

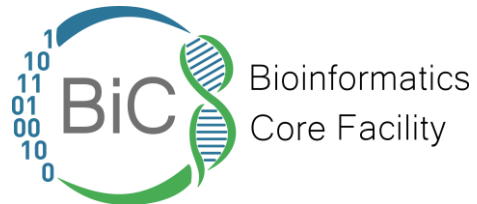
Programmieren für Einsteiger

- Arbeiten mit numerischen Daten in Python -

Tag 3

28.07.2022

Emanuel Barth, Daria Meyer



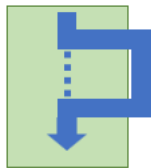
Kurze Wiederholung Tag 2

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.



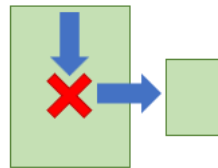
lineares Programm



Programmverzweigung



Programmschleife

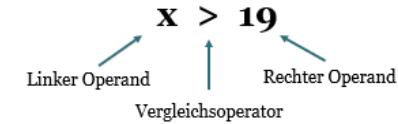


Fehlerbehandlung

6

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

9

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.

14

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.')
```

23

Kurze Wiederholung Tag 2

Programmschleifen



- 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.

31

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 *= 2

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

32

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
```

33

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
```

35

Kurze Wiederholung Tag 2

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.

47

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.



49

Typische Einsteigerfehler

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

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

- 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')
```

- Verwendung des Zuweisungsoperators (=) statt des Vergleichoperators (==), z. B. x = y statt x == y

- Falscher Wertebereich in Kollektionen, weil bei 0 angefangen wird zu zählen:

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

50

Laufzeitfehler verhindern

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

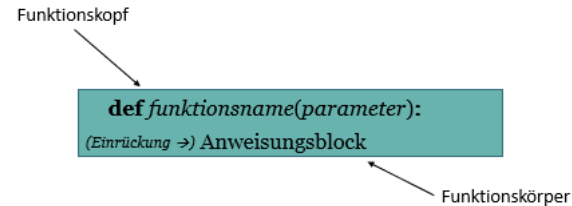
```
try:
    zahl = int(input('Bitte gib eine ganze Zahl ein: '))
    ergebnis = 5 / zahl
    print('Danke für die Zahl. Das Ergebnis ist', ergebnis)
except ValueError:
    print('Die Eingabe war nicht in Ordnung und wird auf den wert 19 gesetzt.')
    zahl = 19
    ergebnis = 5 / zahl
    print('Danke für die Zahl. Das Ergebnis ist', ergebnis)
except ZeroDivisionError:
    print('Division durch 0 ist nicht möglich. Ergebnis wird auf 1 gesetzt.')
    ergebnis = 1
```

54

Kurze Wiederholung Tag 2

Informatische Funktionen

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



- 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.

60

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
```

62

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
```

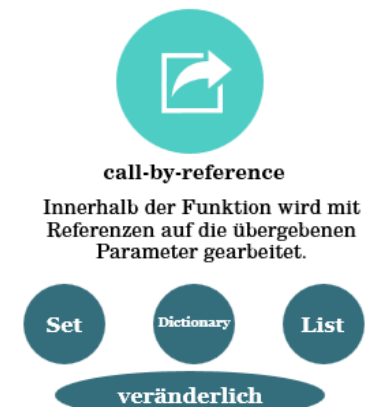
- Achtung:** Optionale Argumente müssen im Funktionskopf immer ganz rechts von allen nicht-optionalen Argumenten stehen:

```
def multiplizieren(y=1, x):  
    return x*y  
  
SyntaxError: non-default argument follows default argument
```

69

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:



70

Lernziele für Heute

- Du weißt was Namensräume sind, welche Bedeutung sie für Variablen und Funktionen haben und wie Python mit Namenskonflikten umgeht.
- Du kennst den Unterschied zwischen einem Pythonprogramm und einem Modul und kannst mit Hilfe von *pip* externe Module installieren und verwenden.
- Du bist in der Lage Datei-Objekte zu erzeugen und mit diesen Textdateien zu lesen und zu schreiben.
- Du kannst Grundlegende Typfunktionen von Strings zur Verarbeitung (un)strukturierter Textdaten anwenden.
- Du weißt was NumPy ist und wofür es verwendet wird.

NAMESPACE LAND RUSH CHEAT SHEET

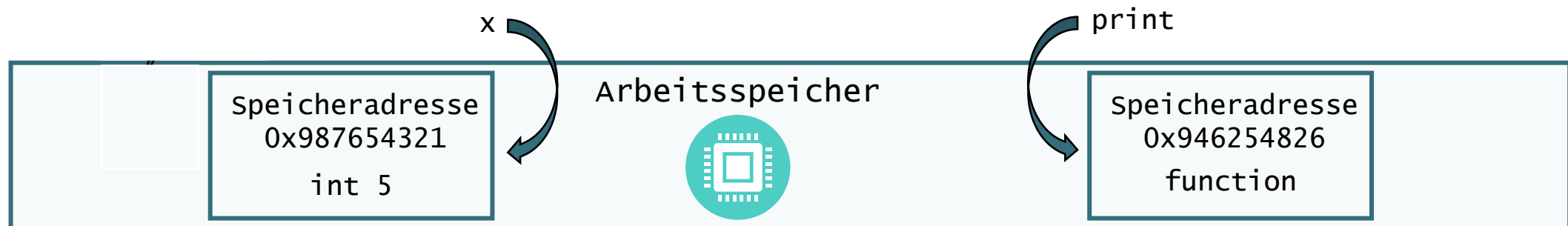
WHEN A NEW SERVICE APPEARS THAT LETS YOU REGISTER A NAME, HERE ARE SOME YOU MAY WANT TO TRY TO GET FIRST:

<u>STRAIGHTFORWARD</u>		<u>RECOGNIZABLE</u>	
<YOUR USUAL USERNAME, IF ANY>		GOOGLE	IPHONE
<YOUR GIVEN NAME>		FACEBOOK	BITCOIN
<YOUR FULL NAME>		OBAMA	CANADA
<INITIAL><SURNAME>		NFL	GARFIELD
<SURNAME> (BOLD & SLIGHTLY UNCONVENTIONAL)		<YOUR CITY>	NASA
		<NAME OF PERSON WHO RUNS THE SERVICE>	
<u>CAUSING TROUBLE</u>	<u>CAUSING MORE TROUBLE</u>	<u>IMPOSSIBLE TO SAY</u>	
USER	ADMIN	HYPHEN-EMDASH	
USERNAME	ADMINISTRATOR	DASH-8HYPHEN-8	
NAME	SYSTEM	ZEROONE2NUMERAL2	
YOU	<NAME OF SERVICE>	KRISASINHEMSWORTH	
GUEST	HELP	THEWORD&ERSAND	
ACCOUNT	ERROR	ZETTAWITH3TEES	
<u>MISC</u>		<u>PERMISSIVE CHARACTER SETS</u>	
<SINGLE LETTERS>		<SPACE>	@ é " "
<SINGLE NUMBERS>		<NBSP>	\ . # " '
<COMMON WORDS>		<RTL OVERRIDE>	-- / ' ' "
<SQL/JS INJECTION>		<ANY EMOJI>	" , " &NBSP;
ASDF	QWERTY	</HTML>	</HTML>
YES	BOT	OKTHISISKINDOFCONFUSINGBUTIT'S	
COMPUTER	BLOCKED	<LESS THAN\FORWARD SLASH HTML	
DELETED	JEEVES	GREATER THAN ACTUAL GREATER THAN	
NARRATOR	INTERNET	SYMBOL>YES, THAT WAS ALL PART OF THE	
NPC	PASSWORD	NAME, BUT SO IS...OK, LET ME START OVER"	

Namensräume

Namen in Python

- Bisher haben wir Namen immer im Zusammenhang von Variablennamen oder Funktionsnamen kennenngelernt → Für den Pythoninterpreter ist das tatsächlich das Selbe.
- Ein Name in Python ist eine Referenz auf ein bestimmtes Objekt, das im Arbeitsspeicher liegt.
- Ein Objekt kann ein einfacher Wert sein, wie etwa der Integer 5, der String 'Hallo Welt', oder die Liste [1, 'Python', ('a', 7, 9), [36.8, 86.1]].
- Aber auch jede Funktion ist ein Objekt, da eine Funktion eine Sammlung von Pythonanweisungen ist, welche im Arbeitsspeicher liegt und ebenfalls einen Namen hat.



- Alle Namen (Referenzen) die in einem Pythonprogramm existieren, werden durch den Pythoninterpreter in so genannten Namensräumen verwaltet.
- Die Namen der Standardfunktionen und Typfunktionen sind im *built-in Namespace* gespeichert und beim Starten jedes Pythonprogramms sofort verfügbar.
- Alle weiteren Variablennamen und Namen von eigenen Funktionen, welche in einem Pythonprogramm implementiert sind, werden zur Laufzeit des Programms durch den Interpreter im *global Namespace* verwaltet.

Namensräume

- Durch das Anlegen einer neuen Variable oder das Definieren einer neuen Funktion, wird der dazugehörige Name im *global Namespace* abgelegt.
- Wird ein Name im Programm aufgerufen, so sucht der Pythoninterpreter zuerst im *global Namespace* um das entsprechende Objekt im Arbeitsspeicher zu finden.
- Wird der Name nicht im *global Namespace* gefunden, wird eine Ebene darüber, im *built-in Namespace*, gesucht.
- Kann ein Name nicht gefunden werden, wirft der Interpreter einen Laufzeitfehler.

built-in Namespace

print, len, min, range, ...

global Namespace

x

tolle_funktion

y

z

```
x = 36.119
```

```
def tolle_funktion():  
    print('Ich bin eine tolle Funktion!')  
    return True
```

```
y = [149, 5]  
z = tolle_funktion()  
Ich bin eine tolle Funktion!
```

```
noch_tollere_funktion()  
NameError: name 'noch_tollere_funktion' is not defined
```

Namenskonflikte

- Der Pythoninterpreter überprüft nicht, ob ein Name bei einer Variablenzuweisung oder bei der Definition einer Funktion schon in einem der Namensräume existiert:

```
x = 'python'
x = 19

print = 5
print('Hallo welt')
TypeError: 'int' object is not callable
```

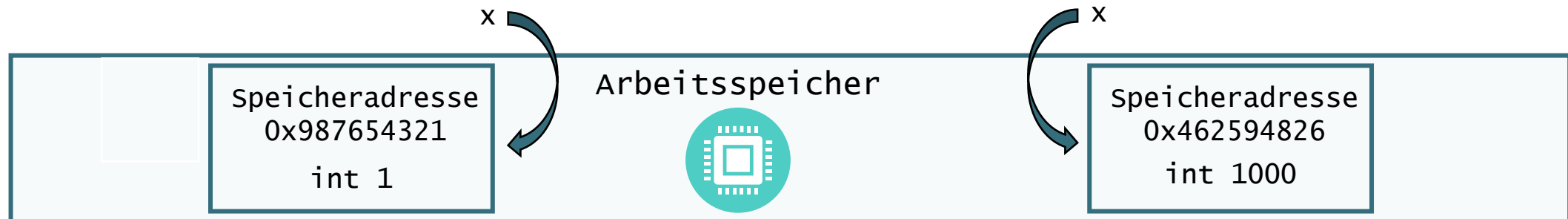


Namenskonflikte und Funktionen

```
def meine_Funktion():  
    x = 1000  
    print('Die Funktion sagt x =', x)  
  
x = 1  
meine_Funktion()  
print('Das Hauptprogramm sagt x =', x)
```

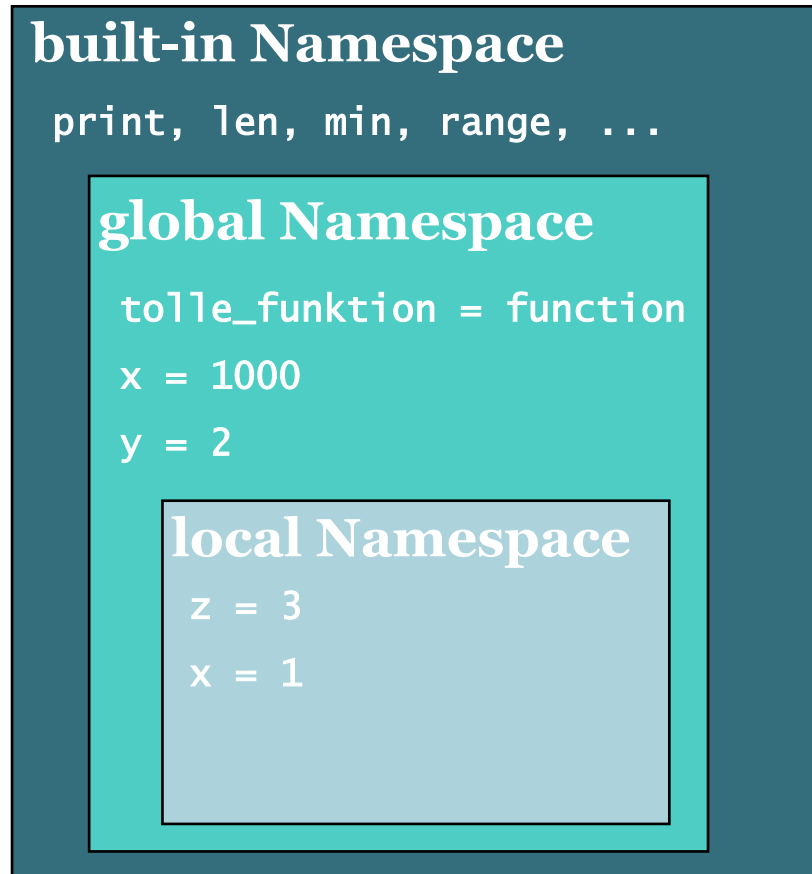
Die Funktion sagt x = 1000
Das Hauptprogramm sagt x = 1

- Erklärung: Im Arbeitsspeicher liegen zwei verschiedene und unabhängige Objekte mit dem selben Namen:

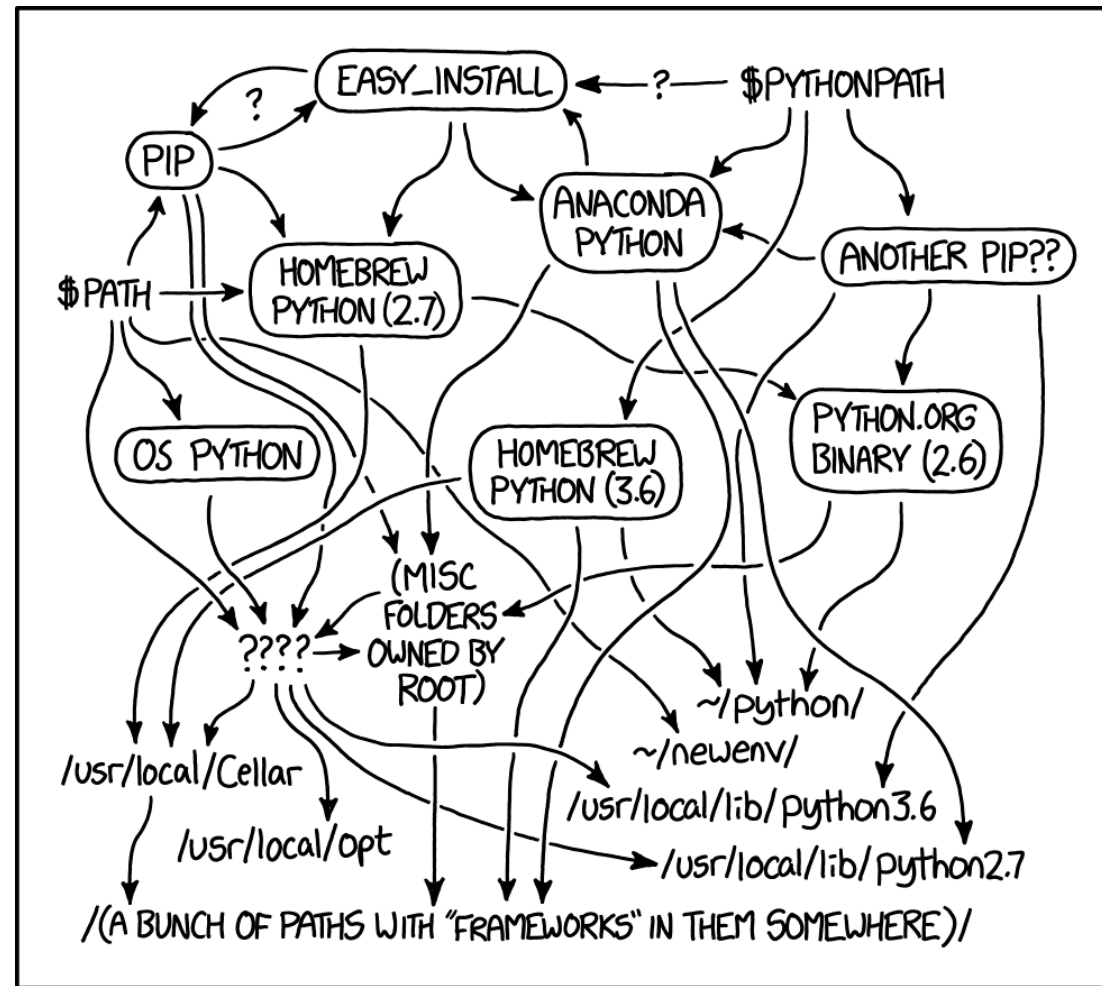


Lokale Namensräume

- Neben dem übergeordneten *build-in Namespace* und dem *global Namespace* des Hauptprogramms, erzeugt jede Funktion ihren eigenen *local Namespace*, wenn sie aufgerufen wird.
- Namen werden immer zuerst im aktuellen Namensraum gesucht. Wird ein Name im aktuellen Namensraum nicht gefunden, dann sucht der Pythoninterpreter im nächsthöheren Namensraum.



```
def tolle_funktion(z):  
    x = 1  
    print(x + y + z)  
  
x = 1000  
y = 2  
  
tolle_funktion(3)  
6
```



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

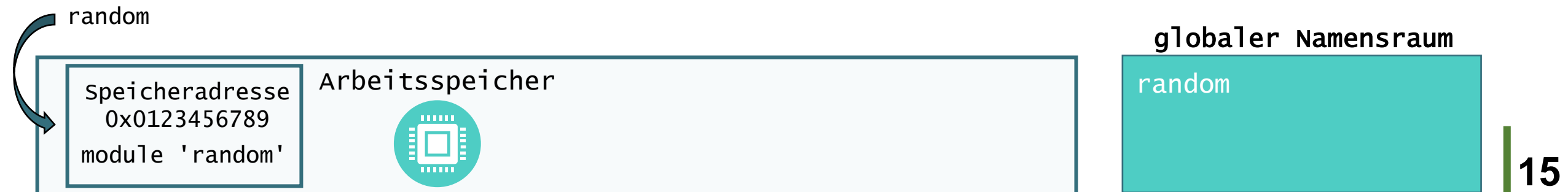
Eigene und externe Pythonmodule

Was ist eigentlich ein Modul?

- Ein Modul ist eine "Bibliothek" von Variablen und Funktionen die man in den Namensraum eines Pythonprogramms laden kann (man sagt auch: importieren) und einem dann in diesem Programm zur Verfügung stehen.
- Mit der **import** Anweisung importiert man sich den Inhalt eines Moduls in den *global Namespace* seines Programms:

```
random.randint(0,10)
NameError: name 'random' is not defined

import random
random.randint(0,10)
9
```



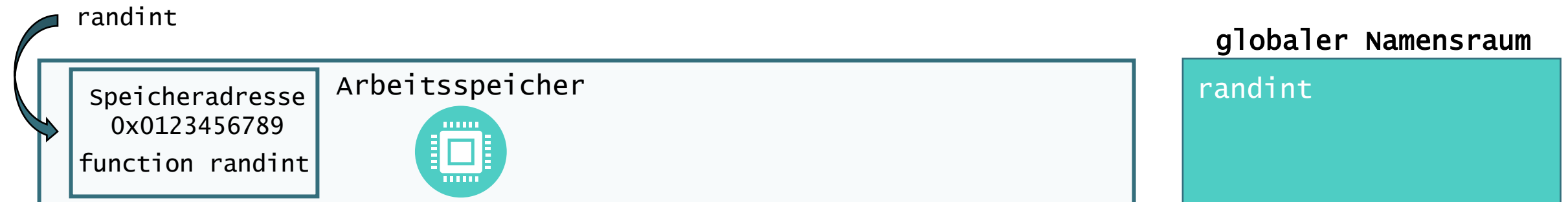
Was ist eigentlich ein Modul?

- Ein Modul ist im Prinzip selbst nur eine normale Pythondatei, die normalen Pythoncode enthält, d. h. streng genommen ist jede beliebige Pythondatei auch ein Modul.

```
import random
print(random)
<module 'random' from '/home/Emanuel/miniconda3/lib/python3.7/random.py'>
```

- Man kann mit der `from import` Anweisung auch gezielt einzelne Namen zu Variablen oder Funktionen aus einem Modul importieren, ohne den Inhalt des gesamten Moduls in den *global Namespace* zu laden:

```
from random import randint
randint(0,10)
5
```



Modul Suchpfade

- Importiert man ein Modul in ein eigenes Pythonprogramm mit der Anweisung `import NAME`, so sucht der Pythoninterpreter in folgender Reihenfolge an diesen Speicherorten nach einer gleichnamigen Datei mit der Endung `.py`:
 1. Im selben Ordner in dem das eigene Pythonprogramm gespeichert ist.
 2. In allen Ordnern die in der Betriebssystemvariable `PYTHONPATH` gespeichert sind.
 3. Im Installationspfad des Pythoninterpreters (z. B. "C:\Programme\Python39\") und dem dort liegendem Library Ordner (z. B. "C:\Programme\Python39\Lib\").
- Die Listenvariable `path` aus dem Modul `sys` enthält alle Suchpfade des Pythoninterpreters:

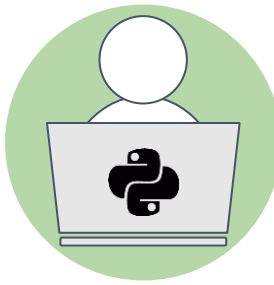
```
import sys
print(sys.path)
['C:\programs\Python39\',
 'C:\programs\Python39\Lib\',
 'C:\programs\Python39\site-packages\']
```



Aufgabe:

Besucht die offizielle Dokumentation von Python und informiert euch über ein selbstgewähltes Standardmodul: <https://docs.python.org/3/library/>

Wofür kann es eingesetzt werden?



Aufgabe:

Schreibt ein Programm, welches gezielt die Potenzfunktion und die Fakultätsfunktion aus eurem selbstgeschriebenen Programm von der Übungsaufgabe "32_mathematische_Funktionen" importiert. Danach soll das Programm auch das gesamte von euch geschriebene Modul von der Übungsaufgabe "33_Steganographie" importieren. Anschließend soll euer Programm die importierten Funktionen aus beiden Übungsaufgaben testen.

Tipp: Erhaltet ihr beim Ausführen eures Programms einen `ModuleNotFoundError` Laufzeitfehler, dann überprüft die Suchpfade eures Pythoninterpreters mit Hilfe des `sys` Moduls.

Hinweis:

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

Externe Module installieren

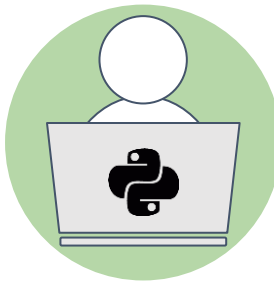
- Jede Pythoninstallation kommt bereits mit einer großen Zahl von Standardmodulen, welche man sich nach Bedarf in seine eigenen Programme importieren kann.
- Nichtsdestotrotz gibt es eine Vielzahl von externen Modulen, entwickelt durch verschiedene einzelne Programmierer oder Teams, welche viele sehr hilfreiche Funktionen und Variablentypen bereitstellen, die nicht in der Pythoninstallation enthalten sind.
- Diese externen Module lassen sich einfach über das Programm *pip* herunterladen und installieren.

Externe Module installieren

- Normalerweise wird *pip* zusammen mit Python installiert, falls nicht: <https://pip.pypa.io/en/stable/installation/>
- *pip* ist ein Verwaltungsprogramm für externe Pythonmodule, welches diese und alle abhängigen Module automatisch herunterlädt und installiert.

```
C:\Benutzer\Emanuel> pip install PAKETNAME
```

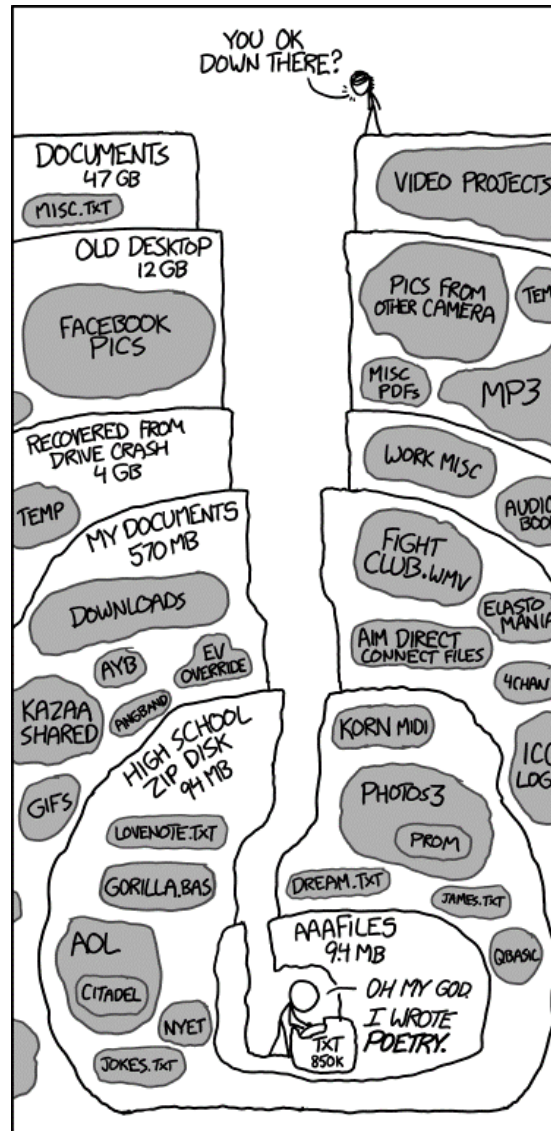
- *pip* greift auf den PyPI (Python Package Index) zu: <https://pypi.org/>



Aufgabe:

Installiert mit Hilfe von *pip* die Module *numpy*, *pandas* und *matplotlib*. Testet anschließend in einer interaktiven Pythonshell, ob die Installationen erfolgreich waren.

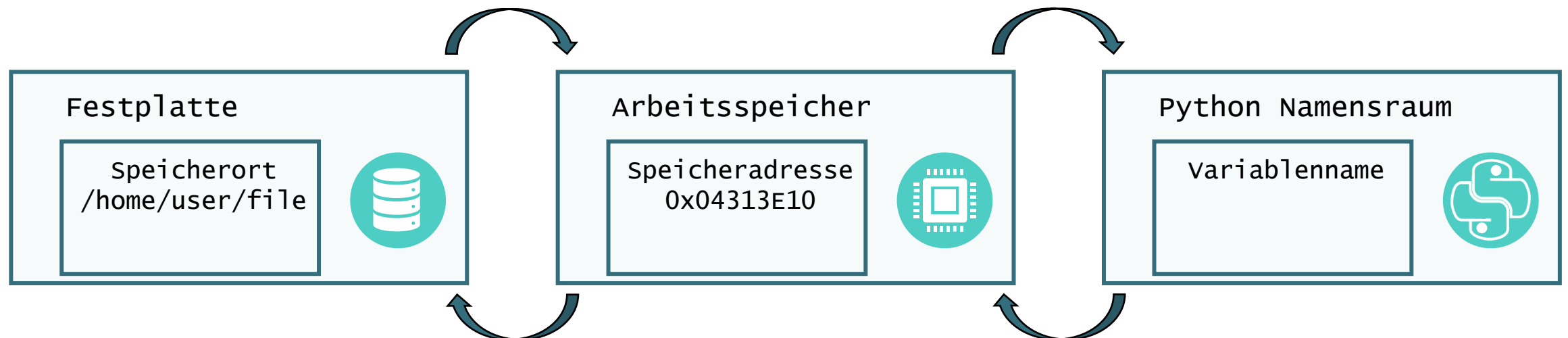
Besucht die Seite des Python Package Index (<https://pypi.org/>) und informiert euch wofür diese drei Module gedacht sind.



Textdateien lesen und schreiben

Das Datei-Objekt

- Wenn wir über ein Pythonprogramm eine Datei, die auf der Festplatte unseres PC gespeichert ist, verarbeiten wollen (lesen und/oder schreiben), dann geht das immer nur über den Arbeitsspeicher.
- Beim einlesen einer Datei wird ihr Inhalt zuerst in den Arbeitsspeicher kopiert dann erhält sie vom Pythonprogramm einen Namen der in einem der Namensräume landet.
→ Es wurde ein neues Objekt erstellt, welches eine Datei (und ihren Inhalt) repräsentiert.
- Ändern wir den Inhalt einer eingelesenen Datei, so ändern wir zuerst nur den Inhalt im Arbeitsspeicher. Erst durch eine explizite Pythonanweisung werden diese Änderungen auch in die Datei auf der Festplatte übernommen.



Das Datei-Objekt

- Um ein Datei-Objekt zu erzeugen nutzt man die Funktion `open()`, welche zwei Argumente erwartet:

`open(Dateiname, Modus)`

String der den Pfad zur Datei beschreibt,
z. B. "C:/Users/Emanuel/text.txt"

String der den Bearbeitungsmodus
der Datei definiert, z. B. "r"

Modus	Erklärung
r	Die Datei wird ausschließlich zum Lesen geöffnet. Falls die Datei nicht existiert erhält man beim Öffnen, eine Fehlermeldung
w	Es wird eine Datei zum Schreiben erstellt. Falls eine Datei mit dem gleichen Namen schon existiert, wird deren Inhalt gelöscht und neu beschrieben.
a	Die Datei ist zum Anhängen neuer Daten bestimmt. Der bisherige Inhalt wird nicht gelöscht, es wird am Ende des Inhalts der Datei weiter geschrieben.

Das Datei-Objekt

- Erzeugt man ein Datei-Objekt im Schreibmodus ('w'), dann wird unter dem gegebenem Pfad eine leere Datei mit diesem Namen angelegt.

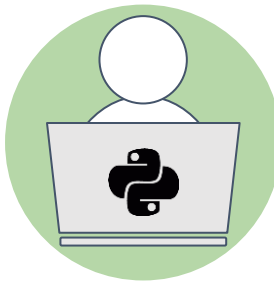
Achtung: Existiert unter dem gegebenem Pfad schon eine Datei mit dem selben Namen, so wird der Inhalt dieser Datei unwiderruflich gelöscht!

```
y = open('C:/Users/Emanuel/neue_textdatei.txt', 'w')
y.close()

x = open('C:\\Users\\Emanuel\\neue_textdatei.txt', 'r')
x.close()
```

- Um eventuelle Änderungen am Inhalt einer Datei vom Datei-Objekt im Arbeitsspeicher auf die Datei auf der Festplatte zu übertragen, muss man mit der Typfunktion `close()` das Datei-Objekt "schließen".
- Ein geschlossenes Datei-Objekt kann nicht weiter verarbeitet werden, sondern müsste durch einen neuen Funktionsaufruf von `open()` wieder neu im Arbeitsspeicher erzeugt werden.

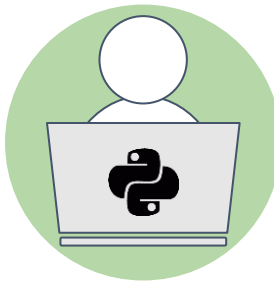
Datei-Objekte öffnen und schließen



Aufgabe:

Ladet euch die Datei "34_Python_Mythologie.txt" aus dem Übungsmaterialordner herunter und führt folgende Aktionen aus:

1. Überprüft ob sich Text in der Datei befindet, z. B. mit Hilfe eines Texteditors. Schließt den Texteditor dann wieder.
2. Erzeugt ein Datei-Objekt dieser Datei im Lesemodus ('r') in Python und weist sie einer Variable zu.
3. Findet etwas über den Typ und den Inhalt dieser Variable heraus.
4. Schließt das Datei-Objekt in Python wieder und erzeugt es erneut im Lesemodus ('w').
5. Hat sich etwas am Typ oder Inhalt der Variable geändert?
6. Überprüft den Inhalt der heruntergeladenen Text.
7. Schließt das Datei-Objekt in Python wieder.



Aufgabe:

Legt euch einen neuen Ordner namens "voll" an.

Erstellt ein Programm, welches in diesem neuen Ordner 10000 leere Dateien erstellt, wobei die erste Datei "1.txt", die zweite Datei "2.txt", die dritte Datei "3.txt", usw. heißen soll.

Hinweis:

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

Dateien lesen und schreiben - Beispiele

- Nach dem Erzeugen eines Datei-Objekt, hängt von dessen Modus ab, welche Typfunktionen auf diesem Datei-Objekt ausgeführt werden können:

Lesemodus ('r')

Typfunktion	Erklärung
read()	Der gesamte Inhalt der Datei, wird als String zurückgegeben.
readline()	Die nächste Zeile (bis zum nächsten Zeilenumbruch) der Datei wird als String zurückgegeben.
readlines()	Der gesamte Inhalt der Datei, wird zeilenweise eingelesen und als eine Liste von Strings zurückgegeben.
close()	Das Datei-Objekt wird geschlossen.

Schreibmodus ('w')

Erweiterungsmodus ('a')

Typfunktion	Erklärung
write(x)	Das Argument x wird als String in das Datei-Objekt geschrieben (aber noch nicht auf die Datei auf der Festplatte).
flush()	Alle Änderungen am Inhalt des Datei-Objekts werden auf die Datei auf der Festplatte übertragen. Das Datei-Objekt wird <u>nicht</u> geschlossen.
close()	Das Datei-Objekt wird geschlossen und alle Änderungen an dessen Inhalt werden auf die Datei auf der Festplatte übertragen.

Dateien lesen und schreiben - Beispiele

- Mit der `read()` Funktion lässt sich der gesamte Inhalt einer Datei als String zurückgeben.

```
output_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'w')
output_datei.write('willkommen bei\n')
output_datei.write('der fabelhaften welt\n')
output_datei.write('der Pythonprogrammierung!')
output_datei.close()

input_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'r')
x = input_datei.read()
print(x)
willkommen bei
der fabelhaften welt
der Pythonprogrammierung!

y = input_datei.read()
print(y)
```

Dateien lesen und schreiben - Beispiele

- Mit der `readline()` Funktion lassen sich Zeilen des Inhalts einer Funktion einzeln einlesen:

```
output_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'w')
output_datei.write('willkommen bei\n')
output_datei.write('der fabelhaften welt\n')
output_datei.write('der Pythonprogrammierung!')
output_datei.close()

input_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'r')
x = input_datei.readline()
print(x)
willkommen bei

x = input_datei.readline()
print(x)
der fabelhaften welt

x = input_datei.readline()
print(x)
der Pythonprogrammierung!
```

Dateien lesen und schreiben - Beispiele

- Mit der `readlines()` Funktion wird der gesamte Dateiinhalt zeilenweise in einer Liste eingelesen:

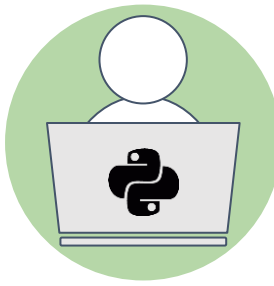
```
output_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'w')
output_datei.write('willkommen bei\n')
output_datei.write('der fabelhaften welt\n')
output_datei.write('der Pythonprogrammierung!')
output_datei.close()

input_datei = open('C:/Users/Emanuel/neue_textdatei.txt', 'r')
x = input_datei.readlines()
print(x)
['willkommen bei\n', 'der fabelhaften welt\n', 'der Pythonprogrammierung!']

print(x[2])
der Pythonprogrammierung!
```

- Braucht man das Datei-Objekt nur um einmalig den Dateiinhalt einzulesen, kann man auch folgendes schreiben:

```
x = open('C:/Users/Emanuel/neue_textdatei.txt', 'r').readlines()
print(x[2])
der Pythonprogrammierung!
```

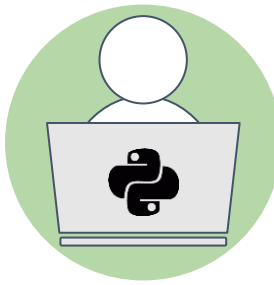



Aufgabe:

Schreibt ein Programm, dass eine Person nach einer ganzen Reihe von Daten fragt, wie z. B. Name, Alter, Geburtsort, Lieblingsessen. Nach der Eingabe der Daten, soll das Programm diese in einer Datei auf eurer Festplatte speichern. Dann stellt das Programm weitere Fragen zur Person, wie z. B. Lieblingsfarbe, Hobby, Name des Haustiers und speichert auch diese Angaben in die selbe Datei. Im Anschluss liest das Programm die erstellte Datei ein und gibt die angegebenen Daten der Person zur Kontrolle noch einmal auf dem Bildschirm wieder aus.

Hinweis:

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



Aufgabe:

Ladet euch die Datei "42_Farben.txt" aus dem Übungsmaterialordner herunter.

Schreibt ein Programm, welches die Datei einliest und die Häufigkeit der einzelnen in der Datei beschriebenen Farben ermittelt. Im Anschluss gibt das Programm eine Gesamtübersicht der gezählten Farben, sowie die häufigste und die seltenste Farbe auf dem Bildschirm zurück.

Hinweis:

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

Zuverlässigkeit beim Lesen und Schreiben

- Wenn der Pfad zu einer Datei ungültig ist, z. B. durch einen Tippfehler oder die Datei/der Pfad existiert nicht, dann kommt es zu einem Laufzeitfehler → das Programm bricht ab.
- So kann es schnell zu ungewolltem Datenverlust kommen und man muss sich immer überlegen, ob man die Arbeit an einem Datei-Objekt durch eine `try except` Anweisung absichern sollte:

```
daten = 'Sehr wichtige Daten'
try:
    output_datei = open('/pfad/existiert/nicht/file_auch_nicht.dat', 'w')
    output_datei.write(daten)
    output_datei.close()
except FileNotFoundError:
    output_datei = open('/tmp/backup.dat', 'w')
    output_datei.write(daten)
    output_datei.close()
```

- Öffnet man ein Datei-Objekt mit der Anweisung `with as`, so wird das Datei-Objekt automatisch geschlossen, wenn der `with` Anweisungsblock verlassen wird:

```
with open('/home/python/wichtige_daten.dat', 'w') as output:  
    output.write('Dieses Datei-Objekt wird auch ohne close() geschlossen.')
```

`output.write('Ausserhalb des with Blocks ist das Datei-Objekt schon geschlossen')`
`ValueError: I/O operation on closed file.`

- Die `write` Funktion akzeptiert nur genau ein Argument, welches vom Typ `String` sein muss:

```
l = [1, 2, 3]
with open('/home/python/test.txt', 'w') as output:
    output.write('Liste:', l, '1tes Element:', l[0], 'Summe:', l[0] + l[1] + l[2])

TypeError: TextIOWrapper.write() takes exactly one argument (6 given)
```

- Die `print` Funktion akzeptiert beliebig viele Argumente und wandelt diese auch alle automatisch in `Strings` um:

```
l = [1, 2, 3]

print('Liste:', l, '1tes Element:', l[0], 'Summe:', l[0] + l[1] + l[2])
Liste: [1, 2, 3] 1tes Element: 1 Summe: 6
```

- Anstatt jedes Nicht-String Argument manuell in einen String umzuwandeln, kann man so genannte f-Strings (format Strings) mit Platzhaltern {} benutzen:

```
l = [1, 2, 3]
with open('/home/python/test.txt', 'w') as output:
    output.write(f'Liste: {l} 1tes Element: {l[0]} Summe: {l[0] + l[1] + l[2]}')
```

- f-Strings lassen sich überall einsetzen, wo man auch normale Strings verwenden kann:

```
l = [1, 2, 3]

print(f'Liste: {l} 1tes Element: {l[0]} Summe: {l[0] + l[1] + l[2]}')
Liste: [1, 2, 3] 1tes Element: 1 Summe: 6
```

STRATEGIES FOR FULL-WIDTH JUSTIFICATION

THEIR THINDUS PAPER
ON THE RELATIONSHIP
BETWEEN
DEINDUSTRIALIZATION
AND THE GROWTH OF

GIVING UP

THEIR THINDUS PAPER
ON THE RELATIONSHIP
B E T W E E N
DEINDUSTRIALIZATION
AND THE GROWTH OF

LETTER
SPACING

THEIR THINDUS PAPER
ON THE RELATIONSHIP
BETWEEN DEINDUS -
TRIALIZATION AND THE
GROWTH OF ECOLOGICAL


HYPHENATION

THEIR THINDUS PAPER
ON THE RELATIONSHIP
BETWEEN
DEINDUSTRIALIZATION
AND THE GROWTH OF

STRETCHING

THEIR THINDUS PAPER
ON THE RELATIONSHIP
BETWEEN CRAP LIKE
DEINDUSTRIALIZATION
AND THE GROWTH OF

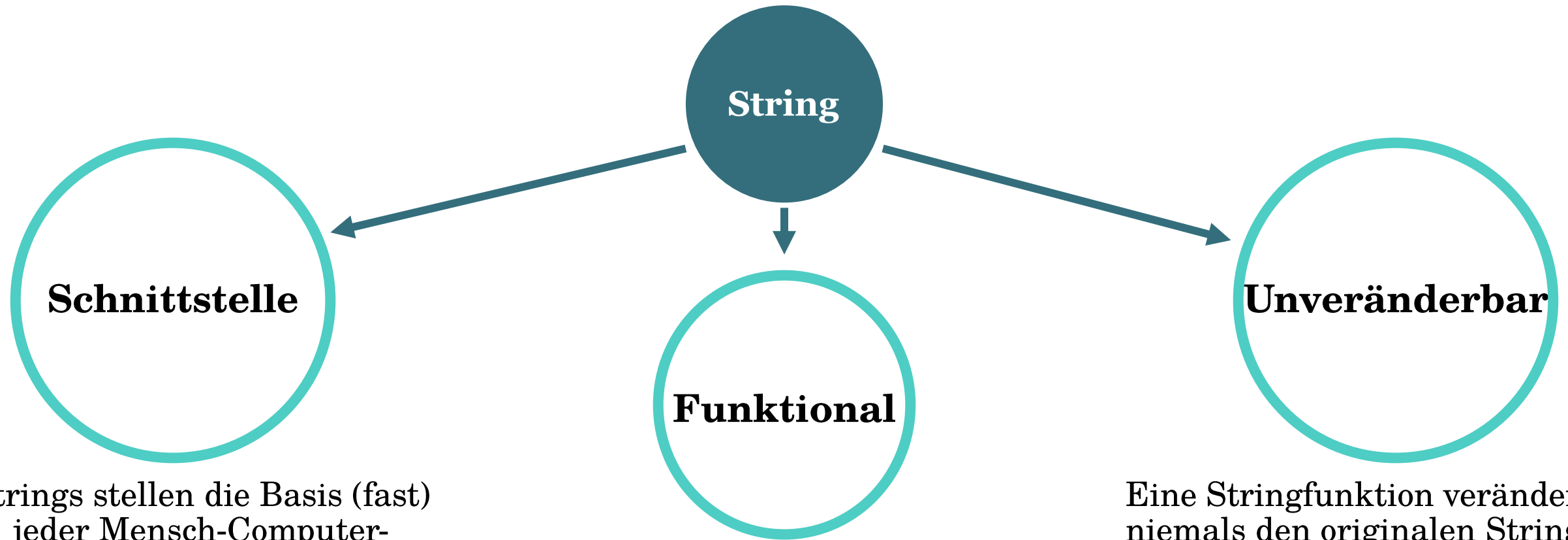
FILLER

THEIR THINDUS PAPER
ON THE RELATIONSHIP
BETWEEN 
DEINDUSTRIALIZATION
AND THE GROWTH OF

SNAKES

Verarbeitung von Strings

Warum Strings?



Strings stellen die Basis (fast) jeder Mensch-Computer-Kommunikation dar.

In Python besitzen Objekte vom Typ String bereits eine ganze Reihe von Funktionen um Texte zu analysieren oder zu verändern.

Eine Stringfunktion verändert niemals den originalen String, sondern gibt eine Kopie des Strings in veränderter Form wieder.

Strings sind unveränderbare Datentypen

- Eine Typfunktion verändert niemals den originalen String, sondern erzeugt eine Kopie des Strings in veränderter Form:

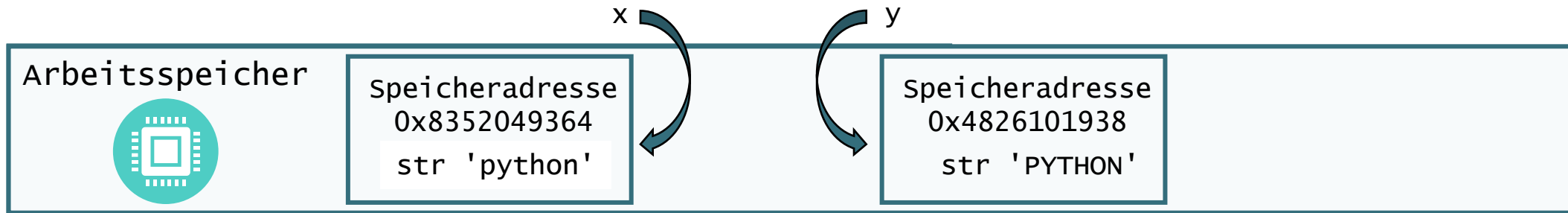
```
x = 'python'  
x.upper()
```



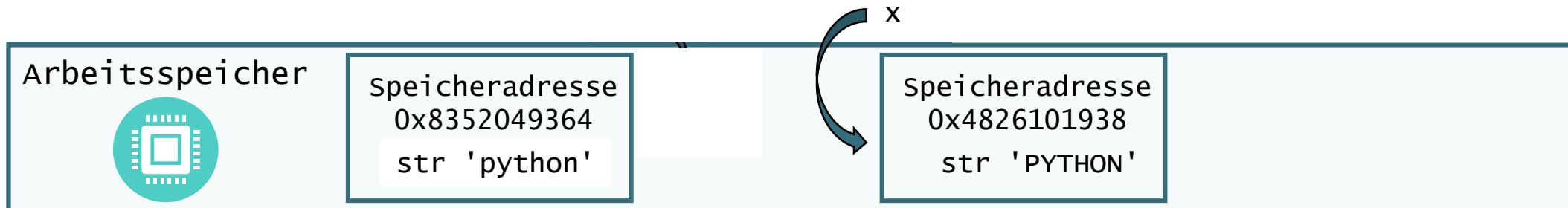
Strings sind unveränderbare Datentypen

- Das heißt, nachdem wir eine Typfunktion auf einen String angewendet haben müssen wir die erzeugte und veränderte Kopie einer Variable zuweisen, wenn wir mit ihr arbeiten wollen:

```
x = 'python'  
y = x.upper()
```



```
x = 'python'  
x = x.upper()
```



Typfunktionen von Strings - Beispiele

- Strings besitzen in Python bereits eine Vielzahl an Typfunktionen, z. B. zur Änderung der Schreibweise:

Funktion	Erklärung
capitalize()	Wandelt den ersten Buchstaben in einen großen und alle folgenden in kleine um.
lower()	Wandelt alle Buchstaben in Kleinbuchstaben um.
upper()	Wandelt alle Buchstaben in Großbuchstaben um.

```
'die fabelhafte welt der Pythonprogrammierung'.capitalize()  
Die fabelhafte welt der pythonprogrammierung
```

```
'die fabelhafte welt der Pythonprogrammierung'.lower()  
die fabelhafte welt der pythonprogrammierung
```

```
'die fabelhafte welt der Pythonprogrammierung'.upper()  
DIE FABELHAFTE WELT DER PYTHONPROGRAMMIERUNG
```

Typfunktionen von Strings - Beispiele

- Strings besitzen in Python bereits eine Vielzahl an Typfunktionen, z. B. um einfache logische Tests durchzuführen:

Funktion	Erklärung
endswith(suffix)	Liefert TRUE, falls der String mit der Zeichenkette suffix endet.
isalnum()	Liefert TRUE, falls der String nicht leer ist und alle Zeichen alphanummerisch sind.
isalpha()	Liefert TRUE, falls alle Zeichen des Strings Buchstaben sind.
isdigit()	Liefert TRUE, falls alle Zeichen des Strings Zahlen sind.

```
'Hallo welt!'.endswith('!')
True

'Hallo welt!'.endswith('!t!')
True

'Hallo welt!!!1'.isalpha()           #Sonderzeichen, Leerzeichen und Ziffern sind keine Buchstaben
False

'Hallowelt'.isalpha()
True

'1990'.isdigit()
True

'C8H10N4O2'.isalnum()
True
```

Typfunktionen von Strings - Beispiele

- Strings besitzen in Python bereits eine Vielzahl an Typfunktionen, z. B. zum Entfernen von Stringelementen oder Aufspalten von Strings:

Funktion	Erklärung
strip(chars)	Am Anfang und Ende des Strings werden Buchstaben aus chars oder bei fehlendem Argument Whitespaces entfernt.
split(sep)	Der String wird in eine Liste von Teilstrings aufgeteilt. Dabei wird sep als Trennsymbol verwendet, wenn es fehlt wird nach allen Whitespaces getrennt (Leerzeichen, Tabulator, Zeilenumbruch).

[illegible]

Typfunktionen von Strings - Beispiele

- Strings besitzen in Python bereits eine Vielzahl an Typfunktionen, z. B. zum Suchen und Ersetzen von Teilstrings:

Funktion	Erklärung
count(sub)	Liefert Anzahl der Vorkommen der Zeichenkette sub.
replace(old, new)	Vorkommen der Zeichenkette old werden durch new ersetzt.

```
t = 'Wir sind die Ritter die immer Ni sagen! Ni ni ni ni ni!'
t.count('Ni')
2

t.count('ni')                #Groß-\Kleinschreibung beachten
4

t.lower().replace('ni', 'nö') #Typfunktionen lassen sich auch verketteten
wir sind die ritter die immer nö sagen! nö nö nö nö nö!

t.replace(' ', '')
WirsinddieRitterdieimmerNisagen!Nininini!
```

Das Gelernte auf strukturierte Textdateien anwenden

Textdatei

Name	Vorname	Alter	Rolle
Chapman	Graham	48	König Arthur, Wächter, Stimme Gottes
Cleese	John	66	Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam	Terry	65	Knappe Patsy, Sir Bors, Grüner Ritter
Idle	Eric	63	Sir Robin, Diener Concord, Bruder Maynard
Jones	Terry	65	Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin	Michael	64	Sir Galahad, Dennis, Herr des Sumpfschlosses

```
with open('/home/dokumente/monty_python.csv') as inputFile:
    for zeile in inputFile:
        print(zeile.split('\t'))
```

```
['Name', 'Vorname', 'Alter', 'Rolle\n']
['Chapman', 'Graham', '48', 'König Arthur, wächter, Stimme Gottes\n']
['Cleese', 'John', '66', 'Sir Lancelot, Tim der Zauberer, Schwarzer Ritter\n']
['Gilliam', 'Terry', '65', 'Knappe Patsy, Sir Bors, Grüner Ritter\n']
['Idle', 'Eric', '63', 'Sir Robin, Diener Concord, Bruder Maynard\n']
['Jones', 'Terry', '65', 'Sir Bedevere, Prinz Herbert, Landarbeiterin\n']
['Palin', 'Michael', '64', 'Sir Galahad, Dennis, Herr des Sumpfschlosses\n']
```

Das Gelernte auf strukturierte Textdateien anwenden

Textdatei

Name	Vorname	Alter	Rolle
Chapman	Graham	48	König Arthur, Wächter, Stimme Gottes
Cleese	John	66	Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam	Terry	65	Knappe Patsy, Sir Bors, Grüner Ritter
Idle	Eric	63	Sir Robin, Diener Concord, Bruder Maynard
Jones	Terry	65	Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin	Michael	64	Sir Galahad, Dennis, Herr des Sumpfschlosses

```
schauspieler_liste = open('/home/dokumente/monty_python.csv').readlines()[1:]
for element in schauspieler_liste:
    if element.split('\t')[2] <= 63:
        print(element)
```


Das Gelernte auf strukturierte Textdateien anwenden

Textdatei

Name	Vorname	Alter	Rolle
Chapman	Graham	48	König Arthur, Wächter, Stimme Gottes
Cleese	John	66	Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam	Terry	65	Knappe Patsy, Sir Bors, Grüner Ritter
Idle	Eric	63	Sir Robin, Diener Concord, Bruder Maynard
Jones	Terry	65	Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin	Michael	64	Sir Galahad, Dennis, Herr des Sumpfschlosses

```
schauspieler_liste = open('/home/dokumente/monty_python.csv').readlines()[1:]
for element in schauspieler_liste:
    if int(element.split('\t')[2]) <= 63:
        print(element)
```

```
Chapman Graham 48      König Arthur, Wächter, Stimme Gottes
Idle      Eric  63      Sir Robin, Diener Concord, Bruder Maynard
```

Das Gelernte auf strukturierte Textdateien anwenden

Textdatei

Name	Vorname	Alter	Rolle
Chapman	Graham	48	König Arthur, Wächter, Stimme Gottes
Cleese	John	66	Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam	Terry	65	Knappe Patsy, Sir Bors, Grüner Ritter
Idle	Eric	63	Sir Robin, Diener Concord, Bruder Maynard
Jones	Terry	65	Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin	Michael	64	Sir Galahad, Dennis, Herr des Sumpfschlosses

```
schauspieler_liste = open('/home/dokumente/monty_python.csv').readlines()[1:]
for element in schauspieler_liste:
    if 'Sir' in element.split('\t')[-1]:
        print(element)
```

```
Cleese John 66 Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam Terry 65 Knappe Patsy, Sir Bors, Grüner Ritter
Idle Eric 63 Sir Robin, Diener Concord, Bruder Maynard
Jones Terry 65 Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin Michael 64 Sir Galahad, Dennis, Herr des Sumpfschlosses
```

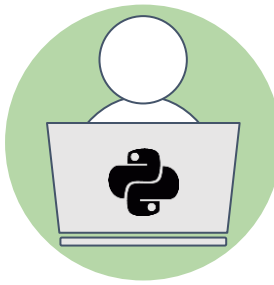
Das Gelernte auf strukturierte Textdateien anwenden

Textdatei

Name	Vorname	Alter	Rolle
Chapman	Graham	48	König Arthur, Wächter, Stimme Gottes
Cleese	John	66	Sir Lancelot, Tim der Zauberer, Schwarzer Ritter
Gilliam	Terry	65	Knappe Patsy, Sir Bors, Grüner Ritter
Idle	Eric	63	Sir Robin, Diener Concord, Bruder Maynard
Jones	Terry	65	Sir Bedevere, Prinz Herbert, Landarbeiterin
Palin	Michael	64	Sir Galahad, Dennis, Herr des Sumpfschlosses

```
summeAlter = 0
schauspieler_liste = open('/home/dokumente/monty_python.csv').readlines()[1:]
for element in schauspieler_liste:
    summeAlter = int(element.split('\t')[2])

print(summeAlter)
371
```



Aufgabe:

Ladet euch die Datei "44_Messwerte.txt" aus dem Übungsmaterialordner herunter. Schreibt ein Programm, welches die Datei einliest, die folgenden Werte berechnet und anschließend auf dem Bildschirm ausgibt:

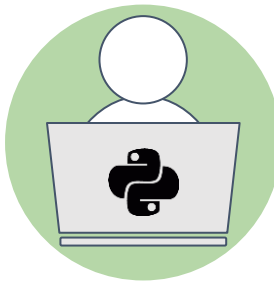
- Die Anzahl aller Messungen die mit einem Laser durchgeführt wurden.
- Den kleinsten Messwert der mit einem Radiometer gemessen wurde.
- Der größte Messwert der nicht von einer Lasermessung stammt.
- Den Mittelwert und den Median aller Lasermessungen.

Tipp 1: Schaut euch die Datei mit den Messwerten gut an, um ihren Aufbau und ihre Struktur zu verstehen.

Tipp 2: In der Übungsaufgabe "31_weitere_Funktionen" musstet ihr bereits eine Mittelwerts- und eine Medianfunktion schreiben. Importiert und nutzt diese, anstatt deren Code zu kopieren oder neuzuschreiben.

Hinweis:

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



Aufgabe:

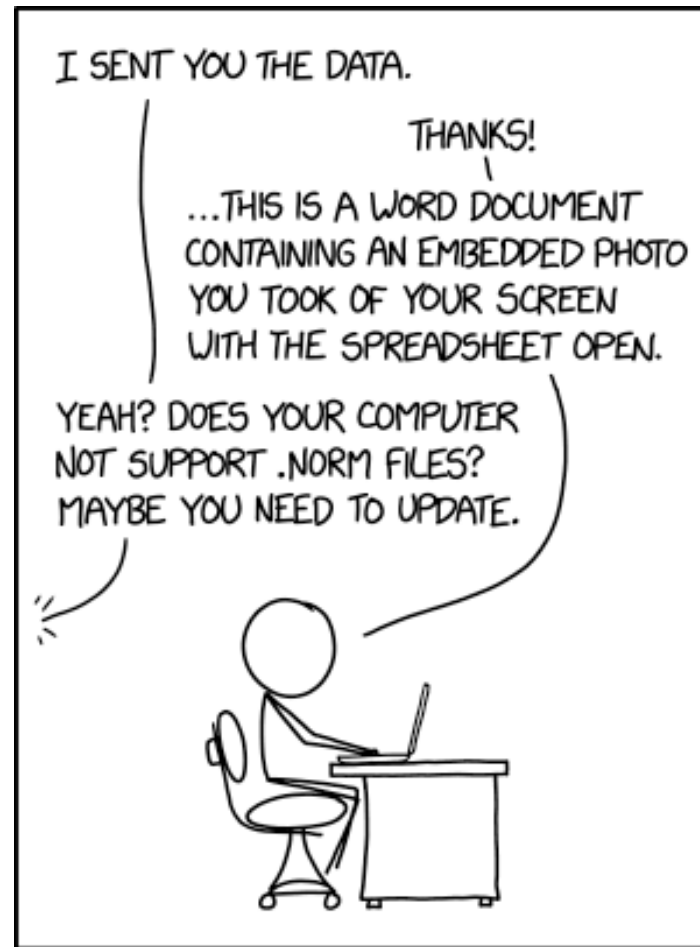
Schreibt ein Programm, welches eine Datei mit Fließtext einlesen und nach folgenden Mermalen analysieren kann:

- Es kann die Anzahl der Sätze im Text bestimmen.
- Es kann Jahreszahlen aus dem Text herausfiltern (zur Vereinfachung nehmen wir an das Jahreszahlen immer aus genau vier Ziffern bestehen).
- Es kann die herausgefilterten Jahreszahlen in Kategorien nach Jahrhunderten einsortieren.
Wenn z. B. im gesamten Text die fünf Jahreszahlen 1990, 1746, 1500, 1992 und 1842 auftauchen, dann werden sie wie folgt kategorisiert:
 - 1500er : 1500
 - 1700er : 1746
 - 1800er : 1842
 - 1900er : 1990, 1992

Testet euer Programm an den beiden Textdateien "46_Ada_Lovelace.txt" und "47_Monty_Python.txt" aus dem Übungsmaterialordner.

Hinweis:

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



SINCE EVERYONE SENDS STUFF THIS WAY ANYWAY, WE SHOULD JUST FORMALIZE IT AS A STANDARD.

Beliebige Daten speichern & wiederverwenden

Speichern und Wiederverwenden beliebiger Datentypen

- Da Datei-Objekte die von der open Funktion erzeugt werden Daten immer nur in Form von Strings einlesen und auch schreiben können, gehen eventuell wichtige Informationen verloren:

```
wichtigeListe = ['E=mc2', 19, 36.149]
with open('/home/python/wichtigeListe.txt', 'w') as output:
    output.write(str(wichtigeListe))

wichtigeListe = open('/home/python/wichtigeListe.txt').read()
type(wichtigeListe)
<class 'str'>

wichtigeListe[2]
m

list(wichtigeListe)
['[', '"', 'E', '=', 'm', 'c', '2', '"', ',', ' ', '1', '9', ',', ' ', '3', '6',
',', '1', '4', '9', ']']
```

- Die Wiederherstellung der ursprünglichen Liste und der eigentlichen Datentypen ist unter Umständen so nicht mehr möglich.

Das Modul JSON

- Mit Hilfe des Moduls `json` lassen sich (fast) beliebige Objekte als Datei auf der Festplatte speichern und auch direkt in ihrer ursprünglichen Form wieder in ein Pythonprogramm laden.
- Zum Speichern eines Objekts verwendet man die `dump` Funktion:

```
import json
wichtigeDaten = {'Formel': 'E=m*c**2', 'Messwerte': [5.19, 36.00, 11.74], 'Zahl': 8}

json.dump(wichtigeDaten, open('/home/Documents/wichtigeDaten.json', 'w'))
```

- Zum Laden eines Objekts verwendet man die `load` Funktion:

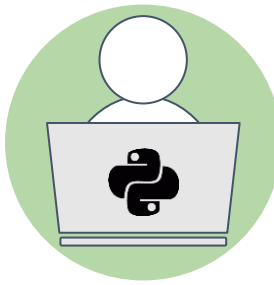
```
import json
wichtigeDaten = json.load(open('/home/Documents/wichtigeDaten.json', 'r'))

type(wichtigeDaten)
<class 'dict'>

type(wichtigeDaten['Messwerte'][0])
<class 'float'>
```


Das json Dateiformat

- Im Prinzip ist eine Datei im `json` Format eine einfache JavaScript Datei die schon fast mit der Python Syntax kompatibel ist.
- Das `json` Modul versteht alle fundamentale Datentypen (Kollektionen, Zahlen, Wahrheitswerte, ...) ineinander verschachtelte Datentypen und generell jede Form eines Objekts (z. B. auch Funktionen).
- Es ist ein sehr beliebtes Datenformat zur Kommunikation zwischen Webservern und Hosts (Webanwendungen, mobile Apps, ...).
- Das Format ist prinzipiell menschenlesbar und relativ leicht zu verarbeiten.



Aufgabe:

Schreibt ein Programm, das Telefonnummern verwaltet. Die Besonderheit hier soll die Benutzerschnittstelle sein. Nach jeder Aktion erscheint auf dem Bildschirm ein Textmenü und der Nutzer kann eine der folgenden Programmaktionen wählen:

- Nummer suchen: Der Benutzer gibt einen Namen ein und erhält als Antwort die Telefonnummer, falls sie gespeichert ist. Ansonsten gibt es eine Nachricht, dass der Name nicht bekannt sei.
- Nummer eintragen: Der Benutzer wird nach einem Namen und der zugehörigen Telefonnummer gefragt.
- Telefonbuch ausgeben: Auf dem Bildschirm erscheint eine gut lesbare Übersicht mit allen gespeicherten Namen und Telefonnummern.
- Programm beenden: Das aktuelle Telefonbuch wird gespeichert und das Programm beendet.

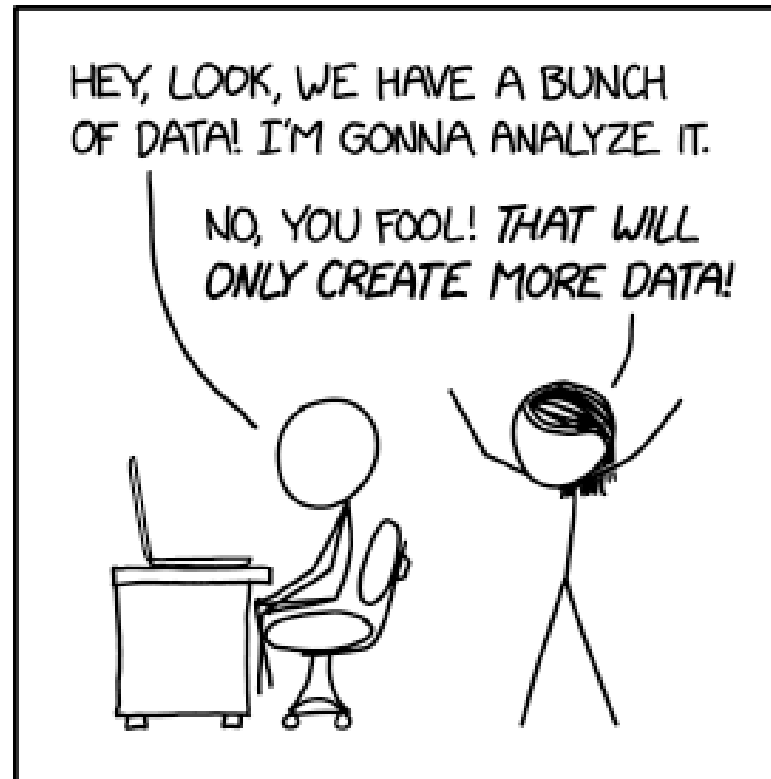
So lange das Programm nicht über die Aktion "Programm beenden" geschlossen wird, fragt es immer wieder nach der nächsten Programmaktion.

Jedes Mal wenn das Programm gestartet wird, soll es eine eventuell schon vorhandene Telefonbuchdatei laden, so dass vorher eingegebene Telefonnummern schon abgefragt werden können.

Tipp: Schreibt für jede Programmaktion eine eigene Funktion und nutzt zum speichern der Telefonbuchdatei das `json` Format.

Hinweis:

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

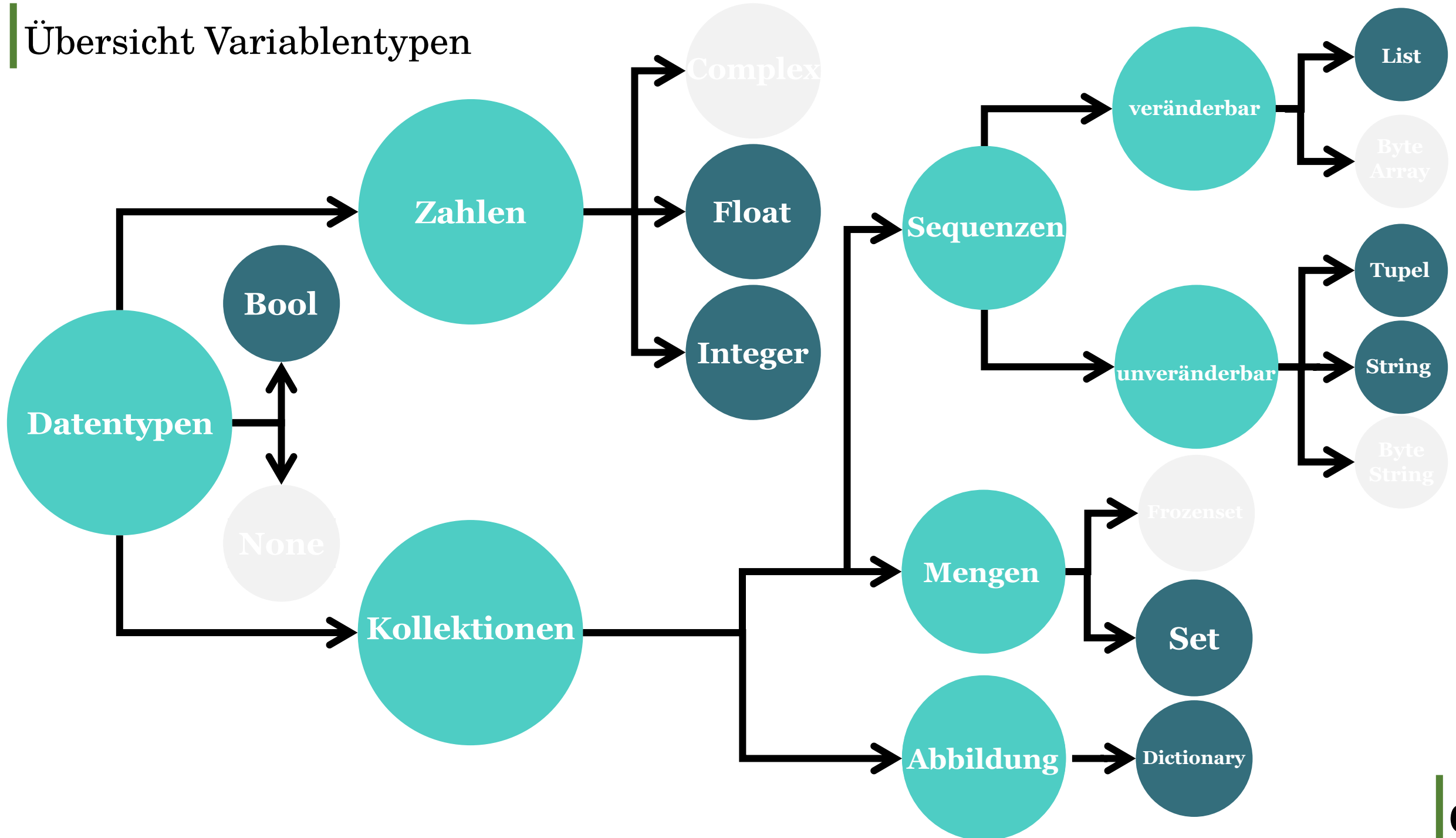


Datenanalyse mit NumPy & Pandas

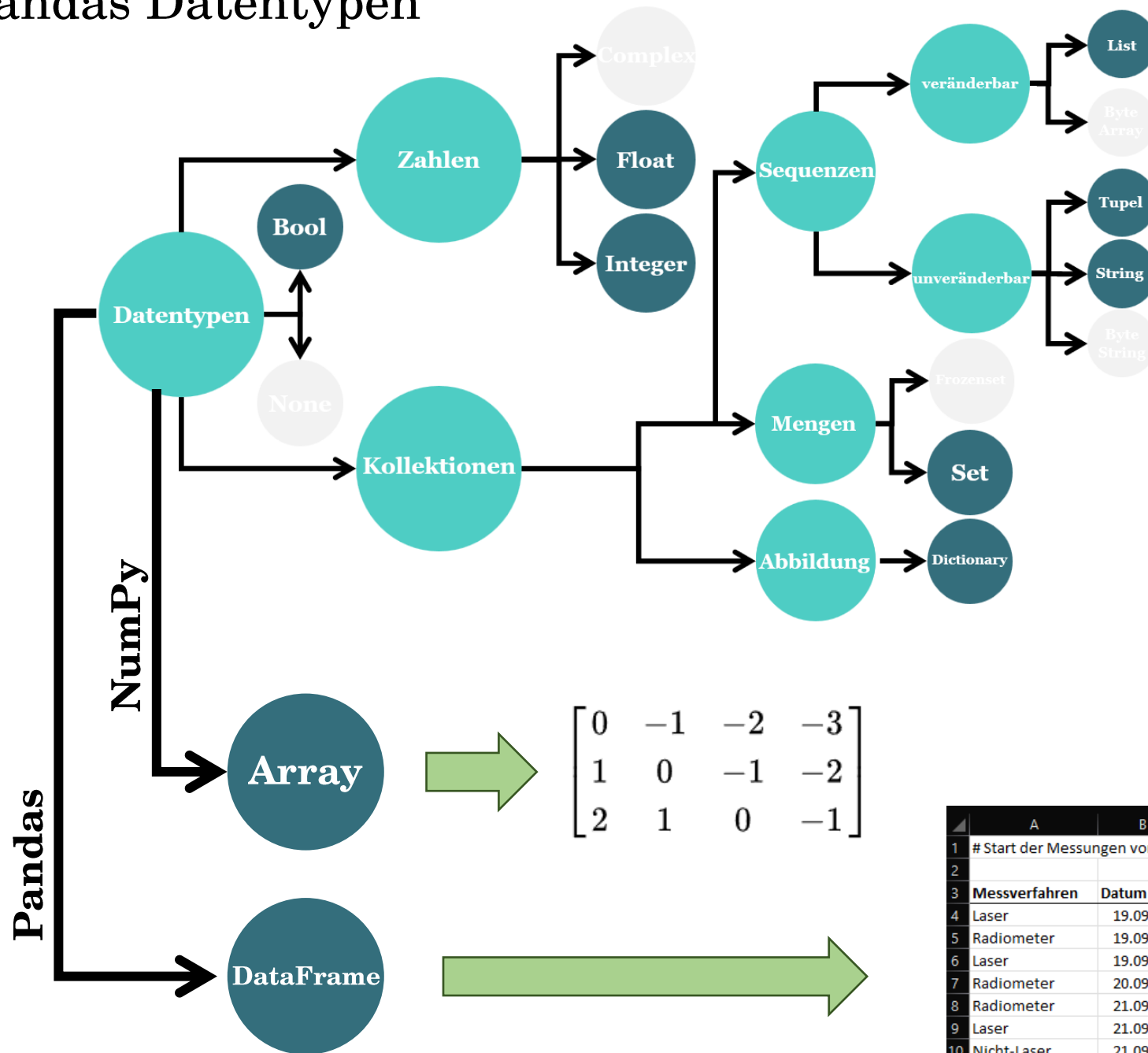
Was ist NumPy und Pandas?

- NumPy und Pandas sind beides Pakete von Modulen die Python um eine Reihe von neuen Datentypen und dazugehörigen Typfunktionen erweitern.
- NumPy eignet sich insbesondere zur effizienten Modellierung, Verarbeitung und Analyse (sehr) großer numerischer Datensätze.
- Pandas bietet darüber hinaus die Möglichkeit Datensätze bestehend aus numerischen und nicht-numerischen Werten einfach, flexibel und schnell zu verarbeiten.

Übersicht Variablentypen



NumPy & Pandas Datentypen



$$\begin{bmatrix} \cos 90^\circ & \sin 90^\circ \\ -\sin 90^\circ & \cos 90^\circ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0 \\ a_1 \end{bmatrix}$$

NumPy

Importieren von Modulen unter neuen Namen

- Mit der `import as` Anweisung importiert man sich den Inhalt eines Moduls unter einen selbstgewählten Namen in den *global Namespace* seines Programms:

```
import random as ran
ran.randint(0,10)
3
```

- Dadurch spart man sich Schreibarbeit, wenn man in seinem Programm viel Gebrauch von einem importierten Modul macht. Die Module NumPy und Pandas werden üblicherweise so importiert:

```
import numpy as np
import pandas as pd
```



Der NumPy Datentyp `array`

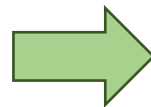
- Ein Objekt vom Datentyp `array` stellt eine homogene multidimensionale Matrix dar, d. h. alle Elemente in dieser Matrix haben den selben Typ (normalerweise Integer oder Float).
- Zur Erzeugung eines 1-dimensionalen `array` Objekts (eines Vektors) gibt es verschiedene Funktionen innerhalb des NumPy Moduls:

```
import numpy as np

x = np.array([1, 2, 3])
x
array([1, 2, 3])

type(x)
<class 'numpy.ndarray'>
```

`np.array([1, 2, 3])`



Der NumPy Datentyp `array`

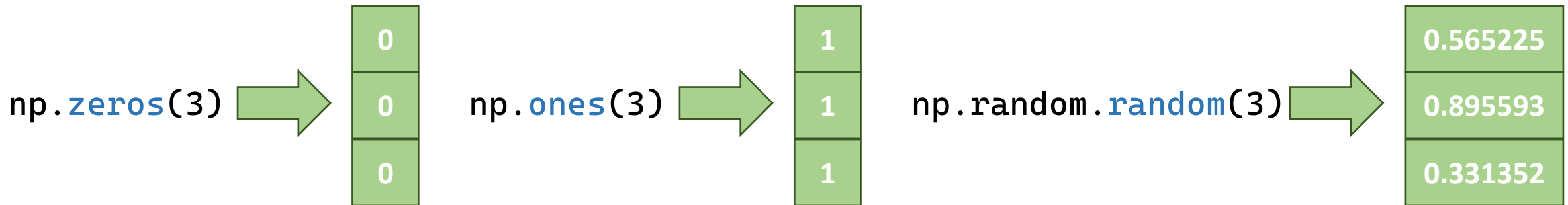
- Zur Erzeugung eines 1-dimensionalen `array` Objekts gibt es verschiedene Funktionen innerhalb des NumPy Moduls:

```
import numpy as np

np.zeros(3)
array([0. , 0. , 0.])

np.ones(3)
array([1. , 1. , 1.])

np.random.random(3)
array([0.56522538, 0.89559304, 0.33135279])
```



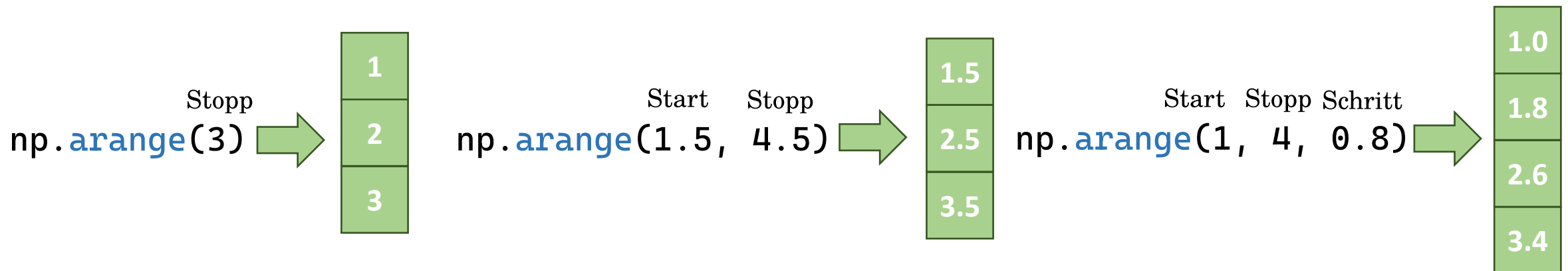
Die NumPy arange Funktion

- Die NumPy arange Funktion arbeitet wie die normale Python range Funktion, nur das sie zusätzlich auch mit Float Argumenten arbeiten kann:

```
import numpy as np

np.arange(0, 10, 1)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

np.arange(0, 10, 0.3)
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. , 3.3, 3.6,
       3.9, 4.2, 4.5, 4.8, 5.1, 5.4, 5.7, 6. , 6.3, 6.6, 6.9, 7.2, 7.5,
       7.8, 8.1, 8.4, 8.7, 9. , 9.3, 9.6, 9.9])
```



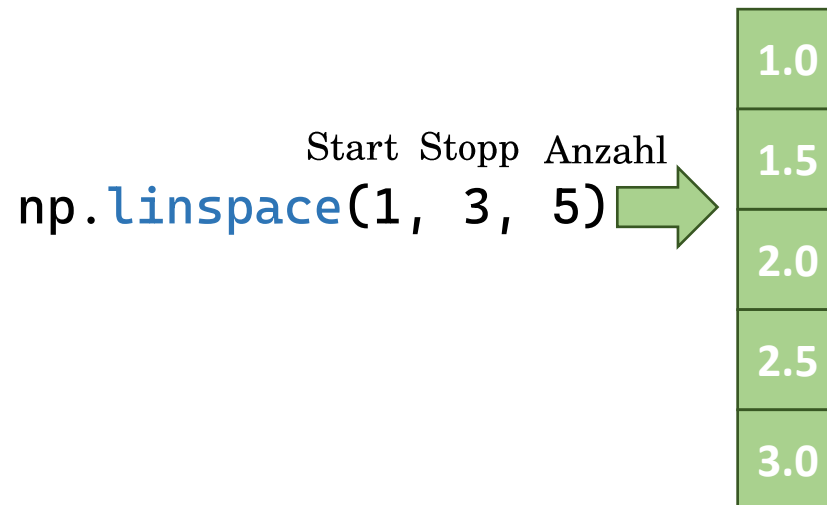
Die NumPy `linspace` Funktion

- Die NumPy `linspace` Funktion erzeugt ein Array bestehend aus Zahlen, welche alle gleichmäßig zwischen einem Start- und einem Stoppwert verteilt sind:

```
import numpy as np

np.linspace(0, 0.5, 6)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5])

np.linspace(5, 15, 11)
array([ 5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14., 15.] )
```



NumPy Arrays versus Python Listen

- Zwei gute Gründe die für den Datentyp `array` sprechen: 1. intuitives Rechnen

```
import numpy as np
```

```
x = np.array([1, 2, 3])
```

```
y = np.array([3, 2, 1])
```

```
z = x + y
```

```
z  
array([4, 4, 4])
```

```
x = [1, 2, 3]
```

```
y = [3, 2, 1]
```

```
x + y
```

```
[1, 2, 3, 3, 2, 1]
```

```
z = []
```

```
for i in range(len(x)):
```

```
    z.append(x[i] + y[i])
```

```
z  
[4, 4, 4]
```

NumPy Arrays versus Python Listen

- Zwei gute Gründe die für den Datentyp `array` sprechen: 2. Rechenzeit

```
import numpy as np
import time

startZeit = time.time()
x, y = range(10000000), range(10000000)
z = []
for i in range(len(x)):
    z.append(x[i] + y[i])
print(time.time() - startZeit)

startZeit = time.time()
x, y = np.arange(10000000), np.arange(10000000)
z = x + y
print(time.time() - startZeit)
```

Rechnen mit NumPy Arrays

- Rechenoperationen zwischen NumPy Arrays entsprechen den Vektor- bzw. Matrixrechenoperationen aus der Algebra:

```
x = np.array([5, 9])  
y = np.array([4, 6])  
x - y  
array([1, 5])
```

$$\begin{array}{|c|} \hline x \\ \hline 5 \\ \hline 9 \\ \hline \end{array} - \begin{array}{|c|} \hline y \\ \hline 4 \\ \hline 6 \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hline 3 \\ \hline \end{array}$$

```
x = np.array([5, 9])  
y = np.array([4, 6])  
x * y  
array([20, 54])
```

$$\begin{array}{|c|} \hline x \\ \hline 5 \\ \hline 9 \\ \hline \end{array} * \begin{array}{|c|} \hline y \\ \hline 4 \\ \hline 6 \\ \hline \end{array} = \begin{array}{|c|} \hline 20 \\ \hline 54 \\ \hline \end{array}$$

```
x = np.array([5, 9])  
y = np.array([4, 6])  
x / y  
array([1.25, 1.5])
```

$$\begin{array}{|c|} \hline x \\ \hline 5 \\ \hline 9 \\ \hline \end{array} / \begin{array}{|c|} \hline y \\ \hline 4 \\ \hline 6 \\ \hline \end{array} = \begin{array}{|c|} \hline 1.25 \\ \hline 1.5 \\ \hline \end{array}$$

```
x = np.array([5, 9])  
x * 0.9  
array([4.5, 8.1])
```

$$\begin{array}{|c|} \hline x \\ \hline 5 \\ \hline 9 \\ \hline \end{array} * 0.9 = \begin{array}{|c|} \hline 4.5 \\ \hline 8.1 \\ \hline \end{array}$$

Mehrdimensionale Arrays

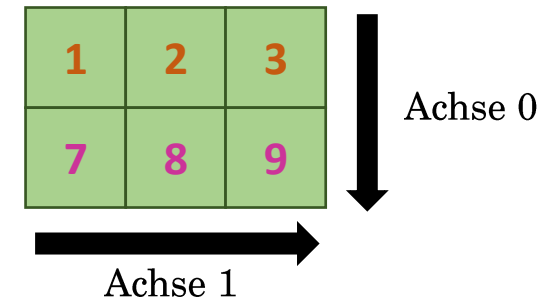
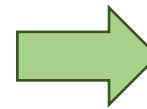
- Neben 1-dimensionalen Arrays (Vektoren), können auch relativ einfach mehrdimensionale Arrays (Matrizen) erzeugt werden. Statt von Dimensionen spricht man auch von Achsen.

```
import numpy as np

x = np.array([[1, 2, 3], [7, 8, 9]])
x
array([[1, 2, 3],
       [7, 8, 9]])

x.ndim
2
x.shape
(2, 3)
```

`np.array([[1, 2, 3], [7, 8, 9]])`



Mehrdimensionale Arrays

- Neben 1-dimensionalen Arrays (Vektoren), können auch relativ einfach mehrdimensionale Arrays (Matrizen) erzeugt werden. Statt von Dimensionen spricht man auch von Achsen.

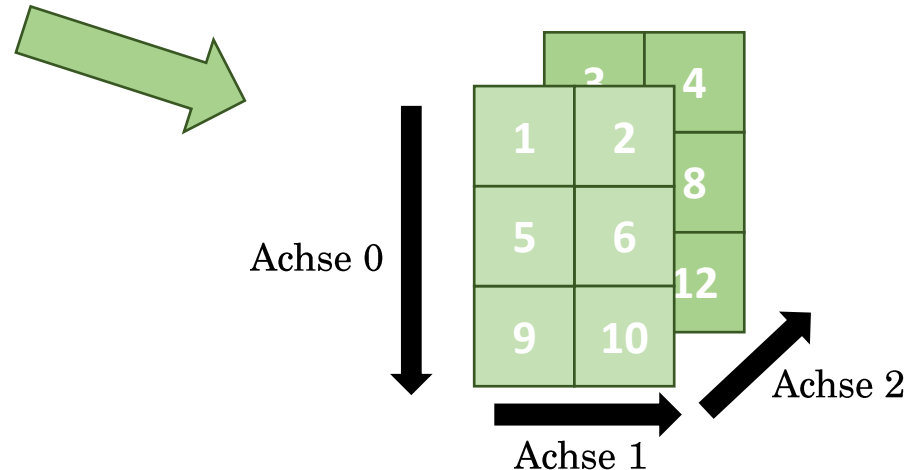
```
import numpy as np

x = np.array([[[ 1,  2],[ 3,  4]],[[ 5,  6],[ 7,  8]],[[ 9, 10],[11, 12]]])

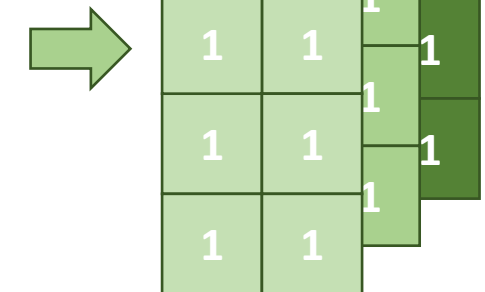
x.ndim
3
x.shape
(3, 2, 2)

np.ones((4,2,3))
```

```
np.array([[[1,2],[3,4]],[[5,6],[7,8]],[[9,10],[11,12]]])
```



```
np.ones((4,2,3))
```



Mehrdimensionale Arrays

- Mit Hilfe der **reshape** Funktion lässt sich aus einem 1-dimensionalen Array sehr leicht ein mehrdimensionales Array umformen (so lange die Anzahl der Elemente stimmt):

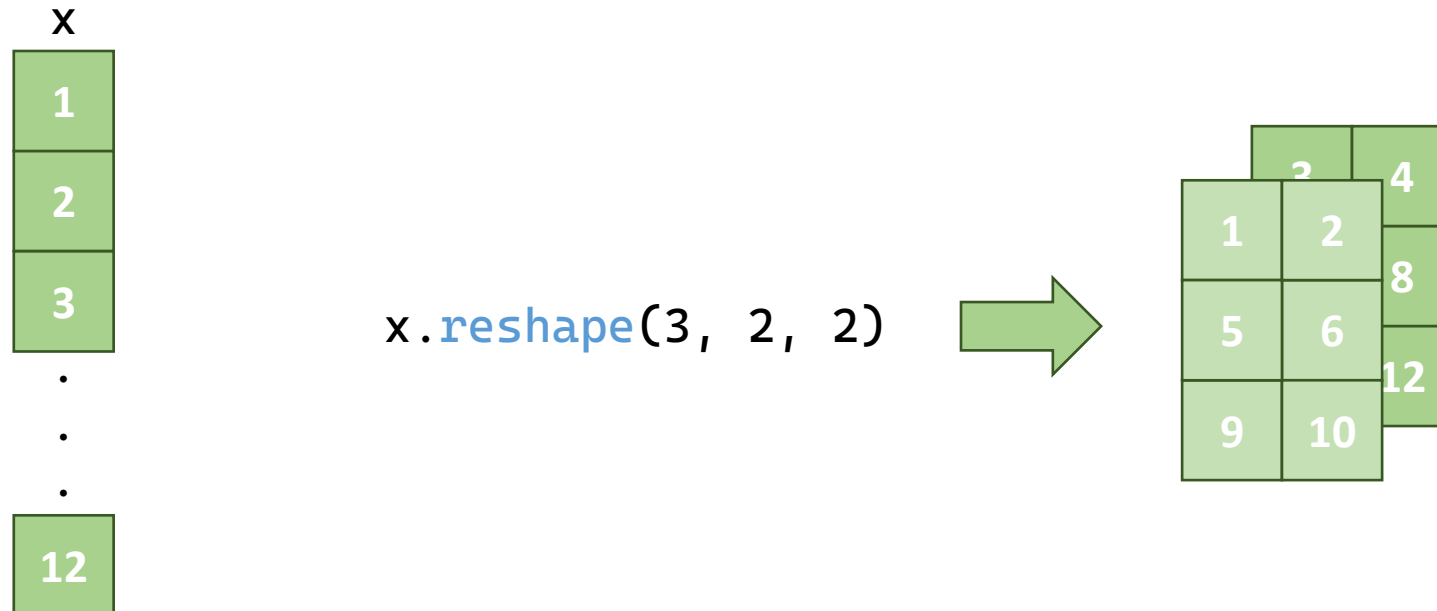
```
import numpy as np
```

```
x = np.arange(1, 13)
```

```
x = x.reshape(3, 2, 2)
```

```
x = x.reshape(3, 3, 3)
```

```
ValueError: cannot reshape array of size 12 into shape (3,3,3)
```



Rechnen mit mehrdimensionalen NumPy Arrays

- Rechenoperationen zwischen mehrdimensionalen NumPy Arrays entsprechen den Vektor- bzw. Matrixrechenoperationen aus der Algebra:

```
import numpy as np  
  
x = np.array([1, 2, 3, 4]).reshape(2, 2)  
y = np.ones((2,2))  
z = np.array([5, 6])
```

$$\begin{array}{|c|c|} \hline x & \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline y & \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 4 & 5 \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline x & \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline z & \\ \hline 5 & 6 \\ \hline \end{array} = \begin{array}{|c|c|} \hline x & \\ \hline 1 & 2 \\ \hline 3 & 4 \\ \hline \end{array} + \begin{array}{|c|c|} \hline z & \\ \hline 5 & 6 \\ \hline 5 & 6 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 6 & 8 \\ \hline 8 & 10 \\ \hline \end{array}$$

NumPy Array Slicing

- NumPy Arrays unterstützen auch die Slicingfunktion, wie wir sie von den Sequenzen (Strings, Tupel, Listen) kennen.
- So kann man gezielt auf einzelne Elemente, einzelne Achsen oder Teilarrays zugreifen.
- Hier ein paar Beispiele mit einem 2-dimensionalen Array (Slicing bei Arrays mit noch mehr Dimensionen führt schnell zu Kopfschmerzen):

```
x = np.arange(1,13).reshape(3, 4)
x[0]
array([1, 2, 3, 4])
```

x =

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

Index Achse 0

Index Achse 1

x[0]

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

Index Achse 0

Index Achse 1

NumPy Array Slicing

- NumPy Arrays unterstützen auch die Slicingfunktion, wie wir sie von den Sequenzen (Strings, Tupel, Listen) kennen.
- So kann man gezielt auf einzelne Elemente, einzelne Achsen oder Teilarrays zugreifen.
- Hier ein paar Beispiele mit einem 2-dimensionalen Array (Slicing bei Arrays mit noch mehr Dimensionen führt schnell zu Kopfschmerzen):

```
x = np.arange(1,13).reshape(3, 4)
x[:,0]
array([1, 5, 9])
```

x =

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

Index Achse 0

Index Achse 1

x[:,0]

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

Index Achse 0

Index Achse 1

NumPy Array Slicing

- NumPy Arrays unterstützen auch die Slicingfunktion, wie wir sie von den Sequenzen (Strings, Tupel, Listen) kennen.
- So kann man gezielt auf einzelne Elemente, einzelne Achsen oder Teilarrays zugreifen.
- Hier ein paar Beispiele mit einem 2-dimensionalen Array (Slicing bei Arrays mit noch mehr Dimensionen führt schnell zu Kopfschmerzen):

```
x = np.arange(1,13).reshape(3, 4)
x[2][3]
12
```

x =

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

Index Achse 0

Index Achse 1

x[2][3]

1	2	3	4	0
5	6	7	8	1
9	10	11	12	2
	0	1	2	3

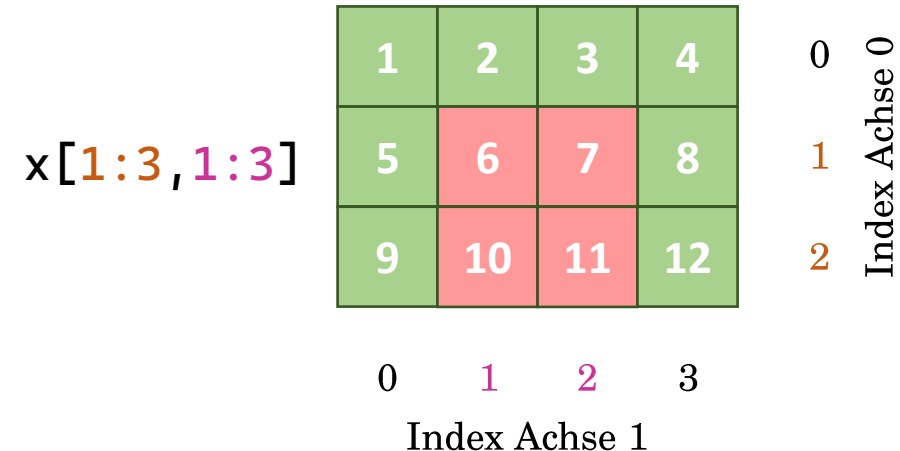
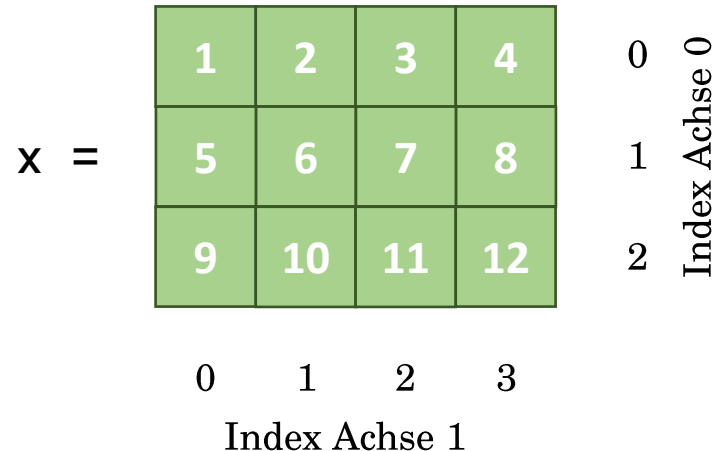
Index Achse 0

Index Achse 1

NumPy Array Slicing

- NumPy Arrays unterstützen auch die Slicingfunktion, wie wir sie von den Sequenzen (Strings, Tupel, Listen) kennen.
- So kann man gezielt auf einzelne Elemente, einzelne Achsen oder Teilarrays zugreifen.
- Hier ein paar Beispiele mit einem 2-dimensionalen Array (Slicing bei Arrays mit noch mehr Dimensionen führt schnell zu Kopfschmerzen):

```
x = np.arange(1,13).reshape(3, 4)
x[1:3,1:3]
array([[ 6,  7],
       [10, 11]])
```



NumPy Array Typfunktionen

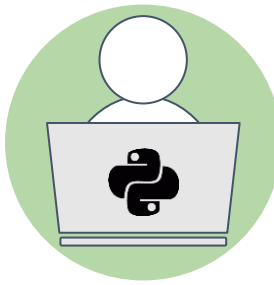
- NumPy Arrays besitzen eine Vielzahl von Typfunktionen, um die gespeicherten numerischen Werte zu verarbeiten:

```
x = np.array([1, 2, 3])  
x.min()  
1  
  
x.max()  
3  
  
x.sum()  
6  
  
x.mean()  
2.0
```


Lernziele für Heute

- Du weißt was Namensräume sind, welche Bedeutung sie für Variablen und Funktionen haben und wie Python mit Namenskonflikten umgeht.
- Du kennst den Unterschied zwischen einem Pythonprogramm und einem Modul und kannst mit Hilfe von *pip* externe Module installieren und verwenden.
- Du bist in der Lage Datei-Objekte zu erzeugen und mit diesen Textdateien zu lesen und zu schreiben.
- Du kannst Grundlegende Typfunktionen von Strings zur Verarbeitung (un)strukturierter Textdaten anwenden.
- Du weißt was NumPy ist und wofür es verwendet wird.

Die `arange` und `linspace` Funktionen



Aufgabe:

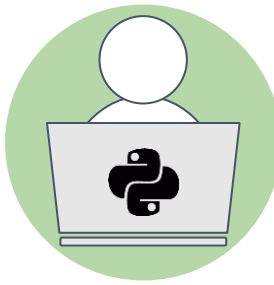
Erzeugt die folgenden 1-dimensionalen NumPy Arrays sowohl mit Hilfe der `arange` Funktion, als auch mit der `linspace` Funktion:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. , 3.3, 3.6, 3.9, 4.2, 4.5])
```

```
array([ 5.   ,  5.625,  6.25 ,  6.875,  7.5   ,  8.125,  8.75 ,  9.375, 10.   ])
```

```
array([ 0.          ,  2.85714286,  5.71428571,  8.57142857, 11.42857143,  
       14.28571429, 17.14285714, 20.          ])
```



Aufgabe:

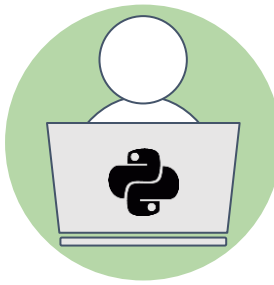
Erzeugt ein 5x4 großes 2-dimensionales NumPy Array bestehend aus zufälligen Integern zwischen 1 und 20.

Die vier besprochenen Typfunktionen `min`, `max`, `sum` und `mean` lassen sich sowohl auf dem gesamten Array, also auch mit Hilfe des *axis* Arguments auf den einzelnen Achsen aufrufen:

```
x = np.arange(1,13).reshape(3, 4)
x.sum()
78
x.sum(axis=0)
array([15, 18, 21, 24])
```

Testet diese vier Typfunktionen auf euren erzeugten Arrays, sowohl mit als auch ohne das *axis* Argument. Welchen Einfluss hat das *axis* Argument auf die Berechnung bzw. die Ausgabe?

In wie viele unterschiedliche 1-, 2- und 3-dimensionale Formen lässt sich euer erzeugtes Array mit Hilfe der `reshape` Funktion umwandeln?



Aufgabe:

Erstellt einen leeren neuen Ordner namens "beispieldaten".

Ladet euch die Datei "51_Daten.py" aus dem Übungsmaterialordner herunter und speichert sie in dem gerade erstellten leeren Ordner. Führt die heruntergeladene Datei über die Kommandozeile mit Hilfe des Pythoninterpreters in dem leeren Ordner aus. Öffnet anschließend die Datei und schaut euch den Programmcode genauer an.

Was tut das Programm und wie?