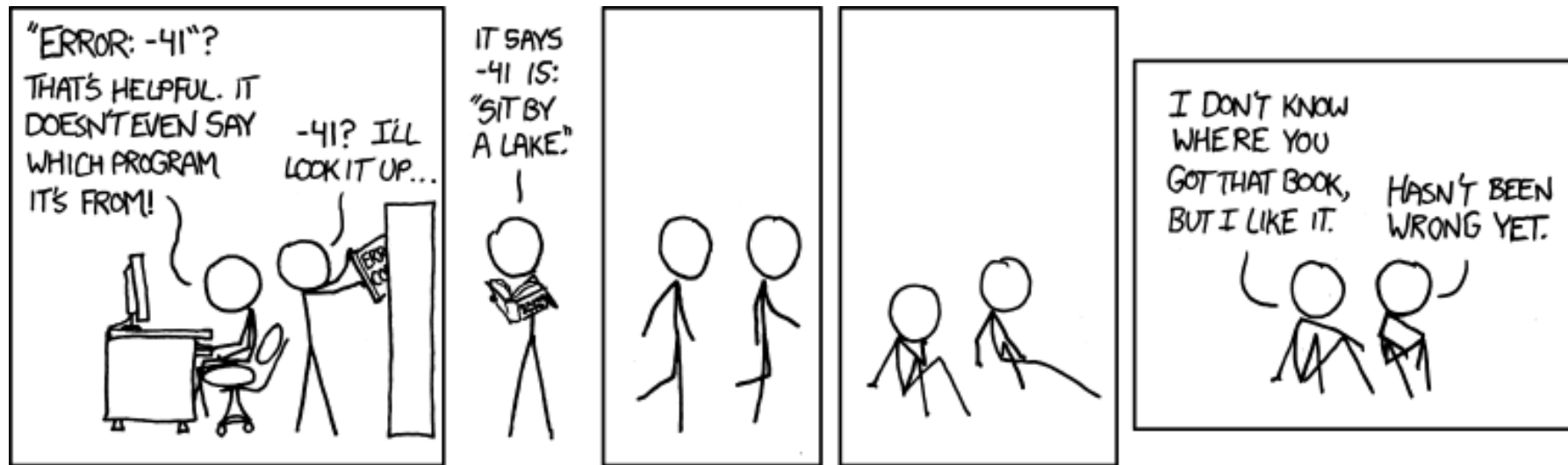
## Aufgabe:

Schreibt ein Programm, welches sämtliche möglichen "Wörter" bestehend aus 3 kleinen Buchstaben auf den Bildschirm schreibt.

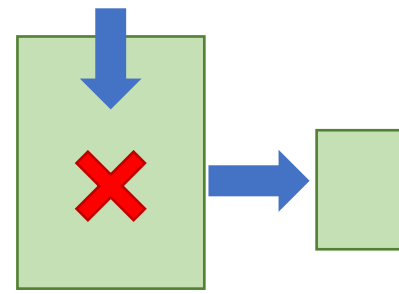
Tipp: Verwendet als Iterationskollektion einen String der aus allen kleinen Buchstaben des Alphabets besteht.

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



# Fehler abfangen mit Exceptions



- "Programme werden von Menschen geschrieben und von ihnen ausgeführt, daher enthalten sie Fehler oder werden fehlerhaft benutzt. Oft ist beides der Fall." - *Unbekannt*

```
zahl = int(input('Bitte gib eine ganze Zahl ein: '))  
Bitte gib eine ganze Zahl ein: neunzehn  
  
ValueError: invalid literal for int() with base 10: 'neunzehn'
```

- Da die `int` Funktion nur Strings in Integer umwandeln kann, die ausschließlich aus Ziffern bestehen, kommt es hier zu einem Fehler und dadurch zu einem Programmabbruch.
- Ungewollte Programmabbrüche können fatale Folgen haben, wie etwa Datenverlust.

# Fehlerarten

- Prinzipiell gibt es drei Arten von Fehlern beim Programmieren:
  - Syntaktische Fehler, d. h. Fehler in der Grammatik von Anweisungen.  
Diese werden automatisch beim Start des Programms vom Pythoninterpreter erkannt und gemeldet:

```
x = [5, 19]]  
SyntaxError: unmatched ']
```

- Laufzeitfehler, d. h. Fehler die in syntaktisch korrekten Programm erst auftreten, wenn das Programm schon läuft:

```
zahl = int(input('Bitte gib eine ganze Zahl ein: '))  
Bitte gib eine ganze Zahl ein: neunzehn  
ValueError: invalid literal for int() with base 10: 'neunzehn'
```

- Semantikfehler (Logikfehler), d. h. das Programm läuft "korrekt" weil es keine Fehlermeldungen liefert, aber das Ergebnis nicht dem entspricht was man sich vorgestellt hat.



# Semantikfehler - Beispiele

- Im Gegensatz zu Syntax- und Laufzeitfehlern kann der Pythoninterpreter Semantikfehler nicht erkennen und liefert daher auch keine Fehlermeldung, wie z. B. hier:

```
if x > 19:
    if x == y:
        print('x ist gleich y')
else:
    print('x ist ungleich y')
```

```
if x > 19:
    if x == y:
        print('x ist gleich y')
else:
    print('x ist ungleich y')
```

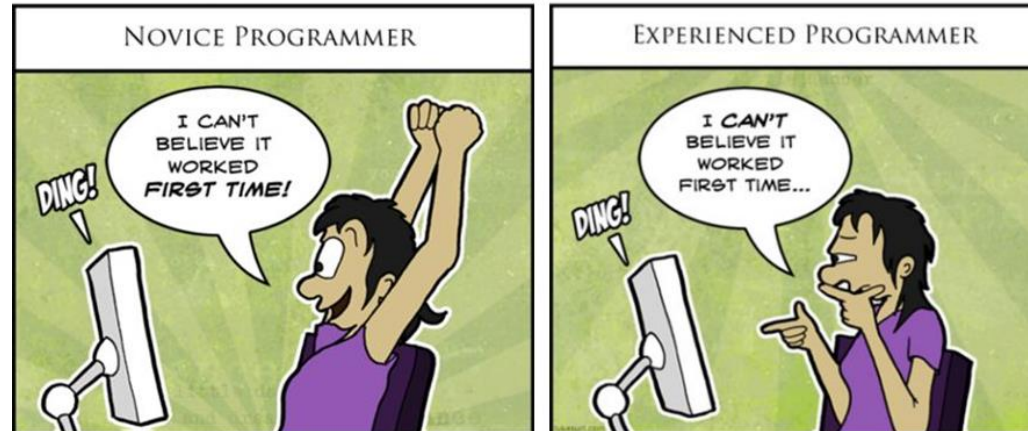
- Auch unser erstes Code-Beispiel mit den Typumwandlungen von gestern stellt einen Semantikfehler dar:

```
x = input('Bitte x eingeben: ')
y = input('Bitte y eingeben: ')
summe = x + y
print('Summe:', summe)
```

```
Bitte x eingeben: 2
Bitte y eingeben: 3
Summe: '23'
```

# Semantikfehler vermeiden

- Manche Semantikfehler können sehr offensichtlich sein, andere hingegen extrem schwer zu finden.
- Ein paar allgemeine Ansätze um Semantikfehler zu vermeiden:
  - Beim Arbeiten an einem umfangreicheren Programm, solltest du immer Zwischenlösungen testen.  
→ Das heißt, ein paar Zeilen Code schreiben, dann einen Testlauf starten, ob bis hierhin alles funktioniert wie es soll, dann Fehler beseitigen. Dann die nächsten Zeilen Code schreiben und wieder testen, etc.
- Guter Programmierstil hilft, Fehler zu vermeiden. Also z. B.:
  - das Programm möglichst genau Kommentieren und Variablennamen sinnvoll auswählen
  - komplexe logische Ausdrücke in einer einzelnen Bedingung vermeiden, stattdessen lieber mehrere Bedingungen mit einfacheren logischen Ausdrücken formulieren
- Immer misstrauisch gegenüber dem eigenen Code bleiben, denn oft sind es Kleinigkeiten die einen Fehler verursachen.



# Typische Einsteigerfehler

- Die häufigsten Fehler die beim Programmieren mit Python auftreten:
  1. Schreibfehler bei Variablennamen (z. B. `meinVariable` statt `meineVariable`)
  2. Groß-/Kleinschreibung beim Arbeiten mit Strings:

```
'python' in 'wir programmieren mit Python!'  
False
```

3. Falsche Einrückung oder vergessen des `:` Symbols bei einem Anweisungsblock:

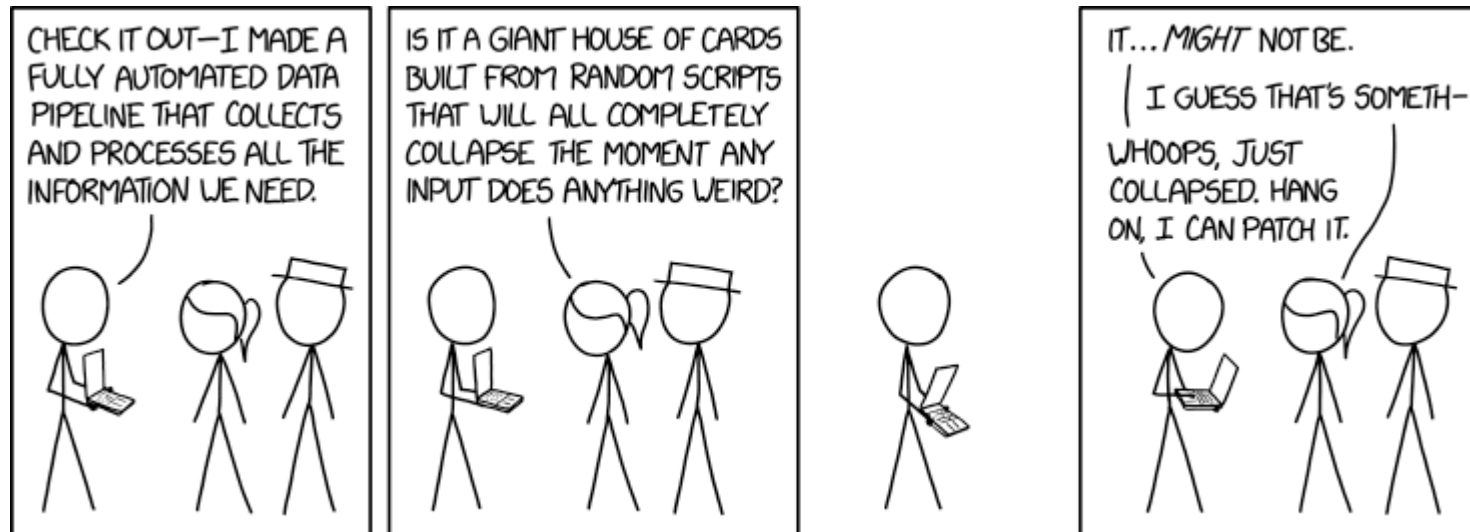
```
if x > 19:  
    if x == y:  
        print('x ist gleich y')  
else:  
    print('x ist ungleich y')
```

4. Verwendung des Zuweisungsoperators (`=`) statt des Vergleichoperators (`==`),  
z. B. `x = y` statt `x == y`
5. Falscher Wertebereich in Kollektionen, weil bei 0 angefangen wird zu zählen:

```
liste = ['a', 'b', 'c']  
liste[3]  
IndexError: list index out of range
```

# Laufzeitfehler verhindern

- Laufzeitfehler, also Fehler die der Pythoninterpreter erst nach dem Start eines Programms erkennen kann und dann das Programm abbricht, können viele Ursachen haben:
  - Variablen wurden falsch oder überhaupt nicht umgewandelt
  - Es wird versucht auf ein nicht existierendes Element einer Kollektion zuzugreifen
  - Es wird versucht auf eine Datei zuzugreifen die nicht existiert
  - Eine Zahl wird durch 0 dividiert
  - ...



# Laufzeitfehler verhindern

- Jedes Mal, wenn ein Laufzeitfehler auftritt "wirft" der Pythoninterpreter eine spezifische Fehlermeldung (man sagt auch *Exception*) die den aufgetretenen Fehler beschreibt:

```
liste = ['a', 'b', 'c']  
liste[3]  
IndexError: list index out of range  
  
19/0  
ZeroDivisionError: division by zero  
  
int('zwei')  
ValueError: invalid literal for int() with base 10: 'zwei'
```

- Eine Exception kann "aufgefangen" werden, d. h. wenn ein bestimmter Laufzeitfehler auftritt, dann kann man diesen durch einen eigens dafür geschriebenen Anweisungsblock abfangen und behandeln.
- Wird eine geworfene Exception aufgefangen, dann bricht der Pythoninterpreter das laufende Programm nicht mehr ab.

# Laufzeitfehler verhindern

- Exceptions können durch die **try except** Anweisung abgefangen werden:

```
try:  
    (Einrückung →) Anweisungsblock  
except (Ausnahmetyp):  
    (Einrückung →) Anweisungsblock
```

```
try:  
    zahl = int(input('Bitte gib eine ganze Zahl ein: '))  
    print('Danke für die Zahl.')except ValueError:  
    print('Die Eingabe war nicht in Ordnung und wird auf den Wert 19 gesetzt.')    zahl = 19
```

- Tritt innerhalb des **try** Anweisungsblocks eine Exception auf, dann schaut der Pythoninterpreter ob es einen passenden **except** Anweisungsblock gibt um den Fehler zu behandeln.
- Der **except** Anweisungsblock wird nur aufgerufen, wenn der dazu passende Fehler auftritt, sonst wird er einfach übersprungen.

- Zu einem `try` Anweisungsblock können auch mehrere `except` Anweisungsblöcke definiert werden, um verschiedene Fehlertypen/Exceptions abzufangen:

```
try:
    zahl = int(input('Bitte gib eine ganze Zahl ein: '))
    ergebnis = 5 / zahl
    print('Danke für die Zahl. Das Ergebnis ist', ergebnis)

except ValueError:
    print('Die Eingabe war nicht in Ordnung und wird auf den Wert 19 gesetzt.')
    zahl = 19
    ergebnis = 5 / zahl
    print('Danke für die Zahl. Das Ergebnis ist', ergebnis)

except ZeroDivisionError:
    print('Division durch 0 ist nicht möglich. Ergebnis wird auf 1 gesetzt.')
    ergebnis = 1
```

- Definiert man keinen genauen Laufzeitfehlertyp bei einem `except` Anweisungsblock, dann wird jede auftretende Exception dort aufgefangen:

```
try:
    zahl = int(input('Bitte gib eine ganze Zahl ein: '))
    ergebnis = 5 / zahl
    print('Danke für die Zahl. Das Ergebnis ist', ergebnis)
except:
    print('Irgendein Fehler ist aufgetreten. Die Variablen werden auf 1 gesetzt.')
    zahl = 1
    ergebnis = 1
```





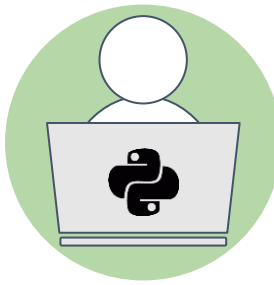
## Aufgabe:

1. Erweitert eure Programme "Zahlen raten", so dass bei einer falschen Eingabe das Programm nicht einfach abbricht, sondern sich mit einer Nachricht verabschiedet, dass nur Ziffern eingegeben werden dürfen.
2. Erweitert euer Programm "Schaltjahr", so dass immer wieder nach einer neuen Jahreszahl gefragt wird um zu berechnen ob es sich um ein Schaltjahr handelt oder nicht. Das Programm soll erst dann enden, wenn keine gültige Eingabe an das Programm übergeben wurde. In diesem Fall soll es sich mit einer kurzen Nachricht verabschieden.

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

# Kreditrückzahlungen



Aufgabe: Schreibt ein Programm, das die Rückzahlung eines Kredits berechnet, der dieses Jahr aufgenommen wurde. Das Programm soll für jedes Jahr die Zinsen, Tilgung und die Restschulden berechnen und ausgeben. Vom Nutzer sollend folgende Daten abgefragt werden:

- Kreditbetrag in Euro
- Zinssatz in Prozent
- Betrag in Euro, der jedes Jahr an den Kreditgeber zurückgezahlt werden soll und von dem die Zinsen und die Tilgung bezahlt werden

Eine Beispielausgabe könnte so aussehen:

```
Kredit in Euro: 1000
Zinssatz (Prozent pro Jahr): 6
Jährliche Rückzahlung in Euro: 200
2022 Zinsen: 60 EUR    Tilgung: 140 EUR    Rest: 860 EUR
2023 Zinsen: 51 EUR    Tilgung: 149 EUR    Rest: 711 EUR
2024 Zinsen: 42 EUR    Tilgung: 158 EUR    Rest: 553 EUR
2025 Zinsen: 33 EUR    Tilgung: 167 EUR    Rest: 386 EUR
2026 Zinsen: 23 EUR    Tilgung: 177 EUR    Rest: 209 EUR
2027 Zinsen: 12 EUR    Tilgung: 188 EUR    Rest: 21 EUR
Restforderung: 21 Euro
```

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
              // guaranteed to be random.  
}
```

# Funktionen

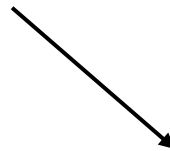
# Mathematische Funktionen

- Funktionen kennt man meist schon aus der Mathematik, wie z. B.:
  - $f(x) = x^2 + 2$
  - $g(x, y) = \frac{x-y}{x+y}$
  - $\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$
- Mathematische Funktionen bestehen aus einem Namen, einem oder mehrerer Parameter und einer Berechnungsvorschrift.
- Setzt man für die Parameter konkrete Werte ein, so gibt die Funktion eine berechnete Lösung zurück:
  - $f(2) = 2^2 + 2 = 6$
  - $g(5, 14) = \frac{5-14}{5+14} = \frac{-9}{19}$
  - $\sin(90.45) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} 90^{2n} \approx 0.6101001254913061$

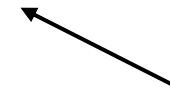
# Informatische Funktionen

- Funktionen in der Informatik sind nach dem selben Prinzip wie mathematische aufgebaut:

Funktionskopf



```
def funktionsname(parameter):  
    (Einrückung →) Anweisungsblock
```



Funktionskörper

- Der große Unterschied: Wir können nicht nur Zahlen, sondern beliebige Datentypen verarbeiten, d. h. wir können beliebige Datentypen als Eingabeparameter und auch als Rückgabewerte definieren.

# Informatische Funktionen

- Funktionen sind aufrufbare Anweisungsblöcke, welche eine spezifische Teilaufgabe eines Programms lösen sollen.
- Sie helfen nicht nur Code (durch schrittweise Verfeinerung) besser zu strukturieren, sondern ermöglichen einen effizienten Programmierstil.
- Python besitzt viele Standardfunktionen (*built-in functions*) von denen wir schon einige selbst benutzt haben:

```
print('Hallo welt!')  
Hallo welt!  
  
len([5, 19, (36, 64, 91), 119])  
4  
  
minimum = min(68, 43, 17, 89, 33, 12, 80)
```

- Zuerst schreibt man den Namen einer Funktion und danach in runden Klammern die Parameter (auch Argumente genannt) mit Komma voneinander getrennt, anschließend gibt die Funktion ein Ergebnis zurück.

# Eigene Funktionen in Python

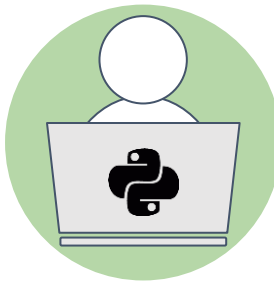
- Zur Definition einer eigenen Funktion in Python verwendet man die **def** Anweisung:

```
def f(x):  
    x = x + 1  
  
f(1)
```

- Damit eine eigene Funktion auch ein Ergebnis liefert, verwendet man die **return** Anweisung um einen Rückgabewert festzulegen:

```
def f(x):  
    x = x + 1  
    return x  
  
f(1)  
2
```

# Erste eigene Funktionen



Aufgabe: Schreibt ein Programm, in welchem ihr für die untenstehenden Funktionen, jeweils eine eigene Funktion definiert habt. Testet im Anschluss eure Funktionen mit verschiedenen Argumenten.

- $f(x) = x^2 + 2$
- $g(x, y) = \frac{x-y}{x+y}$
- Eine Funktion, welche die Xte Fibonacci-Zahl zurückgibt.
- Die Ulam-Funktion, welche einen Integer-Parameter namens `a` als Eingabe erwartet. Ist `a` gleich 1, stoppt die Funktion, ist `a` gerade wird die Variable durch zwei geteilt, in allen anderen Fällen wird der Wert von `a` verdreifacht und anschließend noch um 1 erhöht. Dies wird so lange wiederholt bis die Funktion stoppt, wobei in jeder Wiederholung der aktuelle Wert von `a` auf dem Bildschirm ausgegeben wird.

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.



# Eigene Funktionen in Python

- Eine einzelne Funktion kann auch mehrere **return** Anweisungen haben, um in Abhängigkeit der übergebenen Parameter oder der berechneten Werte unterschiedliche Rückgabewerte wiedergeben zu können:

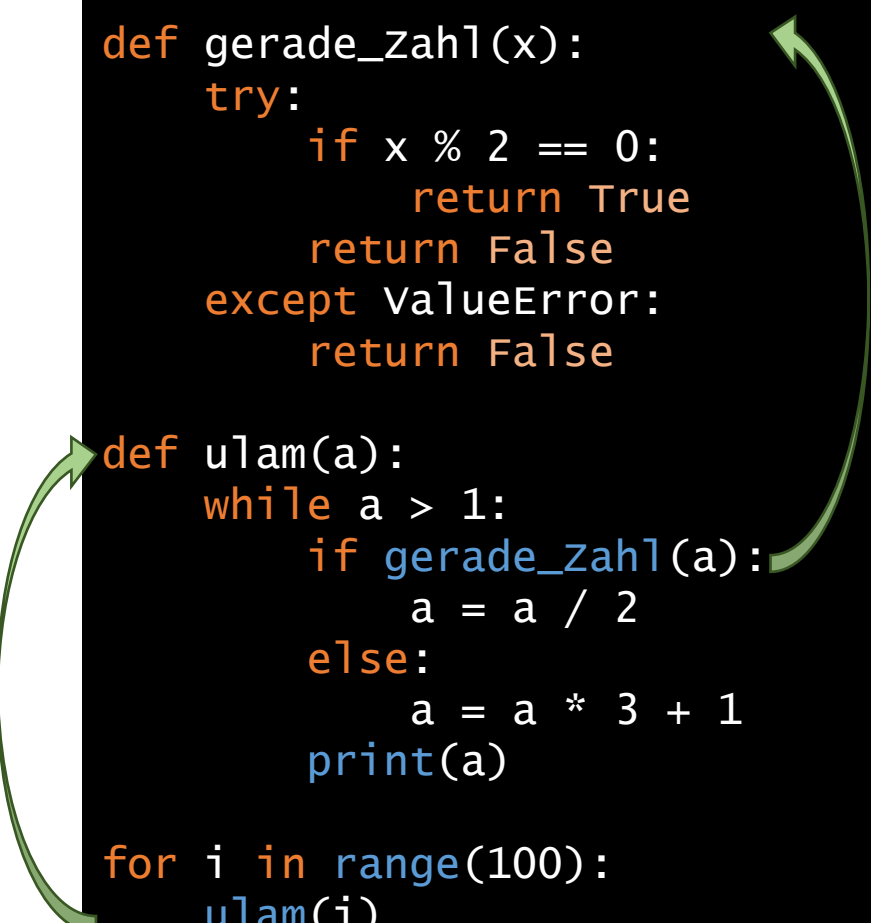
```
def gerade_Zahl(x):  
    #Diese Funktion überprüft ob der Parameter x eine gerade Zahl ist  
    try:  
        if x % 2 == 0:  
            return True  
            print('Diese Codezeile kann nie erreicht werden!')  
        return False  
    except ValueError:  
        return False  
  
gerade_Zahl(1)  
False  
gerade_Zahl(2)  
True
```

- So bald innerhalb einer Funktion eine **return** Anweisung erreicht wurde, wird der Anweisungsblock der Funktion sofort verlassen und nicht weiter abgearbeitet.

# Eigene Funktionen in Python

- Eigene Funktionen können wie Standardfunktionen an beliebigen Stellen in einem Programm aufgerufen werden, z. B. auch innerhalb anderer selbstdefinierter Funktionen:

```
def gerade_Zahl(x):  
    try:  
        if x % 2 == 0:  
            return True  
        return False  
    except ValueError:  
        return False  
  
def ulam(a):  
    while a > 1:  
        if gerade_Zahl(a):  
            a = a / 2  
        else:  
            a = a * 3 + 1  
        print(a)  
  
for i in range(100):  
    ulam(i)
```



# Funktionsargumente

- In der Regel akzeptiert jede Funktion nur eine bestimmte Anzahl an Argumenten, die auch jeweils einem bestimmten Datentyp angehören müssen.
- Argumente können konkrete Werte oder auch Ausdrücke sein.

```
len('eins')                #Ein Literal als Argument
4

len('zwei', 'drei')        #Die Funktion len() akzeptiert nur ein Argument
TypeError: len() takes exactly one argument (2 given)

len('zwei' + 'drei')       #Ein Ausdruck als Argument
8

len(1)                     #Die Funktion len() akzeptiert nicht jeden Datentyp
TypeError: object of type 'int' has no len()
```

# Beliebig viele Funktionsargumente

- Manche Funktionen akzeptieren aber auch eine beliebige Anzahl an Argumenten:

```
min(68, 43, 17, 89, 33, 12, 80)  
12
```

```
min(68, 43, 17, 89)  
17
```

```
min(68)  
68
```

- Um das bei eigenen Funktionen zu ermöglichen, verwendet man den *Sternoperator* vor einem Argumentsnamen im Funktionskopf. Alle übergebenen Argumente werden dann in einem Tupel abgelegt:

```
def summe(*zahlen):  
    summe = 0  
    for zahl in zahlen:  
        summe = summe + zahl  
    return summe
```

```
summe(68, 43, 17, 89, 33, 12, 80)  
342
```

# Optionale Funktionsargumente

- Manche Funktionen können optionale Argumente haben, d. h. man kann diese beim Aufruf der Funktion übergeben, muss es aber nicht.  
→Übergibt man die optionalen Argumente nicht, so werden stattdessen vorher festgelegte Standardwerte (Defaults) übernommen.
- Bspw. arbeitet die `range` Funktion mit optionalen Argumenten:

```
list(range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
list(range(2, 10))  
[2, 3, 4, 5, 6, 7, 8, 9]  
  
list(range(2, 10, 2))  
[2, 4, 6, 8]
```

- Gibt man der `range` Funktion keinen Startwert wird dieser standardmäßig auf 0 gesetzt.
- Gibt man der `range` Funktion keine Schrittweite wird diese standardmäßig auf 1 gesetzt.

# Optionale Funktionsargumente

- Um in eigenen Funktionen optionale Argumente zu ermöglichen, kann man im Funktionskopf den entsprechenden Argumenten per Zuweisungsoperator (=) einen Standardwert zuweisen:

```
def multiplizieren(x, y=1):  
    return x * y
```

```
multiplizieren(19.36)  
19.36
```

```
multiplizieren(19.36, 5)  
96.8
```

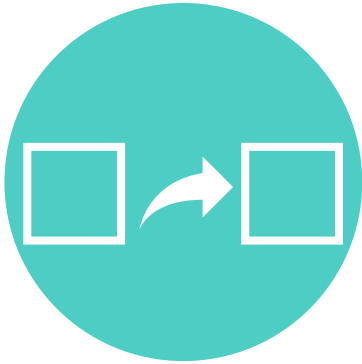
- Achtung: Optionale Argumente müssen im Funktionskopf immer ganz rechts von allen nicht-optionalen Argumenten stehen:

```
def multiplizieren(y=1, x):  
    return x * y
```

```
SyntaxError: non-default argument follows default argument
```

# Datentypen und Parameterübergabe

- Wird eine Variable als Argument in eine Funktion gegeben und dort verarbeitet, ändert sich eventuell der Inhalt dieser Variable. Dies hängt vom Datentyp der Variable ab:

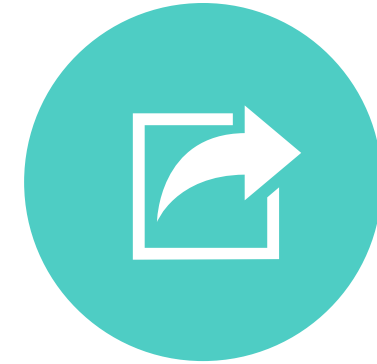


## call-by-value

Innerhalb der Funktion wird mit Kopien der übergebenen Parameter gearbeitet.



unveränderlich



## call-by-reference

Innerhalb der Funktion wird mit Referenzen auf die übergebenen Parameter gearbeitet.



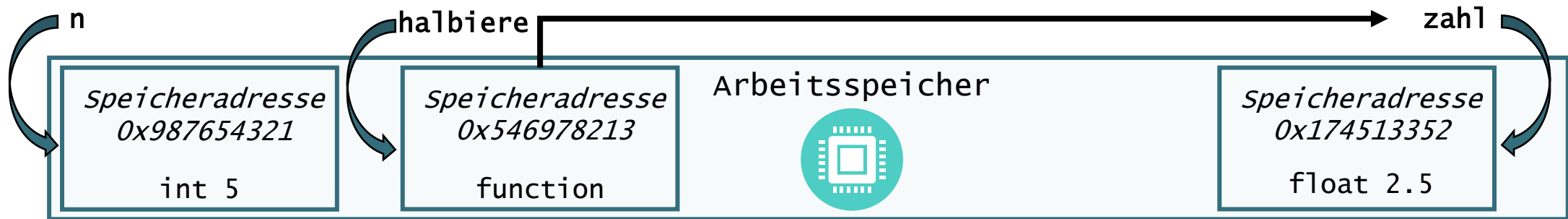
veränderlich

# Datentypen und Parameterübergabe

- Wird eine Zahl-, String-, oder Tupel-Variable in eine Funktion als Argument übergeben, wird stattdessen eine exakte Kopie dieser Variable innerhalb der Funktion verarbeitet:

```
n = 5
def halbiere(zahl):
    zahl = zahl/2
    return zahl

halbiere(n)
2.5
print(n)
5
```



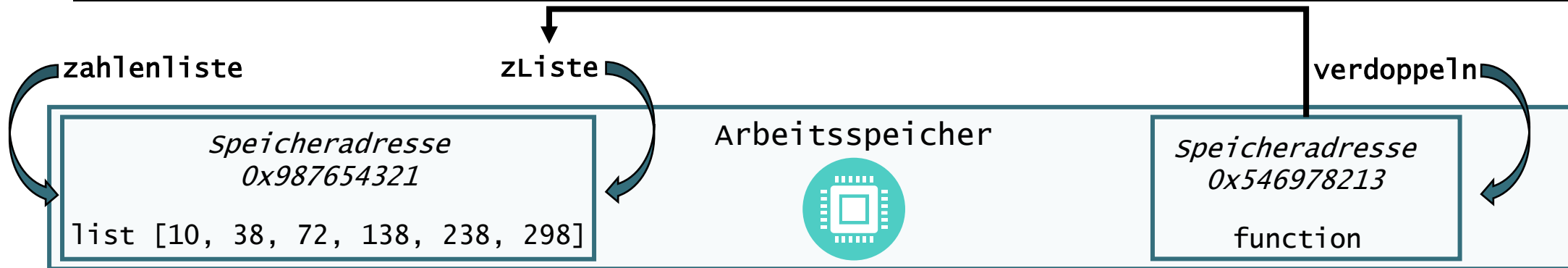


# Datentypen und Parameterübergabe

- Wird eine Listen-, Set-, oder Dictionary-Variable in eine Funktion als Argument übergeben, wird auch genau diese Variable innerhalb der Funktion verarbeitet:

```
zahlenliste = [5, 19, 36, 69, 119, 149]
def verdoppeln(zListe):
    for i in range(len(zListe)):
        zListe[i] = zListe[i] * 2
    return zListe

doppelt = verdoppeln(zahlenliste)
print(doppelt)
[10, 38, 72, 138, 238, 298]
print(zahlenliste)
[10, 38, 72, 138, 238, 298]
```

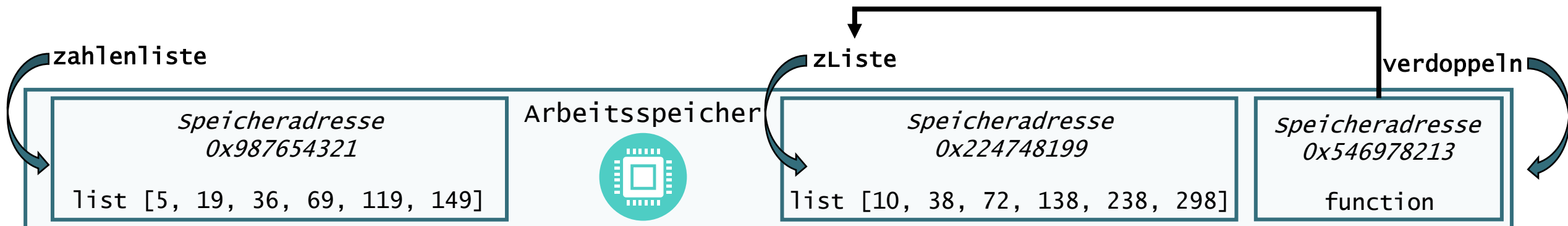


# Datentypen und Parameterübergabe

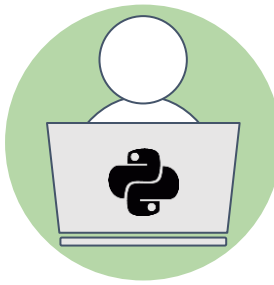
- Möchte man die ursprünglichen Variablenwerte aber nicht verändern, so muss man explizit eine Kopie der Variable erzeugen und diese Kopie als Argument an die Funktion übergeben:

```
zahlenliste = [5, 19, 36, 69, 119, 149]
def verdoppeln(zListe):
    for i in range(len(zListe)):
        zListe[i] = zListe[i] * 2
    return zListe

doppelt = verdoppeln(zahlenliste.copy())
print(doppelt)
[10, 38, 72, 138, 238, 298]
print(zahlenliste)
[5, 19, 36, 69, 119, 149]
```



- Du bist in der Lage (komplexe) logische Vergleiche und Bedingungen zu formulieren und anzuwenden.
- Du kannst nicht-lineare Programme mithilfe von Kontrollstrukturen verstehen und selbst entwickeln.
- Du weißt warum bestimmte Arten von Fehlern (Exceptions) auftreten können und wie man diese abfangen kann.
- Du kannst erklären wofür selbstdefinierte Funktionen gut sind und wie man diese implementieren kann.



Aufgabe: Programmiert folgende mathematischen Funktionen:

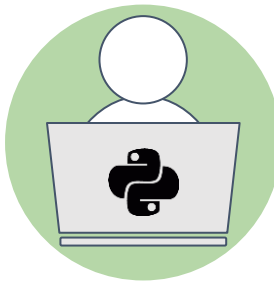
- Die Potenzfunktion:  $\text{potenz}(x, n) = x^n = \underbrace{x * x * x \dots * x}_{n \text{ Mal.}}$
- Die Fakultätsfunktion:  $\text{fakultät}(x) = x! = x * (x - 1) * (x - 2) * (x - 3) * \dots * 1$
- Die Sinusfunktion:  $\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$

Es gilt:  $x^0 = 1$  und  $0! = 1$

Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

## weitere Funktionen



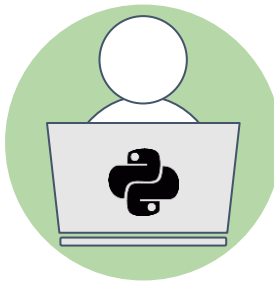
Aufgabe: Programmiert folgende Funktionen, ohne bereits existierende Standardfunktionen zu benutzen:

- Eine Minimumsfunktion, welche für eine beliebige Anzahl an übergebenen Zahlen das Minimum findet und zurückgibt.
- Eine Mittelwertsfunktion, welche für eine beliebige Anzahl an übergebenen Zahlen den Mittelwert berechnet und zurückgibt.
- Eine Range Funktion, welche wie die Standardfunktion `range` eine Liste mit auf- oder absteigenden Integern zurückgibt. Als Argumente braucht eure Range Funktion auch mindestens den Stopp-Wert. Als optimale Argumente können noch entweder ein Start-Wert oder ein Schritt-Wert oder auch beides übergeben werden.
- Eine Sortierfunktion, welche als Eingabe eine Liste erwartet, deren Elemente aus beliebig vielen Zahlen besteht. Rückgabewert der Sortierfunktion ist die von klein nach groß sortierte Eingabeliste.
- Eine Medianfunktion, welche für eine beliebige Anzahl an übergebenen Zahlen den Median berechnet und zurückgibt.

### Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.

# komplexere Funktionen



## Aufgabe:

Unter Steganographie versteht man die Technik, eine Information in einem Dokument so zu verstecken, dass sie nur von einer "eingeweihten" Person gelesen werden kann.

Schreibt eine Funktion mit zwei Argumenten `s` und `n`, die einen Klartext `s` auf folgende Weise durch Steganographie unleserlich macht:

- Der String `s` wird in Großbuchstaben umgewandelt.
- Hinter jedes Zeichen werden `n` zufällige Großbuchstaben eingefügt.
- Das Argument `n` ist optional (Default=1).

Tipp: Mit Hilfe einer Zufallszahl kann aus einem String ein zufälliger Buchstabe gewählt werden. Mit den folgenden zwei Anweisungen erzeugt ihr eine Zufallszahl zwischen 0 und 25 und weist sie der Variable `zufallszahl` zu:

```
import random
zufallszahl = random.randint(0,25)
```

Tipp: Die Typfunktion `upper()` liefert zu einem String eine Kopie bestehend aus Großbuchstaben:

```
'abc'.upper()
'ABC'
```

## Hinweis:

Bevor ihr anfangt Code zu schreiben: Zerlegt die Aufgabe zuerst möglichst kleinschrittig in die einzelnen Teilaufgaben, die vom Programm erledigt werden müssen.