Software engineering for Data Scientists

Chapter 1 - What is good code?

Finn Armistead & Jannik Fischer, Ingolstadt, 27.03.2025

Wofür brauche ich überhaupt “guten” Code?

Directors Version

- Cleaner Code -> Guter Code
- Guter Code ist wie eine Investition
- Fundament jedes komplexen und langjährigem Softwareprojektes
- Tech debt

Technical debt (often abbreviated as tech debt) is a commonly used term for deferred work resulting from when code is written quickly instead of correctly. Tech debt can take the form of missing documentation, poorly structured code, poorly named variables, or any other cut corners. These make the code harder to maintain or refactor, and it’s likely that you will spend more time in the future fixing bugs than you would have spent writing the code well in the first place. That said, tech debt is often necessary because of business deadlines and budgets. You don’t always have time to polish your code.

Die 5 Säulen von “gutem” Code

- Simplicity
- Modularity
- Readability
- Performance
- Robustness

Simplicity

Simplicity

Verhindern von zu komplexen Codes

- Komplexe Codes sind schwieriger an die sich ständig änderten Bedingungen anzupassen
- Komplexität entsteht nicht bewusst sondern durch das andauernde hinzufügen von neuen Aufgaben, Modellen oder auch kleinen Fehlern
- Bedeutet: Es entstehen zwangsweise immer komplexe Stellen im Code welche man später bewusst simplifizieren muss
    - Man kann Komplexität nicht verhindern sondern nur möglichst stark eindämmen

Simplicity

Don’t repeat yourself

- Man sollte versuchen möglichst wenig ähnliche Codepassagen in einem Programm zu haben
  - Unübersichtlich, große Angriffsfläche für kleine Fehler, schwieriger zu verstehen und zu verändern
- Arbeiten mit allgemeinen Funktionen die man auf das Problem mit gewissen Inputs anpassen kann
- Rückführen von gesamten Code auf nur wenige Ursprungsinformationen (verhindern von zu viel “Hardcode”)

Simplicity

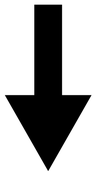
Beispiel

```python
import pandas as pd

df = pd.read_csv("sdg_literacy_rate.csv")
df = df.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df = df.set_index("Country Name").transpose()

df2 = pd.read_csv("sdg_electricity_data.csv")
df2 = df2.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df2 = df2.set_index("Country Name").transpose()

df3 = pd.read_csv("sdg_urban_population.csv")
df3 = df3.drop(["Series Name", "Series Code", "Country Code"], axis=1)
df3 = df3.set_index("Country Name").transpose()
```



```python
def process_sdg_data(csv_file, columns_to_drop):
    df = pd.read_csv(csv_file)
    df = df.drop(columns_to_drop, axis=1)
    df = df.set_index("Country Name").transpose()
    return df
```

Modularity

Modularity

Teilsysteme

- Komplexe Projekte -> Teilsysteme
- Fehlerminimierung
- Verbesserte Lesbarkeit und Verständnis

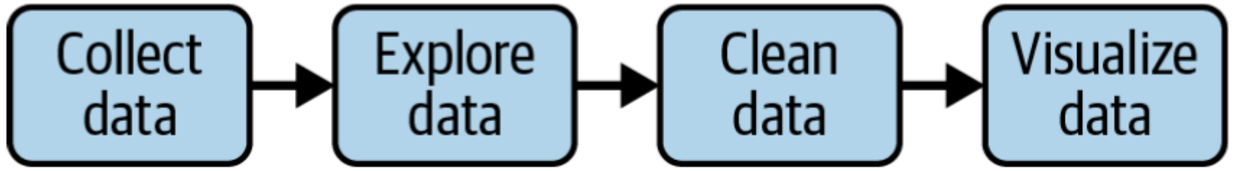
Modularity
Beispiel

- Aufteilen eines großen Data-Science-Projektes wie folgt:

```
def load_data(csv_file):
    pass

def clean_data(input_data, max_length):
    pass

def plot_data(clean_data, x_axis_limit, line_width):
    pass
```



Readability

…code is read much more often than it is written..

-PEP8

```
import numpy as n
def game(x,y):
        if x=='a' and y=='b':return"x wins"
        if x=='a' and y=='c':return"y wins"
        if x=='b' and y=='a':return"y wins"
        if x=='b' and y=='c':return"x wins"
        if x=='c' and y=='a':return"x wins"
        if x=='c' and y=='b':return"y wins"
        if x == y:return"Draw"
x1=input("x:")
y1=input("y:")
print(game(x1,y1))
```

Readability

Verwenden von verständlichen Namen

- Namen sollten kurz aber dennoch verständlich sein
- Wählt man zu lange Namen wird der Code unübersichtlich
- Wählt man zu kurze Namen kann man später vergessen haben was man meinte

Readability

Dokumentieren des Codes

- One-liners
- Docstrings
- readme-pages
- Wichtig: updaten des Codes -> Dokumentation Update

Readability

Halten an Konventionen

- Richtig: i = 2 + 5
- Falsch: i=2+5

- PEP8

Readability

Bereinigen des Codes

```
def SchereSteinPapier(spieler1,spieler2):

        if spieler1 == spieler2: return "Unentschieden"

        if spieler1=='Schere' and spieler2=='Papier':
                return"Spieler 1 gewinnt"
        if spieler1=='Papier' and spieler2=='Schere':
                return"Spieler 2 gewinnt"

        if spieler1 == 'Stein' and spieler2 == 'Schere':
                return "Spieler 1 gewinnt"
        if spieler1=='Schere' and spieler2=='Stein':
                return"Spieler 2 gewinnt"

        if spieler1=='Papier' and spieler2=='Stein':
                return"Spieler 1 gewinnt"
        if spieler1=='Stein' and spieler2=='Papier':
                return"Spieler 2 gewinnt"

Spieler1 = input("Spieler 1 Eingabe: ")
Spieler2 = input("Spieler 2 Eingabe: ")

print(game(Spieler1,Spieler2))
```

Performance

Performance

- Datensparende und effiziente Algorithmen
- Man sollte verstehen welcher Teil des Codes eine lange Zeit benötigt
  - MODULARITY!

Robustness

Robustness

Reaktion des Codes auf Fehler

- Falsche Inputs?
  - Möchte ich eine Errormeldung erzeugen?
  - Möchte ich, dass der Code mit den gegebenen Daten bestmöglich weiterarbeitet?

Robustness

Testing & logging

- Es sollten alle möglichen Inputs abgedeckt sein, dass bekannt ist wie der Code reagiert
- Nur weil der Code auf unserem Device gut läuft heißt das nicht dass er auf anderen Devices läuft
- Viele Möglichkeiten des Testens (mehr in Kapitel 7)