

Cheatsheet Python

Darian Patzak, Anton Obersteiner

November 18, 2022

Console (bash/Powershell/...)

Variablen und Datentypen

if statements

Loops

try / catch

Dictionaries

Git

Funktionen

mit Dateien arbeiten

Klassen und Objekte

Bibliotheken

Navigation

```
ls #Inhalt des aktuellen Verzeichnisses  
pwd #aktueller Ordner/Pfad  
#Wechsel in anderen Ordner:  
cd .. #übergeordneter Ordner  
#Unterordner des aktuellen Verzeichnisses:  
cd myfolder\mysubfolder #Windows  
cd myfolder/mysubfolder #Mac u. Linux
```

Py-Datei in Console ausführen

```
python3 myprogram.py #myprogramm.py ausführen  
Ctrl-C #bricht laufendes Program ab  
python3 #öffnet eigene Python-Console  
#quit() oder exit() eingben um sie zu beenden.
```

Augabe auf der Console

```
print("Hello World") #-> "Hello World"  
a = 5  
print("a ist:", a) #-> "a ist: 5"  
print(f"a ist: {a}") #-> "a ist: 5"
```

Input von der Console lesen

```
myinput = input("Bitte Wert eingeben: ")  
#gibt string zurück.  
#Falls Zahl gewünscht:  
#zu int oder float konvertieren
```

Wertzuweisung

```
myvalue = 3    #die Variable wird auf 3 gesetzt  
myvalue = myvalue + 2    #myvalue um 2 erhöht  
#geht auch mit anderen Rechenoperationen  
myvalue += 2    #Abkürzung von vorheriger Zeile
```

Datentypen

```
#Standard
int    #Integer (Ganzzahlen)
float  #Floatingpoint (Kommazahlen)
str    #String (Zeichenketten / Text)
#"text" oder 'text' ist egal

bool   #Wahrheitswert (True/False bzw. Ja/Nein)
list   #Listen von irgendwelchen Werten
#z.B. ["Hanna", 20, "Kaiserschmarrn"]
```


Advanced

```
object #Klassen (Komplexe Datenstruktur)  
tuple #n-Tuple, nicht veränderbar  
dict #weist Werten andere Werte zu
```

Listen-Operationen

```
my_list = [] #leere Liste erstellen
my_list = [1, 2, 3] #Liste erstellen
my_list.append(4) #hängt 4 an Liste an
#-> my_list ist jetzt [1, 2, 3, 4]
my_list.remove(2) #entfernt die erste 2
#-> [1, 3, 4]
```

Komplexere Operationen

```
my_list.reverse() #kehrt Reihenfolge um  
#-> [4, 3, 1]  
my_list.sort() #sortiert Liste -> [1, 3, 4]  
sorted(my_list)  
#gibt sortierte Kopie der Liste zurück  
#Originalliste bleibt unsortiert
```

Werte in Listen

```
my_list = [10, 20, 31, 40, 50] #Beispielliste  
my_list[0] #Zugriff auf erstes Element -> 10  
my_list[2] = 30 #setzt Element an Index 2  
#-> [10, 20, 30, 40, 50]  
my_list[-1] #letztes Element der Liste  
#-> 50
```

Bereiche in Listen

```
my_list[1:4] #neue Liste von 1 bis vor Index 4
#-> [20, 30, 40]
my_list[:4] #alles bis vor Index 4
#-> [10, 20, 30, 40]
my_list[2:] #alles Werten ab Index 2
#-> [30, 40, 50]
[a, b, c] + [1, 2, 3] # +   konkateniert Listen
#-> [a, b, c, 1, 2, 3]
```

Mehrdimensionale Listen

```
#Listen die Listen enthalten
personen = [ ["Anna", 18], ["Ben", 19],
             ["Carla", 20]]
personen[1]          #gibt ["Ben", 19] zurück
personen[1][0]       #gibt "Ben" zurück
personen[2][1] = 21  #setzt Carlas Alter auf 21
```

casten (Datentypen konvertieren)

```
int(5.0) #-> 5
int(5.7) #-> 5
int("5") #-> 5
int("5.0") #-> error
float(5) #-> 5.0
float("5.0") #-> 5.0
str(5.0) #-> "5.0"
```

Vergleichen

```
a == b  #nicht dasselbe wie a = b  
a != b  # $a \neq b$   
a < b  
a >= b  # $a \geq b$ 
```


if-statement mit Vergleich

```
if my_age > your_age:
    print("ich bin älter als du.")
elif my_age < your_age:
    print("du bist älter als ich.")
else:
    print("wir sind gleichalt.")
#elif und else sind optional
```

einfache for-Schleife

```
#range von 0 bis vor 10 (also bis 9)
for current_value in range(0, 10):
    print(current_value)
```

erweiterte for-Schleife

```
names = ["Anna", "Ben", "Carl", "David"]  
for person in names:  
    print(person)  
#output:  
#Anna  
#Ben  
#Carl  
#David
```

erweiterte for-Schleife

```
persons = [ ["Anna", 18], ["Ben", 19],  
            ["Carla", 20] ]  
for person in persons:  
    print(  
        person[0], "ist",  
        person[1], "Jahre alt."  
    )  
#output:  
#Anna ist 18 Jahre alt.  
#Ben ist 19 Jahre alt.  
#Carla ist 20 Jahre alt.
```

while-Schleife

```
#runterzählen
current_value = 10
while current_value > 0:
    current_value = current_value - 1

while True:
    my_input = input("Loop abbrechen?")
    if my_input == "ja":
        break
#break bricht while- oder for-schleife ab
#hier geht es danach weiter
```

gute Fehlermeldung

```
num_tickets = input("Wie viele Tickets: ")
try:
    num_tickets = int(num_tickets)
except ValueError:
    print("bitte valide Ganzzahl eingeben")
```

sehr allgemeine Fehlermeldung

```
num_tickets = input("Wie viele Tickets: ")
try:
    num_tickets = int(num_tickets)
except Exception as e:
    print("ein Fehler ist aufgetreten:", e)
```

wenig hilfreiche Fehlermeldung

```
num_tickets = input("Wie viele Tickets: ")
try:
    num_tickets = int(num_tickets)
except:
    print("das hat nicht geklappt.")
```


dictionaries anlegen

```
# leeres dictionary anlegen
car = {}

# bereits gefülltes dictionary anlegen
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964,
    "previous_owners": ["Anna", "Ben"]
}

# Wert ausgeben lassen
print( car["brand"] ) # -> "Ford"
print( car["previous_owners"] ) # -> ["Anna", "Ben"]

# Wert einfügen oder überschreiben
car["miles"] = 10500 # fügt den key "miles" mit dem
    value 10500 ein
car["year"] = 2000   # überschreibt den value im key "
    year"
```

über dictioanries iterieren

```
for key in car:
    print(key)
# "brand"
# "model"
# "year"
# "previous_owners"

for key in car:
    print( key, car[key] )
# "brand" "Ford"
# "model" "Mustang"
# "year" 1964
# "previous_owners" ["Anna", "Ben"]
for key, value in car.items():
    print( key, value )
# "brand" "Ford"
# "model" "Mustang"
# "year" 1964
# "previous_owners" ["Anna", "Ben"]
```

Repository anlegen

```
git init # legt Repository an
git branch # printet alle branches und markiert den
           aktuellen
# standardmäßig wird der Master oder Main branch
  erstellt. Da kommen nur stabile Stände hin, also
  direkt neuen Arbeitsbranch erstellen
git branch my_branch # erstellt einen neuen Branch mit
  dem namen my_branch
git checkout my_branch # wechselt auf den branch
  my_branch
git checkout -b my_branch # erstellt und wechselt auf
  my_branch in einem Schritt
```

Repository verwalten

```
git status # gibt aktuellen Stands aus (veränderte,  
            gelöschte Dateien)  
git log # printet die Historie des Branches. Mit 'q'  
         kommt man da wieder raus  
git diff # git geänderte/neue Zeilen seit dem letzten  
          commit aus
```

Dateien tracken

```
git add my_file.py # added die Datei 'my_file.py' in den
                    Staging-Bereich
git add . # added alle neuen/veränderten Dateien in den
          Staging-Bereich
git commit -m "aussagekräftige commit-message" #
          erstellt einen Commit mit den Dateien aus dem staging
          Bereich. Gibt eine Commit message mit, die genau
          beschreibt was du verändert hast
```

Mergen

```
git merge anderer_branch # merged den branch '
    anderer_branch' in den branch auf dem ich gerade bin
# bei merge-Konfliketen wird angezeigt in welchen Dateien
    Konflikte sind (falls nicht, dann mal git status
    aufrufen)
# die Dateien mit einem Editor öffnen und manuell den '
    incoming' oder 'current' change akzeptieren. (macht
    sich blöd in der Konsole)
# GOOGELN!! Merge Konflikte sind unschön, deshalb nicht
    hektisch reagieren und alle fehlermeldungen googeln
```

remtote Repositories

```
# Repositories können auch im Internet zb. auf GitHub  
oder GitLab gehosted werden  
git clone url_to_repository # macht euch eine lokale  
Kopie in das aktuelle Verzeichnis  
git branch -r # gibt alle branches die Remote getrackt  
werden aus. Manchmal aber auch NICHT! Dann nicht  
verzagen und trotzdem mal git checkout branch_name  
probieren!  
git pull # downloaded die neuesten Änderungen vom Server  
. Bevor du pullst, solltest du selbst committed haben  
um Konflikte zu vermeiden  
git push # uploaded deine Änderungen auf den Server.  
Sicherheitshalber vorher mal pullen!  
git fetch # dowloaded Metadaten wie neue Branches aber  
läd nicht aktiv Dateien runter
```

Funktionen definieren

```
def say_hello():  
    print("Hihi")  
  
def add_two_numbers(number_1, number_2):  
    """addiere zwei Werte und  
    gib das Ergebnis zurück  
    """  
    result = number_1 + number_2  
    return result  
  
def make_list(start, stop, step = 1):  
    result = []  
    for i in range(start, stop, step):  
        result.append(i)  
    return result
```


Funktionen aufrufen

```
say_hello() #-> schreibt "Hiii" in das Terminal  
add_two_numbers(40, 60) #-> 100, aber  
#nichts passiert mit dem Ergebnis  
uneven = make_list(1, 10, 2)  
#setzt die Variable uneven auf das Ergebnis  
#der Rechnung, die in der Funktion stattfindet
```

mit Dateien arbeiten

```
filename = "myfile.txt"
#file lesen
with open(filename, 'r') as file:
    lines = file.readlines()

#file schreiben (erstellen oder überschreiben)
with open(filename, 'w') as file:
    file.write("Inhalt der neuen Textdatei")

#an file text anhängen
with open(filename, 'a') as file:
    file.write("\nInhalt der neuen Textdatei")
```

Klassen und Objekte

```
class Dog:
    """ repräsentiert einen Hund """
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def sit(self):
        print(self.name + " sitzt brav.")

my_dog = Dog("Bello")
print(my_dog.name + " ist ein good boy.")
my_dog.age += 1      #Hat wohl Geburtstag
my_dog.sit()
```

Bibliotheken

```
#importieren
import random
würfel = random.randint(1, 6)
#einzelne Klassen importieren
from random import randint
würfel = randint(1, 6)  #etwas kürzer
```

gängige Bibliotheken

```
import math           #sin, log, pi, ...
import random         #random, randint
import requests       #Web-Interfaces
import json           #JSON
import csv            #CSV
import PIL            #Image Processing
import numpy          #Numerik
import matplotlib     #mathematisches Plotten
import tkinter        #GUIs
```