Inhalt

[Diverses 2](#_Toc70326334)

[Funktionen in Dictionarys speichern 4](#_Toc70326335)

[Scope (Angela Yu Day 12) 5](#_Toc70326336)

[Reguläre Ausdrücke: 5](#_Toc70326337)

[Datentypen 6](#_Toc70326338)

[None 6](#_Toc70326339)

[Sequenziellen Datentypen 6](#_Toc70326340)

[Datenstrukturen 8](#_Toc70326341)

[Lamda-Funktion 12](#_Toc70326342)

[Entscheidungen 13](#_Toc70326343)

[Schleifen 14](#_Toc70326344)

[Funktionen 15](#_Toc70326345)

[Standardbibliotheke 18](#_Toc70326346)

[Datetime 18](#_Toc70326347)

[String 18](#_Toc70326348)

[Rand 19](#_Toc70326349)

[Objektorientierung 20](#_Toc70326350)

[Exceptions 22](#_Toc70326351)

[Dateien/Datenspeicherung 24](#_Toc70326352)

[NumPy 26](#_Toc70326353)

[Basics 26](#_Toc70326354)

[Filtern eindimensionaler Arrays 26](#_Toc70326355)

[Mehrdimensionale Arrays 26](#_Toc70326356)

[Exkurs: Wie Numpy intern Abläuft 27](#_Toc70326357)

[Pandas 28](#_Toc70326358)

[Laufzeitverhalten 32](#_Toc70326359)

[Requests 34](#_Toc70326360)

# Diverses

|  |  |
| --- | --- |
| import docx  import json  import pandas  import requests  import smtplib  import twilio  import cologram  import prettytable | Word Dokumente  Built-In  Datenanalyse  API Calls  Emails  SMS, WhatsApp, Telefon, …  Farben aus Bildern extrahieren  Daten in ASCI Tabelle organisieren für Konsolenausgabe |

|  |
| --- |
| main.py 🡪 Compiler 🡪 main.pyc 🡪 Interpreter 🡪 ausgeführtes Programm  Compiler: Baut ein ausführbares Programm und Interpreter: Führt das Programm aus |

|  |
| --- |
| import sys  print(sys.argv)   * Wenn ich mein Programm über die Konsole aufrufe kann ich nun beliebig viele Paramter als Liste ab index 1 mit ins Programm übergeben |

|  |
| --- |
| Relativer Pfad ab main()  # \_\_file\_\_ ist eine Konstante die den Pfad meiner aktuellen Datei liefert (hier main.py)  # os.path.dirname übergebe ich die Datei und dieser liefert mir den Pfad zum Order worin meine Datei liegt # Die Join methode setzt nun den Pfad mit dateinamen (oder unterordmer) zusammen (os abhängig mit / oder \)  path\_data = os.path.join(os.path.dirname(\_\_file\_\_), "data.json")  path\_logo = os.path.join(os.path.dirname(\_\_file\_\_), "unterordner", "logo.png")  so könnte ich Projektordner rumkopieren, der Pfad zu den Unterdateien würde immer passen.  Die join Methode nimmt beliebig viele Paramter an (z.B. auch “..“ eine Eben hoch). So kann ich einen ganzen Pfad relativ zu meiner main() bauen  **Beispiel:**  Ich habe meine main(), im selben Ordner habe ich einen Ordner names worin 20 .txt dateien sind.  import os  # Hier baue ich den Pfad zusammen für den Ordner names  pfad = os.path.join(os.path.dirname(\_\_file\_\_), "names")  # in all\_files ist nun eine List mit allem was in dem Ordner names ist  all\_files = os.listdir(pfad)  # jetzt öffne ich alle 20 Files aus dem Ordner names  for file in all\_files:      # In jedem Schleifendurchgang den Pfad zur konkreten Datei      one\_file\_path = os.path.join(pfad, file)      # Diese dann öffnen und auslesen      with open(one\_file\_path, "r", encoding="UTF-8") as file:          file\_text = file.read()  **WICHTIG:** Das oben funktioniert manchmal nicht (interaktive Shell), aber das schon: pfad\_names\_csv = os.path.join(os.path.dirname(os.path.abspath("\_\_file\_\_")), "names.csv") |
| Operatoren in und not:   * string in liste -> True oder False * char in string -> True oder False * string not in liste -> True oder False * not string in liste -> True oder False |

|  |
| --- |
| type und isinstanceof von Variablen überprüfen:    Anwendungsbeispiel |

|  |
| --- |
| Ganzzahl Division mit // 🡪 5//3 = 1 Pendant: 5%3 = 2 |

|  |
| --- |
| **Import Teil 1/2 – Für Module die es schon gibt:**  Import (main und modul im selben Ordner ohne Klassen):   1. import modul 🡪 Importiert die gesamte Datei „modul.py“ 2. from modul import def1, def2 🡪 Importiert nur die Funktionen def1 und def2 von „modul“ 3. Beispiel Paket import Varianten:    1. Import matplotlib.pyplot 🡪 matplotlib.pyplot.plot([1,2,3],[4,5,4])    2. Import matplotlib.pyplot as plt 🡪 plt.plot([1,2,3],[4,5,4])    3. from matplotlib import pyplot 🡪 pyplot.plot([1,2,3],[4,5,4])   Merke zu Klassen in modul (import modul):   * Wenn in der modul Datei nur Funktionen sind dann rufe ich sie in der main so auf: 🡪 modul.def1() * Wenn in der modul Datei eine Klasse mit der Funktion ist, dann muss ich in der main so: 🡪 modul.klasse.def1 Und hier kommt dann das ins Spiel: klasse = modul.klasse 🡪 dann kann ich in der main wieder einfach nur mit klasse.def1() arbeiten   **Import Teil 2/2 – Für Module die ich selber erstelle:**   1. **main.py** und **module (Ordner)** liegen im selben Ordner 2. Im **Ordner module** liegt eine leere **\_\_init\_\_.py**  🡪 Damit definiere ich den Ordner als Python Modul 3. Im **Ordner module** liegen jetzt auch meine Python Datei welche Funktionen oder Klassen/Methoden enthalten     Jetzt kann ich in der **main** verschiedene Dateien innerhalb vom **Ordner module** importieren!  Problem: from module import \* 🡪 Also **alle Python-Files** aus **Ordner** **module** funktioniert nicht! 🡪 Muss explizit erlaubt werden:    Siehe in meine \_\_init\_\_.py, hier liste ich alle Python-Files auf welche mit from module import \* importiert werden sollen (implizit erlaube ich damit auch die **\*** Schreibweise)  Problem: die gewohnte **Import** Schreibweise funktioniert immer noch nicht:  import module  module.hello.show()  Dafür müssen wir in der **\_\_init\_\_()** wieder was ergänzen, Code für **Import-Variante** schaut so aus:    Siehe in meine \_\_init\_\_.py, hier liste ich auf andere Weise wieder alle Python-Files auf welche mit import module importiert werden sollen (implizit erlaube ich damit auch die **import** Schreibweise)  **Zusammenfassung – Der Komplette Code in dem ich beide Varianten erlaube/einstelle:**    Hello = module.hello  Hello.show() |

|  |  |
| --- | --- |
| Funktionen in Dictionarys speichern def add(x, y):      return x + y  def sub(x, y):      return x - y  operations = {"+": add, "-": sub}  num1 = 5  operator = "-"  num2 = 3  #            1)         2)        4)  result = operations[operator](num1, num2) | 1. In Dictionary „operations“… 2. … hole ich mir den Value für den entsprechenden Operator 🡪 was in diesem Fall eine Funktion ist. 3. Als Ergebnis von 1 & 2 habe ich quasi den Funktionsname 4. Und übergebe der Funktion direkt die Paramter womit ich insgesamt die Funktion aufrufen 5. In diesem Fall steht in der letzten Zeile quasi: 1) result = sub(5, 3) 2) result = 2 |

|  |
| --- |
| Scope (Angela Yu Day 12)  * Alles was man definieren kann (Variable, Funktionen,..) hat einen Scope, bzw. Namespace. * Eine Funktion erschafft EINEN neuen Local-Scope * If und while erschaffen KEINEN neuen Local-Scope   Wenn ich z.B. in einer Funktion eine Variable definiere die GLEICH heißt wie eine gültige globale Variable, dann verändere ich die globale NICHT sondern erschaffe eine eigene lokale Variable -> Im Umkehrschluss kann ich auf die globale IN der Funktion nicht mehr zugreifen!!!  name = "Yagmur"  def test1():      # name (= Yagmur) wäre hier gültig wird aber durch eine lokale "überschattet"      name = "Ilhan" # Auf globale name (= Yagmur) kann nicht mehr zugegriffen werden      print(name) # Ausgabe = Ilhan    test1()  print(name) # Ausgabe = Yagmur | Obwohl vorher name = Ilhan in FKT gesetzt wurde   * Lesen der globalen Variable name in Funktion würde ohne weiteres gehen, aber verändern der Variable nur wenn ich diese in der Funktion mit “global name“ zuvor erlaube:   ABER: **VERMEIDEN**!!! -> Mit **return** arbeiten und neune Wert einfach zuweisen!!!!  Globale variablen (nur)/besonders Sinnvoll für Konstanten die man vermutlich nie verändern wird:   * Diese dann anders wie normaler variablen ALL\_UPPER\_CASE benennen |

|  |
| --- |
| Reguläre Ausdrücke: Mit regulären Ausdrücken kann ich nach beliebigen Strings in Strings suchen. Es ist eine formale Sprache. Bei Bedarf separat weiterbilden, relativ großes/komplexes Thema.   * Pythex.org -> Reguläre Ausdrücke Editor * Regexr.com -> Sehr große Hilfe (Ressourcen, dynamische Erklärungen, CheatSheet,…) * Import re -> Das Paket für Python |

# Datentypen

Datentyp einer Variablen prüfen: type(variable)

## None

* Spezielles Schlüsselwort; es gibt nur eine Instanz des None.
* if ref is None effizienter als if ref == None

## Sequenziellen Datentypen

|  |  |
| --- | --- |
| Echte Kopie einer Sequenz | |
| s1 = [“abc“]  s2 = s1  s1 == s2 🡪 True  s1 is s2 🡪 True | s1 = [“abc“]  s2 = s1[:]  s1 == s2 🡪 True  s1 is s2 🡪 False |
| Hier wird nur die Referenz von s1 der Variablen s2 zugewiesen, wodurch es sich bei beiden um dasselbe Objekt (Identitäten) handelt. | Hier wird eine echte Kopie von s1 angefertigt und als eigene Instanz im Speicher abgelegt wohin die Variable s2 referenziert. s1 und s2 verweisen nun auf zwei unterschiedliche Objekte (Identitäten). |
| Wenn es um unveränderliche Datentypen geht, kann es sein das auch im rechten Beispiel die Überprüfung mit is auf True geht, da beide Variablen auf denselben Wert zweier unveränderlichen Instanzen zeigen würden, referenziert Python einfach für beide Variablen aus effizienzgründen auf dieselbe Instanz (mach ja keinen Unterschied, da man den Inhalt e nicht ändern kann). | |

|  |  |
| --- | --- |
| Packing / Unpacking | |
| Packing Tuple:  datum = 26, 7, 1987 🡪 Ausgabe datum: (26, 7, 1987)  Unpacking Tuple:  Tag, monat, jahr = datum 🡪 Ausgabe monat: 7 | Klammern können entfallen, es wird automatisch als ein Tuple behandelt. |
| Unpacking str:  a,b,c = “abc“ -> Ausgabe b: “b“ | Auch strings können entpackt werden. |
| Beispiel Unpacking Listen:  zahlen = [15, 18, 12, 11,10]  elf, \*andere, zehn = zahlen  oder  \*irgend, elf, zehn bzw., fünfzehn, achtzehn, irgend\* | Mit \* können auch Blockweise entpackt werden. Gilt nicht nur für Listen |
| Nützlicher Hack:  a, b = 10, 20  a, b = b, a | Mit Packing und unpacking, kannn z.B. sehr elegant der Inhalt zweier Variablen vertauscht werdend |

**Speziell zu List**

* Mutabler Datentyp
* Es gibt zusätzliche Operationen und Methoden auf diesen Datentyp
* Beachte auch List Comprehensions, also „inline“ zur Definition von Listen

**Speziell zu Tuple**

* Unmutabler Datentyp
* Keine erweiterten Funktionen gegenüber Listen
* Keine zusätzlichen Operationen/Methoden zzgl. zu Grundstock für sequenzielle Datentypen
* Wenn dieser Grundstock reicht, dann immer Tuple nutzen 🡪 Effizienter als Listen

Achtung: Das Tuple unmuatle sind kann irre führen wenn darin Referenzen zu Instanzen gespeichert werden (z.B. ein Tuple voller Listen). Die Referenz kann sich zwar nicht mehr ändern aber der Inhalte der Instanz sehr wohl!!!

**Speziell zu str**

Asd

**Speziell zu byte**

Asd

**Speziell zu bytearray**

asd

# Datenstrukturen

**Liste**

* Letztes Element zurückgeben und löschen: a = fahrzeug1.pop()
* Mit + kann man Listen zusammenfügen [“A“]+[“B“]=[“A“, “B“]
* Eintrag löschen: del liste[index] oder liste.remove(“Eintrag“)
* letzte Element der Liste: liste[-1]
* Teilliste: liste[1:3] oder liste[2:-2]
* Index eines Elements: liste.index(„element“)
* Liste zusammenführen liste = ["a", "b", "c"] print(", ".join(liste))
* String in Liste splitten i = "a, b, x" print(i.split(", "))

List Comprehension 🡪 hinter in können alle Arten von Sequenzen stecken (string, range, tuple, list):

* + Beispiel 1  
     x = [1, 2, 3, 4]  
     y = [2 \* element for element in x] 🡺 y = [1, 4, 6, 8]
  + Beispiel 2  
     x = [“Max“, “Peter“, “Susi“]  
     y = [len(element) for element in x] 🡪 y = [3, 5, 4]
  + Beispiel 3  
     x = [element / 10 for element in range(0,5)] 🡪 x = [0, 0.1, 0.2, 0.3, 0.4]  
     y = [element \* 2 for element in x] 🡪 y = [0, 0.2, 0.4, 0.6, 0.8]
  + Man kann sogar if in die comprehension noch dazu bauen
    - Der zweite Teil: welche elemente ich aus welcher Liste durchgehen möchte
    - Der erste Teil: Hier kann ich die Elemente jetzt modifizieren
    - Der dritte Teil: und hier kann ich sogar mit if selektieren welche elemente ich betrachten möchte  
      x = [1, 2, 3, 4]  
      y = [2 \* element for element in x if element % 2 == 0] 🡺 y = [4, 8]

Möglichkeiten eine Liste zu kopieren:

* Slicing: newrow = row[:]
* List-Comprehension: newrow = [i for i in row]
* Funktion: newrow = row.copy()
* Merke auch: row = row[:]  
  rechte Seite von = erstelle ich neue Liste und dann überschreibe ich die alte Referenz mit der neue Referenz auf die neu erstellt Liste! Gilt auch für die anderen beiden Möglichkeiten?

**Dictionaries**

* Dict erstellen: fahrzeug1 = {“Marke“: “VW“, “Modell“: “Golf“}  
  Abfgragen 1: fahrzeug1[“Modell“] 🡪 Golf sonst Exception
* Abfragen 2: fahrzeug1.get(“Modell“) 🡪 Golf sonst None
* Dict ausgeben: print(fahrzeug1)
* Dict erweitern/editieren: fahrzeug1[“Preis“] = 5000   
  🡪 Falls Key vorhanden: Value editieren, falls Key nicht enthalten Key/Value hinzufügen
* Eintrag löschen: del fahrzeug1[“Preis“]
* Dict löschen: fahrzeug1 = {}
* Zwei weitere alternative Möglichkeiten Dictionaries zu erstellen:
  + fahrzeug1 = dict ([("marke", "VW"), ("modell", "Golf"), ("baujahr", 2012), ("preis", 5000)])
  + fahrzeug1 = dict (marke="VW", modell="Golf", baujahr=2012, preis=5000)
* Es gibt auch Dict-Comprehensions:

|  |
| --- |
| names = ["Alex", "Beth", "Caroline", "Dave", "Eleandro", "Freddie"]  # Eine dict mit dict-comprehension aus einer Liste erstellen  students\_score = {student: random.randint(1, 100) for student in names}  # Eine neue dict mit dict-comprehension aus einer alten dict erstellen  passed\_students = {student: score for student, score in students\_scor.items() if score > 50}  # # Merke: das Objekt worin iteriert wird, kann alle Arten von Sequenzen sein |

* asd

Dictinary ist Pyhon buil-in. Es gibt auch ein defaultdict aus Modul Collections das sehr praktisch ist:

from collections import defaultdict

# ======== Langversion ========

# def generate():

#     return 0

# d = defaultdict(generate)

# ======== Kurzversion ========

d = defaultdict(int)    # die Funktion int() liefert immer 0 zurück

# Hier wird also immer die Zahl null geliefert wenn eine Eintrag im Dict nicht existiert"

# Anwendung:

Es werden alle Wörder in words gezählt und in d als Anzahl gespeichert. Der Clou, immer wenn ein word in d nicht existiert, wird er automatisch mit 0 als Value angelegt.

words = ["Hallo", "Hallo", "Welt"]

for word in words:

    d[word] = d[word] +  1

print(d)

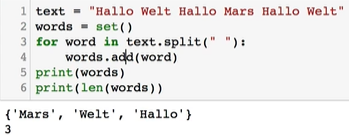
|  |  |
| --- | --- |
| dic = {"eins": "one", "zwei": "two"}  # Variante 1  for key in dic:      print(key)      print(dic[key]) | dic = {"eins": "one", "zwei": "two"}  # Variante 2  for key, value in dic.items():      print(key)      print(value) |

**Tupel**

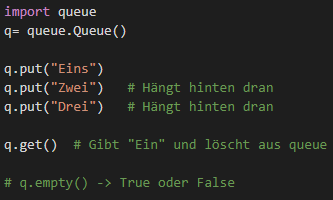
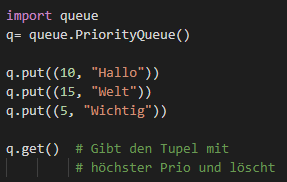
* Tupel sind unveränderlich
* Tupel anlegen: fahrzeug1 = (“VW“, “Golf“, 2011, 5000)
* Werte hinzufügen indem man den Tupel überschreibt:
  + farbe = “blau“
  + fahrzeug1 = fahrzeug1 + farbe
* Tupel entpacken Beispiel 1:  
   student = [“Max“, 22, “Mathe“]  
   name, alter, fach = students
* Tupel entpacken Beispiel 2:  
   def getStudent():  
   return (“Max“, 22, “Mathe“)  
   name, alter, fach = getStudent()
* Tupel entpacken Beispiel 3:  
   students [(“Max“, “22“), (“Monika“, “19“)]  
   for name, alter in students:  
   print(name + “, “ alter)
* Tupel entpacken aus Dictionarys:  
   d = {"München":"MUC", "Berlin":"BER", "Istanbul":"IST"}  
   print(d.items()) 🡪 dict\_items([('München', 'MUC'), ('Berlin', 'BER'), ('Istanbul', 'IST')])  
   for a, b in d.items():   
   print(a + " : " + b)

**Set**

* Wie eine Liste, nur das automatisch sichergestellt ist, dass alle Elemente nur einmal vorkommen.
* Set ist unsortiert.
* autos = {“BMW“, “AUDI“, “Mercedes“}
* autos.add(“Opel“)
* autos = Set() —> leeres Set erstellen (analog zu dict = {})
* z.B. eine Liste in einen Set packen und damit automatisch die Duplikate eliminieren



**Queue und Priority Queue:**

Priority Queue erwartet Tupel als Input und sortiert automatisch anhand des ersten Parameters!  
Merke: wenn die Queue absteigend sortiert werden soll, einfach ein - vor den ersten Parameter

**Verschachtelung – List und Dictionary IN Dictionary**

travel\_log = {

    "France": {

        "cities\_visited": ["Paris", "Lille", "3"],

        "total\_visits": 4

        },

    "Germany": {

        "cities\_visited": ["Berlin", "München", "Dortmund"],

        "total\_visits": 12

        }

}

**Verschachtelung – List und Dictionary IN List**

travel\_log = [

    {

        "Country": "France",

        "cities\_visited": ["Paris", "Lille", "3"],

        "total\_visits": 4

    },

    {

        "Country": "Germany",

        "cities\_visited": ["Berlin", "München", "Dortmund"],

        "total\_visits": 12

    }

]

**Zusammenfassung:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Liste | Dictionary | Set | Prioroty Queue | Tupel |
| list = [] ODER list = list() | dict = {} ODER dict = dict() | set = set() | Import queue  que = queue.PriorityQueue() | tup = () ODER tup = tuple() ?!? |
| list = [“VW“, 2000] | dict = {“Mar“:“VW“, “Mod“:“3“} | set = {“BMW“, “AUDI“} | tup = (“VW“, “Golf“) |
| Kann nachträglich erweitert werden | Kann nachträglich erweitert werden | Kann nachträglich erweitert werden | Kann nachträglich erweitert werden | Kann nachträglich nicht erweitert werden |
| Reihenfolge wird beibehalten | Reihenfolge wird nicht beibehalten | Reihenfolge wird nicht beibehalten | Automatische Sortierung je nach Priorität |  |
| Elemente dürfen mehrfach vorkommen | Values dürfen mehrfach vorkommen, Keys aber nur einmal | Elemente dürfen nicht mehrfach vorkommen |  | Elemente dürfen mehrfach vorkommen |
|  | Speichert Key:Value Zuordnungen | Indexschreibweise nicht möglich | Zugriff nur auf nächstes Element in der Liste möglich (wird danach entfernt) | Sonst ähnlich zu Liste |

**Standard Sortierung:**

sort() Funktion einer Liste vs. generelle sorted() Funktion von Python

|  |  |
| --- | --- |
| **Liste**  Listen haben eine sort() Funktion die man nutzen kann. Standardfall:  l = [3, 1, 8] # l = ["F", "A", "E"]  l.sort()# l.sort(reverse=True)  - Strings können genauso sortiert werden  - reverse=true ist Sortierreihenfolge ändern  Eigene Sortierung:  def getLen(item):      return len(item)  l = ["Max", "Monika", "Erik", "Franziska"]  l.sort(key=getLen)  !! hier übergebe ich die Funktion getLen NICHT dessen Rückgabewert get(). Mit Klammer würde er zuerst die Funktion getLen ausführen und nur den Rückgabewert an die Sort übergeben!!  Kürzer:  l = ["Max", "Monika", "Erik", "Franziska"]  l.sort(key=len)  Meine getLen oben macht nichts anderes als die Länge zu berechnen, aber dafür hat Python ja e schon die len() Funktion. Einfach die direkt nutzen. | **z.B. für Dictionary**  Dictionarys haben KEINE eigene sort() Funktion. Hier die generelle sorted() Funktion von Python nutzen:  d = {"Köln":"CGN", "Budapest":"BUD"}  print(sorted(d))  oder andere Sortierreihenfolge:  d = {"Köln":"CGN", "Budapest":"BUD"}  print(sorted(d, reverse=True)) |
| Die sort() Funktion sortiert inplace, also die originale Liste wird soritert. Die sorted() Funktion sortiert auch, verändert aber die originale Liste nicht, sondern erstellt eine neue Liste. | |

**Sortierung per Lambda Funktion:**

Lamda-Funktion bedeutet quasi inlin-Funktion.

|  |  |
| --- | --- |
| Konventionelle Variante:  # Nach dem Semester der Studenten  # sortieren (mit spezieller sort-Funktion der Liste)  students = [      ("Max", 3),      ("Monika", 2),      ("Erik", 3),      ("Franziska", 1)  ]  def student\_key(student):      return student[1]  students.sort(key=student\_key) | Lambda-Funktion Variante:  students = [      ("Max", 3),      ("Monika", 2),      ("Erik", 3),      ("Franziska", 1)  ]  students.sort(key=lambda student: student[1])  Meiner lambda Funktion wird student als Parameter übergeben und der Rückgabewert ist student[1]. Das es sich um die liste students handelt ist klar weil ich die sort-Funktion direkt von der Liste ausführen. |

Mini-Exkurs: Lambda-Funktion

|  |  |
| --- | --- |
| f = lambda student: student[1]  f(("Max", 1)) | Wenn ich der Funktion f das Tupe (“Max“, 1) übergebe, dann ist der return = 1 |

# Entscheidungen

Einzeiler if:

* if x: print(“Hallo“) bzw. if x: print(“Hallo“); print(“Welt“)

Conditional Expressions:

* var = (20 if x == 1 else 30)
* print("x hat den Wert 1" if x == 1 else "x ist ungleich 1")
* xyz = (a \* 2 if (a > 10 and b < 5) else b \* 2)

# Schleifen

**For-Schleife**

Besonders praktisch bei Listentyp Dictionarys:

dic = {"Marke":"VW","Modell":"Golf","Baujahr":2011,"Preis":5000}

for inhalt in dic:

print (inhalt, dic[inhalt])

Die Variable Inhalt nimmt den Schlüssel auf, daher mit dic[inhalt] auf den Wert zugreifen

For erstellt eine Kopie der Liste, daher verändern der original Liste in der for-Schleife nicht möglich, sondern mit while oder aber for und index:

liste = [3, 6, 123, 54, 927]

for inhalt in enumerate(liste):

liste[inhalt[0]] = inhalt[1] \* 2

print (liste)

* enumerate gibt Tupel einer Liste zurück, mit 0 greift man auf den Index zu 1 auf den Inhalt
* Wenn man Inhalte verändern will, ist die while-Schleife sinnvoller

Auch praktisch beim zählen (vorwärts. Rückwärts, in verschiedenen Schritten:

for n in range(100): 10,30 oder 50,10,-10 usw

print (n+1)

Nur zum zählen nutzen:

liste = [3, 6, 123, 54, 927]

for \_ in len(liste):

print (Hier könnte ich jetzt genau 5 mal was machen)

**break und continue und else**

* Continue: Bricht den Schleifendurchgang ab und kehrt zu Bedingung für eine erneute Überprüfung zurück.
* Break: Bricht die gesamte Schleife ab und macht hinter der Schleife weiter.  
  Man kann einer for-Schleife auch ein else dranhängen, wenn die Schleife nie mit einem break abgebrochen wurde und korrekt zu ende läuft wird am Ende der else Block ausgeführt.
* Else: Beide Schleifen können einen else-Block haben. Sie wird genau einmal am Ende Durchlaufen wenn die Schleife nicht mit Break vorzeitig beendet wurde.

# Funktionen

Parameter = Der Name der Variablen; quasi was in der () von der definierten FKT steht  
Argument = Die tatsächlichen Daten, quasi das was in der () steht während man die FKT aufruft

**Standardparameter:**

|  |  |
| --- | --- |
| def hello(number = 3, word = "Hallo"):      for i in range(0,number):          print(word)  hello() #1  hello(5) #2  hello(word="World") #3  hello(number=5, word="World") #4  hello(2,"Universe") #5 | Mit Aufruf 3 kann man ein Argument einem Übergabewert auch zuordnen  -> Keyword Arg vs Positional Arg  Wenn man Übergabewerte verbindlich machen möchte darf man kein Defaultwert eingeben  Es können alle Arten von Datentypen und -strukturen übergeben werden |

**Exkurs: Funktionsparameter primitiv vs. Referenz:**

* Wenn ich einer Funktion eine Variable als Parameter übergebe und diesen in der Funktion ändere, dann ändere ich die KOPIE nicht aber den Inhalt der original Variable!
* Wenn ich einer Funktion ein Objekt/Datenstruktur übergebe und diese in der Funktion ändere, dann ändert sich AUCH das Original, da ich in die Funktion „nur“ eine Referenz übergebe.
* Wenn ich jedoch in der Funktion dem übergebenen Parameter (der Referenz zum Objekt/Datenstruktur) ein neues Objekt/Datenstruktur zuweise – also eine neue Referenz – dann habe ich dem übergebenen Parameter eine neue Referenz auf ein anderes/neues Objekt/Datenstruktur zugewiesen und ändere von nun an NICHT mehr das originale Objekt!

**Variable positional Funktionsparameter:**

|  |  |  |
| --- | --- | --- |
| liste = [1, 2, 3]  def funktion(a, b, c):      print(a, b, c)  funktion(\*liste)  Mit \*liste übergebe ich alle Elemente der liste ohne diese einzeln angeben zu müssen. | def funktion(\*args):      for i in params:          print(i)  funktion(1, 3, 5)  Auf diese Weise kann ich der Funktion beliebig viele Parameter übergeben. | def funktion(a, \*args):      print(a)      for i in params:          print(i)  funktion(1, 3, 5)  Ich kann auch mischen, sodass in diesem Fall mindestens ein Parameter angeben muss. |
| Merke: die Variablen Parameter werden hier immer in Form eines Tupels übergeben! D.h. in der Funktion muss ich auf ein Tupple voller Argumente zugreifen. | | |

**Variable keyword Funktionsparameter:**

|  |  |
| --- | --- |
| def funktion(\*\*kwargs):      print(args)    funktion(k1="v1", k2="v2")  Ausgabe:    Ich kann so beliebige Parameter in Form eines Dictionary übergeben | def calculate(n, \*\*kwargs):      add = n + kwargs["add"]      mult = n \* kwargs["mult"]  calculate(5, add=5, mult=3)  Ausgabe wäre: 10 und 15  kwargs in der Funktion ist immer ein Dict |
| Merke: die Variablen Parameter werden hier immer in Form eines Dictionarys übergeben! D.h. in der Funktion muss ich auf ein Dictionary voller Argumente zugreifen. | |
| **WICHTIG:** z.B. in einer Klasse wenn ich mit \*\*kwargs ein Objekt initialisiere, dann auf die \*\*kwargs argumente nicht mit self.args1 = kwargs[“arg1“] sonder mit self.args1 = kwargs.get(“arg1“) zugrei-fen. Sonst gibt es eine Exception wenn ich nicht alle möglichen optionalen Parameter übergebe | |

|  |
| --- |
| def g(key, par2):      print(key, par2)  d = {"key": "v1", "par2": "v2"}  g(key=d["key"], par2=d["par2"])  g(\*\*d)  # BESSER  Ausgabe: |
| Es geht auch andersrum, ich kann die Elemente einer beliebig langen Dictionary als als variable Keyword Argumente übergeben. Diese werden „Entschlüsselt“ an die Funktion weitergeleitet. |

Cooles Beispiel:

|  |  |
| --- | --- |
| Alt:  import matplotlib.pyplot as plt  def zeichne(farbe):      plt.plot([1,2,3], [5,6,5], color=farbe)      plt.show()  zeichne("r")  Neu:  import matplotlib.pyplot as plt  def zeichne(\*\*plot\_params):      plt.plot([1,2,3], [5,6,5], \*\*plot\_params)      plt.show()  zeichne(color="r", linewidth=10) | **Neu:**  plot\_params wird in Funktionsparameter konvertiert und der Funktion plot übergeben.  Jetzt kann ich ohne alle Parameter in der Funktion definiere und an die plot() weitergeben zu müssen einfach alle Parameter währende des Funktionsaufrufs angeben. |

**Funktion in eigene Datei auslagern:**

Variante 1: Hauptprogramm und Funktionsdatei im selben Ordner

|  |
| --- |
| Datei: modul1.py  def ausgabe:  print(„Hello World“)  Hauptprgramm:  import modul1  modul1.ausgabe() |

Variante 2a: Hauptprogramm und Funktionsdatei in unterschiedlichen Ordner - Langversion

|  |
| --- |
| Datei: modul1.py liegt im Unterordner module  def ausgabe:  print(„Hello World“)  Hauptprgramm:  import module.modul1  module.modul1.ausgabe() |

Variante 2b: Hauptprogramm und Funktionsdatei in unterschiedlichen Ordner – Kurzversion

|  |
| --- |
| Datei: modul1.py liegt im Unterordner module  def ausgabe:  print(„Hello World“)  Hauptprgramm:  import module.modul1  ausgabe= module.modul1.ausgabe  ausgabe()   * Mit der Zeile unter dem Import weise ich der Funktion selber den Pfad mit original Funktion zu, sodass die Funktion jetzt immer nur noch ohne Pfad aufrufbar ist |

**Built-In Funktionenen**

<https://docs.python.org/3/library/functions.html>

# Standardbibliotheke

1. Built-In 🡪 Einfach direkt verwendbar
2. Module 🡪 Müssen importiert werden

Die ersten viel Kapitel beschreiben die Built-In, alle weiteren die „externen“ Module:  
<https://docs.python.org/3/library/index.html>

## Datetime

|  |  |
| --- | --- |
| from datetime import datetime  now = datetime.now()  print(now.strftime("%d.%m.%y"))  **Ausgabe:** 13.12.20  Gibt auch:  from datetime import date, time  Oder:  from datetime import timedelta | Methode: strftime() zum Formatieren der Datumsangabe  <https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes>  Nur als Beispiel, im Kurs kam mehr, aber sehr allgemein. Bei Bedarf nochmal nachsehen. |

## String

# Verschiedenes

"String?".upper()

"String?".lower()

"String?".endswith("?")

"String?".startswith("S")

# links und rechts alle Leerzeichen und . entfernen

"    Hallo.  ".strip(" .")

"    Hallo.  ".lstrip(" .") # nur links

"    Hallo.  ".rstrip(" .") # nur rechts

# Index des Kommas

"Hallo, World.".find(",")

# Komma mit leerzeichen ersetzten

"Hallo, World.".replace(",", " ")

------------------------------------------------------

# Strings Formatieren 2 Varianten

"Ich habe {0} {1} {0}".format(5, "Hunde")

"Ich habe {number} {animal} {number}".format(number = 5, animal = "Hunde")

# der 0er kriegt den Datentyp float mit 3 Nachkommastellen

"Pi hat den Wert: {0:.3f}".format(3.141529)

------------------------------------------------------

"Hallo, World."[0]

"Hallo, World."[:3]

"Hallo, World."[:]

"Hallo, World."[0:4]

“mein name ist Yagmur“.title() -> Mein Name Ist Yagmur

**Arten der Ausgabe bzw. String-Verkettung:**

|  |
| --- |
| print("Das ist “ + eins + str(zwei))   * eins = “eine“; zwei = 5 |
| print("Der Logarithmus von %d zur Basis %d ist %f."%(var1, var2, var3))   * %s string / %d int / %f float |
| print("Eins: ", eins, "Zwei: ", zwei, "Drei: ", drei)   * eins = “eins“; zwei = 2; drei = True |
| print(f“Das ist ein f String mit einer Variablen {test} Ausgabe darin“)   * test = 5 |

**Weitere**

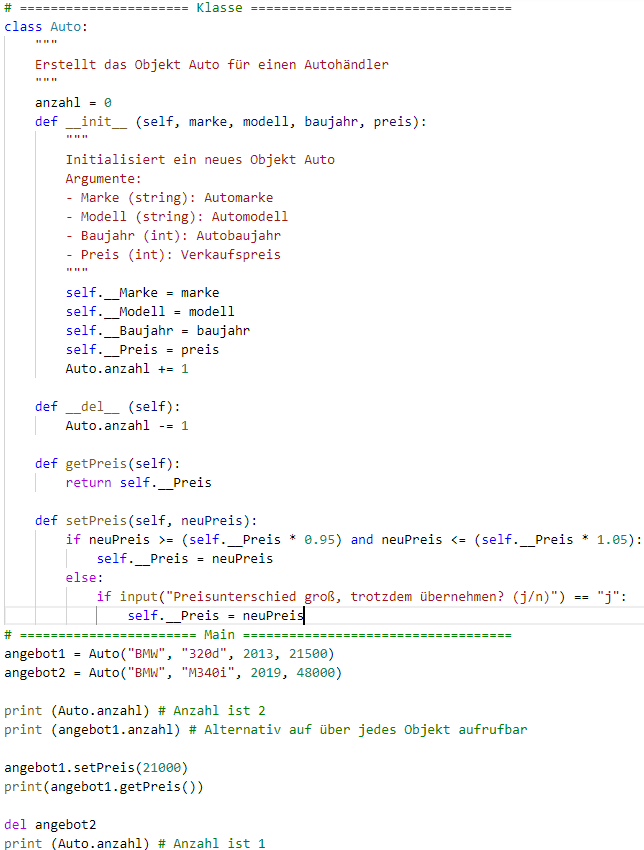
* .lower()
* .count()

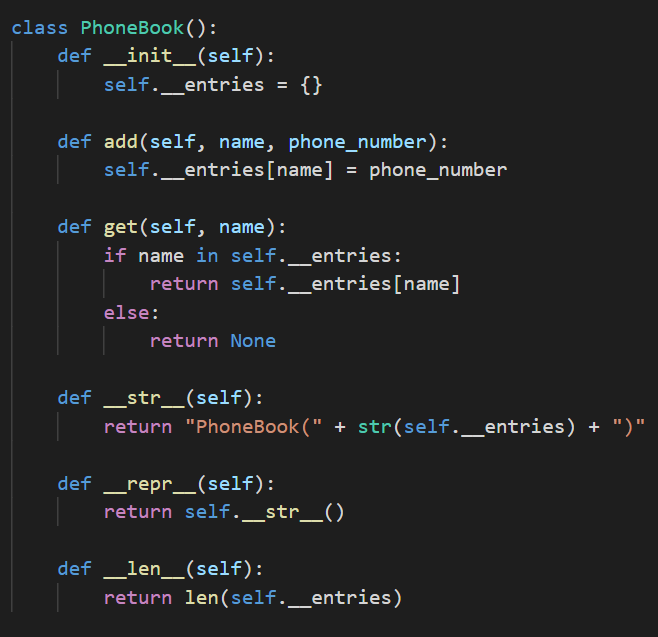
## Rand

* Import random  
  random.randint(a,b)
* random.choice(liste) 🡪 Anstatt Zufallszahl im Wertebereich der Liste

# Objektorientierung

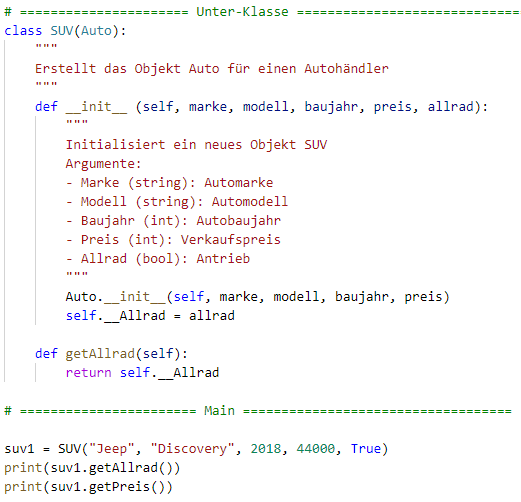
**Klasse / Instanz / Datenkapselung / Methoden (Getter/Setter) / Klassen vs. Objektvariable:**

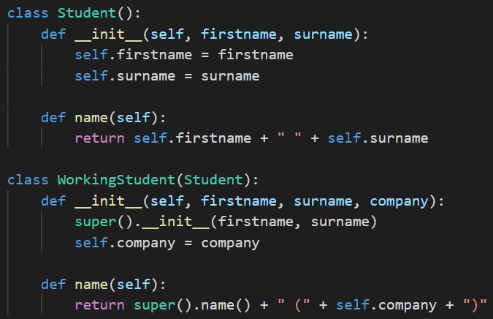
* “““ 🡪 DOCSTRING
* Immer erst self als Parameter angeben, damit sagt man das es für das konkrete Objekt der Klasse gilt
* Entsprechend bei der Variablen-Initialisierung auch immer ein self voranstellen.
* Mit \_\_ (private) vor der Variablen Initialisierung kapsele ich die Daten, sodass sie von außen nicht direkt erreichbar sind
* Bei Objektinitialisierung muss für self kein Argument übergeben werden, das passiert automatisch
* Im Setter: Bei Preisunterschieden über 5% wird nochmal nachgefragt
* Klassenvariable: Gilt für alle Instanzen der Klasse gleichermaßen (für alle konkreten Objekte). Zugriff über Klassenname oder beliebige Objektname.

Spezielle Methoden:

1. def \_\_init\_\_ 🡪 Konstruktor
2. def \_\_del\_\_ 🡪 Destruktor
3. def \_\_str\_\_ 🡪 Wenn ich das Objekt in einem print ausgebe wird der return dieser Methode ausgegeben, anstatt die Speicher Adresse des Objekts
4. def \_\_repr\_\_ 🡪 Wenn ich das Objekt direkt angebe ohne in ein print() zu packen wird diese Methode ausgeführt und der return ausgegeben.
5. def \_\_len\_\_ 🡪 Wenn ich len(Objekt) in einem print ausgebe, kann ich hier die definieren was ausgegeben werden soll (überschreibe quasi die len() Funktion für mein Objekt
6. Es gibt noch einige \_\_...\_\_ Methoden mehr die ich auf diese Weise überschreiben kann.

**Vererbung:**

* ****Diese Klasse SUV erbt von der Klasse Auto (siehe oben)
* Identischer Konstruktor wie Oberklasse, nur mit zusätzlichen   
   Parametern (allrad).
* Konstruktor der Oberklasse wird in diesem Konstruktor   
   aufgerufen und die identischen Parameter weitergegeben.  
   🡪 Anstatt Auto.\_\_init\_\_(…) geht auch super().\_\_init\_\_()
* Die zusätzliche Variable wird wie gehabt hier initialisiert
* Zusätzlich noch ein Getter für die neue privaten   
   Objektvariable und fertig

Wenn ich in Kindklasse eine Methode habe die gleich heißt   
wie in der Elternklasse, dann überschreibe ich die Methode   
aus der Elternklasse in der Sub-Klasse. (Bei gleicher   
Paramteranzahl). Wenn ich also Subklasse Methode aufrufe,   
dann wird auch die Methode aus der Subklasse ausgeführt   
und nicht mehr die vererbte Methode aus der Elternklasse.   
Aber auch hier kann ich mit super() die vererbte Methode aus   
Elternklasse aufrufen und quasi ergänzen/ändern.

**In Python ist alles ein Objekt:**

|  |  |
| --- | --- |
|  | a ist eigentlich ein Objekt der Klasse float |
|  | 3 Varianten für den absolut selben Code:   1. Kurzschreibeise 2. Was eigentlich passiert Long!!! 3. Was eigentlich passiert Short!!! |
|  | Ein Beispiel mit string :   1. Ein String ist auch ein Objekt der Klasse str 2. Kurzschreibeise vs Was eigentlich passiert!!! 3. Hier sieht man das man die spezielle Methode auch überschreiben kann, bzw. das man in den Prozess eingreifen kann! |

# Exceptions

* Syntaxfehler – Interpreter erkennt die sofort, statisch.
* Laufzeitfehler – Tritt während der Laufzeit dynamisch auf.
* Wenn ein Laufzeitfehler auftritt, wird das Programm vollständig abgebrochen:
  + Exceptions definieren, damit das Programm weiterlaufen kann
  + Besonders anfällige Programmteile darin verpacken, um darauf reagieren zu können

**Eine Exception**

try:

    print(4 / input("Zahl"))

except ZeroDivisionError as error\_message:

    print(f"Der {error\_message} ist aufgetreten")

* Im except kann/sollte man die Art des Fehlers angeben, welchen man abfangen möchte
* Wenn ein Laufzeitfehler vorkommt, steht die Art des Fehlers in der Fehlermeldung
* Alle Arten von Exceptions sind auch hier: [Built-in Exceptions — Python 3.9.1 documentation](https://docs.python.org/3/library/exceptions.html)

**Mehrere Exceptions und else**

try:

    print(4 / int(input("Zahl")))

except (ZeroDivisionError, ValueError):

    print("Teilen durch 0 nicht erlaubt ODER Nur ganze zahlen eingeben")

===================================================================================

try:

    print(4 / int(input("Zahl")))

except ZeroDivisionError:

    print("Teilen durch 0 nicht erlaubt")

except ValueError:

    print("Nur ganze Zahlen erlaubt")

except:

    print("Sonstige Exception ", sys.exc\_info()[0])

else:

    print("Try Block ohne Fehler durchgelaufen")

finally:

    # z.b. Objekt zerstören oder eine Datei wieder schließen

* Man kann auch mehrere Exceptions gebündelt abfangen (oben).
* Oder separat:
  + Ein Block pro Exception um verschieden reagieren zu können
  + Optional ein Block ohne definierte Exception (MUSS unter allen anderen spezifischen Exceptions stehen)  
    Hier laufen dann alle Laufzeitfehler rein, die keinen eigenen Block darüber hatten!  
    WICHTIG: Mit „sys.exc\_info()[0])“ zumindest den aufgetretenen Laufzeitfehler anzeigen damit es nicht untergeht
  + Optional ein Block mit else der immer dann ausgeführt wird, wenn im Try kein Fehler kam 🡪 Also entweder eine Exception ODER die else
  + Optional ein Block mit finally, der IMMER ausgeführt wird, egal ob eine Exception ausgeworfen wurde oder nicht (z.B. hier ein im try geöffnete Datei schließen)  
    MERKE: wenn im try eine Exception auftritt die überhaupt nicht abgefangen wurde bricht das Programm ab, ABER auch dann wird zuerst der finally Block ausgeführt.

**Eigene Exceptions**

* Ausnahmen sind in der Klasse Exception definiert
* Für eigene Ausnahmen diese Klasse erweitern
* Konvention: den Namen der benutzerdefinierten Exception mit …Error abschließen
* Selbst definierte Ausnahmen lösen nicht automatisch aus; mit raise selber auslösen
* Mit raise kann ich auch schon existierende Exceptions manuell auslösen  
  raise TypeError(“Eine Message die ich mit ausgeben möchte“)

class testError(Exception):

    def \_\_init\_\_(self, wert):

        self.wert = wert

try:

    x = input("Zahl")

    if x < 10 or x > 20:

        raise testError(x)

except testError as fehler:

    print(str(fehler.wert) + "muss zwischen 10 und 20 liegen")

**Info:**

* Vorteil ist auch, wenn ich in einem try-block Funktionen aufrufe (und sogar mehrfach verschachtelte Funktionen) wo irgendwo innerhalb dieser Funktionen eine Exception geworfen wird, wird dieser bis zu meinem try Block hochgereicht, wo ich es abfangen kann.
* Exceptions selten werfen, sondern mit normalem Python Code Sachen prüfen/sicherstellen. Exceptions eher bei kritischen Sachen, die sehr selten auftreten verwenden

# Dateien/Datenspeicherung

**Datei lesen/ schreiben:**

* Zeichensatz mit angeben: encode=“UTF-8“ als Paramter in open()
* ?! Pandas nutzen wo immer möglich und sinnvoll

autos = ["BMW", "Audi", "Mercedes", "VW", "Opel"]

with open("C:/Users/guemu/OneDrive/Desktop/Python/Udemy/Ubung/schreiben.csv", "w") as file:

    for i in autos:

        file.write(i + ";") # ; = In Excel jeweils eigenen Zelle

with open("C:/Users/guemu/OneDrive/Desktop/Python/Udemy/Ubung/schreiben.csv", "r") as file:

    for line in file:

        print(line.strip().split(";")) #aus dem ganzen string wird hier eine liste

Das with-Konstrukt oben entspricht grob dem open/close mit exception unten. Daher ist with besser, weil es kürzer ist und immer selber dafür sorgt das die Datei geschlossen wird:

try:

    file = open("datei.txt", "r")

except FileNotFoundError:

    print("Datei wurde nicht gefunden")

finally:

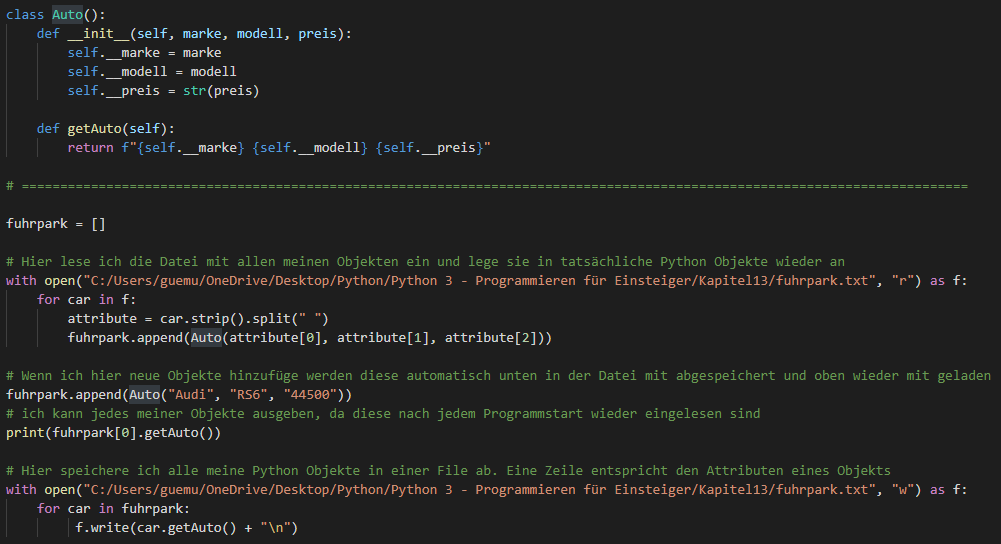
    file.close()

Verschiedene Modi zum öffnen einer Datei möglich, bei Bedarf nachsehen.

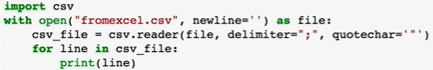
**Methoden des Dateiobjekts:**

* Gesamten Inhalt der Datei auslesen: inhalt = file.read()
* Zeilenweise manuell mit while-Schleife: readline()
* Datei pro Zeile in eine Liste auslesen: inhalt = file.readlines()
* Und einige mehr

**Beispiel – Objekte in Datei speichern und laden:**



**BEACHTE:** Einlesen einer CSV Datei würde man in der Praxis mit dem CSV Modul aus Python machen



# NumPy

Immer wenn ich einen Topf voller Zahlen habe mit denen ich arbeiten muss, macht es Sinn diese als array in Numpy zu führen und weiterzuverarbeiten.

## Basics

|  |  |
| --- | --- |
| xs = np.arange(10)  ys = xs \*\* 2  oder  xs = np.arange(10) \*\*2 | Ein Array mit Zahlen von 0 bis 9 erstellen und mit sich selbst potenzieren.  Quasi die range() aus Python |
| np.array([1, 2, 3, 4, 5, 10]) | Damit erzeuge ich aus einer Liste ein Array |
| np.zeros(10)  np.ones(10) | Ein Array mit 10 nullen bzw. einsen erstellen |
| print(np.array([1, 2, 3, 4, 5, 10]).mean())  print(np.array([1, 2, 3, 4, 5, 10]).max()) | Vorgefertigte Funktionen auf das array anwenden. |

## Filtern eindimensionaler Arrays

|  |  |
| --- | --- |
| a = np.array([1, 2, 3, 4])  b = np.array([False, True, True, False])  a[b] | Filtere mein Array a entsprechend den Werten aus dem Array b |
| a = np.array([1, 2, 3, 4])  a[a >= 3] | Der Ausdruck a>=3 führt auch zu einem Array voller booleans wie oben |
|  |  |
|  |  |

## Mehrdimensionale Arrays

|  |  |
| --- | --- |
| a = np.array([1, 2, 3, 4, 5, 6, 7, 8])  b = np.array([[1, 2, 3, 4], [5, 6, 7, 8]]) | Eindimesnional vs Zweidimensional |
| b.shape() | Gibt ein Tupel aus (Zeilen, Spalten) |
| a.reshape((2, 4)) | Macht aus a 2 Zeilen 4 Spalten draus  Es müssen genau 8 Elemente in a sein |
| a.reshape((-1, 4))  a.reshape((4, -1)) | Zeilen werden automatisch gesetzt  Spalten werden automatisch gesetzt |
| b = np.array([[1, 2, 3], [4, 5, 6]])  b.reshape(-1) | b wird wieder zusammengeführt |

## Exkurs: Wie Numpy intern Abläuft

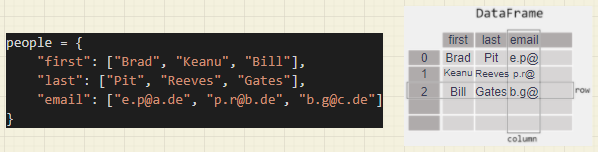
Funktionen wie z.B. xs \*\* 2 welche wir geradeaus in Python bzw. NumPy schreiben sind keine Magie sondern vorimplementiert (gilt z.B. auch für das Zugreifen eines index xs[0] usw.)

|  |
| --- |
| class MyArray():        # im Constructor übergeben wir lediglich die Liste      def \_\_init\_\_(self, liste):          self.liste = liste        # die Multiplikationsfunktion überschreiben wir hier derart, dass sie für jedes   Element aus der Liste aufgerufen wird      def \_\_mul\_\_(self, other):          nliste = []          for element in self.liste:              nliste.append(element \* other)          # oder alternativ: nliste = [element \* other for element in self.liste]          return MyArray(nliste)    a = MyArray([1, 2, 3])  b = a \* 2  print(a.liste) # [1, 2, 3]  print(b.liste) # [2, 4, 6] |

# Pandas

* Öffnen/Laden einer CSV Datei **bzw**. DataFrame Object: DataFrame()

|  |
| --- |
| df = pd.read\_csv(“pfad.csv“, delimiter=“,“) |
| df = pd.DataFrame(python\_dict) |

df ist nun ein: <class 'pandas.core.frame.DataFrame'> was quasi einer Tabelle entspricht  
 

* DataFrame Attribut: columns

|  |
| --- |
| df.columns |

Gibt mir alle Spaltennamen aus meinem DataFrame aus

* DataFrame Attribut: index

|  |
| --- |
| df.index |

Gibt mir den Index der Tabelle aus. Bei Default 0,1,2,… oder aber der selber gesetzte

* DataFrame Attribut: shape

|  |
| --- |
| df.shape |

Gibt ein Tupel aus mit Zeilen und Spalten in dem Dataframe

* DataFrame Method: info()

|  |
| --- |
| df.info() |

Gibt eine Übersicht über den DataFrame zurück/aus

* DataFrame Method: set\_option()

|  |
| --- |
| pd.set\_option(„display.max\_columns“, 85)  pd.set\_option(„display.max\_rows“, 85) |

Hiermit stelle ich ein das alle meine Spalten und Zeilen angezeigt werden sollen

* DataFrame Method: head() und tail()

|  |
| --- |
| df.head()  df.tail() |

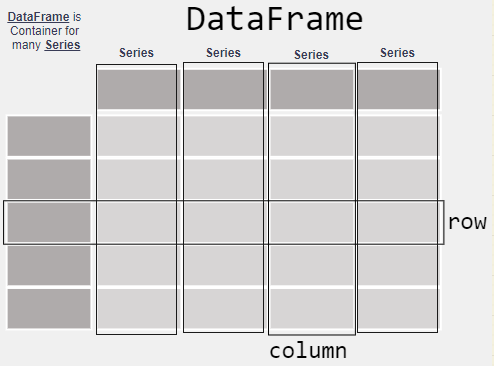
* Gibt nur die ersten/letzten 5 Zeilen meines DataFrames (Tabelle) aus
* In die Klammer kann ich ein int eintragen für die Anzahl der x Zeilen
* DataFrame Method: sort()

|  |
| --- |
| df2 = df.sort\_values(“Name“, ascending=False) |

Sortieren absteigend nach dem Name

* asd

Exkurs:



Wenn man eine Spalte (Series) Selektiert und ausgibt, hat auch dieser als erste Spalte ein Index.

|  |
| --- |
| Python  (siehe 2. für dict) |
| people[“email“] |
| Hier liefert Python eine List zurück |

* Eine Spalte Abfragen:

|  |
| --- |
| df[“email“] (Alternativ: df.email) |

* Hier wird eine Series zurück gelifert, was einer Liste der Daten aus der Spalte entspricht: <class 'pandas.core.series.Series'>
* **Genauer:** ROWS of a single COLUMN
* Erste Variante bevorzugen
  + Spalte könnte ein String mit Leerzeichen sein
  + Auf dem Objekt könnte es ein gleichnamiges Attribut oder Methode geben.
* Mehrere Spalten Abfragen:

|  |  |  |
| --- | --- | --- |
| Über Zugriff auf Spalte | Iloc[] | loc[] |
| df[[“last“, “email“] ] | df.loc[:,1:3] | df.loc[:,“last“:“email“] |
| Einfach aufzählen der Spalten welche ausgegeben werden sollen | Über alle Zeilen hinweg die Spalten von **index** 1 bis 2 (exkl. 3) | Über alle Zeilen hinweg die Spalten von **label** last bis email (inkl. email) |
| In diesem Fall kriege ich kein Series Objekt mehr sondern wieder ein DataFrame Objekt, da es wieder mehr als nur eine Spalte ist | | |

* Eine Zeile Abfragen:

|  |  |
| --- | --- |
| iloc[] | loc[] |
| df.iloc[0] | df.loc[0] |
| Gibt die erste Zeile (**index** 0) in Form eines Series Objekts aus (also als eine Spalte) | Gibt die erste Zeile (**Label** 0) in Form eines Series Objekts aus (also als eine Spalte) |

* Mehrere Zeilen Abfragen:

|  |  |  |  |
| --- | --- | --- | --- |
| iloc[] | | loc[] | |
| df.iloc[[0, 1]] | df.iloc[0:2] | df.loc[[0,1]] | df.loc[0:1] |
| Explizit die Zeile 0 und die Zeile 1 | Die Zeilen 0 bis 1  Exkl.dem Stop | Explizit die Zeile 0 und die Zeile 1 | Die Zeilen 0 bis 1  Inkl. dem Stop |
| Hier sind die Paramter indexes welche wie gewohnt über int angesprochen werden | | Hier sind die Paramter Labels/Strings auch wenn 0,1,usw steht | |
| In diesem Fall kriege ich wieder kein Series Objekt mehr sondern wieder ein DataFrame Objekt, da es wieder mehr als nur eine Zeile ist die ich kriege. | | | |

* Zelle Abfragen (eine):

|  |  |  |  |
| --- | --- | --- | --- |
| iloc[] | | loc[] | |
| df.iloc[0, 0] | df.iloc[[0], [0]] | df.loc[0, "first"] | df.loc[[0], ["first"]] |
| Ausgabe: str | Ausgabe: 1x1 Tabelle  DataFrame | Ausgabe: str | Ausgabe: 1x1 Tabelle  DataFrame |
| Mit diesen Varianten kann ich nun auch für die Zeilen/Spalten beliebig viele Angaben machen. Das Ergebnis ist stets eine entsprechende Zieltabelle (DataFrame) | | | |

Zellen Abfragen (mehrere):

**Für iloc**[] **🡪 Immer index bezogen (0 – len()-1) und exkl. Stopp:**

|  |  |  |  |
| --- | --- | --- | --- |
| Df.iloc[[0, 1], 2] | df.iloc[0:2, 2] | df.iloc[[0, 1], [0, 1]] | df.iloc[0:2, 0:2] |
| Für die Zeilen 0 **und** 1 die 3. Spalte ausgeben | Für die Zeilen 0 **bis** 1 die 3. Spalte ausgeben | Für die Zeilen 0 **und** 1 die Spalten 0 **und** 1 ausgeben | Für die Zeilen 0 **bis** 1 die Spalten 0 **bis** 1 ausgeben |

**Für loc**[] **🡪 Immer label bezogen und inkl. Stopp:**

|  |  |  |  |
| --- | --- | --- | --- |
| df.loc[0:2, "first"] | df.loc[[0, 1, 2], "first"] | df.loc[[0,1],[“strA“, “strB“]] | df.loc[0:1, „strA“:“strB“] |
| Für die Zeilen 0 **bis** 2 (3 Zeilen) die Spalte „first“ ausgeben | Für die Zeilen 0 **und** 1 **und** 2 (3 Zeilen) die Spalte „first“ ausgeben | Für Zeilenbezeich-nungen „0“ **und** „1“ die Spalten mit der Bezeichnung „strA“ **und** „strB“ | Für Zeilenbezeich-nungen „0“ **bis** „1“ die Spalten von Bezeichnung „strA“ **bis** „strB“ |

Das geht auch, glaub ist aber eher Python Style und weniger Pandas Style:

|  |
| --- |
| df.iloc[0]["first"] oder df.iloc[0][0] |

Von der ersten Zeile gehe ich hier in die Spalte „first“ bzw. den Spaltenindex [0] (Wobei ich hier ja die Zeile in Form einer Serie (Spalte) bekomme und der index 0 eigentlich wieder eine Zeile bedeutet. **Das Ergebnis ist KEIN DataFrame, sondern der Datentyp des Elements in der Zelle!!!!**

* Eigenen Index setzen / reseten:

|  |
| --- |
| df.set\_index(“email“, inplace=True)  df.reset\_index(inplace=True) |

* Damit setze ich die Spalte Email als Index (erste Spalte) für die Tabelle.
* Ohne den inplace Paramter mache ich das nur temporär, mit dem Paramter persistent
* Nun kann ich mit der loc() auch die Zeilen aussagekräftiger selektieren in dem ich keinen index für die Zeile angeben muss sondern eben z.b. einfach nur die email adresse der Person
* Nach Index sortieren:

|  |
| --- |
| df.sort\_index(ascending=False, inplace=True) |

Wenn ich ein Index aussuche welches z.B. aus Wörter besteht würde ich hier auch Alphabetisch sortieren können. Oder Eben wenn zahlen dann auf/absteigend usw.

* Zeilen filtern 1 – Eine Bedingung:

|  |  |
| --- | --- |
| filt = df["Age"] == 18 | Hier nehme ich eine Spalte und selektiere nach einem Wert dieser Spalte. Resultat ist <class 'pandas.core.series.Series'> mit lauter True und False für die Zeilen, wo es zutrifft und wo nicht. |
| df.loc[filt, ["Country", "Age"]] | Hier zeige ich nun die Spalten Country und Age nur für die zutreffenden (im filter True) Zeilen. |
| Oder auch | |
| filt = df["Age"] == 18  df[filt]  bzw.  df[df["Age"] == 18] | df["Age"] == 18 🡪 Hier kriege ich als Output für jede Zeile in meiner Tabelle ein True oder False für die Age == 18  wenn ich diese Liste voller boolean in mein df reinschiebe wird mir meine Tabelle auch gefiltert |

* Zeilen filtern 2 – Zwei Bedingungen mit UND verknüpft:

|  |  |
| --- | --- |
| filt = (df["Age"] == 18) & (df["first"] == “John“) | Analog zu 1) nur das ich hier zwei Bedingungen mit UND für die Zeilen verknpüfe |
| df.loc[filt, ["Country", "Age"]] | Hier zeige ich nun die Spalten Country und Age nur für die zutreffenden (im filter True) Zeilen. |

* Zeile filtern 3 – Zwei Bedingungen mit ODER verknüpft:

|  |  |
| --- | --- |
| filt = (df["Age"] == 18) | (df["first"] == “John“) | Analog zu 1) nur das ich hier zwei Bedingungen mit ODER für die Zeilen verknpüfe |
| df.loc[filt, ["Country", "Age"]] | Hier zeige ich nun die Spalten Country und Age nur für die zutreffenden (im filter True) Zeilen. |

* Zeilen filtern 4 – Negieren der Bedingung:

|  |  |
| --- | --- |
| filt = df["Age"] == 18 | Hier nehme ich eine Spalte und selektiere nach einem Wert dieser Spalte. Resultat ist <class 'pandas.core.series.Series'> mit lauter True und False für die Zeilen, wo es zutrifft und wo nicht. |
| df.loc[~filt, ["Country", "Age"]] | Mit ~ negiere ich den Filter, gebe also alle außer 18 aus |

* Durch ein DataFrame iterrieren:

|  |
| --- |
| for index, row in df.iterrows():  if row[“student“] == “Yagmur“:  print(row[“score“]) |
| row ist hier nun ein Pandas Series Objekt, in for kann ich dann durch eingrezen der Spalte auf eine entsprechende Zelle zugreifen |

* asd

# Laufzeitverhalten

Instanz: Konkretes Datenobjekt im Speicher entsprechend dem Bauplan (Datentyp)  
Referenz: Die Adresse zur eindeutigen Instanz (Identität) im Speicher

**Immutable Datentypen (z.B. str):**

|  |  |
| --- | --- |
|  | a enthält die Adresse zur Instanz von str in dem der Wert “Wasser“ gespeichert ist. Diese Instanz hat eine eindeutige Identität |
| Wenn ich eine zweite Variable erstelle welche dieselbe Adresse auf dieselbe Instanz, habe ich immer noch EIN Objekt im Speicher! |
| Da der Datentyp str unveränderlich ist, wird bei dieser Zuweisung eine neue Instanz erstellt und die Referenzen entsprechend angepasst. |

**Mutable Datentypen (z.B. lst):**

|  |  |
| --- | --- |
|  | a enthält die Adresse zur Instanz einer liste in dem die Werte 1 und 2 gespeichert sind. Diese Instanz hat eine eindeutige Identität |
| Wenn ich eine zweite Variable erstelle welche dieselbe Adresse auf dieselbe Instanz, habe ich immer noch EIN Objekt im Speicher! |
| Da der Datentyp list veränderlich ist, wird bei dieser Zuweisung **keine** neue Instanz erstellt und die Referenzen zeigen immer noch auf die selbe Instanz. |

**Details:**

|  |  |
| --- | --- |
|  | |
| Datentyp:  Bauplan für die Instanz, bestimmt den Wertebereich!  Mit type() lässt sich der Datentyp einer Instanz bestimmen. |  |
| Wert:  Mit == lassen sich die Werte verschiedener Instanzen vergleichen.  Ob ein Vergleich zwischen verschiedenen Datentypen definiert ist, hängt von den Datentypen selbst ab. Wenn nicht werden stattdessen die Identitäten verglichen. |  |
| Identität:  Programmweit eindeutiger Bezeichner für die Instanz.  Mit id() lässt sich die Identität einer Instanz ermitteln.  id(ref1) == id(ref2) ist dann True wenn beide Referenzen auf das gleiche Objekt zeigen. Kurz: ref1 is ref2 |  |

**Garbage Collector:**

Instanzen (Identitäten) werden automatisch gelöscht, wenn keine Referenzen mehr auf diese existiert (interner Referenzzähler für die Instanz == 0). Referenzen können mit der del-Anweisung gelöscht werden.

# Requests

1. Attributs and Methods from any Object: dir(object)   
   z.B. print(dir(r)) wobei r = requests.get(…)   
   Anstatt dir() kann ich auch help() verwenden  Für genauere Infos
2. Im request ein z.B. timeout=5 einbauen.   
   Damit ich ein ewiges warten auf die Response verhindere.   
   Hier: Wenn es länger als 5 Sekunden dauert wird ne Exception geschmissen