



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Kapitel 02

CODE-QUALITÄT IN DER PRAXIS

Wenn funktionierender Code nicht ausreicht

Das Szenario:

```
$ git push origin feature/road-profile-viewer
```

Running automated checks...

- ✗ Ruff check: 14 errors found
- ✗ Format check: 8 files need formatting
- ✗ Type check: 6 type errors detected

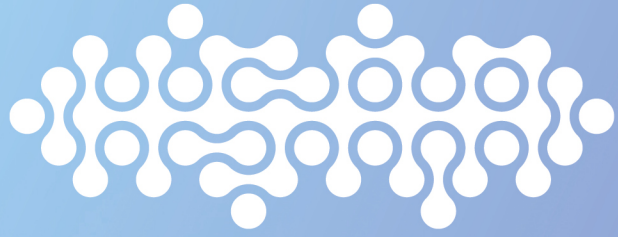
Pull request blocked - fix issues before merging

Ihr Code *funktioniert...* aber er erfüllt nicht die professionellen Standards.

Lernziele

Am Ende dieser Vorlesung werden Sie:

1. ☒ **Verstehen**, worin der Unterschied zwischen funktionierendem und professionellem Code besteht
2. ☒ **Erkennen** häufiger Code-Qualitätsprobleme anhand realer Beispiele
3. ☒ **Kennenlernen** von Pythons Stilrichtlinie (PEP 8) und warum sie wichtig ist
4. ☒ **Beherrschen** moderner Werkzeuge: Ruff (Linter/Formatter) und Pyright (Typprüfer)
5. ☒ **Anwenden** automatisierter Prüfungen, um Probleme vor dem menschlichen Review zu erkennen
6. ☒ **Integrieren** von Qualität in Ihren Workflow von Tag eins an



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Die Code-Qualitäts-Reise

1

Stufe 1: Können Sie das lesen?

Nehmen Sie sich 30 Sekunden. Was macht diese Funktion?

```
def f(d, t=50):  
    s = 0  
    c = 0  
    for i in d:  
        if i >= 0 and i <= 100:  
            s = s + i  
            c = c + 1  
    if c == 0:  
        return 0, "NA"  
    a = s / c  
    if a >= t:  
        r = "PASS"  
    else:  
        r = "FAIL"  
    return a, r
```

Wie lange haben Sie gebraucht, um **f**, **d**, **t**, **s**, **c**, **a**, **r** zu entschlüsseln?

Stufe 2: Namen helfen... oder doch nicht?

Gleiche Funktion, jetzt mit "echten" Namen:

```
def Calculate_Result(ScoreList, passingThreshold=50):
    Total_sum = 0
    COUNT = 0
    for currentScore in ScoreList:
        if currentScore >= 0 and currentScore <= 100:
            Total_sum = Total_sum + currentScore
            COUNT = COUNT + 1
    if COUNT == 0:
        return 0, "NA"
    Average = Total_sum / COUNT
    if Average >= passingThreshold:
        Status = "PASS"
    else:
        Status = "FAIL"
    return Average, Status
```

Sechs Konventionen in einer Funktion

Zählen Sie die Benennungsstile:

Variable	Konvention
Calculate_Result	PascalCase + Unterstrich
ScoreList	PascalCase
passingThreshold	camelCase
Total_sum	Gemischt
COUNT	SCREAMING_CASE
currentScore	camelCase

Ihr Gehirn muss für jede Variable das Muster neu erlernen.

Inkonsistenz kostet *kognitive Reibung* in jeder Zeile.

Stufe 3: Konsistent, aber sauber?

Konsistentes `snake_case`, aber...

```
import math
import os

def calculate_result(score_list, passing_threshold=50):
    total_sum = 0
    count = 0
    for score in score_list:
        if score >= 0 and score <= 100:
            total_sum = total_sum + score
            count = count + 1
    if count == 0:
        return 0, "NA"
    average = total_sum / count
    ...
```

- `import math`, `import os` -- **unbenutzte Imports**
- Kein **Docstring** -- was macht diese Funktion?

Stufe 4: Professionelle Qualität

```
def calculate_result(
    scores: list[float],
    passing_threshold: float = 50.0,
) -> tuple[float, str]:
    """Calculate average exam score and determine pass/fail.

    Filters out invalid scores (outside 0-100 range).
    """
    valid_scores = [s for s in scores if 0 <= s <= 100]

    if not valid_scores:
        return 0.0, "NA"

    average = sum(valid_scores) / len(valid_scores)
    status = "PASS" if average >= passing_threshold else "FAIL"

    return average, status
```

Gleiche Logik -- aber in Sekunden lesbar, nicht in Minuten.

Stufe 5: Zwei Richtige ergeben keinen Standard

Version A -- Kompakt:

```
def calculate_result(
    scores: list[float],
    passing_threshold: float = 50.0,
) -> tuple[float, str]:
    """Return (average, status)."""
    valid = [s for s in scores
              if 0 <= s <= 100]
    if not valid:
        return 0.0, "NA"
    avg = sum(valid) / len(valid)
    return avg, ("PASS" if avg
                >= passing_threshold
                else "FAIL")
```

Version B -- Explizit:

```
def calculate_result(
    scores: list[float],
    passing_threshold: float = 50.0,
) -> tuple[float, str]:
    """Durchschnitt berechnen und
    Bestanden/Nicht bestanden."""
    valid_scores = [
        score for score in scores
        if 0 <= score <= 100
    ]
    if len(valid_scores) == 0:
        return 0.0, "NA"
    average = (sum(valid_scores)
               / len(valid_scores))
    if average >= passing_threshold:
        status = "PASS"
    else:
        status = "FAIL"
    return average, status
```

Warum Style Guides existieren

Beide Versionen sind korrekt, lesbar und bestehen jeden Linter.

Aber stellen Sie sich vor, die Hälfte des Teams schreibt Version A, die andere Version B.

Jede Datei liest sich anders. Jede Funktion wirkt unvertraut.

Style Guides geben Ihrer Codebasis eine konsistente Identität -- wie Markenrichtlinien für visuelles Design.

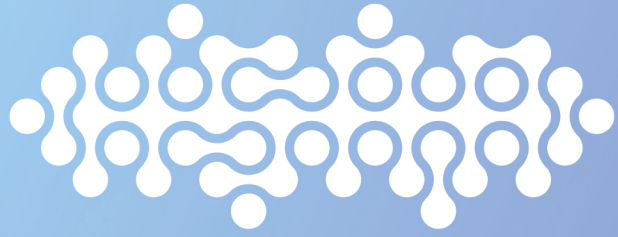
Was sich änderte	Warum es wichtig ist
Aussagekräftige Namen	Verstehen ohne Nachverfolgung
Konsistente Konventionen	Gehirn kämpft nicht gegen Format
Kein toter Code/Imports	Nichts lenkt ab
Docstrings + Type Hints	Funktion erklärt sich selbst
Ein gemeinsamer Style Guide	Liest sich wie ein Autor

Stellen Sie sich das im großen Maßstab vor

Nicht eine 15-Zeilen-Funktion, sondern **2.000 Zeilen in 15 Dateien**.

- Stufe 1 → Stufe 4: der Unterschied zwischen einer wartbaren Codebasis und einer, bei der Sie aufgeben wollen
- Ohne Style Guide: eine Codebasis, die sich wie ein Flickwerk anfühlt

Schauen wir uns an, was passiert, wenn wir automatisierte Qualitäts-Tools auf ein echtes Projekt loslassen...



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Der Realitätscheck

2

Unser Code unter der Lupe

Schauen wir uns „funktionierenden“ Code aus unserem Road Profile Viewer an:

```
import numpy as np
from dash import Dash, html, dcc, Input, Output
import sys,os # Multiple imports on one line

def generate_road_profile(num_points=100,x_max=80):
    y=0.015 * x_norm**3 * x_max
    return x,y

def HelperFunction(val): # Never used
    result=val*2
    return result
```

Sieht vernünftig aus, oder? Schauen wir, was die Werkzeuge sagen...

Automatisierte Prüfungen ausführen

```
$ uv run ruff check .
```

```
main.py:16:1: E401 Multiple imports on one line  
main.py:16:8: F401 `sys` imported but unused  
main.py:29:5: E225 Missing whitespace around operator  
main.py:36:1: N802 Function name should be lowercase  
main.py:36:1: F841 Local variable is assigned but never used  
main.py:47:1: E501 Line too long (149 > 88 characters)
```

```
Found 14 errors.
```

14 Fehler in Code, der einwandfrei funktioniert!



Diskussion: Warum gibt es diese Regeln?

Nehmen Sie sich 2 Minuten mit Ihrem Nachbarn:

Betrachten Sie diese Fehler: `E401` (mehrere Imports), `F401` (ungenutzter Import), `E225` (fehlende Leerzeichen), `N802` (falsche Benennung)

Diskutieren Sie:

1. Wird der Code durch das Beheben schneller ausgeführt?
2. Wer profitiert von diesen Regeln?
3. Was passiert in einem Team, wenn alle sie ignorieren?

Problem 1: Import-Verstöße

✗ Problem:

```
import sys,os # Multiple + unused
```

Ruff meldet:

- **E401** : Mehrere Imports in einer Zeile
- **F401** : 'sys' importiert, aber nicht verwendet
- **F401** : 'os' importiert, aber nicht verwendet

✓ Lösung:

```
# Removed - not needed
```

Warum es wichtig ist:

- Ungenutzte Imports verschwenden Speicher
- Verwirren zukünftige Leser
- Sammeln sich als technische Schulden an

Problem 2: Fehlende Leerzeichen

✗ Schwer lesbar:

```
marker=dict(size=12,color='red')
y=0.015*x_norm**3
camera_x,camera_y = 0,2.0
```

✓ Leicht lesbar:

```
marker = dict(size=12, color='red')
y = 0.015 * x_norm**3
camera_x, camera_y = 0, 2.0
```

Studien zeigen: Entwickler erfassen gut formatierten Code ~30% schneller.

Ihr Gehirn gruppiert Informationen an Leerzeichen-Grenzen.

Problem 3: Namenskonventionen

Python-Benennung signalisiert Bedeutung:

Stil	Bedeutung	Beispiel
PascalCase	Klasse	RoadProfile , CameraRay
snake_case	Funktion/Variable	calculate_total , camera_x
UPPER_CASE	Konstante	MAX_SPEED , PI

✗ **Problem:** `HelperFunction()` → Sieht aus wie eine Klasse!

✓ **Lösung:** `helper_function()` → Eindeutig eine Funktion

Problem 4: Toter Code

```
def HelperFunction(val): # Never called anywhere!  
    result = val * 2  
    return result
```

Toter Code kostet:

-  Muss bei API-/Python-Änderungen aktualisiert werden
-  Leser müssen ungenutzten Code verstehen
-  Kann veraltete Logik enthalten, wenn er später reaktiviert wird

Lösung: Löschen Sie ihn! Git erinnert sich an alles.

Problem 5: Zeilenlänge

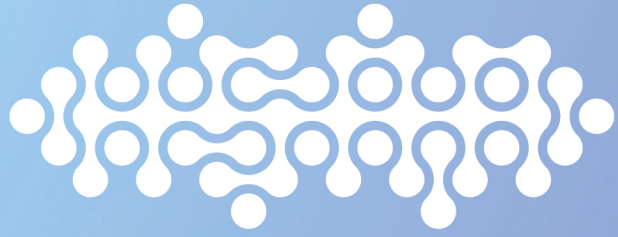
✗ Schwer zu überprüfen in der geteilten Ansicht:

```
marker=dict(size=12,color='red',symbol='circle',shape='diamond',opacity=0.8)
```

✓ Leicht Änderungen in Diffs zu erkennen:

```
marker = dict(  
    size=12,  
    color='red',  
    symbol='circle',  
    shape='diamond',  
)
```

Regel: Zeilen auf 88 Zeichen begrenzen (Ruff/Black-Standard)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Warum Code-Qualität wichtig ist

3

Das große Ganze

Vier Kernprinzipien:

1.  **Code wird 10x häufiger gelesen als geschrieben**
2.  **Konsistenz reduziert die kognitive Belastung**
3.  **Früherkennung verhindert teure Fehler**
4.  **Automatisierung skaliert, Menschen nicht**

Schauen wir uns jedes einzelne an...

Prinzip 1: Die 10:1-Regel

Für jede Stunde Code schreiben werden 10 Stunden mit Lesen verbracht:

Aktivität	Zeit
Sie schreiben ein Feature	1 Stunde
Sie debuggen es	30 Min.
Ein Kollege reviewt es	20 Min.
Ihr zukünftiges Ich fügt ein Feature hinzu	1 Stunde
Ein neues Teammitglied lernt die Codebasis	2 Stunden
Jemand behebt einen Fehler	1 Stunde

Gesamt: 5,5 Stunden lesen, 1 Stunde schreiben

→ Code-Qualität optimiert für das **Lesen**, nicht das Schreiben!

Prinzip 3: Die Kosten von Fehlern

Wann gefunden	Relative Kosten	Beispiel
Beim Schreiben	1x	„Ups, Tippfehler!" (5 Sekunden)
Code Review	10x	„Ich pushe einen Fix..." (5 Min.)
QA-Test	100x	„Debuggen, fixen, erneut testen" (1 Stunde)
Produktion	1000x+	„Notfall-Hotfix, Reputationsschaden"

Automatisierte Prüfungen fangen Probleme bei 1x Kosten ab, nicht bei 1000x!

Prinzip 4: Automatisierung skaliert

Die Rechnung (10 Entwickler, 50 PRs/Woche):

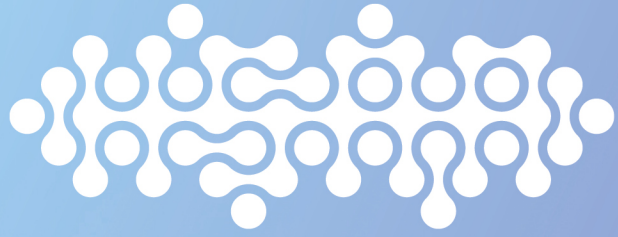
Ohne Automatisierung:

- 30 Min. pro Review
- 25 Stunden/Woche
- Zusätzliche Einstellung nötig!

Mit Automatisierung:

- 2 Min. automatisiert + 10 Min. manuell
- 10,3 Stunden/Woche
- **Spart ~87.000 \$/Jahr**

Außerdem: Zufriedenere Entwickler, schnellere Lieferung, weniger Fehler!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

PEP 8: Das Fundament

4

Pythons offizielle Stilrichtlinie

PEP 8 ist Pythons offizielle Stilrichtlinie.

- **Geschrieben von:** Guido van Rossum (Pythons Erfinder)
- **Erstveröffentlichung:** 2001
- **Status:** Aktiv, community-vereinbarter Standard

Warum ihr folgen?

-  Sie ist eine gemeinsame Sprache, die alle Python-Entwickler verstehen
-  Alle Werkzeuge erwarten sie (IDEs, Linter, Formatter)
-  Arbeitgeber erwarten sie (Bewerbungen, Portfolios, Open-Source-Beiträge)

Wichtige PEP 8-Regeln

Einrückung:

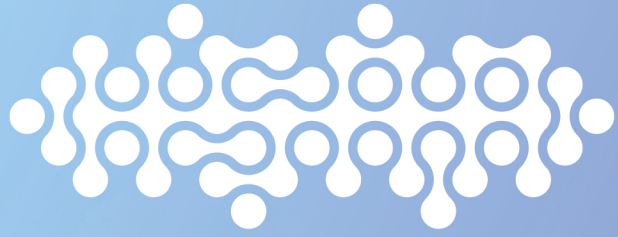
```
def example():  
    if True:  
        print("4 spaces")
```

Benennung:

```
def calculate_average(): # snake  
class RoadProfile:      # Pascal  
MAX_SPEED = 100         # UPPER
```

Imports (gruppiert, sortiert):

```
import os          # stdlib  
import sys  
  
import numpy       # third-party  
from dash import Dash  
  
from myapp import utils # local
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Moderne Werkzeuge: Die Ruff-Revolution

5

Von vier Werkzeugen zu zweien

Traditionell (vor 2023):

- Flake8 (Linting) + Black (Formatierung) + isort (Imports) + mypy (Typen)
- Vier Werkzeuge, vier Konfigurationen, langsam

Modern (2024+):

- **Ruff** (Linting + Formatierung + Imports)
- **Pyright** (Typprüfung)
- Zwei Werkzeuge, eine Konfiguration, blitzschnell! 🚀

Ruff ist 10-100x schneller als Python-basierte Werkzeuge (in Rust geschrieben)

Ruff installieren - Abhängigkeitsverwaltung

Wohin gehören Entwicklungswerkzeuge?

Typ	Befehl	Beispiel
Produktions-Abhängigkeiten	<code>uv add <pkg></code>	dash, numpy, plotly
Entwicklungs-Abhängigkeiten	<code>uv add --dev <pkg></code>	ruff, pytest, pyright
Globale Werkzeuge	<code>uv tool install <pkg></code>	Überall verfügbar

Ruff ist eine Entwicklungs-Abhängigkeit - zum Entwickeln nötig, nicht zum Ausführen der App!

```
uv add --dev ruff
```

pyproject.toml nach Hinzufügen der Entwicklungs-Abhängigkeiten

```
[project]
dependencies = ["dash", "plotly", "numpy"] # Production

[dependency-groups]
dev = ["ruff", "pyright"] # Development tools
```

Alles installieren: `uv sync --group dev`

Wichtige Erkenntnis: Produktions-Abhängigkeiten führen die App aus, Entwicklungs-Abhängigkeiten helfen bei der Entwicklung.

Ruff Check - Der Linter

Befehle:

```
uv run ruff check .          # Check all files
uv run ruff check --fix .    # Auto-fix issues
```

Beispielausgabe:

```
main.py:16:1: E401 [*] Multiple imports on one line
main.py:29:5: E225 [*] Missing whitespace around operator
Found 5 errors. [*] 3 fixable with --fix
```

Das `[*]` zeigt automatisch behebbare Probleme an.

Ruff Format - Der Formatter

Formatiert Code automatisch nach einheitlichem Stil:

```
uv run ruff format .          # Format all files
uv run ruff format --check .   # Check only (CI mode)
uv run ruff format --diff .    # Show what would change
```

Vorher:

```
def generate(n=100,x=80):
    y=0.015*x_norm**3
    return x,y
```

Nachher:

```
def generate(n=100, x=80):
    y = 0.015 * x_norm**3
    return x, y
```

Ruff konfigurieren

Zu `pyproject.toml` hinzufügen:

```
[tool.ruff]
line-length = 88
target-version = "py312"

[tool.ruff.lint]
select = [
    "E", "W",    # pycodestyle (errors, warnings)
    "F",         # Pyflakes (bugs)
    "I",         # isort (imports)
    "N",         # pep8-naming
    "B",         # flake8-bugbear
    "UP",        # pyupgrade (modern syntax)
]
fixable = ["ALL"]
```

Ruff in VS Code

Installation:

1. `Ctrl+Shift+X` (Erweiterungen)
2. Nach „Ruff“ suchen
3. Installieren (von Astral)

Funktionen:

- ● Rote Linien = Fehler
- ● Gelbe Linien = Warnungen
- `Ctrl+.` → Schnellfix
- `Shift+Alt+F` → Formatieren

Beim Speichern formatieren - Qualität ohne Aufwand

Zu `.vscode/settings.json` hinzufügen:

```
{
  "[python]": {
    "editor.defaultFormatter": "charliermarsh.ruff",
    "editor.formatOnSave": true,
    "editor.codeActionsOnSave": {
      "source.fixAll.ruff": "explicit",
      "source.organizeImports.ruff": "explicit"
    }
  }
}
```

Ergebnis: Drücken Sie `Ctrl+S` → Code wird sofort formatiert! 🎉

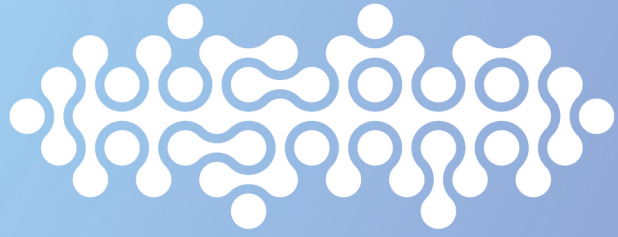
Hands-On: Reparieren wir Code!

Öffnen Sie Ihren Road Profile Viewer in VS Code:

1. Führen Sie `uv run ruff check .` aus - Wie viele Fehler?
2. Führen Sie `uv run ruff check --fix .` aus - Wie viele automatisch behoben?
3. Führen Sie `uv run ruff format .` aus - Sehen Sie, wie sich die Formatierung korrigiert
4. Prüfen Sie verbleibende Probleme - diese erfordern manuelle Korrekturen

Verbleibende Probleme können sein:

- Funktionsbenennung (ändern Sie `HelperFunction` → `helper_function`)
- Toter Code (löschen Sie ungenutzte Funktionen)
- Fehlende Type Hints (das behandeln wir als Nächstes!)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Typprüfung mit Pyright

6

Warum Typprüfung?

Python ist dynamisch typisiert - flexibel, aber gefährlich:

```
def calculate_discount(price, discount_percent):  
    return price * (discount_percent / 100)  
  
result = calculate_discount("50", "10") # 💥 Crash at runtime!
```

Type Hints fangen dies VOR der Laufzeit ab:

```
def calculate_discount(price: float, discount_percent: float) -> float:  
    return price * (discount_percent / 100)  
  
result = calculate_discount("50", "10")  
# ❌ Pyright: Expected float, got str
```

Pyright installieren

```
# Add as dev dependency
uv add --dev pyright

# Sync dependencies
uv sync --group dev

# Test it works
uv run pyright --version
```

! Windows-Nutzer: Möglicherweise muss der Entwicklermodus für Symlinks aktiviert werden.

```
# Run PowerShell as Admin:
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\AppModelUnlock" /t REG_DWORD /f /v "AllowDevelopmentWithoutDevLicense" /d "1"
```

Pyright konfigurieren - Strikter Modus

Zu `pyproject.toml` hinzufügen:

```
[tool.pyright]
typeCheckingMode = "strict"
pythonVersion = "3.12"
include = ["src"]
exclude = [".venv", "__pycache__", "build", "dist"]

reportMissingTypeStubs = true
reportUnknownMemberType = "warning"
reportUnknownParameterType = "warning"
```

Modi: `off` → `basic` (Standard) → `standard` → `strict`

Zum Lernen: Verwenden Sie **strict**, um alles zu erkennen!

Type Hints verwenden

Grundtypen:

```
name: str = "Alice"  
age: int = 25  
height: float = 1.75  
is_student: bool = True
```

Funktionen:

```
def greet(name: str) -> str:  
    return f"Hello, {name}"
```

Optional (None-Behandlung):

```
def find(id: int) -> User | None:  
    ...  
  
user = find(123)  
# user.name ✗ Might be None!  
if user is not None:  
    print(user.name) # ✓
```

Type Hints für den Road Profile Viewer

```
import numpy as np
from numpy.typing import NDArray

def generate_road_profile(
    num_points: int = 100,
    x_max: float = 80
) -> tuple[NDArray[np.float64], NDArray[np.float64]]:
    """Generate a road profile using clothoid approximation."""
    x = np.linspace(0, x_max, num_points)
    x_norm = x / x_max
    y = 0.015 * x_norm**3 * x_max + 0.3 * np.sin(2 * np.pi * x_norm)
    y = y - y[0]
    return x, y
```



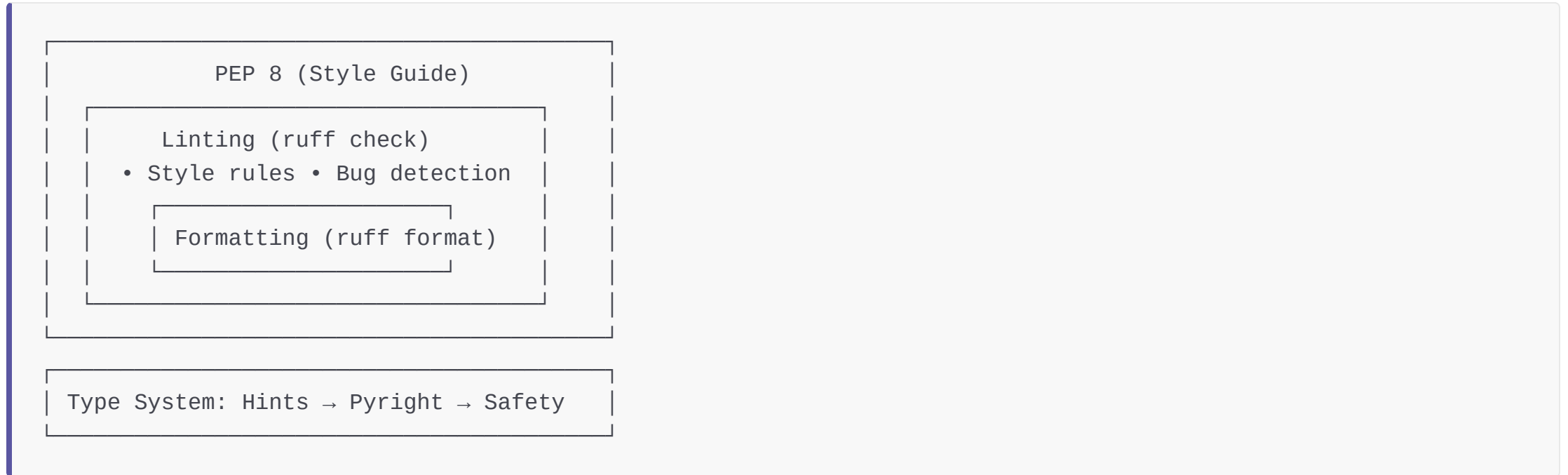
Kurze Umfrage: Qualitätskonzepte verstehen

Ordnen Sie das Konzept dem Werkzeug zu:

Konzept	Werkzeug
1. Stil prüfen + Fehler finden	A. <code>pyright</code>
2. Formatierung automatisch korrigieren	B. <code>ruff check</code>
3. Typannotationen validieren	C. <code>ruff format</code>

Antworten: 1-B, 2-C, 3-A

Die Qualitätslandschaft verstehen



Kernaussage: Ruff = Stil + Fehler, Pyright = Typen (ergänzend!)

Ruff vs Pyright: Sich ergänzende Werkzeuge

Ruff behandelt:

-  Stilverstöße (PEP 8)
-  Ungenutzte Imports/Variablen
-  Häufige Fehlermuster
-  Code-Formatierung
-  Import-Sortierung

Pyright behandelt:

-  Typ-Inkompatibilitäten
-  None-Behandlungsfehler
-  Funktionssignatur-Validierung
-  Fehlende Typannotationen
-  Typinferenz

Sie brauchen BEIDE! Ruff erkennt nicht `greet(42)`, wenn die Signatur `greet(name: str)` ist.

Ihr Qualitäts-Workflow

Schritt für Schritt:

```
# 1. Write code (with type hints)
def calculate_total(prices: list[float]) -> float:
    x=sum(prices)  # Oops, missing spaces
    return x

# 2. Format automatically
$ uv run ruff format .      # Fixes spacing

# 3. Check for issues
$ uv run ruff check .       # Style + bugs ✓

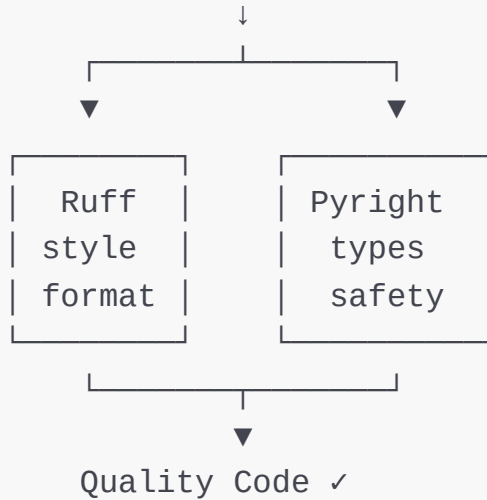
# 4. Validate types
$ uv run pyright            # Type safety ✓
```

Häufige Stolperfallen

Stolperfalle	Lösung
„Ich füge Qualitätsprüfungen später hinzu“	Von Tag eins an starten!
„Formatter machen meinen Code hässlich“	Konsistenz > Vorlieben
„Type Hints sind zu viel Aufwand“	Sie sparen 10x Debugging-Zeit
„Mein Code funktioniert, wozu Stil?“	Code wird 10x häufiger gelesen als geschrieben

Der vollständige Qualitäts-Stack

Developer writes code (with type hints)



Ergebnis: PEP 8-konformer, typsicherer, wartbarer Code!

Zusammenfassung: Die wichtigsten Erkenntnisse

1. Funktionierender Code $\neq$ Qualitätscode

- Professionelle Software muss lesbar, wartbar und typsicher sein

2. Die richtigen Werkzeuge verwenden

- **Ruff**: Linting + Formatierung in einem schnellen Werkzeug
- **Pyright**: Typprüfung integriert in VS Code

3. **PEP 8 befolgen** - Es ist der Python-Community-Standard

4. **Type Hints sparen Zeit** - Fehler vor der Laufzeit erkennen

5. **Alles automatisieren** - Qualität von Tag eins, nicht nachträglich angehängt

Wie geht es weiter?

Sie wissen jetzt:

-  Was geprüft werden muss (PEP 8, Typen, Fehler)
-  Wie Sie lokal prüfen (Ruff, Pyright)

Nächste Vorlesung (Kapitel 02 (Feature-Entwicklung)):

-  **CI/CD** - Prüfungen automatisch bei jedem Push ausführen
-  **GitHub Actions** - Automatisierungsplattform
-  **Branch Protection** - Qualität vor dem Merge erzwingen
-  **Pre-commit Hooks** - Lokale Qualitätsschranken

Lokale Prüfungen → Automatisierte Pipeline!

Übungszeit: Reparieren Sie Ihren Code!

Schritte:

1. `uv add --dev ruff pyright` → Entwicklungs-Abhängigkeiten hinzufügen
2. `uv run ruff check --fix .` → Linting-Probleme beheben
3. `uv run ruff format .` → Code formatieren
4. Type Hints zu den Hauptfunktionen hinzufügen
5. `uv run pyright` → Typfehler beheben

Ziel: Alle Prüfungen bestanden!

```
uv run ruff check . && uv run pyright # Zero errors 
```

Ressourcen

Offizielle Dokumentation:

- PEP 8: python.org/dev/peps/pep-0008
- Ruff: docs.astral.sh/ruff
- Pyright: microsoft.github.io/pyright

VS Code-Erweiterungen:

- Ruff: Nach „Ruff“ von Astral suchen
- Pylance: Enthält Pyright (standardmäßig mit Python-Erweiterung)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK