Werkzeuge für das wissenschaftliche Arbeiten

Python for Machine Learning and Data Science

Magnus Bender
bender@ifis.uni-luebeck.de
Wintersemester 2023/24

Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) Grundlagen
   c) Fortgeschritten
2. Auszeichnungssprachen
   a) LaTeX, Markdown
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) Jupyter Notebooks lokal und in der Cloud (Google Colab)
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers

Themen

I. Projektaufgabe 1

- Herangehensweise & Tipps
- Fragen

II. Grundlagen

- Pakte und Importe
- Virtuelle Umgebungen
- Objektorientierung

Projektaufgabe 1

„Textbasierter Taschenrechner“

- Eine oder mehrere Aufgaben zeilenweise auf Standardeingabe `sys.stdin`
  - Ergebnisse zeilenweise auf Standardausgabe `print()`
  - Dezimalzahlen werden abgerundet
- Zahlen als Worte ihrer Ziffern
- Operationen `+ - * /` und **vertauscht**
- Undefinierte Worte und Zeichen werden ignoriert

```
12 / 7 + 5 =
```

```
79
```

```
EInS - DREI DREI * fUeNF GLEICH
  1  +    33     /   5     =
```

```
7
```

Herangehensweise & Tipps

1. Problem durchlesen und verstehen
2. Prol
   A.
   B.
3. Beis
4. Programmierung
   - Nützliche Funktionen bestimmen

Wichtig, andere Herangehensweise:
Die vorhandenen Daten sollen sinnvoll gelesen werden können. Es müssen aber nicht alle Sonderfälle abgedeckt werden. Es muss auf *den Daten* funktionieren.

- Keine eigenen Funktionen oder Klassen nötig
- Inhalte der 1. Vorlesung reichen aus
- Auswahl der Operatoren & Funktionen in 1. Vorlesung hatte einen „Hintergedanken“

Warum diese Art von Aufgabe?

Typisches Data Science Problem
Data preparation

Wiederholung: Beispiel (aus letzter Vorlesung)

CSV Datei einlesen, zeilenweise alle Zahlen rausfiltern und nur die Zahlen wieder als CSV ausgeben.

```
def extract_numbers(l):
  l = l.strip()
  numbers = []
  for p in l.split(","):
    if p.strip().isnumeric():
      numbers.append(int(p))
  return numbers

def build_csv(nl):
  csv = ""
  for line in nl:
    csv += ','.join([
      str(n) for n in line
    ]) + "\n"
  return csv
```

```
f = open("name.csv", "r")
lines = f.readlines()
f.close()

new_lines = []
for line in lines:
  numbers = extract_numbers(line)
  new_lines.append([n ** 2 for n in numbers])

print(build_csv(new_lines))
```

```
$> python3 name.py

144,4182025
169,32364721
7921,1489496836
81,8008900
```

II.
Grundlagen Python

Aber zuerst:

Fragen zur Aufgabe 1?

Fragen zur Einführung letzte Woche?

Pakete & Import

- Einige Funktionen sind immer verfügbar

```
len(), str(), int(), range(), print()
```

- Weitere Funktionen oder Klassen können per `import` geladen werden

```
import time, import sys
```

- Auch eigene Funktionen und Klassen können aus einer anderen Datei geladen werden, nennt man dann *Modul*

- Pakete beinhalten Module, Klassen und Funktionen von Dritten, also Module die weder Teil von Python sind noch selbst geschrieben wurden

ort

Dateiname als Modul laden

Modul ist auch nur eine Datei, Annahme Dateien im gleichen Ordner!

name.py

```
import example_import

print(example_import)
print(example_import.bye, example_import.hello)

print(example_import.VARIABLE)
example_import.bye()
example_import.hello()
```

example_import.py

```
def hello():
  print("Hello World!")

def bye():
  print("Bye!")

False
```

Modulname „.“ Funktionsname

```
$> python3 name.py

<module 'example_import' from './example_import.py'>
<function bye at 0x100be7760> <function hello at 0x100be6200>

False
Bye!
Hello World!
```

Import mit From

name.py

```
import example_import
from example_import import hello

print(example_import.hello)
print(hello)
hello()
```

Nur die Funktion hello importieren

Funktion wird global im Skript verfügbar

example_import.py

```
def hello():
  print("Hello World!")

def bye():
  print("Bye!")

VARIABLE = False
```

```
$> python3 name.py

  <function hello at 0x100be6200>
  <function hello at 0x100be6200>
  Hello World!
```

Dieselbe Funktion!

```
import sys, time

from example_import import *

import example_import as ei
```

Was machen diese drei Import-Statements wohl?

der Standardbibliothek

Python sucht beim Import in der mitgelieferten Standardbibliothek im Installationsverzeichnis!

Keine Datei `html.py` in der Nähe, warum geht es trotzdem?

```python
import html
print(html.escape("<div></div>"))

import html.entities
print(type(html.entities.html5))
```

```
./python3.10
├── LICENSE.txt
├── abc.py
...
├── copy.py
├── datetime.py
├── html
│   ├── __init__.py
│   ├── entities.py
│   └── parser.py
├── http
│   ├── __init__.py
..
```

`html5 = {`

`def escape(s, quote=True):`

- `html` ist Teil von Python und in der Standardbibliothek
- `entities` ist ein Untermodul von `html`

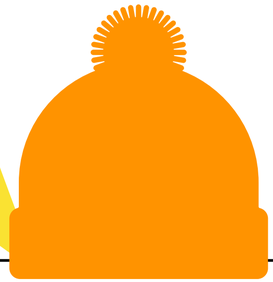
Module können nicht nur Dateien, auch Ordner sein, die dann wieder Dateien oder Order enthalten → Untermodule (beim Import mit „.“ untereilen)

https://docs.python.org/3/library/html.html

Import und Pakete

- Python sucht beim Import nach Modulen und Paketen im *Import Path* (`sys.path`)
  - Standardbibliothek
  - Installierte Pakete, z.B. mittels PiP
  - Eigene (manuell installiert, selbst geschrieben) Pakete
  - Aktuelles Verzeichnis

„Insbesondere durch viele Pakete von Dritten ist Python die erste Wahl für Anwendungen im Bereich des Data Science!“



PiP: Python Package Installer

- Installiert Pakete aus PyPi (https://pypi.org/)

`python3 -m pip`

```
$> pip3 install numpy

$> pip3 install numpy==1.23.2

$> pip3 install -r requirements.txt


$> pip3 list

$> pip3 freeze > requirements.txt
```

Einzelnes Paket (unter Angabe der Version) installieren

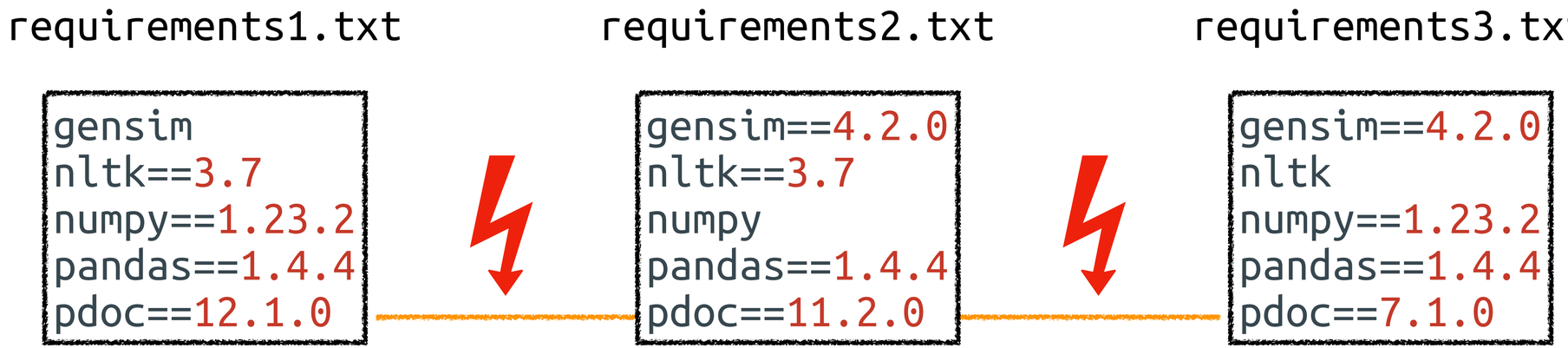
Eine Liste von Paketen installieren

Pakete auflisten/ in eine Datei mit installierter Version schreiben

requirements.txt

```
gensim
nltk
numpy==1.23.2
pandas==1.4.4
pdoc==12.1.0
requests==2.28.1
scikit-learn==1.1.2
scipy==1.9.1
sklearn==0.0
```

Paketversionen



- Verschiedene Projekte mit verschiedenen Abhängigkeiten
- Problem:
  - Passende Version jedes Pakets für jedes Projekt installieren

Wo liegt das Problem?

Virtuelle Umgebungen

```
$> cd ./Projekt1/

$> python3 -m venv ./

$> source ./bin/activate

(Projekt1) $> pip3 install -r requirements1.txt

(Projekt1) $> …

(Projekt1) $> deactivate

$>
```

Neue Umgebung in einem Ordner

Aktuelle Umgebung wird vorne im Terminal angezeigt

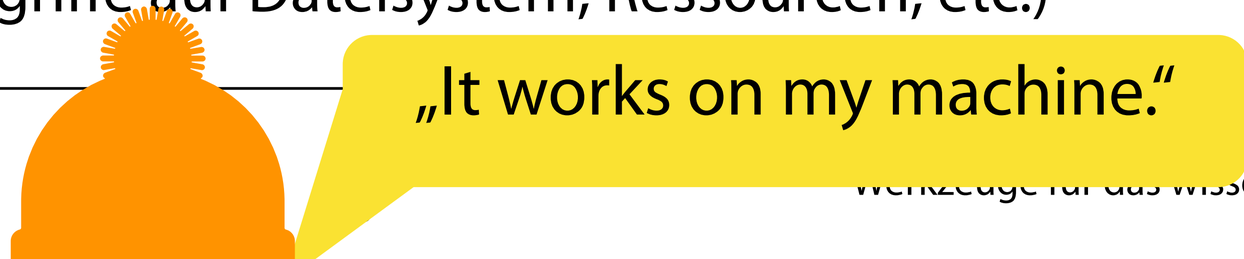
Pakete werden nur für diese Umgebung installiert.

```
./Projekt1/
├── bin
│   ├── activate
│   ├── pip3
│   └── python
├── lib
│   └── python3.10
│       └── site-packages
│           ├── numpy
│           └── pip
└── pyvenv.cfg
```

Eigenes numpy nur für diese virtuelle Umgebung!

Fazit: Virtuelle Umgebungen

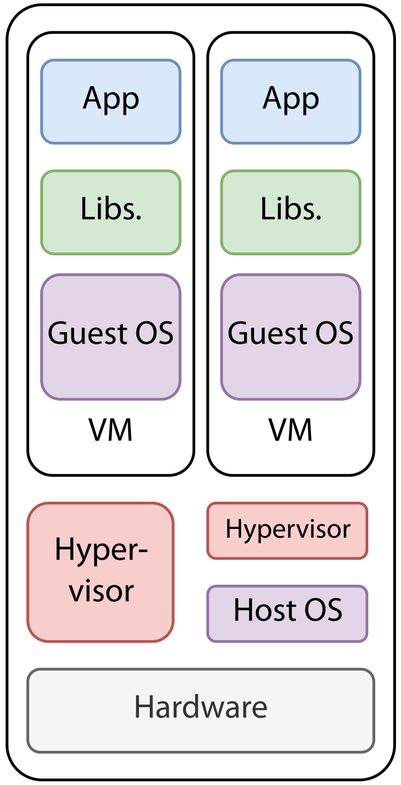
- Virtuelle Umgebungen erlauben …
  - … verschiedene Versionen gleicher (Python) Pakete
  - … verschiedene (Python) Abhängigkeiten auf dem selben Computer
- Probleme
  - Bibliotheken und Abhängigkeiten des Betriebssystems bleiben bestehen
  - Keine Absicherung (Zugriffe auf Dateisystem, Ressourcen, etc.)



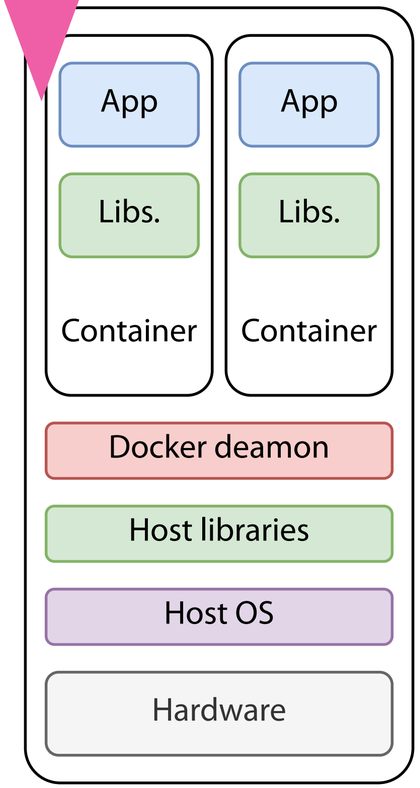
Exkurs: Virtu…l)

Programme laufen in Containern, zusammen mit ihren Bibliotheken. Die Container sind nach außen isoliert.

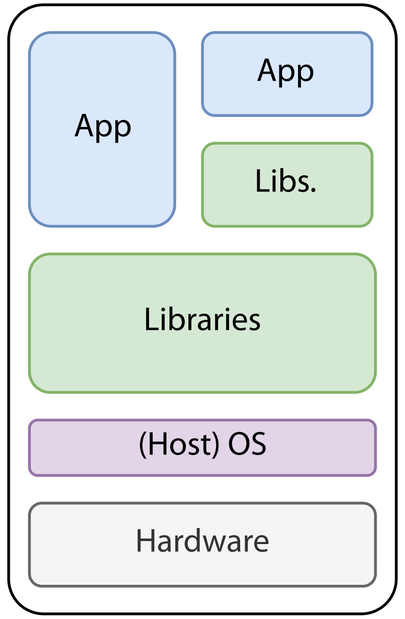
Innerhalb von Python können eigene Bibliotheken geladen werden.



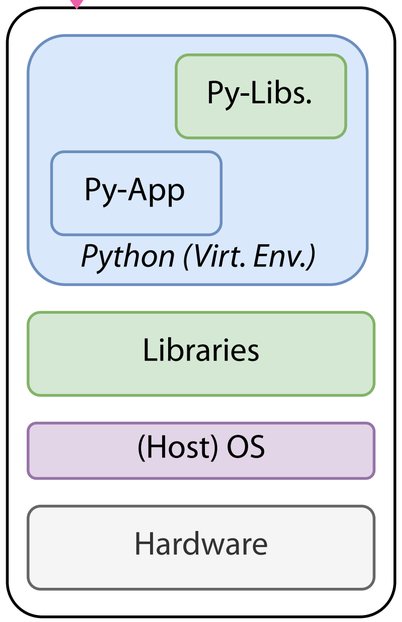
Virtuelle Maschine



Container, z.B. Docker



Keine Isolierung



Beispiel: Virtuelle Umgebung

Programme laufen in einer VM mit Bibliotheken und eigenem Betriebssystem. Die VM sind isoliert.

Programme laufen direkt auf dem OS und nutzen eigene oder geteilte Bibliotheken.

Exkurs: Virtuelle Maschinen & Container

- Virtuelle Maschinen
  - … stellen virtuelle Hardware bereit
  - … erfordern ein eigens Betriebssystem pro VM
  - … erzeugen einen Overhead
  - … isolieren jede VM vom Host und untereinander
- Container
  - … bieten Flexibilität von virtuellen Maschinen
  - … bieten nahezu native Performance
  - … isolieren jeden Container vom Betriebssystem und untereinander

Sicherheit ist relativ, viele Sicherheitsfeatures lassen z.B. bei Docker abschalten

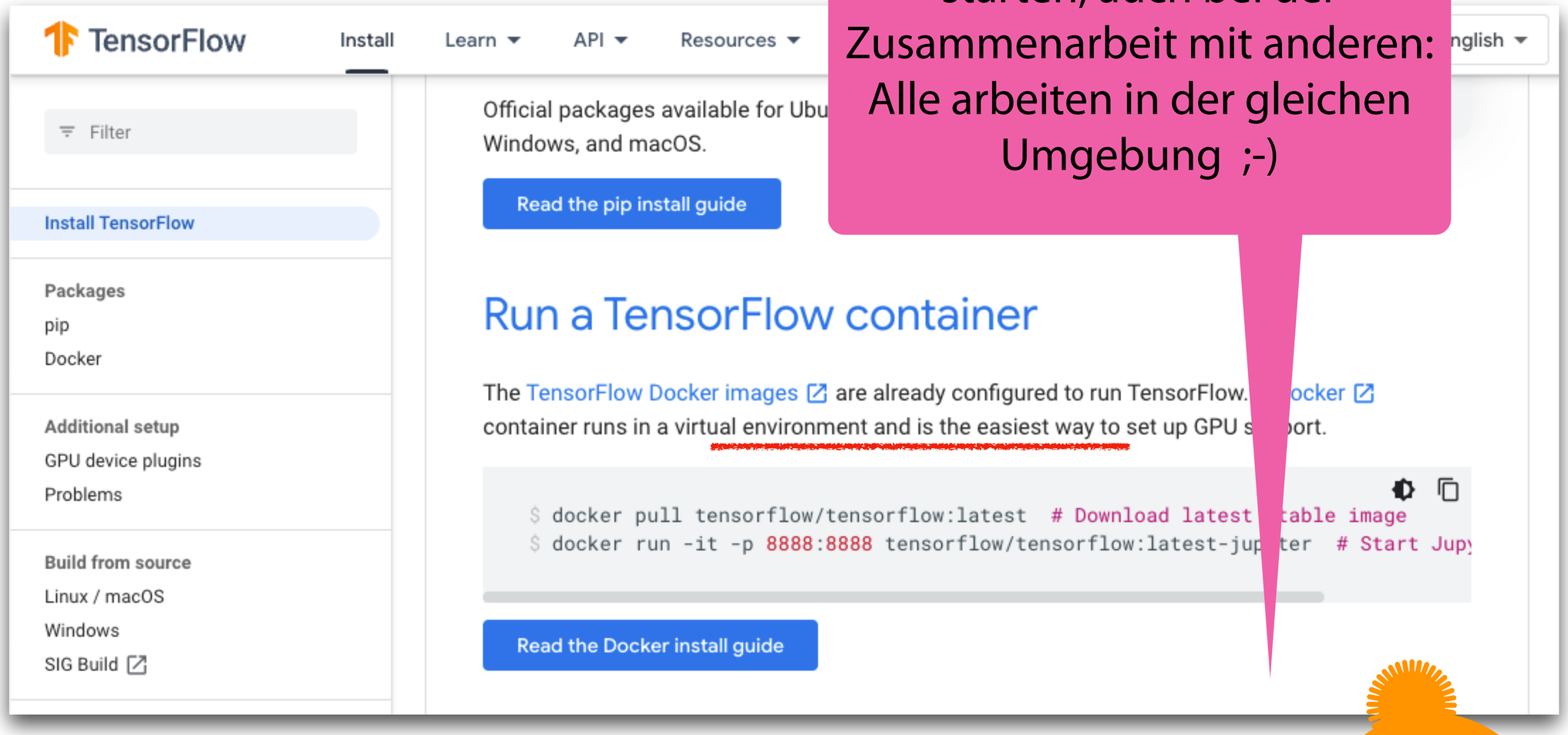
Container: Docker

Der Name „Docker“ beschreibt mehrere verschieden Dinge:

- **Dockerfile**
  Anweisungsdatei zur Erstellung von Docker-Images
- **Docker-Image**
  Ausführbares „Dateisystem“ eines Containers
- **Docker-Container**
  Aktuell ausgeführtes Docker-Images
- **Docker-Runtime**
  Ausführungsumgebung für Docker-Container/-Images
- **Docker-Daemon**
  Verwaltet Docker-Container und Images, vergibt auch Rechte über Netzwerkzugriff, Dateizugriff und Ressourcen
- **Docker-Hub**
  Externes Repository für Docker-Images der „Docker Inc.“

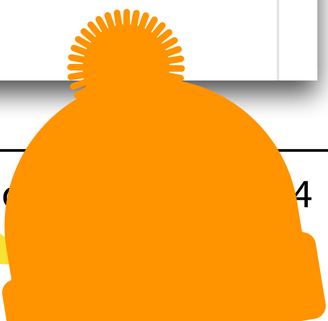
Python und Do

1. Herunterladen
2. Starten
3. Ausprobieren
4. Löschen



Und nicht nur um schnell zu starten, auch bei der Zusammenarbeit mit anderen: Alle arbeiten in der gleichen Umgebung ;-)

„Works on my machine.“



Beispiele: Python mit Docker

- Schnell etwas aus

```
docker run -it --rm python:3 python
```

- Ein Skript starten

```
docker run -it --rm
    -v "$PWD":/usr/src/myapp
    -w /usr/src/myapp
  python:3 python my-script-file.py
```

`-it` Interaktiv, TTY
`--rm` Container anschließend löschen
`python:3` Python Version 3
`python` Python direkt starten

```
un -it --rm python:3 python
nd image 'python:3' locally
3: Pulling from library/python
f606d8928ed3: Pull complete
47db815c6a45: Pull complete
bf4849400000: Pull complete
a572f7a256d3: Pull complete
8f7d05258955: Pull complete
7110f04115ae: Pull complete
c4b413c6a489: Pull complete
22311b72a3cb: Pull complete
8dcbfe38b6fa: Pull complete
Digest: sha256:53e577204d362233ee92aeb511944927
Status: Downloaded newer image for python:3
Python 3.10.7 (main, Oct  5 2022, 14:33:54) [GC
Type "help", "copyright", "credits" or "license
>>>
```

`-v` Aktuellen Order in den Container verlinken (nach `/usr/src/myapp`)
`-w` Container im Ordner `/usr/src/myapp` starten
Direkt das Skript `my-script-file.py` starten

Beispiele: Dockerfile

Nutze Debian Buster mit Python als Basis

Dockerfile

```
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Installiere die Pakete

Kopiere den Code in das Image

Definiere, wie der Code zu starten ist

```
$> docker build
      --tag python-docker
      .

$> docker run
      --publish 8000:5000
      python-docker
```

https://docs.docker.com/language/python/build-images/

Beispiel: Docker-Compose

docker-compose.yml

```yaml
services:
  app:
    image: python-docker
    ports:
      - "5000:8000"
    depends_on:
      - db
      - redis
    environment:
      - ...
  db:
    image: mariadb:latest
    restart: unless-stopped
    volumes:
      - /var/lib/mysql
    environment:
      - MYSQL_PASSWORD=secret
      - MYSQL_DATABASE=app
      - MYSQL_USER=user
  redis:
    image: redis:alpine
    restart: unless-stopped
    volumes:
      - ./redis/:/data
```

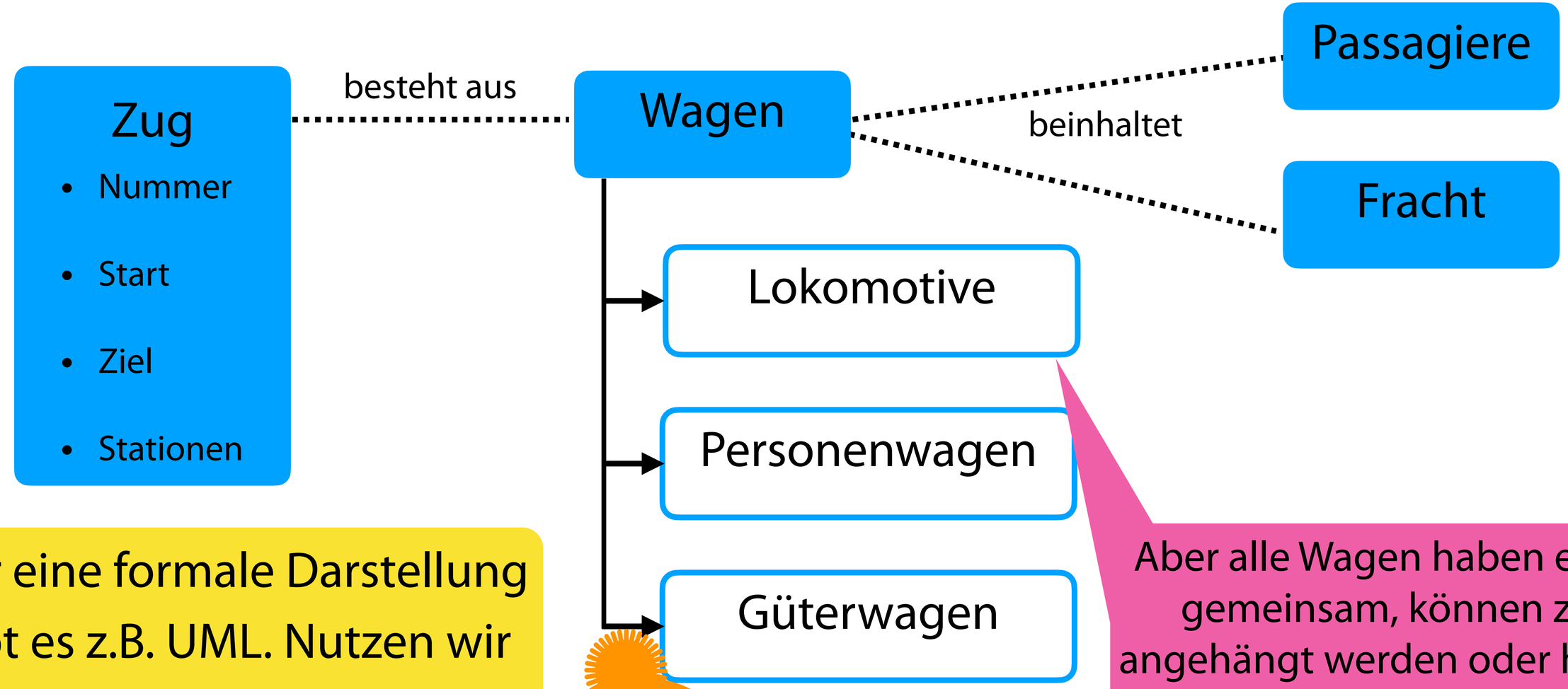
Alle benötigten Komponeten können mit Konfiguration hier angegeben werden

Images werden aus DockerHub geladen

Und anschließend direkt gestartet

```
$> docker-compose up -d

  Pulling db
  Pulling redis

  Starting db ... done
  Starting redis ... done
  Starting app ... done
```

Objektorientierte Programmierung (OOP)



„Für eine formale Darstellung gibt es z.B. UML. Nutzen wir hier aber nicht.“

Aber alle Wagen haben etwas gemeinsam, können z.B. angehängt werden oder haben einen Standort, …

Klassen & Objekte

```
class Train():

  def __init__(self, n, s, d):
    self.number = n
    self.start = s
    self.destination = d
    self.stations = []

  def add_station(self, s):
    self.stations.append(s)

hl_hh = Train("RE 8", "Lubeck Hbf", "Hamburg Hbf")
hl_hh.add_station("Lubeck Hbf")
hl_hh.add_station("Reinfeld (Holstein)")
hl_hh.add_station("Bad Oldesloe")
hl_hh.add_station("Hamburg Hbf")

print(hl_hh)
print(hl_hh.stations)
```

Klasse mit Attributen und Methoden

Objekt erzeugen

Methode (Funktion eines Objekts) aufrufen

Zug

- Nummer
- Start
- Ziel
- Stationen

```
$> python3 name.py

<__main__.Train object at
0x106b9c0a0>
['Lubeck Hbf', 'Reinfeld
(Holstein)', 'Bad Oldesloe',
'Hamburg Hbf']
```

Klassen & Objekte

- Klasse bündelt Funktionalität (Methoden) und Werte (Attribute) eines „Typs“
- Objekt ist die Instanz einer Klasse
  - Attribute (Variablen der Klasse) für jedes Objekt verschieden
  - Methoden (Funktionen der Klasse) für alle Objekte gleich

- Keine Kapselung durch `private`, `protected`, … wie z.B. in Java
- Alles ist intern mit `self` und extern über das Objekt zugreifbar
- *Konvention: Namen interner Attribute und Funktionen beginnen mit _*

Man beachte auch `__init__` für Python-Interne Namen

Ober- und Unterkla

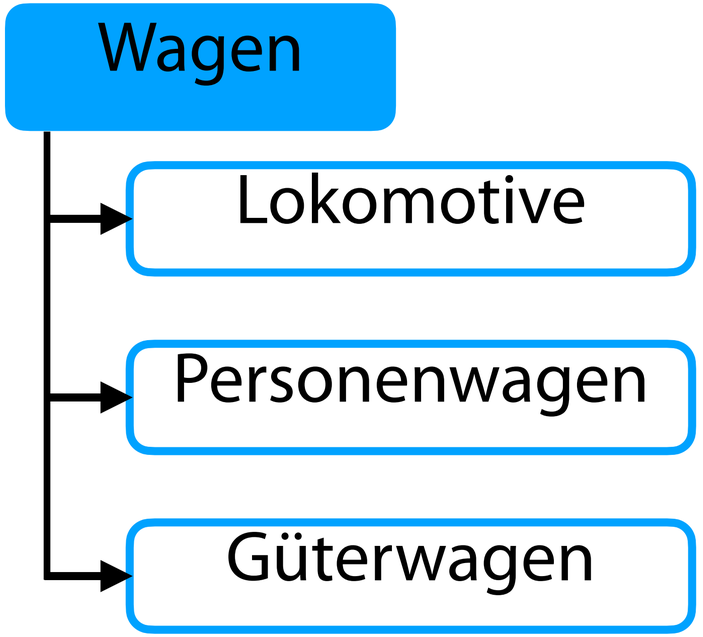
Güterwagen ist ein Wagen und hat somit auch alle Methoden von Wagen sowie weitere eigene.

```
class Wagon():

  def __init__(self, p, n):
    self.previous_wagon = p
    self.next_wagon = n
```

Jeder Wagen hat Vorgänger und Nachfolger

```
class GoodsWagon(Wagon):
  SELF_POWERED = False

  def __init__(self, *args):
    super().__init__(*args)
    self.goods = []

  def add_good(self, g):
    self.goods.append(g)

class PersonWagon(Wagon):
  SELF_POWERED = True

  def person_enter(self, p):
    pass

class Locomotive(Wagon):
  SELF_POWERED = True
```

Ein Güterwagen kann nicht selbst fahren, das gilt für alle, d.h. `SELF_POWERED` ist ein statisches Attribut.



`super()` gibt die Oberklasse an und hier dann den Konstruktor `__init__()`

Warum steht hier `pass`?

`*args` nimmt alle Parameter als Tupel auf und `*args` gibt sie dann wieder als Parameter weiter.

Ober- und Unterklassen

Zur Erinnerung (verkürzt):

```
class Wagon():
  def __init__(self, p, n):
    self.previous_wagon = p
    self.next_wagon = n

class GoodsWagon(Wagon):
  SELF_POWERED = False
  def __init__(self, *args):
    super().__init__(*args)
    self.goods = []
  def add_good(self, g)

class PersonWagon(Wagon):
  SELF_POWERED = True
  def person_enter(self, p)

class Locomotive(Wagon):
  SELF_POWERED = True
```

```
w = Wagon("P", "N")
print(w.previous_wagon,
  w.next_wagon)

gw = GoodsWagon("a", "c")
print(gw.previous_wagon,
  gw.next_wagon, gw.goods)
gw.add_good("Metal")
gw.add_good("Silver")
print(gw.previous_wagon,
  gw.next_wagon, gw.goods)

pw = PersonWagon("b", "d")
pw.person_enter("Magnus")

l = Locomotive(None, "a")
print(l.previous_wagon, l.next_wagon)
print(l.SELF_POWERED,
  Locomotive.SELF_POWERED)
```

```
$> python3 name.py

P N

a c []

a c ['Metal', 'Silver']

None a
True True
```

urück zum Zug

Code von den Folien davor zusammengefügt und leich verändert.


```
l = Locomotive(None)
gw1 = GoodsWagon(l)
gw1.add_good("Getreide")
gw2 = GoodsWagon(gw1)
gw2.add_good("Hafer")
gw2.add_good("Haferflocken")
gw3 = GoodsWagon(gw2)

t = Train("GW 12", "Luebeck", "Muenchen")
t.add_station("Hannover")
t.add_station("Kassel")
t.add_station("Regensburg")
t.set_wagons([l, gw1, gw2, gw3])

print("Zuglänge", len(t))
print(t)
print("Wagen 2 mit", gw2.goods)
```

Formatieru
Zuges behan

Zug „GW 12“

- Wagen: Lokomotive
- Wagen: GW 1
  - Getreide
- Wagen: GW 2
  - Hafer
  - Haferflocken
- Wagen: GW 3

OOP mit Python

- Python unterstützt Mehrfachvererbung (zwei Oberklassen)

```
class MyClass( Class1, Class2 ):
```

- Statische Methoden (analog zu statischen Attributen) haben kein `self` in der Definition

```
def my_static_method(a, b):
MyClass.my_static_method(a, b)
```

- Objekte und Klassen haben einige interne Attribute

```
my_object.__class__
my_object.__class__.__name__
my_object.__dict__
my_object.__class__.__dict__
```

Klasse des Objekts

Name der Klasse (`str`)

Wörterbuch mit allen Attributen

Wörterbuch mit Methoden und statischen Attributen

Zusammenfassung

- Aufgabe 1

- Pakete & Import
- Virtuelle Umgebungen & Docker
- OOP

Die wichtigsten Grundlagen haben wir damit geschafft.

Nächste Woche ist Feiertag und danach kommen noch einige fortgeschrittene Themen zu Python ;-)

Inhaltsübersicht

1. Programmiersprache Python
   a) *Einführung, Erste Schritte*
   b) *Grundlagen*
   c) Fortgeschritten
2. Auszeichnungssprachen
   a) LaTeX, Markdown
3. Benutzeroberflächen und Entwicklungsumgebungen
   a) Jupyter Notebooks lokal und in der Cloud (Google Colab)
4. Versionsverwaltung
   a) Git, GitHub
5. Wissenschaftliches Rechnen
   a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
   a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
   a) Grundlegende Ansätze (Datensätze, Auswertung)
   b) Einfache Verfahren (Clustering, …)
8. DeepLearning
   a) TensorFlow, PyTorch, HuggingFace Transformers