Objektorientiertes Programmieren (OOP)

```python
# Problematisch: Datenmanagement ohne OOP
student_grades = {}
student_courses = {}

def take_course(student_name, course):
    # Speichere Kursbelegung
    if student_name not in student_courses:
        student_courses[student_name] = []
    student_courses[student_name].append(course)

def add_grade(student_name, course, grade):
    # Speichere Note
    if student_name not in student_grades:
        student_grades[student_name] = {}
    student_grades[student_name][course] = grade

def calculate_average(student_name):
    # Berechne Durchschnitt
    if student_name not in student_grades:
        return 0
    grades = student_grades[student_name].values()
    return sum(grades) / len(grades) if grades else 0
```

```python
# Benutzung ist fehleranfällig:
take_course("Alice", "C")
add_grade("Alice", "Python", 85)

print(calculate_average("Alice"))  # 85.0

print(calculate_average("Alise"))
# 0.0 - Ups, Tippfehler!
```

Gängige Probleme bei getrennten Daten

– unzureichende Datenkonsistenz
– oft unklar, auf welchen Daten Funktionen arbeiten
– dadurch fehleranfällig und schwer zu warten

OOP verspricht Abhilfe

– Daten und Funktionen (Methoden) bilden eine Einheit
– ermöglicht konsequente Konsistenzprüfungen
– hilfreich zur Strukturierung größerer Projekte
– potenziell bessere Selbstbeschreibungsfähigkeit des Codes

```python
class Student:
    def __init__(self, name):
        self.name = name
        self.grades = {}   # Noten pro Kurs
        self.courses = []  # Belegte Kurse

    def add_grade(self, course, grade):
        if course not in self.courses:
            print(f"{self.name} belegt "
                  f"{course} nicht.")
            return False

        if not (0 <= grade <= 100):
            print(f"Note {grade} ungültig (0-100 erlaubt)")
            return False

        self.grades[course] = grade
        return True

    def enroll_course(self, course):
        if course in self.courses:
            print(f"{self.name} bereits in {course} eingeschrieben.")
            return False
        self.courses.append(course)
        return True
```

```python
# Wir profitieren von besserer Fehlerbehandlung:
alice = Student("Alice")
alice.add_grade("Python", 85)   # Fehler: belgt Kurs nicht
alice.enroll_course("Python")
alice.add_grade("Python", 185)  # Fehler: ungültige Note
alice.add_grade("Python", 85)   # Erfolg
```

OOP-Kernkonzepte

nicht nur in Python, sondern auch z.B. in Java, C++, Ruby, …

- *Encapsulation (Kapselung)*
  - Daten und Methoden als Einheit
  - geschützter Zugriff auf interne Daten
- *State Management*
  - jedes Objekt verwaltet seinen eignen Zustand
  - Objekte stellen Konsistenz sicher; validieren Daten
- *Explizite Schnittstellen*
  - definierte Methoden für Interaktion mit Daten
  - Implementierungsdetails versteckt
- *Potenzielle Vorteile*
  - Wiederverwendbarkeit durch Modularität
  - bessere Wartbarkeit durch klarere Struktur
  - reduzierte Fehleranfälligkeit

```python
# Wir haben schon objektorientiert programmiert ohne es zu merken!

text = "Hello"              # String-Objekt
print(type(text))           # <class 'str'>
print(text.upper())         # Methodenaufruf!

numbers = [1, 2, 3]         # Listen-Objekt
print(type(numbers))        # <class 'list'>
numbers.append(4)           # Noch eine Methode!


# Sogar Funktionen sind Objekte
def greet():
    print("Hi")

print(type(greet))          # <class 'function'>
x = greet  # x zeigt nun auf das greet-Objekt
x()        # ruft greet() auf
```

OOP-Bausteine
im Überblick

- *Klassen* *(Namenskonvention:* `PascalCase`*)*
  - Bauplan für Objekte
  - definieren Attribute und Verhalten
  - Keyword: `class`
- *Objekte*
  - konkrete Instanzen, die aus einer Klasse erstellt werden
  - voneinander unabhängig; jeweils eigener Zustand
- *Attribute und Methoden* *(Namenskonvention:* `snake_case`*)*
  - Zugriff mittels Punkt-Operator
  - Attribute: Daten eines Objekts (*ohne* Klammern)
  - Methoden: Funktionen, die ein Objekt bereitstellt (*mit* Klammern)
  - `self` ist in Methoden eine Referenz auf das aktuelle Objekt

```python
# Beispiel: Geometrische Formen
class Rectangle:
    def __init__(self, width, height):
        # Initialisierung der Attribute
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        # Berechne Umfang
        return 2 * (self.width + self.height)


small_rect = Rectangle(3, 4)
big_rect = Rectangle(10, 20)

print(small_rect.area())      # 12
print(big_rect.perimeter())   # 60
```

Klassen

- fungieren als Template oder Blueprint für Objekte
- definieren Daten und Verhalten an *einem* Ort im Code
- Dokumentation durch *Docstrings* (dreifache Anführungszeichen)
- Schreibweise von Klassennamen: `PascalCase`
  (also `camelCase`, aber mit großem Anfangsbuchstaben)

```
class Student:
    """Eine Person mit Kursen und Noten.

    Jede Person kann mehrere Kurse belegen.
    Pro Kurs kann eine Note hinterlegt werden.
    """

    # Methoden folgen hier...
```

Aus Klassen Instanzen erstellen

– Objekte sind konkrete Exemplare, die aus einer Klasse erstellt werden

– jedes Objekt hat seinen eigenen Zustand und existiert unabhängig von anderen

– Schreibweise der Namen von Objekten (wie bei allen Variablen): snake_case

```
# Wir erstellen zwei Objekte
alice = Student("Alice")
bob = Student("Bob")

# Aufruf einer Methode
alice.enroll_course("Python")
bob.enroll_course("Java")

# Zugriff auf Attribut
print(alice.courses)  # ['Python']
print(bob.courses)    # ['Java']
```

Die Verbindung zum Objekt: `self`

`self` referenziert aktuelles Objekt; erster Parameter einer Methode; wird beim Aufruf einer Methode automatisch befüllt.

```python
class Student:
    def __init__(self, name):
        self.name = name
        self.courses = [] # Bei Objekt-Erstellung leere Liste anlegen

    def enroll_course(self, course):
        # self verweist auf die konkrete Instanz
        self.courses.append(course)

# Wenn wir schreiben ...
alice = Student("Alice")
alice.enroll_course("Python")

# ... macht Python daraus:
Student.enroll_course(alice, "Python")
```

Der Initialisierer `__init__` *(ähnelt dem sog. Konstruktur in anderen Sprachen)*

Setzt Anfangszustand; wird automatisch beim Erstellen aufgerufen;
kann Daten validieren und Attribute initialisieren; dokumentiert Objektstruktur.

```
class Student:
    def __init__(self, name, student_id):
        """Initialisiere Student-Objekt.

        Args:
            name (str): Name der Person
            student_id (str): Eindeutige ID
        """
        self.name = name
        self.student_id = student_id

        self.grades = {}
        self.courses = []
```

Instanzattribute (pro Objekt) versus Klassenattribute (von allen geteilt)

```python
class Student:
    # Klassenattribut
    school_name = "Python High School"

    def __init__(self, name, student_id):
        # Instanzattribute
        self.name = name
        self.student_id = student_id

alice = Student("Alice", "A123")
bob = Student("Bob", "B456")

# Zugriff auf Klassenattribut
Student.school_name = "Python Academy"
print(alice.school_name)  # "Python Academy"
print(bob.school_name)    # "Python Academy"

# So nicht! Erzeugt neues Instanzattribut on-the-fly!
alice.school_name = "Private School"
print(alice.school_name)  # "Private School"
print(bob.school_name)    # "Python Academy"
```

Kapselung und Zugriffskontrolle

Direkt zugängliche Attribute sind riskant: keine Validierung oder Protokollierung möglich; keine Kontrolle über Änderungen; besser: private Attribute und Methoden.

```
class Student:
    def __init__(self, name, student_id):
        self.name = name          # Öffentliches Attribut: riskant
        self._id = student_id     # besser: privates Attribut (_ als Präfix))
        self._grades = {}         # auch privat (manchmal sieht man auch __ als Präfix)

    def _calculate_average(self):  # Private Methode (ebenfalls _ als Präfix)
        if not self._grades:       # Methoden haben Zugriff auf alle Attribute.
            return 0.0
        return sum(self._grades.values()) / len(self._grades)

    def add_grade(self, course, grade):
        if not (0 <= grade <= 100):  # Validierung vor Änderung
            print("Ungültige Note")
            return
        self._grades[course] = grade
```

Methoden zum Attribut-Zugriff für eine saubere Schnittstelle

```python
class Student:
    def __init__(self, name):
        self.set_name(name)  # Validierung
        self._attendance = {}

    def get_name(self):
        # strip() löscht alle Whitespace-Zeichen vorn und hinten
        return self._name.strip()

    def set_name(self, name):
        # Fehlerbehandlung mittels Exceptions (werden später erklärt)
        if not isinstance(name, str):
            raise ValueError("Name muss ein String sein")
        if not name.strip():
            raise ValueError("Name darf nicht leer sein")
        self._name = name

    def log_attendance(self, date, present):
        if not isinstance(present, bool):
            raise ValueError("Status muss boolean sein")
        self._attendance[date] = present
```

Properties vereinfachen die Zugriffskontrolle.

- *Motivation*
  - Sind Attribute öffentlich zugänglich, verliert ein Objekt die Kontrolle darüber. Ungünstig!
  - Private Attribute, auf die nur mit Getter- und Setter-Methoden zugegriffen werden kann, sind aber recht umständlich.
- *Properties …*
  - … ermöglichen es, den Zugriff auf private Attribute mit Methoden zu regeln, aber dennoch einfach (d.h. scheinbar ohne Methodenaufruf) darauf zuzugreifen.
  - Typische Anwendungsfälle: Validierung von Daten, On-the-fly-Berechnungen, formatierte Ausgabe.

Beispiel für ein Problem bei direktem Zugriff

```
class BankAccount:
    def __init__(self, initial_balance):
        self.balance = initial_balance  # öffentliches Attribut


konto = BankAccount(1000)
konto.balance = -50  # Ungültige Änderungen können dann nicht verhindert werden.
```

Beispiel für Getter- und Setter-Methoden: guter Zugriffsschutz, aber umständlich

```
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = 0  # privates Attribut
        self.set_balance(initial_balance)

    def get_balance(self):
        return self._balance

    def set_balance(self, value):
        if value < 0:
            print("Fehler: Negativer Kontostand!")
            return
        self._balance = value

# Umständliche Verwendung
konto = BankAccount(1000)
aktuell = konto.get_balance()    # Getter-Methode für Zugriff
konto.set_balance(aktuell + 500) # Setter-Methode für Änderung
```

Eleganter mit Properties

```python
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = 0         # privates Attribut
        self.balance = initial_balance  # nutzt schon die Property

    @property  # macht Methode zu Property (read-only)
    def balance(self):
        """Kontostand auslesen: hier einfach nur Wert zurückgeben"""
        return self._balance

    @balance.setter  # für Schreibzugriff
    def balance(self, value):
        """Ändern des Kontostands mit Validierung"""
        if value < 0:
            print("Fehler: Negativer Kontostand!")
            return
        self._balance = value

# Elegante Verwendung
konto = BankAccount(1000)
print(konto.balance)      # Wie normales Attribut
konto.balance += 500      # Aber mit Validierung!
```

Weiterer Anwendungsfall: berechnete Properties

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius  # privates Attribut

    @property
    def radius(self):
        return self._radius

    @property
    def area(self):  # nur Getter -> Read-only
        """Fläche wird bei Zugriff berechnet"""
        return 3.14159 * self._radius ** 2

    @property
    def diameter(self):  # auch nur read-only
        """Durchmesser aus Radius berechnet"""
        return 2 * self._radius
```

Anwendungsfälle von Properties

- *Validierung*
  - Sicherstellen gültiger Wertebereiche
  - Typprüfung
- *Berechnete Werte*
  - Berechnung (erst) beim Zugriff
  - kein duplizierter Code zur Aktualisierung nötig, wenn sich der Zustand des Objekts ändert
- *Ausgabe und Kontrolle*
  - Protokollierung von Änderungen
  - Schreibgeschützte Werte
  - Formatierung von Werten

Spezielle Methoden

– Manchmal ist gewünscht, dass Objekte mit Standardoperationen verwendet werden können.

– Dazu müssen speziellle Methoden definiert werden, die doppelte Unterstriche als Präfix und Suffix haben (**„Dunder Methods“**, Anspielung auf *double underscore*).

– Wenn vorhanden, werden sie von Python automatisch genutzt. Vorteil: einfache, bekannte Syntax verwendbar.

```
__str__  : print(obj)    # String-Darstellung
__len__  : len(obj)      # Länge des Objekts
__getitem__ : obj[i]     # Index-Zugriff
__eq__   : obj1 == obj2  # Gleichheitsvergleich
__lt__   : obj1 < obj2   # Kleiner-Als-Vergleich
```

Beispielhafte Verwendung der Dunder Methods

```python
class GradeBook:
    def __init__(self):
        self._grades = []

    def __str__(self):
        return f"Notenbuch mit {len(self._grades)} Noten"

    def __len__(self):
        return len(self._grades)

    def __getitem__(self, index):
        return self._grades[index]

    def __eq__(self, other):
        """Vergleich mit == prüft Inhalt"""
        if not isinstance(other, GradeBook):
            return False
        return self._grades == other._grades

buch = GradeBook()
print(buch)              # Nutzt __str__
print(len(buch))         # Nutzt __len__
if len(buch) > 0:        # Natürliche Verwendung
    erste_note = buch[0] # Nutzt __getitem__
```

Identität versus Gleichheit

```
rect1 = Rectangle(width=5, height=3)
rect2 = Rectangle(width=5, height=3)  # Gleiche Werte!
rect3 = rect1                         # Gleiche Referenz!

# Identität mit 'is'
print(rect1 is rect2)   # False - verschiedene Objekte
print(rect1 is rect3)   # True - gleiches Objekt

# Gleichheit mit '=='
print(rect1 == rect2)   # True (wenn __eq__ implementiert)


# Implementierung von __eq__ für Rectangle
class Rectangle:
    # ...

    def __eq__(self, other):
        if not isinstance(other, Rectangle):
            return False
        return (self.width == other.width and
                self.height == other.height)
```

EXTRAS IN 3 MINUTEN

FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Aufgabe 1

Erstellen Sie eine Book-Klasse mit Titel und Autor.

Fügen Sie eine Methode für die String-Darstellung hinzu.

```
# Ihre Lösung?
```

Aufgabe 1

Erstellen Sie eine Book-Klasse mit Titel und Autor.

Fügen Sie eine Methode für die String-Darstellung hinzu.

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} von {self.author}"


buch = Book("Python Basics", "John Smith")
print(buch)  # Python Basics von John Smith
```

Aufgabe 2

Erstellen Sie eine Rectangle-Klasse, bei der Breite und Höhe nicht negativ sein dürfen. Nutzen Sie Properties!

```
# Ihre Lösung?
```

```python
class Rectangle:
    def __init__(self, width, height):
        self._width = 0
        self._height = 0
        self.width = width    # Nutzt Property
        self.height = height  # Nutzt Property

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        if value <= 0:
            print("Breite muss positiv sein!")
            return  # besser: hier Exception
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value <= 0:
            print("Höhe muss positiv sein!")
            return  # besser: hier Exception
        self._height = value
```

Aufgabe 3

Verbessern Sie die folgende Student-Klasse, indem Sie die Attribute über Properties zugänglich machen.

Die Funktionalität soll gleich bleiben, aber die Verwendung soll eleganter werden.

```python
class Student:
    def __init__(self, name, student_id):
        self._name = name
        self._id = student_id
        self._grades = {}

    def get_name(self):
        return self._name

    def set_name(self, value):
        if not isinstance(value, str):
            print("Name muss ein String sein!")
            return
        if not value.strip():
            print("Name darf nicht leer sein!")
            return
        self._name = value.strip()

    def get_average(self):
        if not self._grades:
            return 0.0
        return sum(self._grades.values()) / len(self._grades)

    def get_passed_courses(self):
        return [course for course, grade in self._grades.items() if grade >= 4.0]
```

```python
# Aktuell umständliche Verwendung:

student = Student("Alice", "A123")
print(student.get_name())
student.set_name("Alice Smith")
print(student.get_average())
print(student.get_passed_courses())
```

```python
class Student:
    def __init__(self, name, student_id):
        self._id = student_id
        self._grades = {}
        self.name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            print("Name muss ein String sein!")
            return
        if not value.strip():
            print("Name darf nicht leer sein!")
            return
        self._name = value.strip()

    @property
    def average(self):
        if not self._grades:
            return 0.0
        return sum(self._grades.values()) / len(self._grades)

    @property
    def passed_courses(self):
        return [course for course, grade in self._grades.items() if grade >= 4.0]
```

```python
# Elegantere Verwendung:
student = Student("Alice", "A123")
print(student.name)
student.name = "Alice Smith"
print(student.average)
print(student.passed_courses)
```

Klassen sind Baupläne für Objekte.

Klassen definieren sowohl Attribute (Daten) als auch Methoden (Verhalten).

Der Initialisierer __init__ setzt den Grundzustand.

Dadurch dort implizite Dokumentation der genutzten Attribute.

Objekte sind unabhängige Instanzen.

Haben ihren eigenen Zustand, aber alle das gleiche, in der Klasse definierte Verhalten.

self verbindet Methoden mit ihrem Objekt.

Erster Parameter aller Methoden.

Wird beim Methoden-Aufruf automatisch übergeben.

Properties ermöglichen elegante Kapselung von internen Attributen.

Vereinen die Attribut-Syntax mit Methodenfunktionalität.

Spezielle Methoden machen Objekte besser benutzbar:

__str__, __len__, __eq__, …