stackfuel

# Refactoring und Code-Qualität

Python-Code sauberer gestalten

**Refactoring** (auch **Refaktorisierung**, **Refaktorierung** oder **Restrukturierung**) bezeichnet in der Software-Entwicklung die manuelle oder automatisierte Struktur-verbesserung von Quelltexten unter Beibehaltung des beobachtbaren Programmverhaltens.

Dabei sollen Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den jeweiligen Aufwand für Fehleranalyse und funktionale Erweiterungen deutlich zu senken.

*Quelle: https://de.wikipedia.org/wiki/Refactoring*

stackfuel

# „Code is read much more often than it is written.“

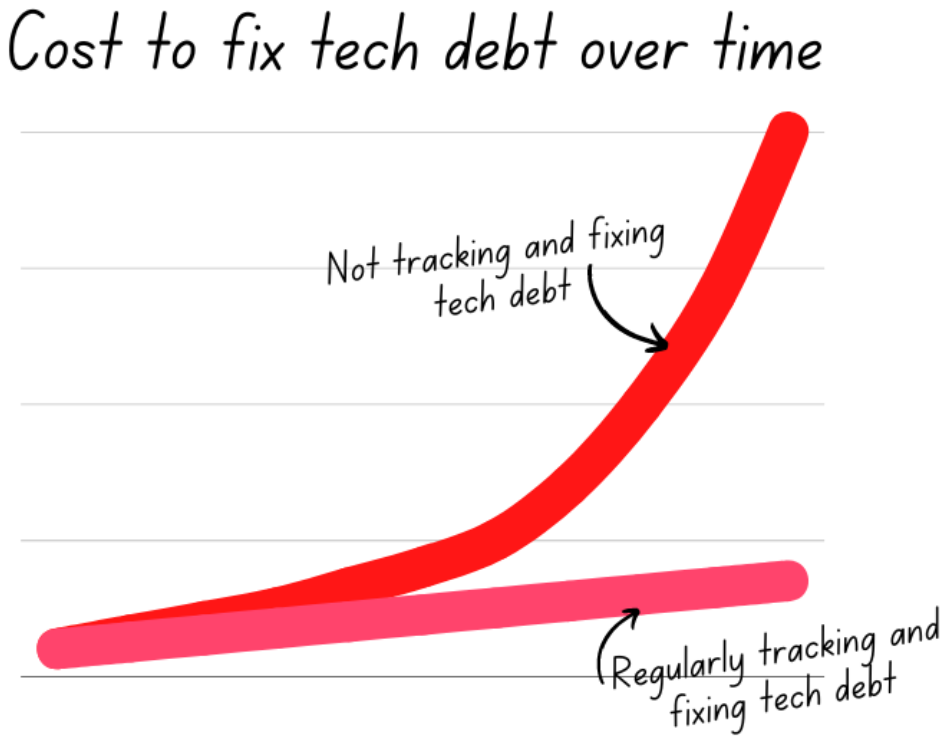
Guido van Rossum, Autor von Python

# Was bringt sauberer Code?

- **Fehler lassen sich schneller finden und beheben**
- **Bessere Zusammenarbeit im Team:**
  Einheitliche und lesbare Struktur erspart allen Zeit, besonders wenn mehrere an einem Projekt arbeiten.
- **Einfachere Wartung und Erweiterung:**
  Kann später leichter neue Funktionen einbauen, ohne gleich alles neu machen zu müssen.
- **Du sparst deinem zukünftigen Ich viel Arbeit:**
  Spätestens beim nächsten Update ist jeder dankbar, wenn der Code nach ein paar Wochen noch verständlich ist.

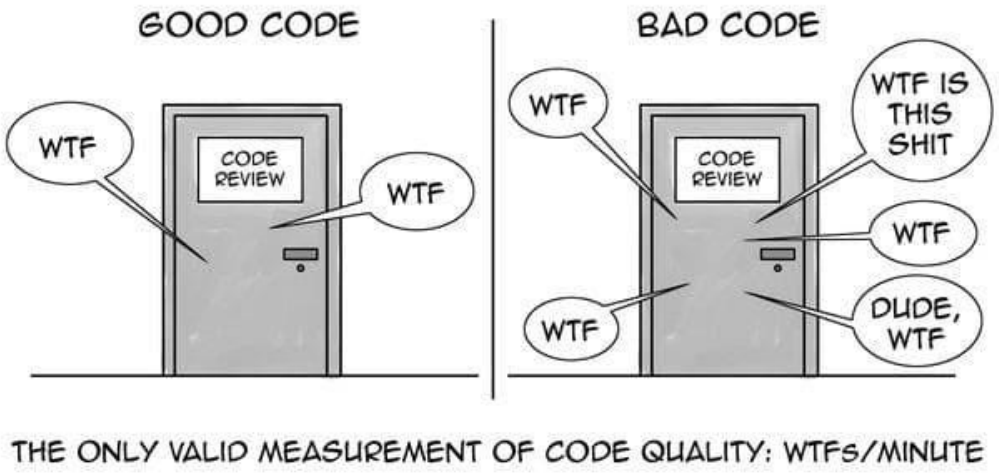
# Refactoring

- Refactoring bedeutet, **den Aufbau von Code zu verbessern**, ohne das ursprüngliche Verhalten zu verändern.
- **Warum ist Refactoring sinnvoll?**
  - Klarheit schaffen: Der Code wird verständlicher, du findest dich schneller zurecht.
  - Wartungsaufwand verringern: Fehler lassen sich früher erkennen; Erweiterungen werden leichter.
  - Nachhaltigkeit: Reduziert „technische Schulden“, die sich sonst im Projekt ansammeln.
- Wichtig: *Funktion bleibt gleich. Struktur wird besser.*




Quelle: stepsize.com

# Wann ist Refactoring sinnvoll?




Quelle: medium.com

- **Code Smells –** typische Strukturen im Code, die ein Refactoring nahe legen:
  - Wiederholter Code (*„Code Duplication“*)
  - Lange, komplexe Funktionen (*„God Function“*)
  - Unklare oder kryptische Namen
  - Funktionen oder Klassen mit zu vielen Aufgaben
  - Inkonsistente Formatierung oder Stil
- Keine objektiven Kriterien: Jedes Projekt hat seine eigenen Anforderungen

stackfuel

# Tipp 1: Duplizierung entfernen

## Vorher

```python
def calculate_area_rectangle(length, width):
    return length * width

def calculate_area_square(side):
    return side * side  # Duplicate logic for calculating area
```

## Nachher

```python
def calculate_area(length, width=None):
    if width is None:
        width = length  # Handles square case
    return length * width

print(calculate_area(5))      # Square
print(calculate_area(5, 10))  # Rectangle
```

stackfuel

# Tipp 2: Große Funktionen aufteilen

## Vorher

```python
def process_order(order):
    # Validate order
    if not order.get("items"):
        return "Order must contain items"

    # Calculate total price
    total = sum(item["price"] * item["quantity"] for item in order["items"])

    # Apply discount
    if order.get("discount_code") == "SAVE10":
        total *= 0.9
    # Generate receipt
    receipt = f"Total: ${total:.2f}"
    return receipt
```

## Nachher

```python
def validate_order(order):
    if not order.get("items"):
        return False, "Order must contain items"
    return True, ""

def calculate_total(order):
    return sum(item["price"] * item["quantity"] for item in order["items"])

def apply_discount(total, discount_code):
    return total * 0.9 if discount_code == "SAVE10" else total

def generate_receipt(total):
    return f"Total: ${total:.2f}"

def process_order(order):
    valid, message = validate_order(order)
    if not valid:
        return message

    total = calculate_total(order)
    total = apply_discount(total, order.get("discount_code"))

    return generate_receipt(total)
```

stackfuel

# Tipp 3: (List-)Comprehensions

## Vorher

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

## Nachher

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = [num for num in numbers if num % 2 == 0]
print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

stackfuel

# Tipp 4: Komplexe Bedingungen vereinfachen

## Vorher

```python
def check_access(user):
    if (user["role"] == "admin" or user["role"] == "manager") and
user["is_active"] and not user["is_suspended"]:
        return "Access Granted"
    else:
        return "Access Denied"

user = {"role": "manager", "is_active": True, "is_suspended": False}
print(check_access(user))  # Output: Access Granted
```

## Nachher

```python
def check_access(user):
    has_permission = user["role"] in ["admin", "manager"]
    is_active = user["is_active"]
    is_not_suspended = not user["is_suspended"]

    if has_permission and is_active and is_not_suspended:
        return "Access Granted"
    return "Access Denied"

user = {"role": "manager", "is_active": True, "is_suspended": False}
print(check_access(user))  # Output: Access Granted
```

stackfuel

# Tipp 5: Hardcoding vermeiden

## Vorher

```python
def calculate_discount(price):
    discount_rate = 0.10  # Hard-coded discount value
    discounted_price = price - (price * discount_rate)
    return discounted_price

print(calculate_discount(100))
```

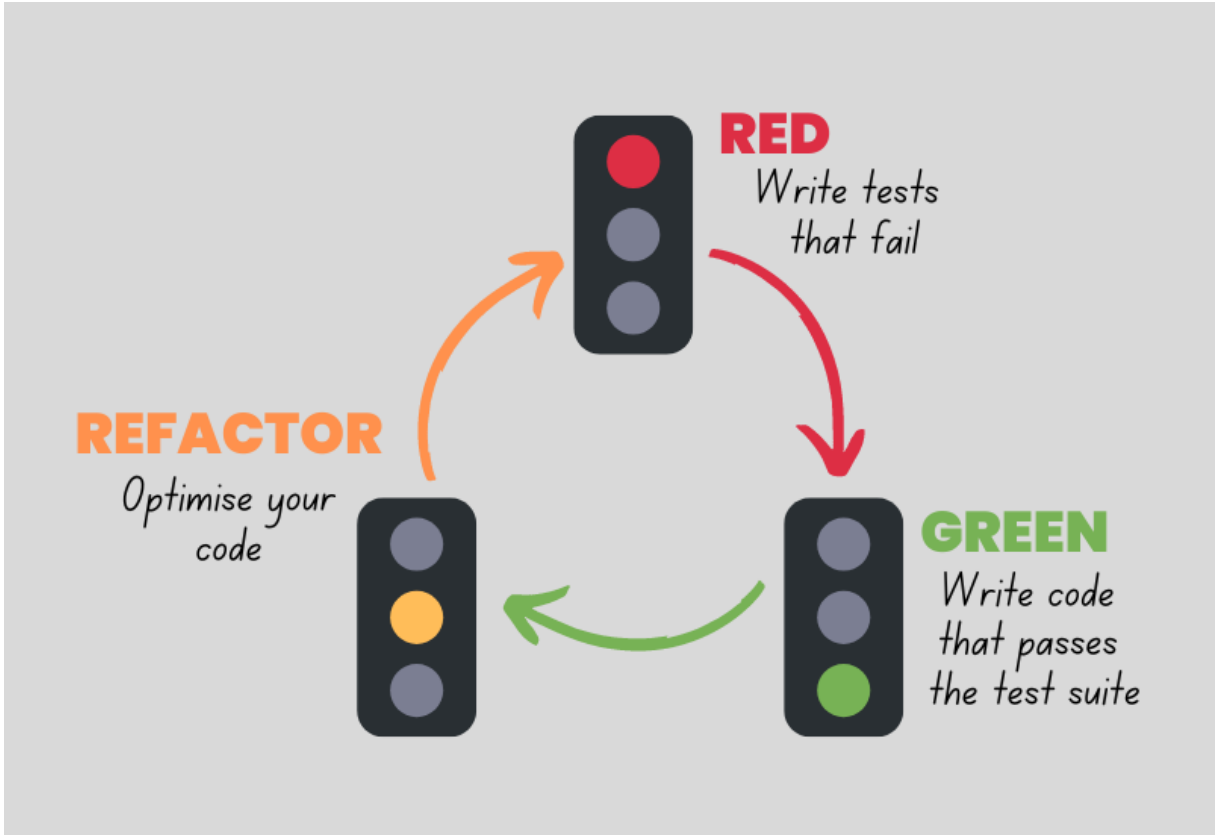
## Nachher

```python
DISCOUNT_RATE = 0.10  # Defined as a named constant

def calculate_discount(price):
    discounted_price = price - (price * DISCOUNT_RATE)
    return discounted_price

print(calculate_discount(100))
```

# Refactoring im Business-Kontext




- Beim Entwickeln neuer Features ist kontinuierliches Refactoring wichtig
- **Red-Green-Refactor:**
  Best Practice aus dem Testgetriebenen Entwickeln (TDD, *„test driven development“*), auch fester Bestandteil agiler Methoden:
  - **Red:** Schreibe einen neuen Test, der fehlschlägt (gewünschtes Verhalten absichern).
  - **Green:** Schreibe nur so viel Code, dass der Test besteht.
  - **Refactor:** Optimiere jetzt den Code, ohne Funktionalität zu verändern – Tests müssen weiterhin grün bleiben.

# PEP 8 – Python Style Guide

- PEP = *Python Enhancement Proposals*
- PEP 8 ist der **offizielle Styleguide** für Python
  - Vorgaben zur Formatierung und Namensgebung
  - Macht Code vertraut und leicht zu lesen – unabhängig vom Projekt
- Wozu das Ganze?
  - Teamarbeit: Konsistenz verringert Stolperfallen
  - Bessere Code-Review, leichterer Einstieg für Außenstehende

> *Readability counts* – The Zen of Python

# Zentrale PEP8-Richtlinien

- **Einrückung:** 4 Leerzeichen pro Ebene, keine Tabs verwenden
- **Zeilenlänge:** Maximal 79 Zeichen
  - (nutzt Zeilenumbrüche, insbesondere für Auflistungen und Funktionsparameter!)
  - für Kommentare oder Tests auch länger möglich
- **Leerzeilen:** Trenne Klassen und Funktionen zur besseren Übersicht
- **Importe:** Gesammelt zu Beginn des Notebooks/Skripts
  - Erst Standardbibliothek, dann externe Pakete, dann lokale Importe
- **Namenskonventionen:**
  - `snake_case` für Variablen und Funktionen
  - `CamelCase` für Klassen
  - `UPPER_CASE` für Konstanten
- **Abstände:** Leerzeichen um Operatoren und nach Kommas
  - **Ausnahme: Kein** Leerzeichen für Standardwerte: `def foo(x=42)`

Link zum vollständigen Style-Guide

stackfuel

# Beispiel: Anwendung von PEP8

## Vorher

```
def Calcprice(qTY,price,Discount=0):
 result=0
 if(qTY>0):
  result=qTY*price-Discount
  print ("Preis beträgt: ",result)
 else:
  print("FEHLER!")
 return result
```

## Nachher

```
def calc_price(quantity, price, discount=0):
    result = 0
    if quantity > 0:
        result = quantity * price - discount
        print("Preis beträgt:", result)
    else:
        print("Fehler!")
    return result
```

# Der Sudoku-Löser als Praxisbeispiel

- Im nächsten Teil der Live-Session arbeiten wir praktisch:
  - Erkennen von „**Code Smells**“ und Stilbrüchen im Beispiel eines Sudoku-Lösers
  - Schrittweises **Refactoring** für bessere Lesbarkeit und klare Verantwortlichkeiten
  - Anwendung von **PEP8** für professionelle, einheitliche Struktur
  - **Ziel:** Aus „Hauptsache es läuft“-Code machen wir „Darauf bin ich stolz“-Code!