Programmierung von CAx-Systemen

David Straub

Software Engineering Basics

1. Versionsverwaltung mit Git
2. Unittests mit Pytest
3. **Type Hints und statische Codeanalyse**

Motivation: Warum Type Hints?

```python
def finde_material(name):
    for m in materialdatenbank:
        if m.name == name:
            return m
    # Material nicht gefunden → kein return → implizites None
```

Irgendwo anders im Projekt:

```python
dichte = finde_material("Stahl").dichte   # Tippfehler im Namen
                                           # → AttributeError: 'NoneType' has no attribute '
```

Kein Hinweis wo der eigentliche Fehler liegt. Der Absturz ist weit vom Tippfehler entfernt.

Warum findet Python das nicht?

Python prüft Typen **zur Laufzeit** – und nur wenn der Code tatsächlich ausgeführt wird.

```python
# Funktioniert für beliebige Typen – Python fragt nicht nach
len("hallo")    # → 5
len([1, 2, 3])  # → 3
```

Duck Typing: Python fragt nicht *was* ein Objekt ist, sondern ob es das Benötigte *kann*. Flexibel – aber `None` kann fast nichts, und der Fehler taucht erst später auf.

Die Lösung: Typ explizit machen

```python
def finde_material(name: str) → dict | None:
    for m in materialdatenbank:
        if m.name == name:
            return m
    return None
```

Jetzt warnt Pylance **sofort beim Schreiben**:

```python
dichte = finde_material("Stahl").dichte   # ⍰ Object is possibly None
```

Ziel: Fehler so früh wie möglich sichtbar machen – idealerweise bevor der Code läuft.

Typisierung – Grundlagen

Statisch vs. dynamisch

	Dynamisch	Statisch
Wann geprüft	zur Laufzeit	vor der Ausführung
Fehler sichtbar	wenn der Code läuft	sofort im Editor / Compiler
Beispiele	Python, JavaScript	C, Java, Rust
Flexibilität	hoch	geringer

Python: dynamisch – aber annotierbar

Seit **Python 3.5** (PEP 484, 2015) gibt es eine offizielle Syntax für Type Hints.

Type Hints ändern das Laufzeitverhalten **nicht**:

```
def finde_material(name: str) → dict | None:
    ...
```

`name: str` ist ein Hinweis – kein Zwang. Python führt den Code genauso aus wie ohne Annotation.
Nutznießer sind: Editor, statische Analyse-Tools, KI-Assistenten, andere Entwickler.

Vorteile und Nachteile

Vorteile

Vorteil	Konkret
Selbstdokumentierend	`def schweissen(a: Solid, b: Solid) -> Solid` braucht keinen Kommentar
Fehler früher sichtbar	Editor zeigt Konflikt sofort – nicht erst nach 30 s Build
Refactoring sicherer	Parameteränderung: Editor findet alle betroffenen Stellen
KI-Tools besser	Copilot, Cursor etc. machen weniger Fehler bei getyptem Code

Kosten

- Etwas mehr Tippaufwand
- Komplexe Typen (`Union`, verschachtelte Container) können unübersichtlich werden
- Kein Laufzeitschutz – das ist die Aufgabe von Pytest

Faustegel: **Öffentliche Funktionen immer annotieren**, interne Hilfsfunktionen nach Ermessen.

Type Hints: Syntax und Regeln

Type Hints haben keinen Laufzeiteffekt

Auch falsche Annotationen verhindern weder die Ausführung noch erzeugen sie eine Warnung:

```python
def verdopple(x: str) -> int:   # beides falsch annotiert
    return x * 2

verdopple(5)   # läuft ohne Fehler oder Warnung durch
```

Python ignoriert Annotationen vollständig – sie existieren nur für externe Tools.

Externe Tools prüfen ohne Ausführung

Tool	Wann	Wie
Pylance	beim Tippen	Unterwellenlinie direkt im Editor
mypy / pyright	auf der Kommandozeile	`mypy mein_projekt/`
CI-Pipeline	bei jedem Commit	Fehler erscheinen am Merge Request

Ergebnis: Typfehler wie das vergessene `None`-Handling fallen auf, **bevor der Code jemals läuft**.

Einfache Parameter und Rückgabewert

```python
import math

def berechne_querschnitt(radius: float, wandstaerke: float) -> float:
    innen = radius - wandstaerke
    return math.pi * (radius**2 - innen**2)

def bauteil_name(nummer: int) -> str:
    return f"Bauteil-{nummer:04d}"

def ist_gueltig(volumen: float) -> bool:
```

```
    return volumen > 0
```

Rückgabewert None

```
def exportiere_step(koerper: Solid, pfad: str) -> None:
    export_step(koerper, pfad)
```

`-> None` bedeutet: die Funktion gibt keinen Wert zurück.
Explizit hinschreiben – auch wenn Python es nicht erzwingt.

Optional: Wert oder nichts

```
from build123d import Solid

def finde_material(name: str) -> dict | None:
    for m in materialdatenbank:
        if m.name == name:
            return m
    return None
```

`dict | None` signalisiert: **der Aufrufer muss mit `None` rechnen**.

Pylance warnt sofort wenn `None` ohne Prüfung verwendet wird:

```
dichte = finde_material("Stahl").dichte   # ? Pylance: Object is possibly None
```

Container-Typen

```
def mittelpunkt(punkte: list[float]) -> float:
    return sum(punkte) / len(punkte)

def werkstoff_lookup(tabelle: dict[str, float], name: str) -> float:
    return tabelle[name]

def bounding_ecken(bb) -> tuple[float, float, float]:
    return bb.min.X, bb.min.Y, bb.min.Z
```

Union: mehrere mögliche Typen

```python
# Funktion akzeptiert int oder float
def skaliere_mm_zu_m(wert: float | int) -> float:
    return float(wert) / 1000
```

Selten nötig – wenn möglich einen einheitlichen Typ wählen.

Variablenannotationen

```python
radius: float = 9.0
name: str = "Flansch"
ergebnisse: list[float] = []
```

Selten explizit nötig – Python und Pylance inferieren einfache Fälle automatisch.
Sinnvoll bei leeren Containern, die später befüllt werden.

KI-Tools und Lesbarkeit

Was die KI „sieht"

Ohne Type Hints:

```
def bearbeite(teil, werkzeug, tiefe):
    ...
```

→ KI muss raten: Was ist `teil`? Eine Zahl? Ein Objekt? Was gibt die Funktion zurück?

Mit Type Hints:

```
def bearbeite(teil: Solid, werkzeug: Solid, tiefe: float) → Solid:
    ...
```

→ KI kennt den Kontext – bessere Vervollständigungen, weniger Halluzinationen.

Typen als Vertrag

- Typen **ersetzen lange Docstrings** für den häufigsten Fall
- In Teams: klare Schnittstelle zwischen Modulen
- In 6 Monaten: versteht man auch eigenen Code ohne Erinnerung
- **Für KI-Agenten** (Copilot, Cursor, Claude): je mehr Kontext, desto zuverlässiger die Ausgabe

Statische Codeanalyse

Was ist statische Codeanalyse?

Python selbst prüft Typen **nicht** – Type Hints sind für den Interpreter unsichtbar.

Statische Analyse übernehmen **externe Tools**, die den Code lesen ohne ihn auszuführen:

Tool	Einsatz
mypy	Kommandozeile, etablierter Standard
pyright	Schneller, von Microsoft entwickelt
Pylance	VS Code-Erweiterung – nutzt pyright intern

Pylance in VS Code

Pylance ist die VS Code-Erweiterung für Python-Analyse – sie läuft **im Hintergrund während man tippt**:

- Unterwellenlinie bei Typkonflikt, sofort beim Schreiben
- Hover zeigt inferierte Typen und Signaturen
- Autovervollständigung kennt die Methoden des richtigen Typs

```python
def berechne_querschnitt(radius: float, wandstaerke: float) -> float:
    ...

berechne_querschnitt(wandstaerke=50, radius=5)   # kein Fehler – gleicher Typ
berechne_querschnitt("50", 5)                    # ▯ Argument of type "str" cannot
                                                  #   be assigned to "float"
```

Kein separates Tool starten – Pylance ist direkt in den Editor integriert.

mypy und pyright in CI

mypy und pyright laufen auch auf der Kommandozeile – damit lassen sie sich in CI einbinden:

```
pip install mypy
mypy mein_projekt/
```

```yaml
# .gitlab-ci.yml
lint:
  script:
    - pip install mypy build123d
    - mypy mein_projekt/
```

Jeder Commit wird geprüft. Typfehler erscheinen als am Merge Request – genauso wie fehlgeschlagene Pytest-Tests.

Type Hints in build123d

Wichtige Typen

Typ	Bedeutung
`Solid`	geschlossener 3D-Körper
`Shell`	offene Hülle aus Flächen
`Face`	einzelne Fläche
`Edge`	einzelne Kante
`Wire`	geschlossene Kantenfolge
`Shape`	Basisklasse – wenn der genaue Typ egal ist
`ShapeList`	build123d-eigene Liste – **nicht** `list[Shape]`

Typische Signaturen

```python
from build123d import Solid, Face, Edge, Wire, ShapeList
from typing import Sequence

def rohr(radius: float, laenge: float) -> Solid:
    ...

def verrunde_kanten(koerper: Solid, radius: float) -> Solid:
    ...

def sammle_aussenflaechen(koerper: Solid) -> ShapeList:
    return koerper.faces().filter_by(GeomType.PLANE)
```

Häufige Tücken (1)

`Shape` vs. `Solid` vs. `Compound`

```python
# Boolean kann lautlos ein leeres Compound zurückgeben
ergebnis = teil_a - teil_b   # Typ laut Stubs: Shape
assert isinstance(ergebnis, Solid)   # narrowt den Typ und prüft zur Laufzeit
```

`ShapeList` ist nicht `list`

```
kanten: ShapeList = koerper.edges()   # ShapeList, nicht list[Edge]
kanten.sort_by(Axis.Z)                # ShapeList-Methode – auf list nicht verfügbar
```

Häufige Tücken (2)

Winkel: Grad oder Bogenmass?

```
# Unklar:
def drehe(koerper: Solid, winkel: float) -> Solid: ...

# Klar durch Namen:
def drehe(koerper: Solid, winkel_grad: float) -> Solid: ...
```

Typen können keine Einheiten ausdrücken – aussagekräftige Parameternamen sind der Ersatz.

Empfehlungen für build123d-Projekte

```
def bearbeite(teil: Solid, werkzeug: Solid) -> Solid:
    return teil - werkzeug

def spiegele(koerper: Solid, ebene: Plane = Plane.XZ) -> Solid:
    return mirror(koerper, about=ebene)
```

- Immer den **spezifischsten Typ** angeben: `Solid` statt `Shape` – Boolean-Operationen geben laut Stubs `Shape` zurück, explizit casten: `return Solid(teil - werkzeug)` (wirft Exception wenn Ergebnis leer oder kein Einzelkörper)
- Rückgabetyp immer angeben – auch `-> None`
- Parameter mit physikalischer Einheit im Namen: `laenge_mm`, `winkel_grad`

Typfehler umgehen – und warum man es nicht sollte

`cast` – dem Tool etwas einreden

Wenn Bibliotheks-Stubs ungenau sind (z. B. Boolean gibt `Shape` zurück statt `Solid`), liegt `cast` nahe:

```
from typing import cast

ergebnis = cast(Solid, teil - werkzeug)   # kein Laufzeiteffekt!
```

`cast` sagt dem Typchecker: „Vertrau mir, das ist ein `Solid`." – Python selbst prüft **nichts**.

Wenn die Annahme falsch ist, gibt es zur Laufzeit einen Fehler weit vom eigentlichen Problem entfernt – genau das, was Type Hints verhindern sollen.

`# type: ignore` – der Notausgang

```
dichte = finde_material("Stahl").dichte   # type: ignore[union-attr]
```

Der Fehler verschwindet – aber nur aus dem Tool. Im laufenden Programm ist er noch da.

Wann legitim: Bibliotheks-Stubs sind nachweislich falsch und es gibt keine bessere Alternative.

Wann Alarmsignal: KI-generierter Code führt `# type: ignore` ein um einen Typkonflikt zu verstecken, den man selbst nicht versteht → **nachfragen, nicht akzeptieren**.

`assert isinstance` – die saubere Alternative

```
def bearbeite(teil: Solid, werkzeug: Solid) -> Solid:
    ergebnis = teil - werkzeug          # Typ laut Stubs: Shape
    assert isinstance(ergebnis, Solid)  # prüft zur Laufzeit UND narrowt den Typ
    return ergebnis                     # Typchecker weiß jetzt: Solid ✓
```

Im Gegensatz zu `cast`: **beweist** statt zu behaupten. Scheitert sofort an der richtigen Stelle wenn das Ergebnis kein `Solid` ist.

Faustregel: `assert isinstance` bevorzugen. `cast` und `# type: ignore` nur wenn `isinstance` nicht praktikabel ist – und nie um Fehler im eigenen Code zu verstecken.

`assert` als Laufzeitprüfung

Rolle: was statische Analyse nicht kann

Type Hints und Pylance/mypy prüfen **vor** der Ausführung. Manche Bedingungen kennt man erst zur Laufzeit:

```python
def bearbeite(teil: Solid, werkzeug: Solid) -> Solid:
    ergebnis = Solid(teil - werkzeug)
    assert ergebnis.volume > 0, "Boolean-Operation hat leeren Körper erzeugt"
    return ergebnis
```

`assert` ergänzt statische Analyse – es prüft inhaltliche Korrektheit, nicht Typen.

Grenzen von `assert`

1. Deaktivierbar: Python überspringt alle `assert`-Anweisungen bei `python -O` (Optimierungsmodus). Produktionscode darf sich nicht auf `assert` verlassen.

```
python -O mein_skript.py   # assert wird komplett ignoriert
```

2. Kein Ersatz für Eingabevalidierung: Nutzereingaben oder Funktionsargumente mit `assert` zu prüfen ist falsch – dafür `if`/`raise` verwenden:

```python
assert radius > 0  # Falsch

if radius <= 0:
    raise ValueError(f"radius muss positiv sein, war {radius}")  # Richtig
```

3. Für interne Invarianten gedacht: `assert` dokumentiert Annahmen über den eigenen Code – nicht über externe Eingaben.

Aufgabe: Rohr annotieren

Teil 1 – Signaturen ergänzen

Gegeben:

```
import math
from build123d import *

def rohr(r_aussen, wandstaerke, laenge):
    r_innen = r_aussen - wandstaerke
    querschnitt = Circle(r_aussen) - Circle(r_innen)
    return extrude(querschnitt, laenge)
```

Ergänzen Sie vollständige Type Hints für beide Funktionen.

Teil 2 – Optionaler Parameter

Erweitern Sie `rohr` um einen optionalen Fasenparameter:

```
def rohr(r_aussen: float, wandstaerke: float, laenge: float,
         fase: float | None = None) -> Solid:
    ...
    if fase is not None:
        # Fase an den Stirnflächen
        ...
    return ergebnis
```

Was ändert sich am Rückgabetyp? Was nicht?

Teil 3 – Pylance ausprobieren

1. Entfernen Sie die Type Hints aus `rohr`
2. Rufen Sie `rohr("50", 5, 100)` auf
3. Fügen Sie die Type Hints wieder hinzu – was zeigt Pylance jetzt?
4. Führen Sie `mypy` auf der Datei aus und vergleichen Sie die Ausgabe