Collections

Von Arrays zu Listen

```c
// C
int noten[100];    // Feste Größe
noten[105] = 95;   // Gefährlich!
```

```python
# Python
noten = []         # Leere Liste
noten.append(95)   # Wächst automatisch
```

Alles ist ein Objekt und Variablen sind wie Namensschilder

C

```c
// Variablen sind Speicherstellen
int x = 42;    // x IST 42
```

Python

```python
# Variablen sind Namensschilder
x = 42         # x ZEIGT AUF 42
y = x          # y zeigt auf dasselbe Objekt
x = 23         # x zeigt jetzt woanders hin

# Funktionsparameter verhalten sich genauso
def aendere(x):
    x = x + 1    # Neues Namensschild!
```

Was ein Objekt genau ist, behandeln wir später.

Referenzen auf Objekte am Beispiel von Listen

```
noten = [95, 92, 98]  # Eine Liste erstellen
kopie = noten         # Kein neuer Container!


# Beide Namen zeigen auf dieselbe Liste
kopie.append(90)      # Wirkt sich auf 'noten' aus
print(noten)          # [95, 92, 98, 90]


# Lösung: Echte Kopie erstellen
echte_kopie = list(noten)  # Neues Listenobjekt
```

Variablen als Namensschilder

- Variablen sind keine Behälter, sondern verhalten sich wie Namensschilder, die an Objekten hängen
- Mehrere Namensschilder können auf dasselbe Objekt zeigen
- Namensschilder können umgehängt werden

```
# Ein Objekt, ein Name
x = 42
print(x) # 42

# Ein Objekt, zwei Namen
y = x # y zeigt auf dasselbe 42
print(y) # 42

# Namensschild umhängen
x = 23 # x zeigt jetzt auf 23
print(y) # Immer noch 42!
```

Listen in Python – Einführung

```python
noten = [95, 92, 98]    # Liste erstellen
print(noten[0])         # Erstes Element: 95
noten[1] = 93           # Element ändern


# Listen wachsen dynamisch
noten.append(90)        # Am Ende anfügen
noten.insert(0, 97)     # Am Anfang einfügen


# Länge und Existenz
print(len(noten))       # Anzahl Elemente mit len(liste)
if 92 in noten:         # Suche mit "if element in liste"
    print("Gefunden!")
```

Wie verhalten sich Listen beim Funktionsaufruf?

```
# Funktion mit int-Parameter (verhält sich wie "Pass by Value")
def increment(x):
    x = x + 1
    print(f"In Funktion: {x}")  # 43

# Funktion mit Listen-Parameter (verhält sich wie "Pass by Reference")
def add_grade(grades):
    grades.append(100)
    print(f"In Funktion: {grades}")  # [95, 92, 98, 100]

# Aufruf und Vergleich
zahl = 42
increment(zahl)
print(f"Nach Aufruf: {zahl}")  # Immer noch 42!

noten = [95, 92, 98]
add_grade(noten)
print(f"Nach Aufruf: {noten}")  # [95, 92, 98, 100]
# Listen können also in Funktionen verändert werden!
```

Objekte sind „mutable“ oder „immutable“.

Immutable (unveränderlich): Änderung erzeugt neues Objekt

Bsp.: Zahlen (int, float), Strings, Tupel (später...)

Mutable (veränderlich): Objekt kann intern verändert werden

Bsp.: Listen, Dictionaries (später...), Sets (später...), …

Unterschied wird relevant bei Funktionsparametern und Dictionary-Keys.

```
x = 42          # Immutable
x += 1          # Neues Objekt!

lst = [1, 2]    # Mutable
lst.append(3)   # Gleiches Objekt!
```

Es gibt kein „Pass by Reference“ mittels Pointern wie in C.

C

```c
void increment(int* x) {
    (*x)++;    // Modifies caller's variable
}
```

Python

```python
# wir könnten  einen mutablen Container nutzen
def increment(num_ref):
    num_ref[0] += 1  # Ändert Listenobjekt

# Verwendung
x = [42]        # int in Liste wrappen
increment(x)
print(x[0])     # 43

# Üblicheres Vorgehen: Neuen Wert zurückgeben
def increment(x):
    return x + 1
x = increment(x)  # Neuzuweisung um Änderung zu sichern
```

Zuweisung versus Objektänderung bei mutablen Objekten

– Objektänderung: Wirkt sich auf alle Namen aus

– Neuzuweisung: Betrifft nur lokale Variable

– Unterschied sichtbar bei Funktionen

```
def add_grade(grades):
    # Fall 1: Objekt ändern
    grades.append(100) # Ändert Original

    # Fall 2: Neue Zuweisung
    grades = [95, 96, 97] # Ändert lokale Variable
    print(f"In Funktion: {grades}")

# Test beider Fälle
noten = [90, 91, 92]
add_grade(noten)
print(f"Nach Aufruf: {noten}") # [90, 91, 92, 100]
```

Mutabilität ist nicht an der Punktnotation erkennbar.

Auch immutable Typen haben Methoden (geben neue Objekte zurück).

```
# Strings haben auch Methoden, sind aber immutable
text = "hallo"
gross = text.upper()     # liefert geänderte Kopie zurück
print(text)              # Immer noch "hallo"
print(gross)             # "HALLO"


# Versuch der Änderung
text = "Python"
text.replace('P', 'J')   # Erstellt neue Kopie
print(text)              # Immer noch "Python"!


# Richtiges Vorgehen: Zuweisung
text = text.replace('P', 'J')  # Jetzt ist es "Jython"
```

Zugriff und Änderung von Strings

Zugriff (**Indexing**) ist möglich; Änderung nicht möglich.
String-Methoden geben neue Strings zurück.

```
# Zugriff funktioniert
text = "Python"
print(text[0])     # 'P' - Zugriff geht!


# Änderungsversuch
text[0] = 'J'      # TypeError: Strings sind immutable!

# Richtige Lösung
text = 'J' + text[1:]  # Neuen String erstellen
```

Slicing: Zugriff auf Teile von Strings mit `[Start:Ende:Schrittweite]`

Verkettung (*Concatenation* mit +) erzeugt neuen String.

Tupel sind *immutable* Listen.

Nutzung für multiple Rückgabewerte bei Funktionen (automatisches Packen und Entpacken in Tupeln); Nutzung als Dictionary Keys (später!)

```
# Tupel sind wie Listen, aber mit runden Klammern
punkt = (3, 4)        # Tupel erstellen
x = punkt[0]          # Zugriff wie bei Listen
# punkt[0] = 5        # geht nicht: Tupel sind immutable!



# Der wahre Nutzen zeigt sich hier:
def get_bounds(numbers):
    return min(numbers), max(numbers)  # Tupel!

# Automatisches Entpacken
minimum, maximum = get_bounds([1,2,3,4,5])
print(f"Min: {minimum}, Max: {maximum}")
```

Tupel in Aktion: Gruppieren mehrerer Werte, mehrfache Rückgabewerte, automatisches Packen und Entpacken

```
# Der berühmte Swap
x = 5
y = 10
x, y = y, x          # Tupel-Magic!

# Mehrere Rückgabewerte
def get_statistik(zahlen):
    summe = sum(zahlen)
    mittel = summe / len(zahlen)
    kleinster = min(zahlen)
    return summe, mittel, kleinster

# Elegant entpacken
gesamt, durchschnitt, minimum = get_statistik([1,2,3,4,5])

# Teilweises Entpacken
summe, mittel, _ = get_statistik([1,2,3,4,5])
# guter Stil: _ ignoriert Wert(e), die man nicht benutzt
```

Tupel in Aktion
Weitere Beispiele

```
# Unveränderliche Sequenzen
punkt = (3, 4)      # Wie eine "eingefrorene" Liste

# Hauptanwendungen:
# 1. Mehrere Rückgabewerte
def get_position():
    return 3, 4     # Implizites Tupel

# 2. Elegantes Entpacken
x, y = get_position()
x, y = y, x         # Der berühmte Swap

# 3. Gruppierung zusammengehöriger Werte
person = ("Alice", 25, "Informatik")
name, alter, fach = person
```

Etwas unpraktisch:

```
namen = ["Alice", "Bob", "Charlie"]
noten = [95, 87, 91]
```

Dictionaries: Namen für Werte

- Dictionaries speichern Schlüssel-Wert-Zuordnungen
- Alternative zu parallelen Listen
- Schneller Zugriff über Schlüssel
- Dynamisch erweiterbar

```
# Werte mit Schlüsseln verbinden
noten = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 91
}

# Einfacher Zugriff
print(noten["Alice"])     # 95
noten["David"] = 93       # Neuen Eintrag hinzufügen
```

Nutzung von Dictionaries

Direkter Zugriff mit `[key]` wirft Fehler, wenn Element nicht existiert.

```python
noten = {         # Schlüssel müssen eindeutig sein
    "Alice": 95,
    "Bob": 87,
    "Charlie": 91
}

# Sicherer Zugriff
if "Alice" in noten:
    print(noten["Alice"])     # Existiert

print(noten.get("David", 0))  # Default wenn nicht da

# Werte aktualisieren
noten["Bob"] = 90             # Überschreiben
noten.update({"David": 88, "Eve": 92})  # Mehrere
```

Nutzung von Dictionaries

```
students = {
    "Alice": {
        "Noten": [95, 92, 98],
        "Fach": "Informatik"
    },
    "Bob": {
        "Noten": [87, 85, 90],
        "Fach": "Physik"
    }
}

# Iteration
for name in students:  # iteriert über die Schlüssel (die liefert keys() zurück)
    print(name)

for name, info in students.items():  # Iteriert über die Key-Value-Paare
    print(f"{name} studiert {info['Fach']}")

# Es gibt auch values(), mit dem man über die Werte iterieren kann.
```

Dictionaries sind mächtiger als structs in C.

C

```c
struct Eintrag {
    char name[50];
    char nummer[20];
};
```

Python

```python
telefonbuch = [
    {"Name": "Alice", "Nummer": "123-4567"},
    {"Name": "Bob", "Nummer": "987-6543"}
]

eintrag = {"Name": "Alice", "Nummer": "123-4567"}

telefonbuch_messed_up = [
    eintrag,
    {"Name": "Bob", "Nummer": "987-6543"},
    {"Charlie": "444-3321"}
]

andere_telefonbuch_struktur = {
    "Alice": "123-4567",
    "Bob": "987-6543",
    "Charlie": "444-3321"
}
```

Viele Freiheiten!

aber: Komplexität begünstigt Fehler

daher: Zurückhaltung und Dokumentation wichtig

Schlüssel von Dictionaries müssen immutable sein

- Dictionaries nutzen Hash-Tabellen.
- Hash-Wert muss konstant bleiben.
- Bei mutablen Objekten kann sich der Hash-Wert ändern.
- Verlorene Einträge wären möglich.

Als Schlüssel geeignet:

- Strings und Tupel: ja
- Listen: nein
- andere änderbare Objekte: nein

Mengen (Sets)

– Enthalten nie Duplikate, sind ungeordnet, sind mutable.

– Überprüfung auf Enthaltensein geht schneller als in Listen.

```
# Effiziente Speicherung eindeutiger Werte
zahlen = {1, 2, 3}       # Set erstellen
zahlen.add(4)            # Element hinzufügen
zahlen.add(2)            # Wird ignoriert!

# Praktische Anwendung
besucher = ["Alice", "Bob", "Alice", "Charlie", "Bob"]
eindeutig = set(besucher)  # Duplikate entfernen
print(eindeutig)           # {'Alice', 'Bob', 'Charlie'}

# Mengenoperationen
team_a = {"Alice", "Bob", "Charlie"}
team_b = {"Charlie", "David", "Eve"}
beide_teams = team_a | team_b     # Vereinigung
gemeinsam = team_a & team_b       # Schnittmenge
nur_a = team_a - team_b           # Differenz
```

Collections im Überblick

Listen [1, 2, 3]
geordnet, veränderbar, wie Arrays, aber dynamisch

Tupel (1, 2, 3)
geordnet, unveränderbar, perfekt für Wertegruppen

Dictionaries {"a": 1}
Schlüssel-Wert Paare, schneller Zugriff (Hash-Tabelle)

Sets {1, 2, 3}
keine Duplikate, keine Ordnung, schnelle Suche, Mengenoperationen

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Slicing mit [start:end:step]

start: erster Index (inklusiv), end: letzter Index (exklusiv), step: Schrittweite

Negative Indizes sind möglich.

```
# Slicing funktioniert für alle Sequenzen
text = "Python"
liste = [1, 2, 3, 4, 5]
tupel = (10, 20, 30, 40, 50)

print(text[1:4])     # "yth"
print(liste[1:4])    # [2, 3, 4]
print(tupel[1:4])    # (20, 30, 40)

# Negative Indizes
print(text[-2:])     # "on"
print(liste[:-1])    # [1, 2, 3, 4]

# Schrittweite
print(text[::2])     # "Pto"
```

Elegante Lösung für gängige Probleme

```python
# Wortfrequenz (vorher kompliziert mit Schleifen)
text = "the quick brown fox jumps over the lazy dog"
woerter = text.split()
haeufigkeit = {}

for wort in woerter:
    haeufigkeit[wort] = haeufigkeit.get(wort, 0) + 1
```

```python
# Doppelte Buchstaben (vorher umständlich)
def finde_doppelte(text):
    return {text[i:i+2] for i in range(len(text)-1)
            if text[i] == text[i+1]}

print(finde_doppelte("hello bookkeeper"))  # {'ll', 'oo', 'ee'}
```

Beispielhafte Wiederholung der Nutzung

```
# 1. Listen sind veränderbar
scores = [10, 20]
scores.append(30)      # Ändert die Liste

# 2. Strings und Tupel sind unveränderbar
name = "Python"
# name[0] = "J"       # Geht nicht!
name = "J" + name[1:] # Neuer String

# 3. Dictionaries verbinden Schlüssel und Werte
noten = {"Alice": 95, "Bob": 87}
print(noten["Alice"])  # Schneller Zugriff

# 4. Sets speichern eindeutige Werte
teilnehmer = {"Alice", "Bob", "Alice"}
print(teilnehmer)     # Nur einmal "Alice"
```

Beispiel für verschachtelte Datenstruktur 1/2

```
# Tic-Tac-Toe Brett
brett = [
    [" ", "X", "O"],
    ["X", "O", " "],
    ["O", " ", "X"]
]


# Zeile ausgeben
print(brett[0])         # Erste Zeile

# Einzelnes Feld
print(brett[1][1])      # Mitte: "O"
```

Beispiel für verschachtelte Datenstruktur 2/2

```python
schule = {
    "Informatik": {
        "participants": 24,
        "topics": ["Python", "Java", "SQL"],
        "instructors": ["Dr. Schmidt", "Prof. Meyer"]
    },
    "Mathematik": {
        "participants": 18,
        "topics": ["Analysis", "Algebra"],
        "instructors": ["Dr. Weber"]
    }
}

for subject, info in schule.items():
    instructors = len(subject["instructors"])
    num_participants = info["participants"]
    print(f"{subject}: {num_participants} Studierende, " # String-Verkettung
          f"{instructors} Dozierende")
```

Collections in Python:
Listen, Tupel, Mengen, Dictionaries

Geschachtelte Strukturen flexibel kombinierbar:

Listen von Dictionaries, Dictionaries mit Tupel-Keys, ...

Mutable vs. Immutable:
Entscheidend für das Verhalten, v.a. beim Funktionsaufruf

Slice-Notation:
Einheitlich für alle Sequenzen

[start:end:step]

Negative Indizes möglich.

Listen für Sequenzen, Dictionaries für Zuordnungen, Sets für eindeutige Werte

Flexibilität hat ihren Preis:

Die vielen Möglichkeiten ermöglichen hohe Komplexität, die schwer zu verstehen ist und Fehler begünstigt.