

Python: Sprache & Standardwerkzeuge

Python 3.10+ • Standardbibliothek • Beispiele sind unabhängig; freie Namen stehen für vorhandene Werte

01 Syntax, Werte & Typen

Blöcke: Doppelpunkt und Einrückung, üblicherweise 4 Leerzeichen; keine Tabs mischen. Kommentare beginnen mit #.

```
count = 3          # int
ratio = 0.75        # float
name = "Ada"        # str
is_ready = True     # bool
result = None       # kein Wert
```

Weitere Typen: complex, bytes, list, tuple, dict, set. Namen werden an Objekte gebunden; Typen gehören zu den Objekten.

```
type(count)
isinstance(count, int)
int("42"); float("2.5")
str(42); bool([]) # False
```

Wahrheitswerte: False, None, 0 und leere Container sind falsch. bool ist Unterklasse von int. Unterscheide **if value** und **if value is not None**.

02 Operatoren & Vergleiche

```
a + b; a - b; a * b
a / b      # echte Division
a // b     # abrunden, auch negativ
a % b      # Rest
a ** b     # Potenz
a += 1     # auch -=, *= usw.
```

Vergleiche: ==, !=, <, <=, >, >=; verkettet: 0 <= x < 10. **is** prüft Identität, == Werte. None mit is vergleichen.

Logik: not, and, or; Kurzschlussauswertung. and/or liefern Operanden, nicht zwingend bool. Zugehörigkeit: x in items; x not in items.

Bitweise: &, |, ^, ~, <<, >>. Klammern zur Klarheit verwenden. Float-Vergleich bei Rundung: math.isclose(a, b).

03 Ausgabe, Input & Diagnose

```
print(f"{count=}, {type(count)=}")
print(f"Quote: {ratio:.1%}")
print(f"Preis: {12.5:.2f} EUR")
text = input("Name: ")
```

input() liefert str. print(..., sep=";", end="") steuert Trennzeichen und Ende. repr(x) zeigt eine technische Darstellung; breakpoint() startet den Debugger.

04 Container & Zeichenketten

```
items = [10, 20, 30] # list
point = (3, 4)       # tuple
single = (3,)        # 1-Tupel
unique = {1, 2, 2}    # set
empty_set = set()    # {}: dict
user = {"name": "Ada"} # dict
```

Veränderbar: list, dict, set. **Unveränderbar:** Zahlen, str, tuple; ein Tupel kann veränderbare Objekte enthalten. dict bewahrt Einfügereihenfolge; set ist ungeordnet.

```
items.append(40); items.extend([50])
items.insert(0, 5); items.pop()
items.remove(20) # erster Treffer
user.get("age", 0)
user.keys(); user.values()
user.items(); user.update(age=25)
unique.add(3); unique.discard(9)
```

Schlüssel und Set-Elemente müssen hashbar sein, z. B. str oder int; keine list. len(x), x in container. sort() ändert die Liste und liefert None; sorted(x) erzeugt eine neue Liste.

```
text = "  Ada Lovelace  "
text.strip().lower()
text.split() # Leerraum
";".join(["Ada", "Grace"])
text.replace("Ada", "A.")
text.startswith("  ")
```

05 Index, Slice & Kopien

```
items[0]; items[-1]
items[1:3] # Ende exklusiv
items[:2]; items[:2]
items[::-1] # umgekehrt
a, b = (3, 4)
first, *rest = [1, 2, 3]
```

Referenz: other = items teilt dasselbe Objekt. items.copy() / items[:] kopiert flach; verschachtelte Objekte bleiben geteilt. Für tiefe Kopie: copy.deepcopy(items).

Fehler: ungültiger Index → IndexError; fehlender dict-Schlüssel → KeyError. Während einer Iteration Container nicht unbedacht verändern.

06 Bedingungen & Schleifen

```
if score >= 90:
    grade = "sehr gut"
elif score >= 75:
    grade = "gut"
else:
    grade = "weiter üben"
label = "ja" if score > 0 else "nein"
```

```
for i in range(3): # 0, 1, 2
    print(i)
for i, name in enumerate(names, 1):
    print(i, name)
for key, value in user.items():
    print(key, value)
while count > 0:
    count -= 1
```

range(start, stop, step): stop exklusiv. **break** beendet, **continue** überspringt einen Durchlauf. Schleifen-else läuft bei normalem Ende, nicht nach break.

```
squares = [x*x for x in range(5)]
even = [x for x in items if x%2 == 0]
total = sum(x*x for x in range(5))
```

zip() endet standardmäßig an der kürzesten Eingabe. **zip(..., strict=True)** meldet verschiedene Längen. Generatordruck (...) berechnet bei Bedarf; Iteratoren können verbraucht werden.

07 Funktionen & Gültigkeit

```
def greet(name: str, *, loud=False):
    """Erzeugt einen Gruß."""
    text = f"Hallo {name}"
    return text.upper() if loud else text

greet("Ada", loud=True)
```

Parameter: / markiert nur positional, * nur keyword; *args sammelt positionale, **kwargs benannte Argumente. Aufruf: func(*values, **options). Ohne return wird None zurückgegeben.

Scope: lokal → umschließende Funktion → Modul → builtins (LEGB). global / nonlocal ändern Bindungen; meist Werte über Parameter und return austauschen.

Defaults: einmal bei Definition ausgewertet! Keine veränderbare Liste als Standard. Typannotationen dokumentieren und helfen Prüfwerkzeugen; sie erzwingen Typen nicht zur Laufzeit.

08 Exceptions & Ressourcen

```
try:
    number = int(text)
except ValueError as exc:
    print(f"Ungültige Zahl: {exc}")
else:
    print(number)
finally:
    print("fertig")
```

Nur erwartete Fehler gezielt fangen. **raise ValueError(...)** signalisiert ungültige Werte. **raise** wirft den aktuellen Fehler erneut; **raise ... from exc** bewahrt die Ursache.

```
with open("data.txt",
          encoding="utf-8") as file:
    text = file.read()
```

with verwaltet Ressourcen auch bei Fehlern. Modi: r lesen, w überschreiben, a anhängen, b binär. assert für interne Annahmen / Tests; für Eingabeprüfung ungeeignet, da mit -O abschaltbar.

09 Module, Klassen & Protokolle

```
import math
from pathlib import Path

class Counter:
    def __init__(self, start=0):
        self.value = start

    def increment(self):
        self.value += 1

counter = Counter()
counter.increment()
```

self: Instanz; **cls:** Klassenmethoden (@classmethod). @staticmethod ohne Instanzbezug. Vererbung: class Child(Base); super() ruft Basisverhalten auf.

Protokolle: __str__ Text, __repr__ Diagnose, __len__ Länge, __iter__ Iteration. @property: Attributzugriff mit Logik.

yield: Generator liefert Werte schrittweise. **@name:** Dekorator. **async def / await:** asynchrone I/O, keine automatische CPU-Parallelität.

10 Nützliche Builtins

len, sum, min, max, abs, round; sorted, reversed; any / all; iter / next. **lambda x: x[1]** als kurze Schlüssel-Funktion.

Python: Codefragmente & Codedesign

Wiederkehrende Muster mit Erklärung • Klare Verträge, lesbarer Code und sinnvolle Namen

11 Einstiegspunkt & Argumente

```
import argparse
def main() -> int:
    parser = argparse.ArgumentParser()
    parser.add_argument("path")
    args = parser.parse_args()
    print(args.path)
    return 0
if __name__ == "__main__":
    raise SystemExit(main())
```

Direkter Start führt main() aus; beim Import bleibt main() unaufgerufen. argparse erstellt Hilfe und prüft Argumente. Exitcode 0 bedeutet Erfolg.

12 Dateien, JSON & CSV

```
from pathlib import Path
import json

with Path("settings.json").open(
    encoding="utf-8") as f:
    settings = json.load(f)

with open("settings.json", "w",
    encoding="utf-8") as f:
    json.dump(settings, f,
        ensure_ascii=False, indent=2)
```

JSON: Objekt → dict, Array → list, null → None. w überschreibt! Relative Pfade gelten ab Arbeitsverzeichnis.

```
import csv

with open("data.csv", newline="",
    encoding="utf-8") as f:
    for row in csv.DictReader(f,
        delimiter=";"):
        print(row["name"])
```

CSV-Modul statt split() verwenden: Anführungszeichen und Trennzeichen werden korrekt verarbeitet. Felder sind zunächst str; Zahlen ausdrücklich konvertieren.

13 Projekt starten (Konsole)

```
python -m venv .venv
source .venv/bin/activate
.venv\Scripts\Activate.ps1
python -m pip install paketname
```

Shell-Befehle; Windows ggf. py statt python. Aktivieren: source unter Linux, Activate.ps1 unter PowerShell. Abhängigkeiten in pyproject.toml pflegen.

14 Validieren & sauber rechnen

```
def mean(
    values: list[float],
) -> float:
    """Mittelwert nichtleerer Werte."""
    if not values:
        raise ValueError("Leere Liste")
    return sum(values) / len(values)

assert mean([2.0, 4.0]) == 3.0
```

Grenzfälle ausdrücklich festlegen: leer, fehlend, ungültig. Bei öffentlicher API prüfen, was der Vertrag verlangt; keine stillen Ersatzwerte für Fehler.

```
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
```

None verhindert eine geteilte Standardliste. Übergebene Listen werden hier verändert: Seiteneffekt dokumentieren.

15 Gruppieren, zählen & sortieren

```
from collections import Counter
from collections import defaultdict

counts = Counter(["a", "b", "a"])
groups = defaultdict(list)
for name in ["Ada", "Alan", "Grace"]:
    groups[name[0]].append(name)
rows = [("Ada", 9), ("Alan", 7)]
ranked = sorted(rows,
    key=lambda row: row[1],
    reverse=True)
```

Counter zählt; defaultdict erzeugt fehlende Werte beim Schlüsselzugriff. sorted ist stabil und erzeugt eine neue Liste.

16 Streaming & Logging

```
import logging

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)
logger.info("Anzahl: %d", 42)
```

Dateizeilen iterieren spart Speicher. Logging lässt sich nach Schweregrad filtern; Konfiguration gehört in den Programmeinstieg, nicht in Bibliotheksmodule.

17 Datenklassen & Tests

```
from dataclasses import (
    dataclass, field,
)

@dataclass
class Student:
    name: str
    scores: list[float] = field(
        default_factory=list)

student = Student("Ada")
student.scores.append(12.0)
```

dataclass erzeugt u. a. __init__, __repr__ und Gleichheitsvergleich. default_factory gibt jeder Instanz eine eigene Liste; Annotationen validieren keine Eingaben.

```
import unittest

def square(value):
    return value * value

class SquareTests(unittest.TestCase):
    def test_positive(self):
        self.assertEqual(square(3), 9)

    def test_negative(self):
        self.assertEqual(square(-2), 4)
```

Start: **python -m unittest**. Prüfe normale Fälle, Grenzen und Fehler. Teste Verhalten und Vertrag, nicht bloß dieselbe Rechnung nochmals.

18 Gutes Codedesign

Eine Aufgabe je Funktion: klare Eingaben und Rückgaben. Fachlogik von Datei-, Netzwerk- und UI-Zugriff trennen; so wird sie leichter testbar.

Klare Verträge: Einheiten, erlaubte Werte, Seiteneffekte und Fehler dokumentieren. Kurze Docstrings erklären Zweck; Kommentare begründen Entscheidungen.

Lesbarkeit: frühe Rückgaben gegen tiefe Verschachtelung; einfache Schleifen statt verschachtelter Comprehensions. Wiederholung sinnvoll bündeln, keine vorschnellen Abstraktionen.

Zustand: globale veränderbare Daten vermeiden. Klassen für zusammengehörige Daten und Verhalten; einfache Funktionen für unabhängige Berechnungen.

Qualität: Formatierung und Projektstil einheitlich halten. Typprüfung, Linting und Tests ergänzen einander. Performance erst messen, dann optimieren.

19 Namen: Konvention & Beispiele

Funktionen / Methoden: **snake_case** calculate_total(), load_config(), save_report(), parse_date(), validate_email(). Verben benennen die Tätigkeit.

Variablen / Attribute: **snake_case** student_count, input_path, total_price, elapsed_seconds. Substantive benennen Inhalte; Einheiten bei Mehrdeutigkeit ergänzen.

Klassen: **CapWords** Student, Invoice, SensorReading, ReportGenerator. Substantive für das Konzept; Instanz: sensor_reading.

Konstanten: **UPPER_SNAKE_CASE** MAX_RETRIES, DEFAULT_TIMEOUT_SECONDS. Python erzwingt keine Unveränderbarkeit.

Module / Pakete: kurze Kleinbuchstaben; Module ggf. snake_case: data_loader.py. Eigene Dateien nicht json.py, csv.py oder typing.py nennen: verdeckt Standardmodule.

Booleans: is_valid, has_errors, can_retry. Klar positiv formulieren: is_enabled statt not_disabled. Prädikat-Funktion z. B. is_even(number).

Sammlungen: students, scores, paths. Aussagekräftige Zuordnung: price_by_product, students_by_class. i/j für kurze Indizes, x/y für Mathematik passend.

Namenskonflikt: class_ statt class. Builtins wie list, dict, str, sum nicht als eigene Variablennamen verwenden. _helper signalisiert intern; __name löst in Klassen Name Mangling aus, keine Zugriffssperre.

Übliche Rollen: self für Instanz, cls für Klasse, args/kwargs für Argumente, exc für Exception. _ für absichtlich unbenutzten Wert.

Ausnahmen: ValueError, ConfigurationError; Suffix Error üblich. Reservierte __dunder__-Namen nur für definierte Python-Protokolle.

Sprache: englische Namen sind in gemeinsamen Projekten üblich. Deutsch ist möglich; pro Projekt konsistent bleiben. Keine kryptischen Kürzel wie calcData2 oder temp, wenn die Rolle unklar ist.

20 Häufige Stolperstellen

None statt Ergebnis: items = items.sort() ist falsch; sort() liefert None. Nutze items.sort() oder sorted(items).

Verschachtelte Listen: [[0] * 3] * 2 teilt Zeilen. Besser: [[0] * 3 for _ in range(2)].

Mutationen: kopierte Referenzen teilen Daten. Bei Filtern neue Listen erzeugen. Kein pauschales except: pass; Fehler nicht verschlucken.