Datenverarbeitung

Einige praktische Operationen

Praktische Operationen (1/2)

```
# 1. Einfache Aggregationen
zahlen = [1, 5, 3, 2, 4]
print(sum(zahlen))     # 15
print(max(zahlen))     # 5
print(min(zahlen))     # 1



# 2. Range mit Start, Stop, Step
print(list(range(1, 10, 2)))  # [1, 3, 5, 7, 9]

# Vergleich mit Slice-Notation:
zahlen[1:10:2]  # Gleiches Muster!
```

Praktische Operationen (2/2)

```python
# 3. Index + Element mit enumerate()
spieler = ["Anna", "Bob", "Charlie"]
for i, name in enumerate(spieler):
    print(f"Spieler {i+1}: {name}")



# 4. Parallel iteration mit zip()
punkte = [95, 82, 88]
for name, punkt in zip(spieler, punkte):
    print(f"{name}: {punkt}")

# Ausgabe:
# Anna: 95
# Bob: 82
# Charlie: 88
```

Hauptteil

Elegante Datenverarbeitung

Traditionelle vs. elegante Datenverarbeitung

Verschiedene Wege zum Ziel; bisher immer imperativ programmiert; deklarative Programmierung ist angenehmer zu lesen; *pythonic code* ist lesbar und elegant.

```python
numbers = [1, 2, 3, 4, 5]


# Traditioneller Weg: umständlich und "verbose"
quadrate = []
for n in numbers:
    quadrate.append(n * n)


# Eleganter Weg
quadrate = [n * n for n in numbers]
```

Motivation sich für Eleganz einzusetzen (1/2)

Code sollte sich wie natürliche Sprache lesen, dadurch klarer und besser wartbar…

```python
spieler_punkte = [12, 45, 23, 89, 34]


# Kompliziert zu lesen und zu warten
ergebnis = []
for p in spieler_punkte:
    if p > 30:
        ergebnis.append(p * 2)


# Besser, da kompakt, klar und präzise
ergebnis = [p * 2 for p in spieler_punkte if p > 30]
```

Motivation sich für Eleganz einzusetzen (2/2)

… und einfacher vorhersagbar und testbar.

```
# Schwer zu testen - abhängig von globalem Zustand
gesamt_punkte = 0
def punkte_addieren(neue_punkte):
    global gesamt_punkte
    gesamt_punkte += neue_punkte
    return gesamt_punkte


# Einfach zu testen - nur Input/Output
def berechne_gesamt(alte_punkte, neue_punkte):
    return alte_punkte + neue_punkte
```

Motivation sich für Eleganz einzusetzen (1/2)

Code sollte sich wie natürliche Sprache lesen, dadurch klarer und besser wartbar…

```python
spieler_punkte = [12, 45, 23, 89, 34]


# Kompliziert zu lesen und zu warten
ergebnis = []
for p in spieler_punkte:
    if p > 30:
        ergebnis.append(p * 2)


# Besser, da kompakt, klar und präzise
ergebnis = [p * 2 for p in spieler_punkte if p > 30]
```

Wichtiges Konzept: Seiteneffekte bzw. Freiheit von Seiteneffekten (1/2)

```python
# Beispiel 1: Funktion mit Seiteneffekten
highscore = 0
def update_highscore(punkte):
    global highscore          # Achtung: Verändert globalen Zustand!
    if punkte > highscore:
        highscore = punkte    # Seiteneffekt: Ändert Variable außerhalb
    return highscore


# Beispiel 2: "Reine" Funktion - vorhersagbar und testbar
def get_new_highscore(alter_highscore, neue_punkte):
    # Nur Input -> Output
    return max(alter_highscore, neue_punkte)
```

Wichtiges Konzept: Seiteneffekte bzw. Freiheit von Seiteneffekten (2/2)

```
# Beispiel 3: Sehr häufig in der Praxis - In-Place Modifikation
def sortiere_liste(numbers):
    numbers.sort()           # Verändert die Liste direkt!
    return numbers


# Beispiel 4: Reine Alternative
def get_sorted(numbers):
    return sorted(numbers)   # Erstellt neue, sortierte Liste
```

Zusammenfassung: Seiteneffekte bzw. Freiheit von Seiteneffekten

Reine Funktionen (pure functions)

- Arbeiten nur mit ihren Eingabeparametern
- Gleicher Input → immer gleicher Output
- Keine „versteckten“ Änderungen

Seiteneffekte

- Änderungen außerhalb der Funktion
- z.B. globale Variablen, Datenbankzugriffe, Netzwerk
- Machen Code schwerer nachvollziehbar

Wichtiger Baustein: Funktionen als Werte (1/2)

```
# Zwei Funktionen, die Strings verarbeiten
def grossbuchstaben(text):
    return text.upper()

def kleinbuchstaben(text):
    return text.lower()

# Eine Funktion, die eine andere Funktion nutzt
def verarbeite_name(name, format_funktion):
    formatiert = format_funktion(name)    # Übergebene Funktion wird ausgeführt
    return f"Verarbeitet: {formatiert}"

# Verwendung
name = "Max Mustermann”

# Funktion als Parameter
ergebnis1 = verarbeite_name(name, grossbuchstaben)

# Eine andere Funktion einsetzen ist jetzt einfach!
ergebnis2 = verarbeite_name(name, kleinbuchstaben)
```

Wichtiger Baustein: Funktionen als Werte (1/2)

```python
# Zwei Funktionen, die Strings verarbeiten
def grossbuchstaben(text):
    return text.upper()

def kleinbuchstaben(text):
    return text.lower()

# Eine Funktion, die eine andere Funktion nutzt
def verarbeite_name(name, format_funktion):
    formatiert = format_funktion(name)    # Übergebene Funktion wird ausgeführt
    return f"Verarbeitet: {formatiert}"

# Verwendung
name = "Max Mustermann”

# Funktion als Parameter
ergebnis1 = verarbeite_name(name, grossbuchstaben)

# Eine andere Funktion einsetzen ist jetzt einfach!
ergebnis2 = verarbeite_name(name, kleinbuchstaben)
```

Dieses Konzept ist verwandt mit dem bereits bekannten Prinzip **Inversion of Control**, das bei den Bausteinen für größere Programme erklärt wurde (*ExplosionHandler*).

Wichtiger Baustein: Funktionen als Werte (2/2)

Funktionen sind „first-class citizens“

```
# Funktionen in Variablen speichern
text = "Python ist cool"
text_processor = grossbuchstaben  # Funktion in Variable speichern
ergebnis = text_processor(text)   # Funktion über Variable aufrufen


# Liste von Funktionen
text_funktionen = [
    grossbuchstaben,
    kleinbuchstaben,
    lambda x: x.title(),         # Neue Funktion: Jedes Wort groß
]


# Alle Funktionen nacheinander anwenden
for funk in text_funktionen:
    print(funk(text))            # Jede Funktion wird aufgerufen
```

Wo brauchen wir das? Zum Beispiel beim Sortieren.

Zwei gängige Ansätze zum Sortieren in Python.

liste.sort()
- *sort!* – klingt wie ein Befehl
- Verändert Liste direkt (*in place*)
- Nur für Listen verfügbar
- Speichereffizienter

sorted(liste)
- *sorted* – ist ein Adjektiv
- Erstellt neue Liste (*non-destructive*)
- Original bleibt unverändert
- Funktioniert mit allen *Iterables*

Wo brauchen wir das? Zum Beispiel beim Sortieren.

```
# Liste sortieren - zwei Wege:
namen = ["Zoe", "Anna", "Yves", "Ben"]


# 1. sorted() - erstellt neue Liste
sortierte_namen = sorted(namen)    # namen bleibt unverändert
print(namen)          # ["Zoe", "Anna", "Yves", "Ben"]
print(sortierte_namen)  # ["Anna", "Ben", "Yves", "Zoe"]


# 2. sort() - verändert bestehende Liste
namen.sort()         # Verändert namen direkt
print(namen)         # ["Anna", "Ben", "Yves", "Zoe"]


# Beide erlauben key-Funktionen:
sortiert = sorted(namen, key=len)  # Nach Länge sortieren
namen.sort(key=str.lower)          # Nach Kleinschreibung
```

Map, Filter und Lambdas

Erste Begegnung mit map() – 1/3

```python
# Eine Liste von Namen
namen = ["anna", "ben", "charlie"]

# Bekannt: Traditionelle Schleife
gross = []
for name in namen:
    gross.append(name.upper())  # wir wollen alle Namen groß schreiben


# Ein Fall für map() - zunächst aber etwas gewöhnungsbedürftig
gross = map(str.upper, namen)
```

Erste Begegnung mit map() – 2/3

```python
# Eine Liste von Namen
namen = ["anna", "ben", "charlie"]

# Bekannt: Traditionelle Schleife
gross = []
for name in namen:
    gross.append(name.upper())  # wir wollen alle Namen groß schreiben


# Ein Fall für map() - zunächst aber etwas gewöhnungsbedürftig
gross = map(str.upper, namen)

# Moment... was kommt da zurück?
print(gross)  # <map object at 0x...>  🤔
```

Erste Begegnung mit map() – 3/3

```python
# Eine Liste von Namen
namen = ["anna", "ben", "charlie"]

# Bekannt: Traditionelle Schleife
gross = []
for name in namen:
    gross.append(name.upper())  # wir wollen alle Namen groß schreiben


# Ein Fall für map() - zunächst aber etwas gewöhnungsbedürftig
gross = map(str.upper, namen)

# Moment... was kommt da zurück?
print(gross)  # <map object at 0x...>  🤔

# Ah, wir brauchen list()!
gross = list(map(str.upper, namen))  # so passt es
print(gross)  # ['ANNA', 'BEN', 'CHARLIE']
```

map()

- Wendet eine Funktion auf jedes Element an
- Gibt zunächst nur ein map-Objekt zurück
- Braucht list() für echte Liste
- Speichereffizient bei großen Datenmengen

```
# Direktes Iterieren über map-Objekt
for name in map(str.upper, namen):
    print(name)          # Funktioniert ohne list()!
```

map()

- Wendet eine Funktion auf jedes Element an
- Gibt zunächst nur ein map-Objekt zurück
- Braucht list() für echte Liste
- Speichereffizient bei großen Datenmengen
- map-Objekt kann nur einmal iteriert werden

```
mapped = map(str.upper, namen)
for name in mapped:
    print(name, end=' ')  # => ANNA BEN CHARLIE
print()

for name in mapped:
    print(name, end=' ')  # => [keine Ausgabe]
```

Erste Begegnung mit filter() – 1/2

```
# Eine Liste von Zahlen
zahlen = [1, -4, 7, 0, -3, 12, -8, 5]

# Bekannt: Positive Zahlen finden mit Schleife
positiv = []
for z in zahlen:
    if z > 0:
        positiv.append(z)


# Ein Fall für filter() - gleiche Syntax wie map()

def ist_positiv(x):
    return x > 0

positiv = filter(ist_positiv, zahlen)    # Wieder ein Iterator!
positiv_liste = list(positiv)            # [1, 7, 12, 5]
```

Erste Begegnung mit filter() – 2/2

```
# Eine Liste von Zahlen
zahlen = [1, -4, 7, 0, -3, 12, -8, 5]

# Bekannt: Positive Zahlen finden mit Schleife
positiv = []
for z in zahlen:
    if z > 0:
        positiv.append(z)


# Ein Fall für filter() - gleiche Syntax wie map()

def ist_positiv(x):
    return x > 0

positiv = filter(ist_positiv, zahlen)    # Wieder ein Iterator!
positiv_liste = list(positiv)            # [1, 7, 12, 5]

# Oft verwendet: filter mit lambda
kurze_namen = filter(lambda x: len(x) < 4, namen)
```

filter()

- Behält nur Elemente, für die das Prädikat (die Funktion) True zurückgibt
- Wie map(): Gibt Iterator zurück
- Auch hier: *lazy evaluation*
- Häufig mit *Lambda-Funktionen* verwendet

Lambda-Funktionen

- λ (Lambda): Symbol aus der mathematischen Logik
- Alonzo Church entwickelt λ-Kalkül (1930er-Jahre)
- Mathematische Notation für eine Funktion: $\lambda x.x^2$
- In Python: kleine, anonyme Funktionen: `lambda x: x * x`

```python
# Herkömmliche Funktion
def quadrat(x):
    return x * x

# Lambda-Funktion - gleiche Funktionalität
quadrat = lambda x: x * x
```

Lambdas in der Praxis

```
# Beispiel 1: Lambda mit sorted()
teams = ["FC Bayern", "BVB", "RB Leipzig", "FC Schalke"]
nach_länge = sorted(teams, key=lambda x: len(x))


# Beispiel 2: Lambda mit filter()
nur_fc = list(filter(lambda x: x.startswith("FC"), teams))


# Beispiel 3: Lambda mit map()
kurzformen = list(map(lambda x: x[:3], teams))
                    # erste 3 Buchstaben pro Team
```

Wann keine Lambdas verwenden?

```python
# DON'T: Komplexe Logik
kompliziert = lambda x: (x * 2 if x > 0
                         else x * 3 if x < 0
                         else 42)  # Schwer zu lesen!
```

```python
# DO: Stattdessen normale Funktion
def verarbeite_zahl(x):
    if x > 0:
        return x * 2
    elif x < 0:
        return x * 3
    return 42
```

(List) Comprehensions

The Pythonic Way

```
# Tim Peters, Zen of Python:
# "Simple is better than complex"
# "Beautiful is better than ugly"


# Perl-Style-Code (kompliziert und schwer lesbar, write-only code, DON'T)
squares = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, range(10))))
```

The Pythonic Way

```python
# Tim Peters, Zen of Python:
# "Simple is better than complex"
# "Beautiful is better than ugly"


# Perl-Style-Code (kompliziert und schwer lesbar, write-only code, DON'T)
squares = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, range(10))))

# Der Pythonische Weg (elegant und klar, DO)
squares = [x**2 for x in range(10) if x % 2 == 0]
```

The Pythonic Way

```
# Tim Peters, Zen of Python:
# "Simple is better than complex"
# "Beautiful is better than ugly"


# Perl-Style-Code (kompliziert und schwer lesbar, write-only code, DON'T)
squares = list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, range(10))))

# Der Pythonische Weg (elegant und klar, DO)
squares = [x**2 for x in range(10) if x % 2 == 0]


# Weitere Beispiele:

namen = ["Anna", "Bob", "Charlie"]

# List Comprehension statt map()
gross = [name.upper() for name in namen]

# List Comprehension statt filter()
lang = [name for name in namen if len(name) > 3]
```

List Comprehensions: Syntax und Beispiele

```
# [ausdruck for element in sequenz if bedingung]
#  │            │          │       │
#  │            │          │       └── Optional: Filterkriterium
#  │            │          └── Quelldaten
#  │            └── Schleifenvariable
#  └── Was mit jedem Element passiert
```

```
# Beispiel 1: Zahlen verarbeiten

zahlen = [1, -2, 3, -4, 5]

absolutwerte = [abs(x) for x in zahlen]  # [1, 2, 3, 4, 5]

positive = [x for x in zahlen if x > 0]  # [1, 3, 5]
```

List Comprehensions: Syntax und Beispiele

```
# Beispiel 2: Strings verarbeiten

texte = ["Python", "Java", "C++", "JavaScript"]

kurznamen = [t[:3] for t in texte]  # ["Pyt", "Jav", "C++", "Jav"]

lang_mit_j = [t for t in texte
              if t.startswith("J") and len(t) > 3]  # ["Java", "JavaScript"]


# Beispiel 3: Datentransformation

personen = [("Anna", 25), ("Bob", 17)]

namen = [name for name, _ in personen]          # ["Anna", "Bob"]

erwachsene = [name for name, alter in personen
              if alter >= 18]                   # ["Anna"]
```

Set und Dict Comprehensions

```
# Set Comprehension: Geschweifte Klammern

zahlen = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]

quadrate = {x**2 for x in zahlen}  # {1, 4, 9, 16}
                                   # → Duplikate automatisch eliminiert!


# Dict Comprehension: Geschweifte Klaemmern mit Key-Value-Paaren

namen = ["Anna", "Bob", "Charlie"]

namenslaengen = {name: len(name)
                 for name in namen}  # {"Anna": 4, ...}


# Ein weiteres Beispiel

emails = ["anna@firma.de", "bob@firma.de"]
nutzer = {email: email.split("@")[0]
          for email in emails}  # {"anna@firma.de": "anna", ...}
```

Runde Klammern: Generator Expressions

```python
# List Comprehension verwendet [...]
quadrate_liste = [x**2 for x in range(10)]

# Set Comprehension verwendet {...}
quadrate_set = {x**2 for x in range(10)}


# Also mit runden Klammern...?
quadrate_tuple = (x**2 for x in range(10))
print(quadrate_tuple)  # <generator object ...>
                       # Ein "Generator" - kein Tuple!


# Erinnerung: Das kennen wir schon von map()

zahlen = map(str, range(10))  # <map object ...>
zahlen_gen = (str(x) for x in range(10))
                              # <generator object ...>
```

Generator Expressions sind ideal für große Daten (*Streaming*-Verarbeitung) – 1/2

```
# Speicherverbrauch demonstrieren
import sys

zahlen_liste = [x for x in range(1_000_000)]  # List Comprehension - alles auf einmal im Speicher
print(f"Liste Größe: {sys.getsizeof(zahlen_liste) / 1024 / 1024:.2f} MB")

zahlen_gen = (x for x in range(1_000_000))  # Besser: Generator - erzeugt Werte bei Bedarf
print(f"Generator Größe: {sys.getsizeof(zahlen_gen)} Bytes")
```

Generator Expressions sind ideal für große Daten (*Streaming*-Verarbeitung) – 2/2

```
# Speicherverbrauch demonstrieren
import sys

zahlen_liste = [x for x in range(1_000_000)]  # List Comprehension - alles auf einmal im Speicher
print(f"Liste Größe: {sys.getsizeof(zahlen_liste) / 1024 / 1024:.2f} MB")

zahlen_gen = (x for x in range(1_000_000))  # Besser: Generator - erzeugt Werte bei Bedarf
print(f"Generator Größe: {sys.getsizeof(zahlen_gen)} Bytes")


# Praktisches Beispiel: Große Datei verarbeiten
def zeilen_verarbeiten(dateiname):
    return (zeile.strip().upper()
            for zeile in open(dateiname))  # Generator zurückgeben

# Generator verwenden
for zeile in zeilen_verarbeiten("grosse_datei.txt"):
    # Verarbeitet eine Zeile nach der anderen
    pass  # Hier wäre die eigentliche Verarbeitung
```

Beispiel: Alle möglichen 8-stelligen Passwörter generieren

```python
import string
from itertools import product

alphabet = string.ascii_lowercase + string.digits  # a-z, 0-9

def generate_passwords(length=8):  # Generator für alle möglichen 8-stelligen Passwörter
    return (''.join(chars)
            for chars in product(alphabet, repeat=length))
    # Generiert: 'aaaaaaaa', 'aaaaaaab', ..., '99999999'
    # Aber immer nur eins nach dem anderen!


# Verwendung (etwas anders als im vorigen Beispiel)
pw_gen = generate_passwords()
# Erste paar Passwörter ansehen
for _ in range(5):
    print(next(pw_gen))  # Zeigt die ersten 5 möglichen Passwörter

# Anzahl möglicher Passwörter (36^8)
print(f"Mögliche Kombinationen: {len(alphabet) ** 8:,}")
# Aber der Generator braucht nur Speicher für ein Passwort!
```

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Aufgabe 1

Was macht dieser Code?

```
namen = ["Anna", "Bob", "Charlie", "David"]

result = [len(name) for name in namen if name.lower().startswith('c’)]

print(result)  # Was wird ausgegeben?
```

Aufgabe 1

Was macht dieser Code?

```
namen = ["Anna", "Bob", "Charlie", "David"]

result = [len(name) for name in namen if name.lower().startswith('c')]

print(result)  # Was wird ausgegeben?


# Ausgabe:
# [7]
```

Aufgabe 2

Finden Sie den/die Fehler!

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

quadrate = map(lambda x: x**2, zahlen)

for zahl in quadrate:
    print(zahl)

for zahl in quadrate:
    print(zahl)
```

Aufgabe 2

Finden Sie den/die Fehler!

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

quadrate = map(lambda x: x**2, zahlen)

for zahl in quadrate:
    print(zahl)

for zahl in quadrate:
    print(zahl)

# Doppeltes Iterieren über quadrate!
# map-Objekt ist aber bereits aufgebraucht.
# Daher: keine Ausgabe in zweiter Schleife.
```

Aufgabe 3

Verbessern Sie den Code. Nutzen Sie eine geeignete Comprehension.

```
emails = ["max@firma.de", "lisa@firma.de"]
email_dict = {}
for email in emails:
    name = email.split('@')[0]
    email_dict[email] = name
```

Aufgabe 3

Verbessern Sie den Code. Nutzen Sie eine geeignete Comprehension.

```python
emails = ["max@firma.de", "lisa@firma.de"]
email_dict = {}
for email in emails:
    name = email.split('@')[0]
    email_dict[email] = name


# Mit dict comprehension:

email_dict = {email: email.split('@')[0] for email in emails}
```

Reine Funktionen:

Gleiche Eingabe, gleiche Ausgabe, keine Seiteneffekte.

Macht Code vorhersehbar und testbar.

Lambda-Funktionen:

Kurze, anonyme Funktionen für einfache Operationen.

Nicht überstrapazieren!

Funktionen als Werte:

Python kann Funktionen wie normale Werte weitergeben und speichern ("first-class citizens").

List Comprehensions:

Eleganter Ersatz für map/filter, "pythonic”.

Gibt es auch für Sets und Dicts.

Map und Filter:

Wenden Funktionen auf Listen an oder filtern Elemente.

Erzeugen Iterator-Objekte (Speichereffizienz).

Generators:

Erzeugen Werte erst bei Bedarf.

Ideal für große Datenmengen (Streaming).