

Pig-Übungen

Würfelspiel · Funktionen · Funktionale Programmierung

Modul: Objektorientierte und Funktionale Programmierung
Sommersemester 2026

Hinweis zur Erstellung: Die Aufgaben dieser Übungsreihe wurden mit Unterstützung von Claude (Anthropic) erstellt. Bitte prüft Code und Ergebnisse stets selbst, bevor ihr euch auf sie verlasst.

Aufbau: Dieses Dokument enthält zuerst alle Aufgaben (Teil 1) und danach die Musterlösungen (Teil 2). Versucht die Aufgaben zuerst selbst zu lösen und schaut erst anschließend in die Lösungen.

Inhalt

Teil 1 — Aufgaben

Übung 1 — Pig (Würfelspiel) · Stufen 1–6

Übung 2 — Pig mit Funktionen · Phase 1 (Aufg. 1–4), Phase 2 (Stufen 1–5)

Übung 3 — Funktionale Programmierung · Phase A (A1–A5), Phase B (B1–B5)

Teil 2 — Musterlösungen

Musterlösungen Übung 1 · Übung 2 · Übung 3

TEIL 1

Aufgaben

Übung 1 — Pig (Würfelspiel)

Modul: Objektorientierte und Funktionale Programmierung · **Thema:** `if / elif / else`, `while`, Listen, `import random` · **Dauer:** ca. 90 Minuten

SPIELREGELN

Zwei Spieler*innen spielen abwechselnd. Wer am Zug ist, würfelt mit einem sechsseitigen Würfel und addiert die Augenzahl zu seiner Rundensumme. Nach jedem Wurf darf entschieden werden:

- **Weiter würfeln** – mehr Punkte sammeln, aber Risiko erhöhen.
- **Halten** – die Rundensumme wird zur Gesamtsumme addiert, der/die andere Spieler*in ist dran.

Aber: Wer eine **1** würfelt, verliert die gesamte Rundensumme und ist sofort dran mit Übergeben. Wer zuerst **100** Punkte Gesamtsumme erreicht, gewinnt.

AUFBAU IN 6 STUFEN

Wir bauen das Spiel schrittweise auf. Nach jeder Stufe gibt es ein lauffähiges Zwischenergebnis – wenn eine Stufe nicht klappt, könnt ihr trotzdem mit der vorherigen weiterarbeiten.

Diese Übung greift Konzepte aus den letzten beiden Vorlesungen auf: `if / elif / else`, `while`-Schleifen, Listen mit ihren Methoden (`append`, `pop`, `count`, ...), Slicing, `import random`, f-strings und den Modulo-Operator `%`. Wo immer ihr `print(...)` schreibt, dürft (und sollt) ihr gerne f-strings verwenden, also z. B. `print(f"Wurf: {wurf}")` statt `print("Wurf:", wurf)`.

Stufe 1 — Wer hat gewonnen?

Aufgabe: Schreibt ein Programm, das zu zwei festen Punkteständen `gesamt_spieler_1` und `gesamt_spieler_2` prüft, ob jemand gewonnen hat. Bei Pig gewinnt, wer als erste*r 100 oder mehr Punkte erreicht. Solange beide noch unter 100 sind, läuft das Spiel weiter – ein Unentschieden gibt es nicht.

Drei Fälle sollen unterschieden werden:

- Spieler 1 hat `>= 100` → „Spieler 1 hat gewonnen!“
- Spieler 2 hat `>= 100` → „Spieler 2 hat gewonnen!“
- sonst → „Noch kein Sieger – weiterspielen!“

Probiert es mit verschiedenen Werten aus (z. B. `(105, 80)`, `(40, 99)`, `(120, 110)`).

Hinweis: `if`, `elif`, `else` – wie beim Goofspiel aus der Vorlesung, nur mit anderer Bedingung.

Stufe 2 — Eine einzelne Runde würfeln

Aufgabe: Programmiert eine einzelne Runde für eine*n Spieler*in:

1. Solange weitergewürfelt wird, einen W6 werfen (`random.randint(1, 6)`).
2. Bei einer 1: Rundensumme verfällt, Runde sofort vorbei.
3. Sonst: Augenzahl zur Rundensumme addieren und per `input` fragen, ob weitergewürfelt werden soll.
4. Am Ende ausgeben, wie viele Punkte in dieser Runde tatsächlich „gesichert“ wurden.

Hinweis: Ihr braucht eine `while`-Schleife mit einer Boolean-Variablen wie `runde_laeuft = True`, die an zwei Stellen auf `False` gesetzt werden kann (Wurf einer 1 oder Spieler hält an). Reihenfolge bewusst beachten: erst würfeln, dann auswerten, dann fragen.

Stufe 3 — Wurfhistorie als Liste

Aufgabe: Erweitert Stufe 2 so, dass alle Würfe einer Runde in einer Liste `wurfhistorie` gespeichert werden. Gebt am Ende der Runde aus:

- die komplette Wurfhistorie,
- wie oft gewürfelt wurde (Tipp: `len(...)`),
- die gesicherte Rundensumme.

Hinweis: `wurfhistorie = []` zu Beginn der Runde, dann mit `wurfhistorie.append(wurf)` nach jedem Wurf erweitern.

Stufe 4 — Zwei Spieler bis 100

Aufgabe: Baut das volle Spiel:

1. Beide Spieler*innen starten bei `0`.
2. Solange niemand `>= 100` Punkte hat, werden abwechselnd Runden gespielt (jede Runde: Code aus Stufe 3).
3. Nach jeder Runde Punktestand beider ausgeben.
4. Am Ende den Sieger ermitteln (Code aus Stufe 1).

Hinweis: Eine sehr saubere Lösung ist, beide Spielstände in einer Liste `gesamt = [0, 0]` zu halten und mit einer Variable `aktueller_spieler` zwischen 0 und 1 zu wechseln. Dann reicht `gesamt[aktueller_spieler] += rundensumme`. Der Wechsel geht z. B. mit `aktueller_spieler = 1 - aktueller_spieler`.

Stufe 5 (Bonus) — Computergegner

Aufgabe: Lasst Spieler 2 vom Computer übernehmen. Eine einfache, aber überraschend gute Strategie ist die „Hold-at-20“-Regel: der Computer würfelt weiter, solange seine Rundensumme unter 20 ist, und hält ansonsten an. Erweitert den Code so, dass nur Spieler 1 per `input` gefragt wird.

Optional: Macht die Strategie spannender – z. B. mit 70 % Wahrscheinlichkeit „Hold-at-20“, sonst zufällig weiterwürfeln (`random.random()`).

Stufe 6 — Spielstatistik (Next Level)

Bisher war die `wurfhistorie` immer auf eine einzelne Runde beschränkt und ist am Anfang jeder Runde wieder geleert worden. Spannender wird es, wenn wir uns am Ende des Spiels die kompletten Spielverläufe anschauen.

Aufgabe: Sammelt für beide Spieler*innen alle gespielten Runden in einer Liste von Listen. Analog zu `gesamt = [0, 0]` aus Stufe 4 legt ihr vor dem Spiel an:

```
alle_runden = [[], []]
```

`alle_runden[0]` und `alle_runden[1]` sind also Listen, die selbst Runden enthalten – und jede Runde ist wiederum eine Liste von Würfeln. Am Ende jeder Runde, bevor ihr den Spieler wechselt, hängt ihr die fertige `wurfhistorie` als Ganzes an:

```
alle_runden[aktueller_spieler].append(wurfhistorie)
```

`alle_runden[0]` könnte am Spielende dann z. B. so aussehen:

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

Die Struktur ist hier wichtiger als bei einer flachen Wurfliste: Pech-Runden enden mit einer 1, Halten-Runden enden auf einer anderen Zahl – das könnt ihr direkt aus der Liste ablesen.

Gebt am Spielende für jeden Spieler eine Statistik aus mit:

- Anzahl gespielter Runden – `len(alle_runden[i])`
- Erste 3 Runden mit Slicing: `alle_runden[i][:3]`
- Anzahl Pech-Runden – wie viele Runden enden mit einer 1? Lauft mit einer `while`-Schleife durch die Runden und nutzt einen negativen Index: `runde[-1] == 1`.
- Längste Runde – `while`-Schleife durch die Runden, jeweils `len(runde)` mit dem bisher größten Wert vergleichen.
- Anzahl Würfe insgesamt und Anzahl gerader Würfe – hier braucht ihr eine verschachtelte `while`-Schleife: eine äußere durch die Runden, eine innere durch die Würfe innerhalb der Runde. Modulo `% 2 == 0`.

Nutzt f-strings für die Ausgabe, z. B.:

```
print(f"Spieler {i+1} hat {len(alle_runden[i])} Runden gespielt.")
```

WENN IHR FERTIG SEID

Probiert aus:

- Was passiert, wenn ihr das Ziel auf 30 statt 100 setzt?
- Lasst zwei Computerstrategien gegeneinander antreten (z. B. „Hold-at-15“ gegen „Hold-at-25“) und spielt 1000 Runden ohne `print` – wer gewinnt häufiger?
- Erweitert die Statistik aus Stufe 6: Was war die lukrativste Runde (höchste gesicherte Punktsumme)? Tipp: Lauft mit verschachtelten `while`-Schleifen durch die Runden, summiert die Würfe einer Runde – aber nur, wenn die Runde nicht mit einer 1 endet (sonst wären die Punkte ja verfallen).

Übung 2 — Pig mit Funktionen

Modul: Objektorientierte und Funktionale Programmierung · **Thema:** Funktionen — Definition, Parameter, Rückgabewerte, Default-Werte, Funktionen als Argumente · **Dauer:** ca. 90 Minuten · **Voraussetzung:** Euer funktionierender Pig-Code aus der letzten Übung.

EINLEITUNG

In der letzten Übung habt ihr Pig gebaut – und an einigen Stellen ist der Code dabei unnötig groß geworden: wiederholte `print`-Aufrufe, eine innere Schleife, die mehrere Aufgaben gleichzeitig erledigt, und schwer zu lesende Bedingungen. Heute lernt ihr das Werkzeug kennen, das genau diese Schmerzen heilt: Funktionen.

- **Phase 1** (ca. 20 Min): Vier kleine Vorab-Aufgaben außerhalb von Pig, in denen ihr die Syntax von Funktionen kennenlernt.
- **Phase 2** (ca. 70 Min): In fünf Stufen baut ihr euer Pig-Spiel um. Bei jeder Stufe lohnt es sich, vorher und nachher zu vergleichen.

PHASE 1 — FUNKTIONEN KENNENLERNEN

Aufgabe 1 — Funktion ohne Parameter

Schreibt eine Funktion `begruessung()`, die *Hallo, willkommen in der Übung!* ausgibt. Ruft sie zweimal auf.

Schlüsselwörter: `def`, Doppelpunkt, Einrückung, Aufruf mit `()`.

Aufgabe 2 — Funktion mit einem Parameter

Schreibt eine Funktion `begruesse(name)`, die *Hallo, {name}!* ausgibt. Ruft sie mit drei verschiedenen Namen auf.

Aufgabe 3 — Funktion mit Rückgabewert

Schreibt eine Funktion `quadrat(zahl)`, die das Quadrat der Zahl **zurückgibt** (nicht ausgibt!). Speichert das Ergebnis in einer Variable und druckt es danach aus.

Hinweis: Der wichtigste Lernschritt hier ist der Unterschied zwischen `print` (Wert wird angezeigt, ist danach weg) und `return` (Wert wird zurückgegeben und kann weiterverwendet werden). `quadrat(3) + quadrat(4)` funktioniert nur mit `return`.

Aufgabe 4 — Default-Werte

Schreibt eine Funktion `wuerfle(seiten=6)`, die eine zufällige Zahl zwischen 1 und `seiten` zurückgibt. Ruft sie einmal ohne Argument auf (soll wie ein W6 funktionieren) und einmal mit `seiten=20` (W20).

PHASE 2 — PIG REFAKTORIEREN

Nehmt euren funktionierenden Pig-Code aus der letzten Übung. In jeder Stufe ändert ihr einen klar abgegrenzten Teil – wenn ihr unsicher seid, kopiert euren bisherigen Code vorher in eine neue Datei.

Stufe 1 — Eine kleine Helferfunktion

Aufgabe: Schreibt eine Funktion `hat_gewonnen(punkte)`, die `True` zurückgibt, wenn `punkte >= 100`, sonst `False`. Ersetzt damit in eurem Code:

- die Bedingung der äußeren `while`-Schleife,
- die `if`-Abfragen am Spielende.

Vorher:

```
while gesamt[0] < 100 and gesamt[1] < 100:
    ...
if gesamt[0] >= 100:
    print("Spieler 1 hat gewonnen!")
elif gesamt[1] >= 100:
    print("Spieler 2 hat gewonnen!")
```

Nachher:

```
while not hat_gewonnen(gesamt[0]) and not hat_gewonnen(gesamt[1]):
    ...
if hat_gewonnen(gesamt[0]):
    print("Spieler 1 hat gewonnen!")
elif hat_gewonnen(gesamt[1]):
    print("Spieler 2 hat gewonnen!")
```

Was hat sich verbessert? Lesbarkeit. Aus einer Zahl wird ein Konzept mit Namen. Bonus: wenn ihr das Ziel später auf 50 ändern wollt, müsst ihr es nur an einer Stelle anpassen.

Stufe 2 — Eine Funktion ohne Rückgabewert

Aufgabe: Schreibt eine Funktion `zeige_runden_status(spieler, wurfhistorie, gesamt)`, die die drei Status-Zeilen am Rundenende ausgibt:

- Wurfhistorie der Runde: [...]
- Anzahl der Würfe: N
- Punktestand: Spieler 1 = X | Spieler 2 = Y

Ersetzt die drei `print`-Aufrufe am Rundenende durch einen einzigen Funktionsaufruf.

Was hat sich verbessert? Drei Zeilen werden zu einer. Wenn ihr später das Ausgabeformat ändern wollt, gibt es nur noch eine Stelle. Außerdem: eine Funktion muss nicht zwingend etwas zurückgeben – manche Funktionen *tun* etwas, manche *liefern* etwas, manche beides.

Stufe 3 — Eine ganze Runde als Funktion ★

Das ist der große Refactoring-Schritt. Hier passiert die wichtigste Erkenntnis dieser Übung.

Aufgabe: Schreibt eine Funktion `spiele_runde(spieler)`, die

1. eine komplette Runde für `spieler` durchführt (innere `while`-Schleife mit Würfeln, Pech-Abfrage, „weiter?“-Frage),
2. am Ende zwei Werte zurückgibt: die gesicherte Rundensumme und die Wurfhistorie.

Ersetzt im Hauptprogramm die komplette innere `while`-Schleife durch einen einzigen Aufruf:

```
rundensumme, wurfhistorie = spiele_runde(aktueller_spieler)
```

Hinweis: Eine Funktion gibt mit `return a, b` ein sogenanntes Tupel zurück, das mit `x, y = ...` direkt in zwei Variablen aufgeteilt werden kann.

Was hat sich verbessert? Massiv. Die äußere Schleife passt jetzt auf einen Bildschirm und liest sich fast wie die Spielregel selbst. Was vorher 15 Zeilen verschachtelter Logik war, ist jetzt ein einziger Aufruf: `spiele_runde(...)`. Funktionen verstecken Komplexität, ohne sie zu verlieren.

Stufe 4 — Strategie als Funktion ★★

Bisher fragen wir in `spiele_runde` immer per `input`, ob weitergewürfelt werden soll. Das macht es unmöglich, einen Computer spielen zu lassen. Lösung: Die „weiter?“-Entscheidung wird selbst zu einer Funktion – und `spiele_runde` bekommt diese Funktion als Argument übergeben.

Aufgabe:

1. Definiert zwei Strategie-Funktionen:
 - `menschliche_strategie(rundensumme)` – fragt per `input`, gibt `True` zurück, wenn der Mensch weitermachen will, sonst `False`.
 - `computer_strategie(rundensumme)` – gibt `True` zurück, solange `rundensumme < 20` ist (die „Hold-at-20“-Regel).
2. Erweitert `spiele_runde` um einen zusätzlichen Parameter `strategie`. Statt direkt `input` aufzurufen, ruft die Funktion jetzt `strategie(rundensumme)` auf.
3. Im Hauptprogramm legt ihr eine Liste der Strategien an: `strategien = [menschliche_strategie, computer_strategie]` – und überlegt beim Aufruf die passende.

Hinweis: Beim Aufruf wird `menschliche_strategie` **ohne Klammern** übergeben! `strategien[aktueller_spieler]` ist die Funktion selbst, nicht ihr Ergebnis. Die Klammern kommen erst *innerhalb* von `spiele_runde`, wenn die Funktion wirklich aufgerufen wird. Funktionen als Werte, die man weiterreichen, in Listen packen und später aufrufen kann – genau das meint „Funktionale Programmierung“ im Modultitel.

Was hat sich verbessert? Jetzt ist es egal, ob ein Mensch oder ein Computer spielt – die Spiellogik bleibt dieselbe. Ihr könntet beliebig viele Strategien definieren und sie alle mit derselben `spiele_runde`-Funktion benutzen.

Stufe 5 — Das ganze Spiel als Funktion + Strategie-Turnier ★★★

Bisher steht das gesamte Hauptprogramm „frei“ im Skript. Damit können wir das Spiel genau einmal spielen. Wenn wir es aber in eine Funktion packen, können wir es beliebig oft aufrufen – z. B. 1000 Mal, um herauszufinden, welche Strategie wirklich besser ist.

Aufgabe:

1. Packt das Hauptprogramm in eine Funktion `spiele_pig(strategie_1, strategie_2, ziel=100, verbose=True)`. Sie soll am Ende den Sieger zurückgeben (1 oder 2).
2. Erweitert auch `spiele_runde` um einen Parameter `verbose=True`, sodass die `print`-Aufrufe nur dann ausgegeben werden, wenn `verbose True` ist. So könnt ihr stille Spiele laufen lassen.
3. Schreibt zusätzlich zwei einfache Strategien: `def hold_at_15(s): return s < 15` und `def hold_at_25(s): return s < 25`.
4. Lasst die beiden Strategien 1000 Mal gegeneinander antreten und zählt die Siege. Welche Strategie gewinnt häufiger? Nutzt dafür eine `for`-Schleife mit `range(1000)`.

Was hat sich verbessert? Alles. Aus einem Skript, das ein Spiel spielt, ist ein Werkzeug geworden, mit dem ihr beliebige Strategien gegeneinander testen könnt. Genau dafür sind Funktionen da: aus Code wird ein Baustein, der immer wieder eingesetzt werden kann.

WENN IHR FERTIG SEID

- Erweitert die Statistik aus Übung 1 (Stufe 6) ebenfalls zu einer Funktion `zeige_statistik(alle_runden, spieler)`.
- Spielt 1000 Turniere und plottet die Siegquoten verschiedener „Hold-at-X“-Strategien (X von 10 bis 30). Bei welchem X liegt das Optimum?
- Schreibt eine adaptive Strategie, die je nach Spielstand unterschiedlich vorsichtig spielt: bei großem Rückstand mehr Risiko, bei großem Vorsprung weniger. Schlägt sie die starren Hold-at-X-Strategien?

Übung 3 — Funktionale Programmierung

Modul: Objektorientierte und Funktionale Programmierung · **Thema:** lambda, mutable/immutable, scopes, pure functions, map/filter/reduce, Tupel/Set/Dict, time-Modul, Currying · **Dauer:** ca. 90 Minuten
· **Voraussetzung:** Eure Pig-Lösung aus Übung 2 (mit Funktionen).

EINLEITUNG

In der letzten Vorlesung habt ihr viele neue Konzepte kennengelernt: `lambda`, den Unterschied zwischen veränderbaren und unveränderbaren Variablen, globale vs. lokale Variablen, pure functions als Grundpfeiler der funktionalen Programmierung, und nützliche Funktionen höherer Ordnung wie `map`, `filter` und `reduce`.

- **Phase A** (ca. 45 Min): Fünf kleine eigenständige Aufgaben, in denen ihr die Konzepte isoliert ausprobiert.
- **Phase B** (ca. 45 Min): Ihr nehmt eure Pig-Lösung von letzter Woche und macht sie funktionaler – mit lambda-Strategien, map/filter/reduce-Statistiken und einem zeitgemessenen Turnier.

PHASE A — EIGENSTÄNDIGE AUFGABEN

Aufgabe A1 — Lambda als Warmup

Schreibt mit `lambda` drei kurze Funktionen und testet sie:

1. `quadrat`, die eine Zahl quadriert.
2. `addiere`, die zwei Zahlen addiert.
3. `ist_gerade`, die für eine Zahl `True` zurückgibt, wenn sie gerade ist, sonst `False`.

Hinweis: `lambda` ersetzt `def` für kurze Funktionen, die in eine Zeile passen. Es gibt kein `return` – der Ausdruck rechts vom `:` ist die Rückgabe.

Aufgabe A2 — Was gibt der Code aus?

Überlegt vorher, was ausgegeben wird – und prüft es dann in Python. Diskutiert mit eurem Sitznachbarn, falls ihr unterschiedlicher Meinung seid.

A2.1 — Immutable

```
a = 1
def f():
    a = 5
f()
print(a)
```

A2.2 — Mutable

```
a = [1, 2, 3]
def f():
    a[0] = 99
f()
print(a)
```

A2.3 — Liste neu zugewiesen

```
a = [1, 2, 3]
def f():
    a = [99]
f()
print(a)
```

A2.4 — global

```
a = 1
def f():
    global a
    a = a + 1
f()
f()
print(a)
```

Aufgabe A3 — map, filter, reduce

Gegeben sei die Liste `zahlen = [3, 7, 2, 8, 5, 10, 1, 6]`.

1. Verwendet `map` und ein `lambda`, um eine neue Liste mit den Quadraten aller Zahlen zu erzeugen.
2. Verwendet `filter` und ein `lambda`, um eine Liste mit nur den geraden Zahlen zu erhalten.
3. Verwendet `reduce` (aus `functools`!), um die Summe aller Zahlen zu berechnen.
4. Kombiniert alle drei: Berechnet die Summe der Quadrate aller geraden Zahlen.

Hinweis: `map` und `filter` geben Iteratoren zurück – um sie als Liste zu sehen, müsst ihr `list(...)` aussenrum schreiben.

Aufgabe A4 — Tupel, Set, Dictionary

Schreibt für jede der drei Datenstrukturen ein kleines Beispiel:

1. **Tupel:** Schreibt eine Funktion `min_max(liste)`, die das kleinste und das größte Element einer Liste als Tupel zurückgibt. Testet: `kleinstes, groesstes = min_max([3, 7, 1, 8, 4])`.
2. **Set:** Gegeben sei `wuerfe = [1, 3, 6, 2, 3, 1, 5, 6, 4, 2]`. Findet heraus, welche unterschiedlichen Augenzahlen vorkamen.
3. **Dict:** Gegeben sei dieselbe Liste. Zählt, wie oft jede Augenzahl vorkam. Ergebnis soll z. B. `{1: 2, 2: 2, 3: 2, 4: 1, 5: 1, 6: 2}` sein.

Aufgabe A5 — Currying

Currying bedeutet: Eine Funktion mit mehreren Argumenten wird in eine Kette von Funktionen umgewandelt, die jeweils nur ein Argument nehmen.

Klassisches Beispiel: Statt `addiere(3, 5)` schreibt ihr eine Funktion `addiere(3)`, die eine neue Funktion zurückgibt, welche dann 5 entgegennimmt: `addiere(3)(5)` ergibt `8`.

Aufgabe:

1. Schreibt eine „gecurryte“ Version von `addiere`: `def addiere(x): return lambda y: x + y`
2. Erzeugt damit eine spezialisierte Funktion `plus_5 = addiere(5)`.
3. Ruft `plus_5(3)`, `plus_5(10)` und `plus_5(-2)` auf.
4. **Bonus:** Schreibt `hold_at` so, dass `hold_at(20)` eine Strategie-Funktion zurückgibt, die wie `lambda s: s < 20` funktioniert. Erzeugt `hold_at_15`, `hold_at_20`, `hold_at_25` mit nur einer Zeile pro Strategie.

PHASE B — PIG FUNKTIONAL MACHEN

In dieser Phase nehmt ihr eure Lösung von Übung 2 (Pig mit Funktionen) und macht sie funktionaler. Wenn ihr noch keine fertige Lösung habt, könnt ihr auch nur die Codeschnipsel isoliert anschauen.

Stufe B1 — Pure functions: Welche unserer Funktionen ist eine?

Aufgabe: Geht eure Pig-Funktionen aus Übung 2 durch und ordnet jede in eine der drei Kategorien:

- **Pure:** hängt nur von Argumenten ab, hat keine Nebeneffekte.
- **Nicht pure (lesend):** liest globale Variablen oder externe Daten.
- **Nicht pure (mit Nebeneffekt):** verändert etwas außerhalb (gibt etwas auf der Konsole aus, fragt per `input` ab, modifiziert eine übergebene Liste, würfelt zufällig...).

Funktionen aus eurem Code: `hat_gewonnen`, `zeige_runden_status`, `spiele_runde`, `menschliche_strategie`, `computer_strategie`, `spiele_pig`.

Wichtig: „nicht pure“ ist kein Werturteil! Programme müssen an irgendeiner Stelle Nebeneffekte haben (sonst kommt nichts auf den Bildschirm). Das Ziel ist nicht, alles pure zu machen – sondern die Logik von den Nebeneffekten zu trennen.

Stufe B2 — Strategien als lambda

Eure `hold_at_15` und `hold_at_25` aus dem Turnier am Ende von Übung 2 sind kurze Funktionen mit nur einem Ausdruck. Perfekte Kandidaten für `lambda`.

Aufgabe:

1. Ersetzt die `def hold_at_15(...)` und `def hold_at_25(...)` durch `lambda`-Ausdrücke.
2. Mit Currying aus Aufgabe A5 könnt ihr noch einen Schritt weiter gehen: Schreibt `hold_at(schwelle)` als Funktions-Fabrik. Erzeugt damit fünf Strategien gleichzeitig: `hold_at(10)`, `hold_at(15)`, `hold_at(20)`, `hold_at(25)`, `hold_at(30)` – und packt sie in eine Liste.

Stufe B3 — Statistik mit map / filter / reduce

In Übung 1 Stufe 6 habt ihr drei Statistik-Blöcke geschrieben: Pech-Runden zählen, längste Runde finden, gerade Würfe zählen. Jeder dieser Blöcke wird jetzt eine einzige Zeile.

Erinnerung: `alle_runden[i]` ist eine Liste, deren Einträge selbst Listen sind (die Würfe der einzelnen Runden).

Aufgabe: Schreibt die folgenden Statistiken mit `map`, `filter`, `reduce`, `sum`, `max`, `len` – ohne explizite Schleifen:

1. Anzahl Pech-Runden (Runden, die mit einer 1 enden)
2. Länge der längsten Runde
3. Anzahl Würfe insgesamt
4. Anzahl gerader Würfe insgesamt

Stufe B4 — Tournament mit Zeitmessung

Das Tournament am Ende von Übung 2 simuliert 1000 Spiele mit einer `for`-Schleife. Jetzt machen wir es funktional und messen, wie schnell es ist.

Aufgabe:

1. Importiert das `time`-Modul und misst mit `time.time()` die Dauer einer Berechnung.
2. Simuliert 1000 Spiele zwischen `hold_at(15)` und `hold_at(25)`, indem ihr `map` auf `range(1000)` anwendet. Beachtet: `map` gibt einen Iterator zurück, der erst beim Materialisieren (z. B. mit `list(...)`) wirklich rechnet – das ist *lazy evaluation*!
3. Zählt die Siege beider Strategien mit Dict-Ergebnis: `{1: ..., 2: ...}`.
4. Vergleicht die Zeit von `list(map(...))` mit einer reinen `map(...)` ohne `list` – was beobachtet ihr?

Stufe B5 — Die mutable-Falle (Bonus)

Probiert folgenden Code aus und überlegt vorher, was passiert:

```
karten = [1, 2, 3, 4, 5]
import random
random.shuffle(karten)
print(karten)
```

Frage: Ist `karten` nach dem `shuffle` noch dieselbe Liste wie vorher? Wurde sie verändert?

Aufgabe:

1. Schreibt eine eigene Funktion `gemischt(liste)`, die eine gemischte **Kopie** der übergebenen Liste zurückgibt, ohne die ursprüngliche zu verändern. Tipp: `liste.copy()` oder `liste[:]`.
2. Testet:

```
original = [1, 2, 3, 4, 5]
neu = gemischt(original)
print(original) # soll unverändert sein!
print(neu)     # soll gemischt sein
```

WENN IHR FERTIG SEID

- Schreibt eine adaptive Strategie als `lambda` mit einer closure (Zugriff auf Variablen außerhalb): z. B. eine Strategie, die je nach Punktestand vorsichtig oder mutig spielt.

- Lasst alle fünf Strategien aus Stufe B2 in einem Round-Robin gegeneinander antreten (jede gegen jede, je 200 Spiele) und gibt eine Siegestabelle als Dict aus.
- Messt mit `time`, wie lange 10 000 Spiele dauern. Ist die Hauptzeit das Würfeln oder das `print`-Output? (Tipp: `verbose=False` setzen!)

TEIL 2

Musterlösungen

Musterlösungen — Übung 1

Stufe 1 — Wer hat gewonnen?

```
gesamt_spieler_1 = 105
gesamt_spieler_2 = 87

if gesamt_spieler_1 >= 100:
    print("Spieler 1 hat gewonnen!")
elif gesamt_spieler_2 >= 100:
    print("Spieler 2 hat gewonnen!")
else:
    print("Noch kein Sieger – weiterspielen!")
```

Stufe 2 — Eine einzelne Runde würfeln

```
import random

rundensumme = 0
runde_laeuft = True

while runde_laeuft:
    wurf = random.randint(1, 6)
    print("Du hast eine", wurf, "gewürfelt.")
    if wurf == 1:
        print("Pech! Die Rundensumme ist verloren.")
        rundensumme = 0
        runde_laeuft = False
    else:
        rundensumme += wurf
        print("Aktuelle Rundensumme:", rundensumme)
        antwort = input("Nochmal würfeln? (ja/nein) ")
        if antwort != "ja":
            runde_laeuft = False

print("Diese Runde hat dir", rundensumme, "Punkte gebracht.")
```

Stufe 3 — Wurfhistorie als Liste

```
import random

rundensumme = 0
wurfhistorie = []
runde_laeuft = True

while runde_laeuft:
    wurf = random.randint(1, 6)
    wurfhistorie.append(wurf)
    print("Du hast eine", wurf, "gewürfelt.")
    if wurf == 1:
        print("Pech! Die Rundensumme ist verloren.")
        rundensumme = 0
        runde_laeuft = False
    else:
        rundensumme += wurf
        print("Aktuelle Rundensumme:", rundensumme)
        antwort = input("Nochmal würfeln? (ja/nein) ")
        if antwort != "ja":
            runde_laeuft = False

print("\nWurfhistorie dieser Runde:", wurfhistorie)
print("Anzahl der Würfe:", len(wurfhistorie))
print("Gesicherte Punkte:", rundensumme)
```

Stufe 4 — Zwei Spieler bis 100

```
import random

gesamt = [0, 0]
aktueller_spieler = 0

while gesamt[0] < 100 and gesamt[1] < 100:
    print("\n--- Spieler", aktueller_spieler + 1, "ist am Zug ---")
    rundensumme = 0
    wurfhistorie = []
    runde_laeuft = True
    while runde_laeuft:
        wurf = random.randint(1, 6)
        wurfhistorie.append(wurf)
        print("Wurf:", wurf)
        if wurf == 1:
            print("Pech! Rundensumme verfällt.")
            rundensumme = 0
            runde_laeuft = False
        else:
            rundensumme += wurf
            print("Rundensumme:", rundensumme)
            antwort = input("Nochmal würfeln? (ja/nein) ")
            if antwort != "ja":
                runde_laeuft = False
    gesamt[aktueller_spieler] += rundensumme
    print("Wurfhistorie der Runde:", wurfhistorie)
    print("Punktestand: Spieler 1 =", gesamt[0],
          "| Spieler 2 =", gesamt[1])
    aktueller_spieler = 1 - aktueller_spieler
```

```
print("\n=== SPIELENDENDE ===")
if gesamt[0] >= 100:
    print("Spieler 1 hat gewonnen!")
elif gesamt[1] >= 100:
    print("Spieler 2 hat gewonnen!")
```

Stufe 5 (Bonus) — Computergegner

Kern: Ersetzt die „Nochmal würfeln?“-Frage in der inneren Schleife durch eine Fallunterscheidung. Nur Spieler 1 wird gefragt; Spieler 2 spielt „Hold-at-20“.

```
# Innerhalb der inneren while-Schleife (statt der input-Frage):
if aktueller_spieler == 0:
    antwort = input("Nochmal würfeln? (ja/nein) ")
    if antwort != "ja":
        runde_laeuft = False
else:
    if rundensumme >= 20:
        print("Computer hält an.")
        runde_laeuft = False
    else:
        print("Computer würfelt weiter.")
```

Stufe 6 — Spielstatistik (Next Level)

Variante mit `while`-Schleifen (nur die neuen Codeteile; der Rest des Spiels bleibt wie in Stufe 4):

```
# Vor der äußeren while-Schleife, neben gesamt = [0, 0]:
alle_runden = [], []

# Innerhalb der äußeren while-Schleife, nach gesamt[...] += rundensumme,
# vor dem Spielerwechsel:
alle_runden[aktueller_spieler].append(wurfhistorie)

# Nach dem Spielende, statt/zusätzlich zur Sieger-Ausgabe:
print("\n=== STATISTIK ===")
i = 0
while i < 2:
    runden = alle_runden[i]
    print(f"\n-- Spieler {i+1} --")
    print(f"Anzahl gespielter Runden: {len(runden)}")
    print(f"Erste 3 Runden: {runden[:3]}")

    # Pech-Runden zählen: Runden, die mit einer 1 enden
    pech = 0
    j = 0
    while j < len(runden):
        if runden[j][-1] == 1:
            pech += 1
        j += 1
    print(f"Davon Pech-Runden: {pech}")

    # Längste Runde finden
    laengste = 0
    j = 0
    while j < len(runden):
        if len(runden[j]) > laengste:
```

```

        laengste = len(runden[j])
        j += 1
    print(f"Längste Runde hatte: {laengste} Würfe")

    # Verschachtelte Schleife: alle Würfe + gerade Würfe zählen
    gesamt_wuerfe = 0
    gerade = 0
    j = 0
    while j < len(runden):
        k = 0
        while k < len(runden[j]):
            gesamt_wuerfe += 1
            if runden[j][k] % 2 == 0:
                gerade += 1
            k += 1
        j += 1
    print(f"Würfe insgesamt: {gesamt_wuerfe}")
    print(f"Davon gerade Würfe: {gerade}")
    i += 1

```

Kompaktere Variante mit `for`-Schleifen (falls schon bekannt). Der Rest des Codes bleibt identisch:

```

for i, runden in enumerate(alle_runden):
    print(f"\n-- Spieler {i+1} --")
    print(f"Anzahl gespielter Runden: {len(runden)}")
    print(f"Erste 3 Runden: {runden[:3]}")

    pech = 0
    for runde in runden:
        if runde[-1] == 1:
            pech += 1
    print(f"Davon Pech-Runden: {pech}")

    laengste = 0
    for runde in runden:
        if len(runde) > laengste:
            laengste = len(runde)
    print(f"Längste Runde hatte: {laengste} Würfe")

    gesamt_wuerfe = 0
    gerade = 0
    for runde in runden:
        for wurf in runde:
            gesamt_wuerfe += 1
            if wurf % 2 == 0:
                gerade += 1
    print(f"Würfe insgesamt: {gesamt_wuerfe}")
    print(f"Davon gerade Würfe: {gerade}")

```

Musterlösungen — Übung 2

PHASE 1 — FUNKTIONEN KENNENLERNEN

Aufgabe 1 — Funktion ohne Parameter

```
def begruessung():  
    print("Hallo, willkommen in der Übung!")  
  
begruessung()  
begruessung()
```

Aufgabe 2 — Funktion mit einem Parameter

```
def begruesse(name):  
    print(f"Hallo, {name}!")  
  
begruesse("Anna")  
begruesse("Ben")  
begruesse("Cem")
```

Aufgabe 3 — Funktion mit Rückgabewert

```
def quadrat(zahl):  
    return zahl * zahl  
  
ergebnis = quadrat(7)  
print(ergebnis)           # 49  
print(quadrat(3) + quadrat(4)) # 9 + 16 = 25
```

Aufgabe 4 — Default-Werte

```
import random  
  
def wuerfle(seiten=6):  
    return random.randint(1, seiten)  
  
print(wuerfle())           # 1 bis 6  
print(wuerfle(20))         # 1 bis 20  
print(wuerfle(seiten=100)) # 1 bis 100
```

PHASE 2 — PIG REFAKTORIEREN

Stufe 1 — Eine kleine Helferfunktion

```
def hat_gewonnen(punkte):  
    return punkte >= 100
```

Stufe 2 — Eine Funktion ohne Rückgabewert

```
def zeige_runden_status(spieler, wurfhistorie, gesamt):
    print(f"Wurfhistorie der Runde: {wurfhistorie}")
    print(f"Anzahl der Wuerfe: {len(wurfhistorie)}")
    print(f"Punktestand: Spieler 1 = {gesamt[0]} | Spieler 2 = {gesamt[1]}")

# Aufruf im Hauptprogramm:
zeige_runden_status(aktueller_spieler, wurfhistorie, gesamt)
```

Stufe 3 — Eine ganze Runde als Funktion ★

```
def spiele_runde(spieler):
    rundensumme = 0
    wurfhistorie = []
    runde_laeuft = True
    while runde_laeuft:
        wurf = random.randint(1, 6)
        wurfhistorie.append(wurf)
        print(f"Spieler {spieler + 1} wuerfelt: {wurf}")
        if wurf == 1:
            print("Pech! Rundensumme verfaellt.")
            rundensumme = 0
            runde_laeuft = False
        else:
            rundensumme += wurf
            print(f"Aktuelle Rundensumme: {rundensumme}")
            antwort = input("Nochmal wuerfeln? (ja/nein) ")
            if antwort != "ja":
                runde_laeuft = False
    return rundensumme, wurfhistorie
```

Das Hauptprogramm sieht damit so aus:

```
while not hat_gewonnen(gesamt[0]) and not hat_gewonnen(gesamt[1]):
    print(f"\n--- Spieler {aktueller_spieler + 1} ist am Zug ---")
    rundensumme, wurfhistorie = spiele_runde(aktueller_spieler)
    gesamt[aktueller_spieler] += rundensumme
    alle_runden[aktueller_spieler].append(wurfhistorie)
    zeige_runden_status(aktueller_spieler, wurfhistorie, gesamt)
    if not hat_gewonnen(gesamt[aktueller_spieler]):
        aktueller_spieler = 1 - aktueller_spieler
```

Stufe 4 — Strategie als Funktion ★★

```
def menschliche_strategie(rundensumme):
    antwort = input("Nochmal wuerfeln? (ja/nein) ")
    return antwort == "ja"

def computer_strategie(rundensumme):
    return rundensumme < 20

def spiele_runde(spieler, strategie):
    rundensumme = 0
    wurfhistorie = []
    runde_laeuft = True
```

```

while runde_laeuft:
    wurf = random.randint(1, 6)
    wurfhistorie.append(wurf)
    print(f"Spieler {spieler + 1} wuerfelt: {wurf}")
    if wurf == 1:
        print("Pech! Rundensumme verfaellt.")
        rundensumme = 0
        runde_laeuft = False
    else:
        rundensumme += wurf
        print(f"Aktuelle Rundensumme: {rundensumme}")
        if not strategie(rundensumme):
            runde_laeuft = False
return rundensumme, wurfhistorie

# Im Hauptprogramm:
strategien = [menschliche_strategie, computer_strategie]
# Spieler 1 ist Mensch, Spieler 2 ist Computer
while not hat_gewonnen(gesamt[0]) and not hat_gewonnen(gesamt[1]):
    print(f"\n--- Spieler {aktueller_spieler + 1} ist am Zug ---")
    rundensumme, wurfhistorie = spiele_runde(
        aktueller_spieler,
        strategien[aktueller_spieler]
    )
    gesamt[aktueller_spieler] += rundensumme
    alle_runden[aktueller_spieler].append(wurfhistorie)
    zeige_runden_status(aktueller_spieler, wurfhistorie, gesamt)
    if not hat_gewonnen(gesamt[aktueller_spieler]):
        aktueller_spieler = 1 - aktueller_spieler

```

Stufe 5 — Das ganze Spiel als Funktion + Strategie-Turnier ★★★

```

def spiele_runde(spieler, strategie, verbose=True):
    rundensumme = 0
    wurfhistorie = []
    runde_laeuft = True
    while runde_laeuft:
        wurf = random.randint(1, 6)
        wurfhistorie.append(wurf)
        if verbose:
            print(f"Spieler {spieler + 1} wuerfelt: {wurf}")
        if wurf == 1:
            if verbose:
                print("Pech! Rundensumme verfaellt.")
            rundensumme = 0
            runde_laeuft = False
        else:
            rundensumme += wurf
            if verbose:
                print(f"Aktuelle Rundensumme: {rundensumme}")
            if not strategie(rundensumme):
                runde_laeuft = False
    return rundensumme, wurfhistorie

def spiele_pig(strategie_1, strategie_2, ziel=100, verbose=True):
    gesamt = [0, 0]
    aktueller_spieler = 0
    strategien = [strategie_1, strategie_2]
    while gesamt[0] < ziel and gesamt[1] < ziel:

```

```

        rundensumme, _ = spiele_runde(
            aktueller_spieler,
            strategien[aktueller_spieler],
            verbose
        )
        gesamt[aktueller_spieler] += rundensumme
        if verbose:
            print(f"Punktstand: {gesamt[0]} : {gesamt[1]}")
        if gesamt[aktueller_spieler] < ziel:
            aktueller_spieler = 1 - aktueller_spieler
    if gesamt[0] >= ziel:
        return 1
    else:
        return 2

def hold_at_15(s):
    return s < 15

def hold_at_25(s):
    return s < 25

# Turnier: 1000 Spiele
sieg_1 = 0
sieg_2 = 0
for spiel in range(1000):
    sieger = spiele_pig(hold_at_15, hold_at_25, verbose=False)
    if sieger == 1:
        sieg_1 += 1
    else:
        sieg_2 += 1

print(f"Hold-at-15: {sieg_1} Siege von 1000")
print(f"Hold-at-25: {sieg_2} Siege von 1000")

```

Musterlösungen — Übung 3

PHASE A — EIGENSTÄNDIGE AUFGABEN

Aufgabe A1 — Lambda als Warmup

```
quadrat = lambda x: x * x
addiere = lambda x, y: x + y
ist_gerade = lambda x: x % 2 == 0

print(quadrat(7))      # 49
print(addiere(3, 5))   # 8
print(ist_gerade(4))   # True
print(ist_gerade(7))   # False
```

Aufgabe A2 — Was gibt der Code aus?

A2.1 — Immutable → Ausgabe: `1`. `a` innerhalb der Funktion ist eine lokale Variable, die das globale `a` nicht überschreibt, weil `int` immutable ist.

A2.2 — Mutable → Ausgabe: `[99, 2, 3]`. `a` ist eine Liste (mutable). Die Funktion verändert das Element der globalen Liste – das geht ohne `global`, weil die Liste selbst nicht neu zugewiesen wird.

A2.3 — Liste neu zugewiesen → Ausgabe: `[1, 2, 3]`. Hier wird `a` innerhalb der Funktion neu zugewiesen – das erzeugt eine neue lokale Variable und lässt die globale Liste in Ruhe.

A2.4 — global → Ausgabe: `3`. Mit `global` darf die Funktion das globale `a` tatsächlich überschreiben.

Aufgabe A3 — map, filter, reduce

```
from functools import reduce

zahlen = [3, 7, 2, 8, 5, 10, 1, 6]

# 1) Quadrate
quadrat = list(map(lambda x: x * x, zahlen))
print(quadrat)  # [9, 49, 4, 64, 25, 100, 1, 36]

# 2) Gerade Zahlen
gerade = list(filter(lambda x: x % 2 == 0, zahlen))
print(gerade)   # [2, 8, 10, 6]

# 3) Summe
summe = reduce(lambda a, b: a + b, zahlen)
print(summe)    # 42

# 4) Summe der Quadrate aller geraden Zahlen
ergebnis = reduce(
    lambda a, b: a + b,
    map(lambda x: x * x, filter(lambda x: x % 2 == 0, zahlen))
)
print(ergebnis)  # 4 + 64 + 100 + 36 = 204
```

Aufgabe A4 — Tupel, Set, Dictionary

```
# 1) Tupel
def min_max(liste):
    return min(liste), max(liste)

kleinstes, groesstes = min_max([3, 7, 1, 8, 4])
print(kleinstes, groesstes) # 1 8

# 2) Set
wuerfe = [1, 3, 6, 2, 3, 1, 5, 6, 4, 2]
unterschiedliche = set(wuerfe)
print(unterschiedliche) # {1, 2, 3, 4, 5, 6}

# 3) Dict
haeufigkeit = {}
for w in wuerfe:
    if w in haeufigkeit:
        haeufigkeit[w] += 1
    else:
        haeufigkeit[w] = 1
print(haeufigkeit) # {1: 2, 3: 2, 6: 2, 2: 2, 5: 1, 4: 1}
```

Aufgabe A5 — Currying

```
def addiere(x):
    return lambda y: x + y

plus_5 = addiere(5)
print(plus_5(3)) # 8
print(plus_5(10)) # 15
print(plus_5(-2)) # 3

# Bonus: hold_at als Funktions-Fabrik
def hold_at(schwelle):
    return lambda rundensumme: rundensumme < schwelle

hold_at_15 = hold_at(15)
hold_at_20 = hold_at(20)
hold_at_25 = hold_at(25)

print(hold_at_15(10)) # True
print(hold_at_20(20)) # False
print(hold_at_25(24)) # True
```

PHASE B — PIG FUNKTIONAL MACHEN

Stufe B1 — Pure functions

Funktion	Kategorie	Begründung
<code>hat_gewonnen</code>	pure	hängt nur von <code>punkte</code> ab, gibt Bool zurück
<code>computer_strategie</code>	pure	hängt nur von <code>rundensumme</code> ab
<code>zeige_runden_status</code>	nicht pure	<code>print</code> ist ein Nebeneffekt

Funktion	Kategorie	Begründung
<code>menschliche_strategie</code>	nicht pure	<code>input</code> liest „außen“
<code>spiele_runde</code>	nicht pure	<code>random</code> ist nicht-deterministisch, <code>print</code> , <code>input</code>
<code>spiele_pig</code>	nicht pure	aus denselben Gründen

Stufe B2 — Strategien als lambda

```
# 1) lambda statt def
hold_at_15 = lambda s: s < 15
hold_at_25 = lambda s: s < 25

# 2) Mit Currying: eine ganze Familie auf einmal
def hold_at(schwelle):
    return lambda rundensumme: rundensumme < schwelle

strategien = [hold_at(s) for s in [10, 15, 20, 25, 30]]
# strategien ist jetzt eine Liste mit 5 verschiedenen Strategie-Funktionen
```

Statt der List-Comprehension funktioniert auch `list(map(hold_at, [10, 15, 20, 25, 30]))` – noch etwas funktionaler.

Stufe B3 — Statistik mit map / filter / reduce

```
from functools import reduce

runden = alle_runden[0] # Beispiel: Spieler 1

# 1) Pech-Runden
pech = len(list(filter(lambda r: r[-1] == 1, runden)))

# 2) Längste Runde
laengste = max(map(len, runden))

# 3) Würfe insgesamt
gesamt_wuerfe = sum(map(len, runden))

# 4) Gerade Würfe: pro Runde zählen, dann aufsummieren
gerade_pro_runde = map(
    lambda r: len(list(filter(lambda w: w % 2 == 0, r))),
    runden
)
gerade = sum(gerade_pro_runde)

# Alternativ mit reduce:
# gerade = reduce(lambda acc, r: acc + sum(map(lambda w: w % 2 == 0, r)), runden, 0)
```

Stufe B4 — Tournament mit Zeitmessung

```
import time

start = time.time()
ergebnisse = list(map(
    lambda _: spiele_pig(hold_at(15), hold_at(25), verbose=False),
    range(1000)
))
dauer = time.time() - start

# Siege als Dict zaehlen
siege = {1: ergebnisse.count(1), 2: ergebnisse.count(2)}
print(f"Hold-at-15: {siege[1]} Siege")
print(f"Hold-at-25: {siege[2]} Siege")
print(f"Dauer: {dauer:.2f} Sekunden")

# Vergleich mit lazy map (ohne list):
start = time.time()
lazy = map(lambda _: spiele_pig(hold_at(15), hold_at(25), verbose=False), range(1000))
dauer_lazy = time.time() - start
print(f"Erstellung der lazy map: {dauer_lazy:.4f} Sekunden")
# Praktisch 0 - es wurde noch nichts gerechnet! (lazy evaluation)
```

Stufe B5 — Die mutable-Falle (Bonus)

Ja - `random.shuffle` verändert die übergebene Liste direkt (Nebeneffekt). Die pure Variante macht zuerst eine Kopie:

```
import random

def gemischt(liste):
    kopie = liste.copy()
    random.shuffle(kopie)
    return kopie

original = [1, 2, 3, 4, 5]
neu = gemischt(original)
print(original) # [1, 2, 3, 4, 5] - unverändert
print(neu)      # z.B. [3, 1, 5, 2, 4]
```