



Hochschule für Technik  
und Wirtschaft Berlin

University of Applied Sciences

---

# Eignung von Large Language Models (LLMs) zur Generierung von Python Code zur Datenanalyse

---

Bachelorarbeit

Name des Studiengangs  
Wirtschaftsinformatik

**Fachbereich 4**

vorgelegt von  
Maurice Krüger

Datum:  
Berlin, 12.02.2025

Erstgutachter: Prof. Dr.-Ing. Ingo Claßen

Zweitgutachter: Prof. Dr. Axel Hochstein

## Abstract

Diese Bachelorarbeit untersucht die Eignung moderner Large Language Models (LLMs) für die automatisierte Generierung von Python-Code im Kontext typischer Datenanalyseaufgaben. Dabei wird zunächst ein Überblick über die theoretischen Grundlagen von LLMs und der Programmiersprache Python gegeben. Anschließend werden in einer empirischen Untersuchung Codebeispiele durch den Marktführer ChatGPT mit dem Sprachmodell GPTo1 erzeugt und mit manuell geschriebenen Skripten verglichen. Die Bewertung erfolgt anhand mehrerer Kriterien wie Korrektheit, Performanz sowie Verständlichkeit des generierten Codes. Abschließend werden Empfehlungen für den praktischen Einsatz von LLMs in der Datenanalyse abgeleitet sowie ein Ausblick auf zukünftige Entwicklungen gegeben.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>4</b>
1.1	Problemstellung und Forschungsfragen . . . . .	4
1.2	Relevanz der Thematik . . . . .	5
1.3	Zielsetzung . . . . .	5
1.4	Aufbau der Arbeit . . . . .	5
<b>2</b>	<b>Grundlagen</b>	<b>5</b>
2.1	Einführung Large Language Models . . . . .	6
2.2	Einführung Python . . . . .	7
2.2.1	Bedeutung und Bibliotheken . . . . .	7
2.2.2	Typische Schritte einer Datenanalyse . . . . .	7
2.3	Einführung automatisierte Code-Generierung . . . . .	7
2.3.1	Funktionsweise und Vorteile . . . . .	7
2.3.2	Herausforderungen und Grenzen . . . . .	8
2.4	Prompting mit Sprachmodellen . . . . .	8
<b>3</b>	<b>LLMs in der Programmierung – aktueller Stand</b>	<b>9</b>
3.1	Überblick und Vergleich von verschiedenen LLMs . . . . .	9
3.2	Einsatzgebiete von LLMs in der Programmierung . . . . .	11
3.3	Vergangene Studien und Arbeiten zur Code-Generierung . . . . .	11
<b>4</b>	<b>Ausgangsdaten und Testfallspezifikation</b>	<b>14</b>
4.1	Überblick und Reduktion der Datengrundlage . . . . .	14
4.2	Umwandlung in Pandas DataFrames . . . . .	14
4.3	Testfälle und Vorgehen . . . . .	15
4.3.1	Testfall 1: Sortierung und Ausgabe der Fallzahlen 2023 . . . . .	15
4.3.2	Testfall 2: Join aller Tabellen und „Bezirks-Topwert“ . . . . .	16
4.3.3	Testfall 3: Prozentuale Verteilung der Straftaten . . . . .	16
4.3.4	Testfall 4: Zeitreihe über die Jahre 2014–2023 . . . . .	17
<b>5</b>	<b>Methodik</b>	<b>17</b>
5.1	Vorgehensweise der Untersuchung . . . . .	17
5.2	Testfälle der Datenanalyse . . . . .	17
5.2.1	Testfall 1 . . . . .	17
5.2.2	Testfall 2 Verbund und Aggregation . . . . .	18
5.2.3	Testfall 3 . . . . .	19
5.2.4	Testfall 4 . . . . .	19
5.3	Auswertungskriterien . . . . .	20
5.4	Verwendete Tools und Daten . . . . .	21

<b>6</b>	<b>Auswertung der Python-Code-Generierung zur Datenanalyse durch LLMs</b>	<b>21</b>
6.1	Testfall 1: Sortierung und Ausgabe der Fallzahlen 2023 . . . . .	21
6.2	Testfall 2: Join aller Tabellen und Bezirks-Topwert . . . . .	23
6.3	Testfall 3: Prozentuale Verteilung der Straftaten . . . . .	25
6.4	Testfall 4: Zeitreihe über die Jahre 2014–2023 . . . . .	27
6.5	Übersicht der Ergebnisse . . . . .	28
6.6	Vergleich manuell erstellter Code . . . . .	28
<b>7</b>	<b>Fazit und Ausblick</b>	<b>30</b>
<b>8</b>	<b>Anhang</b>	<b>32</b>
8.1	Literaturverzeichnis . . . . .	32
8.2	Quellcodeverzeichnis . . . . .	33
8.2.1	Testfall 1 . . . . .	33
8.2.2	Testfall 2 . . . . .	50
8.2.3	Testfall 3 . . . . .	69
8.2.4	Testfall 4 . . . . .	87
8.2.5	Manuelle Skripte . . . . .	109

# 1 Einleitung

## 1.1 Problemstellung und Forschungsfragen

Die schnelle Entwicklung von Large Language Models (LLMs), wie zum Beispiel ChatGPT, hat in den letzten Jahren sowohl im privaten als auch im beruflichen Bereich viel Aufmerksamkeit erregt. Ursprünglich wurden LLMs hauptsächlich zur Lösung alltäglicher Probleme und der Verarbeitung und Erzeugung menschlicher Sprache eingesetzt, doch zunehmend zeigt sich, dass sie auch Programmiercode in verschiedenen Sprachen erstellen können. Besonders in der Programmiersprache Python – einer weit verbreiteten Sprache für Datenanalyse und Machine Learning – sind die Fortschritte in der automatisierten Code-Generierung durch LLMs bereits bemerkenswert[1, 2].

Aktuelle Forschungsarbeiten konzentrieren sich auf die systematische Bewertung von solch generierten Codes, um Fehlerquellen und Qualitätsmerkmale zu bemessen. Die Bereitstellung öffentlicher Evaluierungsdatensätze und -frameworks, wie etwa *HumanEval*[2] oder *Eval-Plus*[3], ermöglicht standardisierte Vergleichsstudien verschiedener LLMs. Dies eröffnet neue Anwendungsfelder im Bereich der Datenanalyse: Anstatt den Code manuell zu schreiben, könnten Nutzer in Zukunft lediglich ihre Anforderungen in natürlicher Sprache formulieren und das Modell würde diese für den Nutzer umsetzen.

Vor diesem Hintergrund stellt sich die Frage, ob und inwiefern LLMs tatsächlich qualitativ hochwertigen Python-Code für datenanalytische Aufgaben erzeugen können und wie dieser Code im Vergleich zu manuell geschriebenen Code abschneidet. Auch die möglichen Grenzen dieser automatisierten Generierung, wie etwa in Bezug auf Performanz, Wartbarkeit oder Fehlerraten, sind hierbei von großer Bedeutung.

Daraus ergibt sich die zentrale **Hauptforschungsfrage**:

*Inwieweit eignen sich Large Language Models (LLMs) zur Durchführung gängiger Datenanalyseaufgaben in Python, und wie schneidet dieser Code im Vergleich zu manuell geschriebenen Code hinsichtlich Effizienz, Korrektheit und Wartbarkeit ab?*

Zur weiteren Strukturierung dieser Hauptfrage werden mehrere Unterfragen hinzugezogen:

- **Qualität & Korrektheit:** Wie qualitativ hochwertig ist dieser generierte Code hinsichtlich Syntax und Implementierung von Analyseaufgaben (z. B. Datenfilterung, Aggregation)?
- **Effizienz & Performanz:** Inwieweit entspricht der automatisch erzeugte Code modernen Standards bezüglich Laufzeit und Ressourcenverbrauch?
- **Wartbarkeit & Verständlichkeit:** Wie gut lässt sich der generierte Code verstehen, dokumentieren und erweitern?
- **Einsatzgebiete & Grenzen:** Für welche spezifischen Aufgaben in der Datenanalyse ist der Einsatz von LLMs sinnvoll und wo stoßen diese an ihre Grenzen?

## 1.2 Relevanz der Thematik

Die Fähigkeit, Programmiercode mit Hilfe von LLMs zu erstellen, könnte die Entwicklungsprozesse erheblich beschleunigen und neue Nutzergruppen anziehen, die bisher wenig Erfahrung mit Programmierung hatten. Besonders in der Datenanalyse können viele Arbeitsschritte – vor allem wiederkehrende Aufgaben, wie das Erstellen von Standard-Pipelines zur Datenbeschaffung- und bereinigung – automatisiert werden. Gleichzeitig gibt es jedoch Herausforderungen in Bezug auf Performanz, Wartbarkeit und Transparenz.

## 1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, herauszufinden, wie gut moderne LLMs (Large Language Models) für die automatische Code-Generierung in der Datenanalyse mit Python geeignet sind. Dafür wird in einem Experiment Code von einem LLM generiert und mit manuell geschriebenem Code verglichen. Der Vergleich basiert auf Kriterien wie Korrektheit, Performance und Wartbarkeit. Auf Basis der Ergebnisse werden Empfehlungen für den Einsatz von LLMs in der Praxis gegeben und deren Grenzen diskutiert. Zum Schluss wird ein Ausblick darauf gegeben, wie sich diese Technologie in Zukunft weiterentwickeln könnte und welche Auswirkungen das auf Aufgaben in der Datenanalyse haben könnte.

## 1.4 Aufbau der Arbeit

Nach dieser Einleitung (Kapitel 1) folgt in Kapitel 2 eine Darstellung der **Grundlagen**, dazu gehört eine Einführung in Large Language Models, die Programmiersprache Python, die automatisierte Code-Generierung und Prompting. Kapitel 3 gibt einen Überblick über den aktuellen Stand der Forschung, in dem verschiedene LLM-Modelle, Publikationen und Evaluationstechniken vorgestellt werden. Darauf aufbauend werden in Kapitel 4 die **Ausgangsdaten** und eine genaue **Testfallspezifikation** und in Kapitel 5 die **Methodik** der Arbeit erläutert. Kapitel 6 enthält dann die **Auswertung** der gewonnenen Daten sowie den Vergleich von durch ein LLM generierten und manuell geschriebenen Code. Kapitel 7 fasst die Ergebnisse zusammen, beantwortet die Forschungsfragen und gibt einen **Ausblick** auf weitere Entwicklungen. Schließlich enthält Kapitel 8 den **Anhang**, einschließlich Literaturverzeichnis und relevanter Dokumentationen.

# 2 Grundlagen

Im folgenden Kapitel werden die theoretischen und technischen Grundlagen vorgestellt, die für das Verständnis dieser Arbeit notwendig sind. Abschnitt 2.1 beschäftigt sich mit den Large Language Models, ihrer Funktionsweise und ihrer Bedeutung in der Code-Generierung. Danach wird in Abschnitt 2.2 das Potenzial der Programmiersprache Python für die Datenanalyse erläutert, bevor Abschnitt 2.3 das Konzept der automatisierten Code-Generierung behandelt und Abschnitt 2.4 eine kurze Einführung in Prompting gibt.

## 2.1 Einführung Large Language Models

Bei Large Language Models (LLMs) handelt es sich um eine spezielle Klasse von vortrainierten Sprachmodellen (so genannten *Pre-trained Language Models* (PLMs)) mit einer besonders großen Anzahl an Parametern – typischerweise im Bereich von mehreren zehn bis hundert Milliarden. Sie entstanden als Weiterentwicklung früherer Sprachmodellierungsansätze, die sich über viele Jahre hinweg von *Statistical Language Models* (SLMs) hin zu *Neural Language Models* (NLMs) entwickelt haben. Ein wesentlicher Meilenstein in dieser Entwicklung war die Einführung von Transformer-basierten PLMs, die auf großen Textkorpora trainiert wurden und herausragende Leistungen bei verschiedenen Aufgaben der natürlichen Sprachverarbeitung (*Natural Language Processing* (NLP)) erzielten. In den Forschungen entdeckte man, dass eine Vergrößerung der Modellgröße nicht nur zu erheblichen Leistungssteigerungen führt, sondern dass LLMs ab einer bestimmten Größenordnung auch neue Fähigkeiten entwickeln – beispielsweise *In-Context Learning*, das es ihnen ermöglicht, Aufgaben ohne explizites Nachtrainieren zu lösen. Um diese leistungsfähigeren Modelle von kleineren abzugrenzen, hat sich der Begriff Large Language Models (LLMs) etabliert[4].

Die schon genannten Transformer-Architekturen, welche einen Self-Attention-Mechanismus nutzen, ermöglichen es dem Modell Beziehungen zwischen verschiedenen Wörtern oder Tokens in einer Eingabe zu erkennen, wobei es nicht von Bedeutung ist, an welcher Position diese stehen. Die Transformer-Architektur unterscheidet sich desweiteren von früheren Architekturen, wie zum Beispiel *Recurrent Neuronal Networks* (RNNs) dadurch, dass sie auf die rekursive Verarbeitung der Tokens verzichtet und stattdessen alle Tokens parallel verarbeitet, wodurch die LLMs deutlich effizienter auf Eingaben reagieren können. Diese Leistung der LLMs korreliert jedoch stark mit der Größe der Modelle und der Menge an Trainingsdaten, was dazu führt, dass besonders gute Modelle einen deutlich höheren Ressourcenbedarf und eine deutlich größere Menge an Trainingsdaten benötigen[5].

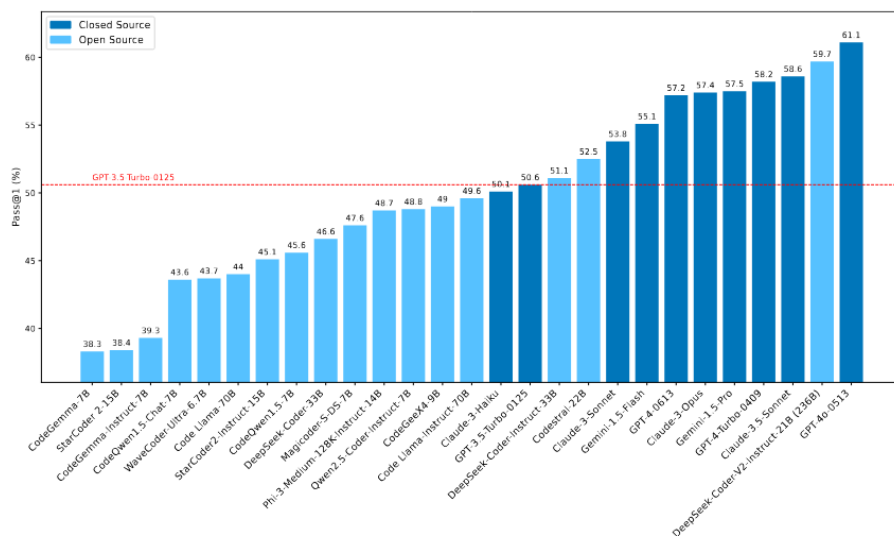


Abbildung 1: Leistungsvergleich verschiedener Modellgrößen (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation').

In den letzten Jahren wurden mehrere Benchmarks und Evaluierungsdatensätze speziell für die Code-Generierung entwickelt. Beispiele dafür sind *HumanEval*[2] und *EvalPlus*[3], die genutzt werden, um die Genauigkeit und Zuverlässigkeit von LLMs in verschiedenen Programmiersprachen zu testen. Erste Studien zeigen, dass LLMs einfache bis mittelschwere Aufgaben oft vollständig lösen können. Bei komplexeren oder sehr speziellen Aufgabenbereichen stoßen sie aber noch an ihre Grenzen[1]. Obwohl LLMs in vielen Sprachen Code generieren können, hat sich Python als einer der Hauptfoki herauskristallisiert. Dies liegt an der weit verbreiteten Nutzung von Python in Wissenschaft und Industrie, insbesondere in den Bereichen Datenanalyse und Machine Learning.

## 2.2 Einführung Python

### 2.2.1 Bedeutung und Bibliotheken

Python ist dank seiner Syntax, aktiven Community und vielen hilfreichen Bibliotheken eine der am weitesten verbreiteten Sprachen für Datenanalyse. Ein paar der wichtigsten Bibliotheken, die in der Datenanalyse verwendet werden, sind:

- **pandas** – Datenstrukturen und -bearbeitung,
- **NumPy** – numerische Berechnungen,
- **scikit-learn** – Machine-Learning-Algorithmen,
- **Matplotlib** – Visualisierung,

Diese und weitere Bibliotheken ermöglichen eine effiziente Umsetzung datenanalytischer Projekte und werden bereits in LLM-Trainings berücksichtigt, wodurch generierter Code auf bekannte Funktionen zurückgreifen kann[3, 2].

### 2.2.2 Typische Schritte einer Datenanalyse

Die Grundschrirte einer klassischen Datenanalyse in Python enthält folgende Schritte:

1. *Datenimport* (z. B. CSV-Dateien, Datenbanken, APIs),
2. *Datenbereinigung* (fehlende Werte, Duplikate, Datentypen),
3. *Analyse und Visualisierung* (Statistiken, Plots),

Im Rahmen dieser Arbeit wird untersucht, ob LLMs diese Schritte automatisieren können und an welchen Stellen manuell eingegriffen werden muss.

## 2.3 Einführung automatisierte Code-Generierung

### 2.3.1 Funktionsweise und Vorteile

Automatisierte Code-Generierung mithilfe von LLMs basiert auf *Prompts*, also Benutzeranfragen in natürlicher Sprache. LLMs haben hierbei die Möglichkeit sich flexibel an den vom



Benutzer gegebenen Kontext anzupassen und können die natürliche Sprache in funktionsfähigen Code umwandeln. Ebenso müssen LLMs nicht spezifisch auf eine Aufgabe trainiert werden, aufgrund der großen Trainingsdaten, die ihnen zur Verfügung stehen[2]. Insbesondere für datenanalytische Aufgaben, bei denen standardisierte Skripte (z. B. für das Einlesen und Bereinigen von Daten) immer wieder benötigt werden, kann dies zu einer erheblichen Zeiterparnis führen und ermöglicht die Nutzung von LLMs auch für weniger erfahrene Personen, die nicht über tiefgreifende Programmierkenntnisse verfügen.

### 2.3.2 Herausforderungen und Grenzen

Trotz beeindruckender Fortschritte stößt die automatisierte Code-Generierung noch häufig an Grenzen:

- **Komplexe Datenstrukturen:** LLMs zeigen teils Schwächen bei Aufgaben mit hochgradiger Komplexität oder spezifischem Wissen, wenn zu wenig Kontext durch den Nutzer gegeben wird[6].
- **Performanz:** Generierter Code ist nicht immer optimal hinsichtlich Laufzeit oder Speicherverbrauch[7].
- **Wartbarkeit:** Kommentare, klare Code-Struktur und Dokumentation fehlen häufig[6].
- **Fehleranfälligkeit:** Auch Code, der vorerst funktionsfähig erscheint, kann immer noch Bugs oder Sicherheitslücken enthalten[2, 6].

Wie präsent diese Herausforderungen in datenanalytischen Aufgaben sind, soll in den folgenden Kapiteln untersucht werden. Vor allem durch den Vergleich von generiertem und manuell geschriebenem Code lassen sich die Stärken und Schwächen von LLMs in der Datenanalyse besser einschätzen.

## 2.4 Prompting mit Sprachmodellen

Mit dem Begriff *Prompting* wird das Verfahren beschrieben, ein zuvor trainiertes Sprachmodell allein durch spezifische Eingabetexte (*Prompts*) zu steuern, ohne weiteres trainieren des Modells. Im Gegensatz zur zuvor verbreiteten Vorgehensweise, ein vortrainiertes Modell für jede Aufgabe mit allen notwendigen Parametern komplett anzupassen, wird beim Prompting direkt auf das bereits eingearbeitete Wissen des Modells zurückgegriffen und steuert dadurch dessen Ausgabe durch passende Formulierungen[8]. Mit der Veröffentlichung großer Modelle wie GPT-3 zog dieses Vorgehen große Aufmerksamkeit auf sich, weil es diesen Modellen ermöglichte, durch spezifische Anweisungen in einer Prompt, komplexe Aufgaben zu lösen, wie Brown et al.(2020)[9] zeigten.

### Grundlegende Strategien

- **Zero-Shot, One-Shot und Few-Shot Prompting:** Bei *Zero-Shot* Prompting wird dem Modell lediglich eine Aufgabenbeschreibung gegeben, ohne Beispiele. Hierbei soll das Modell von alleine auf die richtige und gewünschte Lösung kommen. Bei *One-Shot*

wird dem genau ein Beispiel hinzugefügt, während *Few-Shot* mehrere Demonstrationsbeispiele bereitstellt. Erste Arbeiten, wie jene von Brown et al.[9], zeigten, dass schon wenige Beispiele im Prompt teils große Leistungsunterschiede bewirken können.

- **Instruction-based Prompting:** Anstatt nur Beispiele zu geben, werden präzise Anweisungen in Textform formuliert, wie beispielsweise „*Fasse den Text in drei Sätzen zusammen.*“. Ouyang et al.(2022) führen dafür auch ihr Modell *InstructGPT* ein, welches speziell darauf trainiert wurde, solche Anweisungen verlässlich und mit Berücksichtigung der Wünsche des Nutzers in korrekter Weise umzusetzen[10].
- **Chain-of-Thought Prompting:** Hierbei wird das Modell dazu angewiesen Schrittweise vorzugehen und diese Zwischenschritte explizit auszugeben. Dadurch liefern Modelle oft bessere und eher nachvollziehbare Ergebnisse zurück, wie Wei et al.(2022) zeigen[11].

### 3 LLMs in der Programmierung – aktueller Stand

Die Entwicklung von Large Language Models (LLMs) hat in den letzten Jahren nicht nur die Art und Weise, wie natürliche Sprache verarbeitet und generiert wird, verändert, sondern auch große Fortschritte in der automatisierten Code-Erstellung ermöglicht. Durch die Kombination aus leistungsstarken Modellarchitekturen wie Transformers, großen Mengen an Trainingsdaten und moderner Hardware haben LLMs heute eine große Präsenz in vielen Bereichen der Softwareentwicklung. In diesem Kapitel werden die aktuellen Entwicklungen und verfügbaren Modelle vorgestellt. Außerdem wird ein Überblick über ihre Einsatzmöglichkeiten in der Softwareentwicklung und Datenanalyse gegeben. Zum Schluss werden wichtige Studien und Arbeiten zur Code-Generierung betrachtet, darunter etwa die von Chen et al. (2021) vorgestellte Arbeit zu Codex, einem Modell, das speziell für die automatisierte Programmierung entwickelt wurde[2] und die von Liu et al. (2023) veröffentlichte Arbeit zur Evaluation von generiertem Code mithilfe von *EvalPlus*[1].

#### 3.1 Überblick und Vergleich von verschiedenen LLMs

Derzeit existiert eine Vielzahl an LLMs, darunter auch viele, die gezielt zur Code-Generierung entwickelt wurden. Zu den bekanntesten Beispielen zählen ChatGPT (GPTo1 als das modernste Modell), OpenAI Codex, Code Llama[12], StarCoder [13], CodeT5[14] oder CodeGen[15].

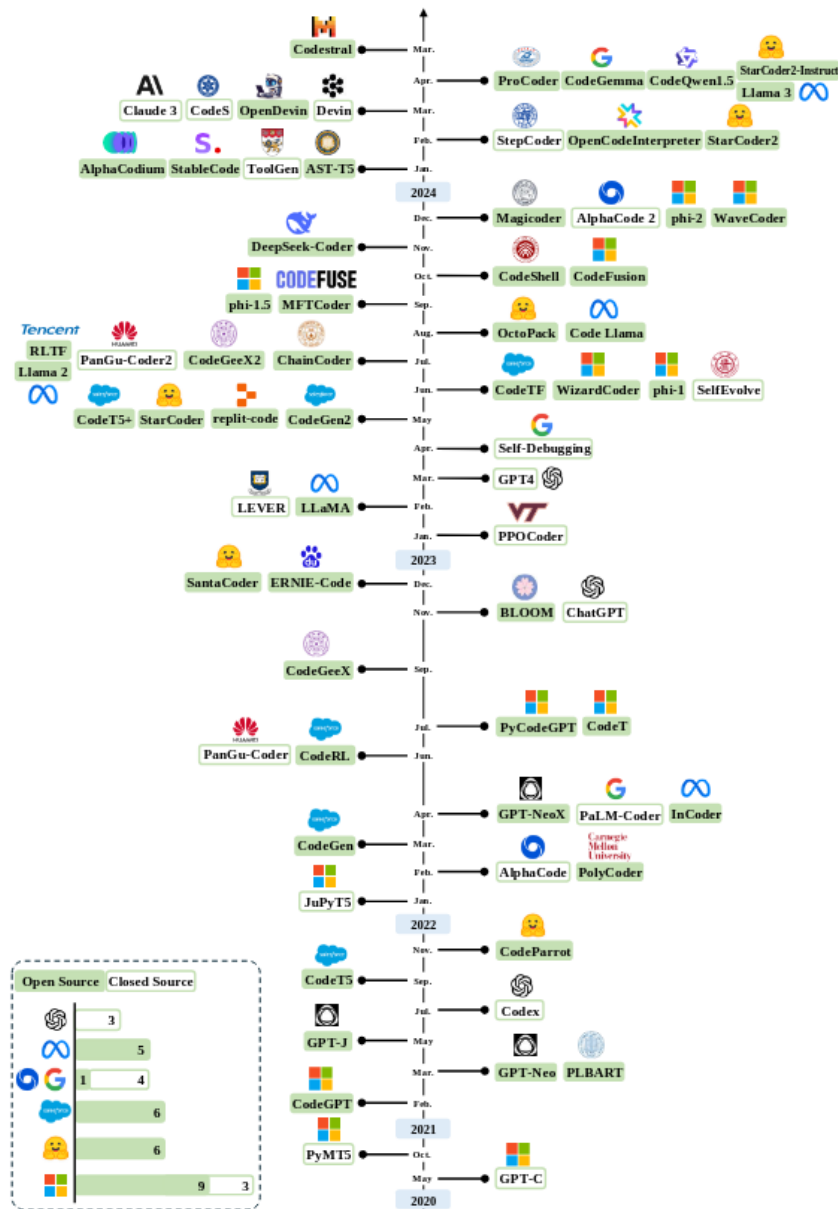


Abbildung 2: Chronologische Übersicht von Large Language Models für die Code Generierung der letzten Jahre (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation').

Diese Modelle teilen sich häufig folgende Merkmale:

1. **Transformer-Architektur:** Nahezu alle modernen LLMs beruhen auf dem Transformer-Modell.
2. **Große Parameteranzahl:** Typische LLMs verfügen über eine Vielzahl an Parametern und benötigen entsprechend umfangreiche Trainingsdaten, zu denen in vielen Fällen öffentlich verfügbare Code-Repositories (z. B. GitHub) zählen[2].
3. **Breite Sprachenunterstützung:** Neben Python werden häufig Java, JavaScript und andere Programmiersprachen abgedeckt[2, 5].

Ein Vergleich der LLMs lässt sich anhand verschiedener Kriterien vornehmen:

- **Größe und Trainingsdaten:** Modelle wie GPT-4 oder Code Llama sind mit einer Vielzahl an Code-Datensätzen trainiert und erreichen dadurch in Benchmarks eine hohe Erfolgsquote[1].
- **Lizenz und Offenheit:** Neben proprietären Modellen, wie GitHub Copilot und ChatGPT, existieren mit Code Llama, StarCoder[13] oder CodeGen[15] auch Open Source Alternativen.
- **Spezialisierung:** Einige Modelle sind speziell auf Code-Generierung abgestimmt (z.B. Code Llama, StarCoder[13]), wohingegen andere (z.B. ChatGPT) einen generellen Sprachkontext haben, um auch andere Fragen zu beantworten, der sich jedoch auch auf Code-Aufgaben anwenden lässt.

### 3.2 Einsatzgebiete von LLMs in der Programmierung

Die zunehmende Leistungsfähigkeit von Large Language Models (LLMs) ermöglicht es, Programmieraufgaben in diversen Bereichen zu automatisieren oder zu beschleunigen. Häufig genannte *Einsatzgebiete* sind dabei:

- **Code-Generierung:** Ermöglicht die Code-Generierung auf Grundlage von Beschreibungen aus natürlicher Sprache[2]. Ebenso bieten manche Modelle die Möglichkeit zu fertigen Funktionen Tests zu generieren, um dessen Funktionalität zu überprüfen.
- **Autovervollständigung:** Integriert in Entwicklungsumgebungen wie Visual Studio Code können Tools wie GitHub Copilot repetitive Abläufe direkt im Code vervollständigen oder Vorschläge zur Vervollständigung von neu begonnenem Code liefern[2].
- **Refactoring und Fehlersuche:** Dank ihrer Kontextsensitivität können LLMs bestehenden Code analysieren und an einigen Stellen Möglichkeiten zur Optimierung oder Korrektur vorschlagen[2, 14]. Dadurch lassen sich Bugs, Redundanz und ineffiziente Code-Strukturen frühzeitig identifizieren und beheben.
- **Automatisierte Dokumentation und Code-Kommentierung:** Viele Modelle bieten die Möglichkeit vorhandenen Code zu analysieren und dazu Kommentarblöcke oder gar ganze Dokumentationen zu erstellen[14, 5].

Obwohl diese Einsatzgebiete großes Potenzial bieten, sind LLMs nicht frei von Fehlern. Gerade bei komplexen Entscheidungen zur Programm- und Codearchitektur können diese oft mit dem Level durch das menschliche Fachwissen nicht mithalten [16].

### 3.3 Vergangene Studien und Arbeiten zur Code-Generierung

Die Forschung zur automatisierten Code-Generierung hat in den letzten Jahren eine rasante Entwicklung erlebt, wobei Arbeiten aus den Bereichen *Software Engineering*, *Large Language Models* und *Machine Learning* zusammenfließen. Jiang et al. (2024) haben in ihrer Arbeit “A Survey on Large Language Models for Code Generation“ eine Übersicht über die Entwicklung

der veröffentlichten Arbeiten zu LLMs und Software Engineering erstellt, welche in Abbildung 3 dargestellt ist.

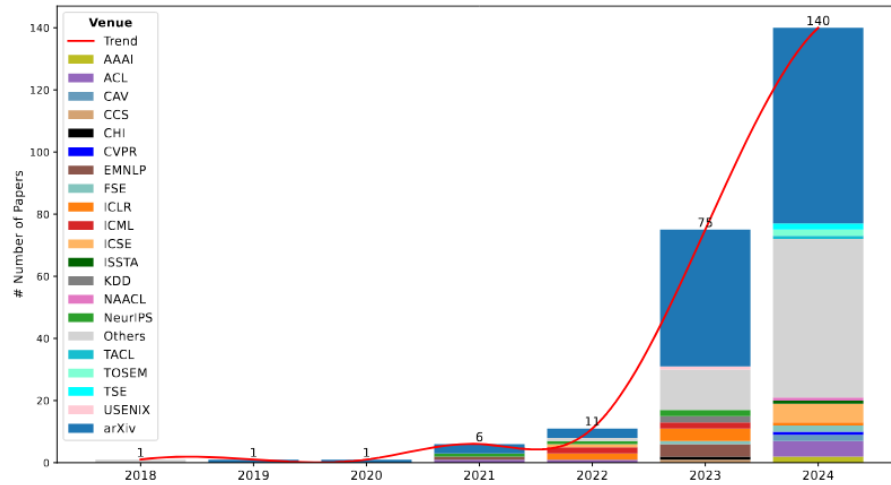


Abbildung 3: Übersicht der Verteilung von veröffentlichten Arbeiten zu LLMs und Software Engineering der letzten Jahren (nach Jiang et al. (2024), basierend auf 'A Survey on Large Language Models for Code Generation'[5]).

Im Folgenden werden einige vergangene Studien/Arbeiten zur Code Generierung mit LLMs vorgestellt:

1. **“Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”** von Liu et al. (2023) [1]:  
In diesem Paper wird untersucht, wie korrekt der von LLMs wie ChatGPT, Code Llama etc. generierte Code ist. Dafür wird *EvalPlus* eingeführt. Dies ist ein neues Evaluierungsframework, das bestehende Testdatensätze wie *HumanEval* durch weitere automatisierte Testfälle erweitert. Hier kommen Liu et al. zu dem Ergebnis, dass bisher viele Fehler in generiertem Code nicht erkannt wurden, wodurch die Modelle in ihrer Leistung überschätzt wurden. Die Autoren weisen darauf hin, wie wichtig umfassende Tests sind, um die tatsächliche Funktionalität der LLMs für die Codegenerierung zu bewerten.
2. **“Evaluating Large Language Models Trained on Code”** von Chen et al. (2021) [2]:  
In diesem Paper wird Codex vorgestellt. Dies ist ein LLM, das speziell auf öffentlich verfügbarem Code von Github trainiert wurde, um dessen Fähigkeiten Python Code zu schreiben zu analysieren. Dies wird mithilfe des HumanEval-Datensatzes untersucht. Hierbei soll im genauen Python-Code aus Docstrings generiert und dieser dann bewertet werden. Die Ergebnisse zeigen, dass Codex im Vergleich zu anderen Modellen wie GPT-3 deutlich besser abschneidet, jedoch bei komplexeren Aufgaben seine Grenzen erreicht. Eine mehrfach wiederholte Lösungsgenerierung verbessert die Erfolgsrate, was das Potenzial ihres Ansatzes verdeutlicht.

	Size	pass@k	k=1*	k=1	k=10	k=100	T <sub>1</sub> <sup>*</sup>	T <sub>10</sub> <sup>*</sup>	T <sub>100</sub> <sup>*</sup>
GPT-4 [49]	N/A	base	88.4						
		+extra	76.2						
Phind-CodeLlama [52]	34B	base	71.3	71.6	90.5	96.2	.2	.8	.8
		+extra	67.1	67.0	85.0	92.5	.2	.8	.8
WizardCoder-CodeLlama [38]	34B	base	73.2	61.6	85.2	94.5	.2	.8	.8
		+extra	64.6	54.5	78.6	88.9	.2	.8	.8
ChatGPT [48]	N/A	base	73.2	69.4	88.6	94.0			
		+extra	63.4	62.5	82.1	91.1			
CODELLAMA [54]	34B	base	51.8	52.0	82.4	95.0	.2	.8	.8
		+extra	42.7	43.1	73.7	89.4	.2	.8	.8
	13B	base	42.7	44.6	77.6	92.7	.4	.8	.8
		+extra	36.6	37.4	69.4	88.2	.4	.8	.8
	7B	base	37.8	39.2	69.1	89.7	.2	.8	.8
		+extra	34.1	34.5	61.4	82.9	.2	.8	.8
StarCoder [13]	15B	base	34.1	32.2	56.7	84.2	.2	.8	.8
		+extra	29.3	27.8	50.3	75.4	.2	.8	.8
CodeGen [46]	16B	base	32.9	32.2	56.0	81.5	.2	.6	.8
		+extra	26.8	27.2	48.4	71.4	.2	.6	.8
	6B	base	29.3	27.7	46.9	72.7	.2	.6	.8
		+extra	25.6	23.6	41.0	64.6	.2	.6	.8
	2B	base	24.4	18.4	39.8	66.8	.2	.8	.8
		+extra	20.7	15.1	34.8	55.8	.2	.2	.8
CODET5+ [64]	16B	base	31.7	32.2	58.5	83.5	.2	.6	.8
		+extra	26.2	27.4	51.1	76.4	.2	.6	.8
MISTRAL [26]	7B	base	28.7	28.1	55.2	83.8	.2	.8	.8
		+extra	23.8	23.7	48.5	76.4	.2	.8	.8
CodeGen2 [45]	16B <sup>4</sup>	base	19.5						
		+extra	16.5						
	7B	base	18.3	17.9	30.9	50.9	.2	.6	.8
		+extra	16.5	15.9	27.1	45.4	.2	.6	.8
	3B	base	15.9	15.2	23.9	38.6	.2	.4	.8
		+extra	12.8	12.9	21.2	34.3	.2	.4	.8
	1B	base	11.0	10.2	15.1	24.7	.2	.6	.6
		+extra	9.1	8.7	13.7	21.2	.2	.6	.6
VICUNA [12]	13B	base	16.5	15.3	30.1	54.8	.2	.8	.8
		+extra	15.2	13.9	25.8	46.7	.2	.8	.8
	7B	base	11.6	10.9	23.8	42.3	.2	.6	.6
		+extra	11.0	10.3	20.3	35.0	.2	.6	.6
SantaCoder [2]	1.1B	base	14.6	16.6	29.2	45.4	.4	.6	.8
		+extra	12.8	14.2	26.2	40.6	.4	.6	.8
INCODER [18]	6.7B	base	15.9	15.6	27.7	45.0	.2	.4	.6
		+extra	12.2	12.4	22.2	38.9	.2	.6	.6
	1.3B	base	12.2	10.0	15.9	25.2	.2	.6	.6
		+extra	10.4	7.9	13.5	20.7	.2	.6	.4
GPT-J [63]	6B	base	12.2	11.3	17.7	31.8	.2	.6	.6
		+extra	10.4	9.5	15.2	25.9	.2	.6	.6
GPT-NEO [5]	2.7B	base	7.9	6.5	11.8	20.7	.2	.6	.6
		+extra	6.7	6.0	9.0	16.8	.2	.6	.6
PolyCoder [70]	2.7B	base	6.1	5.9	10.2	17.1	.2	.4	.6
		+extra	5.5	5.3	7.9	13.6	.2	.6	.6
StableLM [60]	7B	base	2.4	2.7	7.5	15.8	.2	.6	.6
		+extra	2.4	2.6	6.2	11.9	.2	.6	.6

Abbildung 4: Vergleich der Ergebnisse von *HumanEval* und *EvalPlus* (nach Liu et al. (2023), basierend auf “Is your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation”[1]).

### 3. “A Survey on Large Language Models for Code Generation” von Jiang et al. (2024) [5]:

Dieses Paper gibt einen allgemeinen und umfassenden Überblick über den aktuellen Forschungsstand zu LLMs für die Codegenerierung. Es greift Themen wie Datenaufbereitung, Modellarchitekturen und Benchmarks auf. Zudem werden Herausforderungen, wie die praktische Einführung und ethische Fragen diskutiert. Die Autoren leiten sich wichtige Forschungsfragen ab und verdeutlichen, dass LLMs in der Codegenerierung große Fortschritte gemacht haben, aber es weiterhin Potenzial zur Optimierung gibt.

4. **“Evaluating Language Models for Efficient Code Generation”** von Liu et al. (2024) [17]:

Auch in dieser Arbeit von Jiawei Liu wird die Effizienz von Code untersucht, welcher von LLMs generiert wird. Hierbei mit Fokus auf Performance und Ressourcennutzung. Dafür wird *Differential Performance Evaluation (DPE)* entwickelt und der *EvalPerf*-Benchmark eingeführt. Dieser enthält komplexere Programmieraufgaben als der zuvor eingeführt *EvalPlus*. Hier kommt man zu dem Entschluss, dass größere Modelle nicht automatisch auch effizienteren Code erzeugen. Stattdessen werden Effizienz und Korrektheit des Codes durch *Instruction Tuning* (gezieltes Trainieren des Modells, um besser auf Anweisungen in natürlicher Sprache zu reagieren) verbessert.

Zusammenfassend zeigen die genannten Studien, dass LLMs zwar großes Potenzial zur automatisierten Code Generierung besitzen, sie aber immer noch Probleme aufweisen und menschliche Entwickler nicht komplett ersetzen können. Besonders bei komplexeren Aufgaben, spezifischen Anforderungen oder Fragen zur Softwarearchitektur stoßen sie an ihre Grenzen.

## 4 Ausgangsdaten und Testfallspezifikation

### 4.1 Überblick und Reduktion der Datengrundlage

Die Datengrundlage für die empirische Untersuchung bilden Kriminalitätsstatistiken (sogenannte *Fallzahlen*) der Stadt Berlin[18]. Die zugehörige Excel-Datei umfasst mehrere Sheets, jeweils zu den Jahren 2014–2023. Darin sind die Straftaten pro Bezirk (bzw. Ober- und Unterbezirke) aufgelistet. In der Ursprungsform gliedert sich die Tabelle wie folgt:

- **Oberbezirk:** Enthält aggregierte Zahlen der jeweiligen Unterbezirke.
- **Unterbezirke:** Ausführlichere Aufschlüsselung der Straftaten innerhalb des Oberbezirks.
- **Spalten mit Straftat-Kategorien:** u. a. „Straftaten insgesamt“, „Körperverletzungen“, „Diebstahl“, ...

Für die weitere Analyse wird jedoch nur auf *Oberbezirks*-Daten zurückgegriffen. Die Unterbezirke werden **nicht** berücksichtigt, um die Komplexität zu reduzieren und das Fokusgebiet auf obergeordnete Bezirke zu legen. Ziel ist eine übersichtlichere Zusammenfassung, bei der „Oberbezirk“ die wichtigste Bezugsgröße ist.

### 4.2 Umwandlung in Pandas DataFrames

Jedes Jahr (bzw. jedes Sheet) soll in einer separaten **pandas**-DataFrame-Tabelle abgebildet werden. Dies geschieht folgendermaßen:

1. **Reduzieren der Datengrundlage:** Entfernen der Unterbezirke, um nur die Oberbezirke zu behalten.
2. **Bereinigung und Umbenennung:** Unnötige Spalten werden entfernt, Spaltennamen ggf. standardisiert (z. B. „Bezirk“, „Straftaten\_insgesamt“).

3. **Einlesen der Excel-Sheets:** Mit `pandas.read_excel(...)` wird jede Jahres-Tabelle eingelesen.
4. **Speicherung in DataFrame:** Pro Sheet entsteht ein bereinigtes und vereinheitlichtes DataFrame.

### 4.3 Testfälle und Vorgehen

Insgesamt sind vier Testfälle definiert, die unterschiedliche Aspekte der Datenanalyse abdecken. Für jeden Testfall werden **15 Ausführungen** erzeugt, wobei drei verschiedene Prompting-Strategien (jeweils fünf Wiederholungen) zum Einsatz kommen:

- **Strategie A: Prompt wie ein „normaler User“**  
Hier wird eine einfache, natürlichsprachliche Anfrage formuliert, ohne viele zusätzliche Informationen.
- **Strategie B: Prompt mit Metadaten**  
In diesem Ansatz werden neben der eigentlichen Anfrage auch relevante Details wie Spaltennamen oder Strukturhinweise explizit übergeben.
- **Strategie C: Prompt mit *Chain of Thought***  
Das Modell erhält schrittweise Gedankenanstöße oder Zwischenlogik (z. B. „Zuerst filtern, dann sortieren, ...“), um den Code schrittweise aufzubauen und zu erklären.

#### 4.3.1 Testfall 1: Sortierung und Ausgabe der Fallzahlen 2023

**Zielsetzung:** Die Daten des Jahres 2023 (`Fallzahlen_2023`) sollen nach der Spalte „Straftaten insgesamt“ sortiert und anschließend in einem `pandas`-DataFrame ausgegeben werden.

**Vorgehen:**

1. Einlesen der `Fallzahlen_2023`.
2. Extraktion der relevanten (Ober-)Bezirke.
3. Sortierung nach `Straftaten_insgesamt` in absteigender oder aufsteigender Reihenfolge.
4. Ausgabe als `pandas` DataFrame.

**Erwartete Ausgabe:** Ein DataFrame mit mindestens folgenden Spalten:

Bezirk	Straftaten_insgesamt
(z. B. Mitte)	(z. B. 82 000)
(z. B. Neukölln)	(z. B. 50 000)
...	...

Zusätzlich können weitere Spalten (z. B. Raub, Diebstahl) enthalten sein, sofern sie nicht entfernt wurden.



#### 4.3.2 Testfall 2: Join aller Tabellen und „Bezirks-Topwert“

**Zielsetzung:** Alle DataFrames von 2014–2023 sollen *vereint* werden (Join), sodass die **Summe der Straftaten aller Jahre** pro Bezirk ermittelbar ist. Anschließend werden die Bezirke nach „den meisten Straftaten insgesamt“ sortiert ausgegeben.

**Vorgehen:**

1. Einlesen und Bereinigung der einzelnen DataFrames (2014–2023).
2. Zusammenführen nach dem **Bezirk**-Merkmal.
3. Aggregation der „Straftaten\_insgesamt“ pro Jahr zu einer Gesamtzahl über alle Jahre.
4. Ermittlung und Ausgabe der **Bezirke** sortiert nach den Summenwerten.

**Erwartete Ausgabe:** Ein DataFrame mit mindestens folgenden Spalten:

Bezirk	Straftaten_insgesamt 2014-2023
(z. B. Mitte)	(z. B. 820 000)
(z. B. Neukölln)	(z. B. 560 000)
...	...

#### 4.3.3 Testfall 3: Prozentuale Verteilung der Straftaten

**Zielsetzung:** Für ein ausgewähltes Jahr (etwa 2023) soll ermittelt werden, welcher Anteil aller Berliner Straftaten auf die jeweiligen Bezirke entfällt.

**Vorgehen:**

1. Einlesen des relevanten Sheets (z. B. **Fallzahlen\_2023**).
2. Berechnung des Anteils pro Bezirk:

$$\text{Prozent} = \frac{\text{Straftaten\_insgesamt pro Bezirk}}{\text{Straftaten\_Gesamtsumme}} \times 100$$

3. Ausgabe als DataFrame mit Spalten wie „Bezirk“, „Straftaten\_insgesamt“, „Anteil\_%“.

**Erwartete Ausgabe:** Ein DataFrame, bei dem jede Zeile einen **Bezirk** darstellt und mindestens folgende Spalten beinhaltet:

Bezirk	Straftaten_insgesamt	Anteil_(%)
Mitte	82 000	(z. B. 24,1%)
Neukölln	50 000	(z. B. 14,7%)
...	...	...

#### 4.3.4 Testfall 4: Zeitreihe über die Jahre 2014–2023

**Zielsetzung:** Ausgabe der **prozentualen Entwicklung** (bezogen auf *Berlin insgesamt*) der **Straftaten\_insgesamt** pro Jahr, bezogen auf das Vorjahr. Damit soll ersichtlich werden, wie sich das Gesamtaufkommen an Straftaten im Zeitverlauf verändert hat.

**Vorgehen:**

1. Einlesen sämtlicher Jahres-Sheets.
2. Addition sämtlicher Bezirkswerte für jedes Jahr, um die Gesamtzahl an Straftaten je Jahr zu erhalten.
3. Direkter Vergleich „prozentuale Änderung“ zum Vorjahr.
4. Ausgabe eines **pandas** DataFrames als Zeitreihe.

**Erwartete Ausgabe:** Ein DataFrame mit mindestens zwei Spalten:

Jahr	Straftaten__Veränderung__(%) zu Vorjahr
2014	0% (Basiswert)
2015	+3,5%
2016	-1,2%
...	...

## 5 Methodik

### 5.1 Vorgehensweise der Untersuchung

In der Untersuchung soll geprüft werden, inwieweit Large Language Models in der Lage sind gängige Datenanalyse-Schritte auf Grundlage eines gegebenen Datensatzes durchzuführen. Hierbei wird ChatGPT als aktueller Marktführer mit dem Sprachmodell GPTo1-mini, welches das neueste Modell ist, verwendet. Ebenso gilt es herauszufinden wie qualitativ und effizient diese Lösung ist. Hierbei bezieht es sich auf die Forschungsfragen aus Kapitel 1.1. Für die Vorgehensweise hierbei wird zuerst der verwendete Datensatz von Berlin Open Data, wie in Kapitel 4 beschrieben, heruntergebrochen und dann an das Modell übergeben und dazu, zu jeder verwendeten Prompting Strategie, eine Prompt verfasst. Diese Prompts können in Kapitel 5.2 eingesehen werden. Im Anschluss wird der Code ausgeführt, wobei seine Performanz gemessen wird, und es werden manuelle Analysen durchgeführt um die Qualität und Effizienz des generierten Codes zu bewerten. Die genauen Auswertungskriterien sind in Kapitel 5.3 aufgeführt. Die Ergebnisse der Auswertung werden in Kapitel 6 detailliert dargestellt.

### 5.2 Testfälle der Datenanalyse

#### 5.2.1 Testfall 1

Im ersten Testfall soll der Datensatz nach einer gewissen Spalte sortiert werden. Die Begründung hierfür ist, dass dies eine sehr einfache, aber auch sehr häufig auftretende Datenanalyse-

Aufgabe ist und somit einen guten Einstieg in die Untersuchung darstellt. Die Prompts für diese Aufgabe lauten:

- **Zero-Shot Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Hier ist der Inhalt des Sheets 'Fallzahlen\_2023': [DataFrame]. Erstelle mir ein Skript in Python, das die Daten aus der Excel-Datei einliest, nach den Straftaten insgesamt der Bezirke sortiert und in einem Dataframe speichert.*
- **Instruction Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Hier ist der Inhalt des Sheets 'Fallzahlen\_2023': [DataFrame]. Erstelle mir ein Skript in Python, das die Daten aus der Excel-Datei einliest, nach der Spalte 'Straftaten\_insgesamt' der Bezirke sortiert und in einem Pandas Dataframe speichert. Die Zeilen mit den LOR-Schlüsseln 999900 und 999999 sollen bei der Sortierung außer Acht gelassen werden, da es sich bei diesen nicht um Bezirke handelt.*
- **Chain-of-Thought Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Der Inhalt des Sheets ist als pandas DataFrame [DataFrame] gegeben. Bitte erstelle mir ein Python-Skript, das die folgenden Schritte ausführt:*
  - 1. Lies die Daten des Sheets 'Fallzahlen\_2023' der Excel-Datei 'Fallzahlen.xlsx' ein.*
  - 2. Sortiere die Daten nach der Spalte 'Straftaten\_insgesamt' absteigend. Zur Sortierung sollen die Zeilen mit den LOR-Schlüsseln 999900 und 999999 nicht beachtet werden, da es sich bei diesen nicht um Bezirke handelt. Sie sollen aber am Ende des Dataframes stehen bleiben.*
  - 3. Speichere das Ergebnis der Sortierung in einem Pandas Dataframe ab.**Achte darauf, dass das Skript robust ist und potentielle Fehler, wie fehlende Spalten berücksichtigt.*

### 5.2.2 Testfall 2 Verbund und Aggregation

Für den zweiten Testfall sollen die Tabellen der Excel Datei durch einen Join zusammengeführt und dann die Bezirke nach den Straftaten insgesamt von allen Jahren kombiniert geliefert werden. Die Prompts für diese Aufgabe lauten:

- **Zero-Shot Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, das die Daten aller Sheets zusammenliest, sie nach der Anzahl der Straftaten insgesamt pro Bezirk sortiert und in einem Pandas DataFrame speichert. Hier sind die Daten eines der Sheets als Beispiel: [DataFrame]*
- **Instruction Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, das die Daten der einzelnen Bezirke (Zeilen) aller Sheets mit einem Join zusammenfügt, sie nach der akkumulierten Spalte 'Straftaten\_insgesamt' pro Bezirk sortiert und in einem Pandas DataFrame speichert. Die Zeilen mit den LOR-Schlüsseln 999900 und 999999 sollen bei der Sortierung nicht beachtet werden, da es sich hierbei nicht um Bezirke handelt. Hier sind die Daten eines der Sheets als Beispiel: [DataFrame]*

- **Chain-of-Thought Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, das folgende Anforderungen erfüllen soll:*
  1. *Die Excel-Datei einlesen und die Sheets als DataFrames speichern.*
  2. *Die DataFrames der einzelnen Sheets zusammen joinen, sodass pro Zeile (jede Zeile ist ein eigener Bezirk) der akkumulierte Wert der einzelnen Straftaten steht.*
  3. *Das neue gejointe DataFrame nach der Spalte 'Straftaten\_insgesamt' sortieren. Für die Sortierung sollen die Zeilen mit den LOR-Schlüsseln 999900 und 999999 nicht beachtet werden, da es sich hierbei nicht um Bezirke handelt. Sie sollen aber am Ende des DataFrames stehen bleiben.*
  4. *Das sortierte Pandas DataFrame zurückgeben.**Hier ist der Inhalt eines der Sheets als Beispiel: [DataFrame]*

### 5.2.3 Testfall 3

Im dritten Testfall soll das Sprachmodell die prozentualen Anteile der gesamten Straftaten der Bezirke von ganz Berlin berechnen. Die Prompts für diese Aufgabe lauten:

- **Zero-Shot Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, welches den prozentualen Anteil der gesamten Straftaten der einzelnen Bezirke von den gesamten Straftaten von ganz Berlin berechnet. Hier ist der Inhalt des Sheets 'Fallzahlen\_2023': [DataFrame]*
- **Instruction Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, welches den prozentualen Anteil der einzelnen Bezirke von ganz Berlin für die Spalte 'Straftaten\_insgesamt' berechnet. Jede Zeile der Tabelle ist ein einzelner Bezirk und 'Berlin (PKS gesamt)' ist die Gesamtanzahl von ganz Berlin. Hier ist der Inhalt des Sheets 'Fallzahlen\_2023': [DataFrame]*
- **Chain-of-Thought Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, welches folgende Anforderungen erfüllt:*
  1. *Die Excel-Datei einlesen*
  2. *Die Tabelle als Pandas DataFrame speichert*
  3. *Überprüfen, ob die notwendigen Spalten 'Bezirke' und 'Straftaten\_insgesamt' vorhanden sind*
  4. *Finde die Gesamtzahl der Straftaten für ganz Berlin in der Zeile mit dem Bezirk 'Berlin (PKS gesamt)'*
  5. *Berechne den prozentualen Anteil der einzelnen Bezirke von ganz Berlin für die Spalte 'Straftaten\_insgesamt'*
  6. *Das Ergebnis als DataFrame zurückgeben**Hier ist der Inhalt des Sheets 'Fallzahlen\_2023': [DataFrame]*

### 5.2.4 Testfall 4

Im vierten Testfall sollen die Skripte eine Zeitreihe der prozentualen Veränderung der gesamten Straftaten von ganz Berlin als Pandas Dataframe erstellen. Die Prompts für diese Aufgabe lauten:

- **Zero-Shot Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, das die Daten aller Sheets analysiert und eine Zeitreihe mit der prozentualen Veränderung zum jeweiligen Vorjahr der gesamten Straftaten von ganz Berlin als Pandas Dataframe erstellt.*  
*Hier sind die Daten eines der Sheets als Beispiel: [DataFrame]*
- **Instruction Prompting:** *Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Erstelle mir ein Python Skript, das die Daten aller Sheets analysiert und eine Zeitreihe mit der prozentualen Veränderung der Spalte SStraftaten\_insgesamt zum jeweiligen Vorjahr von der Zeile "Berlin (PKS gesamt)" als Pandas Dataframe erstellt. Die Sheets folgen der Namensgebung 'Fallzahlen\_2014', 'Fallzahlen\_2015', etc.*  
*Hier sind die Daten eines der Sheets als Beispiel: [DataFrame]*
- **Chain-of-Thought Prompting:** *Ich habe eine Excel-Datei mit dem Namen 'Fallzahlen.xlsx'. Diese Datei enthält mehrere Sheets, die nach dem Muster 'Fallzahlen\_2014', 'Fallzahlen\_2015', usw. benannt sind. Jedes Sheet enthält Daten, darunter eine Spalte namens 'Straftaten\_insgesamt'. Erstelle mir ein Python Skript mit den folgenden Schritten:*
  - 1. Lese alle Sheets der Excel-Datei ein und speichere jedes Sheet in einem separaten Pandas DataFrame.*
  - 2. Extrahiere den Wert der Spalte 'Straftaten\_insgesamt' für die Zeile 'Berlin (PKS gesamt)' aus jedem DataFrame.*
  - 3. Berechne die prozentuale Veränderung des Werts 'Straftaten\_insgesamt' zum jeweiligen Vorjahr.*
  - 4. Speichere die Ergebnisse in einem neuen Pandas DataFrame, das die Jahre und die prozentuale Veränderung enthält.**Hier sind die Daten eines der Sheets als Beispiel: [DataFrame]*

### 5.3 Auswertungskriterien

Die Auswertung der generierten Python Skripte erfolgt anhand der in Kapitel 1.1 definierten Kriterien. Um die Korrektheit des Codes zu messen wird das Pass@k Verfahren verwendet, dabei steht "k" für die Anzahl der ausgeführten Versuche pro Testfall. In diesem Experiment wird sich auf k=5 pro Prompt beschränkt, um eine gute Balance zwischen Genauigkeit und Rechenzeit zu finden. Bei diesem Verfahren ergibt sich als Ergebnis ein Prozentsatz über die Anzahl der erfolgreichen Versuche. Die ausgeführten Versuche werden anschließend in erfolgreich und nicht erfolgreich unterteilt und getrennt genauer betrachtet. Um zu entscheiden, ob ein Versuch erfolgreich war, wird darauf geachtet, ob die Ausführung fehlerfrei verläuft und ob das Ergebnis wie erwartet ist. Im Code wird dabei darauf geschaut, welche Bibliotheken, Funktionen und Pandas Dataframes benutzt wurden. In der genaueren Analyse des Codes wird bei den nicht erfolgreichen Versuchen untersucht, warum der Code nicht korrekt ausgeführt wurde und was für Verbesserungen vorgenommen werden können. Bei den erfolgreichen Versuchen hingegen wird analysiert, wie der Code strukturiert ist, ob er gut dokumentiert ist, ob er erweiterbar ist und wie die Laufzeit und Ressourcennutzung des Codes abschneidet.

## 5.4 Verwendete Tools und Daten

1. **Large Language Model:** Als Large Language Model wird ChatGPT mit GPTo1-mini verwendet, da ChatGPT als aktueller Marktführer gilt und GPTo1-mini das neueste und leistungsfähigste Modell ist, welches mit der OpenAI API verfügbar ist.
2. **Libraries:** Für die Datenanalyse wird die Python-Bibliothek `pandas` verwendet, um die Excel-Dateien einzulesen und zu verarbeiten.
3. **Datenquelle:** Die Daten stammen aus den Kriminalitätsstatistiken der Stadt Berlin, die auf der Plattform Berlin Open Data veröffentlicht wurden.

## 6 Auswertung der Python-Code-Generierung zur Datenanalyse durch LLMs

### 6.1 Testfall 1: Sortierung und Ausgabe der Fallzahlen 2023

Im ersten Testfall wurde ein Python-Skript generiert, das die Excel-Tabelle „Fallzahlen\_2023“ nach der Anzahl der Straftaten insgesamt in 2023 sortieren sollte. Hierfür wurden drei verschiedene Prompting-Strategien (A, B, C4.3) verwendet, wobei jede Strategie fünfmal ausgeführt wurde. Die wichtigsten Beobachtungen sind:

#### Erfolgsquote (Pass@15):

- **Prompt 1 (Zero-Shot Prompting):** Alle fünf Ausführungen waren teils erfolgreich. Die Daten wurden zwar korrekt sortiert, jedoch wurden die Zeilen mit den LOR-Schlüsseln 999900 und 999999 (die nicht zu den Bezirken gehören) in die Sortierung einbezogen, was nicht der gewünschten Anforderung entsprach.
- **Prompt 2 (Instruction Prompting):** Alle fünf Ausführungen waren teils erfolgreich. Die Zeilen mit den LOR-Schlüsseln 999900 und 999999 wurden korrekt aus der Sortierung ausgeschlossen, jedoch wurden sie vollständig aus dem DataFrame entfernt, was ebenfalls nicht der gewünschten Anforderung entsprach.
- **Prompt 3 (Chain of Thought Prompting):** Alle fünf Ausführungen waren erfolgreich. Die Skripte sortierten die Daten korrekt nach der Spalte „Straftaten\_insgesamt“ und behielten dabei die Zeilen mit den LOR-Schlüsseln 999900 und 999999 am Ende des DataFrames, wie gewünscht.

Berücksichtigt man die Teilerfolge als fehlgeschlagen, ergibt sich ein **Pass@15 von 33%** (5 von 15 Ausführungen waren vollständig erfolgreich). Betrachtet man die Teilerfolge jedoch als erfolgreich, ergibt sich ein **Pass@15 von 100%** (15 von 15 Ausführungen waren teils erfolgreich).

#### Korrektheit und typische Fehlerquellen:

- **Prompt 1:** Der Hauptfehler lag darin, dass die Zeilen mit den LOR-Schlüsseln 999900 und 999999 (also keinem Bezirk zuzuordnende Straftaten und Berlin gesamt) in die Sortierung einbezogen wurden. Dies führte dazu, dass die Sortierung nicht den Anforderungen entsprach, da diese Zeilen nicht zu den Bezirken gehören.
- **Prompt 2:** Hier wurden die Zeilen mit den LOR-Schlüsseln 999900 und 999999 (also keinem Bezirk zuzuordnende Straftaten und Berlin gesamt) zwar korrekt aus der Sortierung ausgeschlossen, jedoch wurden sie vollständig aus dem DataFrame entfernt, was ebenfalls nicht den Anforderungen entsprach.
- **Prompt 3:** Diese Strategie war die einzige, die alle Anforderungen erfüllte. Die Daten wurden korrekt sortiert, und die Zeilen mit den LOR-Schlüsseln 999900 und 999999 (also keinem Bezirk zuzuordnende Straftaten und Berlin gesamt) wurden am Ende des DataFrames belassen.

### Performance und Ressourcenverbrauch:

- Die Laufzeit betrug bei allen erfolgreichen und teils erfolgreichen Skripten nur wenige Zehntelsekunden (zwischen 0,49 und 0,56 Sekunden).
- Der Speicherverbrauch bewegte sich bei rund 150,MB *Maximum Resident Set Size*, was für diesen kleinen Datensatz sehr effizient ist.
- Die CPU-Auslastung lag bei allen Ausführungen zwischen 445% und 498%, was auf eine effiziente Nutzung der verfügbaren Ressourcen hinweist.

### Codequalität und Wartbarkeit:

- **Struktur:** Die meisten Skripte bestanden aus wenigen, übersichtlichen Schritten: *Daten einlesen, Spalte bereinigen/umbenennen, sortieren, Datei abspeichern*. Die Skripte von Prompt 3 waren dabei am strukturiertesten und enthielten zusätzliche Schritte zur Fehlerbehandlung und Robustheit.
- **Dokumentation:** Die Skripte waren in der Regel gut kommentiert und erklärten die Schritte und die Funktionsweise. Dies war insbesondere bei den Skripten von Prompt 3 der Fall, die zusätzliche Kommentare zur Fehlerbehandlung und zur Logik der Sortierung enthielten.
- **Erweiterbarkeit:** Die Skripte von Prompt 3 waren am einfachsten zu erweitern, da sie bereits eine robuste Fehlerbehandlung enthielten und die Logik der Sortierung klar dokumentiert war. Die Skripte von Prompt 1 und 2 waren weniger robust, da sie keine Fehlerbehandlung für mögliche Änderungen in den Spaltennamen oder Datenstrukturen enthielten.

**Fazit zu Testfall 1:** Die Ergebnisse zeigen, dass die **Prompting-Strategie C (Chain of Thought)** die zuverlässigste Methode zur Generierung von Python-Code für die Sortierung eines Excel-Datensatzes ist. Diese Strategie lieferte in allen Fällen korrekte und robuste Skripte, die den Anforderungen entsprachen. Die Strategien A und B waren weniger zuverlässig, da sie entweder die Sortierung nicht korrekt durchführten oder die Daten unvollständig zurückgaben.

In Bezug auf **Performance** gab es keine Probleme, und alle Skripte waren effizient. Die Codequalität war bei den Skripten von Prompt 3 am höchsten, da sie gut dokumentiert und robust waren. Für produktive Einsätze ist daher die Verwendung von **Chain of Thought**-Prompts zu empfehlen, um sicherzustellen, dass der generierte Code den Anforderungen entspricht und leicht erweitert werden kann.

## 6.2 Testfall 2: Join aller Tabellen und Bezirks-Topwert

Im zweiten Testfall wurde ein Python-Skript generiert, das die Daten aller Excel-Sheets (2014–2023) zusammenführt, die Straftaten pro Bezirk über die Jahre summiert und die Bezirke nach der Gesamtzahl der Straftaten sortiert ausgibt. Hierfür wurden drei verschiedene Prompting-Strategien (Zero-Shot, Instruction Prompting, Chain of Thought) verwendet, wobei jede Strategie fünfmal ausgeführt wurde. Die wichtigsten Beobachtungen sind:

### Erfolgsquote (Pass@15):

- **Prompt 1 (Zero-Shot Prompting):** Drei von fünf Ausführungen waren teils erfolgreich. Die Daten wurden korrekt zusammengeführt und sortiert, jedoch wurden die unerwünschten LOR-Schlüssel (999900 und 999999) nicht korrekt behandelt. Zwei Ausführungen scheiterten vollständig, da sie die Daten nicht korrekt aggregierten oder falsche Spaltennamen verwendeten.
- **Prompt 2 (Instruction Prompting):** Eine von fünf Ausführungen war erfolgreich. Die erfolgreiche Ausführung (Execution 4) führte den Join korrekt durch und filterte die unerwünschten LOR-Schlüssel. Die anderen Ausführungen scheiterten entweder an der korrekten Filterung oder an der Aggregation der Daten.
- **Prompt 3 (Chain of Thought Prompting):** Vier von fünf Ausführungen waren erfolgreich. Die Skripte führten den Join korrekt durch, filterten die unerwünschten LOR-Schlüssel (999900 und 999999) aus der Sortierung und behielten sie im DataFrame. Eine Ausführung war teils erfolgreich, da sie Warnungen aufgrund von Pandas-Copy-Warnings warf, aber das korrekte Ergebnis lieferte.

Die Gesamterfolgsquote beträgt **33%** (5 von 15 Ausführungen waren vollständig erfolgreich).

### Erfolgsquoten der einzelnen Prompts:

- **Prompt 1 (Zero-Shot):** 0% vollständig erfolgreich, 60% teils erfolgreich, 40% nicht erfolgreich.



- **Prompt 2 (Instruction Prompting):** 20% vollständig erfolgreich, 20% teils erfolgreich, 60% nicht erfolgreich.
- **Prompt 3 (Chain of Thought):** 80% vollständig erfolgreich, 20% teils erfolgreich, 0% nicht erfolgreich.

#### Korrektheit und typische Fehlerquellen:

- **Prompt 1 (Zero-Shot):** Der Hauptfehler lag darin, dass die unerwünschten LOR-Schlüssel (999900 und 999999) nicht korrekt aus der Sortierung ausgeschlossen wurden. In einigen Fällen wurden sie sogar in die Sortierung einbezogen, was zu falschen Ergebnissen führte. Zusätzlich gab es Probleme bei der Aggregation der Daten über die Jahre.
- **Prompt 2 (Instruction Prompting):** Die meisten Ausführungen scheiterten daran, die unerwünschten LOR-Schlüssel korrekt zu filtern. In einigen Fällen wurden sie vollständig aus dem DataFrame entfernt, was nicht den Anforderungen entsprach. Eine Ausführung war erfolgreich, da sie die Filterung korrekt implementierte.
- **Prompt 3 (Chain of Thought):** Diese Strategie war die zuverlässigste. Die meisten Ausführungen führten den Join korrekt durch, filterten die unerwünschten LOR-Schlüssel aus der Sortierung und behielten sie im DataFrame. Eine Ausführung warf Warnungen aufgrund von Pandas-Copy-Warnings, lieferte aber das korrekte Ergebnis.

#### Performance und Ressourcenverbrauch:

- Die Laufzeit betrug bei allen Ausführungen zwischen 2,83 und 3,28 Sekunden, was für die Verarbeitung von 10 Excel-Sheets mit insgesamt 140 Zeilen sehr effizient ist.
- Der Speicherverbrauch bewegte sich bei rund 150–155 MB *Maximum Resident Set Size*, was für diesen Datensatz angemessen ist.
- Die CPU-Auslastung lag bei allen Ausführungen zwischen 157% und 168%, was auf eine effiziente Nutzung der verfügbaren Ressourcen hinweist.

#### Codequalität und Wartbarkeit:

- **Struktur:** Die Struktur ist in allen drei Prompts gut unterteilt und übersichtlich. Die Skripte von Prompt 3 waren jedoch am strukturiertesten und enthielten eine etwas genauere Unterteilung der Schritte.
- **Dokumentation:** Alle Skripte sind gut kommentiert und erklärten die einzelnen Schritte.
- **Erweiterbarkeit:** Alle Skripte sind leicht erweiterbar und können einfach angepasst werden, um zusätzliche Funktionalitäten hinzuzufügen. Es wurden teils auch Erweiterungsmöglichkeiten für die Skripte gegeben, wie zum Beispiel das generieren einer neuen Excel-Datei, sofern dieses nicht schon im Skript selbst enthalten war.

**Fazit zu Testfall 2:** Die Ergebnisse zeigen, dass die **Chain of Thought (CoT)**-Strategie die zuverlässigste Methode zur Generierung von Python-Code für die Zusammenführung und Sortierung von Excel-Daten ist. Diese Strategie lieferte in den meisten Fällen korrekte und robuste Skripte, die den Anforderungen entsprachen. Die **Zero-Shot**- und **Instruction Prompting**-Strategien waren weniger zuverlässig, da sie entweder die Filterung nicht korrekt durchführten oder die Daten unvollständig zurückgaben.

In Bezug auf **Performance** gab es keine Probleme, und alle Skripte waren effizient. Die Codequalität war bei den Skripten von Prompt 3 am höchsten, da sie gut dokumentiert und robust waren. Für produktive Einsätze ist daher die Verwendung von **Chain of Thought**-Prompts zu empfehlen, um sicherzustellen, dass der generierte Code den Anforderungen entspricht und leicht erweitert werden kann.

### 6.3 Testfall 3: Prozentuale Verteilung der Straftaten

Im dritten Testfall wurde ein Python-Skript generiert, das den prozentualen Anteil der Straftaten pro Bezirk an den gesamten Straftaten in Berlin berechnet. Hierfür wurden drei verschiedene Prompting-Strategien (Zero-Shot, Instruction Prompting, Chain of Thought) verwendet, wobei jede Strategie fünfmal ausgeführt wurde. Die wichtigsten Beobachtungen sind:

#### Erfolgsquote (Pass@15):

- **Prompt 1 (Zero-Shot):** Alle fünf Ausführungen waren erfolgreich. Die Skripte berechneten die prozentualen Anteile korrekt und speicherten die Ergebnisse in einer Excel-Datei.
- **Prompt 2 (Instruction Prompting):** Alle fünf Ausführungen waren erfolgreich. Die Skripte berechneten die prozentualen Anteile korrekt und speicherten die Ergebnisse in einer Excel-Datei.
- **Prompt 3 (Chain of Thought):** Vier von fünf Ausführungen waren erfolgreich. Eine Ausführung scheiterte aufgrund eines Syntaxfehlers, bei dem ein Python-Schlüsselwort („if“ und „not“) ins Deutsche übersetzt wurde.

Die Gesamterfolgsquote beträgt **93%** (14 von 15 Ausführungen waren erfolgreich).

#### Erfolgsquoten der einzelnen Prompts:

- **Prompt 1 (Zero-Shot):** 100% erfolgreich.
- **Prompt 2 (Instruction Prompting):** 100% erfolgreich.
- **Prompt 3 (Chain of Thought):** 80% erfolgreich, 20% nicht erfolgreich (Syntaxfehler).

### Korrektheit und typische Fehlerquellen:

- **Prompt 1 (Zero-Shot):** Alle Ausführungen waren erfolgreich. Die Skripte berechneten die prozentualen Anteile korrekt und behandelten die unerwünschten LOR-Schlüssel (999900 und 999999) korrekt.
- **Prompt 2 (Instruction Prompting):** Alle Ausführungen waren erfolgreich. Die Skripte berechneten die prozentualen Anteile korrekt und behandelten die unerwünschten LOR-Schlüssel korrekt.
- **Prompt 3 (Chain of Thought):** Vier von fünf Ausführungen waren erfolgreich. Eine Ausführung scheiterte aufgrund eines Syntaxfehlers, bei dem ein Python-Schlüsselwort („if“ und „not“) ins Deutsche übersetzt wurde. Dies führte zu einem `SyntaxError` und einem Abbruch des Skripts.

### Performance und Ressourcenverbrauch:

- Die Laufzeit betrug bei allen erfolgreichen Ausführungen zwischen 0,51 und 0,59 Sekunden, was für die Berechnung der prozentualen Anteile sehr effizient ist.
- Der Speicherverbrauch bewegte sich bei rund 150–155 MB *Maximum Resident Set Size*, was für diesen Datensatz angemessen ist.
- Die CPU-Auslastung lag bei allen Ausführungen zwischen 424% und 479%, was auf eine effiziente Nutzung der verfügbaren Ressourcen hinweist.

### Codequalität und Wartbarkeit:

- **Struktur:** Die Struktur ist in allen drei Prompts gut unterteilt und übersichtlich. Die Skripte von Prompt 3 waren jedoch am strukturiertesten und enthielten eine etwas genauere Unterteilung der Schritte.
- **Dokumentation:** Alle Skripte sind gut kommentiert und erklärten die einzelnen Schritte.
- **Erweiterbarkeit:** Alle Skripte sind leicht erweiterbar und können einfach angepasst werden, um zusätzliche Funktionalitäten hinzuzufügen. Es wurden teils auch Erweiterungsmöglichkeiten für die Skripte gegeben, wie zum Beispiel das generieren einer neuen Excel-Datei, sofern dieses nicht schon im Skript selbst enthalten war.

**Fazit zu Testfall 3:** Die Ergebnisse zeigen, dass die **Zero-Shot**- und **Instruction Prompting**-Strategien in allen Fällen erfolgreich waren, während die **Chain of Thought (CoT)**-Strategie aufgrund eines Syntaxfehlers in einer Ausführung scheiterte. Dennoch war die CoT-Strategie in den meisten Fällen zuverlässig und lieferte gut strukturierten und dokumentierten Code.

In Bezug auf **Performance** gab es keine Probleme, und alle Skripte waren effizient. Die Codequalität war bei den Skripten von Prompt 3 am höchsten, da sie gut dokumentiert und robust waren. Für produktive Einsätze ist daher die Verwendung von **Chain of**

**Thought**-Prompts zu empfehlen, um sicherzustellen, dass der generierte Code den Anforderungen entspricht und leicht erweitert werden kann.

#### 6.4 Testfall 4: Zeitreihe über die Jahre 2014–2023

Im vierten Testfall wurde ein Python-Skript generiert, das die prozentuale Veränderung der Straftaten in Berlin im Vergleich zum Vorjahr berechnet und als Zeitreihe ausgibt. Hierfür wurden drei verschiedene Prompting-Strategien (Zero-Shot, Instruction Prompting, Chain of Thought) verwendet, wobei jede Strategie fünfmal ausgeführt wurde. Die wichtigsten Beobachtungen sind:

##### Erfolgsquote (Pass@15):

- **Prompt 1 (Zero-Shot):** Zwei von fünf Ausführungen waren erfolgreich. Die erfolgreichen Ausführungen berechneten die prozentuale Veränderung korrekt. Drei Ausführungen scheiterten: zwei aufgrund von Problemen mit den Sheet-Namen und eine aufgrund eines Syntaxfehlers, bei dem ein Python-Schlüsselwort („not“) ins Deutsche übersetzt wurde.
- **Prompt 2 (Instruction Prompting):** Alle fünf Ausführungen waren erfolgreich. Die Skripte berechneten die prozentuale Veränderung korrekt und speicherten die Ergebnisse in einem DataFrame.
- **Prompt 3 (Chain of Thought):** Alle fünf Ausführungen waren erfolgreich. Eine Ausführung warf eine Warnung aufgrund einer veralteten Pandas-Funktionalität, lieferte aber das korrekte Ergebnis.

Die Gesamterfolgsquote beträgt **80%** (12 von 15 Ausführungen waren erfolgreich).

##### Erfolgsquoten der einzelnen Prompts:

- **Prompt 1 (Zero-Shot):** 40% erfolgreich, 60% nicht erfolgreich (Probleme mit Sheet-Namen und Syntaxfehler).
- **Prompt 2 (Instruction Prompting):** 100% erfolgreich.
- **Prompt 3 (Chain of Thought):** 100% erfolgreich (eine Ausführung mit Warnung).

##### Korrektheit und typische Fehlerquellen:

- **Prompt 1 (Zero-Shot):** Zwei Ausführungen waren erfolgreich. Drei Ausführungen scheiterten: zwei aufgrund von Problemen mit den Sheet-Namen (die Jahre konnten nicht korrekt extrahiert werden) und eine aufgrund eines Syntaxfehlers, bei dem ein Python-Schlüsselwort („not“) ins Deutsche übersetzt wurde.
- **Prompt 2 (Instruction Prompting):** Alle Ausführungen waren erfolgreich. Die Skripte berechneten die prozentuale Veränderung korrekt und behandelten die Daten korrekt.

- **Prompt 3 (Chain of Thought):** Alle Ausführungen waren erfolgreich. Eine Ausführung warf eine Warnung aufgrund einer veralteten Pandas-Funktionalität, lieferte aber das korrekte Ergebnis.

#### Performance und Ressourcenverbrauch:

- Die Laufzeit betrug bei allen erfolgreichen Ausführungen zwischen 2,85 und 3,47 Sekunden, was für die Berechnung der prozentualen Veränderung sehr effizient ist.
- Der Speicherverbrauch bewegte sich bei rund 150–160 MB *Maximum Resident Set Size*, was für diesen Datensatz angemessen ist.
- Die CPU-Auslastung lag bei allen Ausführungen zwischen 155% und 168%, was auf eine effiziente Nutzung der verfügbaren Ressourcen hinweist.

#### Codequalität und Wartbarkeit:

- **Struktur:** Die Struktur ist in allen drei Prompts gut unterteilt und übersichtlich. Die Skripte von Prompt 3 waren jedoch am strukturiertesten und enthielten eine etwas genauere Unterteilung der Schritte.
- **Dokumentation:** Alle Skripte sind gut kommentiert und erklärten die einzelnen Schritte.
- **Erweiterbarkeit:** Alle Skripte sind leicht erweiterbar und können einfach angepasst werden, um zusätzliche Funktionalitäten hinzuzufügen. Es wurden teils auch Erweiterungsmöglichkeiten für die Skripte gegeben, wie zum Beispiel das generieren einer neuen Excel-Datei, sofern dieses nicht schon im Skript selbst enthalten war.

**Fazit zu Testfall 4:** Die Ergebnisse zeigen, dass die **Instruction Prompting-** und **Chain of Thought (CoT)**-Strategien in allen Fällen erfolgreich waren, während die **Zero-Shot**-Strategie aufgrund von Problemen mit den Sheet-Namen und Syntaxfehlern in drei von fünf Ausführungen scheiterte. Dennoch war die CoT-Strategie in den meisten Fällen zuverlässig und lieferte gut strukturierten und dokumentierten Code.

In Bezug auf **Performance** gab es keine Probleme, und alle Skripte waren effizient. Die Codequalität war bei den Skripten von Prompt 3 am höchsten, da sie gut dokumentiert und robust waren. Für produktive Einsätze ist daher die Verwendung von **Chain of Thought**-Prompts zu empfehlen, um sicherzustellen, dass der generierte Code den Anforderungen entspricht und leicht erweitert werden kann.

## 6.5 Übersicht der Ergebnisse

## 6.6 Vergleich manuell erstellter Code

In diesem Unterkapitel werden die Ergebnisse der manuell geschriebenen Skripte mit denen der durch Large Language Models (LLMs) generierten Skripte systematisch verglichen. Ziel ist

Tabelle 1: Übersicht der Pass@15-Ergebnisse pro Testfall und Prompting-Strategie (Es wird nur die Erfolgsquote der komplett korrekten Ergebnisse angegeben)

Testfall	Zero-Shot	Instruction	Chain of Thought
1	0%	0%	100%
2	0%	20%	80%
3	100%	100%	80%
4	40%	100%	100%

es, zu ermitteln, inwieweit die unterschiedlichen Herangehensweisen – manuelles Programmieren versus automatisierte Code-Generierung mittels Prompting – im Hinblick auf Korrektheit der Resultate, Performance, Code-Struktur und Wartbarkeit voneinander abweichen.

**Korrektheit der Resultate** Ein zentrales Kriterium bei der Bewertung ist, ob beide Ansätze zu identischen oder zumindest äquivalenten Analyseergebnissen gelangen. In allen Testfällen (Sortierung und Ausgabe, Join von Tabellen, Berechnung prozentualer Verteilung, Zeitreihenanalyse) liefern sowohl die manuell geschriebenen als auch die generierten Skripte inhaltlich sehr ähnliche Endergebnisse. Kleinere Abweichungen treten in Einzelfällen auf – etwa in der Handhabung spezieller Zeilen (bspw. mit LOR-Schlüsseln) oder bei der Formatierung der Ausgabe. Insgesamt schneiden jedoch beide Varianten gut ab und liefern für die untersuchten Datensätze deckungsgleiche Kernaussagen.

**Performance** Hinsichtlich der Laufzeiten und der Ressourcennutzung (z. B. CPU-Auslastung, Speicherbedarf) sind keine gravierenden Unterschiede zwischen den manuell erstellten Skripten und den generierten Skripten festzustellen. Beide Varianten führen die Analysen innerhalb weniger Sekunden aus und verbrauchen typischerweise zwischen 150 MB und 160 MB Maximum Resident Set Size. In diesem Anwendungsbereich – also bei den vorliegenden Datensätzen und Aufgaben – erweisen sich die Skripte beider Ansätze als annähernd gleichwertig in Bezug auf die Performance.

**Code-Struktur und Wartbarkeit** Aus der Perspektive von Codequalität und Wartbarkeit zeigen sich leichte Unterschiede. Die manuell geschriebenen Skripte sind in der Regel stärker auf die konkrete Aufgabenstellung zugeschnitten und enthalten genau die für die jeweilige Problemstellung nötige Logik. Bei den durch ein LLM generierten Skripten fällt hingegen auf, dass sie teils zusätzliche Codeabschnitte (z. B. zur Fehlerbehandlung) oder Kommentare bereitstellen, da das Modell versucht, einen möglichst robusten und verallgemeinerten Ansatz zu liefern. Dies kann den Quellcode einerseits etwas länger machen, andererseits erhöht es unter Umständen die Wiederverwendbarkeit und Anpassbarkeit. In Summe sind jedoch beide Varianten in klar nachvollziehbare Schritte gegliedert und somit gut wart- und erweiterbar.

**Einschätzung und Fazit** Der direkte Vergleich zeigt, dass LLMs bereits sehr fähig darin sind, für datenanalytische Aufgaben in Python korrekt funktionierenden Code zu erstellen. Manuell erstellte Skripte haben den Vorteil, dass sie oft kompakter und exakt auf die vorhandenen Daten abgestimmt sind. Im Sinne einer schnellen Prototypenerstellung und zur

Beschleunigung wiederkehrender Routinen bietet die automatisierte Code-Generierung durch LLMs jedoch eine effiziente Alternative – zumal sich der automatisch generierte Code in den Testfällen als kaum weniger performant und inhaltlich genauso zuverlässig erwiesen hat wie die manuell entwickelte Lösung.

## 7 Fazit und Ausblick

TODO:check Die vorliegende Arbeit hat untersucht, wie gut sich moderne *Large Language Models* (LLMs) – exemplarisch repräsentiert durch ChatGPT mit dem Modell „GPTo1“ – für die automatisierte Generierung von Python-Code im Kontext typischer Datenanalyseaufgaben eignen. Im Fokus standen vier typische Szenarien: das Sortieren und Filtern von Daten (Testfall1), das Zusammenführen und Aggregieren verschiedener Tabellen (Testfall2), die Berechnung prozentualer Anteile (Testfall3) sowie die Ermittlung von Zeitreihen mitsamt prozentualer Veränderung zum Vorjahr (Testfall4). Dabei wurden unterschiedliche Prompting-Strategien (Zero-Shot, Instruction Prompting und Chain of Thought) eingesetzt und die Qualität der resultierenden Skripte anhand von Kriterien wie Korrektheit, Performanz, Code-Struktur und Wartbarkeit evaluiert.

### Zentrale Erkenntnisse

- **Korrektheit und Erfolgsquote:**

Die Ergebnisse zeigen, dass die Leistungsfähigkeit stark von der Art des Promptings abhängt. Während *Chain of Thought*-Prompts im Durchschnitt sehr robuste und korrekte Lösungen lieferten, kam es bei Zero-Shot-Ansätzen häufiger zu Syntaxfehlern, falschen Sheet-Namen oder fehlerhafter Ausfilterung bestimmter Zeilen (LOR-Schlüssel). Insgesamt bestätigte sich, dass *strukturierte Prompt-Anweisungen* (Instruction oder Chain of Thought) zu einer höheren Erfolgsquote führen als ein unpräziser Zero-Shot-Ansatz.

- **Performanz:**

Bei allen Testfällen war die Ausführungszeit sehr kurz (teils nur wenige Zehntelsekunden bis wenige Sekunden), was für den überschaubaren Datensatz (mehrere Excel-Sheets, aber jeweils nur im Bereich einiger Zeilen pro Sheet) vollkommen ausreichend ist. Damit bestätigt sich, dass ein KI-basierter Code-Generator im Kontext kleiner bis mittlerer Datenanalysen in Sachen Laufzeit absolut konkurrenzfähig ist.

- **Qualität und Wartbarkeit:**

Die generierten Skripte weisen – insbesondere bei Chain of Thought – eine erstaunlich gute Code-Struktur auf. Sie sind in der Regel sinnvoll kommentiert, nutzen etablierte Python-Bibliotheken (z.,B. `pandas`) und sind leicht erweiterbar. Limitationen zeigen sich jedoch in Fällen, in denen das Modell falsche oder irreführende Annahmen über Spaltennamen oder Datenstrukturen trifft. Solche Fehler entstehen meist durch Unklarheiten oder fehlende Details in den Prompts.

- **Grenzen:**

Trotz der hohen Erfolgsquote in den definierten Testfällen darf man nicht vergessen, dass

in realen Szenarien Datenanalysen oft komplexere Anforderungen mit sich bringen: Ungewohnte Datenformate, aufwändige Vorverarbeitungs- und Qualitätschecks oder tiefergehende statistische Verfahren. Hier können LLMs ohne exakte Vorgaben schnell an ihre Grenzen gelangen. Die Modelle tendieren außerdem gelegentlich zu „Halluzinationen“, indem sie nicht-existente Funktionen oder Spalten vorschlagen.

**Ausblick** Die steigende Leistungsfähigkeit von LLMs lässt erwarten, dass die automatische Code-Generierung in den kommenden Jahren eine immer größere Rolle in Datenanalyse- und Data-Science-Projekten spielen wird. Insbesondere die folgenden Entwicklungen erscheinen relevant:

- **Spezialisierte Modelle und Fine-Tuning:**

Mit zunehmender Verfügbarkeit von domänenspezifischen Datensätzen für Code-Generierung könnten LLMs besser auf bestimmte Aufgaben, wie z.,B. Datenbereinigung oder automatisiertes Exploratory Data Analysis (EDA), optimiert werden. Eine engere Anbindung an unternehmenseigene Datenbanken ist ebenfalls denkbar.

- **Bessere Kontextverarbeitung:**

Aktuelle LLMs haben eine begrenzte Kontextlänge. Künftige Modelle werden voraussichtlich größere Kontextfenster bieten und so umfangreichere Code-Dateien oder Datenschemata verarbeiten können. Damit würden komplexere Use Cases (etwa umfangreiche ETL-Pipelines) in den Bereich des Möglichen rücken.

- **Integrierte Fehlersuche und Debugging:**

Erste Ansätze zeigen, dass LLMs nicht nur Code generieren, sondern auch zum Debugging eingesetzt werden können. Wenn die Modelle direkt während der Code-Erstellung mögliche Probleme erkennen und Lösungsvorschläge machen, könnte die Produktivität bei Datenanalyse-Projekten weiter gesteigert werden.

- **Erweiterte Qualitäts- und Sicherheitstests:**

Wie in aktuellen Studien – beispielsweise von Liu et al. (2023) – diskutiert, bedarf es verbesserter Evaluationsmethoden (etwa *EvalPlus*), um die tatsächliche Funktionalität und Sicherheit des generierten Codes sicherzustellen. Automatisierte Unit-Tests und Code-Audits im Prompt bzw. Post-Processing könnten ein wesentlicher Baustein sein.

Zusammenfassend lässt sich festhalten, dass LLMs bereits heute in der Lage sind, für standardisierte Datenanalyseschritte in Python zuverlässig einsatzfähigen Code zu erstellen. Dies kann den Einstieg in die Datenanalyse vereinfachen und auch erfahrenen Anwendern mühsame Routineaufgaben abnehmen. Allerdings bleibt ein gewisses Maß an menschlicher Kontrolle weiterhin unverzichtbar: Sei es zur Prüfung potenzieller Halluzinationen, zur Anpassung spezieller Projektanforderungen oder zur Qualitätssicherung komplexer Analysen. Das Potenzial für zukünftige Anwendungen ist groß – insbesondere, wenn die Modelle durch spezialisierte Trainingsdaten, fortgeschrittene Prompting-Techniken und integrierte Debugging-Funktionen weiter verbessert werden.



## 8 Anhang

### 8.1 Literaturverzeichnis

#### Literatur

- [1] Jiawei Liu u. a. „Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation“. In: *Advances in Neural Information Processing Systems*. Hrsg. von A. Oh u. a. Bd. 36. Curran Associates, Inc., 2023, S. 21558–21572. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf).
- [2] Mark Chen u. a. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: <https://arxiv.org/abs/2107.03374>.
- [3] Jiawei Liu u. a. „Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation“. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: <https://openreview.net/forum?id=1qv610Cu7>.
- [4] Wayne Xin Zhao u. a. *A Survey of Large Language Models*. 2024. arXiv: 2303.18223 [cs.CL]. URL: <https://arxiv.org/abs/2303.18223>.
- [5] Juyong Jiang u. a. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL]. URL: <https://arxiv.org/abs/2406.00515>.
- [6] Shihan Dou u. a. *What’s Wrong with Your Code Generated by Large Language Models? An Extensive Study*. 2024. arXiv: 2407.06153 [cs.SE]. URL: <https://arxiv.org/abs/2407.06153>.
- [7] Dong Huang u. a. *EffiBench: Benchmarking the Efficiency of Automatically Generated Code*. 2024. arXiv: 2402.02037 [cs.SE]. URL: <https://arxiv.org/abs/2402.02037>.
- [8] Pengfei Liu u. a. *Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing*. 2021. arXiv: 2107.13586 [cs.CL]. URL: <https://arxiv.org/abs/2107.13586>.
- [9] Tom B. Brown u. a. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: <https://arxiv.org/abs/2005.14165>.
- [10] Long Ouyang u. a. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: <https://arxiv.org/abs/2203.02155>.
- [11] Jason Wei u. a. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: <https://arxiv.org/abs/2201.11903>.
- [12] Baptiste Rozière u. a. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: <https://arxiv.org/abs/2308.12950>.
- [13] Raymond Li u. a. *StarCoder: may the source be with you!* 2023. arXiv: 2305.06161 [cs.CL]. URL: <https://arxiv.org/abs/2305.06161>.

- [14] Yue Wang u. a. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859 [cs.CL]. URL: <https://arxiv.org/abs/2109.00859>.
- [15] Erik Nijkamp u. a. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG]. URL: <https://arxiv.org/abs/2203.13474>.
- [16] Rudra Dhar, Karthik Vaidhyanathan und Vasudeva Varma. *Can LLMs Generate Architectural Design Decisions? -An Exploratory Empirical study*. 2024. arXiv: 2403.01709 [cs.SE]. URL: <https://arxiv.org/abs/2403.01709>.
- [17] Jiawei Liu u. a. „Evaluating Language Models for Efficient Code Generation“. In: *First Conference on Language Modeling*. 2024. URL: <https://openreview.net/forum?id=IBCBMeAhmC>.
- [18] Polizei Berlin. *Kriminalitätsatlas Berlin*. 2024. URL: <https://daten.berlin.de/datensaetze/kriminalitaetsatlas-berlin>.

## 8.2 Quellcodeverzeichnis

### 8.2.1 Testfall 1

```

1 from openai import OpenAI
2 import os
3 import pandas as pd
4
5 path=os.getenv("THESIS_PATH")
6
7 # OpenAI API Key
8 api_key = os.getenv("OPENAI_API_KEY")
9 client = OpenAI(api_key=api_key)
10
11 # Excel-Datei einlesen und als Dataframe speichern
12 file = path+"/testcases/Fallzahlen.xlsx"
13 sheet_name = "Fallzahlen_2023"
14 df = pd.read_excel(file, sheet_name=sheet_name)
15
16 # Prompt erstellen: CSV-Daten einbetten + Anweisung
17 prompt = f"""
18 Ich habe eine Excel Datei mit dem Namen 'Fallzahlen.xlsx'. Hier ist der
19 Inhalt des Sheets 'Fallzahlen_2023': {df}. Erstelle mir ein Skript in
20 Python, das die Daten aus der Excel-Datei einliest, nach den Straftaten
21 insgesamt der Bezirke sortiert und in einem Dataframe speichert.
22 """
23
24 # Anfrage an die OpenAI Chat-API
25 response = client.chat.completions.create(model="o1-mini",
26 messages=[
27 {"role": "user", "content": prompt}
28 ],)

```

```

27 response_file = path+"testcases/testcase1/prompt1/exec5/response.txt"
28 os.makedirs(os.path.dirname(response_file), exist_ok=True)
29 # Ausgabe des generierten Skripts
30 with open(response_file, "w") as file:
31     file.write(response.choices[0].message.content)
32
33 print(f"Response wurde in {response_file} gespeichert.")

```

Listing 1: OpenAI API Request Skript

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 datei_pfad = 'Fallzahlen.xlsx'
5
6 try:
7     # Einlesen des spezifischen Sheets 'Fallzahlen_2023'
8     df = pd.read_excel(datei_pfad, sheet_name='Fallzahlen_2023')
9
10    # Anzeigen der ersten paar Zeilen zur Überprüfung
11    print("Original DataFrame:")
12    print(df.head())
13
14    # Sortieren nach der Spalte 'Straftaten_insgesamt' in absteigender
    Reihenfolge
15    df_sortiert = df.sort_values(by='Straftaten_insgesamt', ascending=False)
16
17    # Zurücksetzen des Index (optional)
18    df_sortiert.reset_index(drop=True, inplace=True)
19
20    # Anzeige der sortierten DataFrame
21    print("\nSortierter DataFrame nach 'Straftaten_insgesamt':")
22    print(df_sortiert)
23
24    # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
25    sortierte_datei_pfad = 'Fallzahlen_sortiert.xlsx'
26    df_sortiert.to_excel(sortierte_datei_pfad, index=False)
27    print(f"\nDer sortierte DataFrame wurde in '{sortierte_datei_pfad}'
    gespeichert.")
28
29 except FileNotFoundError:
30     print(f"Die Datei '{datei_pfad}' wurde nicht gefunden.")
31 except ValueError as ve:
32     print(f"Fehler beim Einlesen des Sheets: {ve}")
33 except Exception as e:
34     print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")

```

Listing 2: Skript Testfall 1 Prompt 1 Ausführung 1

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'

```

```

5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Einlesen der Excel-Datei
10 df = pd.read_excel(excel_datei, sheet_name=sheet_name)
11
12 # Anzeigen der ersten paar Zeilen (optional)
13 print("Unsortierte Daten:")
14 print(df.head())
15
16 # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
17 df_sortiert = df.sort_values(by='Straftaten_insgesamt', ascending=False)
18
19 # Anzeigen der sortierten Daten (optional)
20 print("\nSortierte Daten nach 'Straftaten_insgesamt':")
21 print(df_sortiert.head())
22
23 # Optional: Speichern des sortierten DataFrames in einer neuen Excel-Datei
24 df_sortiert.to_excel('Fallzahlen_sortiert.xlsx', index=False)
25
26 print("\nDie sortierten Daten wurden in 'Fallzahlen_sortiert.xlsx'
    gespeichert.")

```

Listing 3: Skript Testfall 1 Prompt 1 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 try:
10     # Einlesen der Excel-Datei
11     df = pd.read_excel(excel_datei, sheet_name=sheet_name)
12     print("Excel-Datei erfolgreich eingelesen.")
13
14     # Überprüfen, ob die erforderte Spalte existiert
15     if 'Straftaten_insgesamt' not in df.columns:
16         raise ValueError("Die Spalte 'Straftaten_insgesamt' wurde im
            DataFrame nicht gefunden.")
17
18     # Sortieren des DataFrames nach 'Straftaten_insgesamt' in absteigender
            Reihenfolge
19     df_sorted = df.sort_values(by='Straftaten_insgesamt', ascending=False).
        reset_index(drop=True)
20     print("DataFrame erfolgreich nach 'Straftaten_insgesamt' sortiert.")
21
22     # Optional: Anzeige der sortierten Daten
23     print(df_sorted)
24

```

```

25     # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
26     # df_sorted.to_excel('Fallzahlen_sorted.xlsx', index=False)
27     # print("Sortierte Daten wurden in 'Fallzahlen_sorted.xlsx' gespeichert
    .")
28
29 except FileNotFoundError:
30     print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
31 except ValueError as ve:
32     print(f"Fehler: {ve}")
33 except Exception as e:
34     print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")

```

Listing 4: Skript Testfall 1 Prompt 1 Ausführung 3

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Excel-Datei einlesen
10 df = pd.read_excel(excel_datei, sheet_name=sheet_name)
11
12 # Daten nach 'Straftaten_insgesamt' in absteigender Reihenfolge sortieren
13 df_sortiert = df.sort_values(by='Straftaten_insgesamt', ascending=False)
14
15 # Optional: Den sortierten DataFrame anzeigen
16 print(df_sortiert)
17
18 # Optional: Den sortierten DataFrame in eine neue Excel-Datei speichern
19 df_sortiert.to_excel('Fallzahlen_sortiert.xlsx', index=False)

```

Listing 5: Skript Testfall 1 Prompt 1 Ausführung 4

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 dateipfad = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Einlesen der Excel-Datei
10 try:
11     df = pd.read_excel(dateipfad, sheet_name=sheet_name)
12     print("Daten erfolgreich eingelesen.")
13 except FileNotFoundError:
14     print(f"Die Datei {dateipfad} wurde nicht gefunden.")
15     exit()
16 except ValueError:
17     print(f"Das Sheet '{sheet_name}' existiert nicht in der Datei.")

```

```

18     exit()
19
20 # Überprüfen der Spaltennamen (optional)
21 print("Verfügbare Spalten:")
22 print(df.columns.tolist())
23
24 # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
25 # Falls aufsteigend gewünscht ist, setze ascending=True
26 try:
27     df_sortiert = df.sort_values(by='Straftaten_insgesamt', ascending=False)
28     print("Daten erfolgreich sortiert.")
29 except KeyError:
30     print("Die Spalte 'Straftaten_insgesamt' wurde nicht gefunden.")
31     exit()
32
33 # Zurücksetzen des Indexes (optional)
34 df_sortiert.reset_index(drop=True, inplace=True)
35
36 # Anzeigen des sortierten DataFrames
37 print("Sortiertes DataFrame:")
38 print(df_sortiert)

```

Listing 6: Skript Testfall 1 Prompt 1 Ausführung 5

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # LOR-Schlüssel, die ausgeschlossen werden sollen
10 ausschluss_schluessel = [999900, 999999]
11
12 try:
13     # Excel-Datei einlesen
14     df = pd.read_excel(excel_datei, sheet_name=sheet_name)
15
16     # Anzeige der ersten paar Zeilen zur Überprüfung (optional)
17     print("Originaldaten:")
18     print(df.head())
19
20     # Filter anwenden, um die auszuschließenden LOR-Schlüssel zu entfernen
21     df_filtered = df[~df['LOR-Schlüssel'].isin(ausschluss_schluessel)]
22
23     # Anzeige der gefilterten Daten zur Überprüfung (optional)
24     print("\nGefilterte Daten:")
25     print(df_filtered.head())
26
27     # Sortieren nach 'Straftaten_insgesamt' absteigend (größte zuerst)
28     df_sorted = df_filtered.sort_values(by='Straftaten_insgesamt', ascending=
False)

```

```

29
30     # Ergebnis anzeigen (optional)
31     print("\nSortierte Daten:")
32     print(df_sorted)
33
34     # Optional: Das sortierte DataFrame speichern
35     # df_sorted.to_excel('Fallzahlen_sortiert.xlsx', index=False)
36
37 except FileNotFoundError:
38     print(f"Die Datei {excel_datei} wurde nicht gefunden.")
39 except Exception as e:
40     print(f"Ein Fehler ist aufgetreten: {e}")

```

Listing 7: Skript Testfall 1 Prompt 2 Ausführung 1

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # LOR-Schlüssel, die ausgeschlossen werden sollen
10 ausgeschlossene_lor_schluesel = [999900, 999999]
11
12 # Einlesen der Excel-Datei
13 try:
14     df = pd.read_excel(excel_datei, sheet_name=sheet_name, engine='openpyxl')
15     print("Excel-Datei erfolgreich eingelesen.")
16 except FileNotFoundError:
17     print(f"Die Datei {excel_datei} wurde nicht gefunden.")
18     exit(1)
19 except Exception as e:
20     print(f"Ein Fehler ist aufgetreten beim Einlesen der Excel-Datei: {e}")
21     exit(1)
22
23 # Filtern der Zeilen, die nicht zu Bezirken gehören
24 df_filtered = df[~df['LOR-Schlüssel'].isin(ausgeschlossene_lor_schluesel)]
25
26 # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
27 df_sorted = df_filtered.sort_values(by='Straftaten_insgesamt', ascending=
    False).reset_index(drop=True)
28
29 # Anzeige des sortierten DataFrames
30 print("Sortierte Daten nach 'Straftaten_insgesamt':")
31 print(df_sorted)
32
33 # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
34 # df_sorted.to_excel('Fallzahlen_sortiert.xlsx', index=False)

```

Listing 8: Skript Testfall 1 Prompt 2 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 dateipfad = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Einlesen der Excel-Datei
10 df = pd.read_excel(dateipfad, sheet_name=sheet_name)
11
12 # Anzeigen der ersten Zeilen zur Überprüfung (optional)
13 #print(df.head())
14
15 # Filtere die Bezirke heraus, indem die unerwünschten LOR-Schlüssel
    ausgeschlossen werden
16 ausgeschlossene_lor = [999900, 999999]
17 bezirke_df = df[~df['LOR-Schlüssel'].isin(ausgeschlossene_lor)]
18
19 # Sortiere die Bezirke nach 'Straftaten_insgesamt' in absteigender
    Reihenfolge
20 bezirke_sortiert = bezirke_df.sort_values(by='Straftaten_insgesamt',
    ascending=False)
21
22 # Optional: Zurücksetzen des Indexes nach dem Sortieren
23 bezirke_sortiert.reset_index(drop=True, inplace=True)
24
25 # Anzeigen des sortierten DataFrames (optional)
26 print(bezirke_sortiert)
27
28 # Wenn Sie den sortierten DataFrame weiter verwenden möchten, steht er als '
    bezirke_sortiert' zur Verfügung

```

Listing 9: Skript Testfall 1 Prompt 2 Ausführung 3

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Namen des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # LOR-Schlüssel der Zeilen, die ausgeschlossen werden sollen
10 ausschluss_lor_schluessel = [999900, 999999]
11
12 # Excel-Datei einlesen
13 df = pd.read_excel(excel_datei, sheet_name=sheet_name)
14
15 # Anzeigen der ursprünglichen Daten (optional)
16 print("Ursprüngliche Daten:")
17 print(df)

```



```

18
19 # Filtern der Bezirke, indem die ausgeschlossenen LOR-Schlüssel entfernt
    werden
20 df_gefiltert = df[~df['LOR-Schlüssel'].isin(ausschluss_lor_schluessel)]
21
22 # Anzeigen der gefilterten Daten (optional)
23 print("\nGefilterte Daten (ohne nicht zuzuordnende Bezirke):")
24 print(df_gefiltert)
25
26 # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
27 df_sortiert = df_gefiltert.sort_values(by='Straftaten_insgesamt', ascending=
    False)
28
29 # Zurücksetzen des Index (optional, für sauberere Anzeige)
30 df_sortiert.reset_index(drop=True, inplace=True)
31
32 # Anzeigen der sortierten Daten (optional)
33 print("\nSortierte Daten nach 'Straftaten_insgesamt':")
34 print(df_sortiert)
35
36 # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
37 # df_sortiert.to_excel('Sortierte_Fallzahlen.xlsx', index=False)

```

Listing 10: Skript Testfall 1 Prompt 2 Ausführung 4

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 datei_pfad = 'Fallzahlen.xlsx'
5
6 # Namen des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Einlesen der Excel-Datei
10 try:
11     df = pd.read_excel(datei_pfad, sheet_name=sheet_name)
12     print("Excel-Datei erfolgreich eingelesen.")
13 except FileNotFoundError:
14     print(f"Die Datei {datei_pfad} wurde nicht gefunden.")
15     exit(1)
16 except Exception as e:
17     print(f"Ein Fehler ist aufgetreten beim Einlesen der Datei: {e}")
18     exit(1)
19
20 # Anzeigen der ersten paar Zeilen, um sicherzustellen, dass die Daten korrekt
    eingelesen wurden
21 print("\nErste 5 Zeilen der Originaldaten:")
22 print(df.head())
23
24 # Filterkriterien: Ausschließen der LOR-Schlüssel 999900 und 999999
25 ausschluss_schluessel = [999900, 999999]
26 df_filtered = df[~df['LOR-Schlüssel'].isin(ausschluss_schluessel)]
27

```

```

28 print(f"\nAnzahl der Zeilen nach dem Ausschluss: {len(df_filtered)}")
29
30 # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
31 df_sorted = df_filtered.sort_values(by='Straftaten_insgesamt', ascending=
    False).reset_index(drop=True)
32
33 print("\nErste 5 Zeilen der sortierten Daten:")
34 print(df_sorted.head())
35
36 # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
37 ausgabe_pfad = 'Fallzahlen_sortiert.xlsx'
38 try:
39     df_sorted.to_excel(ausgabe_pfad, index=False)
40     print(f"\nSortierte Daten wurden erfolgreich in '{ausgabe_pfad}'
    gespeichert.")
41 except Exception as e:
42     print(f"Ein Fehler ist aufgetreten beim Speichern der Datei: {e}")

```

Listing 11: Skript Testfall 1 Prompt 2 Ausführung 5

```

1 import pandas as pd
2 import sys
3
4 def read_excel_file(file_path, sheet_name):
5     """
6     Liest eine Excel-Datei und gibt das entsprechende Sheet als DataFrame zur
7     ück.
8     """
9     try:
10         df = pd.read_excel(file_path, sheet_name=sheet_name)
11         print(f"Erfolgreich das Blatt '{sheet_name}' aus '{file_path}'
        gelesen.")
12         return df
13     except FileNotFoundError:
14         print(f"Fehler: Die Datei '{file_path}' wurde nicht gefunden.")
15         sys.exit(1)
16     except ValueError:
17         print(f"Fehler: Das Blatt '{sheet_name}' existiert nicht in '{
        file_path}'")
18         sys.exit(1)
19     except Exception as e:
20         print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")
21         sys.exit(1)
22
23 def validate_columns(df, required_columns):
24     """
25     Überprüft, ob die erforderlichen Spalten im DataFrame vorhanden sind.
26     """
27     missing_columns = [col for col in required_columns if col not in df.
        columns]
28     if missing_columns:
29         print(f"Fehler: Die folgenden erforderlichen Spalten fehlen im
        DataFrame: {missing_columns}")

```

```

29         sys.exit(1)
30     else:
31         print("Alle erforderlichen Spalten sind vorhanden.")
32
33 def sort_dataframe(df, sort_column, exclude_keys, key_column):
34     """
35     Sortiert den DataFrame nach 'sort_column' absteigend, wobei die Zeilen
36     mit 'exclude_keys'
37     am Ende verbleiben.
38     """
39     # Trennen der Zeilen, die ausgeschlossen werden sollen
40     df_excluded = df[df[key_column].isin(exclude_keys)]
41     df_to_sort = df[~df[key_column].isin(exclude_keys)]
42
43     # Sortieren der relevanten Zeilen
44     df_sorted = df_to_sort.sort_values(by=sort_column, ascending=False)
45
46     # Zusammenführen der sortierten und ausgeschlossenen Zeilen
47     df_final = pd.concat([df_sorted, df_excluded], ignore_index=True)
48
49     print("DataFrame erfolgreich sortiert.")
50     return df_final
51
52 def main():
53     # Parameter
54     file_path = 'Fallzahlen.xlsx'
55     sheet_name = 'Fallzahlen_2023'
56     sort_column = 'Straftaten_insgesamt'
57     key_column = 'LOR-Schlüssel'
58     exclude_keys = [9999900, 999999]
59     output_file = 'Fallzahlen_sortiert.xlsx'
60
61     # Schritt 1: Daten einlesen
62     df = read_excel_file(file_path, sheet_name)
63
64     # Schritt 2: Validierung der erforderlichen Spalten
65     required_columns = [sort_column, key_column]
66     validate_columns(df, required_columns)
67
68     # Schritt 3: Sortierung durchführen
69     df_sorted = sort_dataframe(df, sort_column, exclude_keys, key_column)
70
71     # Optional: Weitere Überprüfung, ob alle Zeilen enthalten sind
72     if len(df_sorted) != len(df):
73         print("Warnung: Die Anzahl der Zeilen nach der Sortierung stimmt
74         nicht mit der ursprünglichen Anzahl überein.")
75
76     # Schritt 4: Ergebnis speichern
77     try:
78         df_sorted.to_excel(output_file, index=False)
79         print(f"Das sortierte DataFrame wurde erfolgreich in '{output_file}'
80         gespeichert.")

```

```

78     except Exception as e:
79         print(f"Fehler beim Speichern der Datei: {e}")
80         sys.exit(1)
81
82 if __name__ == "__main__":
83     main()

```

Listing 12: Skript Testfall 1 Prompt 3 Ausführung 1

```

1 import pandas as pd
2
3 def sort_fallzahlen_excel(file_path: str, sheet_name: str) -> pd.DataFrame:
4     """
5     Liest die Excel-Datei, sortiert die Daten nach 'Straftaten_insgesamt'
6     absteigend,
7     wobei die Zeilen mit den LOR-Schlüsseln 999900 und 999999 am Ende bleiben
8     .
9
10    Parameters:
11    - file_path: Pfad zur Excel-Datei.
12    - sheet_name: Name des Sheets in der Excel-Datei.
13
14    Returns:
15    - Sortierter Pandas DataFrame.
16    """
17    try:
18        # Schritt 1: Einlesen der Excel-Datei
19        df = pd.read_excel(file_path, sheet_name=sheet_name)
20        print(f"Excel-Datei '{file_path}' erfolgreich eingelesen.")
21
22        # Überprüfen, ob die notwendigen Spalten vorhanden sind
23        benötigte_spalten = ['LOR-Schlüssel', 'Straftaten_insgesamt']
24        fehlende_spalten = [spalte for spalte in benötigte_spalten if spalte
25                             not in df.columns]
26        if fehlende_spalten:
27            raise ValueError(f"Fehlende Spalten in der Excel-Datei: {
28                             fehlende_spalten}")
29
30        # Schritt 2: Trennen der Bezirke und Nicht-Bezirke
31        nicht_bezirke_keys = [999900, 999999]
32
33        # Sicherstellen, dass 'LOR-Schlüssel' als numerisch betrachtet wird
34        df['LOR-Schlüssel'] = pd.to_numeric(df['LOR-Schlüssel'], errors='
35        coerce')
36
37        bezirke_df = df[~df['LOR-Schlüssel'].isin(nicht_bezirke_keys)].copy()
38        nicht_bezirke_df = df[df['LOR-Schlüssel'].isin(nicht_bezirke_keys)].
39        copy()
40
41        # Schritt 3: Sortieren der Bezirke nach 'Straftaten_insgesamt'
42        absteigend
43        bezirke_df.sort_values(by='Straftaten_insgesamt', ascending=False,
44                               inplace=True)

```

```

37
38     # Schritt 4: Zusammenfügen der sortierten Bezirke mit den Nicht-
        Bezirken
39     sortiertes_df = pd.concat([bezirke_df, nicht_bezirke_df],
        ignore_index=True)
40
41     print("Daten erfolgreich sortiert.")
42     return sortiertes_df
43
44     except FileNotFoundError:
45         print(f"Datei '{file_path}' wurde nicht gefunden.")
46     except ValueError as ve:
47         print(f"Wertfehler: {ve}")
48     except Exception as e:
49         print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")
50
51 # Anwendung des Skripts
52 if __name__ == "__main__":
53     datei_pfad = 'Fallzahlen.xlsx'
54     sheet = 'Fallzahlen_2023'
55
56     sortiertes_dataframe = sort_fallzahlen_excel(datei_pfad, sheet)
57
58     if sortiertes_dataframe is not None:
59         # Schritt 5: Optional - Speichern des sortierten DataFrames in einer
        neuen Excel-Datei
60         ausgabe_pfad = 'Sortierte_Fallzahlen.xlsx'
61         try:
62             sortiertes_dataframe.to_excel(ausgabe_pfad, index=False)
63             print(f"Sortiertes DataFrame wurde erfolgreich in '{ausgabe_pfad
        }' gespeichert.")
64         except Exception as e:
65             print(f"Fehler beim Speichern der Datei: {e}")

```

Listing 13: Skript Testfall 1 Prompt 3 Ausführung 2

```

1 import pandas as pd
2 import sys
3
4 def main():
5     excel_file = 'Fallzahlen.xlsx'
6     sheet_name = 'Fallzahlen_2023'
7     lor_keys_to_exclude = [999900, 999999]
8
9     try:
10         # Schritt 1: Einlesen der Excel-Datei
11         df = pd.read_excel(excel_file, sheet_name=sheet_name)
12         print("Excel-Datei erfolgreich eingelesen.")
13     except FileNotFoundError:
14         print(f"Fehler: Die Datei '{excel_file}' wurde nicht gefunden.")
15         sys.exit(1)
16     except ValueError as e:
17         print(f"Fehler beim Einlesen des Sheets '{sheet_name}': {e}")

```

```

18     sys.exit(1)
19 except Exception as e:
20     print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")
21     sys.exit(1)
22
23 # Überprüfen, ob die notwendigen Spalten vorhanden sind
24 required_columns = ['LOR-Schlüssel', 'Straftaten_insgesamt']
25 missing_columns = [col for col in required_columns if col not in df.
26 columns]
27 if missing_columns:
28     print(f"Fehler: Fehlende Spalten in den Daten: {missing_columns}")
29     sys.exit(1)
30
31 # Sicherstellen, dass 'LOR-Schlüssel' numerisch ist
32 try:
33     df['LOR-Schlüssel'] = pd.to_numeric(df['LOR-Schlüssel'])
34 except ValueError:
35     print("Fehler: Die Spalte 'LOR-Schlüssel' enthält nicht-numerische
36 Werte.")
37     sys.exit(1)
38
39 # Schritt 2: Trennen der Bezirke und der auszuschließenden Schlüssel
40 df_bezirke = df[~df['LOR-Schlüssel'].isin(lor_keys_to_exclude)]
41 df_excluded = df[df['LOR-Schlüssel'].isin(lor_keys_to_exclude)]
42
43 # Schritt 3: Sortieren der Bezirke nach 'Straftaten_insgesamt' absteigend
44 if 'Straftaten_insgesamt' not in df_bezirke.columns:
45     print("Fehler: Die Spalte 'Straftaten_insgesamt' ist nicht in den
46 Daten vorhanden.")
47     sys.exit(1)
48
49 try:
50     df_bezirke_sorted = df_bezirke.sort_values(by='Straftaten_insgesamt',
51 ascending=False)
52 except Exception as e:
53     print(f"Fehler beim Sortieren der Daten: {e}")
54     sys.exit(1)
55
56 # Schritt 4: Zusammenführen der sortierten Bezirke mit den
57 ausgeschlossenen Zeilen
58 df_final = pd.concat([df_bezirke_sorted, df_excluded], ignore_index=True)
59
60 # Optional: Ausgabe oder Speicherung des finalen DataFrames
61 # Beispiel: Anzeigen der ersten 5 Zeilen
62 print("Sortiertes DataFrame:")
63 print(df_final.head())
64
65 # Beispiel: Speichern in eine neue Excel-Datei
66 output_file = 'Fallzahlen_sortiert.xlsx'
67 try:
68     df_final.to_excel(output_file, index=False)
69     print(f"Das sortierte DataFrame wurde erfolgreich in '{output_file}'

```

```

    gespeichert.")
65     except Exception as e:
66         print(f"Fehler beim Speichern der Datei '{output_file}': {e}")
67         sys.exit(1)
68
69 if __name__ == "__main__":
70     main()

```

Listing 14: Skript Testfall 1 Prompt 3 Ausführung 3

```

1  import pandas as pd
2
3  def sort_fallzahlen_excel(file_path: str, sheet_name: str) -> pd.DataFrame:
4      """
5      Reads the specified sheet from an Excel file, sorts the data by '
6      Straftaten_insgesamt'
7      in descending order excluding rows with LOR-Schlüssel 999900 and 999999,
8      which are
9      appended at the end.
10
11     Parameters:
12     - file_path: Path to the Excel file.
13     - sheet_name: Name of the sheet to read.
14
15     Returns:
16     - A sorted pandas DataFrame.
17     """
18     try:
19         # Read the Excel sheet
20         df = pd.read_excel(file_path, sheet_name=sheet_name)
21         print(f"Successfully read the sheet '{sheet_name}' from '{file_path}'")
22     except FileNotFoundError:
23         print(f"Error: The file '{file_path}' was not found.")
24         return pd.DataFrame()
25     except ValueError:
26         print(f"Error: The sheet '{sheet_name}' does not exist in '{file_path}'")
27         return pd.DataFrame()
28     except Exception as e:
29         print(f"An unexpected error occurred while reading the Excel file: {e}")
30         return pd.DataFrame()
31
32     # Check if necessary columns exist
33     required_columns = ['LOR-Schlüssel', 'Straftaten_insgesamt']
34     missing_columns = [col for col in required_columns if col not in df.columns]
35     if missing_columns:
36         print(f"Error: The following required columns are missing from the data: {missing_columns}")
37         return pd.DataFrame()

```

```

37 # Separate the rows with LOR-Schlüssel 999900 and 999999
38 exclusion_keys = [999900, 999999]
39 excluded_df = df[df['LOR-Schlüssel'].isin(exclusion_keys)]
40 included_df = df[~df['LOR-Schlüssel'].isin(exclusion_keys)]
41
42 if excluded_df.empty:
43     print("Warning: No rows found with LOR-Schlüssel 999900 or 999999.")
44
45 # Sort the included DataFrame by 'Straftaten_insgesamt' in descending
46 # order
47 try:
48     included_sorted_df = included_df.sort_values(by='Straftaten_insgesamt',
49     ascending=False)
50     print("Successfully sorted the included data by 'Straftaten_insgesamt'
51     in descending order.")
52 except KeyError:
53     print("Error: The column 'Straftaten_insgesamt' does not exist in the
54     data.")
55     return pd.DataFrame()
56 except Exception as e:
57     print(f"An unexpected error occurred during sorting: {e}")
58     return pd.DataFrame()
59
60 # Concatenate the sorted included data with the excluded data
61 sorted_df = pd.concat([included_sorted_df, excluded_df], ignore_index=
62 True)
63 print("Successfully concatenated the excluded rows at the end of the
64 sorted data.")
65
66 return sorted_df
67
68 if __name__ == "__main__":
69     # Define the Excel file path and sheet name
70     excel_file = 'Fallzahlen.xlsx'
71     sheet = 'Fallzahlen_2023'
72
73     # Get the sorted DataFrame
74     sorted_fallzahlen_df = sort_fallzahlen_excel(excel_file, sheet)
75
76     # Check if the DataFrame is not empty before proceeding
77     if not sorted_fallzahlen_df.empty:
78         print("Sorted DataFrame:")
79         print(sorted_fallzahlen_df)
80         # Optionally, you can save the sorted DataFrame to a new Excel file
81         # or another format
82         # sorted_fallzahlen_df.to_excel('Sorted_Fallzahlen.xlsx', index=False
83         # )
84     else:
85         print("No data to display due to previous errors.")

```

Listing 15: Skript Testfall 1 Prompt 3 Ausführung 4

```

1 import pandas as pd

```



```

2
3 def sort_fallzahlen_excel(file_path: str, sheet_name: str) -> pd.DataFrame:
4     """
5     Liest die Daten aus dem angegebenen Excel-Sheet, sortiert sie nach '
6     Straftaten_insgesamt' absteigend,
7     wobei die Zeilen mit den LOR-Schlüsseln 999900 und 999999 am Ende des
8     DataFrames stehen.
9
10    Parameters:
11    - file_path: Pfad zur Excel-Datei.
12    - sheet_name: Name des Sheets, das eingelesen werden soll.
13
14    Returns:
15    - Ein sortierter Pandas DataFrame.
16    """
17    try:
18        # Schritt 1: Einlesen der Excel-Datei
19        df = pd.read_excel(file_path, sheet_name=sheet_name)
20        print(f"Erfolgreich das Sheet '{sheet_name}' aus '{file_path}'
21        eingelesen.")
22    except FileNotFoundError:
23        raise FileNotFoundError(f"Die Datei '{file_path}' wurde nicht
24        gefunden.")
25    except ValueError:
26        raise ValueError(f"Das Sheet '{sheet_name}' existiert nicht in '{
27        file_path}'")
28    except Exception as e:
29        raise Exception(f"Ein unerwarteter Fehler ist beim Einlesen der Excel
30        -Datei aufgetreten: {e}")
31
32    # Überprüfen, ob die notwendigen Spalten vorhanden sind
33    required_columns = ['LOR-Schlüssel', 'Straftaten_insgesamt']
34    missing_columns = [col for col in required_columns if col not in df.
35    columns]
36    if missing_columns:
37        raise KeyError(f"Die folgenden erforderlichen Spalten fehlen im
38        DataFrame: {missing_columns}")
39    else:
40        print("Alle erforderlichen Spalten sind vorhanden.")
41
42    # Schritt 2: Separiere die Bezirke und die nicht zuzuordnenden Einträge
43    # Definiere die LOR-Schlüssel, die nicht sortiert werden sollen
44    exclude_lor_keys = [999900, 999999]
45
46    # Konvertiere 'LOR-Schlüssel' zu integer, falls nötig
47    if df['LOR-Schlüssel'].dtype != 'int64' and df['LOR-Schlüssel'].dtype !=
48    'int32':
49        try:
50            df['LOR-Schlüssel'] = df['LOR-Schlüssel'].astype(int)
51            print("'LOR-Schlüssel' erfolgreich in Integer konvertiert.")
52        except ValueError:
53            raise ValueError("'LOR-Schlüssel' enthält nicht-numerische Werte,

```

```

    die nicht in Integer konvertiert werden können.")
45
46 # Teile den DataFrame in Bezirke und Nicht-Bezirke
47 bezirke_df = df[~df['LOR-Schlüssel'].isin(exclude_lor_keys)]
48 nicht_bezirke_df = df[df['LOR-Schlüssel'].isin(exclude_lor_keys)]
49
50 print(f"Anzahl der Bezirke: {bezirke_df.shape[0]}")
51 print(f"Anzahl der nicht zuzuordnenden Einträge: {nicht_bezirke_df.shape
[0]}")
52
53 # Schritt 3: Sortiere die Bezirke nach 'Straftaten_insgesamt' absteigend
54 try:
55     bezirke_sorted = bezirke_df.sort_values(by='Straftaten_insgesamt',
ascending=False)
56     print("Bezirke erfolgreich nach 'Straftaten_insgesamt' absteigend
sortiert.")
57 except KeyError:
58     raise KeyError("Die Spalte 'Straftaten_insgesamt' wurde nicht
gefunden und kann nicht zum Sortieren verwendet werden.")
59 except Exception as e:
60     raise Exception(f"Ein unerwarteter Fehler ist beim Sortieren der
Bezirke aufgetreten: {e}")
61
62 # Schritt 4: Konkateneriere die sortierten Bezirke mit den nicht
zuzuordnenden Einträgen
63 final_df = pd.concat([bezirke_sorted, nicht_bezirke_df], ignore_index=
True)
64 print("Bezirke und nicht zuzuordnende Einträge erfolgreich kombiniert.")
65
66 return final_df
67
68 if __name__ == "__main__":
69     # Definiere den Pfad zur Excel-Datei und den Sheet-Namen
70     excel_file = 'Fallzahlen.xlsx'
71     sheet = 'Fallzahlen_2023'
72
73     try:
74         # Führe die Sortierung durch und hole das sortierte DataFrame
75         sortiertes_df = sort_fallzahlen_excel(excel_file, sheet)
76         print("Daten erfolgreich sortiert.")
77
78         # Optional: Anzeige der ersten paar Zeilen des sortierten DataFrames
79         print(sortiertes_df.head())
80
81         # Optional: Speichern des sortierten DataFrames in eine neue Excel-
Datei
82         # sortiertes_df.to_excel('Fallzahlen_sortiert.xlsx', index=False)
83         # print("Sortiertes DataFrame wurde in 'Fallzahlen_sortiert.xlsx'
gespeichert.")
84
85     except Exception as e:
86         print(f"Fehler: {e}")

```

## 8.2.2 Testfall 2

```

1 import pandas as pd
2
3 def lese_und_sortiere_fallzahlen(excel_datei):
4     """
5     Liest alle Sheets aus der angegebenen Excel-Datei, kombiniert die Daten,
6     sortiert sie nach 'Straftaten_insgesamt' in absteigender Reihenfolge
7     und gibt den sortierten DataFrame zurück.
8
9     :param excel_datei: Pfad zur Excel-Datei (z.B. 'Fallzahlen.xlsx')
10    :return: Sortierter Pandas DataFrame
11    """
12    try:
13        # Alle Sheets der Excel-Datei lesen. sheet_name=None liest alle
14        # Sheets als Dict.
15        alle_sheets = pd.read_excel(excel_datei, sheet_name=None)
16
17        # Liste zur Speicherung aller DataFrames
18        dataframe_liste = []
19
20        # Durch alle Sheets iterieren und die DataFrames zur Liste hinzufügen
21        for sheet_name, df in alle_sheets.items():
22            # Optional: Hinzufügen einer Spalte mit dem Sheet-Namen, falls nö
23            # tig
24            # df['Sheet_Name'] = sheet_name
25            dataframe_liste.append(df)
26
27        # Alle DataFrames zu einem einzigen DataFrame kombinieren
28        kombiniertes_df = pd.concat(dataframe_liste, ignore_index=True)
29
30        # Prüfen, ob die Spalte 'Straftaten_insgesamt' existiert
31        if 'Straftaten_insgesamt' not in kombiniertes_df.columns:
32            raise ValueError("Die Spalte 'Straftaten_insgesamt' wurde in den
33            Daten nicht gefunden.")
34
35        # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
36        sortiertes_df = kombiniertes_df.sort_values(by='Straftaten_insgesamt',
37            ascending=False).reset_index(drop=True)
38
39        return sortiertes_df
40
41    except FileNotFoundError:
42        print(f"Die Datei {excel_datei} wurde nicht gefunden.")
43    except Exception as e:
44        print(f"Ein Fehler ist aufgetreten: {e}")
45
46 if __name__ == "__main__":

```

```

43     # Pfad zur Excel-Datei
44     dateipfad = 'Fallzahlen.xlsx'
45
46     # Funktion aufrufen und sortierten DataFrame erhalten
47     df_sortiert = lese_und_sortiere_fallzahlen(dateipfad)
48
49     if df_sortiert is not None:
50         # Sortierten DataFrame anzeigen
51         print(df_sortiert)
52
53         # Optional: Sortierten DataFrame in eine neue Excel-Datei speichern
54         df_sortiert.to_excel('Fallzahlen_sortiert.xlsx', index=False)

```

Listing 17: Skript Testfall 2 Prompt 1 Ausführung 1

```

1  import pandas as pd
2
3  # Pfad zur Excel-Datei
4  excel_datei = 'Fallzahlen.xlsx'
5
6  # Einlesen aller Sheets in ein Dictionary von DataFrames
7  try:
8      alle_sheets = pd.read_excel(excel_datei, sheet_name=None)
9  except FileNotFoundError:
10     print(f"Die Datei {excel_datei} wurde nicht gefunden.")
11     exit(1)
12 except Exception as e:
13     print(f"Fehler beim Einlesen der Excel-Datei: {e}")
14     exit(1)
15
16 # Liste zur Speicherung der relevanten Daten aus allen Sheets
17 daten_liste = []
18
19 # Durch alle DataFrames iterieren und relevante Daten extrahieren
20 for sheet_name, df in alle_sheets.items():
21     # Überprüfen, ob die notwendigen Spalten vorhanden sind
22     if 'Bezirke' in df.columns and 'Straftaten_insgesamt' in df.columns:
23         # Auswahl der relevanten Spalten
24         df_relevant = df[['Bezirke', 'Straftaten_insgesamt']]
25         daten_liste.append(df_relevant)
26     else:
27         print(f"Sheet '{sheet_name}' enthält nicht die erforderlichen Spalten")
28
29 # Kombinieren aller relevanten Daten
30 if daten_liste:
31     alle_daten = pd.concat(daten_liste)
32 else:
33     print("Keine relevanten Daten gefunden.")
34     exit(1)
35
36 # Aggregation der Straftaten pro Bezirk
37 aggregierte_daten = alle_daten.groupby('Bezirke', as_index=False)['

```

```

    Straftaten_insgesamt'].sum()
38
39 # Sortieren der Bezirke nach der Anzahl der Straftaten (absteigend)
40 aggregierte_daten_sortiert = aggregierte_daten.sort_values(by='
    Straftaten_insgesamt', ascending=False).reset_index(drop=True)
41
42 # Optional: Anzeige des resultierenden DataFrames
43 print(aggregierte_daten_sortiert)
44
45 # Optional: Speichern des DataFrames in eine neue Excel-Datei
46 # aggregierte_daten_sortiert.to_excel('Aggregierte_Fallzahlen.xlsx', index=
    False)

```

Listing 18: Skript Testfall 2 Prompt 1 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Alle Sheet-Namen aus der Excel-Datei auslesen
7 xls = pd.ExcelFile(excel_datei)
8 sheet_namen = xls.sheet_names
9
10 # Liste zur Speicherung der DataFrames aller Sheets
11 daten_frames = []
12
13 # Iteriere über jedes Sheet und lese die Daten ein
14 for sheet in sheet_namen:
15     df = pd.read_excel(excel_datei, sheet_name=sheet)
16     daten_frames.append(df)
17
18 # Alle DataFrames zu einem einzigen DataFrame zusammenführen
19 kombiniert_df = pd.concat(daten_frames, ignore_index=True)
20
21 # Optional: Duplikate entfernen, falls Bezirke in mehreren Sheets vorkommen
22 # kombiniert_df = kombiniert_df.drop_duplicates(subset=['Bezirke'])
23
24 # Gruppieren nach 'Bezirke' und Summieren der 'Straftaten_insgesamt'
25 aggregiert_df = kombiniert_df.groupby('Bezirke', as_index=False)['
    Straftaten_insgesamt'].sum()
26
27 # Sortieren nach der Anzahl der Straftaten in absteigender Reihenfolge
28 sortiert_df = aggregiert_df.sort_values(by='Straftaten_insgesamt', ascending=
    False)
29
30 # Optional: Index zurücksetzen
31 sortiert_df = sortiert_df.reset_index(drop=True)
32
33 # Ergebnis anzeigen
34 print(sortiert_df)
35
36 # Optional: Das sortierte Ergebnis in eine neue Excel- oder CSV-Datei

```

```

    speichern
37 # sortiert_df.to_excel('Aggregierte_Fallzahlen.xlsx', index=False)
38 # sortiert_df.to_csv('Aggregierte_Fallzahlen.csv', index=False)

```

Listing 19: Skript Testfall 2 Prompt 1 Ausführung 3

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # Schritt 1: Alle Sheets der Excel-Datei einlesen
7 # Das Argument sheet_name=None liest alle Sheets und gibt ein Dictionary zurück
8 all_sheets = pd.read_excel(excel_file, sheet_name=None)
9
10 # Schritt 2: Daten aus allen Sheets zusammenführen
11 # Wir gehen davon aus, dass alle Sheets die gleiche Struktur haben
12 # Daher können wir sie einfach mit pd.concat zusammenfügen
13 combined_df = pd.concat(all_sheets.values(), ignore_index=True)
14
15 # Schritt 3: Daten nach 'Bezirke' gruppieren und 'Straftaten_insgesamt'
    summieren
16 # Dies aggregiert die Straftaten pro Bezirk über alle Sheets hinweg
17 grouped_df = combined_df.groupby('Bezirke')['Straftaten_insgesamt'].sum().
    reset_index()
18
19 # Schritt 4: Die aggregierten Daten nach 'Straftaten_insgesamt' sortieren (
    absteigend)
20 sorted_df = grouped_df.sort_values(by='Straftaten_insgesamt', ascending=False)
    .reset_index(drop=True)
21
22 # Optional: Anzeigen des sortierten DataFrames
23 print(sorted_df)
24
25 # Optional: Den sortierten DataFrame in eine neue Excel-Datei speichern
26 sorted_df.to_excel('Aggregierte_Straftaten_pro_Bezirk.xlsx', index=False)

```

Listing 20: Skript Testfall 2 Prompt 1 Ausführung 4

```

1 import pandas as pd
2
3 def lese_und_sortiere_fallzahlen(excel_datei):
4     """
5     Liest alle Sheets aus der gegebenen Excel-Datei, kombiniert die Daten,
6     sortiert sie nach der Gesamtanzahl der Straftaten pro Bezirk und
7     gibt das sortierte DataFrame zurück.
8
9     :param excel_datei: Pfad zur Excel-Datei 'Fallzahlen.xlsx'
10    :return: Sortiertes pandas DataFrame
11    """
12    try:
13        # Alle Sheets lesen; sheet_name=None gibt ein Dictionary zurück

```

```

14     alle_sheets = pd.read_excel(excel_datei, sheet_name=None)
15     print(f"Anzahl der gelesenen Sheets: {len(alle_sheets)}")
16 except FileNotFoundError:
17     print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
18     return None
19 except Exception as e:
20     print(f"Ein Fehler ist aufgetreten: {e}")
21     return None
22
23 # Kombinieren aller DataFrames in einen
24 try:
25     kombiniertes_df = pd.concat(alle_sheets.values(), ignore_index=True)
26     print(f"Anzahl der Zeilen nach dem Kombinieren: {len(kombiniertes_df)}")
27 except ValueError as ve:
28     print(f"Fehler beim Kombinieren der Daten: {ve}")
29     return None
30
31 # Sicherstellen, dass die Spalte 'Straftaten_insgesamt' numerisch ist
32 if not pd.api.types.is_numeric_dtype(kombiniertes_df['Straftaten_insgesamt']):
33     kombiniertes_df['Straftaten_insgesamt'] = pd.to_numeric(
34         kombiniertes_df['Straftaten_insgesamt'], errors='coerce'
35     )
36     kombiniertes_df = kombiniertes_df.dropna(subset=['Straftaten_insgesamt'])
37
38 # Sortieren nach 'Straftaten_insgesamt' absteigend
39 sortiertes_df = kombiniertes_df.sort_values(
40     by='Straftaten_insgesamt', ascending=False
41 ).reset_index(drop=True)
42
43 print("Die Daten wurden erfolgreich sortiert.")
44 return sortiertes_df
45
46 def main():
47     excel_datei = 'Fallzahlen.xlsx'
48     df = lese_und_sortiere_fallzahlen(excel_datei)
49     if df is not None:
50         # Optional: Das sortierte DataFrame anzeigen
51         print(df)
52
53         # Optional: Das sortierte DataFrame speichern
54         # df.to_csv('Sortierte_Fallzahlen.csv', index=False, encoding='utf-8-sig')
55         # print("Das sortierte DataFrame wurde als 'Sortierte_Fallzahlen.csv' gespeichert.")
56
57 if __name__ == "__main__":
58     main()

```

Listing 21: Skript Testfall 2 Prompt 1 Ausführung 5

```

1 import pandas as pd
2
3 def zusammenfuegen_und_sortieren(excel_datei):
4     """
5     Liest alle Sheets einer Excel-Datei, fügt sie zusammen,
6     entfernt bestimmte LOR-Schlüssel, sortiert nach 'Straftaten_insgesamt'
7     und gibt den resultierenden DataFrame zurück.
8
9     Parameters:
10    - excel_datei: str, Pfad zur Excel-Datei.
11
12    Returns:
13    - pd.DataFrame: Gefilterter und sortierter DataFrame.
14    """
15
16    # Laden Sie die Excel-Datei und holen Sie alle Sheet-Namen
17    xls = pd.ExcelFile(excel_datei)
18    sheet_namen = xls.sheet_names
19
20    # Liste zur Speicherung der DataFrames aus jedem Sheet
21    df_list = []
22
23    # Durchlaufen Sie jede Sheet und lesen Sie die Daten
24    for sheet in sheet_namen:
25        df = pd.read_excel(excel_datei, sheet_name=sheet)
26        df_list.append(df)
27        print(f"Gesheet '{sheet}' erfolgreich geladen.")
28
29    # Verbinden Sie alle DataFrames zu einem einzigen DataFrame
30    kombiniert_df = pd.concat(df_list, ignore_index=True)
31    print("Alle Sheets wurden erfolgreich zusammengeführt.")
32
33    # Entfernen Sie die Zeilen mit den LOR-Schlüsseln 999900 und 999999
34    filtered_df = kombiniert_df[~kombiniert_df['LOR-Schlüssel'].isin([999900,
35    999999])]
36    print("Unerwünschte LOR-Schlüssel wurden entfernt.")
37
38    # Sortieren Sie den DataFrame nach 'Straftaten_insgesamt' in absteigender
39    Reihenfolge
40    sorted_df = filtered_df.sort_values(by='Straftaten_insgesamt', ascending=
41    False).reset_index(drop=True)
42    print("DataFrame wurde nach 'Straftaten_insgesamt' sortiert.")
43
44    return sorted_df
45
46 if __name__ == "__main__":
47     # Pfad zur Excel-Datei
48     excel_pfad = 'Fallzahlen.xlsx'
49
50     # Rufen Sie die Funktion auf und speichern Sie das Ergebnis in 'final_df'
51     final_df = zusammenfuegen_und_sortieren(excel_pfad)

```



```

50 # Optional: Zeigen Sie die ersten paar Zeilen des finalen DataFrames an
51 print(final_df.head())
52
53 # Optional: Speichern Sie den finalen DataFrame als neue Excel-Datei
54 # final_df.to_excel('Zusammengefuellte_Fallzahlen.xlsx', index=False)

```

Listing 22: Skript Testfall 2 Prompt 2 Ausführung 1

```

1 import pandas as pd
2
3 def load_and_process_excel(file_path):
4     """
5     Lädt alle Sheets aus der angegebenen Excel-Datei, fügt sie zusammen,
6     filtert unerwünschte Zeilen, sortiert die Daten und gibt einen DataFrame
7     zurück.
8
9     Parameters:
10         file_path (str): Der Pfad zur Excel-Datei.
11
12     Returns:
13         pd.DataFrame: Der verarbeitete DataFrame.
14     """
15     try:
16         # Lade alle Sheets in ein Dictionary von DataFrames
17         all_sheets = pd.read_excel(file_path, sheet_name=None, dtype={'LOR-
18 Schlüssel': str})
19
20         # Liste zum Speichern der DataFrames
21         df_list = []
22
23         for sheet_name, df in all_sheets.items():
24             print(f"Lade Sheet: {sheet_name} mit {len(df)} Zeilen.")
25             df_list.append(df)
26
27         # Kombiniere alle DataFrames in einen einzigen DataFrame
28         combined_df = pd.concat(df_list, ignore_index=True)
29         print(f"Gesamtanzahl der Zeilen nach dem Zusammenführen: {len(
30 combined_df)}")
31
32         # Entferne Zeilen mit LOR-Schlüssel 999900 und 999999
33         filtered_df = combined_df[~combined_df['LOR-Schlüssel'].isin([999900,
34 999999])]
35         print(f"Anzahl der Zeilen nach dem Filtern: {len(filtered_df)}")
36
37         # Sortiere nach 'Straftaten_insgesamt' absteigend
38         sorted_df = filtered_df.sort_values(by='Straftaten_insgesamt',
39 ascending=False)
40         print("Daten wurden nach 'Straftaten_insgesamt' sortiert.")
41
42         # Optional: Setze den Index zurück
43         sorted_df.reset_index(drop=True, inplace=True)
44
45         return sorted_df

```

```

41
42     except FileNotFoundError:
43         print(f"Die Datei {file_path} wurde nicht gefunden.")
44     except Exception as e:
45         print(f"Ein Fehler ist aufgetreten: {e}")
46
47 if __name__ == "__main__":
48     # Pfad zur Excel-Datei
49     excel_file = 'Fallzahlen.xlsx'
50
51     # Verarbeite die Excel-Datei
52     final_df = load_and_process_excel(excel_file)
53
54     if final_df is not None:
55         # Zeige die ersten paar Zeilen des finalen DataFrames
56         print("\nErgebnis:")
57         print(final_df.head())
58
59         # Optional: Speichere den DataFrame in eine neue Excel- oder CSV-
        Datei
60         # final_df.to_excel('Zusammengefuegte_Fallzahlen.xlsx', index=False)
61         # final_df.to_csv('Zusammengefuegte_Fallzahlen.csv', index=False)

```

Listing 23: Skript Testfall 2 Prompt 2 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # LOR-Schlüssel, die ausgeschlossen werden sollen
7 exclude_lor_keys = [999900, 999999]
8
9 # Alle Sheets laden
10 sheets_dict = pd.read_excel(excel_file, sheet_name=None)
11
12 # Liste zur Speicherung der bereinigten DataFrames
13 filtered_dfs = []
14
15 for sheet_name, df in sheets_dict.items():
16     # Sicherstellen, dass die benötigten Spalten vorhanden sind
17     if 'LOR-Schlüssel' not in df.columns or 'Bezirke' not in df.columns:
18         print(f"Sheet '{sheet_name}' übersprungen, da erforderliche Spalten
        fehlen.")
19         continue
20
21     # Ausschließen der unerwünschten LOR-Schlüssel
22     df_filtered = df[~df['LOR-Schlüssel'].isin(exclude_lor_keys)].copy()
23
24     # Optional: Hinzufügen einer Spalte zur Kennzeichnung des Sheets
25     df_filtered['Sheet'] = sheet_name
26
27     filtered_dfs.append(df_filtered)

```

```

28
29 # Überprüfen, ob es DataFrames zum Zusammenführen gibt
30 if not filtered_dfs:
31     raise ValueError("Keine gültigen Daten zum Zusammenführen gefunden.")
32
33 # Zusammenführen aller DataFrames auf Basis von 'LOR-Schlüssel' und 'Bezirke'
34 # Falls es gemeinsame Spalten außer den Schlüsseln gibt, werden sie mit
    Suffixen versehen
35 from functools import reduce
36
37 # Füge die DataFrames schrittweise zusammen
38 merged_df = reduce(lambda left, right: pd.merge(left, right, on=['LOR-Schlüssel', 'Bezirke'], how='outer', suffixes=('', '_dup')), filtered_dfs)
39
40 # Optional: Entfernen von doppelten Spalten, die durch die Zusammenführung
    entstehen könnten
41 merged_df = merged_df.loc[:, ~merged_df.columns.duplicated()]
42
43 # Sortieren nach 'Straftaten_insgesamt' absteigend
44 if 'Straftaten_insgesamt' in merged_df.columns:
45     sorted_df = merged_df.sort_values(by='Straftaten_insgesamt', ascending=False)
46 else:
47     print("'Straftaten_insgesamt' Spalte nicht gefunden. Sortierung übersprungen.")
48     sorted_df = merged_df
49
50 # Reset des Indexes
51 sorted_df.reset_index(drop=True, inplace=True)
52
53 # Ergebnis anzeigen
54 print(sorted_df)
55
56 # Optional: Speichern des sortierten DataFrames in eine neue Excel-Datei
57 sorted_df.to_excel('Zusammengefuegte_Fallzahlen.xlsx', index=False)

```

Listing 24: Skript Testfall 2 Prompt 2 Ausführung 3

```

1 import pandas as pd
2
3 def merge_and_sort_fallzahlen(excel_file):
4     """
5     Liest alle Sheets der angegebenen Excel-Datei, filtert unerwünschte LOR-
        Schlüssel,
6     fasst die Daten der Bezirke zusammen, sortiert sie nach
        Straftaten_insgesamt
7     und gibt das resultierende DataFrame zurück.
8
9     Parameters:
10     - excel_file (str): Pfad zur Excel-Datei (z.B. 'Fallzahlen.xlsx')
11
12     Returns:
13     - pd.DataFrame: Gefiltertes und sortiertes DataFrame

```

```

14     """
15
16     # Definiere die unerwünschten LOR-Schlüssel
17     exclude_lor_keys = [999900, 999999]
18
19     try:
20         # Lade alle Sheets in ein Dictionary von DataFrames
21         sheets_dict = pd.read_excel(excel_file, sheet_name=None)
22         print(f"Es wurden {len(sheets_dict)} Sheets gefunden und geladen.")
23     except FileNotFoundError:
24         print(f"Die Datei '{excel_file}' wurde nicht gefunden.")
25         return None
26     except Exception as e:
27         print(f"Ein Fehler ist beim Lesen der Excel-Datei aufgetreten: {e}")
28         return None
29
30     # Liste zur Speicherung gefilterter DataFrames
31     filtered_dfs = []
32
33     # Iteriere über jedes Sheet und filtere die unerwünschten Zeilen
34     for sheet_name, df in sheets_dict.items():
35         print(f"Verarbeite Sheet: {sheet_name}")
36
37         # Überprüfe, ob die erforderlichen Spalten vorhanden sind
38         required_columns = ['LOR-Schlüssel', 'Bezirke', 'Straftaten_insgesamt']
39
40         if not all(col in df.columns for col in required_columns):
41             print(f"Sheet '{sheet_name}' fehlt eine oder mehrere
42             erforderliche Spalten. Überspringe dieses Sheet.")
43             continue
44
45         # Filtere die unerwünschten LOR-Schlüssel
46         df_filtered = df[~df['LOR-Schlüssel'].isin(exclude_lor_keys)].copy()
47         print(f"Anzahl der Bezirke nach Filtern in Sheet '{sheet_name}': {len
48         (df_filtered)}")
49
50         # Optional: Falls du weitere Daten aggregieren möchtest, kannst du
51         hier weitere Schritte hinzufügen
52
53         filtered_dfs.append(df_filtered)
54
55     if not filtered_dfs:
56         print("Keine gültigen Daten zum Zusammenfügen gefunden.")
57         return None
58
59     # Füge alle gefilterten DataFrames zusammen
60     combined_df = pd.concat(filtered_dfs, ignore_index=True)
61     print(f"Gesamtanzahl der Zeilen nach dem Zusammenfügen: {len(combined_df)
62     }")
63
64     # Gruppiere die Daten nach 'LOR-Schlüssel' und 'Bezirke' und summiere '
65     Straftaten_insgesamt'

```

```

60 aggregated_df = combined_df.groupby(['LOR-Schlüssel', 'Bezirke'],
as_index=False)['Straftaten_insgesamt'].sum()
61 print("Daten wurden nach 'LOR-Schlüssel' und 'Bezirke' aggregiert.")
62
63 # Sortiere das DataFrame nach 'Straftaten_insgesamt' in absteigender
Reihenfolge
64 sorted_df = aggregated_df.sort_values(by='Straftaten_insgesamt',
ascending=False).reset_index(drop=True)
65 print("Daten wurden nach 'Straftaten_insgesamt' sortiert.")
66
67 return sorted_df
68
69 if __name__ == "__main__":
70     # Pfad zur Excel-Datei
71     excel_file = 'Fallzahlen.xlsx'
72
73     # Rufe die Funktion auf und erhalte das sortierte DataFrame
74     result_df = merge_and_sort_fallzahlen(excel_file)
75
76     if result_df is not None:
77         # Zeige die ersten paar Zeilen des Ergebnisses an
78         print("\nSortiertes Ergebnis:")
79         print(result_df.head())
80
81         # Optional: Speichere das Ergebnis in einer neuen Excel-Datei oder
CSV
82         # result_df.to_excel('Aggregierte_Fallzahlen.xlsx', index=False)
83         # result_df.to_csv('Aggregierte_Fallzahlen.csv', index=False)

```

Listing 25: Skript Testfall 2 Prompt 2 Ausführung 4

```

1 import pandas as pd
2
3 def process_fallzahlen_excel(file_path):
4     """
5     Liest die Excel-Datei 'Fallzahlen.xlsx', verarbeitet die Daten aus allen
Sheets,
6     entfernt bestimmte LOR-Schlüssel, fasst die Daten zusammen, sortiert sie
nach
7     'Straftaten_insgesamt' und gibt den finalen DataFrame zurück.
8
9     :param file_path: Pfad zur Excel-Datei
10    :return: Gefilterter und sortierter Pandas DataFrame
11    """
12    # Lese die Excel-Datei
13    try:
14        xls = pd.ExcelFile(file_path)
15    except FileNotFoundError:
16        print(f"Die Datei {file_path} wurde nicht gefunden.")
17        return None
18    except Exception as e:
19        print(f"Ein Fehler ist aufgetreten: {e}")
20        return None

```

```

21
22 # Liste zur Speicherung der einzelnen DataFrames
23 df_list = []
24
25 # Iteriere über alle Sheets
26 for sheet_name in xls.sheet_names:
27     try:
28         df = pd.read_excel(xls, sheet_name=sheet_name)
29
30         # Überprüfe, ob die notwendigen Spalten vorhanden sind
31         required_columns = ['LOR-Schlüssel', 'Bezirke', '
Straftaten_insgesamt']
32         if not all(col in df.columns for col in required_columns):
33             print(f"Ein oder mehrere erforderliche Spalten fehlen im
Sheet '{sheet_name}'.")
34             continue
35
36         # Entferne die Zeilen mit unerwünschten LOR-Schlüsseln
37         df_filtered = df[~df['LOR-Schlüssel'].isin([999900, 999999])]
38
39         # Wähle nur die benötigten Spalten
40         df_selected = df_filtered[required_columns]
41
42         # Füge eine neue Spalte für das Sheet hinzu (optional, falls benö
tigt)
43         df_selected['Sheet'] = sheet_name
44
45         # Füge den DataFrame der Liste hinzu
46         df_list.append(df_selected)
47
48     except Exception as e:
49         print(f"Fehler beim Verarbeiten des Sheets '{sheet_name}': {e}")
50         continue
51
52 if not df_list:
53     print("Keine Daten zum Verarbeiten gefunden.")
54     return None
55
56 # Füge alle DataFrames zusammen
57 combined_df = pd.concat(df_list, ignore_index=True)
58
59 # Gruppiere nach 'LOR-Schlüssel' und 'Bezirke' und summiere '
Straftaten_insgesamt'
60 grouped_df = combined_df.groupby(['LOR-Schlüssel', 'Bezirke'], as_index=
False)['Straftaten_insgesamt'].sum()
61
62 # Sortiere nach 'Straftaten_insgesamt' absteigend
63 sorted_df = grouped_df.sort_values(by='Straftaten_insgesamt', ascending=
False).reset_index(drop=True)
64
65 return sorted_df
66

```

```

67 if __name__ == "__main__":
68     # Pfad zur Excel-Datei
69     excel_file_path = 'Fallzahlen.xlsx'
70
71     # Verarbeite die Excel-Datei und erhalte den finalen DataFrame
72     final_df = process_fallzahlen_excel(excel_file_path)
73
74     if final_df is not None:
75         # Zeige die ersten paar Zeilen des finalen DataFrames an
76         print(final_df)
77
78         # Optional: Speichere den finalen DataFrame in einer neuen Excel-
oder CSV-Datei
79         # final_df.to_excel('Zusammengefasste_Fallzahlen.xlsx', index=False)
80         # final_df.to_csv('Zusammengefasste_Fallzahlen.csv', index=False)

```

Listing 26: Skript Testfall 2 Prompt 2 Ausführung 5

```

1 import pandas as pd
2
3 def main():
4     # 1. Excel-Datei einlesen und alle Sheets als DataFrames speichern
5     excel_datei = 'Fallzahlen.xlsx'
6     try:
7         # Liest alle Sheets und speichert sie in einem Dictionary
8         sheets_dict = pd.read_excel(excel_datei, sheet_name=None)
9     except FileNotFoundError:
10         print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
11         return
12     except Exception as e:
13         print(f"Fehler beim Lesen der Excel-Datei: {e}")
14         return
15
16     # 2. DataFrames der einzelnen Sheets zusammenführen
17     df_liste = []
18     for sheet_name, df in sheets_dict.items():
19         # Überprüfen, ob die erforderlichen Spalten existieren
20         if 'LOR-Schlüssel' not in df.columns or 'Bezirke' not in df.columns:
21             print(f"Sheet '{sheet_name}' enthält nicht die erforderlichen
Spalten.")
22             continue
23
24         # Setzen von 'LOR-Schlüssel' und 'Bezirke' als Index
25         df = df.set_index(['LOR-Schlüssel', 'Bezirke'])
26         df_liste.append(df)
27
28     if not df_liste:
29         print("Keine gültigen Sheets zum Verarbeiten gefunden.")
30         return
31
32     # Alle DataFrames zusammenfügen und numerische Spalten akkumulieren
33     zusammengefuegt_df = pd.concat(df_liste)
34     # Gruppieren nach den Index-Spalten und summieren der numerischen Werte

```

```

35     akkumuliert_df = zusammengefuegt_df.groupby(level=['LOR-Schlüssel', '
    Bezirke']).sum().reset_index()
36
37     # 3. DataFrame sortieren nach 'Straftaten_insgesamt', spezielle Zeilen am
    Ende platzieren
38     # Definieren der speziellen LOR-Schlüssel
39     spezielle_keys = [999900, 999999]
40
41     # Trennen der regulären und speziellen Zeilen
42     regulär_df = akkumuliert_df[~akkumuliert_df['LOR-Schlüssel'].isin(
    spezielle_keys)]
43     spezielle_df = akkumuliert_df[akkumuliert_df['LOR-Schlüssel'].isin(
    spezielle_keys)]
44
45     # Sortieren der regulären Zeilen nach 'Straftaten_insgesamt' absteigend
46     sortiert_regulär_df = regulär_df.sort_values(by='Straftaten_insgesamt',
    ascending=False)
47
48     # Kombinieren der sortierten regulären Zeilen mit den speziellen Zeilen
49     endgültiges_df = pd.concat([sortiert_regulär_df, spezielle_df],
    ignore_index=True)
50
51     # 4. Sortiertes DataFrame zurückgeben (hier als Ausgabe angezeigt)
52     print(endgültiges_df)
53
54 if __name__ == '__main__':
55     main()

```

Listing 27: Skript Testfall 2 Prompt 3 Ausführung 1

```

1 import pandas as pd
2
3 def process_fallzahlen_excel(file_path='Fallzahlen.xlsx'):
4     """
5     Liest die Excel-Datei ein, führt die Sheets zusammen, akkumuliert die
    Straftaten,
6     sortiert das DataFrame nach 'Straftaten_insgesamt' und platziert
    spezielle
7     LOR-Schlüssel am Ende.
8
9     Parameters:
10    - file_path: str, der Pfad zur Excel-Datei.
11
12    Returns:
13    - pd.DataFrame: Das sortierte DataFrame.
14    """
15    # Schritt 1: Excel-Datei einlesen und alle Sheets als DataFrames
    speichern
16    try:
17        sheets = pd.read_excel(file_path, sheet_name=None)
18    except FileNotFoundError:
19        print(f"Die Datei {file_path} wurde nicht gefunden.")
20        return None

```



```

21     except Exception as e:
22         print(f"Fehler beim Einlesen der Datei: {e}")
23         return None
24
25     # Schritt 2: DataFrames zusammenführen und Straftaten akkumulieren
26     # Dabei werden alle Sheets kompiliert und numerische Werte summiert
27     try:
28         # Alle DataFrames zu einem einzigen DataFrame zusammenführen
29         df_all = pd.concat(sheets.values(), ignore_index=True)
30
31         # Gruppieren nach 'LOR-Schlüssel' und 'Bezirke' und numerische
32         # Spalten summieren
33         group_columns = ['LOR-Schlüssel', 'Bezirke']
34         df_grouped = df_all.groupby(group_columns, as_index=False).sum()
35     except KeyError as e:
36         print(f"Erforderliche Spalte fehlt: {e}")
37         return None
38     except Exception as e:
39         print(f"Fehler beim Zusammenführen der DataFrames: {e}")
40         return None
41
42     # Schritt 3: Sortieren nach 'Straftaten_insgesamt', spezielle LOR-Schlüssel am Ende
43     try:
44         # Definieren der speziellen LOR-Schlüssel
45         special_keys = [999900, 999999]
46
47         # Separieren der speziellen und normalen Zeilen
48         df_special = df_grouped[df_grouped['LOR-Schlüssel'].isin(special_keys)]
49         df_normal = df_grouped[~df_grouped['LOR-Schlüssel'].isin(special_keys)]
50
51         # Sortieren der normalen Zeilen nach 'Straftaten_insgesamt'
52         # absteigend
53         df_normal_sorted = df_normal.sort_values(by='Straftaten_insgesamt',
54         ascending=False)
55
56         # Zusammenführen der sortierten normalen Zeilen mit den speziellen
57         # Zeilen
58         df_final = pd.concat([df_normal_sorted, df_special], ignore_index=True)
59     except KeyError:
60         print("Die Spalte 'Straftaten_insgesamt' ist im DataFrame nicht
61         vorhanden.")
62         return None
63     except Exception as e:
64         print(f"Fehler beim Sortieren des DataFrames: {e}")
65         return None
66
67     # Schritt 4: Rückgabe des sortierten DataFrames
68     return df_final

```

```

64
65 if __name__ == "__main__":
66     # Verarbeiten der Excel-Datei und Erhalten des sortierten DataFrames
67     sorted_df = process_fallzahlen_excel('Fallzahlen.xlsx')
68
69     if sorted_df is not None:
70         # Anzeige der ersten paar Zeilen des sortierten DataFrames
71         print(sorted_df.head())
72
73         # Optional: Speichern des sortierten DataFrames in einer neuen Excel-
74         # Datei
75         # sorted_df.to_excel('Fallzahlen_sortiert.xlsx', index=False)

```

Listing 28: Skript Testfall 2 Prompt 3 Ausführung 2

```

1 import pandas as pd
2
3 def main():
4     # 1. Excel-Datei einlesen und alle Sheets als DataFrames speichern
5     excel_datei = 'Fallzahlen.xlsx'
6     sheets_dict = pd.read_excel(excel_datei, sheet_name=None)
7
8     # 2. DataFrames der einzelnen Sheets zusammenführen (joinen)
9     # Alle DataFrames vertikal concatenieren
10    concatenated_df = pd.concat(sheets_dict.values(), ignore_index=True)
11
12    # Gruppieren nach 'LOR-Schlüssel' und 'Bezirke' und die numerischen
13    # Spalten akkumulieren
14    gruppiert_df = concatenated_df.groupby(['LOR-Schlüssel', 'Bezirke'],
15    as_index=False).sum()
16
17    # 3. DataFrame sortieren nach 'Straftaten_insgesamt', dabei LOR-Schlüssel
18    # 999900 und 999999 am Ende lassen
19    # Definieren der auszuschließenden LOR-Schlüssel
20    ausschluss_keys = [999900, 999999]
21
22    # DataFrame ohne die auszuschließenden LOR-Schlüssel
23    zu_sortieren_df = gruppiert_df[~gruppiert_df['LOR-Schlüssel'].isin(
24    ausschluss_keys)]
25
26    # DataFrame mit den auszuschließenden LOR-Schlüsseln
27    ausgeschlossen_df = gruppiert_df[gruppiert_df['LOR-Schlüssel'].isin(
28    ausschluss_keys)]
29
30    # Sortieren nach 'Straftaten_insgesamt' in absteigender Reihenfolge
31    sortiert_df = zu_sortieren_df.sort_values(by='Straftaten_insgesamt',
32    ascending=False)
33
34    # Ausschließen Zeilen am Ende des sortierten DataFrames anhängen
35    final_df = pd.concat([sortiert_df, ausgeschlossen_df], ignore_index=True)
36
37    # 4. Sortiertes DataFrame zurückgeben (hier wird es angezeigt)
38    print(final_df)

```

```

33
34 if __name__ == '__main__':
35     main()

```

Listing 29: Skript Testfall 2 Prompt 3 Ausführung 3

```

1 import pandas as pd
2
3 def process_fallzahlen(excel_file):
4     # 1. Die Excel-Datei einlesen und die Sheets als DataFrames speichern.
5     # Alle Sheets werden in einem Dictionary gespeichert, wobei der Sheet-
6     # Name der Schlüssel ist.
7     xls = pd.ExcelFile(excel_file)
8     sheets_dict = {}
9     for sheet_name in xls.sheet_names:
10         df = pd.read_excel(xls, sheet_name=sheet_name)
11         sheets_dict[sheet_name] = df
12
13     # 2. Die DataFrames der einzelnen Sheets zusammen joinen,
14     # sodass pro Zeile der akkumulierte Wert der einzelnen Straftaten steht.
15     # Annahme: Alle DataFrames haben die selben 'LOR-Schlüssel' und 'Bezirke'
16     # Wir setzen 'LOR-Schlüssel' und 'Bezirke' als Index und summieren die
17     # numerischen Spalten.
18
19     # Liste der DataFrames
20     df_list = []
21     for df in sheets_dict.values():
22         # Setze 'LOR-Schlüssel' und 'Bezirke' als Index
23         df = df.set_index(['LOR-Schlüssel', 'Bezirke'])
24         df_list.append(df)
25
26     # Addiere alle DataFrames
27     # Dabei werden numerische Spalten summiert, nicht-numerische ignoriert
28     joined_df = pd.concat(df_list).groupby(['LOR-Schlüssel', 'Bezirke']).sum()
29     joined_df = joined_df.reset_index()
30
31     # 3. Das neue gejointe DataFrame nach der Spalte "Straftaten_insgesamt"
32     # sortieren.
33     # Zeilen mit den LOR-Schlüsseln 999900 und 999999 werden nicht in die
34     # Sortierung einbezogen
35     # und am Ende des DataFrames angehängt.
36
37     # Trenne die speziellen LOR-Schlüssel
38     special_keys = [999900, 999999]
39     normal_df = joined_df[~joined_df['LOR-Schlüssel'].isin(special_keys)]
40     special_df = joined_df[joined_df['LOR-Schlüssel'].isin(special_keys)]
41
42     # Sortiere das normale DataFrame nach 'Straftaten_insgesamt' absteigend
43     sorted_normal_df = normal_df.sort_values(by='Straftaten_insgesamt',
44                                              ascending=False)
45
46     # Füge die speziellen Zeilen am Ende hinzu

```

```

41     final_df = pd.concat([sorted_normal_df, special_df], ignore_index=True)
42
43     # 4. Das sortierte Pandas DataFrame zurückgeben.
44     return final_df
45
46 if __name__ == "__main__":
47     # Pfad zur Excel-Datei
48     excel_file = 'Fallzahlen.xlsx'
49
50     # Verarbeite die Excel-Datei
51     sorted_df = process_fallzahlen(excel_file)
52
53     # Optional: Zeige die ersten Zeilen des sortierten DataFrames an
54     print(sorted_df.head())
55
56     # Optional: Speichere das sortierte DataFrame in eine neue Excel-Datei
57     sorted_df.to_excel('Fallzahlen_sortiert.xlsx', index=False)

```

Listing 30: Skript Testfall 2 Prompt 3 Ausführung 4

```

1 import pandas as pd
2
3 def process_fallzahlen_excel(file_path='Fallzahlen.xlsx'):
4     """
5     Liest die Excel-Datei ein, verbindet die Sheets, sortiert das Ergebnis
6     nach
7     'Straftaten_insgesamt' und platziert die nicht zuordenbaren Einträge am
8     Ende.
9
10    Parameters:
11    - file_path: Pfad zur Excel-Datei (Standard: 'Fallzahlen.xlsx')
12
13    Returns:
14    - final_df: Sortiertes Pandas DataFrame
15    """
16    # Schritt 1: Alle Sheets der Excel-Datei einlesen
17    try:
18        excel_sheets = pd.read_excel(file_path, sheet_name=None)
19    except FileNotFoundError:
20        print(f"Datei '{file_path}' wurde nicht gefunden.")
21        return
22    except Exception as e:
23        print(f"Fehler beim Einlesen der Datei: {e}")
24        return
25
26    # Liste zur Speicherung der DataFrames
27    df_list = []
28
29    for sheet_name, df in excel_sheets.items():
30        # Sicherstellen, dass 'LOR-Schlüssel' und 'Bezirke' als Schlüssel
31        # vorhanden sind
32        if 'LOR-Schlüssel' not in df.columns or 'Bezirke' not in df.columns:
33            print(f"Sheet '{sheet_name}' enthält nicht die notwendigen

```

```

        Spalten.")
31         continue
32         df_list.append(df)
33
34     if not df_list:
35         print("Keine gültigen Sheets gefunden.")
36         return
37
38     # Schritt 2: DataFrames zusammenführen (joinen) auf 'LOR-Schlüssel' und '
    Bezirke'
39     merged_df = df_list[0]
40     for df in df_list[1:]:
41         merged_df = pd.merge(merged_df, df, on=['LOR-Schlüssel', 'Bezirke'],
    how='outer', suffixes=('', '_dup'))
42
43         # Entfernen von doppelten Spalten, falls vorhanden
44         dup_columns = [col for col in merged_df.columns if col.endswith('_dup
    ')]
45         if dup_columns:
46             merged_df.drop(columns=dup_columns, inplace=True)
47
48     # Schritt 3: Sortieren nach 'Straftaten_insgesamt', wobei 999900 und
    999999 am Ende stehen
49     # Zuerst sicherstellen, dass 'Straftaten_insgesamt' numerisch ist
50     merged_df['Straftaten_insgesamt'] = pd.to_numeric(merged_df['
    Straftaten_insgesamt'], errors='coerce')
51
52     # Trennen der DataFrames
53     districts_df = merged_df[~merged_df['LOR-Schlüssel'].isin([999900,
    999999])]
54     non_districts_df = merged_df[merged_df['LOR-Schlüssel'].isin([999900,
    999999])]
55
56     # Sortieren der Bezirke
57     sorted_districts_df = districts_df.sort_values(by='Straftaten_insgesamt',
    ascending=False)
58
59     # Kombinieren der sortierten Bezirke mit den nicht zuordenbaren Einträgen
60     final_df = pd.concat([sorted_districts_df, non_districts_df],
    ignore_index=True)
61
62     return final_df
63
64 if __name__ == "__main__":
65     # Pfad zur Excel-Datei
66     excel_file = 'Fallzahlen.xlsx'
67
68     # Verarbeitung der Excel-Datei
69     sorted_df = process_fallzahlen_excel(excel_file)
70
71     if sorted_df is not None:
72         # Ausgabe des sortierten DataFrames

```

```

73     print(sorted_df)
74
75     # Optional: Speichern des Ergebnisses in einer neuen Excel-Datei
76     # sorted_df.to_excel('Sorted_Fallzahlen.xlsx', index=False)

```

Listing 31: Skript Testfall 2 Prompt 3 Ausführung 5

### 8.2.3 Testfall 3

```

1  import pandas as pd
2
3  def berechne_straftaten_anteil(excel_datei, sheet_name, ausgabe_datei):
4      # Excel-Datei einlesen
5      df = pd.read_excel(excel_datei, sheet_name=sheet_name)
6
7      # Anzeigen der Daten zur Überprüfung (optional)
8      # print(df.head())
9
10     # Identifizieren der Gesamtzeile für Berlin
11     # Annahme: Die Gesamtzeile hat 'Bezirke' als "Berlin (PKS gesamt)"
12     gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
13
14     if gesamt_row.empty:
15         # Falls keine explizite Gesamtzeile vorhanden ist, summiere alle
16         # relevanten Bezirke
17         # Annahme: Die Gesamtzeile hat einen LOR-Schlüssel von 999999
18         gesamt_row = df[df['LOR-Schlüssel'] == 999999]
19
20     if gesamt_row.empty:
21         raise ValueError("Gesamtdaten für Berlin nicht gefunden.")
22
23     # Gesamtzahl der Straftaten in Berlin
24     gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].values[0]
25
26     # Filtere nur die Bezirke, die zu Berlin gehören und nicht die
27     # Gesamtzeile oder nicht zuzuordnend
28     # Annahme: 'Stadtgebiet Berlin, nicht zuzuordnen' und 'Berlin (PKS gesamt)'
29     # sollen ausgeschlossen werden
30     relevante_bezirke = df[~df['Bezirke'].isin(['Berlin (PKS gesamt)', '
31     Stadtgebiet Berlin, nicht zuzuordnen'])]
32
33     # Berechne den prozentualen Anteil
34     relevante_bezirke = relevante_bezirke.copy() # Um eine Kopie zu
35     erstellen
36     relevante_bezirke['Prozentualer Anteil (%)'] = (relevante_bezirke['
37     Straftaten_insgesamt'] / gesamt_straftaten) * 100
38
39     # Optional: Runden auf zwei Dezimalstellen
40     relevante_bezirke['Prozentualer Anteil (%)'] = relevante_bezirke['
41     Prozentualer Anteil (%)'].round(2)
42
43     # Sortiere die Bezirke nach dem prozentualen Anteil absteigend

```

```

37     relevante_bezirke = relevante_bezirke.sort_values(by='Prozentualer Anteil
    (%)', ascending=False)
38
39     # Auswahl der relevanten Spalten für die Ausgabe
40     ausgabe_df = relevante_bezirke[['Bezirke', 'Straftaten_insgesamt', '
    Prozentualer Anteil (%)']]
41
42     # Ausgabe in eine neue Excel-Datei
43     ausgabe_df.to_excel(ausgabe_datei, index=False)
44
45     print(f"Die prozentualen Anteile wurden erfolgreich berechnet und in '{
    ausgabe_datei}' gespeichert.")
46
47 if __name__ == "__main__":
48     # Pfad zur Eingabe-Excel-Datei
49     input_excel = 'Fallzahlen.xlsx'
50
51     # Name des Sheets
52     sheet = 'Fallzahlen_2023'
53
54     # Name der Ausgabedatei
55     output_excel = 'Straftaten_Anteil_Bezirke.xlsx'
56
57     # Funktion aufrufen
58     berechne_straftaten_anteil(input_excel, sheet, output_excel)

```

Listing 32: Skript Testfall 3 Prompt 1 Ausführung 1

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Lesen des Excel-Sheets
10 df = pd.read_excel(excel_file, sheet_name=sheet_name)
11
12 # Annahmen, dass die Gesamtzahl in der Zeile mit 'Berlin (PKS gesamt)' steht
13 # Suchen der Gesamtzeile
14 total_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
15
16 if total_row.empty:
17     raise ValueError("Die Gesamtzeile 'Berlin (PKS gesamt)' wurde nicht
    gefunden.")
18
19 # Extrahieren der Gesamtzahl der Straftaten
20 total_crimes = total_row['Straftaten_insgesamt'].values[0]
21
22 # Filtern der Bezirke (ohne Gesamtzeile)
23 districts_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
24

```

```

25 # Berechnen des prozentualen Anteils
26 districts_df['Prozentualer_Anteil (%)'] = (districts_df['Straftaten_insgesamt
    ']/ total_crimes) * 100
27
28 # Optional: Runden auf zwei Dezimalstellen
29 districts_df['Prozentualer_Anteil (%)'] = districts_df['Prozentualer_Anteil
    (%)'].round(2)
30
31 # Anzeigen der Ergebnisse
32 print(districts_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil
    (%)']])
33
34 # Optional: Speichern der Ergebnisse in eine neue Excel-Datei
35 output_file = 'Straftaten_Prozentual_Anteil.xlsx'
36 districts_df.to_excel(output_file, sheet_name='Prozentuale Anteile', index=
    False)
37 print(f"\nDie Ergebnisse wurden in '{output_file}' gespeichert.")

```

Listing 33: Skript Testfall 3 Prompt 1 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Name des Sheets
7 sheet_name = 'Fallzahlen_2023'
8
9 # Lesen der Excel-Datei
10 try:
11     df = pd.read_excel(excel_datei, sheet_name=sheet_name)
12 except FileNotFoundError:
13     print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
14     exit(1)
15 except Exception as e:
16     print(f"Fehler beim Lesen der Excel-Datei: {e}")
17     exit(1)
18
19 # Anzeigen der geladenen Daten (optional)
20 #print(df.head())
21
22 # Herausfinden der Gesamtstraftaten in Berlin
23 # Annahme: Die Gesamtstraftaten stehen in der Zeile mit 'Berlin (PKS gesamt)'
24 gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
25
26 if gesamt_row.empty:
27     print("Die Zeile 'Berlin (PKS gesamt)' wurde nicht gefunden.")
28     exit(1)
29
30 gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].values[0]
31
32 # Filtern der Bezirke (Ausschluss von Gesamtzeilen)
33 bezirke_df = df[~df['Bezirke'].isin(['Berlin (PKS gesamt)', 'Stadtgebiet

```



```

        Berlin, nicht zuzuordnen']]])
34
35 # Berechnung des prozentualen Anteils
36 bezirke_df = bezirke_df.copy() # Vermeidung von SettingWithCopyWarning
37 bezirke_df['Prozentualer_Anteil (%)'] = (bezirke_df['Straftaten_insgesamt'] /
    gesamt_straftaten) * 100
38
39 # Optional: Runden auf zwei Dezimalstellen
40 bezirke_df['Prozentualer_Anteil (%)'] = bezirke_df['Prozentualer_Anteil (%)'
    ].round(2)
41
42 # Sortieren nach Prozentualem Anteil absteigend (optional)
43 bezirke_df = bezirke_df.sort_values(by='Prozentualer_Anteil (%)', ascending=
    False)
44
45 # Ergebnis anzeigen
46 ergebnis = bezirke_df[['Bezirke', 'Straftaten_insgesamt', '
    Prozentualer_Anteil (%)']]
47 print(ergebnis)
48
49 # Optional: Ergebnis in eine neue Excel-Datei speichern
50 # ergebnis.to_excel('Prozentuale_Anteile_Straftaten.xlsx', index=False)

```

Listing 34: Skript Testfall 3 Prompt 1 Ausführung 3

```

1 import pandas as pd
2
3 def berechne_prozentuale_anteile(dateipfad: str, sheet_name: str = '
    Fallzahlen_2023'):
4     """
5     Berechnet den prozentualen Anteil der Straftaten jedes Bezirks an den
    gesamten Straftaten in Berlin.
6
7     :param dateipfad: Pfad zur Excel-Datei 'Fallzahlen.xlsx'.
8     :param sheet_name: Name des Sheets in der Excel-Datei. Standard ist '
    Fallzahlen_2023'.
9     :return: DataFrame mit Bezirken und deren prozentualen Anteilen.
10    """
11    try:
12        # Excel-Datei lesen
13        df = pd.read_excel(dateipfad, sheet_name=sheet_name)
14    except FileNotFoundError:
15        print(f"Die Datei {dateipfad} wurde nicht gefunden.")
16        return
17    except Exception as e:
18        print(f"Ein Fehler ist beim Lesen der Excel-Datei aufgetreten: {e}")
19        return
20
21    # Gesamtstraftaten für Berlin abrufen
22    total_crimes_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
23    if not total_crimes_row.empty:
24        total_crimes = total_crimes_row['Straftaten_insgesamt'].values[0]
25    else:

```

```

26     # Falls 'Berlin (PKS gesamt)' nicht vorhanden ist, summiere die
    Bezirke
27     exclude_bezirke = ['Berlin (PKS gesamt)', 'Stadtgebiet Berlin, nicht
    zuzuordnen']
28     total_crimes = df.loc[~df['Bezirke'].isin(exclude_bezirke), '
    Straftaten_insgesamt'].sum()
29
30     # Bezirke filtern (ausschließen: Gesamtdaten und nicht zuordenbare
    Gebiete)
31     exclude_bezirke = ['Berlin (PKS gesamt)', 'Stadtgebiet Berlin, nicht
    zuzuordnen']
32     bezirke_df = df.loc[~df['Bezirke'].isin(exclude_bezirke)].copy()
33
34     # Prozentualen Anteil berechnen
35     bezirke_df['Prozentualer_Anteil'] = (bezirke_df['Straftaten_insgesamt'] /
    total_crimes) * 100
36
37     # Optional: Auf zwei Dezimalstellen runden
38     bezirke_df['Prozentualer_Anteil'] = bezirke_df['Prozentualer_Anteil'].
    round(2)
39
40     # Ergebnis anzeigen
41     print("Prozentualer Anteil der Straftaten nach Bezirk:")
42     print(bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil
    ']])
43
44     # Optional: Ergebnis in eine neue Excel-Datei speichern
45     output_datei = 'Prozentuale_Anteile_Straftaten.xlsx'
46     try:
47         bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil'
    ]].to_excel(output_datei, index=False)
48         print(f"\nDie Ergebnisse wurden in '{output_datei}' gespeichert.")
49     except Exception as e:
50         print(f"Ein Fehler ist beim Speichern der Ergebnisdatei aufgetreten:
    {e}")
51
52 if __name__ == "__main__":
53     # Pfad zur Excel-Datei
54     dateipfad = 'Fallzahlen.xlsx'
55
56     # Prozentuale Anteile berechnen
57     berechne_prozentuale_anteile(dateipfad)

```

Listing 35: Skript Testfall 3 Prompt 1 Ausführung 4

```

1 import pandas as pd
2
3 def berechne_prozentualer_anteil_straftaten():
4     try:
5         # Excel-Datei laden
6         dateipfad = 'Fallzahlen.xlsx' # Pfad zur Excel-Datei
7         sheet_name = 'Fallzahlen_2023' # Name des Sheets
8

```

```

9         df = pd.read_excel(dateipfad, sheet_name=sheet_name)
10
11         # Überprüfen, ob die benötigten Spalten vorhanden sind
12         erforderliche_spalten = ['Bezirke', 'Straftaten_insgesamt']
13         fehlende_spalten = [spalte for spalte in erforderliche_spalten if
14                             spalte not in df.columns]
15         if fehlende_spalten:
16             raise ValueError(f"Die folgenden benötigten Spalten fehlen im
17                               Excel-Sheet: {fehlende_spalten}")
18
19         # Gesamte Straftaten aus der Gesamtzeile holen
20         gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
21         if not gesamt_row.empty:
22             gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].iloc[0]
23             print(f"Gesamte Straftaten in Berlin: {gesamt_straftaten}")
24         else:
25             # Falls die Gesamtzeile nicht vorhanden ist, summiere alle
26             # Bezirkswerte
27             gesamt_straftaten = df[~df['Bezirke'].isin(['Stadtgebiet Berlin,
28                                                         nicht zuzuordnen'])]['Straftaten_insgesamt'].sum()
29             print(f"Gesamte Straftaten in Berlin (berechnet): {
30                   gesamt_straftaten}")
31
32         # Bezirke ausschließen, die nicht zugeordnet werden können oder die
33         # Gesamtzeile sind
34         bezirke_df = df[~df['Bezirke'].isin(['Berlin (PKS gesamt)', '
35                                               Stadtgebiet Berlin, nicht zuzuordnen'])]
36
37         # Prozentualen Anteil berechnen
38         bezirke_df = bezirke_df.copy() # Kopie erstellen, um
39         SettingWithCopyWarning zu vermeiden
40         bezirke_df['Prozentualer_Anteil (%)'] = (bezirke_df['
41                                                   Straftaten_insgesamt'] / gesamt_straftaten) * 100
42
43         # Prozentualen Anteil auf 2 Dezimalstellen runden
44         bezirke_df['Prozentualer_Anteil (%)'] = bezirke_df['
45                                                   Prozentualer_Anteil (%)'].round(2)
46
47         # Ergebnis anzeigen
48         print("\nProzentualer Anteil der Straftaten pro Bezirk:")
49         print(bezirke_df[['Bezirke', 'Straftaten_insgesamt', '
50                           Prozentualer_Anteil (%)']])
51
52         # Optional: Ergebnis in eine neue Excel-Datei speichern
53         ausgabe_datei = 'Straftaten_Prozentual.xlsx'
54         bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil
55                     (%)']].to_excel(ausgabe_datei, index=False)
56         print(f"\nDas Ergebnis wurde in '{ausgabe_datei}' gespeichert.")
57
58     except FileNotFoundError:
59         print("Fehler: Die Datei 'Fallzahlen.xlsx' wurde nicht gefunden.
60               Bitte überprüfe den Dateipfad.")

```

```

48     except ValueError as ve:
49         print(f"Fehler: {ve}")
50     except Exception as e:
51         print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")
52
53 if __name__ == "__main__":
54     berechne_prozentualer_anteil_straftaten()

```

Listing 36: Skript Testfall 3 Prompt 1 Ausführung 5

```

1  import pandas as pd
2
3  # Pfad zur Excel-Datei
4  excel_datei = 'Fallzahlen.xlsx'
5
6  # Name des Sheets
7  sheet_name = 'Fallzahlen_2023'
8
9  # Lesen der Excel-Datei
10 df = pd.read_excel(excel_datei, sheet_name=sheet_name)
11
12 # Überprüfen, ob die notwendigen Spalten vorhanden sind
13 notwendige_spalten = ['Bezirke', 'Straftaten_insgesamt']
14 for spalte in notwendige_spalten:
15     if spalte not in df.columns:
16         raise ValueError(f"Die Spalte '{spalte}' fehlt in der Excel-Datei.")
17
18 # Extrahieren der Gesamtanzahl von ganz Berlin
19 gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
20
21 if gesamt_row.empty:
22     raise ValueError("Die Gesamtzeile 'Berlin (PKS gesamt)' wurde nicht gefunden.")
23
24 gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].values[0]
25
26 # Entfernen der Gesamtzeile aus dem DataFrame
27 bezirk_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
28
29 # Berechnung des prozentualen Anteils
30 bezirk_df['Prozent_Straftaten_insgesamt'] = (bezirk_df['Straftaten_insgesamt']
31     ] / gesamt_straftaten) * 100
32
33 # Optional: Runde die Prozentwerte auf zwei Dezimalstellen
34 bezirk_df['Prozent_Straftaten_insgesamt'] = bezirk_df['
35     Prozent_Straftaten_insgesamt'].round(2)
36
37 # Ausgabe des Ergebnisses
38 print(bezirk_df[['Bezirke', 'Straftaten_insgesamt', '
39     Prozent_Straftaten_insgesamt']])
40
41 # Optional: Speichern in einer neuen Excel-Datei
42 ausgabe_datei = 'Fallzahlen_mit_Prozentsatz.xlsx'

```

```

40 bezirk_df.to_excel(ausgabe_datei, index=False)
41 print(f"\nDie Ergebnisse wurden in '{ausgabe_datei}' gespeichert.")

```

Listing 37: Skript Testfall 3 Prompt 2 Ausführung 1

```

1  import pandas as pd
2
3  # Pfad zur Excel-Datei
4  excel_datei = 'Fallzahlen.xlsx'
5
6  # Name des Sheets
7  sheet_name = 'Fallzahlen_2023'
8
9  # Laden der Excel-Datei
10 try:
11     df = pd.read_excel(excel_datei, sheet_name=sheet_name)
12 except FileNotFoundError:
13     print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
14     exit(1)
15 except Exception as e:
16     print(f"Ein Fehler ist aufgetreten: {e}")
17     exit(1)
18
19 # Sicherstellen, dass die notwendigen Spalten vorhanden sind
20 erforderliche_spalten = ['Bezirke', 'Straftaten_insgesamt']
21 for spalte in erforderliche_spalten:
22     if spalte not in df.columns:
23         print(f"Die Spalte '{spalte}' fehlt in der Excel-Datei.")
24         exit(1)
25
26 # Extrahieren der Gesamtanzahl von ganz Berlin
27 gesamt_zeile = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
28
29 if gesamt_zeile.empty:
30     print("Die Zeile 'Berlin (PKS gesamt)' wurde nicht gefunden.")
31     exit(1)
32
33 # Annahme: Es gibt genau eine Zeile mit 'Berlin (PKS gesamt)'
34 gesamt_straftaten = gesamt_zeile['Straftaten_insgesamt'].values[0]
35
36 if gesamt_straftaten == 0:
37     print("Die Gesamtanzahl der Straftaten für Berlin ist 0. Division durch Null ist nicht möglich.")
38     exit(1)
39
40 # Filtern der Bezirke (ohne die Gesamtzeile)
41 bezirke_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
42
43 # Berechnung des prozentualen Anteils
44 bezirke_df['Prozentualer_Anteil (%)'] = (bezirke_df['Straftaten_insgesamt'] /
45     gesamt_straftaten) * 100
46
47 # Optional: Runden auf zwei Dezimalstellen

```

```

47 bezirke_df['Prozentualer_Anteil (%)'] = bezirke_df['Prozentualer_Anteil (%)'
   ].round(2)
48
49 # Sortieren nach dem prozentualen Anteil absteigend
50 bezirke_df.sort_values(by='Prozentualer_Anteil (%)', ascending=False, inplace
   =True)
51
52 # Ergebnis anzeigen
53 print(bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil (%)
   ']])
54
55 # Optional: Speichern in eine neue Excel-Datei
56 output_datei = 'Fallzahlen_prozentual.xlsx'
57 try:
58     bezirke_df.to_excel(output_datei, sheet_name='Prozentualer_Anteil', index
   =False)
59     print(f"\nDie Ergebnisse wurden in '{output_datei}' gespeichert.")
60 except Exception as e:
61     print(f"Fehler beim Speichern der Datei: {e}")

```

Listing 38: Skript Testfall 3 Prompt 2 Ausführung 2

```

1  import pandas as pd
2
3  # Pfad zur Excel-Datei
4  excel_datei = 'Fallzahlen.xlsx'
5
6  # Name des Sheets
7  sheet_name = 'Fallzahlen_2023'
8
9  # Lesen des Excel-Sheets
10 try:
11     df = pd.read_excel(excel_datei, sheet_name=sheet_name)
12 except FileNotFoundError:
13     print(f"Die Datei '{excel_datei}' wurde nicht gefunden.")
14     exit(1)
15 except Exception as e:
16     print(f"Ein Fehler ist aufgetreten beim Lesen der Excel-Datei: {e}")
17     exit(1)
18
19 # Überprüfen, ob die notwendigen Spalten vorhanden sind
20 if 'Bezirke' not in df.columns or 'Straftaten_insgesamt' not in df.columns:
21     print("Die erforderlichen Spalten 'Bezirke' oder 'Straftaten_insgesamt'
   fehlen in der Tabelle.")
22     exit(1)
23
24 # Finden des Gesamtwerts für Berlin
25 gesamt_zeile = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
26
27 if gesamt_zeile.empty:
28     print("Die Gesamtzeile 'Berlin (PKS gesamt)' wurde nicht gefunden.")
29     exit(1)
30

```

```

31 gesamt_straftaten = gesamt_zeile['Straftaten_insgesamt'].values[0]
32
33 # Entfernen der Gesamtzeile aus den Bezirken
34 bezirke_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
35
36 # Berechnen des prozentualen Anteils
37 bezirke_df['Prozentualer_Anteil'] = (bezirke_df['Straftaten_insgesamt'] /
    gesamt_straftaten) * 100
38
39 # Optional: Runden auf zwei Dezimalstellen
40 bezirke_df['Prozentualer_Anteil'] = bezirke_df['Prozentualer_Anteil'].round
    (2)
41
42 # Anzeigen der Ergebnisse
43 print(bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil']])
44
45 # Speichern der Ergebnisse in einer neuen Excel-Datei
46 ausgabe_datei = 'Fallzahlen_Prozente.xlsx'
47 try:
48     bezirke_df.to_excel(ausgabe_datei, sheet_name='Prozentuale_Fallzahlen',
    index=False)
49     print(f"\nDie prozentualen Anteile wurden in '{ausgabe_datei}'
    gespeichert.")
50 except Exception as e:
51     print(f"Ein Fehler ist aufgetreten beim Speichern der Excel-Datei: {e}")

```

Listing 39: Skript Testfall 3 Prompt 2 Ausführung 3

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei und Name des Sheets
4 excel_datei = 'Fallzahlen.xlsx'
5 blatt_name = 'Fallzahlen_2023'
6
7 # Lade die Daten aus der Excel-Datei
8 df = pd.read_excel(excel_datei, sheet_name=blatt_name)
9
10 # Finde die Gesamtzahl der Straftaten für ganz Berlin
11 berlin_gesamtreihe = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
12 if berlin_gesamtreihe.empty:
13     raise ValueError("Die Zeile 'Berlin (PKS gesamt)' wurde in den Daten
    nicht gefunden.")
14
15 berlin_gesamt = berlin_gesamtreihe['Straftaten_insgesamt'].iloc[0]
16
17 # Entferne die Gesamtzeile aus dem DataFrame
18 bezirke_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
19
20 # Berechne den prozentualen Anteil der Straftaten für jeden Bezirk
21 bezirke_df['Prozentualer_Anteil'] = ((bezirke_df['Straftaten_insgesamt'] /
    berlin_gesamt) * 100).round(2)
22
23 # Ausgabe der Ergebnisse

```

```

24 print(bezirke_df[['Bezirke', 'Straftaten_insgesamt', 'Prozentualer_Anteil']])
25
26 # Speichere das Ergebnis in eine neue Excel-Datei
27 bezirke_df.to_excel('Fallzahlen_mit_Prozentanteilen.xlsx', index=False)

```

Listing 40: Skript Testfall 3 Prompt 2 Ausführung 4

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei und Name des Sheets
4 excel_datei = 'Fallzahlen.xlsx'
5 sheet_name = 'Fallzahlen_2023'
6
7 # Excel-Datei einlesen
8 df = pd.read_excel(excel_datei, sheet_name=sheet_name)
9
10 # Gesamtanzahl der Straftaten in Berlin extrahieren
11 # Annahme: Der Eintrag für "Berlin (PKS gesamt)" ist eindeutig
12 gesamt_strafaten = df.loc[df['Bezirke'] == 'Berlin (PKS gesamt)', '
    Straftaten_insgesamt'].values[0]
13
14 # Entfernen der Gesamtzeile, um nur die Bezirke zu betrachten
15 df_bezirke = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
16
17 # Berechnung des prozentualen Anteils der Straftaten für jeden Bezirk
18 df_bezirke['Prozent_Straftaten'] = (df_bezirke['Straftaten_insgesamt'] /
    gesamt_strafaten) * 100
19
20 # Optional: Sortieren nach dem prozentualen Anteil
21 df_bezirke.sort_values(by='Prozent_Straftaten', ascending=False, inplace=True)
22
23 # Ergebnis anzeigen
24 print(df_bezirke[['Bezirke', 'Straftaten_insgesamt', 'Prozent_Straftaten']])
25
26 # Optional: Ergebnisse in eine neue Excel-Datei speichern
27 ausgabe_datei = 'Fallzahlen_mit_Prozent.xlsx'
28 df_bezirke.to_excel(ausgabe_datei, index=False)
29 print(f"\nDie Ergebnisse wurden in '{ausgabe_datei}' gespeichert.")

```

Listing 41: Skript Testfall 3 Prompt 2 Ausführung 5

```

1 import pandas as pd
2
3 def main():
4     # 1. Excel-Datei einlesen
5     excel_datei = 'Fallzahlen.xlsx'
6     sheet_name = 'Fallzahlen_2023'
7
8     try:
9         df = pd.read_excel(excel_datei, sheet_name=sheet_name)
10        print(f"Excel-Datei '{excel_datei}' erfolgreich eingelesen.")
11    except FileNotFoundError:

```



```

12         print(f"Fehler: Die Datei '{excel_datei}' wurde nicht gefunden.")
13         return
14     except Exception as e:
15         print(f"Ein Fehler ist beim Einlesen der Excel-Datei aufgetreten: {e}")
16         return
17
18     # 2. Tabelle als Pandas DataFrame speichern
19     # (bereits durch das Einlesen mit pandas erledigt)
20
21     # 3. Überprüfen, ob die notwendigen Spalten vorhanden sind
22     erforderliche_spalten = ['Bezirke', 'Straftaten_insgesamt']
23     fehlende_spalten = [spalte for spalte in erforderliche_spalten if spalte
24                          not in df.columns]
25
26     if fehlende_spalten:
27         print(f"Fehler: Fehlende Spalten in der Excel-Datei: {
28               fehlende_spalten}")
29         return
30     else:
31         print("Alle erforderlichen Spalten sind vorhanden.")
32
33     # 4. Gesamtzahl der Straftaten für ganz Berlin finden
34     gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
35
36     if gesamt_row.empty:
37         print("Fehler: Die Zeile mit 'Berlin (PKS gesamt)' wurde nicht
38               gefunden.")
39         return
40
41     gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].values[0]
42     print(f"Gesamtzahl der Straftaten für Berlin: {gesamt_straftaten}")
43
44     # 5. Prozentualen Anteil der einzelnen Bezirke berechnen
45     # Ausschließen der Gesamtzeile für die Berechnung
46     df_bezirke = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
47
48     if df_bezirke.empty:
49         print("Fehler: Es gibt keine Bezirke nach dem Ausschluss von 'Berlin
50               (PKS gesamt)'.")
51         return
52
53     # Prozentualer Anteil berechnen
54     df_bezirke['Prozentualer_Anteil (%)'] = (df_bezirke['Straftaten_insgesamt']
55                                               / gesamt_straftaten) * 100
56
57     # Optional: Runden auf zwei Dezimalstellen
58     df_bezirke['Prozentualer_Anteil (%)'] = df_bezirke['Prozentualer_Anteil
59               (%)'].round(2)
60
61     print("Prozentualer Anteil der Straftaten pro Bezirk berechnet.")

```

```

57 # 6. Ergebnis als DataFrame zurückgeben
58 # Hier zeigen wir das Ergebnis an. Sie können es auch in eine Datei
    speichern.
59 ergebnis_df = df_bezirke[['Bezirke', 'Straftaten_insgesamt', '
    Prozentualer_Anteil (%)']]
60 print("\nErgebnis DataFrame:")
61 print(ergebnis_df.to_string(index=False))
62
63 # Optional: Ergebnis in eine neue Excel-Datei speichern
64 ausgabe_datei = 'Fallzahlen_prozentual.xlsx'
65 try:
66     ergebnis_df.to_excel(ausgabe_datei, index=False)
67     print(f"\nErgebnis erfolgreich in '{ausgabe_datei}' gespeichert.")
68 except Exception as e:
69     print(f"Ein Fehler ist beim Speichern der Ergebnisdatei aufgetreten:
    {e}")
70
71 if __name__ == "__main__":
72     main()

```

Listing 42: Skript Testfall 3 Prompt 3 Ausführung 1

```

1 import pandas as pd
2
3 def verarbeite_fallzahlen(dateipfad: str, sheet_name: str = 'Fallzahlen_2023'
    ) -> pd.DataFrame:
4     """
5     Liest eine Excel-Datei ein, überprüft notwendige Spalten, berechnet
6     den prozentualen Anteil der Straftaten pro Bezirk und gibt das Ergebnis
        als DataFrame zurück.
7
8     :param dateipfad: Pfad zur Excel-Datei 'Fallzahlen.xlsx'.
9     :param sheet_name: Name des Sheets in der Excel-Datei. Standard ist '
        Fallzahlen_2023'.
10    :return: DataFrame mit Bezirken, Straftaten insgesamt und prozentualen
        Anteil.
11    """
12    try:
13        # 1. Excel-Datei einlesen
14        df = pd.read_excel(dateipfad, sheet_name=sheet_name)
15        print("Excel-Datei erfolgreich eingelesen.")
16    except FileNotFoundError:
17        raise FileNotFoundError(f"Die Datei '{dateipfad}' wurde nicht
        gefunden.")
18    except Exception as e:
19        raise Exception(f"Fehler beim Einlesen der Excel-Datei: {e}")
20
21    # 2. Die Tabelle als Pandas DataFrame gespeichert (bereits in 'df')
22
23    # 3. Überprüfen, ob die notwendigen Spalten vorhanden sind
24    erforderliche_spalten = ['Bezirke', 'Straftaten_insgesamt']
25    fehlende_spalten = [spalte for spalte in erforderliche_spalten if spalte
        not in df.columns]

```

```

26     if fehlende_spalten:
27         raise ValueError(f"Die folgenden erforderlichen Spalten fehlen in der
           Tabelle: {fehlende_spalten}")
28     else:
29         print("Alle erforderlichen Spalten sind vorhanden.")
30
31     # 4. Gesamtzahl der Straftaten für ganz Berlin finden
32     gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
33     if gesamt_row.empty:
34         raise ValueError("Die Zeile mit 'Berlin (PKS gesamt)' wurde nicht
           gefunden.")
35
36     gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].iloc[0]
37     print(f"Gesamtzahl der Straftaten in Berlin: {gesamt_straftaten}")
38
39     if gesamt_straftaten == 0:
40         raise ValueError("Die Gesamtzahl der Straftaten ist 0, kann keine
           Prozente berechnen.")
41
42     # 5. Prozentualen Anteil der einzelnen Bezirke berechnen
43     # Ausschließen der Gesamtzeile von den einzelnen Bezirken
44     einzel_bezirke = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
45
46     # Berechnung des prozentualen Anteils
47     einzel_bezirke['Prozentualer_Anteil'] = (einzel_bezirke['
           Straftaten_insgesamt'] / gesamt_straftaten) * 100
48
49     # Optional: Auf zwei Dezimalstellen runden
50     einzel_bezirke['Prozentualer_Anteil'] = einzel_bezirke['
           Prozentualer_Anteil'].round(2)
51
52     # 6. Ergebnis als DataFrame zurückgeben
53     ergebnis = einzel_bezirke[['Bezirke', 'Straftaten_insgesamt', '
           Prozentualer_Anteil']]
54
55     return ergebnis
56
57 if __name__ == "__main__":
58     dateipfad = 'Fallzahlen.xlsx' # Pfad zur Excel-Datei anpassen, falls
           erforderlich
59     try:
60         ergebnis_df = verarbeite_fallzahlen(dateipfad)
61         print("\nProzentualer Anteil der Straftaten pro Bezirk in Berlin:")
62         print(ergebnis_df.to_string(index=False))
63     except Exception as e:
64         print(f"Ein Fehler ist aufgetreten: {e}")

```

Listing 43: Skript Testfall 3 Prompt 3 Ausführung 2

```

1 import pandas as pd
2
3 def calculate_bezirk_straftaten_percentage(excel_file: str, sheet_name: str =
           'Fallzahlen_2023') -> pd.DataFrame:

```

```

4      """
5      Liest eine Excel-Datei ein und berechnet den prozentualen Anteil der
6      Straftaten_insgesamt für jeden Bezirk im Vergleich zu ganz Berlin.
7
8      Parameters:
9      - excel_file: Pfad zur Excel-Datei.
10     - sheet_name: Name des Sheets in der Excel-Datei (standardmäßig '
11     Fallzahlen_2023').
12
13     Returns:
14     - Ein Pandas DataFrame mit den Bezirken und ihrem prozentualen Anteil.
15     """
16
17     try:
18         # 1. Excel-Datei einlesen
19         df = pd.read_excel(excel_file, sheet_name=sheet_name)
20         print("Excel-Datei erfolgreich eingelesen.")
21     except FileNotFoundError:
22         raise FileNotFoundError(f"Die Datei '{excel_file}' wurde nicht
23         gefunden.")
24     except Exception as e:
25         raise Exception(f"Fehler beim Einlesen der Excel-Datei: {e}")
26
27     # 2. DataFrame gespeichert (bereits in df)
28
29     # 3. Überprüfen, ob die notwendigen Spalten vorhanden sind
30     required_columns = ['Bezirke', 'Straftaten_insgesamt']
31     missing_columns = [col for col in required_columns if col not in df.
32     columns]
33     if missing_columns:
34         raise ValueError(f"Die folgenden erforderlichen Spalten fehlen: {'',
35         '.join(missing_columns)}")
36     print("Alle erforderlichen Spalten sind vorhanden.")
37
38     # 4. Gesamtzahl der Straftaten für ganz Berlin finden
39     total_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
40     if total_row.empty:
41         raise ValueError("Die Zeile mit 'Berlin (PKS gesamt)' wurde nicht
42         gefunden.")
43     total_straftaten = total_row['Straftaten_insgesamt'].values[0]
44     print(f"Gesamtzahl der Straftaten für ganz Berlin: {total_straftaten}")
45
46     # 5. Prozentualen Anteil der einzelnen Bezirke berechnen
47     # Ausschließen der Gesamtzeile aus den Bezirken
48     bezirk_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
49
50     # Berechnung des prozentualen Anteils
51     bezirk_df['Prozentualer_Anteil'] = (bezirk_df['Straftaten_insgesamt'] /
52     total_straftaten) * 100
53
54     # Optional: Runden auf zwei Dezimalstellen
55     bezirk_df['Prozentualer_Anteil'] = bezirk_df['Prozentualer_Anteil'].round

```

```

(2)
50
51 # 6. Ergebnis als DataFrame zurückgeben
52 result_df = bezirk_df[['Bezirke', 'Straftaten_insgesamt', '
Prozentualer_Anteil']]
53
54 print("Prozentualer Anteil der Straftaten pro Bezirk berechnet.")
55 return result_df
56
57 if __name__ == "__main__":
58     # Pfad zur Excel-Datei
59     excel_file = 'Fallzahlen.xlsx'
60
61     # Berechnung durchführen
62     try:
63         prozent_df = calculate_bezirk_straftaten_percentage(excel_file)
64         print("\nErgebnis:")
65         print(prozent_df)
66     except Exception as e:
67         print(f"Ein Fehler ist aufgetreten: {e}")

```

Listing 44: Skript Testfall 3 Prompt 3 Ausführung 3

```

1 import pandas as pd
2
3 def main():
4     # 1. Die Excel-Datei einlesen
5     try:
6         df = pd.read_excel('Fallzahlen.xlsx', sheet_name='Fallzahlen_2023')
7     except FileNotFoundError:
8         print("Die Datei 'Fallzahlen.xlsx' wurde nicht gefunden.")
9         return
10    except Exception as e:
11        print(f"Ein Fehler ist beim Einlesen der Excel-Datei aufgetreten: {e}
")
12        return
13
14    # 2. Die Tabelle als Pandas DataFrame gespeichert (bereits in 'df')
15
16    # 3. Überprüfen, ob die notwendigen Spalten vorhanden sind
17    notwendige_spalten = ['Bezirke', 'Straftaten_insgesamt']
18    fehlende_spalten = [spalte for spalte in notwendige_spalten wenn spalte
nicht in df.columns]
19    wenn fehlende_spalten:
20        print(f"Die folgenden notwendigen Spalten fehlen: {'', '.join(
fehlende_spalten)}")
21        return
22
23    # 4. Finde die Gesamtzahl der Straftaten für ganz Berlin
24    gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
25    wenn gesamt_row.empty:
26        print("Die Zeile mit Bezirk 'Berlin (PKS gesamt)' wurde nicht
gefunden.")

```

```

27         return
28     gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].values[0]
29
30     # 5. Berechne den prozentualen Anteil der einzelnen Bezirke von ganz
    Berlin
31     # Exkludiere die Gesamtzeile aus der Berechnung
32     bezirke_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
33     bezirke_df['Prozentualer_Anteil'] = (bezirke_df['Straftaten_insgesamt'] /
    gesamt_straftaten) * 100
34
35     # Optional: Runden auf zwei Dezimalstellen
36     bezirke_df['Prozentualer_Anteil'] = bezirke_df['Prozentualer_Anteil'].
    round(2)
37
38     # 6. Das Ergebnis als DataFrame zurückgeben
39     ergebnis_df = bezirke_df[['Bezirke', 'Straftaten_insgesamt', '
    Prozentualer_Anteil']]
40
41     # Ergebnis anzeigen
42     print(ergebnis_df)
43
44 if __name__ == "__main__":
45     main()

```

Listing 45: Skript Testfall 3 Prompt 3 Ausführung 4

```

1 import pandas as pd
2
3 def berechne_prozentanteile(dateipfad: str, sheet_name: str = '
    Fallzahlen_2023') -> pd.DataFrame:
4     """
5     Liest eine Excel-Datei ein, überprüft notwendige Spalten, berechnet den
    prozentualen Anteil
6     der Straftaten pro Bezirk und gibt das Ergebnis als DataFrame zurück.
7
8     :param dateipfad: Pfad zur Excel-Datei (z.B. 'Fallzahlen.xlsx')
9     :param sheet_name: Name des Sheets in der Excel-Datei (Standard: '
    Fallzahlen_2023')
10    :return: DataFrame mit Bezirken und deren prozentualem Anteil an den
    Straftaten
11    """
12    # 1. Excel-Datei einlesen
13    try:
14        df = pd.read_excel(dateipfad, sheet_name=sheet_name)
15        print(f"Excel-Datei '{dateipfad}' erfolgreich eingelesen.")
16    except FileNotFoundError:
17        print(f"Fehler: Die Datei '{dateipfad}' wurde nicht gefunden.")
18        return
19    except Exception as e:
20        print(f"Ein Fehler ist beim Einlesen der Excel-Datei aufgetreten: {e}
    ")
21        return
22

```

```

23 # 2. DataFrame speichern (bereits durch pd.read_excel erledigt)
24
25 # 3. Überprüfen, ob die notwendigen Spalten vorhanden sind
26 notwendige_spalten = {'Bezirke', 'Straftaten_insgesamt'}
27 vorhandene_spalten = set(df.columns)
28
29 fehlende_spalten = notwendige_spalten - vorhandene_spalten
30 if fehlende_spalten:
31     print(f"Fehler: Die folgenden notwendigen Spalten fehlen in der
32     Tabelle: {fehlende_spalten}")
33     return
34 else:
35     print("Überprüfung der notwendigen Spalten erfolgreich.")
36
37 # 4. Gesamtzahl der Straftaten für ganz Berlin finden
38 gesamt_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
39 if gesamt_row.empty:
40     print("Fehler: Keine Zeile mit 'Bezirke' als 'Berlin (PKS gesamt)'
41     gefunden.")
42     return
43
44 gesamt_straftaten = gesamt_row['Straftaten_insgesamt'].iloc[0]
45 print(f"Gesamtzahl der Straftaten in Berlin: {gesamt_straftaten}")
46
47 # 5. Prozentualen Anteil der einzelnen Bezirke berechnen
48 # Ausschließen der Gesamtzeile aus den Bezirken
49 bezirke_df = df[df['Bezirke'] != 'Berlin (PKS gesamt)'].copy()
50
51 # Berechnung des prozentualen Anteils
52 bezirke_df['Prozentualer_Anteil'] = (bezirke_df['Straftaten_insgesamt'] /
53     gesamt_straftaten) * 100
54
55 # Optional: Auf zwei Dezimalstellen runden
56 bezirke_df['Prozentualer_Anteil'] = bezirke_df['Prozentualer_Anteil'].
57     round(2)
58
59 # 6. Ergebnis als DataFrame zurückgeben
60 ergebnis_df = bezirke_df[['Bezirke', 'Straftaten_insgesamt', '
61     Prozentualer_Anteil']]
62
63 return ergebnis_df
64
65 # Beispielhafte Nutzung des Skripts
66 if __name__ == "__main__":
67     dateipfad = 'Fallzahlen.xlsx' # Pfad zur Excel-Datei
68     ergebnis = berechne_prozentanteile(dateipfad)
69
70     if ergebnis is not None:
71         print("\nProzentuale Anteile der Straftaten pro Bezirk:")
72         print(ergebnis)
73
74     # Optional: Ergebnis in eine neue Excel-Datei speichern

```

```

70     # ergebnis.to_excel('Prozentuale_Anteile_Straftaten.xlsx', index=
        False)

```

Listing 46: Skript Testfall 3 Prompt 3 Ausführung 5

#### 8.2.4 Testfall 4

```

1  import pandas as pd
2
3  def extract_year(sheet_name):
4      """
5      Extrahiert das Jahr aus dem Sheetnamen.
6      Annahme: Der Sheetname enthält das Jahr als vierstellige Zahl, z.B. "2020
7      ", "2021".
8      """
9      import re
10     match = re.search(r'(\d{4})', sheet_name)
11     if match:
12         return int(match.group(1))
13     else:
14         raise ValueError(f"Kein Jahr im Sheetnamen '{sheet_name}' gefunden.")
15
16 def get_total_crimes(df):
17     """
18     Extrahiert die Gesamtzahl der Straftaten für Berlin aus dem DataFrame.
19     Annahme: Die Zeile mit 'Bezirke' == 'Berlin (PKS gesamt)' enthält die
20     Gesamtzahl.
21     """
22     total_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
23     if total_row.empty:
24         raise ValueError("Keine Zeile mit 'Berlin (PKS gesamt)' gefunden.")
25     return total_row['Straftaten_insgesamt'].values[0]
26
27 def main():
28     # Pfad zur Excel-Datei
29     excel_file = 'Fallzahlen.xlsx'
30
31     # Lade die Excel-Datei
32     try:
33         xls = pd.ExcelFile(excel_file)
34     except FileNotFoundError:
35         print(f"Datei '{excel_file}' wurde nicht gefunden.")
36         return
37     except Exception as e:
38         print(f"Fehler beim Laden der Datei: {e}")
39         return
40
41     # Liste der Sheets
42     sheet_names = xls.sheet_names
43
44     # Liste zur Speicherung der Ergebnisse
45     data = []

```



```

44
45     for sheet in sheet_names:
46         try:
47             # Lade das Sheet als DataFrame
48             df = pd.read_excel(xls, sheet_name=sheet)
49
50             # Extrahiere das Jahr
51             year = extract_year(sheet)
52
53             # Extrahiere die Gesamtzahl der Straftaten
54             total_crimes = get_total_crimes(df)
55
56             # Speichere das Ergebnis
57             data.append({'Jahr': year, 'Straftaten_insgesamt': total_crimes})
58
59         except Exception as e:
60             print(f"Fehler beim Verarbeiten des Sheets '{sheet}': {e}")
61             continue
62
63     # Erstelle einen DataFrame aus den gesammelten Daten
64     crimes_df = pd.DataFrame(data)
65
66     # Sortiere den DataFrame nach Jahr
67     crimes_df.sort_values('Jahr', inplace=True)
68     crimes_df.reset_index(drop=True, inplace=True)
69
70     # Berechne die prozentuale Veränderung zum Vorjahr
71     crimes_df['Prozentuale_Veraenderung'] = crimes_df['Straftaten_insgesamt']
72     ].pct_change() * 100
73
74     # Optional: Runde die Prozentwerte auf zwei Dezimalstellen
75     crimes_df['Prozentuale_Veraenderung'] = crimes_df['
76     Prozentuale_Veraenderung'].round(2)
77
78     # Setze das Jahr als Index (optional)
79     crimes_df.set_index('Jahr', inplace=True)
80
81     # Anzeige des resultierenden DataFrames
82     print(crimes_df)
83
84     # Optional: Speichern des DataFrames als CSV
85     # crimes_df.to_csv('Prozentuale_Veraenderung_Straftaten_Berlin.csv')
86
87 if __name__ == "__main__":
88     main()

```

Listing 47: Skript Testfall 4 Prompt 1 Ausführung 1

```

1 import pandas as pd
2
3 def extract_berlin_total(df):
4     """
5     Extrahiert die Gesamtzahl der Straftaten für Berlin aus dem DataFrame.

```

```

6
7     Es wird entweder nach dem 'LOR-Schlüssel' 999999 oder nach dem
      Bezirksnamen
8     'Berlin (PKS gesamt)' gesucht.
9
10    Args:
11        df (pd.DataFrame): Der DataFrame des aktuellen Sheets.
12
13    Returns:
14        int oder float: Die Gesamtzahl der Straftaten in Berlin.
15    """
16    # Versuch 1: Nach LOR-Schlüssel 999999 filtern
17    berlin_row = df[df['LOR-Schlüssel'] == 999999]
18
19    if not berlin_row.empty:
20        return berlin_row['Straftaten_insgesamt'].values[0]
21
22    # Versuch 2: Nach Bezirksnamen 'Berlin (PKS gesamt)' filtern
23    berlin_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
24
25    if not berlin_row.empty:
26        return berlin_row['Straftaten_insgesamt'].values[0]
27
28    # Wenn nichts gefunden wurde, gebe NaN zurück
29    return float('nan')
30
31 def main():
32     # Pfad zur Excel-Datei
33     excel_file = 'Fallzahlen.xlsx'
34
35     try:
36         # Lade alle Sheets der Excel-Datei
37         xls = pd.ExcelFile(excel_file)
38     except FileNotFoundError:
39         print(f"Die Datei {excel_file} wurde nicht gefunden.")
40         return
41     except Exception as e:
42         print(f"Fehler beim Laden der Excel-Datei: {e}")
43         return
44
45     # Liste zur Speicherung der Daten
46     data = []
47
48     for sheet_name in xls.sheet_names:
49         try:
50             # Lese das aktuelle Sheet
51             df = pd.read_excel(xls, sheet_name=sheet_name)
52
53             # Extrahiere die Gesamtzahl der Straftaten für Berlin
54             total_crimes = extract_berlin_total(df)
55
56             # Versuche, das Jahr aus dem Sheet-Namen zu extrahieren

```

```

57         try:
58             year = int(sheet_name)
59         except ValueError:
60             # Wenn der Sheet-Name kein Jahr ist, überspringe dieses Sheet
61             print(f"Sheet '{sheet_name}' entspricht nicht dem erwarteten
Jahresformat und wird übersprungen.")
62             continue
63
64         # Füge die Daten zur Liste hinzu
65         data.append({'Jahr': year, 'Straftaten_insgesamt': total_crimes})
66
67     except Exception as e:
68         print(f"Fehler beim Verarbeiten des Sheets '{sheet_name}': {e}")
69         continue
70
71     # Erstelle einen DataFrame aus den gesammelten Daten
72     df_totals = pd.DataFrame(data)
73
74     if df_totals.empty:
75         print("Keine gültigen Daten gefunden.")
76         return
77
78     # Sortiere den DataFrame nach Jahr
79     df_totals.sort_values('Jahr', inplace=True)
80
81     # Setze das Jahr als Index
82     df_totals.set_index('Jahr', inplace=True)
83
84     # Berechne die prozentuale Veränderung zum Vorjahr
85     df_totals['Prozentuale_Veraenderung'] = df_totals['Straftaten_insgesamt'
].pct_change() * 100
86
87     # Optional: Runde die Prozentwerte auf zwei Dezimalstellen
88     df_totals['Prozentuale_Veraenderung'] = df_totals['
Prozentuale_Veraenderung'].round(2)
89
90     # Anzeige des resultierenden DataFrames
91     print(df_totals)
92
93     # Optional: Speichere das Ergebnis in eine neue Excel-Datei
94     output_file = 'Straftaten_Zeitreihe.xlsx'
95     try:
96         df_totals.to_excel(output_file)
97         print(f"Das Ergebnis wurde erfolgreich in '{output_file}' gespeichert
.")
98     except Exception as e:
99         print(f"Fehler beim Speichern der Ergebnisdatei: {e}")
100
101 if __name__ == "__main__":
102     main()

```

Listing 48: Skript Testfall 4 Prompt 1 Ausführung 2

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_datei = 'Fallzahlen.xlsx'
5
6 # Lade alle Sheets in ein Dictionary
7 alle_sheets = pd.read_excel(excel_datei, sheet_name=None)
8
9 # Liste zur Speicherung der jährlichen Straftaten
10 jährliche_straftaten = []
11
12 for sheet_name, df in alle_sheets.items():
13     try:
14         # Versuche, das Jahr aus dem Sheetnamen zu extrahieren
15         jahr = int(sheet_name)
16     except ValueError:
17         # Wenn der Sheetname kein Jahr ist, überspringe dieses Sheet
18         print(f"Sheet '{sheet_name}' übersprungen (kein gültiges Jahr).")
19         continue
20
21     # Finde die Zeile für 'Berlin (PKS gesamt)'
22     berlin_gesamt = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
23
24     if nicht berlin_gesamt.empty:
25         # Extrahiere die Gesamtzahl der Straftaten
26         gesamt_straftaten = berlin_gesamt['Straftaten_insgesamt'].values[0]
27         jährliche_straftaten.append({'Jahr': jahr, 'Straftaten_insgesamt':
gesamt_straftaten})
28     else:
29         print(f"Keine Daten für 'Berlin (PKS gesamt)' in Sheet '{sheet_name}'
gefunden.")
30
31 # Erstelle einen DataFrame aus den gesammelten Daten
32 df_jährlich = pd.DataFrame(jährliche_straftaten)
33
34 # Sortiere den DataFrame nach Jahr
35 df_jährlich = df_jährlich.sort_values('Jahr').reset_index(drop=True)
36
37 # Berechne die prozentuale Veränderung zum Vorjahr
38 df_jährlich['Veränderung_prozent'] = df_jährlich['Straftaten_insgesamt'].
pct_change() * 100
39
40 # Runde die Veränderungen auf zwei Dezimalstellen
41 df_jährlich['Veränderung_prozent'] = df_jährlich['Veränderung_prozent'].round
(2)
42
43 # Anzeige des resultierenden DataFrames
44 print(df_jährlich)

```

Listing 49: Skript Testfall 4 Prompt 1 Ausführung 3

```

1 import pandas as pd

```

```

2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # Lese alle Sheets in der Excel-Datei
7 # sheet_name=None lädt alle Sheets und gibt ein Dictionary zurück
8 sheets_dict = pd.read_excel(excel_file, sheet_name=None)
9
10 # Liste zur Speicherung der jährlichen Gesamtstraftaten
11 data = []
12
13 # Iteriere über alle Sheets
14 for sheet_name, df in sheets_dict.items():
15     # Überprüfe, ob 'LOR-Schlüssel' und 'Straftaten_insgesamt' in den Spalten
16     # vorhanden sind
17     if 'LOR-Schlüssel' in df.columns and 'Straftaten_insgesamt' in df.columns
18     :
19         # Suche die Zeile, die die Gesamtzahl für Berlin enthält
20         # Annahme: 'LOR-Schlüssel' 999999 oder 'Berlin (PKS gesamt)' reprä
21         # sentiert die Gesamtdaten
22         total_row = df[df['LOR-Schlüssel'] == 999999]
23
24         # Falls nicht gefunden, versuche mit dem Namen
25         if total_row.empty:
26             total_row = df[df['LOR-Schlüssel'].astype(str).str.contains('
27 Berlin \ (PKS gesamt)\ ', regex=True)]
28
29         # Falls immer noch nicht gefunden, überspringe das Sheet
30         if total_row.empty:
31             print(f"Warnung: Gesamtdaten nicht in Sheet '{sheet_name}'
32 gefunden.")
33             continue
34
35         # Extrahiere das Jahr aus dem Sheet-Namen
36         # Annahme: Der Sheet-Name enthält das Jahr, z.B. "2020", "Jahr 2020",
37         # etc.
38         # Hier wird versucht, eine vierstellige Zahl zu extrahieren
39         import re
40         match = re.search(r'\b(19|20)\d{2}\b', sheet_name)
41         if match:
42             year = int(match.group())
43         else:
44             print(f"Warnung: Jahr konnte nicht aus dem Sheet-Namen '{
45 sheet_name}' extrahiert werden. Überspringe dieses Sheet.")
46             continue
47
48         # Extrahiere die Gesamtzahl der Straftaten
49         total_crimes = total_row['Straftaten_insgesamt'].values[0]
50
51         # Füge die Daten zur Liste hinzu
52         data.append({'Jahr': year, 'Straftaten_insgesamt': total_crimes})
53     else:

```

```

47     print(f"Warnung: Erforderliche Spalten nicht in Sheet '{sheet_name}'
48         vorhanden.")
49
50 # Erstelle einen DataFrame aus den gesammelten Daten
51 df_total = pd.DataFrame(data)
52
53 # Sortiere den DataFrame nach dem Jahr
54 df_total = df_total.sort_values('Jahr').reset_index(drop=True)
55
56 # Berechne die prozentuale Veränderung zum Vorjahr
57 df_total['Prozentuale_Veraenderung_zum_Vorjahr (%)'] = df_total['
58     Straftaten_insgesamt'].pct_change() * 100
59
60 # Optional: Rundung der prozentualen Veränderung auf zwei Dezimalstellen
61 df_total['Prozentuale_Veraenderung_zum_Vorjahr (%)'] = df_total['
62     Prozentuale_Veraenderung_zum_Vorjahr (%)'].round(2)
63
64 # Setze das Jahr als Index (optional)
65 df_total.set_index('Jahr', inplace=True)
66
67 # Anzeige des Ergebnis-DataFrames
68 print(df_total)
69
70 # Optional: Speichere das Ergebnis in eine neue Excel-Datei
71 df_total.to_excel('Zeitreihe_Straftaten_Berlin.xlsx')

```

Listing 50: Skript Testfall 4 Prompt 1 Ausführung 4

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # Excel-Datei laden
7 xls = pd.ExcelFile(excel_file)
8
9 # Liste zur Speicherung der jährlichen Gesamtdaten
10 jahresdaten = []
11
12 # Annahme: Die Sheet-Namen entsprechen den Jahreszahlen (z.B. '2018', '2019',
13     ...)
14 for sheet in xls.sheet_names:
15     # Daten des aktuellen Sheets lesen
16     df = pd.read_excel(xls, sheet_name=sheet)
17
18     # Sicherstellen, dass die 'LOR-Schlüssel' Spalte als String behandelt
19     # wird
20     df['LOR-Schlüssel'] = df['LOR-Schlüssel'].astype(str)
21
22     # Filtern nach der Zeile für ganz Berlin (angenommen '999999' ist der
23     # Schlüssel)
24     berlin_gesamt = df[df['LOR-Schlüssel'] == '999999']

```

```

23     if not berlin_gesamt.empty:
24         # Gesamtzahl der Straftaten für Berlin aus der entsprechenden Zeile
25         straftaten = berlin_gesamt['Straftaten_insgesamt'].values[0]
26     else:
27         # Wenn die 'Berlin (PKS gesamt)' Zeile nicht vorhanden ist, summiere
28         # alle Bezirke
29         straftaten = df['Straftaten_insgesamt'].sum()
30
31     # Füge die Daten zur Liste hinzu
32     jahresdaten.append({'Jahr': sheet, 'Straftaten_insgesamt': straftaten})
33
34 # Erstelle einen DataFrame aus den gesammelten Daten
35 df_gesamt = pd.DataFrame(jahresdaten)
36
37 # Versuche, die 'Jahr' Spalte in Integer zu konvertieren für die Sortierung
38 try:
39     df_gesamt['Jahr'] = df_gesamt['Jahr'].astype(int)
40     df_gesamt = df_gesamt.sort_values('Jahr')
41 except ValueError:
42     # Falls die Sheet-Namen nicht numerisch sind, sortiere alphabetisch
43     df_gesamt = df_gesamt.sort_values('Jahr')
44
45 # Setze das Jahr als Index
46 df_gesamt.set_index('Jahr', inplace=True)
47
48 # Berechne die prozentuale Veränderung zum Vorjahr
49 df_gesamt['% Veränderung Vorjahr'] = df_gesamt['Straftaten_insgesamt'].
50     pct_change() * 100
51
52 # Runde die Prozentwerte auf zwei Dezimalstellen
53 df_gesamt['% Veränderung Vorjahr'] = df_gesamt['% Veränderung Vorjahr'].round
54     (2)
55
56 # Anzeige des resultierenden DataFrames
57 print(df_gesamt)
58
59 # Optional: Speichern des DataFrames in eine neue Excel-Datei
60 df_gesamt.to_excel('Prozentuale_Veraenderung_Straftaten_Berlin.xlsx')

```

Listing 51: Skript Testfall 4 Prompt 1 Ausführung 5

```

1 import pandas as pd
2
3 # Pfad zur Excel-Datei
4 excel_file = 'Fallzahlen.xlsx'
5
6 # Lade alle Sheet-Namen
7 xls = pd.ExcelFile(excel_file)
8 sheets = xls.sheet_names
9
10 # Extrahiere das Jahr aus den Sheet-Namen und sortiere die Sheets nach Jahr
11 sheet_years = []
12 for sheet in sheets:

```

```

13     try:
14         # Erwartetes Format: 'Fallzahlen_2014', 'Fallzahlen_2015', etc.
15         year = int(sheet.split('_')[-1])
16         sheet_years.append((year, sheet))
17     except ValueError:
18         print(f"Sheet-Name '{sheet}' enthält kein gültiges Jahr und wird ü
19 bersprungen.")
20         continue
21
22 # Sortiere die Sheets nach Jahr
23 sheet_years_sorted = sorted(sheet_years, key=lambda x: x[0])
24
25 # Liste zur Speicherung der Ergebnisse
26 data = []
27
28 # Iteriere über jedes Sheet und extrahiere die benötigten Daten
29 for year, sheet in sheet_years_sorted:
30     # Lese das aktuelle Sheet
31     df = pd.read_excel(excel_file, sheet_name=sheet)
32
33     # Stelle sicher, dass die benötigten Spalten vorhanden sind
34     if 'Bezirke' not in df.columns or 'Straftaten_insgesamt' not in df.
35 columns:
36         print(f"Sheet '{sheet}' fehlt die erforderliche Spalte. Überspringe
37 dieses Sheet.")
38         continue
39
40     # Filtere die Zeile für "Berlin (PKS gesamt)"
41     row = df[df['Bezirke'] == "Berlin (PKS gesamt)"]
42
43     if not row.empty:
44         # Extrahiere den Wert und stelle sicher, dass er numerisch ist
45         value = row['Straftaten_insgesamt'].values[0]
46         if pd.notnull(value):
47             data.append({'Year': year, 'Straftaten_insgesamt': value})
48         else:
49             print(f"Der Wert für 'Straftaten_insgesamt' im Jahr {year} ist
50 NaN.")
51         else:
52             print(f"'Berlin (PKS gesamt)' nicht in Sheet '{sheet}' gefunden.")
53
54 # Erstelle ein DataFrame aus den gesammelten Daten
55 df_data = pd.DataFrame(data)
56
57 # Sortiere das DataFrame nach Jahr, um sicherzustellen, dass die Reihenfolge
58 stimmt
59 df_data = df_data.sort_values('Year').reset_index(drop=True)
60
61 # Berechne die prozentuale Veränderung zum Vorjahr
62 df_data['Prozentuale_Veraenderung'] = df_data['Straftaten_insgesamt'].
63     pct_change() * 100

```



```

59 # Optional: Runde die Prozentwerte auf zwei Dezimalstellen
60 df_data['Prozentuale_Veraenderung'] = df_data['Prozentuale_Veraenderung'].
    round(2)
61
62 # Anzeige des Ergebnisses
63 print(df_data)
64
65 # Optional: Speichere das Ergebnis in eine neue Excel-Datei oder CSV
66 # df_data.to_excel('Prozentuale_Veraenderung_Fallzahlen.xlsx', index=False)
67 # df_data.to_csv('Prozentuale_Veraenderung_Fallzahlen.csv', index=False)

```

Listing 52: Skript Testfall 4 Prompt 2 Ausführung 1

```

1 import pandas as pd
2
3 def compute_percentage_change(excel_file, target_row, target_column,
    sheet_prefix):
4     """
5     Liest eine Excel-Datei mit mehreren Sheets, extrahiert die
    Straftaten_insgesamt für
6     einen bestimmten Eintrag und berechnet die prozentuale Veränderung zum
    Vorjahr.
7
8     Parameters:
9     - excel_file: str, Pfad zur Excel-Datei.
10    - target_row: str, Der Eintrag in der 'Bezirke' Spalte, z.B. "Berlin (PKS
    gesamt)".
11    - target_column: str, Name der Spalte, deren prozentuale Veränderung
    berechnet werden soll, z.B. "Straftaten_insgesamt".
12    - sheet_prefix: str, Präfix der Sheet-Namen, z.B. "Fallzahlen_" für
    Sheets wie "Fallzahlen_2014".
13
14    Returns:
15    - pandas.DataFrame mit den Jahren und der prozentualen Veränderung.
16    """
17    try:
18        # Alle Sheets einlesen
19        sheets_dict = pd.read_excel(excel_file, sheet_name=None, engine='
    openpyxl')
20    except FileNotFoundError:
21        print(f"Die Datei {excel_file} wurde nicht gefunden.")
22        return None
23    except Exception as e:
24        print(f"Ein Fehler ist aufgetreten beim Einlesen der Datei: {e}")
25        return None
26
27    data = []
28
29    for sheet_name, df in sheets_dict.items():
30        # Überprüfen, ob der Sheet-Name dem erwarteten Muster entspricht
31        if not sheet_name.startswith(sheet_prefix):
32            print(f"Überspringe Sheet '{sheet_name}', da es nicht mit '{
    sheet_prefix}' beginnt.")

```

```

33         continue
34
35     # Extrahiere das Jahr aus dem Sheet-Namen
36     try:
37         year_str = sheet_name.replace(sheet_prefix, "")
38         year = int(year_str)
39     except ValueError:
40         print(f"Konnte das Jahr aus dem Sheet-Namen '{sheet_name}' nicht
extrahieren.")
41         continue
42
43     # Suche die Zeile mit dem gewünschten Eintrag
44     row = df[df['Bezirke'] == target_row]
45
46     if row.empty:
47         print(f"Der Eintrag '{target_row}' wurde in Sheet '{sheet_name}'
nicht gefunden.")
48         continue
49
50     # Extrahiere den Wert der Zielspalte
51     try:
52         value = row.iloc[0][target_column]
53         data.append({'Year': year, target_column: value})
54     except KeyError:
55         print(f"Die Spalte '{target_column}' wurde in Sheet '{sheet_name
}' nicht gefunden.")
56         continue
57
58     if not data:
59         print("Keine Daten gefunden, um die prozentuale Veränderung zu
berechnen.")
60         return None
61
62     # Erstelle einen DataFrame aus den gesammelten Daten
63     df_data = pd.DataFrame(data)
64
65     # Sortiere die Daten nach Jahr
66     df_data = df_data.sort_values('Year').reset_index(drop=True)
67
68     # Berechne die prozentuale Veränderung zum Vorjahr
69     df_data['Percentage_Change'] = df_data[target_column].pct_change() * 100
70
71     return df_data[['Year', 'Percentage_Change']]
72
73 if __name__ == "__main__":
74     # Parameter festlegen
75     excel_file = 'Fallzahlen.xlsx'
76     target_row = 'Berlin (PKS gesamt)'
77     target_column = 'Straftaten_insgesamt'
78     sheet_prefix = 'Fallzahlen_'
79
80     # Funktion aufrufen

```

```

81     percentage_change_df = compute_percentage_change(excel_file, target_row,
82     target_column, sheet_prefix)
83
84     if percentage_change_df is not None:
85         print("Prozentuale Veränderung der Straftaten_insgesamt zum Vorjahr:")
86     )
87     print(percentage_change_df)
88
89     # Optional: DataFrame speichern
90     # percentage_change_df.to_csv('prozentuale_veraenderung.csv', index=
91     False)

```

Listing 53: Skript Testfall 4 Prompt 2 Ausführung 2

```

1  import pandas as pd
2
3  # Pfad zur Excel-Datei
4  excel_file = 'Fallzahlen.xlsx'
5
6  # Lese alle Sheetnamen aus der Excel-Datei
7  xls = pd.ExcelFile(excel_file)
8  sheet_names = xls.sheet_names
9
10 # Initialisiere ein Dictionary, um die Daten zu speichern
11 data = {}
12
13 # Iteriere durch alle Sheets
14 for sheet in sheet_names:
15     try:
16         # Extrahiere das Jahr aus dem Sheetnamen, z.B. 'Fallzahlen_2014' ->
17         2014
18         year = int(sheet.split('_')[-1])
19     except ValueError:
20         print(f"Sheet '{sheet}' folgt nicht dem erwarteten Namensformat. Ü
21         berspringe...")
22         continue
23
24     # Lese das aktuelle Sheet
25     df = pd.read_excel(xls, sheet_name=sheet)
26
27     # Filtere die Zeile für 'Berlin (PKS gesamt)'
28     berlin_data = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
29
30     if berlin_data.empty:
31         print(f"Keine Daten für 'Berlin (PKS gesamt)' im Sheet '{sheet}'
32         gefunden.")
33         continue
34
35     # Extrahiere den Wert der Spalte 'Straftaten_insgesamt'
36     total_crimes = berlin_data['Straftaten_insgesamt'].values[0]
37
38     # Speichere die Daten im Dictionary
39     data[year] = total_crimes

```

```

37
38 # Erstelle einen DataFrame aus dem Dictionary
39 df_crimes = pd.DataFrame(list(data.items()), columns=['Jahr', '
    Straftaten_insgesamt'])
40
41 # Sortiere den DataFrame nach Jahr
42 df_crimes = df_crimes.sort_values('Jahr').reset_index(drop=True)
43
44 # Berechne die prozentuale Veränderung zum Vorjahr
45 df_crimes['Prozentuale_Veränderung (%)'] = df_crimes['Straftaten_insgesamt'].
    pct_change() * 100
46
47 # Optional: Runde die prozentuale Veränderung auf zwei Dezimalstellen
48 df_crimes['Prozentuale_Veränderung (%)'] = df_crimes['Prozentuale_Veränderung
    (%)'].round(2)
49
50 # Zeige den Ergebnis-DataFrame
51 print(df_crimes)
52
53 # Optional: Speichere den DataFrame in eine neue Excel-Datei
54 df_crimes.to_excel('Straftaten_Veränderung_Zeitreihe.xlsx', index=False)

```

Listing 54: Skript Testfall 4 Prompt 2 Ausführung 3

```

1 import pandas as pd
2
3 def analyze_fallzahlen(file_path, target_district="Berlin (PKS gesamt)",
4     column_name="Straftaten_insgesamt"):
5     """
6     Analysiert die Excel-Datei 'Fallzahlen.xlsx' und erstellt eine Zeitreihe
7     mit
8     der prozentualen Veränderung der Straftaten zum jeweiligen Vorjahr.
9
10    Parameters:
11    - file_path: Pfad zur Excel-Datei
12    - target_district: Name des Bezirks, der analysiert werden soll
13    - column_name: Name der Spalte, deren Veränderung berechnet werden soll
14
15    Returns:
16    - pandas DataFrame mit den Jahreszahlen und der prozentualen Veränderung
17    """
18    # Lade die Excel-Datei
19    try:
20        xls = pd.ExcelFile(file_path)
21    except FileNotFoundError:
22        print(f"Die Datei {file_path} wurde nicht gefunden.")
23        return
24    except Exception as e:
25        print(f"Ein Fehler ist aufgetreten beim Laden der Datei: {e}")
26        return
27
28    # Initialisiere ein Dictionary zur Speicherung der Ergebnisse

```

```

29     data = {}
30
31     # Iteriere über alle Sheets in der Excel-Datei
32     for sheet in xls.sheet_names:
33         # Extrahiere das Jahr aus dem Sheet-Namen
34         try:
35             year = int(sheet.split('_')[-1])
36         except ValueError:
37             print(f"Das Sheet '{sheet}' folgt nicht dem erwarteten
Namensformat. Überspringe dieses Sheet.")
38             continue
39
40         # Lese das aktuelle Sheet in einen DataFrame
41         try:
42             df = pd.read_excel(xls, sheet_name=sheet)
43         except Exception as e:
44             print(f"Ein Fehler ist aufgetreten beim Lesen des Sheets '{sheet
}': {e}")
45             continue
46
47         # Finde die Zeile für den Zielbezirk
48         row = df[df['Bezirke'] == target_district]
49
50         if row.empty:
51             print(f"Der Bezirk '{target_district}' wurde im Sheet '{sheet}'
nicht gefunden. Überspringe dieses Sheet.")
52             continue
53
54         # Extrahiere den Wert der gewünschten Spalte
55         try:
56             value = row[column_name].values[0]
57             # Stelle sicher, dass der Wert numerisch ist
58             value = float(value)
59         except (IndexError, ValueError) as e:
60             print(f"Fehler beim Extrahieren des Wertes in Sheet '{sheet}': {e
}")
61             continue
62
63         # Speichere das Jahr und den Wert im Dictionary
64         data[year] = value
65
66     # Erstelle einen DataFrame aus dem Dictionary
67     df_data = pd.DataFrame(list(data.items()), columns=['Year', column_name])
68
69     # Sortiere den DataFrame nach dem Jahr aufsteigend
70     df_data.sort_values('Year', inplace=True)
71     df_data.reset_index(drop=True, inplace=True)
72
73     # Berechne die prozentuale Veränderung zum Vorjahr
74     df_data['Percent Change'] = df_data[column_name].pct_change() * 100
75
76     # Optional: Runde die Prozentänderungen auf zwei Dezimalstellen

```

```

77     df_data['Percent Change'] = df_data['Percent Change'].round(2)
78
79     # Setze das Jahr als Index (optional)
80     # df_data.set_index('Year', inplace=True)
81
82     return df_data
83
84 if __name__ == "__main__":
85     # Pfad zur Excel-Datei
86     excel_file = 'Fallzahlen.xlsx'
87
88     # Führe die Analyse durch
89     result_df = analyze_fallzahlen(excel_file)
90
91     if result_df is not None:
92         # Zeige das Ergebnis an
93         print(result_df)
94
95         # Optional: Speichere das Ergebnis in einer neuen Excel- oder CSV-
96         # Datei
97         # result_df.to_excel('Straftaten_Prozentuale_Veraenderung.xlsx',
98         # index=False)
99         # oder
100        # result_df.to_csv('Straftaten_Prozentuale_Veraenderung.csv', index=
101        False)

```

Listing 55: Skript Testfall 4 Prompt 2 Ausführung 4

```

1 import pandas as pd
2 import re
3
4 def extract_year(sheet_name):
5     """
6     Extrahiert das Jahr aus dem Sheet-Namen.
7     Erwartet Format: 'Fallzahlen_YYYY'
8     """
9     match = re.search(r'Fallzahlen_(\d{4})', sheet_name)
10    if match:
11        return int(match.group(1))
12    else:
13        return None
14
15 def get_straftaten_insgesamt(excel_file, sheet_name, target_district):
16     """
17     Liest ein Sheet und extrahiert den Wert von 'Straftaten_insgesamt' für
18     den angegebenen Bezirk.
19     """
20    df = pd.read_excel(excel_file, sheet_name=sheet_name)
21
22    # Annahme: Die Bezeichnung des Bezirks ist in der Spalte 'Bezirke'
23    row = df[df['Bezirke'] == target_district]
24
25    if not row.empty:

```

```

25         return row.iloc[0]['Straftaten_insgesamt']
26     else:
27         print(f"Warnung: Bezirk '{target_district}' nicht in Sheet '{
sheet_name}' gefunden.")
28         return None
29
30 def main():
31     excel_file = 'Fallzahlen.xlsx'
32     target_district = 'Berlin (PKS gesamt)'
33
34     # Lade alle Sheet-Namen
35     xls = pd.ExcelFile(excel_file)
36     sheet_names = xls.sheet_names
37
38     # Filtere die relevanten Sheets und extrahiere die Jahre
39     pattern = re.compile(r'^Fallzahlen_(\d{4})$')
40     sheets_with_year = []
41     for sheet in sheet_names:
42         match = pattern.match(sheet)
43         if match:
44             year = int(match.group(1))
45             sheets_with_year.append((year, sheet))
46
47     if not sheets_with_year:
48         print("Keine Sheets im erwarteten Format 'Fallzahlen_YYYY' gefunden."
)
49         return
50
51     # Sortiere die Sheets nach Jahr
52     sheets_with_year.sort(key=lambda x: x[0])
53
54     # Extrahiere die 'Straftaten_insgesamt' Werte
55     data = {}
56     for year, sheet in sheets_with_year:
57         value = get_straftaten_insgesamt(excel_file, sheet, target_district)
58         if value is not None:
59             data[year] = value
60
61     # Erstelle ein DataFrame
62     df = pd.DataFrame(list(data.items()), columns=['Jahr', '
Straftaten_insgesamt'])
63     df.sort_values('Jahr', inplace=True)
64     df.reset_index(drop=True, inplace=True)
65
66     # Berechne die prozentuale Veränderung zum Vorjahr
67     df['Prozentuale_Veraenderung'] = df['Straftaten_insgesamt'].pct_change()
* 100
68
69     # Optional: Runde die Prozentwerte auf zwei Dezimalstellen
70     df['Prozentuale_Veraenderung'] = df['Prozentuale_Veraenderung'].round(2)
71
72     print(df)

```

```

73
74 if __name__ == "__main__":
75     main()

```

Listing 56: Skript Testfall 4 Prompt 2 Ausführung 5

```

1 import pandas as pd
2
3 # Schritt 1: Alle Sheets der Excel-Datei einlesen
4 excel_datei = 'Fallzahlen.xlsx'
5 xls = pd.ExcelFile(excel_datei)
6
7 # Dictionary zum Speichern der DataFrames
8 dataframes = {}
9
10 # Extrahiere die Jahreszahlen aus den Sheet-Namen und sortiere sie
11 sheet_jahre = []
12 for sheet in xls.sheet_names:
13     try:
14         # Annahme: Sheetnamen sind im Format 'Fallzahlen_Jahr', z.B. '
15         Fallzahlen_2014'
16         jahr = int(sheet.split('_')[-1])
17         sheet_jahre.append((jahr, sheet))
18     except ValueError:
19         print(f"Sheet '{sheet}' hat kein gültiges Jahresformat und wird ü
20         bersprungen.")
21
22 # Sortiere die Sheets nach Jahr
23 sheet_jahre.sort()
24
25 # Lade die DataFrames und speichere sie im Dictionary
26 for jahr, sheet in sheet_jahre:
27     df = pd.read_excel(xls, sheet_name=sheet)
28     dataframes[jahr] = df
29
30 # Schritt 2: Extrahiere 'Straftaten_insgesamt' für 'Berlin (PKS gesamt)'
31 straftaten_gesamt = {}
32
33 for jahr, df in dataframes.items():
34     # Filtern der Zeile 'Berlin (PKS gesamt)'
35     berlin_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
36     if not berlin_row.empty:
37         wert = berlin_row.iloc[0]['Straftaten_insgesamt']
38         straftaten_gesamt[jahr] = wert
39     else:
40         print(f"Die Zeile 'Berlin (PKS gesamt)' wurde im Jahr {jahr} nicht
41         gefunden.")
42
43 # Schritt 3: Berechne die prozentuale Veränderung zum Vorjahr
44 jahre = sorted(straftaten_gesamt.keys())
45 prozentuale_veraenderung = {}
46
47 for i in range(1, len(jahre)):

```



```

45     aktuelles_jahr = jahre[i]
46     vorheriges_jahr = jahre[i-1]
47     wert_aktuell = straftaten_gesamt[aktuelles_jahr]
48     wert_vorher = straftaten_gesamt[vorheriges_jahr]
49     veraenderung = ((wert_aktuell - wert_vorher) / wert_vorher) * 100
50     prozentuale_veraenderung[aktuelles_jahr] = veraenderung
51
52 # Schritt 4: Ergebnisse in einem neuen DataFrame speichern
53 ergebnisse = pd.DataFrame({
54     'Jahr': list(prozentuale_veraenderung.keys()),
55     'Prozentuale Veränderung (%)': list(prozentuale_veraenderung.values())
56 })
57
58 # Optional: Prozentuale Veränderung mit zwei Dezimalstellen formatieren
59 ergebnisse['Prozentuale Veränderung (%)'] = ergebnisse['Prozentuale Verä
    nderung (%)'].round(2)
60
61 # Ergebnisse anzeigen
62 print(ergebnisse)
63
64 # Optional: Ergebnisse in eine neue Excel-Datei speichern
65 ergebnisse.to_excel('Prozentuale_Veraenderung_Fallzahlen.xlsx', index=False)

```

Listing 57: Skript Testfall 4 Prompt 3 Ausführung 1

```

1 import pandas as pd
2
3 # Schritt 1: Alle Sheets der Excel-Datei einlesen
4 excel_datei = 'Fallzahlen.xlsx'
5 xls = pd.read_excel(excel_datei, sheet_name=None)
6
7 # Dictionary zum Speichern der 'Straftaten_insgesamt'-Werte pro Jahr
8 straftaten_dict = {}
9
10 # Schleife durch jedes Sheet
11 for sheet_name, df in xls.items():
12     # Extrahiere das Jahr aus dem Sheet-Namen, z.B. 'Fallzahlen_2014' -> 2014
13     try:
14         jahr = int(sheet_name.split('_')[-1])
15     except ValueError:
16         print(f"Sheet '{sheet_name}' entspricht nicht dem erwarteten Muster
    und wird übersprungen.")
17         continue
18
19     # Schritt 2: Wert der Spalte 'Straftaten_insgesamt' für 'Berlin (PKS
    gesamt)' extrahieren
20     try:
21         wert = df.loc[df['Bezirke'] == 'Berlin (PKS gesamt)', '
    Straftaten_insgesamt'].values[0]
22         straftaten_dict[jahr] = wert
23     except IndexError:
24         print(f"'Berlin (PKS gesamt)' wurde in Sheet '{sheet_name}' nicht
    gefunden.")

```

```

25     straftaten_dict[jahr] = None
26
27 # Umwandlung des Dictionaries in ein sortiertes DataFrame
28 straftaten_series = pd.Series(straftaten_dict).sort_index()
29 straftaten_df = straftaten_series.reset_index()
30 straftaten_df.columns = ['Jahr', 'Straftaten_insgesamt']
31
32 # Schritt 3: Prozentuale Veränderung zum Vorjahr berechnen
33 straftaten_df['Prozentuale_Veraenderung'] = straftaten_df['
    Straftaten_insgesamt'].pct_change() * 100
34
35 # Optional: Prozentuale Veränderung auf zwei Dezimalstellen runden
36 straftaten_df['Prozentuale_Veraenderung'] = straftaten_df['
    Prozentuale_Veraenderung'].round(2)
37
38 # Schritt 4: Ergebnisse in einem neuen DataFrame speichern
39 ergebnisse_df = straftaten_df[['Jahr', 'Prozentuale_Veraenderung']]
40
41 # Ausgabe der Ergebnisse
42 print(ergebnisse_df)
43
44 # Optional: Ergebnisse in eine neue Excel-Datei speichern
45 # ergebnisse_df.to_excel('Prozentuale_Veraenderung.xlsx', index=False)

```

Listing 58: Skript Testfall 4 Prompt 3 Ausführung 2

```

1 import pandas as pd
2
3 def main():
4     # Pfad zur Excel-Datei
5     excel_file = 'Fallzahlen.xlsx'
6
7     # Lade alle Sheet-Namen aus der Excel-Datei
8     xls = pd.ExcelFile(excel_file)
9     sheet_names = xls.sheet_names
10
11     # Dictionary zum Speichern der DataFrames
12     dataframes = {}
13
14     # Dictionary zum Speichern der 'Straftaten_insgesamt' Werte
15     strafaten_dict = {}
16
17     for sheet in sheet_names:
18         # Lese jedes Sheet in einen DataFrame
19         df = pd.read_excel(excel_file, sheet_name=sheet)
20         dataframes[sheet] = df
21
22         # Extrahiere das Jahr aus dem Sheet-Namen (angenommen, Format '
        Fallzahlen_YYYY')
23         try:
24             year = int(sheet.split('_')[-1])
25         except ValueError:
26             print(f"Sheet-Name {sheet} entspricht nicht dem erwarteten Format

```

```

    'Fallzahlen_YYYY'.")
27         continue
28
29     # Finde die Zeile für 'Berlin (PKS gesamt)'
30     berlin_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
31
32     if not berlin_row.empty:
33         # Extrahiere den Wert von 'Straftaten_insgesamt'
34         strafaten_value = berlin_row['Straftaten_insgesamt'].values[0]
35         strafaten_dict[year] = strafaten_value
36     else:
37         print(f"'Berlin (PKS gesamt)' nicht im Sheet {sheet} gefunden.")
38         strafaten_dict[year] = None
39
40     # Erstelle einen DataFrame aus den gesammelten Daten
41     strafaten_df = pd.DataFrame(list(strafaten_dict.items()), columns=['Jahr',
42     'Straftaten_insgesamt'])
43     strafaten_df.sort_values('Jahr', inplace=True)
44     strafaten_df.reset_index(drop=True, inplace=True)
45
46     # Berechne die prozentuale Veränderung zum Vorjahr
47     strafaten_df['Prozentuale_Veraenderung'] = strafaten_df['
48     Straftaten_insgesamt'].pct_change() * 100
49
50     # Optional: Runde die Veränderung auf 2 Dezimalstellen
51     strafaten_df['Prozentuale_Veraenderung'] = strafaten_df['
52     Prozentuale_Veraenderung'].round(2)
53
54     # Anzeige der Ergebnisse
55     print(strafaten_df)
56
57     # Optional: Speichern der Ergebnisse in eine neue Excel-Datei
58     output_file = 'Straftaten_Veraenderung.xlsx'
59     strafaten_df.to_excel(output_file, index=False)
60     print(f"Ergebnisse wurden in '{output_file}' gespeichert.")
61
62 if __name__ == "__main__":
63     main()

```

Listing 59: Skript Testfall 4 Prompt 3 Ausführung 3

```

1 import pandas as pd
2
3 # 1. Lesen Sie alle Sheets der Excel-Datei ein und speichern Sie jedes Sheet
4   in einem separaten DataFrame
5 excel_file = 'Fallzahlen.xlsx'
6 xls = pd.read_excel(excel_file, sheet_name=None) # Liest alle Sheets als ein
7   Dictionary
8
9 # Dictionary zur Speicherung der 'Straftaten_insgesamt' Werte pro Jahr
10 strafaten_dict = {}
11
12 for sheet_name, df in xls.items():

```

```

11 # Extrahiere das Jahr aus dem Sheet-Namen, z.B. 'Fallzahlen_2014' -> 2014
12 try:
13     year = int(sheet_name.split('_')[-1])
14 except ValueError:
15     print(f"Sheet-Name '{sheet_name}' entspricht nicht dem erwarteten
16     Muster 'Fallzahlen_Jahr'. Übersprungen.")
17     continue
18
19 # 2. Extrahiere den Wert der Spalte 'Straftaten_insgesamt' für die Zeile
20 # 'Berlin (PKS gesamt)'
21 # Überprüfen Sie, ob die Spaltennamen korrekt sind. Möglicherweise müssen
22 # Sie 'Bezirke' anpassen.
23 # Hier wird angenommen, dass die Spalte mit Bezirken 'Bezirke' heißt
24 berlin_row = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
25
26 if not berlin_row.empty:
27     straf_value = berlin_row['Straftaten_insgesamt'].values[0]
28     straftaten_dict[year] = straf_value
29 else:
30     print(f"'Berlin (PKS gesamt)' nicht in Sheet '{sheet_name}' gefunden.
31     ")
32
33 # 3. Berechnen Sie die prozentuale Veränderung zum jeweiligen Vorjahr
34 # Sortieren der Jahre
35 sorted_years = sorted(straftaten_dict.keys())
36 percentage_changes = []
37
38 for i in range(1, len(sorted_years)):
39     current_year = sorted_years[i]
40     previous_year = sorted_years[i - 1]
41     current_value = straftaten_dict[current_year]
42     previous_value = straftaten_dict[previous_year]
43
44     # Prozentuale Veränderung berechnen
45     if previous_value != 0:
46         percent_change = ((current_value - previous_value) / previous_value)
47         * 100
48     else:
49         percent_change = None # Vermeiden von Division durch Null
50
51     percentage_changes.append({
52         'Jahr': current_year,
53         'Prozentuale Veränderung': percent_change
54     })
55
56 # 4. Speichern Sie die Ergebnisse in einem neuen DataFrame
57 result_df = pd.DataFrame(percentage_changes)
58
59 # Optional: Hinzufügen des ersten Jahres mit einer NaN für die Veränderung
60 result_df = pd.concat([
61     pd.DataFrame({'Jahr': [sorted_years[0]], 'Prozentuale Veränderung': [None]}),

```

```

57     result_df
58 ], ignore_index=True)
59
60 # Anzeige des Ergebnisses
61 print(result_df)
62
63 # Optional: Speichern des Ergebnisses in eine neue Excel-Datei
64 result_df.to_excel('Prozentuale_Veraenderung_Straftaten.xlsx', index=False)

```

Listing 60: Skript Testfall 4 Prompt 3 Ausführung 4

```

1  import pandas as pd
2
3  # 1. Lesen Sie alle Sheets der Excel-Datei ein
4  excel_file = 'Fallzahlen.xlsx'
5  all_sheets = pd.read_excel(excel_file, sheet_name=None)
6
7  # Liste zur Speicherung der Ergebnisse
8  daten_liste = []
9
10 # 2. Extrahieren Sie den Wert der Spalte 'Straftaten_insgesamt' für 'Berlin (
    PKS gesamt)'
11 for sheet_name, df in all_sheets.items():
12     # Extrahieren Sie das Jahr aus dem Sheet-Namen, z.B. 'Fallzahlen_2014' ->
        2014
13     try:
14         jahr = int(sheet_name.split('_')[-1])
15     except ValueError:
16         print(f"Das Sheet '{sheet_name}' hat keinen gültigen Jahresnamen und
            wird übersprungen.")
17         continue
18
19     # Suche nach der Zeile 'Berlin (PKS gesamt)'
20     berlin_data = df[df['Bezirke'] == 'Berlin (PKS gesamt)']
21
22     if berlin_data.empty:
23         print(f"'Berlin (PKS gesamt)' wurde im Sheet '{sheet_name}' nicht
            gefunden.")
24         continue
25
26     # Extrahiere den Wert der Spalte 'Straftaten_insgesamt'
27     strftaten_insgesamt = berlin_data['Straftaten_insgesamt'].values[0]
28
29     daten_liste.append({'Jahr': jahr, 'Straftaten_insgesamt':
        strftaten_insgesamt})
30
31 # 3. Erstellen Sie ein DataFrame aus den gesammelten Daten
32 daten_df = pd.DataFrame(daten_liste)
33
34 # Sortieren Sie das DataFrame nach Jahr aufsteigend
35 daten_df = daten_df.sort_values('Jahr').reset_index(drop=True)
36
37 # 4. Berechnen Sie die prozentuale Veränderung zum Vorjahr

```

```

38 daten_df['Prozentuale_Veraenderung'] = daten_df['Straftaten_insgesamt'].
    pct_change() * 100
39
40 # Optional: Runden Sie die Prozentsätze auf zwei Dezimalstellen
41 daten_df['Prozentuale_Veraenderung'] = daten_df['Prozentuale_Veraenderung'].
    round(2)
42
43 # 5. Erstellen Sie das Endergebnis DataFrame
44 ergebnis_df = daten_df[['Jahr', 'Prozentuale_Veraenderung']]
45
46 # Optional: Benennen Sie die Spalten auf Deutsch um
47 ergebnis_df.columns = ['Jahr', 'Prozentuale Veränderung (%)']
48
49 # 6. Anzeigen des Ergebnisses
50 print(ergebnis_df)
51
52 # 7. Speichern Sie das Ergebnis in einer neuen Excel-Datei (optional)
53 ergebnis_df.to_excel('Prozentuale_Veraenderung_Fallzahlen.xlsx', index=False)

```

Listing 61: Skript Testfall 4 Prompt 3 Ausführung 5

### 8.2.5 Manuelle Skripte

```

1 import pandas as pd
2
3 file = 'Fallzahlen.xlsx'
4 sheet = 'Fallzahlen_2023'
5
6 # Read the excel file
7 try:
8     df = pd.read_excel(file, sheet_name=sheet)
9 except FileNotFoundError:
10     print(f"Datei '{file}' wurde nicht gefunden.")
11     exit(1)
12
13 # Check for required column
14 required_columns = ['LOR-Schlüssel', 'Straftaten_insgesamt']
15 missing_columns = [col for col in required_columns if col not in df.columns]
16 if missing_columns:
17     raise ValueError(f"Fehlende Spalten in der Excel-Datei: {missing_columns}")
18
19 # Extract non sort rows and save them for later
20 extract_keys = [999900, 999999]
21 extract_rows = df[df['LOR-Schlüssel'].isin(extract_keys)]
22 df = df[~df['LOR-Schlüssel'].isin(extract_keys)]
23
24 # Sort dataframe and concat with extracted rows
25 sorted_df = df.sort_values(by='Straftaten_insgesamt', ascending=False)
26 sorted_df = pd.concat([sorted_df, extract_rows], ignore_index=True)
27
28 print(sorted_df)

```

Listing 62: Manuelles Skript Testfall 1

```
1 import pandas as pd
2
3 # Read excel file
4 file = 'Fallzahlen.xlsx'
5 sheets = pd.read_excel(file, sheet_name=None)
6
7 # Join all sheets
8 joined_df = pd.concat(sheets.values(), ignore_index=True)
9 grouped_df = joined_df.groupby(['LOR-Schlüssel', 'Bezirke'], as_index=False).
    sum()
10
11 # Exclude not to sort rows
12 exclude_keys = [999900, 999999]
13 to_sort_df = grouped_df[~grouped_df['LOR-Schlüssel'].isin(exclude_keys)]
14 excluded_df = grouped_df[grouped_df['LOR-Schlüssel'].isin(exclude_keys)]
15
16 # Sort DF
17 sorted_df = to_sort_df.sort_values(by='Straftaten_insgesamt', ascending=False)
18
19 # Concat excluded rows
20 final_df = pd.concat([sorted_df, excluded_df], ignore_index=True)
21
22 print(final_df)
```

Listing 63: Manuelles Skript Testfall 2

```
1 import pandas as pd
2
3 # Read Excel file
4 df = pd.read_excel('Fallzahlen.xlsx', sheet_name='Fallzahlen_2023', usecols=[
    'Bezirke', 'Straftaten_insgesamt'])
5
6 # Extract total and save for later
7 total = df['Straftaten_insgesamt'].iloc[-1]
8 df = df.iloc[:-1]
9
10 # Calculate percentage
11 df['% Anteil'] = (df['Straftaten_insgesamt'] / total) * 100
12 df['% Anteil'] = df['% Anteil'].round(2)
13
14 # Add total row
15 df_total = pd.DataFrame({
16     'Bezirke': ['Berlin (PKS gesamt)'],
17     'Straftaten_insgesamt': [total],
18     '% Anteil': [100]
19 })
20 df = pd.concat([df, df_total], ignore_index=True)
21
```

```
22 print(df)
```

Listing 64: Manuelles Skript Testfall 3

```
1 import pandas as pd
2
3 # Read Excel
4 df = pd.read_excel('Fallzahlen.xlsx', sheet_name=None, usecols=['Bezirke', '
    Straftaten_insgesamt'])
5
6 gesamtwerte = []
7 # Loop all sheets
8 for sheet_name, data in df.items():
9     # Get year from sheet name
10    year = sheet_name.split('_')[-1]
11    # Get total
12    total = data["Straftaten_insgesamt"].iloc[-1]
13    # Append to list
14    gesamtwerte.append((year, total))
15
16 # Create DataFrame and add % change
17 df_total = pd.DataFrame(gesamtwerte, columns=['Jahr', 'Straftaten_insgesamt'
    ])
18 df_total["% Entwicklung"] = df_total['Straftaten_insgesamt'].pct_change() *
    100
19 df_total["% Entwicklung"] = df_total["% Entwicklung"].round(2)
20
21 print(df_total)
```

Listing 65: Manuelles Skript Testfall 4

TODO: Abbildungsverzeichnis einfügen