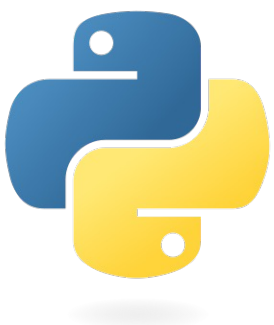
HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Programmierung mit



python™

Wiederholung – Was haben wir bereits alles gelernt?

- Programmierung (Probleme, Algorithmen, Programme, Programmiersprachen)
- Versionsverwaltung (git)
- Variablen (Integer, Float, String, Boolean, None)
- Datenstrukturen (Listen, Tupeln, Dictionaries)
- Schleifen (for, while)
- Bedingungen (if/else, switch/case bzw switch/match)
- Funktionen
- Klassen & Objekte
- Vererbung
- Fehlerbehandlung (Exceptions, try/catch)
- private/public

Rekursion

Python erlaubt die Rekursion von Funktionen

- Funktionen können sich also selbst aufrufen
- Achtung: Infinite-Loops möglich
- Rekursive-Funktionen brauchen **immer** eine Abbruchbedingung

```
def countdown(n):
    print(n)

    if n == 0: # Abbruchbedingung
        return
    else:
        countdown(n - 1) # Funktion ruft sich selbst auf

countdown(10)
```

Rekursion

Beispiel: Die folgende Funktion berechnet die Fakultät von n

4! = 1 * 2 * 3 * 4

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)


print(factorial(4))
```

Call Stack

Aufgaben 1

1. Schreibt eine Rekursive Funktion reverseString(string), welche einen String umdreht.

Beispiel: Aus “Python“ wird „nohtyP“

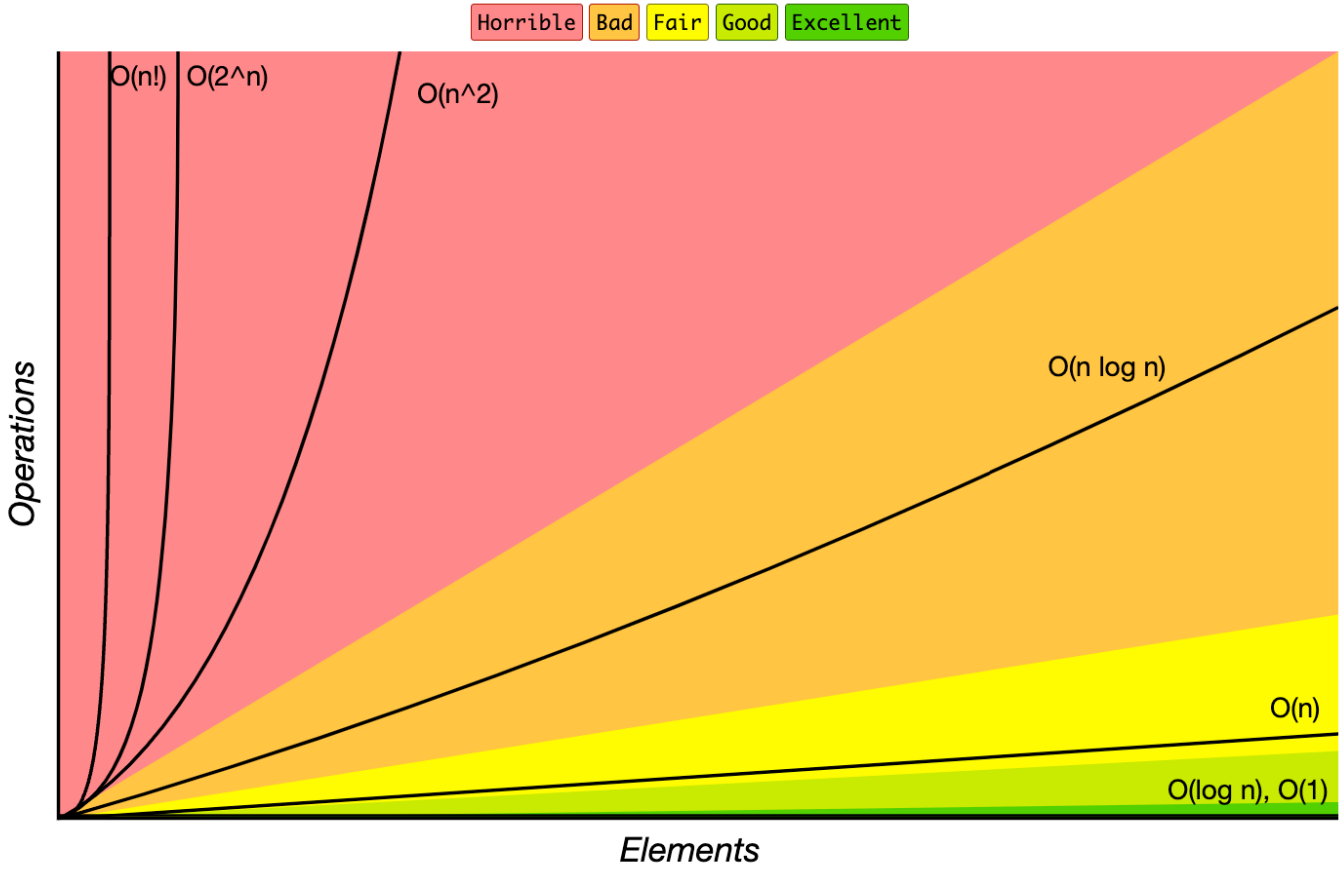
Erinnerung: eine Rekursive-Funktion ruft sich selbst auf und teilt ein Problem in kleinere Probleme.

Aufgaben 1

1. Schreibt eine rekursive Funktion recursiveSum(int_list), welche rekursiv die Summe einer Liste berechnet.

Laufzeitanalyse - Big O Notation

Die Big O Notation beschreibt, wie gut ein Algorithmus bei einer wachsenden Eingabegröße *n* performt.



Anders gesagt: Wie lange läuft mein Algorithmus überhaupt bis er seine Aufgabe erfüllt?

Dabei gibt es gute und inakzeptable Laufzeiten.

Laufzeitanalyse - Big O Notation

Konstante Komplexität *O(1)*
Der Zugriff auf einen Index in einer Liste dauert immer gleichlang, egal wie groß das Liste ist

```
list = [...]

list[53]
```

Lineare Komplexität *O(n)*
Die Laufzeit wächst Linear zur Eingabegröße *n*

```
list = [...]

for x in list:
    print(x)
```

Laufzeitanalyse - Big O Notation

Quadratische Komplexität *O(n^2)*
Die Laufzeit wächst quadratisch zur Eingabegröße *n*

```
list = [...]

for a in list:
    for b in list:
        print(a * b)
```

Logarithmische Komplexität *O(log n)*
Die Laufzeit wächst logarithmisch zur Eingabegröße *n*

- Wie das erreicht werden kann, kommt in den folgenden Folien

Suchalgorithmen - Lineare Suche

Praxisbeispiel: Wie finde ich eine Zahl in einer Liste?

```
def linear_search(list, x):
    for i in list:
        if i == x:
            return i

    return -1

linear_search([1,2,3,4,5,6,7], 3)
```

Welche Komplexität hat dieser Algorithmus?

Antwort: O(n)
Unser Beispiel: O(7)

Zeitmessung – time-Modul

Mit Hilfe des time-Moduls können wir Zeiten in Python bestimmen und messen.

```
import time

begin = time.time() # vergangene Sekunden seit dem 01.01.1970
# do something
# do something else
end = time.time()
print("Dauer der Programmausführung:", end-begin)
```

random-Modul

Mit Hilfe des random-Moduls können wir zufällige Zahlen generieren

```
import random

# Generiert 5 zufällige Zahlen zwischen 10 und 30
randomlist = random.sample(range(10, 30), 5)
print(randomlist)
```

Aufgaben 2

1. Nutzt die Funktion linear_search von Folie 9 und messt wie sich das Laufzeitverhalten bei verschiedenen Listen verhält. Nutzt dafür das time-Modul um die Zeiten zu messen und das random-Modul um die Listen zu generieren.

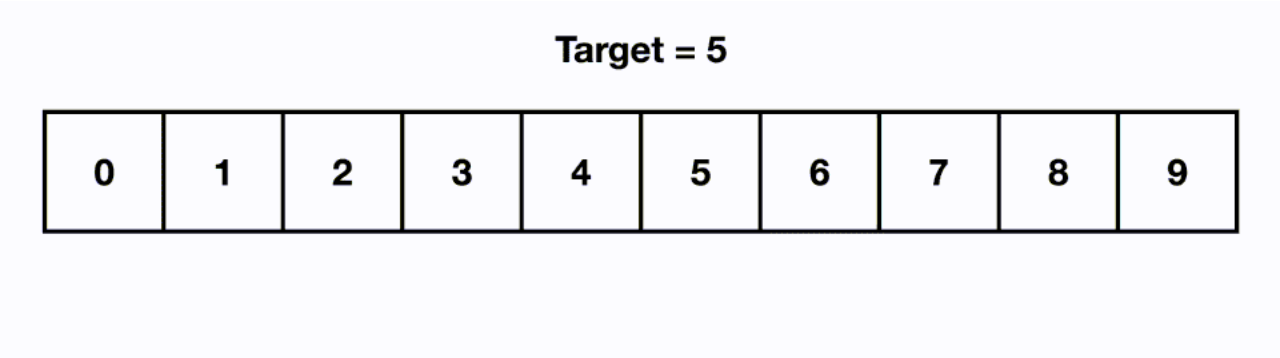
Suchalgorithmen - Binäre Suche

Bei der Binären Suche wird der Teil einer Liste, der das Element enthalten könnte, wiederholt in zwei Hälften geteilt, bis das Zielelement eingegrenzt ist. Annahme: Die Liste ist bereits sortiert.

Algorithmus

1. Mitte der Liste finden
2. Vergleich des Mittelpunkts mit dem Zielwert
3. Wenn Wert und Ziel gleich -> Ziel gefunden
4. Wenn Wert kleiner als das Ziel, erstellen wir eine neue Liste, die vom Mittelpunkt +1 bis zum größten Wert geht
5. Ist der Wert größer als das Ziel, erstellen wir eine neue Liste, die vom kleinsten Wert bis zum Mittelpunkt -1 geht

Dies wird so lange wiederholt, bis das Ziel gefunden oder das letzte Element erreicht ist, das nicht mit dem Ziel übereinstimmt.



Aufgaben

Versucht die Binäre Suche selbst nur anhand der Beschreibung vom Algorithmus der vorherigen Folie zu implementieren.
Nutzt nur die Lösung wenn ihr nicht mehr weiter wisst.

Generiert anschließend große sortierte Listen und testet die Laufzeit der Binären Suche bei unterschiedlich großen Listen.

Suchalgorithmen - Binäre Suche

Bei der Binären Suche wird der Teil einer Liste, der das Element enthalten könnte, wiederholt in zwei Hälften geteilt, bis das Zielelement eingegrenzt ist. Annahme: Die Liste ist bereits sortiert.

```python
def binary_search(my_list, target):
    left = 0
    right = len(my_list) - 1

    while left <= right:
        mid_point = (left + right) // 2

        if target < my_list[mid_point]:
            right = mid_point - 1
        elif target > my_list[mid_point]:
            left = mid_point + 1
        else:
            return mid_point

    return -1
```

Laufzeit O(log n)

Exkurs: Pseudocode

Vereinfachter Programmcode, um den Algorithmus besser (vorab beschreiben zu können).
Meist eine Mischung aus sogenannten höheren Programmiersprachen (wie zB Python oder Java), natürlicher Sprache und mathematischer Notation.

So kompakt, dass er leichter verständlich ist als realer Programmcode und gleichzeitig klarer und unmissverständlicher als natürliche Sprache.

Beispiel: lineare Suche

```
linear_search(list, target)
    durchlaufe die gesamte Liste list
        falls Wert des ElementDerListe gleich target
            then return Index des ElementDerListe

    return Fehlercode
```

wird übersetzt zu:

```python
def linear_search(list, target):
    for i in range(len(list)):
        if list[i] == target:
            return i

    return -1
```

Hinweis: Es gibt keinen Pseudocode-Standard. Faustregel: Alles, was hilft ist erlaubt.

Sortieralgorithmen – Bubble-Sort

Bubble-Sort ist ein einfacher Sortieralgorithmus, der durch wiederholtes Vertauschen der benachbarten Elemente funktioniert, wenn sie in falscher Reihenfolge sind.

```python
def bubbleSort(array):
    n = len(array)

    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
```

6 5 3 1 8 7 2 4

Was ist die Laufzeit von diesem Algorithmus?

$O(n^2)$ -> Bubblesort ist der schlechteste Sortieralgorithmus

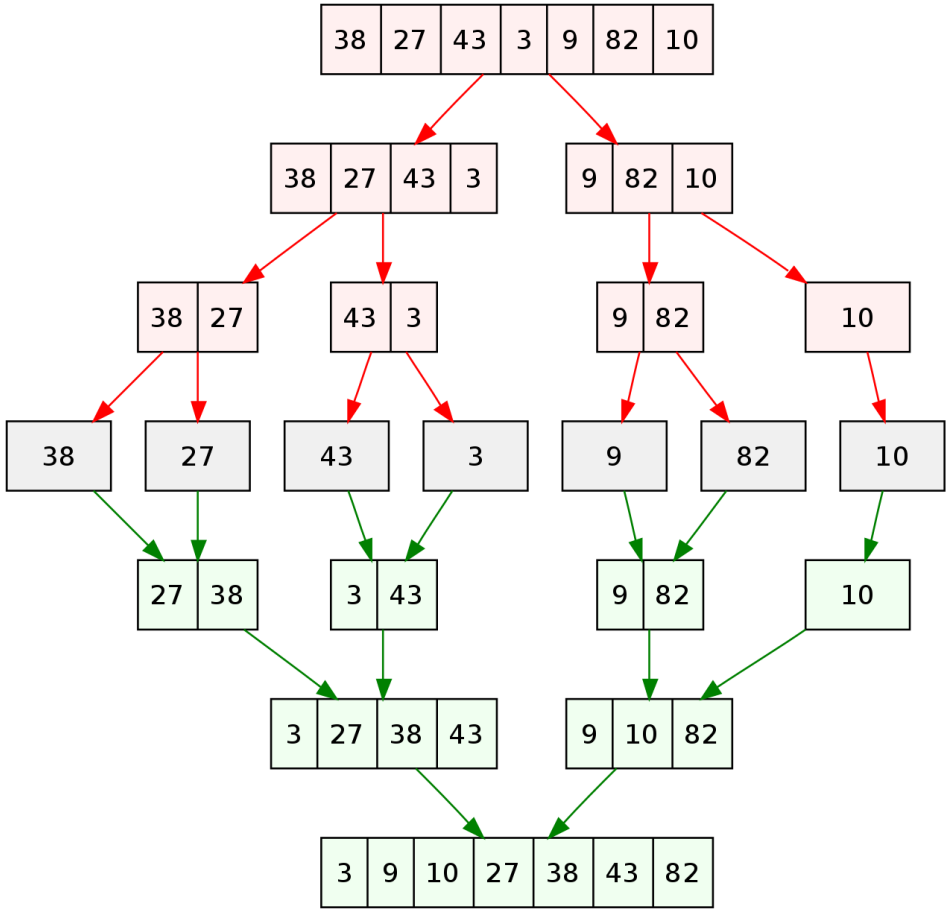
Gif-Quelle: https://de.wikipedia.org/wiki/Datei:Bubble-sort-example-300px.gif

Mergesort - O(n log n)

- Laufzeit: hat im best, average und worst case immer die gleiche Laufzeit O(n log n)

Der Algorithmus besteht aus zwei Schritten:

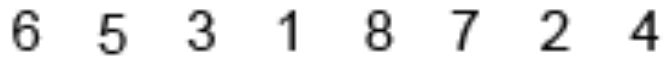
1. Die Ausgangsliste wird so lange in zwei kleinere Listen geteilt, bis nur noch ein Element in jeder Liste vorhanden ist.
2. Elemente werden wieder zu kleinen Listen zusammengefügt und in sich sortiert. Dies wird so lange fortgesetzt, bis am Ende die komplette Liste sortiert und wieder zusammen gefügt wurde.



Quelle des Bildes: https://de.wikipedia.org/wiki/Merge-Algorithmen

Mergesort - O(n log n) – Teil 1

```
funktion merge_sort(liste):
    if (Länge von liste < 2)
        dann return liste
    else
        halbiere die liste in linkeListe, rechteListe
        mitte = len(liste) //2
        merge_sort(linkeListe)
        merge_sort(rechteListe)
        return merge(liste, linkeListe, rechteListe)
```



Gif-Quelle: https://upload.wikimedia.org/wikipedia/commons/c/cc/Merge-sort-example-300px.gif

Mergesort - O(n log n) – Teil 2

```
funktion merge(liste, linkeListe, rechteListe):
    Index aller Listen auf 0 setzen
    while (linkeListe und rechteListe nicht leer)
    |   if (aktuellesElement der linkeListe <= aktuellesElement der rechteListe)
    |       then füge das aktuelleElement der linkeListe in die liste ein
    |       Index linkeListe + 1
    |   else füge aktuellesElement rechteListe in die liste ein
    |       Index rechteListe +1
    |   Index liste + 1
    while (linkeListe nicht leer)  # indexLinks < len(linkeListe)
    |   then füge aktuellesElement linkeListe in die liste ein
    |   Index linkeListe + 1
    |   Index liste + 1
    while (rechteListe nicht leer)
    |   then füge aktuellesElement rechteListe in die liste ein
    |   Index rechteListe +1
    |   Index liste +1
```

Aufgabe

Nehmt euch einen Zettel und führt händisch den Merge-Sort Algorithmus mit der folgenden Liste durch:

6	5	12	10	9	1

Aufgabe

Ich habe eine Implementierung von Merge Sort auf Github hochgeladen.
Schreibt nun ein Programm, welches wieder eine große Liste mit zufälligen Elementen generiert und die Laufzeiten von Bubble Sort und Merge Sort vergleicht.

Testet das Verhalten vom Merge Sort, indem ihr an verschiedenen Stellen *print* nutzt um die einzelnen Schritte vom Algorithmus nachzuvollziehen.

Sortieralgorithmen – Divide and Conquer

Divide: Teile das Problem in (mindestens) 2 annähernd gleich große Teilprobleme

Conquer: Löse Teilprobleme auf dieselbe Art (rekursiv) bis sie trivial zu lösen sind

Merge: Füge die Teillösungen zur Gesamtlösung zusammen

Sortieralgorithmen – Quick-Sort

Quick-Sort ist ein schneller, rekursiver Sortieralgorithmus der nach dem Divide-and-Conquer Prinzip arbeitet.

Prinzip:

1. Liste in zwei Teillisten trennen. Dazu ein Pivotelement auswählen.
2. Alle Elemente < Pivotelement in linke Teilliste und alle Element > Pivotelement in die rechte Teilliste
3. Elemente die gleich Pivotelement sind beliebig platzieren
4. Schritt 1, 2, 3 auf Teillisten rekursiv anwenden
5. Wenn eine Teilliste die Länge 1 oder 0 hat, ist sie sortiert und die Rekursion wird abgebrochen.

Qif-Quelle: https://upload.wikimedia.org/wikipedia/commons/9/9c/Quicksort-example.gif

Sortieralgorithmen – Quicksort

Initialer Aufruf vom Quicksort

```
def quick_sort(start, end, array):
    if (start < end):
        p = partition(start, end, array)

        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)
```

Sortieralgorithmen – Quicksort

Funktion zum Teilen der Liste

```
def partition(start, end, array):
    pivot_index = start
    pivot = array[pivot_index]

    while start < end:
        while start < len(array) and array[start] <= pivot:
            start += 1

        while array[end] > pivot:
            end -= 1

        if (start < end):
            array[start], array[end] = array[end], array[start]

    array[end], array[pivot_index] = array[pivot_index], array[end]

    return end
```

Aufgabe

Nehmt euch einen Zettel und führt händisch den Quick-Sort Algorithmus mit der folgenden Liste durch:

14	12	16	13	11	15

Module

Anstatt bei größeren Programmen, jeglichen Code in eine Datei zu schreiben, sollten der Code in verschiedene Dateien aufgeteilt werden. Die Dateien können dann in anderen Dateien importiert werden.

person.py

```python
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

main.py

```python
from person import Person


person = Person("Peter")
print(person)
```

Module - main

Wenn Python ein Modul als Hauptprogramm ausführt, setzt Python die spezielle Variable __name__ auf den Wert "__main__". So kann unterschieden werden, welcher Code ausgeführt wird wenn die Datei importiert wird, oder wenn sie direkt ausgeführt wird.

```
print("Wird immer ausgeführt")

if __name__ == "__main__":
    print("Wird ausgeführt wenn die Datei direkt ausgeführt wird")
else:
    print("Wird ausgeführt wenn die Datei importiert wird")
```

Module

Anstatt bei größeren Programmen, jeglichen Code in eine Datei zu schreiben, sollten der Code in verschiedene Dateien aufgeteilt werden. Die Dateien können dann in anderen Dateien importiert werden.

person.py

```python
class Person:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name
```

main.py

```python
from person import Person


if __name__ == "__main__":
    person = Person("Peter")
    print(person)
```

Aufgabe

Schreibt ein Programm mit mehreren Modulen: ein modul für bubbleSort, ein Modul für mergeSort und ein Modul für QuickSort. Importiert die 3-Module in einer main.py-Datei.

In der main.py-Datei soll der User über *input* Entscheiden mit welchem Algorithmus er eine Liste sortieren möchte.

NumPy – Numerical Python

Was ist NumPy?
NumPy ist eine Bibliothek, welche die Arbeit mit Arrays vereinfacht und Funktionen für die Anwendung verschiedener Mathematischer Bereiche bereitstellt (Lineare Algebra, Fourier-Transformation usw.)

Welchen Vorteil hat NumPy?
NumPy stellt besonders effiziente Arrays zur Verfügung. Arrays haben wir bereits als Listen in Python kennengelernt. NumPy Arrays können aber deutlich schneller sein in der Verarbeitung als Listen in Python. NumPy nutzt dazu besonders effiziente Speicherkonzepte und arbeitet möglichst maschinennah.

NumPy ist in der **Data Science** weit verbreitet, da dort schnelle Laufzeiten und geringer Ressoucenverbrauch extrem wichtig ist.

Data Science ist der Teil der Informatik, in dem Daten gespeichert, genutzt und analysiert werden, um aus ihnen Informationen zu gewinnen.

NumPy – Installation

Eingabe im Terminal:

Mac: >>> pip3 install numpy
Windows: >>> python3 -m pip install numpy

Erstes NumPy-Programm:

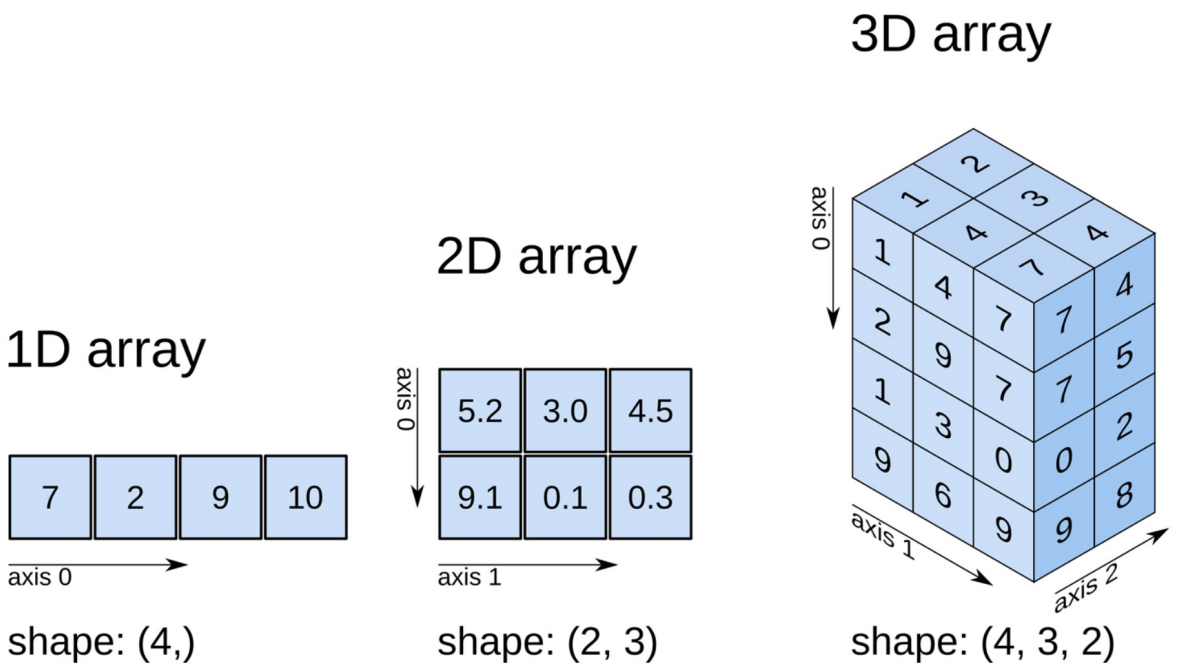
```
import numpy as np # as np Alias ist eine weit verbreitete Konvention

arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

NumPy – Dimensionen von Arrays

Neben numpy.array existiert auch numpy.ndarray um n-dimensionale Arrays abzubilden.

Die Dimension vom Array definiert sich darüber, wie "tief" das nesting ist, also wie viele Arrays ineinander verschachtelt sind.



Bildquelle: https://softwarerecs.stackexchange.com/questions/62286/visualizing-n-dimensional-arrays

NumPy – Dimensionen von Arrays

Neben numpy.array existiert auch numpy.ndarray um n-dimensionale Arrays abzubilden.

Die Dimension vom Array definiert sich darüber, wie "tief" das nesting ist, also wie viele Arrays ineinander verschachtelt sind.

Außerdem wird ein Datentyp definiert.

```
# 1D-Array
ndarray = np.array([1, 2, 3], np.int32)

# 2D-Array
ndarray = np.array([[1, 2, 3], [4, 5, 6]], np.int32)

# 3D-Array
ndarray = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]], np.int32)
```

NumPy – ndarray

Arrays erstellen

Alle ndarrays sind homogen: d.h. jedes Element nimmt einen gleich großen Speicherblock ein und wird immer gleich interpretiert. Wie ein Element interpretiert werden soll, wird über ein Datentyp-Objekt festgelegt. Im Beispiel hier int32, also ein 32-Bit Integer.

```
ndarray = np.array([[1, 2, 3], [4, 5, 6]], np.int32) # 2D-Array
```

Elemente indizieren

Der Zugriff auf Elemente aus dem Array ist ähnlich zu normalen Python Listen

```
array[2] # 1D-Array [1, 2, 3] # Ergebnis 3

array[1, 2] # 2D-Array [[1, 2, 3], [4, 5, 6]] # Ergebnis 6

array[1, 1, 2] # 3D-Array [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]] # Ergebnis 12
```

Aufgabe

Installiert numpy mit >>> ***python3 -m pip install numpy***

Schreibt anschließend ein Python Programm und erstellt ein 1D, ein 2D und ein 3D-Numpy-Array.

Recherchiert wie ihr diese Numpy-Arrays „flatten“ könnt. Also aus einem 3D- oder 2D-Array wird ein 1D-Array.

```
ndarray = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
```

wird zu

```
ndarray = np.array([1, 2, 3, 4, 5, 6], np.int32)
```

NumPy – Hilfsfunktionen

Auf einem NumPy-Array können verschiedene Funktionen aufgerufen werden.

Auflistung aller verfügbaren Funktionen: https://numpy.org/doc/stable/reference/arrays.html

Beispiele:

```
grades = np.array([72, 35, 64, 88, 51, 90, 74, 12])

grades.mean() # Berechne Durchschnitt

grades.max() # Gebe das größte Element zurück

grades.min() # Gebe das niedrigste Element zurück

grades.sort() # Sortiere das Array
```

NumPy – Manipulation von Arrays

NumPy Arrays können ähnlich wie Python Listen mit dem Aufruf verschiedener Funktionen verändert werden.
https://numpy.org/doc/stable/reference/routines.array-manipulation.html

concatenate
Das Verbinden eines Arrays entlang einer bestehenden Achse.

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0) # [[1, 2], [3, 4], [5, 6]]
```

append
Anhängen von Werten an das Ende eines Arrays.

```
np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]]) # [1 2 3 4 5 6 7 8 9]
```

NumPy – Manipulation von Arrays

NumPy Arrays können ähnlich wie Python Listen mit dem Aufruf verschiedener Funktionen verändert werden.
https://numpy.org/doc/stable/reference/routines.array-manipulation.html

insert
Einfügen von Werten entlang der angegebenen Achse vor den angegebenen Indizes.

```
np.insert(arr, obj, values, axis=None)
```

arr: Das Array in welches Daten eingefügt werden sollen
obj: Objekt welches definiert wo die Werte eingefügt werden. Kann einzelne Zahl oder Liste von Zahlen sein
values: Werte welche eingefügt werden sollen
axis (optional): Achse auf welcher Werte eingefügt werden sollen

```
a = np.array([[1, 1], [2, 2], [3, 3]])
np.insert(a, 1, 5) # [1 5 1 2 2 3 3]
```

Aufgabe: Nutzt die Dokumentation um herauszufinden wie Elemente gelöscht werden können. Probiere danach auch noch concatenate, append und insert aus.

NumPy – ndarray Attribute

Oft ist es nötig z.B. die Dimensionen oder die Größe der Arrays abzufragen.
Dafür haben ndarrays verschiedene Attribute die auf dem Array-Objekt abgerufen werden können.
Alle Attribute: https://numpy.org/doc/stable/reference/arrays.ndarray.html#array-attributes

Beispiele:

```
ndarray.shape # Tupel der Array-Dimensionen.

ndarray.ndim # Anzahl der Array-Dimensionen

ndarray.size # Anzahl Elemente im Array

ndarray.itemsize # Speichergröße eines einzelnen Elements in bytes

ndarray.nbytes # Gesamte Speichergröße des Arrays in bytes
```

NumPy – Mathematische Funktionen

Die einfache Implementierung mathematischer Formeln, die mit Arrays arbeiten, ist einer der Gründe, warum NumPy so weit verbreitet ist.

Beispiel:
Elementweise Sinus berechnen

```
np.sin(np.array((0, 30, 45, 60, 90)))
```

Darüber hinaus können noch viele weitere Mathematische Funktionen aufgerufen werden und so recht einfach effizient komplexe Formeln umgesetzt werden.

Siehe: https://numpy.org/doc/stable/reference/routines.math.html

NumPy – Konstanten

NumPy bietet auch viele Konstanten, die für verschiedene Berechnungen nötig sind.

Auflistung aller verfügbaren Konstanten: https://numpy.org/doc/stable/reference/constants.html

Beispiele:

```
numpy.inf # Positive Unendlichkeit (auch z.B. numpy.Inf, numpy.Infinity möglich)

numpy.NINF # Negative Unendlichkeit

numpy.NaN # Not a Number

numpy.e # eulers Konstante: 2.71828182845...

numpy.pi # 3.1415926…
```

pandas – Python Data Analysis

Was ist pandas?

pandas ist eine Bibliothek zur Verarbeitung und Analyse von Daten in Python und ist auf die Verarbeitung von Tabellen und Zeitreihen optimiert.

Installation

Mac: >>> pip3 install pandas
Windows: >>> python -m pip install pandas

```
import pandas as pd # as pd ist verbreitete Konvention
```

Pandas – Datenstrukturen

Pandas implementiert in erster Linie 3 Datenstrukturen.
Wir betrachten insbesondere 2 Datenstrukturen:

- **Series**
- **DataFrame**

Darüber hinaus gibt es noch die Datenstruktur **Panel**. Diese betrachten wir aber nicht näher.

Am besten kann man sich diese Datenstrukturen so vorstellen, dass die höherdimensionale Datenstruktur ein Container für ihre niedrigerdimensionale Datenstruktur ist. So ist **DataFrame** ein Container der **Series** Datenstruktur und **Panel** ein Container von **DataFrame**.

Pandas – Series

Series ist eine eindimensionale arrayartige Datenstruktur mit homogenen Daten. Man kann sie sich vorstellen, wie eine Spalte in einer Tabelle. Die Daten sind homogen, weil jedes Element die gleiche Datenstruktur, wie bspw int32 hat.

```
pd.Series(np.array[1, 2, 3, 4, 5])
```

Der Konstruktor von Series nimmt 4 verschiedene Parameter an.

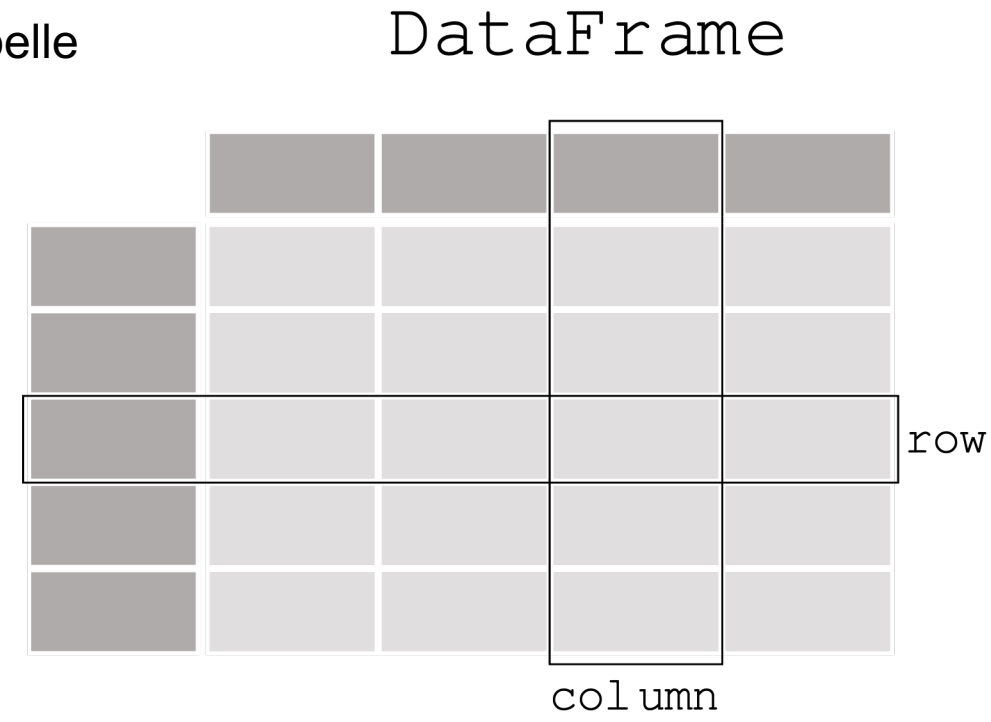
```
pd.Series(data, index, dtype, copy)
```

- **data**: Kann ein NumPy Array, Python Liste oder eine Konstante sein
- index: Eine Liste mit eindeutigen Werten und der gleiche Länge wie data (optional)
- dtype: Der verwendete Datentyp (z.B. np.float32) (optional)
- copy: Boolscher Wert ob die Daten kopiert werden sollen (optional)

Pandas – DataFrame

DataFrame ist ein zweidimensionales Tabelle mit heterogenen Daten.
Die Daten sind heterogen weil jede Spalte eine andere Datenstruktur haben kann.
Jede Spalte ist eine **Series**.

Ein DataFrame ist vergleichbar mit einer Excel-Tabelle



Pandas – DataFrame

Eigenschaften von DataFrame:

- Die Spalten (**Series**) können verschiedene Datentypen haben
- Die Größe ist mutable (Veränderbar)
- Die Achsen (Zeilen und Spalten) können beschriftet werden
- Auf Spalten und Zeilen können arithmetische Funktionen ausgeführt werden

```
df = pd.DataFrame(
    {
        "Name": [
            "Braund, Mr. Owen Harris",
            "Allen, Mr. William Henry",
            "Bonnell, Miss. Elizabeth",
        ],
        "Age": [22, 35, 58],
        "Sex": ["male", "male", "female"],
    }
)
```

Pandas – DataFrame

Der Konstruktor von DataFrame hat 5 Parameter

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

- **data**: Kann eine NumPy, Python Liste, Series, DataFrame oder eine Konstante sein
- index: Eine Liste mit eindeutigen Werten für die Zeilenbeschriftung (optional)
- columns: Eine Liste mit eindeutigen Werten für die Spaltenbeschriftung (optional)
- dtype: Der verwendete Datentyp (z.B. np.float32) (optional)
- copy: Boolscher Wert ob die Daten kopiert werden sollen (optional)

Aufgabe

Erstellt selbst einen DataFrame wie unten mit zufälligen **numerischen** Daten.
Tipp: Ihr braucht für diese Aufgabe sowohl **numpy** als auch **pandas**

Zufällige Werte

Zufällige Werte

Name	Age	Sex	Height
Kelvin	24	male	183
Julia	28	female	175
…	…	…	…

Pandas – CSV-Dateien

Eine einfache Möglichkeit, große Datenmengen zu speichern, ist die Verwendung von CSV-Dateien *(Comma Separated Files).*

CSV-Dateien enthalten reinen Text und sind ein bekanntes Format, das von jedem gelesen werden kann, auch von Pandas.

```
import pandas as pd

data_frame = pd.read_csv('countries.csv')

print(data_frame.info())
```

Pandas – Statistische Auswertung

Es stehen verschiedene Statistische-Methoden zur Verfügung, die auf Spalten mit numerischen Daten angewendet werden können.

Operationen schließen im Allgemeinen fehlende Daten aus und werden standardmäßig zeilenübergreifend durchgeführt.

Beispiele:

```
data_frame = pd.read_csv('titanic.csv')

print(data_frame["Age"].mean()) # Berechnet Durchschnittsalter

print(data_frame.describe()) # Berechnet verschiedenste Methoden wie count, mean,
std, min usw.
```

Weitere Informationen: https://pandas.pydata.org/docs/getting_started/intro_tutorials/06_calculate_statistics.html

Aufgabe

Ich habe auf Github die Datei “countries.csv“ hochgeladen.

Lest diese Datei mit Pandas ein und lasst euch den Durchschnittlichen GDP: Gross domestic product (million current US$) der Länder ausgeben.

Aufgabe

Schreibt eine rekursive Funktion, welche das größte Element in einer Liste aus Integern zurückgibt.

Aufgabe

Schreibt eine rekursive Funktion, welche das Produkt einer Liste aus Integern zurückgibt. Die Werte werden also miteinander multipliziert.

Aufgabe

Schreibt eine rekursive Funktion, welche 1 bis n Zahlen auf der Konsole ausgibt.

Aufgabe

Nehmt euch einen Zettel und führt händisch den Merge-Sort Algorithmus mit der folgenden Liste durch:

5	8	4	10	9	1	3	2

Aufgabe

Nehmt euch einen Zettel und führt händisch den Quick-Sort Algorithmus mit der folgenden Liste durch:

9	-3	5	2	6	8	-6	1	3