

Python

schnell und intensiv Programmieren lernen



Berry Boessenkool



frei verwenden, zitieren

2024-10-30 21:12

1. Intro

2. Objekte

3. Schleifen

4. Programmieren

1.1 Willkommen

1.2 Syntax

1.3 Datentypen

1.4 Funktionen schreiben

1.5 Module importieren

1.6 Zeichenketten



- ▶ Berry
- ▶ 2008-2017 Geoökologie @ Uni Potsdam
- ▶ "versehentlich" ein -Fan geworden
- ▶ Pakete , Community , Training & Beratung 
- ▶ seit 2019 in Teilzeit Dozent am HPI (Lehrstuhl Bert Arnrich)
- ▶ seit 2021 auch -Unterricht

- ▶ Python Grundlagen in zügigem Tempo
- ▶ Rechne mit 5-10 Stunden Aufwand pro Woche!
- ▶ Trotz KI-Assistenten: selbst programmieren (lernen) können
- ▶ Ziel: effizient und reproduzierbar arbeiten

- ▶ Python Eigenschaften (**Quelle**):
 - ▶ interpretierte Sprache (keine Kompilierung)
 - ▶ dynamische Typisierung (Datentypen werden zur Laufzeit überprüft)
 - ▶ objektorientiert (Daten mit Methoden)
 - ▶ high-level (von Menschen lesbar)

 - ▶ Eingesetzt in Data Science, Machine Learning, Webentwicklung, Spielentwicklung, Robotik, autonome Fahrzeuge, Entwicklung graphischer Nutzeroberflächen, Finanzwesen, ...

- ▶ 14 Lektionen
 - ▶ Video
 - ▶ PDF (pro Woche KursPDF mit allen Folien) mit *# Python Code auf grauem Hintergrund*
 - ▶ Multiple Choice Quiz (Selbsttest)
 - ▶ Automatisch bewertete Programmieraufgaben
 - ▶ Zeit und Dauer der Bearbeitung frei wählbar
 - ▶ Kursinhalte in der **Kursbeschreibung**
- ▶ pro Kurswoche (für einen Leistungsnachweis)
 - ▶ Multiple Choice Quiz (Hausaufgabe)
 - ▶ Nach Beginn 60 Minuten Zeit
- ▶ Fragen
 - ▶ **Kursforum** [Diskussionen] + Kommentaranfragen
 - ▶ Themenbezogene Fragen unter den Videos stellen
 - ▶ Beitragstitel zum leichten Suchen im Format **"1.3 A5"** beginnen
 - ▶ Gegenseitig beantworten und intensiver lernen!

- ▶ [Download & Installation](#), Hinweise für [Windows](#)
- ▶ offizielle [Dokumentation](#) und [Sprachreferenz](#)
- ▶ Drucken: RefCard, zB. [Laurent Pointal](#), [Berry](#) (nah am Kurs), [Suche](#)
- ▶ [codingame.com](#): Programmierwettbewerbe mit Suchtpotenzial

- ▶ Ausführlichere (langsamere) Kurse:
 - ▶ [openHPI](#) deutscher Python Junior Kurs
 - ▶ [bodenseo](#) deutsche Webseite + Buch
 - ▶ [Computer Science Circles](#) interaktiver englischer Kurs
 - ▶ [Python.org](#) offizielles englisches Tutorial

- ▶ Programmieraufgaben sind im Browser lösbar
- ▶ Im echten Leben lokal arbeiten - in einer Entwicklungsumgebung
- ▶ Für Python gibt es Hunderte IDEs ([Top 9](#), [RStudio](#))
- ▶ Ich empfehle [VS Code](#) (ggf. [Telemetry ausschalten](#))
- ▶ für viele Sprachen, z.B. Python, R, Julia, JavaScript, C++, SQL
- ▶ kann Script ausführen oder einzelne Zeilen (Demo)
- ▶ Wer's braucht: [Jupyter](#) notebooks (Julia, Python, R), zB. [google](#)

Zu jeder Lektion gibt es interaktive Code-aufgaben im Browser.

1.1 Programmieraufgaben

[Item bearbeiten](#)[Statistics](#)

Anweisungen:

Klicke unten um die Aufgabe zu öffnen.

🔔 Dies ist eine unbenotete Aufgabe. ✓ 2.0 Punkte

▶ Aufgabe starten

Der Zugang erfolgt via openHPI (nicht direkt per URL).

The screenshot displays the CodeOcean web interface. On the left, a file explorer shows two Python scripts: `sipl11_script1.py` and `sipl11_script2.py`. The main editor area contains a Python script with the following content:

```
1 # Willkommen zu den Übungsaufgaben im Kurs Python - schnell und intensiv
  # Programmieren lernen.
2 # Pro Lektion werden die Aufgaben in diesem Format gestaltet sein.
3 # Es gibt meistens 1 bis 3 Skripte mit je mehreren Aufgaben.
4
5 # Aufgabe 1 -----
6 # Erzeuge in der Konsole (rechts) die Ausgabe "Mein erster Text".
7 # Passe einfach den nachstehenden print Befehl an:
8 print("Hello World")
9 # Klicke auf den Knopf 'Ausführen' um das Skript durchlaufen zu lassen.
10
11 # Wenn die Ausgabe gut aussieht, klicke auf "Bewerten".
12 # Dann prüft ein Textskript (für dich unsichtbar) deine Lösung.
13 # In diesem konkreten Fall muss die Ausgabe sehr genau sein (kein Satzzeichen,
  # aber Großbuchstaben M und T).
14 # In den meisten Aufgaben gibt es mehr Freiheit beim Lösungsansatz.
15
16 # In den "echten" Lektionen werden hier mehrere Aufgaben stehen.
17 # Die Vorsilbe "sipl11" im Dateinamen steht übrigens für Schnell und Intensiv
  # Programmieren lernen, Lektion 1.1
18
19 # Weiter geht's in script2.py.
```

At the bottom of the editor, a status bar indicates "Zuletzt gespeichert: 13:33:02" and a button labeled "Diese Datei zurücksetzen".

On the right, the output console shows the result of the script execution:

```
Hello World
Die letzte Code-Ausführung wurde erfolgreich beendet (Statuscode 0).
```

1. Runterladen (lokal bearbeiten)
2. Alle Skripte dieser Lektion (heißt hier "Aufgabe") zurücksetzen
3. Skript ausführen lassen
4. Komplette Aufgabe bewerten lassen
5. Kommentaranfrage stellen (für individuelle Hilfe)
6. (unten): ein einzelnes Skript zurücksetzen

Programmieraufgaben Hinweise

Sollte beim Bewerten rechts unten mal `ERROR: test**` statt `FAIL:` stehen, bitte den Fehler verzeihen und die Nachricht im Forum melden.

Es ist nicht nötig, 100% zu erreichen um mit anderen Aufgaben im Kurs weiterzumachen.

Ich teile keine gebündelten Musterlösungen:

- ▶ es schränkt deine Kreativität und Diversität ein (es gibt meistens mehrere Lösungswege)
- ▶ es geht viel Lerneffekt verloren, wenn man dann doch "kurz reinschaut"
- ▶ du musst unbedingt lernen, selbst Lösungen zu finden
- ▶ zu einzelnen Aufgaben darfst du im Forum nach besseren Ansätzen fragen (didaktisch wertvoll). Schau auch gerne in die Kommentaranfragen anderer Teilnehmer rein.
- ▶ das Testskript schaut bereits, ob deine Lösung generalisiert anwendbar ist
- ▶ Good Coding ist eine Stilfrage, wo ich nicht allzuviel vorschreiben möchte

Vorstellung, Python, Programmieraufgaben:

- ▶ Programmieren ist eine mächtige Kompetenz
- ▶ Python ist toll
- ▶ Fragen im Forum stellen und beantworten
- ▶ RefCard drucken
- ▶ Wenn du lokal arbeiten willst: Python und VScode installieren
- ▶ Programmieraufgaben mit automatischer Bewertung

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro

2. Objekte

3. Schleifen

4. Programmieren

1.1 Willkommen

1.2 Syntax

1.3 Datentypen

1.4 Funktionen schreiben

1.5 Module importieren

1.6 Zeichenketten

Operatoren, Kommentare

```
5 + 8 # Kommentare (nach einer Raute) werden ignoriert
```

```
## 13
```

```
6/7 # Leerzeichen sind irrelevant (hier)
```

```
## 0.8571428571428571
```

```
12 * 3.4 # Dezimalzahlen mit einem Punkt angeben
```

```
## 40.8
```

```
3 ** 2 # Exponenten (nicht 3^2!)
```

```
## 9
```

```
19 // 3 # ganzzahlige Division
```

```
## 6
```

```
19 % 3 # Modulo (Rest nach ganzzahliger Division)
```

```
## 1
```

```
'''
```

```
Dies ist ein Kommentar
```

```
über mehrere Zeilen.
```

```
-> Gut zum Dokumentieren von Code :)
```

```
'''
```

Objekte, Ausgaben auf der Konsole (shell / terminal)

Objekt erstellen mit `=` Zeichen:

```
x = "Hallo, python Welt!"  
x  
## 'Hallo, python Welt!'
```

Zeichenketten sind erstellbar mit `"` und `'`.

In den meisten Umständen (z.B. VS Code, CodeOcean), braucht es einen expliziten Aufruf, um etwas in der Konsole anzuzeigen:

```
print(x)  
## Hallo, python Welt!
```

`print` braucht man nicht in jupyter notebooks, bei zeilenweiser Ausführung und in diesen Folien (erstellt mit Rnw =  + $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$).

Wenn du in VScode Befehle zeilenweise ausführst (`>>>` in console), kannst du mit `quit()` wieder in den normalen Modus wechseln.

print

```
print("Ich schreibe: ", x)
## Ich schreibe:  Hallo, python Welt!
```

`print` konkateniert verschiedene Eingaben, getrennt durch `sep` (default " ") und abgeschlossen mit `end` (default "\n")

```
a = 5
print("a ist:", a, "gut", sep="0", end="\n\n\n")
print("a ist immer noch:", a)
## a ist:050gut
##
## a ist immer noch: 5
```

```
print("Berlin", end = "#")
print("Potsdam")
## Berlin#Potsdam
```

Häufige Fehler bei Objektnamen

```
a = 42
```

Groß-/Kleinschreibung beachten:

```
A + 88
```

```
## NameError: name 'A' is not defined
```

Objektnamen dürfen nicht mit Zahlen beginnen:

```
12b = 67 / 5
```

```
## SyntaxError: invalid decimal literal
```

Schlüsselwörter dürfen nicht als Objektnamen verwendet werden:

```
class = 4
```

```
## SyntaxError: invalid syntax
```

Ausführliche Liste [häufiger Fehler](#), Liste der [reservierten Keywords](#)

SyntaxError: oft Klammern oder Doppelpunkte vergessen (in Schleifen)

Objektnamen sollten kurz und informativ sein,

z.B. `anzahl_teilnehmer` oder `video_dauer`

```
eine_liste = [42, 77, -5, 6]
eine_liste
## [42, 77, -5, 6]
```

```
eine_liste.append(111) # Methode für Listenobjekte
```

Das Objekt wird verändert, ohne es neu zuzuweisen
(wenn es veränderbar ist, siehe Lektion [2.1 Objekttypen](#)).

```
eine_liste
## [42, 77, -5, 6, 111]
```

Wenn eine Methode ein Objekt zurückgibt, können Methoden verknüpft werden (chaining):

```
satz = "Eine normale Zeichenkette"
```

```
satz.count("e") # zählt nur kleine e  
## 6
```

```
satz.lower() # macht alles klein  
## 'eine normale zeichenkette'
```

```
satz.lower().count("e") # zählt alle e  
## 7
```

```
nutzer_eingabe = input()
print("Die Eingabe war:", nutzer_eingabe)
nutzer_eingabe = input("Bitte etwas eingeben: ")
```

Die Ausgabe von `input` ist immer eine Zeichenkette (string), auch wenn eine Zahl eingegeben wird.

Grundlegende Befehle wie `print`, `+`, `input`, etc sind immer verfügbar. Funktionen wie `sqrt` sind in einem eigenen Paket (Modul). Diese müssen zuerst geladen werden:

```
import math
```

```
math.sqrt(700)
```

```
## 26.457513110645905
```

Weitere Methoden in [Lektion 1.5 Module importieren](#)

Zeilen einer Textdatei lesen

```
fname = "textDatei.txt"
with open(fname) as f:
    inhalt = f.read() # .splitlines()
print(inhalt)
## Dies ist ein kleines Beispiel einer Textdatei
## um das Einlesen von Zeilen in Python zu demonstrieren.
## Für echte Daten das Paket `pandas` verwenden!
```

`open("textDatei.txt")` öffnet die Datei im Lesemodus

`with` garantiert, dass die Datei am Ende geschlossen wird (besonders im Falle eines Fehlers)

Die Zeile nach `with` muss eingerückt sein

`inhalt.splitlines()` gibt eine Liste mit einer Textzeile pro Listenelement aus

```
a = 17
a = a + 8
a
## 25
```

```
a = 17
a += 8
a
## 25
```

In der DataScience als schlecht lesbar angesehen und nicht sehr empfohlen, wird aber besonders in Schleifen durchaus verwendet.

```
a *= a+2 # a = a*(a+2)
```

Syntax, Objekte, Operatoren, Funktionen:

- ▶ `+`, `-`, `*`, `/`, `**`, `//`, `%`
- ▶ `obj = "string" # Kommentar ;`
`"""mehrzeiliger Kommentar"""`
- ▶ `print("Zeichen", 42, obj, sep=" ", end="\n")`
- ▶ `obj.method()` ; `string.lower().count("p")`
- ▶ `nutzer_eingabe = input()`
- ▶ `import modul` ; `modul.funktion(x)`
- ▶ `with open("datei.txt") as f: inhalt = f.read()`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro

2. Objekte

3. Schleifen

4. Programmieren

1.1 Willkommen

1.2 Syntax

1.3 Datentypen

1.4 Funktionen schreiben

1.5 Module importieren

1.6 Zeichenketten

Wir haben schon drei Sorten von Daten gesehen: ganze Zahlen, Kommazahlen und Zeichenketten.

```
type(5)
## <class 'int'>
```

```
type(5.67)
## <class 'float'>
```

```
type("Zeichen")
## <class 'str'>
```

```
print(7 > 4, 7 < 4)
## True False
type(7 > 4)
## <class 'bool'>
```

```
type(2+3j) # Komplexer Teil mit j statt mit i
## <class 'complex'>
```

```
isinstance(7.5, int) # Klassenzugehörigkeit prüfen  
## False
```

```
isinstance("Hello", (float, int, str) ) # einer von den 3  
## True
```

```
obj = "Zeichenkette"  
print("Typ ist: ", type(obj) )  
## Typ ist: <class 'str'>
```

```
int("48")
```

```
## 48
```

```
float("48")
```

```
## 48.0
```

```
int("drei") # Fehler falls nicht konvertierbar
```

```
## ValueError: invalid literal for int() with base 10:  
'drei'
```

```
type(float("25"))
```

```
## <class 'float'>
```

```
# Coding demo:
```

```
f = float(input('Temperatur in °Fahrenheit eingeben: '))
```

```
c = (f-32) / 1.8
```

```
print("Temperatur in °Celcius:", round(c, 1))
```

Datentypen ermitteln:

- ▶ `type(objekt)`
- ▶ `int(etwas_das_konvertiert_werden_kann)`
- ▶ `isinstance(objekt, int)`
- ▶ `isinstance(objekt, (float, int, str))`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

- 1. Intro
 - 2. Objekte
 - 3. Schleifen
 - 4. Programmieren
- 1.1 Willkommen
 - 1.2 Syntax
 - 1.3 Datentypen
 - 1.4 Funktionen schreiben**
 - 1.5 Module importieren
 - 1.6 Zeichenketten

Funktionssyntax I

```
def willkommen(name, zeit="morgen"):
    msg = "Hallo, " + name + ". Guten " + zeit + "!"
    return msg
```

```
willkommen(name='Bob', zeit='Abend')
## 'Hallo, Bob. Guten Abend!'

willkommen(name='Berry') # mit dem Standardfall (morgen)
## 'Hallo, Berry. Guten morgen!'
```

- ▶ Doppelpunkt `:` notwendig, danach einrücken
- ▶ mit Leerzeichen oder Tabstop, Hauptsache einheitlich
- ▶ Ohne explizites `return` gibt eine Funktion `None` zurück
- ▶ `return` beendet den Funktionsaufruf
- ▶ `name` & `zeit` sind Parameter, `"Berry"` & `"Abend"` sind Argumente
- ▶ Syntax: `parameter=argument`

Funktionssyntax II

- ▶ Parameternamen können beim Aufruf weggelassen werden, sofern die Argumente in der richtigen Reihenfolge gegeben werden:

```
willkommen("Lydia", "Mittag")  
## 'Hallo, Lydia. Guten Mittag!'
```

- ▶ `print()` zeigt eine Ausgabe in der Konsole an.
- ▶ `return` beschreibt den Funktionsrückgabewert, der danach zugewiesen werden kann
- ▶ und mit dem weitergearbeitet werden kann.
- ▶ In der Realität eigentlich immer `return` verwenden
- ▶ Das Wort "ausgeben" wird für "zurückgeben" **und** "anzeigen" verwendet. In den Programmieraufgaben ist meist `return` oder `print` vorgegeben.
- ▶ **IndentationError**: Zu wenig oder zu viel Einrückung

Beispiel einer Funktion

```
import math
def kugel_volumen_berechnen(radius, nachkommastellen=3):
    volumen = (4/3) * math.pi * radius**3
    gerundet = round(volumen, nachkommastellen)
    return gerundet
```

```
kugel_volumen_berechnen(2)
## 33.51
kugel_volumen_berechnen(2, 5)
## 33.51032
```

Was ist hier gut?

- ▶ Funktionsname ist ein Verb
- ▶ Parameternamen erklären sich selbst
- ▶ gute Objektnamen innerhalb der Funktion
- ▶ `math` außerhalb der Funktion importiert

Scoping (Gültigkeitsbereich) - welche Objekte werden gefunden I

Lokale Objekte in einer Funktion haben Vorrang:

```
import math
def tische_zaehlen(n):
    tische = n/6
    aufgerundet = math.ceil(tische)
    return aufgerundet
```

```
anzahl_gaeste = 78
tische_zaehlen(anzahl_gaeste)
## 13
```

```
tische = 4000
tische_zaehlen(22) # Funktion verwendet internes Objekt
## 4
```

```
aufgerundet
## NameError: name 'aufgerundet' is not defined
```

Lokale Objekte (innerhalb einer Funktion) sind temporär und nach der Ausführung der Funktion nicht global verfügbar

Globale Objekte sind in einer Funktion verfügbar:

```
def anzahl_stuehle(n):  
    return anzahl_gaeste
```

```
anzahl_stuehle(17)  
## 78
```

Mit globalen Objekten innerhalb einer Funktion nur arbeiten, wenn man weiß, was man tut!

wiederverwendbare Codeblöcke:

- ▶ Funktionssyntax:

```
def funktionsname(parameter, mit_default="argument"):
    ausgabe = parameter + 5
    return ausgabe
parameter + 8 # code nach return nie ausgeführt
```

- ▶ Einrückung einheitlich
- ▶ print vs return beachten
- ▶ lokale Objekte existieren nur innerhalb der Funktion
- ▶ globale Objekte nicht in einer Funktion verwenden

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 1.1 Willkommen
 - 1.2 Syntax
 - 1.3 Datentypen
 - 1.4 Funktionen schreiben
 - 1.5 Module importieren**
 - 1.6 Zeichenketten

Eingebaute Pakete (Module)

Python kommt mit einigen Modulen vorinstalliert (Standardbibliothek).
Diese müssen nicht installiert werden ([Liste](#)).
Zur Verwendung in einem Skript müssen sie aber geladen werden.

```
import math
```

```
radius = float(input('Bitte Radius eingeben: '))
print("Der Umfang ist ", 2 * math.pi * radius)
```

Gute Praxis beachten:

```
from math import * # NICHT BENUTZEN!
```

```
from math import sqrt, pi # schwer nachvollziehbar
```

```
import math # sauber, lesbar
math.pi # -> Königsweg
```

```
import math as m # etwas kürzer
m.pi # pandas -> pd
```

Externe Module



Quelle

Populäre Pakete

- ▶ Data science: `numpy`, `pandas`
- ▶ Machine learning: `tensorflow`, `pytorch`
- ▶ Statistical analysis: `scipy`
- ▶ Web application: `django`
- ▶ Plotting: `matplotlib`, `seaborn`

Module von `pypi.org` (PYthonPackageIndex) installieren mit `pip` (bei Python inklusive).

In Konsole / Terminal / Shell / bash / cmd:

```
pip install numpy # pip3 unter MacOS
pip list
```

Auf dem Macbook mache ich das mit R:

```
install.packages("reticulate")
reticulate::install_miniconda()
reticulate::py_install("numpy")
```

ImportError: Name falsch geschrieben?

Zufallszahlen:

```
import random
random.random() # float von 0 (inkl) bis 1 (exklusive)
random.randint(1, 6) # int von 1 (inkl) bis 6 (inkl!!!)
```

Zählung:

```
import collections
f = ['rot', 'blau', 'blau', 'gelb', 'blau', 'rot', 'lila']
collections.Counter(f).most_common(3)
## [('blau', 3), ('rot', 2), ('gelb', 1)]
```

Mehrere Module können in einer Zeile importiert werden:

```
import collections, statistics, random
```

Arbeitsverzeichnis (Pfad, Ordner), current working directory:

```
import os
os.getcwd().replace(os.sep, '/')
## 'C:/Dropbox/pymooc/folien'
```

Siehe auch das Modul `pathlib`

Dort liegt `meinskript.py` mit `anzahl = 25`.

```
from meinskript import anzahl
print(anzahl+2)
## 27
```

Beachte: kein echter Dateiname (mit .py-Endung)!

```
import meinskript
meinskript.anzahl
## 25
```

```
import meinskript
print("\n".join(dir(meinskript))) # Objekte im Modul
## __builtins__
## __cached__
## __doc__
## __file__
## __loader__
## __name__
## __package__
## __spec__
## anzahl
## job
## random
## simuliere_job
```

```
type(meinskript.simuliere_job)
## <class 'function'>
```

```
meinskript.simuliere_job()
## 'coder'
meinskript.simuliere_job()
## 'arzt'
```

```
import inspect
print(inspect.getsource(meinskript.simuliere_job))
## def simuliere_job():
##     jobs = ["lehrer", "handwerker", "arzt", "coder"]
##     return random.choice(jobs)
```

Wenn Argumente vorhanden sind:

```
inspect.getfullargspec(meinskript.simuliere_job)
## FullArgSpec(args=[], ... defaults=None ...)
```

Module und Skripte importieren:

- ▶ `import paket as pak ; pak.fun()`
- ▶ `from paket import fun ; fun()`
- ▶ `pip install externes_paket` in einer Konsole
- ▶ `from lokale_python_datei import ein_objekt`
- ▶ `inspect.getsource(eine_function)`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 1.1 Willkommen
 - 1.2 Syntax
 - 1.3 Datentypen
 - 1.4 Funktionen schreiben
 - 1.5 Module importieren
 - 1.6 Zeichenketten

Teile von Zeichenketten auswählen (subsetting, indexing)

```
k = "Hallo, Welt!"
k[7] # 8. Buchstabe
## 'W'
```

Python indiziert ab 0!

```
k[3:7] # 4. bis 7. (rechts
exklusiv des 8.)
## 'lo, '
```

```
k[:6] # 1. bis 6.
## 'Hallo, '
```

```
k[2:] # 3. bis letzte
## 'llo, Welt!'
```

```
k[-2] # vorletzte
## 't'
```

```
k[-5:-2] # kombinierbar
## 'Wel'
```

```
k[5:-2] # gemischt geht
## ', Wel'
```

```
k[400]
## IndexError: string index out of range
```

```
k[3.5]
## TypeError: string indices must be integers
```

```
k[2] = 'K' # strings sind nicht änderbar (siehe 2.1)
## TypeError: 'str' object does not support item assignment
```

```
a = "Hallo"
```

```
len(a) # Anzahl Zeichen  
## 5
```

```
"Zeichen " + "Kette " + a # zusammenfügen  
## 'Zeichen Kette Hallo'
```

```
"Zeichen " + "Kette " + 77  
## TypeError: can only concatenate str (not "int") to str
```

```
3 * a # wiederholen  
## 'HalloHalloHallo'
```

```
3 / a  
## TypeError: unsupported operand type(s) for /: 'int'  
and 'str'
```

```
"lo" in a # Präsenz prüfen  
## True
```

Zeichenketten aufteilen und zusammenführen

```
zk = "Etwas Text und einige Worte"
zk.split() # Ausgabe: Liste. Standard: sep=" "
## ['Etwas', 'Text', 'und', 'einige', 'Worte']
zk # immutable: nicht verändert
## 'Etwas Text und einige Worte'
```

```
print("Zeichenkette mit\nZeilenumbruch")
## Zeichenkette mit
## Zeilenumbruch
```

```
"Zeichenkette mit\nZeilenumbruch".split(" ")
## ['Zeichenkette', 'mit\nZeilenumbruch']
```

```
"Zeichenkette mit\nZeilenumbruch".split()
## ['Zeichenkette', 'mit', 'Zeilenumbruch']
# sep Default ist jede Art von Leerzeichen, inkl. \n
```

```
"_".join(["Liste", "mit", "Worten"])
## 'Liste_mit_Worten'
```

```
name="Berry" ; anzahl=7
```

```
f"Ich bin {name} und habe {anzahl} Katzen."  
## 'Ich bin Berry und habe 7 Katzen.'
```

```
f"Ich bin {name} und habe {anzahl+4} Katzen."  
## 'Ich bin Berry und habe 11 Katzen.'
```

siehe realpython.com

Zeichenketten (strings) verarbeiten:

- ▶ `string[position]`
- ▶ `len(string)`
- ▶ `+`, `*`, `in`
- ▶ `string.split()`, `string.join()`
- ▶ `f"string mit {Objekt} Referenz"`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
2. Objekte
3. Schleifen
4. Programmieren

2.1 Objekttypen

2.2 Listen

2.3 Dictionaries

Objekte sind zum Teil veränderbar

In Python sind einige Objekte veränderbar (mutable):

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

```
eine_liste[1] = 99 # Listen sind veränderbar  
eine_liste  
## [1, 99, 3]
```

```
eine_liste.append(77) # Methode = Funktion einer Klasse  
eine_liste # verändert, ohne neue Zuweisung  
## [1, 99, 3, 77]
```

Zeichenketten sind nicht veränderbar:

```
kette = "Python ist toll"  
kette[11] = "T"  
## TypeError: 'str' object does not support item  
assignment
```

- ▶ Ähnlich wie Listen

```
(4, 8, 4, -3.14)
```

- ▶ ## (4, 8, 4, -3.14)

```
(4, 8, "drei") # können gemischte Datentypen sein
```

- ▶ ## (4, 8, 'drei')

- ▶ Oft als Subelement einer Liste genutzt
- ▶ um verschachtelte Klammern lesbar zu halten

```
[(8,5), 9, 3, (4,7)]
```

- ▶ ## [(8, 5), 9, 3, (4, 7)]

Mehrfachzuweisung (multiple assignment) in einer Zeile

```
def eingaben_verdoppeln(x, y):  
    return x*2, y*2
```

```
ergebnis = eingaben_verdoppeln(3, 4)  
ergebnis # ein Tupel  
## (6, 8)
```

```
a, b = eingaben_verdoppeln(3, 4)  
a # zwei separate Objekte  
## 6  
b  
## 8
```

Tauschen zweier Variablen:

```
a, b = b, a # rechte Seite wird zuerst ausgeführt
```

Mengen (sets)

```
s1 = {1, 2, 3, 4, 5}
s2 = {3, 4, 5, 6, 7, 8}
```

Operatoren wie in mathematischen Mengen

```
s1 | s2                                # Vereinigung
## {1, 2, 3, 4, 5, 6, 7, 8}
```

```
s1 & s2                                # Schnitt
## {3, 4, 5}
```

```
s1 - s2                                # Differenz: in s1, nicht in s2
## {1, 2}
```

```
s2 - s1
## {8, 6, 7}
# zeigt, dass die Einträge nicht geordnet sind
```

```
{ } # leeres Dictionary (nicht Menge!)
set() # leere Menge
```

Mengen haben einmalige Werte

```
zahlen = [6, 3, 3, 4]
```

Wenn Listen in ein Set konvertiert werden, sind die Einträge (mögliche Ausprägungen) nur einmal drin:

```
set(zahlen)
## {3, 4, 6}
```

```
set([6, 3, "3", 4])
## {3, 4, 6, '3'}
```

"3" ist was anderes als 3

Neue Werte können hinzugefügt werden:

```
s1 = {1, 2, 3, 4, 5}
s1.add(7)
s1
## {1, 2, 3, 4, 5, 7}
```

Typ	Beispiel	änderbar	geordnet	indiziert	Duplikate
Liste	[1,3]	ja	ja	ja	ok
Tupel	(1,2)	nein	ja	ja	ok
Menge	{1,4}	ja *	nein	nein	nein
Dict	{"a":7, "b":6}	ja	nein / ja seit Python 3.6	mit Schlüssel	nein **

*: die Elemente müssen immutable sein

** : die Schlüssel müssen einmalig sein, Werte sind mehrfach erlaubt

Liste: verschiedene Daten

Tupel: gruppieren verwandter Daten

Menge: schnelle Überprüfung, ob ein Wert enthalten ist

Dictionary: schnelles Abrufen anhand von Schlüsselnamen

Überblick Objekte, Tupel und Mengen:

- ▶ Liste, Tupel, Menge, Dictionary
- ▶ veränderbar, geordnet
- ▶ Mengenoperationen

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. **Objekte**
 3. Schleifen
 4. Programmieren
- 2.1 Objekttypen
 - 2.2 **Listen**
 - 2.3 Dictionaries

Liste: Erstellung + Teilmengenbildung I

```
liste = [7, -4, 9, 1, 2, 3, 9, 5]
```

```
len(liste)
```

```
## 8
```

```
liste[0] # erstes Element
```

```
## 7
```

```
liste[1] # zweites Element
```

```
## -4
```

```
liste[5] # Element 6
```

```
## 3
```

```
liste[2:5] # Elemente 3,4,5
```

```
## [9, 1, 2]
```

Bereiche sind exklusiv am rechten Ende

: ist außerhalb der Teilmengenbildung kein Operator

```
leere_liste = []
```

Liste: Erstellung + Teilmengenbildung II

```
liste  
## [7, -4, 9, 1, 2, 3, 9, 5]
```

```
liste[4:] # Teilung: fünftes bis letztes Element  
## [2, 3, 9, 5]
```

```
liste[:6] # 1. - 6.  
## [7, -4, 9, 1, 2, 3]
```

```
liste[-2] # vorletztes Element  
## 9
```

```
liste[2:-3] # hier äquivalent zu liste [2:5]  
## [9, 1, 2]
```

```
liste[2] = "neuerWert" # überschreibe drittes Element  
liste  
## [7, -4, 'neuerWert', 1, 2, 3, 9, 5]
```

verschiedene Datentypen möglich

Listenelement an Index entfernen

```
zahlen = [1,2,3,4,5,6,7]
```

```
ende = zahlen.pop() # letztes Element entfernen + ausgeben  
ende  
## 7  
zahlen # verändert, ohne neue Zuweisung  
## [1, 2, 3, 4, 5, 6]
```

```
zahlen.pop(3) # angegebenes Element entfernen (+ ausgeben)  
zahlen  
## 4  
## [1, 2, 3, 5, 6]
```

```
del(zahlen[2]) # Element an Index entfernen  
zahlen  
## [1, 2, 5, 6]
```

`del` funktioniert auch ohne Klammern

```
werte = [1,2,3,4,5,6,7,4]
```

```
4 in werte # prüft ob in der Liste, gibt boolean zurück  
## True
```

```
9 not in werte # prüft ob nicht in der Liste  
## True
```

```
werte.index(4) # Index vom ersten Auftreten von 4  
## 3
```

```
werte.remove(4) # erste Instanz von 4 entfernen  
werte  
## [1, 2, 3, 5, 6, 7, 4]
```

```
werte.append(66) # am Ende anfügen
werte
## [1, 2, 3, 5, 6, 7, 4, 66]
```

```
werte.insert(3, "neu") # an Position einfügen
werte
## [1, 2, 3, 'neu', 5, 6, 7, 4, 66]
```

`insert` verschiebt Elemente nach Eingefügtem nach hinten

```
zahlen + werte[:6]
## [1, 2, 5, 6, 1, 2, 3, 'neu', 5, 6]
```

```
j_liste = [7, -4, 9, 1, 2, 3]
j_liste.reverse()
j_liste
## [3, 2, 1, 9, -4, 7]
```

```
j_liste.sort()
j_liste
## [-4, 1, 2, 3, 7, 9]
```

```
j_liste.sort(reverse=True)
j_liste
## [9, 7, 3, 2, 1, -4]
```

```
k_liste = [7, -4, "9", 1, 2, 3]
k_liste.sort()
## TypeError: '<' not supported between instances of
'str' and 'int'
```

Listenverschachtelung + verlängerung

```
m_liste = [1, 2, 3, [31,32,33], 4] # Verschachtelung OK
```

```
ch_liste = ["Wörter", "mit", "vielen", "Buchstaben"]  
ch_liste[2][5] # fortlaufend indiziert: Wort 3, Buchst. 6  
## 'n'
```

```
lx = [1, 2, 3, 4]  
ly = [5, 6]  
lz = [7, 8, 9]
```

```
lx.extend(ly) # ändert lx  
lx  
## [1, 2, 3, 4, 5, 6]
```

```
lx + ly # ändert lx nicht, dafür: lx = lx + ly  
## [1, 2, 3, 4, 5, 6, 5, 6]
```

```
lz * 2  
## [7, 8, 9, 7, 8, 9]
```

```
woerter = ["mehrere", "Wörter"]
```

```
"\n".join(woerter)
## 'mehrere\nWörter'
print("\n".join(woerter))
## mehrere
## Wörter
```

```
gemischt = [99, "Luftballons"]
```

```
print("\n".join(gemischt))
## TypeError: sequence item 0: expected str instance,
int found
```

```
# Stern (splat) Operator entpackt Liste
```

```
print(*gemischt, sep="\n")
## 99
## Luftballons
```

Korrigiere den Fehler im folgenden Code. Warum tritt er auf?

```
liste = ["a1", "b1", "b2", "b3", "b4", "b5", "c1", "d1"]
liste = liste.reverse()
print(liste[2]) # sollte "b5" sein
## TypeError: 'NoneType' object is not subscriptable
```

Die Listenmethode 'reverse' ändert die Liste selbst, da das ein veränderbares Objekt ist.

Sie wird für den Seiteneffekt aufgerufen, daher "gibt sie nichts zurück". Sie gibt `None` zurück, was 'liste' überschrieben hat.

```
liste = ["a1", "b1", "b2", "b3", "b4", "b5", "c1", "d1"]
liste.reverse()
print(liste[2])
## b5
```

Liste kopieren für eine unabhängige Instanz

```
liste = [1,2,3,4,5,6]
```

```
andere_liste = liste
```

```
andere_liste
```

```
## [1, 2, 3, 4, 5, 6]
```

```
liste.append(9)
```

```
andere_liste # hat sich auch verändert!
```

```
## [1, 2, 3, 4, 5, 6, 9]
```

Referenziert dasselbe Objekt im Arbeitsspeicher(Pointer, `id(objekt)`)

```
dritte_liste = liste.copy() # unabhängige Kopie
```

```
dritte_liste
```

```
## [1, 2, 3, 4, 5, 6, 9]
```

```
liste[1] = 99
```

```
dritte_liste # wurde nicht verändert
```

```
## [1, 2, 3, 4, 5, 6, 9]
```

Gültigkeitsbereich (Scope) in Funktionen (Quelle)

```
# Liste in einer Funktion kürzen
def maxOhne5(ll):
    ll.remove(5)
    return max(ll)
# verändert globale Liste (veränderbares Objekt)
liste_a = [1,2,3,4,5]
print(liste_a)           # [1, 2, 3, 4, 5]
print(maxOhne5(liste_a)) # 4
print(liste_a)          # [1, 2, 3, 4] <-- verändert!
```

```
def maxOhne5(ll):
    ll = ll.copy() # lokale Kopie mit andere ID
    ll.remove(5)
    return max(ll)
liste_b = [1,2,3,4,5]
print(liste_b)           # [1, 2, 3, 4, 5]
print(maxOhne5(liste_b)) # 4
print(liste_b)          # [1, 2, 3, 4, 5]
```

Zusammenfassung für 2.2 Listen

Listen sind die Basis von allem:

- ▶ Indexierung (Submengen auswählen) ab 0, (slicing)
- ▶ gemischte Datentypen sind möglich
- ▶ `len(liste)`, `liste.pop(index)`, `in`, `not in`
- ▶ `liste.index(wert)`, `liste.remove(wert)`
- ▶ `liste.append(wert)`, `liste.insert(index, wert)`
- ▶ `listeA + listeB`, `listeA.extend(listeB)`
- ▶ `liste.reverse()`, `liste.sort()`
- ▶ `liste.copy()`
- ▶ veränderbare Objekte innerhalb einer Funktion ändern, ändert sie global

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. **Objekte**
 3. Schleifen
 4. Programmieren
- 2.1 Objekttypen
 - 2.2 Listen
 - 2.3 **Dictionaries**

- ▶ speichert Daten in Form Schlüssel:Wert
- ▶ optimiert, Werte schnell abzurufen, wenn der Schlüssel bekannt ist
- ▶ Schlüssel sind einmalig, Datentyp der Werte ist egal

Nutzungsbeispiele:

- ▶ Produktkatalog
- ▶ Patienten IDs mit weiteren Informationen (kann Liste oder Unter-Dictionary sein)
- ▶ HTTP-Statuscodes
- ▶ Kontaktliste (Telefonbuch)
- ▶ Warenkorb in Onlineshops

```
dozent = {'Name': "Berry",  
          'Alter': 32}
```

```
len(dozent)  
## 2
```

```
dozent['Name'] # Zugriff auf einen Eintrag  
## 'Berry'
```

```
dozent['Alter'] = 33 # Eintrag überschreiben  
dozent  
## {'Name': 'Berry', 'Alter': 33}
```

```
dozent['Studis'] = 42 # Eintrag hinzufügen  
dozent  
## {'Name': 'Berry', 'Alter': 33, 'Studis': 42}
```

Ich benutze ' Apostrophen in Dictionaries, da sie in f-strings mit " Anführungszeichen benutzt werden können.

Doppelte und einfache Anführungszeichen können im F-string nicht gemischt werden.

```
f"Erstelle einen String mit {3+4} Berechnungen."  
## 'Erstelle einen String mit 7 Berechnungen.'
```

```
f"Hi {dozent['Name']}, du bist {dozent['Alter']} Jahre  
alt."  
## 'Hi Berry, du bist 33 Jahre alt.'
```

Seit Python 3.12 (2023-10) ist das doch möglich.

fehlersichere Auswahl eines Schlüssels:

```
dozent.get('Name', "Wert falls Schlüssel fehlt")  
## 'Berry'
```

```
dozent.get('NAME', "Wert falls Schlüssel fehlt")  
## 'Wert falls Schlüssel fehlt'
```

```
dozent.keys() # Quasi eine Liste als Ausgabe  
## dict_keys(['Name', 'Alter', 'Studis'])
```

```
dozent.values() # dozent.items() zum Iterieren  
## dict_values(['Berry', 33, 42])
```

```
"Alter" in dozent.keys()  
## True
```

```
"Alter" in dozent # kürzer und schneller :)  
## True
```

Elemente aus einem Dictionary entfernen

```
dozent = {'Name': "Berry", 'Alter': 32, 'Studis': 42, 'z': 0}
del(dozent['Alter']) # entfernt kompletten Eintrag
dozent
## {'Name': 'Berry', 'Studis': 42, 'z': 0}
```

```
studis = dozent.pop('Studis')
studis
## 42
dozent
## {'Name': 'Berry', 'z': 0}
```

```
dozent.pop('Alter') # Schlüssel nicht mehr da
## KeyError: 'Alter'
dozent.pop('Alter', None) # gibt None zurück
```

```
dozent.clear() # Alle Einträge entfernen
dozent
## {}
```

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
```

```
dict2 = dict1 # dict2 ist nur ein Zeiger auf dict1
```

```
dict3 = dict1.copy() # unabhängige Kopie
```

Wenn man `dict1` verändert, ändert sich auch `dict2`.
(Das gilt für alle veränderbaren Objekte).

```
dict1['c'] = 333
```

```
dict2
```

```
## {'a': 1, 'b': 2, 'c': 333}
```

```
dict3
```

```
## {'a': 1, 'b': 2, 'c': 3}
```

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
```

```
dict2 = {'a': 11, 'd': 4}
```

```
dict1.update(dict2) # aktualisiert Werte oder fügt  
Einträge hinzu
```

```
dict1  
## {'a': 11, 'b': 2, 'c': 3, 'd': 4}
```

```
dozent = {'Name': "Berry", 'Alter' : 32}
for k, v in dozent.items():
    print("Der Eintrag '",k,"' hat den Wert: ",v, sep="")
## Der Eintrag 'Name' hat den Wert: Berry
## Der Eintrag 'Alter' hat den Wert: 32
```

Nächste Woche behandeln wir For-Schleifen :)

dictionaries (Schlüssel-Wert Paare):

- ▶ `dict = {'Schluessel': "Wert"}`
- ▶ `dict['Schluessel']`
- ▶ `dict.get('Schluessel', "Ersatz")`
- ▶ `'Schlüssel' in dict`
- ▶ `dict.pop('Schluessel', "Ersatz")`, `dict.clear()`
- ▶ `dict.copy()`
- ▶ `dictA.update(dictB)`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 3.1 Bedingte Codeausführung
 - 3.2 For- und While-Schleifen
 - 3.3 List comprehension

Vergleichende und logische Operatoren

vergleichende Operatoren:

```
a == b # ist a gleich b?
a != b # ungleich?
a < b  # kleiner als
a <= b # kleiner / gleich
a > b  # größer als
a >= b # größer / gleich
```

Kommazahlen gerundet **vergleichen**:

```
summe = 5.1 + 1.1
summe == 6.2
## False

round(summe,8) == round(6.2, 8)
## True

import math
math.isclose(summe, 6.2)
## True
```

```
wert = 8
7 < wert < 9 # nice!
## True
```

```
7 < 8
"9" < "A"
"A" < "B"      # alle
"a" < "b"      # True
"A" < "a"
# zahl < GROSS < klein
```

logische Operatoren:

```
not False      # NICHT
## True
True and 6 > 8 # UND
## False
9 > 8 or False # ODER
## True
```

```
7>1 & 6>1  
## False
```

`&` ist ein Bit-Operator!
(Nicht der UND-Operator wie in anderen Programmiersprachen)

```
7>1 and 6>1  
## True
```

Das gilt auch für das Zirkumflex (^):

```
2^3 # ist eigentlich 8  
## 1
```

`^` ist ein binäres XOR (ausschließliches ODER)
siehe [stackoverflow](#) / [Was macht der Zirkumflex-Operator](#)

`&` führt bitweise UND Operation auf Bits aus
siehe [Wikipedia](#) / [Bitweise Operatoren](#)

Bedingte Codeausführung: allgemeine Struktur

```
a = 10 ; b = 20
if b > a:
    print("b ist größer als a")
## b ist größer als a
```

```
a = 30 ; b = 30
if b > a:
    print("b ist größer als a")
elif a == b:
    print("a und b sind gleich")
## a und b sind gleich
```

```
a = 100 ; b = 30
if b > a:
    print("b ist größer als a")
elif a == b:
    print("a und b sind gleich")
else:
    print("a ist größer als b")
## a ist größer als b
```

Bedingte Codeausführung: Bedingungen kombinieren

```
a = 100 ; b = 30
if a>b and (b==20 or b==30):
    print("a ist größer und b ist 20 oder 30")
## a ist größer und b ist 20 oder 30
```

```
cond = [True, False, True, True, True]
all(cond) # alle wahr?
## False
any(cond) # mindestens 1 wahr?
## True
sum(cond) # Anzahl wahr
## 4
```

Bedingte Codeausführung in einer Funktion: Übersicht

Für eine Vierfeldertafel gibt es zwei Ansätze

		cond 2	
		True	False
cond 1	True	t,t	t,F
	False	F,t	F,F

mit `return`, verschachtelt, ohne `elif`:

```
if cond1:
    if cond2: return "tt"    # Zeilenumbrüche vor return
    else: return "tF"      # aus Platzgründen
if cond2: return "Ft"     # weggelassen. In echt bitte
else: return "FF"        # setzen für den style guide
```

mit `print`, eine Ebene, mit `and` + `elif`:

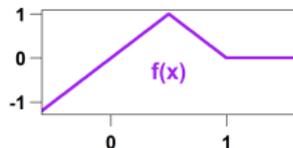
```
if cond1 and cond2: print("tt")
elif cond1: print("tF")
elif cond2: print("Ft")
else: print("FF")
```

Bedingte Codeausführung - Mathe-Beispiel

$$f_n(x) = \begin{cases} 2n^2x & , 0 \leq x \leq \frac{1}{2n} \\ n-2n^2(x-\frac{1}{2n}) & , \frac{1}{2n} \leq x \leq \frac{1}{n} \\ 0 & , \frac{1}{n} \leq x \leq 1 \end{cases}$$

Quelle mit $n=1$ & `inf` als Grenzen:

$$f(x) = \begin{cases} 2x & \text{wenn } x < 0.5 \\ 1-2(x-0.5) & \text{wenn } 0.5 \leq x \leq 1 \\ 0 & \text{wenn } x > 1 \end{cases}$$



```
x = -0.2
if x > 1:
    y = 0
elif x >= 0.5:
    y = 2-2*x
else:
    y = 2*x
print(y)
## -0.4
```

```
username = input('Gib deinen Nutzernamen ein: ')\npassword = input('Gib dein Passwort ein: ')
```

Gib `Login erfolgreich` aus für den Nutzer admin mit Passwort 123456, sonst `Nutzername oder Passwort sind falsch`

```
if username == 'admin' and password == '123456':\n    print('Login erfolgreich')\nelse:\n    print('Nutzername oder Passwort sind falsch')
```

Ein Schaltjahr ist ein Kalenderjahr, das einen zusätzlichen Tag (29. Februar) enthält, um im Gleichschritt mit dem astronomischen Jahr zu bleiben. Diese zusätzlichen Tage treten in jedem Jahr auf, welches ein Vielfaches von 4 ist, außer den Jahren, welche Vielfache von 100 sind, es sei denn, sie sind auch durch 400 teilbar.

Finde heraus, ob ein Jahr ein Schaltjahr ist.

Der Modulo-Operator ist `%`.

Das Jahr sollte durch 4 teilbar sein und (nicht teilbar durch 100 oder teilbar durch 400)

```
jahr = int(input('Gib das Jahr ein: '))
ist_schalt = jahr%4==0 and jahr%100!=0 or jahr%400==0
print(ist_schalt)
```

Logik und Bedingte Codeausführung:

- ▶ Vergleichsoperatoren, `min < value < max`, `not`, `and`, `or`
- ▶ `&` und `^` sind bitweise Operatoren, nicht verwenden (außer das ist beabsichtigt)
- ▶ Struktur bedingten Codes:

```
if bedingung1:  
    ausdrück1  
elif bedingung2:  
    ausdrück2  
else:  
    ausdrück3
```

- ▶ `all`, `any`, `sum`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 3.1 Bedingte Codeausführung
 - 3.2 For- und While-Schleifen
 - 3.3 List comprehension

Code mehrfach ausführen, jeweils mit einem anderen Wert

Nachfolgender Code hat viele Dopplungen:

```
print(list(range(1,4)))  
print(list(range(1,5)))  
print(list(range(1,9)))
```

Das geht einfacher mit einer Schleife:

```
for ende in [4,5,9]:  
    print(list(range(1,ende)))  
## [1, 2, 3]  
## [1, 2, 3, 4]  
## [1, 2, 3, 4, 5, 6, 7, 8]
```

```
for variable in werte_liste :  
    mach_etwas_mit(variable)  
# Doppelpunkt (:) wird benötigt  
# Einrückung ist wichtig
```

```
while bedingung_erfuellt:  
    fuehre_dinge_aus_die_ggf_die_bedingung_aendern()  
    if diesmal_fertig:  
        continue # springe zur nächsten Iteration  
    if ganz_fertig:  
        break # brich die Schleife ab  
nur_ausfuehren_wenn_beide_FERTIGs_falsch_sind()
```

Konvention für eine nicht verwendete Indexvariable:

```
for _ in range(3):  
    print("kram") # -> kram kram kram
```

for-loop print-Beispiel 1

```
for zahl in (0,1,2,3):  
    print(zahl)  
## 0  
## 1  
## 2  
## 3
```

```
for zahl in range(4):  
    print(zahl)  
## 0  
## 1  
## 2  
## 3
```

```
list( range(8, 0, -2) ) # range Ende exklusiv  
## [8, 6, 4, 2]
```

for-loop print-Beispiel 2

Zeige alle Objektnamen eines Moduls auf einer jeweils eigenen Zeile an:

```
import math # eingebautes Modul (Paket)
for f in dir(math):
    print(f)
## __doc__
## ... # manuell ausgewählte Ausgabe
## __name__
## __package__
## acos
## ceil
## exp
## factorial
## gamma
## inf
## isnan
## log10
## pi
## pow
## sin
## sqrt
```

for-Schleife Maximum-Beispiel

```
werte = [923,790,447,617,534,93,895,60,21,  
         962,992,302,435,902,795,482]
```

Stell dir vor, die Funktion `max` wäre nicht verfügbar.
Bestimme den größten Wert mithilfe einer Schleife.

```
maxi = 0  
for w in werte:  
    if w > maxi:  
        maxi = w  
print(maxi)  
## 992
```

Wähle alle geraden Zahlen aus einer Liste aus.

```
zahlen = [468, 976, 701, 269, 841, 7, 917, 698, 689, 526, 307, 791, 718]
```

```
gerade = [] # leere Liste initiieren
```

```
for n in zahlen:  
    if n%2==0:  
        gerade.append(n)
```

```
gerade  
## [468, 976, 698, 526, 718]
```

Mehrere Objekte in einer Codezeile (siehe [2.1 Objekttypen](#))

```
for a,b in [(1,11), (2,22), (3,33), (4,44)]:  
    print("a:", a, " b:", b, "-> Ergebnis:", b-2*a)  
## a: 1  b: 11 -> Ergebnis: 9  
## a: 2  b: 22 -> Ergebnis: 18  
## a: 3  b: 33 -> Ergebnis: 27  
## a: 4  b: 44 -> Ergebnis: 36
```

So lange das `geld` ausreicht, Sachen kaufen (und Restbetrag anzeigen)

```
geld = 53
while geld > 10:
    print(geld)
    geld -= 10
print("finaler Wert:", geld)
## 53
## 43
## 33
## 23
## 13
## finaler Wert: 3
```

while-loop input-Beispiel

wiederholt eine Zahl per `input` abfragen, bis sie korrekt erraten wurde

```
zahl = 0
while zahl != 42 :
    zahl = input("Rate eine Zahl: ")
    zahl = int(zahl)
    if zahl==42:
        print("Du hast die Antwort (auf alles) gefunden.")
    elif zahl > 42:      # print(..., flush=True) in Rstudio
        print(f"Tut mir leid, {zahl} ist zu groß.")
    else:
        print(f"Tut mir leid, {zahl} ist zu klein.")
## Rate eine Zahl: 78
## Tut mir leid, 78 ist zu groß.
## Rate eine Zahl: 31
## Tut mir leid, 31 ist zu klein.
## Rate eine Zahl: 42
## Du hast die Antwort (auf alles) gefunden.
```

```
while(True):
    zahl = input("Rate eine Zahl: ")
    zahl = int(zahl)
    if zahl==42:
        print("Du hast die Antwort (auf alles) gefunden.")
        break
    elif zahl > 42:
        print(f"Tut mir leid, {zahl} ist zu groß.")
    else:
        print(f"Tut mir leid, {zahl} ist zu klein.")
```

`while(True)` beginnt eine Schleife, die manuell beendet werden muss (mittels `break`, in einer Funktion geht auch `return`)

Schleifensteuerung (control commands) Beispiele

```
for buchstabe in 'Python': # strings sind iterierbar :)
    if buchstabe == 'h':
        continue # überspringe den Rest einer Ausführung
    print("Aktuell: " + buchstabe)
## Aktuell: P
## Aktuell: y
## Aktuell: t
## Aktuell: o
## Aktuell: n
```

```
for buchstabe in 'Python':
    if buchstabe == 'h':
        break # die Schleife vollständig abbrechen
    print("Aktuell: " + buchstabe)
## Aktuell: P
## Aktuell: y
## Aktuell: t
```

Abbildung (mapping)

```
buchstabenliste = ["abcdef", "ab", "abc"]
buchstabenlaenge = []
for b in buchstabenliste:
    buchstabenlaenge.append(len(b))
buchstabenlaenge
## [6, 2, 3]
```

```
b_laenge = map(len, buchstabenliste)
b_laenge
## <map object at 0x000001E51057F760>
list(b_laenge)
## [6, 2, 3]
```

```
list(map(len, buchstabenliste))
# ist viel kürzer als die Schleife oben :)
```

Objekte in Schleifen ändern 1

```
ids = ["a1", "b1", "b2", "b3", "b4", "b5", "b6", "c1", "d1"]
# Entferne Werte, die mit b starten
for v in ids:
    if v[0]=="b": ids.remove(v)
ids
## ['a1', 'b2', 'b4', 'b6', 'c1', 'd1']
```

Warum ist die Hälfte der b's noch drin?

Nach der zweiten Iteration ist ids `a1, b2, b3, b4, b5, b6, c1, d1`, weil "b1" entfernt wurde. In der dritten Iteration **wertet Python 'ids' wieder aus** und gibt das dritte Element an `v` ("b3"). Dieses wird wieder entfernt, also verbleiben `a1, b2, b4, b5, b6, c1, d1`. Das wiederholt sich, bis wir bei `a1, b2, b4, b6, c1, d1` ankommen. Ein Index hilft nicht: `for i in range(len(ids))` gibt einen `IndexError`.

Wie können wir korrekt in einer Schleife die Werte auswählen, die nicht mit b beginnen?

Objekte in Schleifen ändern 2

```
ids = ["a1", "b1", "b2", "b3", "b4", "b5", "b6", "c1", "d1"]
ausgabe = []
for v in ids:
    if v[0]!="b": ausgabe.append(v)
print(ausgabe)
## ['a1', 'c1', 'd1']
```

Das Iteratorobjekt in einer Schleife darf nicht verändert werden!

Unerwartete Dinge: **Iteration in Python**

Es gibt hier eine Alternative, die (scheinbar) ohne Schleife auskommt:

```
def beginnt_nicht_mit_b(x):
    return x[0]!="b"
list(filter(beginnt_nicht_mit_b, ids))
## ['a1', 'c1', 'd1']
```

```
list(filter(lambda x : x[0]!="b", ids))
## ['a1', 'c1', 'd1']
```

for- und while-Schleifen :

- ▶ Schleifenstruktur:

```
for var in liste:  
    ausdruck(var)
```

- ▶ `continue` , `break`
- ▶ `for _ in range(n):` für ungenutzte Variable
- ▶ `for x,y in [(x1,y1),(x2,y2)]:`
- ▶ `list(map(funktion, liste))`
- ▶ `list(filter(funktion, liste))`
- ▶ Verändere nicht das Iterator-Objekt während Iteration
- ▶ Im echten Leben: `map/filter`, `list comprehension`, `pandas`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 3.1 Bedingte Codeausführung
 - 3.2 For- und While-Schleifen
 - 3.3 List comprehension

List comprehension (Codelänge verringern) - Beispiel 1

```
def tu_etwas_mit(x):  
    return x + 5  
einige_zahlen = [6, 9, 17, -2, 24]
```

```
ergebnis = []  
for zahl in einige_zahlen:  
    neue_zahl = tu_etwas_mit(zahl)  
    ergebnis.append(neue_zahl)  
ergebnis  
## [11, 14, 22, 3, 29]
```

```
ergebnis = [tu_etwas_mit(zahl) for zahl in einige_zahlen]  
ergebnis  
## [11, 14, 22, 3, 29]
```

```
list(map(tu_etwas_mit, einige_zahlen))  
## [11, 14, 22, 3, 29]
```

List comprehension (Codelänge verringern) - Beispiel 2

```
namen = ["Alex", "Berry", "Beth", "Chris", "Dave"]
```

```
mitB = []  
for wort in namen:  
    if wort[0] == "B":  
        mitB.append(wort)  
mitB  
## ['Berry', 'Beth']
```

```
mitB = [wort for wort in namen if wort[0] == "B"]  
mitB  
## ['Berry', 'Beth']
```

Dictionary comprehension funktioniert auch so

Gib die Buchstabenanzahl jedes Wortes aus:

```
nachricht = "du steigerst dich"  
{wort:len(wort) for wort in nachricht.split()}  
## {'du': 2, 'steigerst': 9, 'dich': 4}
```

Verdopple jeden Wert im Dictionary:

```
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}  
doppel_dict1 = {k:v*2 for (k,v) in dict1.items()}
```

Wähle alle geraden Einträge, die größer als 2 sind:

```
{k:v for (k,v) in dict1.items() if v>2 and v%2==0}  
## {'d': 4, 'f': 6}
```

Wandle von Grad Fahrenheit nach Celcius um:

```
dF = {'t1': -30, 't2': -20, 't3': -10, 't4': 0}  
dC = {k:5/9*float(v-32) for (k,v) in dF.items()}  
dC = {k+"C":round(v) for (k,v) in dC.items()}  
dC  
## {'t1C': -34, 't2C': -29, 't3C': -23, 't4C': -18}
```

List comprehension Übung

Kürze alle drei Schleifen auf eine einzelne Codezeile.

```
werte = [11,10,2,3,15,3,5,7,7,2,8,7,5,6,5,8,5,9,6,3,15,6,9]
quadratsumme = 0
for v in werte:
    quadratsumme += v**2
quadratsumme
## 1351
```

```
zahlen = [951,402,984,651,360,69,408,319,601,485,980,507,725,
547,544,615,83,165,141,501,263,617,865,575,219,390,984,592,236,
105,942,941,386,462,47,418,907,344]
gerade_zahlen = []
for n in zahlen:
    if n%2==0: gerade_zahlen.append(n)
gerade_zahlen
## [402, 984, 360, 408, 980, 544, 390, 984, 592, 236, 942, 386, 462, 418, 344]
```

```
import random
max_exp = []
for i in range(50):
    max_exp.append(random.expovariate(0.2))
max_exp = max(max_exp)
max_exp
## 20.343037401675073
```

```
werte = [11,10,2,3,15,3,5,7,7,2,8,7,5,6,5,8,5,9,6,3,15,6,9]
sum([v**2 for v in werte])
## 1351
```

```
zahlen = [951,402,984,651,360,69,408,319,601,485,980,507,725,
547,544,615,83,165,141,501,263,617,865,575,219,390,984,592,236,
105,942,941,386,462,47,418,907,344]
[n for n in zahlen if n%2==0]
## [402, 984, 360, 408, 980, 544, 390, 984, 592, 236, 942, 386, 462, 418, 344]
```

```
import random
max([random.expovariate(0.2) for _ in range(50)])
## 19.221103122473906
```

Schleifen-code deutlich verkürzen:

- ▶ Listen- und Dict comprehension
- ▶ `[tu_etwas_mit(x) for x in liste if bedingung(x)]`
- ▶ `{k:tu_etwas_mit(v) for (k,v) in eine_dict.items()}`

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
2. Objekte
3. Schleifen
4. Programmieren

- 4.1 Fehlermanagement
- 4.2 Eigene Klassen schreiben
- 4.3 Unit Tests

Fehlermanagement: try - except

Bei einem Fehler bricht die komplette Programmausführung ab.

Beispiel: **TypeError**: falscher Datentyp für Operator oder Funktion

```
ergebnis_einer_berechnung = "2"
```

```
7 + ergebnis_einer_berechnung
## TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

Python kann Code aber auch probeweise ausführen, und im Fall eines Fehlers was anderes machen als abbrechen.

```
try:
    7 + ergebnis_einer_berechnung
except TypeError:
    print("Zeichenkette und Nummer gemischt")
## Zeichenkette und Nummer gemischt
```

Beachte die Einrückung, wie mit allen Python Kontrollstrukturen

```
try:
    7 + nichtExistierendesObjekt
except TypeError:
    print("Zeichenkette und Nummer gemischt")
except:
    print("Ein Fehler ist aufgetreten")
## Ein Fehler ist aufgetreten
```

```
try:
    7 + "2" # Code mit möglichen Fehlern
except:
    print("Code fehlgeschlagen")
else:
    print("Code erfolgreich ausgeführt")
## Code fehlgeschlagen
```

Der `else` Code wird ausgeführt, wenn keine Fehler auftreten. Könnte auch im `try` Teil sein, man sollte aber den potentiellen Fehler und den Umgang damit nah beieinander behalten. Fange nur erwartete Fehler ab (andere Fehler sollten weiterhin auftreten).

`else` wird vor `finally` ausgeführt (nächste Folie).

```
try:
    7 + "2" # Code mit möglichen Fehlern
except:
    print("Code fehlgeschlagen")
finally:
    print("Programm fertig")
## Code fehlgeschlagen
## Programm fertig
```

Code in `finally` wird ausgeführt, selbst wenn `return` / `break` / `continue` aufgerufen wird oder ein anderer (nicht abgefangener) Fehler auftritt.

traceback gibt aus, woher der Fehler kommt:

```
import traceback
try:
    7 + nichtExistierendesObjekt
except:
    print("Fehler aufgetreten:", traceback.format_exc())
## Fehler aufgetreten:  Traceback (most recent call last):
## File "<string>", line 2, in <module> ## NameError:
name 'nichtExistierendesObjekt' is not defined
```

informativer in echter Anwendung (Folienstruktur kann kein traceback).

IDEs mit Debugger stellen oft tracebacks für Fehler zur Verfügung

Fehler mit benutzerdefiniertem Präfix protokollieren

```
def addiere7_mit_print_statt_fehler(x):  
    try:  
        return x + 7  
    except Exception as e: # Fehlermeldung als Variable 'e'  
        print("Ein Fehler ist aufgetreten:", e, sep="\n")
```

```
addiere7_mit_print_statt_fehler(3)  
## 10  
addiere7_mit_print_statt_fehler("3")  
## Ein Fehler ist aufgetreten:  
## can only concatenate str (not "int") to str  
addiere7_mit_print_statt_fehler(None)  
## Ein Fehler ist aufgetreten:  
## unsupported operand type(s) for +: 'NoneType' and 'int'
```

```
def fehler_mit_ganzer_nachricht(x): # siehe auch
    try:                             # sys.exc_info()
        x + 7
    except Exception as e:
        print(f"Ein {type(e).__name__} ist passiert:\n{e}")
```

```
fehler_mit_ganzer_nachricht(3)
fehler_mit_ganzer_nachricht(None)
## Ein TypeError ist passiert:
## unsupported operand type(s) for +: 'NoneType' and 'int'
fehler_mit_ganzer_nachricht("3")
## Ein TypeError ist passiert:
## can only concatenate str (not "int") to str
fehler_mit_ganzer_nachricht(dummyvar)
## NameError: name 'dummyvar' is not defined
```

Nachrichten mit Zeitstempel, nützlich für Logging (protokollieren)

```
import time
jetzt = time.strftime("%Y-%m-%d %H:%M UTC", time.gmtime())
print(jetzt)
## 2024-10-15 08:25 UTC
```

```
def fehler_mit_zeitstempel(x):
    try:
        x + 7
    except:
        n = time.strftime("%Y-%m-%d %H:%M UTC", time.gmtime())
        print("Ein Fehler ist aufgetreten am:", n)
```

```
fehler_mit_zeitstempel(3)
fehler_mit_zeitstempel(None)
## Ein Fehler ist aufgetreten am: 2024-10-15 08:25 UTC
```

```
preis = None
while not isinstance(preis, int):
    try:
        preis = int(input("Gib einen Preis ein: "))
    except ValueError:
        print("Bitte gib eine gültige Nummer ein.")
print("Die eingegebene Nummer war: ", preis)
```

Zusammenfassung für 4.1 Fehlermanagement

Exceptions abfangen und verwalten:

► Struktur

```
try:
    code_mit_moeglichen_Fehlern
except BestimmterError: # möglichst nur konkrete
    was_jetzt_zu_tun_ist # Fehler abfangen!
except Exception as e:
    mach_etwas_damit(e) # z.B. Logging
    traceback.format_exc()
    time.strftime("%Y-%m-%d %H:%M UTC", time.gmtime())
except: # falls Excpetion as e
    andere_Fehler_behandeln # nicht genutzt wurde
else:
    was_zu_tun_ist_falls_BestimmterError_nicht_passiert
finally:
    tu_das_immer_am_Ende
```

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 4.1 Fehlermanagement
 - 4.2 Eigene Klassen schreiben
 - 4.3 Unit Tests

Ein paar Definitionen:

Objekt: Sammlung von Daten (Variablen) und Methoden (Funktionen) die auf Basis dieser Daten operieren. (Attribute + Verhalten)

Klasse: Vorlage für Objekte

Instanz: spezifisches Objekt einer Klasse

```
class Person:  
    pass
```

Python verbietet leere Körper in Klassen. Das `pass` Statement dient als Platzhalter für Code

```
p1 = Person() # erstellt Instanz der Klasse Person  
p1.name = "Berry" ; p1.alter = 32 # Attribute hinzufügen  
p1  
## <__main__.Person object at 0x000001EFD04EB9D0>  
p1.__dict__ # Dictionary mit allen Attributen  
## {'name': 'Berry', 'alter': 32}
```

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

```
person1 = Person('Berry', 23)    # konstruiert neue Person
person2 = Person('Christina', 16)
person1.name
## 'Berry'
```

__init__: spezielle Funktion die beim Erstellen des Objekts aufgerufen wird und die Attribute initialisiert. Sie wirkt wie ein **Konstruktor**.

self: steht für die Instanz der Klasse, mit der wir die Methode aufrufen. Es ist das erste Argument von Methoden wie `__init__`.

Objekte `person1` und `person2` haben eigene Attribute (name, alter). Mit dem self Argument können die Methoden auf Attribute der spezifischen Instanz zugreifen.

Methoden einer benutzerdefinierten Klasse

eine Methode, die überprüft ob eine Person Horrorfilme schauen darf:

```
class Person:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter
    def darf_horrorfilm_sehen(self): # diese Funktion
        if self.alter >= 18:         # ist eine Methode
            return "los gehts!"     # für alle Objekte
        else:                        # der Klasse 'Person'
            return "sorry, zu jung"
```

```
person1 = Person('Berry', 23)
person1.darf_horrorfilm_sehen()
## 'los gehts!'

person2 = Person('Christina', 16)
person2.darf_horrorfilm_sehen()
## 'sorry, zu jung'
```

Wie kann man die Methode `darf_horrorfilm_sehen` vereinfachen, sodass sie True / False zurückgibt?

```
def darf_horrorfilm_sehen(self):  
    return self.alter >= 18
```

Weiterführende Materialien:

Dr Philip Yip

Python101

Pynative

```
class Patient:
    def __init__(self, pid, geschlecht, BD):
        self.pid = pid
        self.geschlecht = geschlecht
        self.BD = BD
        if geschlecht not in ["m", "w", "d"]:
            raise ValueError("geschlecht muss m/w/d sein")
    def hat_bluthochdruck(self):
        return self.BD > 130
```

```
Patient(pid="Pat456", geschlecht="Divers", BD=120)
## ValueError: geschlecht muss m/w/d sein
```

```
armer_kerl = Patient("Pat123", "m", 113)
armer_kerl.BD = 150
print("jetzt behandeln:", armer_kerl.hat_bluthochdruck())
## jetzt behandeln: True
```

```
class Student:
    anzahl = 0 # Klassenvariable
    def __init__(self, name):
        self.name = name
        Student.anzahl += 1 # bei jeder Erstellung + 1
```

```
s1 = Student("Anna Lena")
s2 = Student("Christina")
Student("Berry")
```

```
Student.anzahl
## 3
```

```
s2.anzahl # Instanzvariable greift auf Klassenvariable zu
## 3
s2.anzahl = 1000
```

```
Student.anzahl # unabhängig einer einzelnen Instanz
## 3
```

Klassen (Aufgabe + Teillösung)

Hérons Formel ergibt die Fläche eines Dreiecks, wenn die Länge aller drei Seiten bekannt ist. Eine Seite eines Dreiecks kann nicht länger sein als die Summe der beiden Anderen. Schreibe Code, der die Fläche ausgibt - oder "Kein Dreieck", wenn eine Seite zu lang ist.

halbperimeter `s` = $(a+b+c)/2$ flaeche `a` = $\text{sqrt}(s(s-a)(s-b)(s-c))$

```
a = 10
b = 4
c = 5
if a+b > c and a+c > b and b+c > a:
    s = (a+b+c)/2
    flaeche = (s*(s-a)*(s-b)*(s-c))**0.5   ***0.5 = sqrt
    f"Flaeche des Dreiecks: {flaeche}"
else:
    "Kein Dreieck"
```

Schreibe eine `Dreieck` Klasse, die einen Fehler für ungültige Dreiecke erzeugt und eine Flächenmethode hat.

```
class Dreieck:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
        if not(a+b > c and a+c > b and b+c > a):
            raise ValueError('ungültig: zu lange Seite')
    def flaeche(self):
        p = (self.a+self.b+self.c)/2
        flaeche = (p*(p-self.a)*(p-self.b)*(p-self.c))**0.5
        return flaeche
```

```
eck = Dreieck(a=3, b=4, c=5)
eck.__dict__ # Attribute ausgeben
## {'a': 3, 'b': 4, 'c': 5}
eck.flaeche()
## 6.0
```

Benutzerdefinierte Klassen:

- ▶ generelle Struktur:

```
class MeineKlasse:
    classVariable = "wert"
    def __init__(self, args):
        self.args = args
        if not gueltig(args): raise SomeError("info")
    def eineMethode(self):
        return self.args

eineInstanz = MeineKlasse("werte")
eineInstanz.eineMethode()
```

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.

1. Intro
 2. Objekte
 3. Schleifen
 4. Programmieren
- 4.1 Fehlermanagement
 - 4.2 Eigene Klassen schreiben
 - 4.3 Unit Tests

- ▶ Beim Programmieren können unbemerkt Fehler entstehen und sich im Code manifestieren.
- ▶ Diese Fehler können zu unerwartetem Verhalten oder gar einem Programmabbruch führen.
- ▶ Manuelles Ausprobieren aller möglichen Fälle wird zunehmend aufwändig und unübersichtlich.
- ▶ Daher: automatisiertes Testen

- ▶ Statische Testverfahren (= ohne Codeausführung)
 - ▶ Code-Reviews (durch Kolleg:innen, neuerdings durch KI)
 - ▶ Quellcodeanalyse (z.B. PyLint)

```
def addiere(a, b):  
    summe = a + b  
    return a + b  
  
## warning (W0612, unused-variable, addiere) Unused  
variable 'summe'
```

- ▶ Dynamische Testverfahren (= mit Codeausführung)
 - ▶ Unit Tests (z.B. mit PyUnit)
 - ▶ Integration Tests (z.B. mit PyUnit)
 - ▶ System Tests (z.B. Selenium für Webanwendungen)
 - ▶ Acceptance Tests (z.B. Selenium, Robot Framework)

Beispiel - "zuletzt gesehen"

Eine Chatanwendung soll anzeigen, wann eine Person zuletzt online war. Die Anzeige soll in Minuten / Stunden / Tagen erfolgen.

Skript `onlinestatus.py` mit folgender naiver Umsetzung:

```
def zuletzt_gesehen(sek):  
    if sek == 0:  
        return "Online"  
    elif sek < 60:  
        return "Zuletzt vor weniger als 1 Minute gesehen"  
    elif sek < 60*60:  
        return f"Zuletzt vor {sek // 60} Minuten gesehen"  
    elif sek < 60*60*24:  
        return f"Zuletzt vor {sek // (60*60)} Stunden gesehen"  
    else:  
        return f"Zuletzt vor {sek // (60*60*24)} Tagen gesehen"
```

Unit Test – "zuletzt gesehen"

Skript `test_onlinestatus.py` mit:

```
import unittest
from onlinestatus import zuletzt_gesehen
class ZuletztGesehenTest(unittest.TestCase):
    def test_online(self):
        tatsaechlich = zuletzt_gesehen(0)
        erwartet = "Online"
        self.assertEqual(tatsaechlich, erwartet)
```

Test ausführen - Option 1: Am Ende von `test_onlinestatus.py`:

```
if __name__ == "__main__":    unittest.main()
```

Test ausführen - Option 2: leichter automatisierbar

```
python -m unittest test_onlinestatus.py # Mac: python3
## -----
## Ran 1 test in 0.000s
## OK
```

Mehr Tests!

```
import unittest
from onlinestatus import zuletzt_gesehen
class ZuletztGesehenTest(unittest.TestCase):
    def test_online(self):
        self.assertEqual(zuletzt_gesehen(0), "Online")
    def test_eine_sekunde(self):
        self.assertEqual(zuletzt_gesehen(1),
            "Zuletzt vor weniger als 1 Minute gesehen")
    def test_weniger_als_eine_minute(self):
        self.assertEqual(zuletzt_gesehen(59),
            "Zuletzt vor weniger als 1 Minute gesehen")
    def test_eine_minute(self):
        self.assertEqual(zuletzt_gesehen(60),
            "Zuletzt vor 1 Minute gesehen")
    def test_weniger_als_zwei_minuten(self):
        self.assertEqual(zuletzt_gesehen(119),
            "Zuletzt vor 1 Minute gesehen")
    def test_zwei_minuten(self):
        self.assertEqual(zuletzt_gesehen(120),
            "Zuletzt vor 2 Minuten gesehen")
```

```
python -m unittest test_onlinestatus.py

## F...F.
## =====
## FAIL: test_eine_minute (test_onlinestatus.ZuletztGesehenTest)
## -----
## Traceback (most recent call last):
##   File "test_onlinestatus.py", line 11, in test_eine_minute
##     self.assertEqual(zuletzt_gesehen(60), "Zuletzt vor 1 Minute gesehen")
## AssertionError: 'Zuletzt vor 1 Minuten gesehen' !=
##                                     'Zuletzt vor 1 Minute gesehen'
## - Zuletzt vor 1 Minuten gesehen
## ?                               -
## + Zuletzt vor 1 Minute gesehen
##
## [...]
##
## -----
## Ran 6 tests in 0.001s
##
## FAILED (failures=2)
```

Vorteile von Tests

- ▶ Gesicherte Funktionsweise: Korrektheit der getesteten Einheiten bereits während der Entwicklung prüfen
- ▶ Weniger Fehler: Unerwünschte Seiteneffekte früh finden
- ▶ Unterstützung beim Refactoring: Quellcode ohne Angst ändern
- ▶ Reflektierte Strukturierung: Code Design gut überlegen
- ▶ Zufriedenheit bei der Entwicklung: dem Ergebnis vertrauen und manuelles Testen vermeiden
- ▶ Code testgetrieben entwickeln (TDD, Test-driven development):
 - ▶ Zuerst den Test schreiben (`test_onlinestatus.py`)
 - ▶ Danach die eigentliche Implementierung (`onlinestatus.py`)

Kurzfristig: die Entwicklung verlangsamt sich

Langfristig: der Code wird stabiler und die Struktur verbessert

Zusammenfassung für 4.3 Unit Tests

Code systematisch prüfen:

- ▶ abgeschlossene Einheiten (Units) auf korrekte Funktion überprüfen

```
import unittest # Modul importieren
from datei import funktion # Code importieren
class FunktionsTest(unittest.TestCase):
    def test_rueckgabe(self): # Testfall definieren
        self.assertEqual(funktion(0), "Sollwert")
if __name__ == "__main__":
    unittest.main()
```

- ▶ Für die Test-Klasse immer `(unittest.TestCase)` angeben
- ▶ Jede Test-Methode mit `test_` beginnen und `(self)` angeben
- ▶ Weiterführendes Material: Beispiele **unterschiedlicher Testverfahren**, **Unittest-Dokumentation** (English), **assert-Methoden**

Melde unklare Aufgaben im Forum.

Markiere die Inhalte dieser Lektion in deiner RefCard.