



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Chapter 02

CODE QUALITY IN PRACTICE

When Working Code Isn't Enough

The Scenario:

```
$ 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

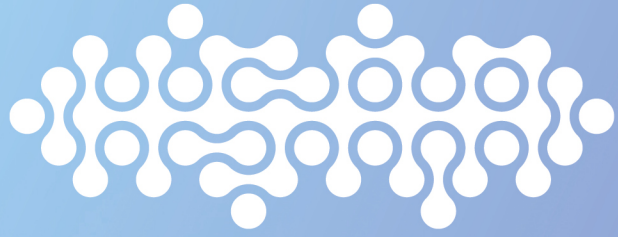
Pull request blocked - fix issues before merging

Your code *works*... but it doesn't meet professional standards.

Learning Objectives

By the end of this lecture, you will:

1.  **Understand** the difference between working code and professional code
2.  **Recognize** common code quality issues through real examples
3.  **Learn** Python's style guide (PEP 8) and why it matters
4.  **Master** modern tools: Ruff (linter/formatter) and Pyright (type checker)
5.  **Apply** automated checks to catch issues before human review
6.  **Build** quality into your workflow from day one



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Code Quality Journey

1

Stage 1: Can You Read This?

Take 30 seconds. What does this function do?

```
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
```

How long did it take you to figure out what **f**, **d**, **t**, **s**, **c**, **a**, **r** mean?

Stage 2: Names Help... Or Do They?

Same function, now with "real" names:

```
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
```

Six Conventions in One Function

Count the naming styles:

Variable	Convention
Calculate_Result	PascalCase + underscore
ScoreList	PascalCase
passingThreshold	camelCase
Total_sum	Mixed
COUNT	SCREAMING_CASE
currentScore	camelCase

Your brain re-learns the pattern for every variable.

Inconsistency costs *cognitive friction* on every line.

Stage 3: Consistent, But Is It Clean?

Consistent `snake_case` now, but...

```
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` -- **unused imports**
- No **docstring** -- what does this function do?

Stage 4: Professional Quality

```
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
```

Same logic -- but readable in seconds, not minutes.

Stage 5: Two Rights Don't Make a Standard

Version A -- Compact:

```
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 -- Explicit:

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

Why Style Guides Exist

Both versions are correct, readable, and pass every linter.

But imagine half the team writes Version A, the other half Version B.

Every file reads differently. Every function feels unfamiliar.

Style guides give your codebase a consistent identity -- like brand guidelines for visual design.

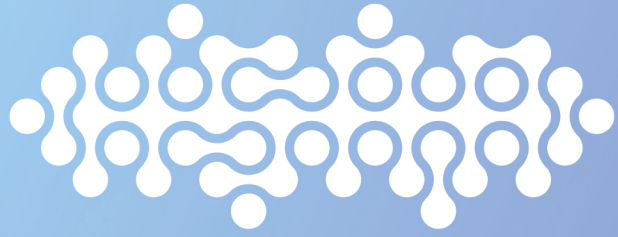
What Changed	Why It Matters
Meaningful names	Understand without tracing
Consistent conventions	Brain stops fighting format
No dead code/imports	Nothing distracts
Docstrings + type hints	Function explains itself
A shared style guide	Reads like one author

Now Imagine This at Scale

Not one 15-line function, but **2,000 lines across 15 files**.

- Stage 1 → Stage 4: the difference between a codebase you can maintain and one that makes you want to quit
- Without a style guide: a codebase that feels like a patchwork

Let's see what happens when we run automated quality tools on a real project...



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Reality Check

2

Our Code Under Scrutiny

Let's look at "working" code from our Road Profile Viewer:

```
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
```

Looks reasonable, right? Let's see what the tools say...

Running Automated Checks

```
$ 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 errors in code that works perfectly!



Discussion: Why Do These Rules Exist?

Take 2 minutes with your neighbor:

Look at these errors: `E401` (multiple imports), `F401` (unused import), `E225` (missing whitespace), `N802` (wrong naming)

Discuss:

1. Does fixing these make the code run faster?
2. Who benefits from these rules?
3. What happens in a team if everyone ignores them?

Issue 1: Import Violations

✗ Problem:

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

Ruff reports:

- E401 : Multiple imports on one line
- F401 : 'sys' imported but unused
- F401 : 'os' imported but unused

✓ Solution:

```
# Removed - not needed
```

Why it matters:

- Unused imports waste memory
- Confuse future readers
- Accumulate as technical debt

Issue 2: Missing Whitespace

✗ Hard to read:

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

✓ Easy to read:

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

Research shows: Developers parse well-spaced code ~30% faster.

Your brain chunks information at whitespace boundaries.

Issue 3: Naming Conventions

Python naming signals meaning:

Style	Means	Example
PascalCase	Class	RoadProfile , CameraRay
snake_case	Function/variable	calculate_total , camera_x
UPPER_CASE	Constant	MAX_SPEED , PI

✗ **Problem:** `HelperFunction()` → Looks like a class!

✓ **Solution:** `helper_function()` → Clearly a function

Issue 4: Dead Code

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

Dead code costs:

-  Must update for API/Python changes
-  Readers must understand unused code
-  May use outdated logic if revived later

Solution: Delete it! Git remembers everything.

Issue 5: Line Length

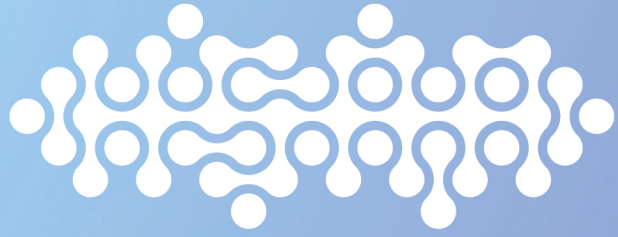
✗ Hard to review in split view:

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

✓ Easy to spot changes in diffs:

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

Rule: Limit lines to 88 characters (Ruff/Black default)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why Code Quality Matters

3

The Bigger Picture

Four core principles:

1. 📖 **Code is read 10x more than written**
2. 🧠 **Consistency reduces cognitive load**
3. 💰 **Early detection prevents expensive bugs**
4. 🚀 **Automation scales, humans don't**

Let's explore each...

Principle 1: The 10:1 Rule

For every 1 hour writing code, 10 hours are spent reading it:

Activity	Time
You write a feature	1 hour
You debug it	30 min
Colleague reviews it	20 min
Future you adds a feature	1 hour
New team member learns codebase	2 hours
Someone fixes a bug	1 hour

Total: 5.5 hours reading, 1 hour writing

→ Code quality optimizes for **reading**, not writing!

Principle 3: The Cost of Bugs

When Found	Relative Cost	Example
While writing	1x	"Oops, typo!" (5 seconds)
Code review	10x	"Let me push a fix..." (5 min)
QA testing	100x	"Debug, fix, retest" (1 hour)
Production	1000x+	"Emergency hotfix, reputation damage"

Automated checks catch issues at 1x cost, not 1000x!

Principle 4: Automation Scales

The Math (10 developers, 50 PRs/week):

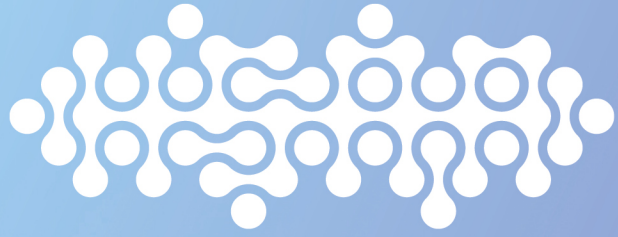
Without automation:

- 30 min per review
- 25 hours/week
- Need extra hire!

With automation:

- 2 min automated + 10 min human
- 10.3 hours/week
- **Saves ~\$87,000/year**

Plus: Happier developers, faster delivery, fewer bugs!



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

PEP 8: The Foundation

4

Python's Official Style Guide

PEP 8 is Python's official style guide.

- **Written by:** Guido van Rossum (Python's creator)
- **First published:** 2001
- **Status:** Active, community-agreed standard

Why follow it?

-  It's a common language all Python devs understand
-  All tools expect it (IDEs, linters, formatters)
-  Hiring expects it (interviews, portfolios, OSS contributions)

Key PEP 8 Rules

Indentation:

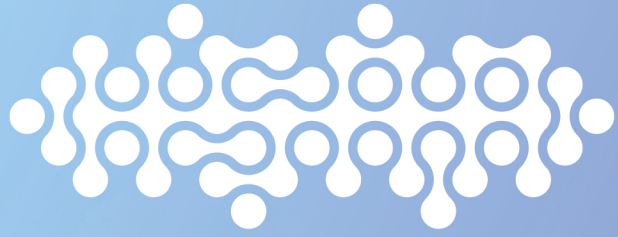
```
def example():  
    if True:  
        print("4 spaces")
```

Naming:

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

Imports (grouped, ordered):

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



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Modern Tools: The Ruff Revolution

5

From Four Tools to Two

Traditional (Pre-2023):

- Flake8 (linting) + Black (formatting) + isort (imports) + mypy (types)
- Four tools, four configs, slow

Modern (2024+):

- **Ruff** (linting + formatting + imports)
- **Pyright** (type checking)
- Two tools, one config, blazingly fast! 🚀

Ruff is 10-100x faster than Python-based tools (written in Rust)

Installing Ruff - Dependency Management

Where do development tools go?

Type	Command	Example
Production deps	<code>uv add <pkg></code>	dash, numpy, plotly
Dev deps	<code>uv add --dev <pkg></code>	ruff, pytest, pyright
Global tools	<code>uv tool install <pkg></code>	Available everywhere

Ruff is a DEV dependency - needed for development, not to run the app!

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

pyproject.toml After Adding Dev Deps

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

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

Install everything: `uv sync --group dev`

Key insight: Production deps run the app, dev deps help you develop it.

Ruff Check - The Linter

Commands:

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

Example output:

```
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
```

The `[*]` indicates auto-fixable issues.

Ruff Format - The Formatter

Automatically reformats code to follow consistent style:

```
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
```

Before:

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

After:

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

Configuring Ruff

Add to `pyproject.toml`:

```
[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

Install:

1. `Ctrl+Shift+X` (Extensions)
2. Search "Ruff"
3. Install (by Astral)

Features:

- ● Red lines = errors
- ● Yellow lines = warnings
- `Ctrl+.` → Quick fix
- `Shift+Alt+F` → Format

Format on Save - Zero Effort Quality

Add to `.vscode/settings.json`:

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

Result: Press `Ctrl+S` → Code is instantly formatted! 🎉



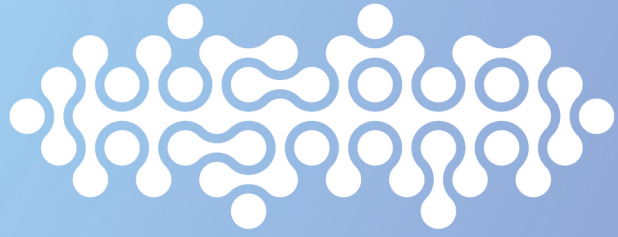
Hands-On: Let's Fix Some Code!

Open your Road Profile Viewer in VS Code:

1. Run `uv run ruff check .` - How many errors?
2. Run `uv run ruff check --fix .` - How many auto-fixed?
3. Run `uv run ruff format .` - Watch the spacing fix itself
4. Check remaining issues - these need manual fixes

Remaining issues might include:

- Function naming (change `HelperFunction` → `helper_function`)
- Dead code (delete unused functions)
- Missing type hints (we'll cover this next!)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Type Checking with Pylint

6

Why Type Checking?

Python is dynamically typed - flexible but dangerous:

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

Type hints catch this BEFORE runtime:

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

Installing Pyright

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

# Sync dependencies
uv sync --group dev

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

 **Windows users:** May need Developer Mode enabled for symlinks.

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

Configuring Pyright - Strict Mode

Add to `pyproject.toml`:

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

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

Modes: `off` → `basic` (default) → `standard` → `strict`

For learning: Use **strict** to catch everything!

Using Type Hints

Basic Types:

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

Functions:

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

Optional (None handling):

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

Type Hints for 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
```



Quick Poll: Understanding Quality Concepts

Match the concept to its tool:

Concept	Tool
1. Checking style + finding bugs	A. <code>pyright</code>
2. Auto-fixing spacing/indentation	B. <code>ruff check</code>
3. Validating type annotations	C. <code>ruff format</code>

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

Understanding the Quality Landscape



Key: Ruff = style + bugs, Pyright = types (complementary!)

Ruff vs Pyright: Complementary Tools

Ruff handles:

- ✓ Style violations (PEP 8)
- ✓ Unused imports/variables
- ✓ Common bug patterns
- ✓ Code formatting
- ✓ Import sorting

Pyright handles:

- ✓ Type mismatches
- ✓ None handling errors
- ✓ Function signature validation
- ✓ Missing type annotations
- ✓ Type inference

You need BOTH! Ruff won't catch `greet(42)` when signature is `greet(name: str)`.

Your Quality Workflow

Step-by-step:

```
# 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

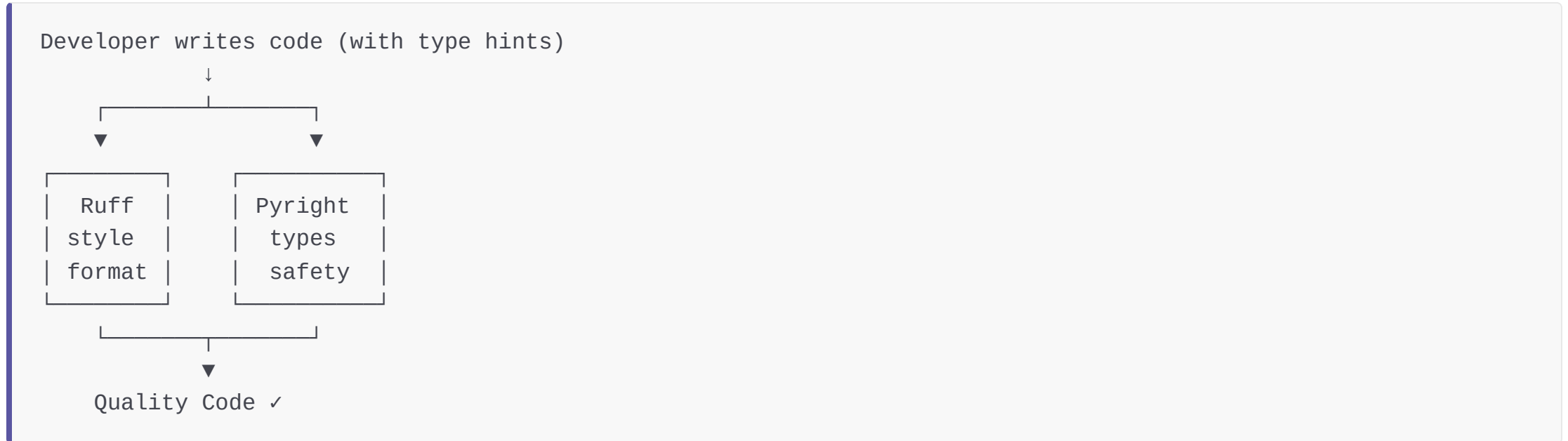
# 3. Check for issues
$ uv run ruff check .       # Style + bugs ✓

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

Common Pitfalls

Pitfall	Solution
"I'll add quality checks later"	Start from day one!
"Formatters make my code ugly"	Consistency > preference
"Type hints are too much work"	They save 10x debugging time
"My code works, why style?"	Code is read 10x more than written

The Complete Quality Stack



Result: PEP 8 compliant, type-safe, maintainable code!

Summary: Key Takeaways

1. **Working code $\neq$ Quality code**

- Professional software must be readable, maintainable, type-safe

2. **Use the right tools**

- **Ruff**: Linting + formatting in one fast tool
- **Pyright**: Type checking integrated with VS Code

3. **Follow PEP 8** - It's the Python community standard

4. **Type hints save time** - Catch bugs before runtime

5. **Automate everything** - Quality from day one, not bolted on later

What's Next?

You now know:

-  What to check (PEP 8, types, bugs)
-  How to check locally (Ruff, Pyright)

Next lecture (Chapter 02 (Feature Development)):

-  **CI/CD** - Running checks automatically on every push
-  **GitHub Actions** - Automation platform
-  **Branch Protection** - Enforcing quality before merge
-  **Pre-commit Hooks** - Local quality gates

Transform local checks → Automated pipeline!

Practice Time: Fix Your Code!

Steps:

1. `uv add --dev ruff pyright` → Add dev dependencies
2. `uv run ruff check --fix .` → Fix linting issues
3. `uv run ruff format .` → Format code
4. Add type hints to main functions
5. `uv run pyright` → Fix type errors

Goal: All checks pass!

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

Resources

Official Documentation:

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

VS Code Extensions:

- Ruff: Search "Ruff" by Astral
- Pylance: Includes Pyright (default with Python extension)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU