Business Analytics

03 | Deskriptive Analyse - Part 2

Prof. Dr. Felix Zeidler | FH Bielefeld | SoSe 2023

Inhaltsverzeichnis

Lernziele

1️⃣ Programmierkonzept: **“Bedingte Anweisungen”**

2️⃣ BA-Prozess: Transformieren von Daten

3️⃣ BA-Prozess: Visualisierung von Daten

(1) Programmierkonzept: “Bedingte Anweisungen”

Wofür benötigen wir bedingte Anweisungen?

Wieso benötigen wir bedingte Anweisungen?

- **Entscheidungen treffen:** Bedingte Anweisungen (If-Statements) helfen uns, in einem Programm unterschiedliche Aktionen auszuwählen, je nachdem, ob bestimmte Bedingungen erfüllt sind oder nicht.
- **Anpassung an Situationen:** If-Statements ermöglichen es, Programme zu erstellen, die flexibel auf verschiedene Situationen reagieren und sich an unterschiedliche Bedingungen anpassen können.
- **Einfache Logik:** Bedingte Anweisungen bieten eine leicht verständliche Möglichkeit, um in Programmen logische Zusammenhänge und Abhängigkeiten zwischen Daten und Aktionen abzubilden, was für die Entwicklung von vielfältigen und nützlichen Anwendungen erforderlich ist.

> **Hinweis:** bedingte Anweisungen werden auch als **if-Statements** bezeichnet.

Wie funktionieren bedingte Anweisungen?

Wie funktionieren bedingte Anweisungen?

- **Bedingung:** Wenn eine Bedingung erfüllt ist, wird ein bestimmter Codeblock ausgeführt. Wenn die Bedingung nicht erfüllt ist, wird ein anderer Codeblock ausgeführt.

Syntax:

```
1 if Bedingung_1:
2     Aktion_A
3
4 elif Bedingung_2:
5     Aktion_B
6 ...
7 else:
8     Aktion_Z
```

Kommentar

- `if`: Startet die If-Anweisung und prüft die erste Bedingung.
- Bedingung: Ein Ausdruck, der entweder `True` oder `False` ergibt.
- Codeblock: Eingerückter Abschnitt, der bei erfüllter Bedingung ausgeführt wird.
- `elif`: Prüft (optional) zusätzliche Bedingungen, falls vorherige nicht erfüllt sind.
- `else`: Definiert Aktionen, wenn keine der Bedingungen erfüllt ist.

Beispiel: Bedingte Anweisungen

Beispiel: Bedingte Anweisungen

Wir wollen auf Basis der Einnahmen und Ausgaben eines Unternehmens den Gewinnstatus des Unternehmens bestimmen und automatisiert ausgeben.

In diesem Szenario könnten Sie eine bedingte Anweisung verwenden, um den Gewinnstatus des Unternehmens basierend auf den Einnahmen und Ausgaben zu ermitteln:

```
einnahmen = 50_000  # Die Einnahmen des Unternehmens in diesem Quartal
ausgaben = 55_000   # Die Ausgaben des Unternehmens in diesem Quartal

if einnahmen > ausgaben:
    print("Das Unternehmen hat in diesem Quartal Gewinn gemacht.")
elif einnahmen == ausgaben:
    print("Das Unternehmen hat in diesem Quartal weder Gewinn noch Verlust gemacht.")
else:
    print("Das Unternehmen hat in diesem Quartal Verlust gemacht.")
```

Arten von Bedingungen

In Python können **verschiedene Arten von Bedingungen** geprüft werden, dazu gehören:

1️⃣ Vergleichsoperatoren:

- Gleichheit: `a == b`
- Ungleichheit: `a != b`
- Größer als: `a > b`
- Kleiner als: `a < b`
- Größer oder gleich: `a >= b`
- Kleiner oder gleich: `a <= b`

3️⃣ Logische Operatoren:

- Logisches UND (AND): `a and b`
- Logisches ODER (OR): `a or b`
- Logisches NICHT (NOT): `not a`

2️⃣ Prüfung auf Mitgliedschaft:

- Element in einer Liste oder einem anderen Container: `x in container`
- Element nicht in einer Liste oder einem anderen Container: `x not in container`

4️⃣ Prüfung auf Identität:

- Identität: `a is b`
- Nicht identisch: `a is not b`

Aufgabe 1: Bedingte Anweisungen

Schreiben Sie eine Funktion, die basierend auf dem jährlichen Einkommen einer Person den entsprechenden Steuersatz berechnet und die zu zahlende Steuer zurückgibt. Dabei sollen Sie bedingte Anweisungen verwenden, um unterschiedliche Steuersätze für verschiedene Einkommensstufen zu berücksichtigen.

Anforderungen:

1. Erstellen Sie eine Funktion namens `berechne_steuern`, die das jährliche Einkommen als Parameter akzeptiert.
2. Verwenden Sie bedingte Anweisungen, um den Steuersatz basierend auf dem Einkommen zu bestimmen:
   - Einkommen bis 14.000 Euro: Steuersatz von 12%
   - Einkommen zwischen 14.001 und 55.000 Euro: Steuersatz von 24%
   - Einkommen zwischen 55.001 und 200.000 Euro: Steuersatz von 42%
   - Einkommen über 200.000 Euro: Steuersatz von 45%
3. Der Steuersatz gilt für das gesamte Einkommen.
4. Geben Sie die berechnete Steuer zurück.

Beispiel:

```
steuer = berechne_steuern(100_000)
print(steuer) # Sollte 42% * 100_000 = 42_000 zurückgeben
```

Lösung 1: Bedingte Anweisungen

Lösung:

```python
def berechne_steuern(einkommen):
    if einkommen <= 14000:
        steuersatz = 0.12
    elif einkommen <= 55000:
        steuersatz = 0.24
    elif einkommen <= 200000:
        steuersatz = 0.42
    else:
        steuersatz = 0.45

    steuer = einkommen * steuersatz
    return steuer

# Beispiel: Berechnung der Steuer für ein Einkommen von 60.000 Euro
steuer = berechne_steuern(60000)
```

Aufgabe 2: Bedingte Anweisungen

Aufgabe:

1. Erstellen Sie eine Funktion namens `bestimme_kundenbindung`, die den jährlichen Umsatz eines Kunden und die Anzahl der Transaktionen in den letzten drei Jahren als Parameter akzeptiert.

2. Überprüfen Sie, ob der übergebene Umsatz ein numerischer Wert größer oder gleich 0 und die Anzahl der Transaktionen ein numerischer Wert größer oder gleich 1 ist. Ist dies nicht der Fall, geben Sie die Nachricht "Ungültige Eingabe! Bitte geben Sie gültige Werte für Umsatz und Transaktionsanzahl ein." zurück.

3. Verwenden Sie bedingte Anweisungen, um die Kundenbindungskategorie basierend auf dem Umsatz und der Transaktionsanzahl zu bestimmen:

   - Umsatz bis 1.000 Euro und weniger als 10 Transaktionen: "Bronze"
   - Umsatz zwischen 1.001 und 5.000 Euro oder 10 bis 29 Transaktionen: "Silber"
   - Umsatz zwischen 5.001 und 10.000 Euro oder 30 bis 59 Transaktionen: "Gold"
   - Umsatz über 10.000 Euro oder 60 oder mehr Transaktionen: "Platin"

4. Geben Sie die ermittelte Kundenbindungskategorie zurück.

Beispiel:

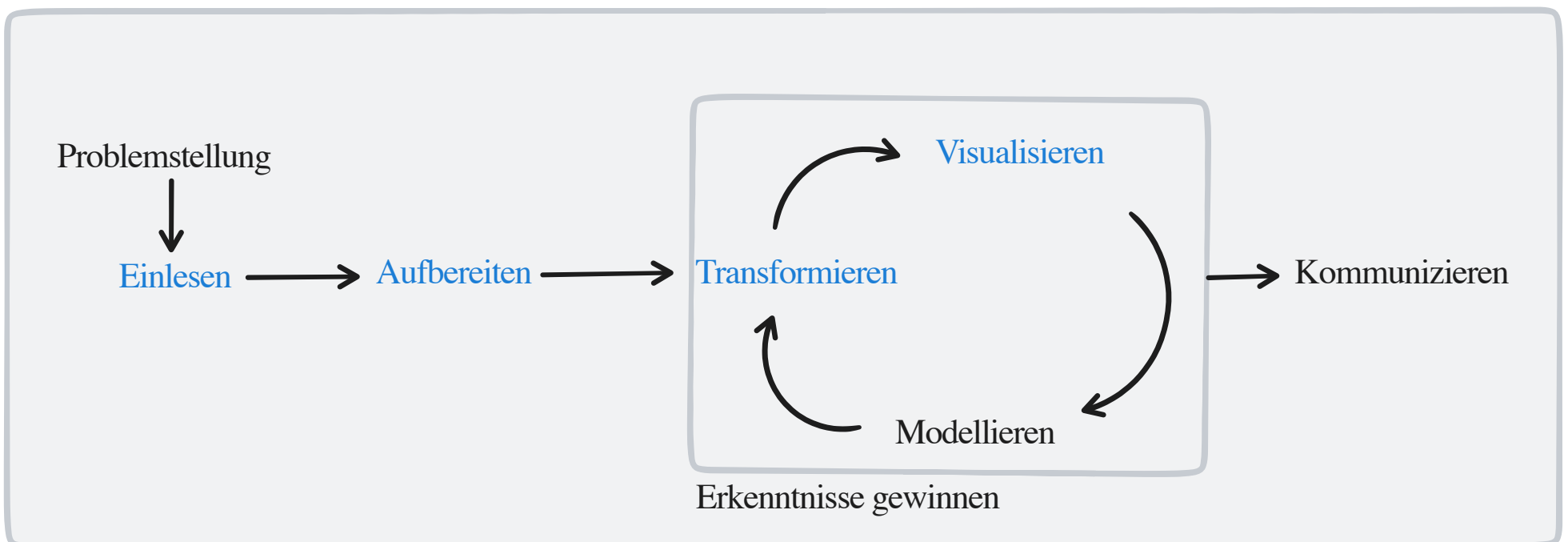
```
1  bestimme_kundenbindung(7000, 40) # Sollte "Gold" ausgeben
2  bestimme_kundenbindung(7000, 3) # Sollte "Gold" ausgeben
```

Lösung 2: Bedingte Anweisungen

```
1  def bestimme_kundenbindung(umsatz, transaktionen):
2
3      if umsatz < 0 or transaktionen < 1:
4          return "Ungültige Eingabe! Bitte geben Sie gültige Werte für Umsatz und Transaktion
5
6      if umsatz > 10_000 or transaktionen >= 60:
7          kategorie = "Platin"
8
9      elif (umsatz <= 10_000 and umsatz > 5000) or (transaktionen >= 30 and transaktionen < 6
10         kategorie = "Gold"
11
12     elif (umsatz <= 5_000 and umsatz > 1000) or (transaktionen >= 10 and transaktionen < 30
13         kategorie = "Silber"
14
15     else:
16         kategorie = "Bronze"
17
18     return kategorie
19
```

(2) BA-Prozess: Transformieren und Visualisieren

Business Analytics Prozess: Transformieren und Visualisieren



Datensatz nach Aufbereitung: Code

Recap: Aufbereitung des Datensates

```
import pandas as pd
link = "https://www.dropbox.com/s/ov6mnmgzrydquie/Construction.csv?dl=1"
df = (pd.read_csv(link)
      .rename(columns={'Project_ID': 'id',
                       'Name Projekt': 'name',
                       'projekt_Beginn': 'beginn',
                       'Plan Bau fertig': 'ende_plan',
                       'Fertig_IST': 'ende_ist',
                       'Kosten Plan': 'kosten_plan',
                       'Ist_Kosten': 'kosten_ist',
                       'Project_team': 'team'})
      .astype({"beginn": "datetime64", "ende_plan": "datetime64", "ende_ist": "datetime64"}
      .dropna()
      .drop_duplicates()
      .query('kosten_ist >= 0 & kosten_plan >= 0')
      .reset_index(drop=True))
```

Datensatz nach Aufbereitung: Auszug der Daten

	id	name	beginn	ende_plan	ende_ist	kosten_plan	kosten_ist	team	dauer
0	HN-399443	Straßenbau // Jennifer-Buchholz-Ring	2014-09-01	2014-10-03	2014-10-02	219817.40	246192.34	Team 3	31 days
1	UD-626094	Elektroarbeiten // Langernstraße	2021-06-12	2021-08-16	2021-09-18	105683.14	144657.38	Team 3	98 days
2	IO-468103	Spielplatz // Dussen vanweg	2016-05-20	2016-06-29	2016-06-27	129851.26	136753.06	Team 1	38 days
3	OG-758899	Stadtpark // Lübsstr.	2014-11-11	2014-12-30	2015-01-28	181236.83	273996.91	Team 2	78 days
4	CZ-107835	Elektroarbeiten // Försterweg	2017-07-25	2017-10-08	2017-10-08	75205.92	77519.27	Team 4	75 days

Schritte Transformation und Visualisierung

Transformation des Datensatzes

Die Transformation der Daten ist der Schritt im Business-Analytics-Prozess, bei dem bestehende Daten analysiert, neue Informationen generiert und der Datensatz so bearbeitet wird, dass er zur Beantwortung der Problemstellung hilfreich ist. Dies kann durch Aggregation, Gruppierung und Filterung von Daten sowie durch das Berechnen neuer Daten aus bestehenden Informationen erfolgen.

Wichtigkeit der Transformation:

- **Erkenntnisgewinn**: Die Transformation von Daten ermöglicht es, neue Erkenntnisse aus den vorhandenen Daten zu gewinnen und hilft bei der Beantwortung der ursprünglichen Problemstellung.

- **Informationsverdichtung**: Durch Aggregation und Gruppierung von Daten können komplexe Datensätze auf übersichtliche und relevante Informationen reduziert werden. Bessere Analyse: Die Transformation von Daten verbessert die Qualität der Analyse, da sie sicherstellt, dass nur relevante Informationen und Zusammenhänge untersucht werden. Basis für Visualisierung und Modellierung: Die Transformation von Daten bildet die Grundlage für anschließende Schritte, wie die Visualisierung und das Modellieren, indem sie die benötigten Daten in einer geeigneten Form bereitstellt.

- **Entscheidungsunterstützung**: Durch die Transformation von Daten können Entscheidungsträger fundierte Entscheidungen auf Basis der gewonnenen Erkenntnisse treffen.

Neue Daten aus bestehenden Informationen berechnen

> **Zur Erinnerung** 💡
>
> Problemstellung: Projekte dauern länger als geplant und kosten mehr als geplant.

Neue Informationen werden benötigt

- Verzögerung: z.B. zeitliche Überschreitung der Projekte ggü. dem Plan in Tagen
- Kostenüberschreitung: z.B. finanzielle Überschreitung der Projekte ggü. dem Plan in Euro
- Verzögerung in Prozent: z.B. zeitliche Überschreitung der Projekte ggü. dem Plan in Prozent
- Kostenüberschreitung in Prozent: z.B. finanzielle Überschreitung der Projekte ggü. dem Plan in Prozent

Neue Spalten einem Dataframe hinzufügen

Neue Spalten können auf verschiedene Arten erstellt werden. Zwei typische Vorgehen sind:

Neue Spalte mit `[]`

```
1 df["Spalte_neu"] = df["Spalte_alt"] + 1
```

Neue Spalte mit `assign()`

```
1 df = df.assign(Spalte_neu = df["Spalte_alt"] + 1)
```

Neue Spalten einem Dataframe hinzufügen: `.assign()`

Die Funktion `.assign()` hat gegenüber der Verwendung von `[]` insbesondere den Vorteil, dass sie

1. mehrere Spalten gleichzeitig erstellen kann.

2. im Rahmen einer Kette von Operationen verwendet werden kann.

Beispiel

```
df = df.assign(Spalte_neu_1 = df["Spalte_alt_1"] + 1,
               Spalte_neu_2 = df["Spalte_alt_2"] + 1,
               Spalte_neu_3 = df["Spalte_alt_3"] + df["Spalte_alt_4"])
```

vs.

```
df["Spalte_neu_1"] = df["Spalte_alt_1"] + 1
df["Spalte_neu_2"] = df["Spalte_alt_2"] + 1
df["Spalte_neu_3"] = df["Spalte_alt_3"] + df["Spalte_alt_4"]
```

Aufgabe 1: Neue Spalten erstellen

Fügen Sie dem Datensatz folgende Informationen hinzu:

- geplante Dauer
- tatsächliche Dauer
- absolute Verzögerung
- relative Verzögerung in Prozent
- absolute Kostenüberschreitung
- relative Kostenüberschreitung in Prozent

> **Hinweis:**
>
> - Achten Sie auf sinnvolle und aussagekräftige Bezeichnungen der neuen Spalten.
> - wandeln Sie die Ergebnisse via `dt.days` in ganze Tage um (für weitere Berechnungen hilfreich).
> - nutzen Sie die `[ ]`-Notation

Lösung 1: Neue Spalten erstellen

Lösung

```
df["dauer_plan"] = (df["ende_plan"] - df["beginn"]).dt.days
df["dauer_ist"] = (df["ende_ist"] - df["beginn"]).dt.days
df["verzögerung_abs"] = (df["ende_ist"] - df["ende_plan"]).dt.days
df["verzögerung_rel"] = df["verzögerung_abs"] / (df["dauer_plan"] + 1) * 100
df["kostenüberschreitung_abs"] = df["kosten_ist"] - df["kosten_plan"]
df["kostenüberschreitung_rel"] = df["kostenüberschreitung_abs"] / (df["kosten_plan"]) * 100
```

Hinweis

- `dt.days` wandelt die Differenz in ganze Tage um und konvertiert den Datentyp in `int`
- `+ 1` wird addiert, um den Fall zu berücksichtigen, dass ein Projekt am selben Tag beginnt und endet (Dauer = 0 Tage)

Fortgeschrittenere Lösung 1: Neue Spalten erstellen (via `.assign()`)

Lösung mit `assign()`

```
df = df.assign(dauer_plan = (df["ende_plan"] - df["beginn"]).dt.days,
               dauer_ist = (df["ende_ist"] - df["beginn"]).dt.days,
               verzögerung_abs = (df["ende_ist"] - df["ende_plan"]).dt.days,
               verzögerung_rel = lambda _df: _df["verzögerung_abs"] / (_df["dauer_plan"] +
               kostenüberschreitung_abs = df["kosten_ist"] - df["kosten_plan"],
               kostenüberschreitung_rel = lambda _df: _df["kostenüberschreitung_abs"] / (_d
```

Hinweis zu `lambda`

Neue Spalten aus Text extrahieren

Eine häufige Aufgabe ist es, Text-Spalten zu transformieren, z.B. um Informationen aus dem Text zu extrahieren.

Beispiel: Der Name des Projektes setzt sich zusammen aus der Art des Projektes und dem Ort des Projektes (Straßenname). Die Art des Projektes könnte uns interessieren und für weitere Analyse nützlich sein.

Lösung: Wir extrahieren die Art des Projektes aus dem Namen des Projektes.

```
1 df["art"] = df["name"].str.split("//").str[0]
```

- `df["name"].str.split("//")`: Wir teilen den Text in der Spalte `name` anhand des Zeichens `//` in zwei Teile auf.
- `.str[0]`: Wir wählen den ersten Teil aus.
- `df["art"] = ...`: Wir speichern den ersten Teil in der Spalte `art`.

Umgang mit Ausreißern

Ausreißer

Werden auch als Outlier bezeichnet und sind Beobachtungen oder Datenpunkte, die sich deutlich von anderen Werten in einer Datensammlung unterscheiden. Sie können auf Messfehler, Zufallsvariationen oder tatsächliche Abweichungen in der zugrunde liegenden Verteilung zurückzuführen sein und können die statistische Analyse beeinflussen, wenn sie nicht angemessen berücksichtigt werden.

Identifikation von Ausreißern: unter andem z.B. durch

- **Deskriptive Statistiken:** Berechnung von Maßen wie Mittelwert, Median, Quartile, um erste Anzeichen von Ausreißern zu erkennen.
- **Visualisierung:** Erstellen von Boxplots, Histogrammen oder Streudiagrammen, um Verteilungen und mögliche Ausreißer visuell zu erfassen.
- **Z-Scores:** Berechnung von Z-Scores (Standardabweichungen vom Mittelwert), um extreme Werte in Bezug auf die Streuung der Daten zu identifizieren.

Deskriptive Analyse

Identifikation von Ausreißern: deskriptive Statistiken

```
cols = ["verzögerung_abs", "verzögerung_rel",
        "kostenüberschreitung_abs", "kostenüberschreitung_rel"]
df[cols].describe()
```

	verzögerung_abs	verzögerung_rel	kostenüberschreitung_abs	kostenüberschreitung_rel
count	9,766.00	9,766.00	9,766.00	9,766.00
mean	10.50	35.35	782,121.85	1,483.71
std	25.50	132.87	27,956,490.56	79,657.12
min	-50.00	-97.22	-129,236.80	-39.76
25%	0.00	0.00	5,652.13	8.29
50%	2.00	6.67	16,535.49	17.44
75%	6.00	36.36	40,305.09	32.95
max	230.00	5,500.00	1,133,999,548.79	7,274,556.46

> **Finding:** Hohe Mittelwerte (im Vergleich zum Median) und Standardabweichungen deuten auf mögliche Ausreißer hin.

Identifikation von Ausreißern: deskriptive Statistiken

Beispiel: Beobachtung mit den drei größten absoluten Kostenüberschreitung

```
rows = df["kostenüberschreitung_abs"].nlargest(3).index
cols = ["id", "name", "kosten_plan",
        "kosten_ist", "kostenüberschreitung_abs", "kostenüberschreitung_rel"]
df.loc[rows, cols]
```

	id	name	kosten_plan	kosten_ist	kostenüberschreitung_abs	kostenüberschreitung_rel
6543	BQ-775034	Stadtpark // Jäntschstraße	117,572.19	1,134,117,120.98	1,133,999,548.79	964,513.42
7188	LJ-552344	Landschaftsbau // Oswin-Bauer-Gasse	15,557.57	1,131,759,771.56	1,131,744,213.99	7,274,556.46
9428	PG-617746	Elektroarbeiten // Liebeltstr.	129,009.58	1,097,313,648.55	1,097,184,638.97	850,467.57

> **Finding:** Kostenüberschreitungen von über 1 Mrd. EUR erscheinen unplausibel.

Aufgabe: Beobachtungen mit den drei größten absoluten Verzögerungen

Extrahieren Sie die Beobachtungen mit den `drei größten relativen Verzögerungen` und geben Sie die folgenden Spalten aus:

- `id`
- `name`
- `dauer_plan`
- `dauer_ist`
- `verzögerung_abs`
- `verzögerung_rel`

Lösung: Beobachtungen mit den drei größten relativen Verzögerungen

Beispiel: Beobachtung mit den `drei größten relativen Verzögerungen`

```
rows = df["verzögerung_rel"].nlargest(3).index
cols = ["id", "name", "dauer_plan",
        "dauer_ist", "verzögerung_abs", "verzögerung_rel"]
df.loc[rows, cols]
```

	id	name	dauer_plan	dauer_ist	verzögerung_abs	verzögerung_rel
4957	KW-447733	Straßenbau // Huhnstr.	0	55	55	5,500.00
6861	FS-809060	Elektroarbeiten // Jüttnerstraße	0	41	41	4,100.00
2013	VW-696455	Landschaftsbau // Jann-Junk-Weg	0	33	33	3,300.00

> **Finding:** Die größten relativen Verzögerungen betreffen Projekte mit einer relativ kurzen Dauer und sind somit durchaus plausibel.

Behandlung von Ausreißern

Behandlung von Ausreißern: unter anderem z.B. durch

- **Ausreißer entfernen:** Entfernen von Ausreißern, wenn sie auf Messfehler zurückzuführen sind.
- **Ausreißer ersetzen:** Ersetzen von Ausreißern durch andere Werte, z.B. durch den Median oder den Mittelwert.
- **Ausreißer ignorieren:** Ignorieren von Ausreißern, wenn sie auf tatsächliche Abweichungen in der zugrunde liegenden Verteilung zurückzuführen sind.

Behandlung von Ausreißern

In unserem Falle ist teilweise nicht klar, ob die Ausreißer auf Messfehler oder tatsächliche Abweichungen in der zugrunde liegenden Verteilung zurückzuführen sind. Beispiel:

- **Kostenüberschreitung:** Die Kostenüberschreitung von über 1 Mrd. EUR erscheint auf den ersten Blick als Eingabe- bzw. Messfehler.
- **Verzögerung:** Die größten relativen Verzögerungen betreffen Projekte mit einer relativ kurzen Dauer und sind durchaus plausibel. Allerdings gibt es auch Projekte, bei denen die Dauer signifikant unterschritten wird (dauer_ist = 0)

➡️ In unserem Beispieldatensatz werden wir die Ausreißer deshalb entfernen.

> **Wichtig**:
> Entscheidung über die Behandlung von Ausreißern sollte immer im Kontext der zugrunde liegenden Daten und der Fragestellung getroffen werden. Der Umgang mit Ausreißern sollte auch immer transparent dokumentiert werden, da wir den Datensatz damit verändern.

Entfernen von Ausreißern via Quantile

Quantile:

- Quantile: Teilen Daten in gleiche Abschnitte
- Arten von Quantilen: Quartile (4 Teile), Quintile (5 Teile), Dezile (10 Teile), Perzentile (100 Teile)
- Perzentile: Spezialfall von Quantilen, teilen Daten in 100 Teile
- Beispiel: 99. Perzentil: Wert, unterhalb dessen 99 % der Daten liegen; zeigt einen sehr hohen Wert im Vergleich zur Gesamtverteilung (z. B. Einkommen, Testergebnisse)

Quantile in Pandas: `quantile()`

```
1  df["kostenüberschreitung_abs"].quantile([0.01, 0.99])
```

```
0.01    -10768.9835
0.99    304094.4865
Name: kostenüberschreitung_abs, dtype: float64
```

Entfernen von Ausreißern via Quantile

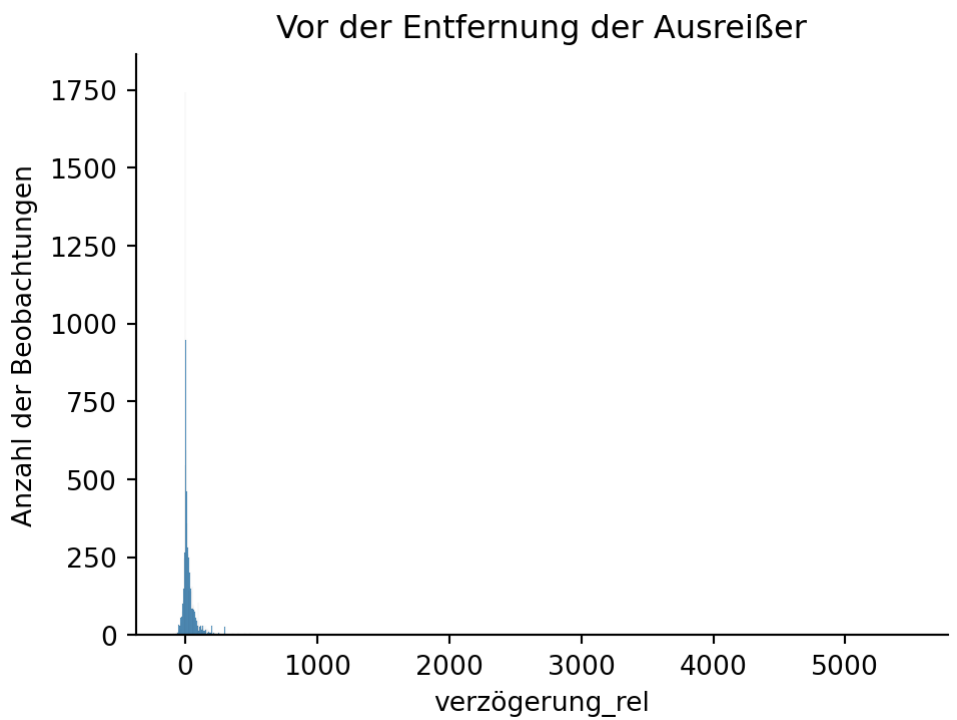
Beispiel: Entfernen der Ausreißer der **relativen Verzögerung** unterhalb des 1. Perzentils und oberhalb des 99. Perzentils

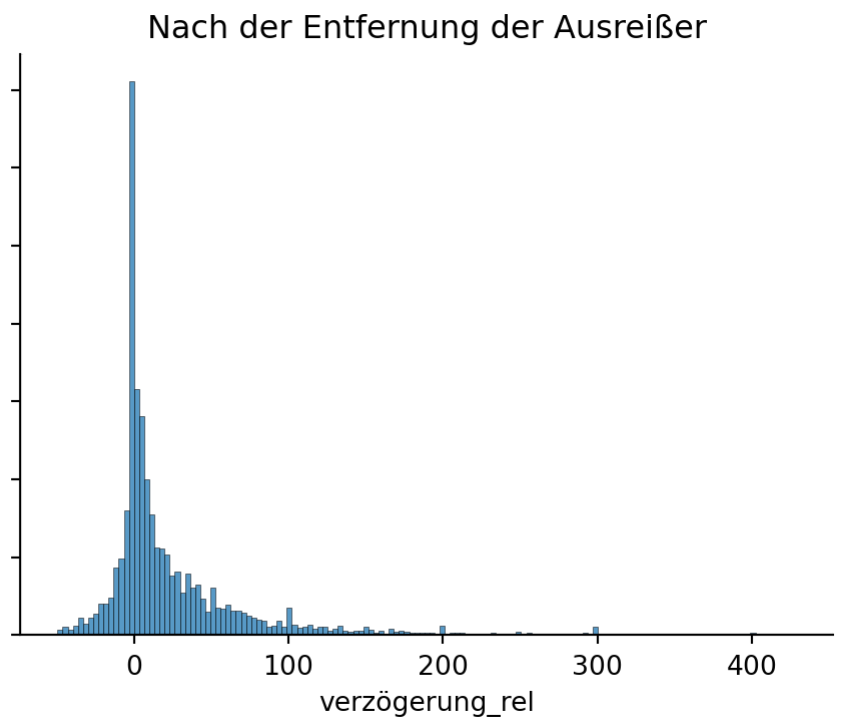
```python
q1 = df["verzögerung_rel"].quantile(0.01)
q99 = df["verzögerung_rel"].quantile(0.99)

# Entfernen der Ausreißer
df_neu = df.query("verzögerung_rel > @q1 and verzögerung_rel < @q99")
```

- `q1` und `q99`: 1. und 99. Perzentil der relativen Kostenüberschreitungen
- `@`: Pandas-Syntax für Variablen in der Query-Syntax

Entfernen von Ausreißern via Quantile: Vergleich der Verteilungen

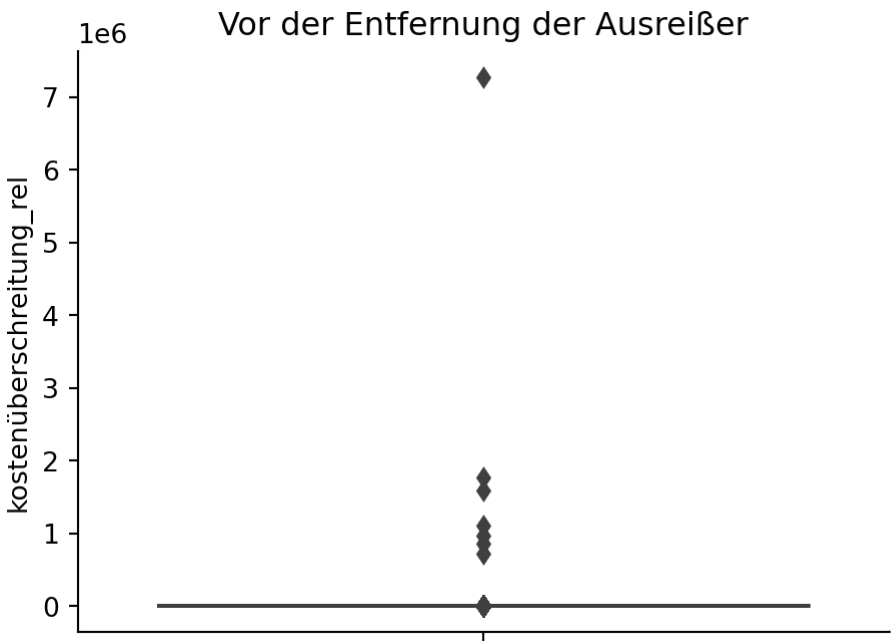
Vergleich der Verteilungen: Wir vergleichen die Verteilungen der **relativen Verzögerung** vor und nach der Entfernung der Ausreißer (1. und 99. Perzentil) via `Histogramm`

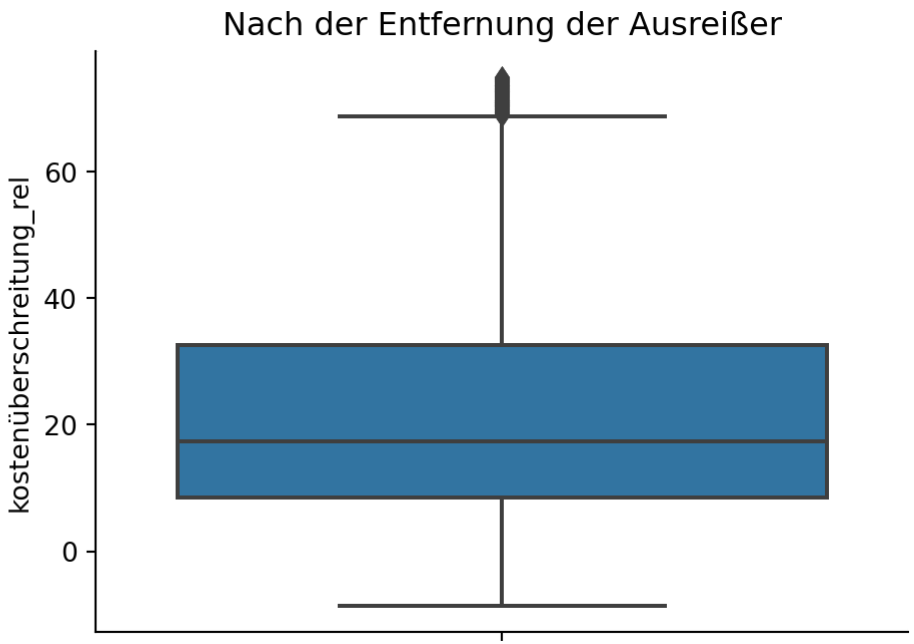




Entfernen von Ausreißern via Quantile: Vergleich der Verteilungen

Vergleich der Verteilungen: Wir vergleichen die Verteilungen der **relativen Kostenüberschreitung** vor und nach der Entfernung der Ausreißer (1. und 99. Perzentil) via `Boxplot`





Aufgabe: Entfernen von Ausreißern

Für die Spalten `kostenüberchreitung_abs`, `kostenüberschreitung_rel`, `verzögerung_abs` und `verzögerung_rel` sollen die Ausreißer unterhalb des 1. Perzentils und oberhalb des 99. Perzentils entfernt werden.

Ergebnis sollte ein bereinigter Dataframe `df` sein.

Lösung: Entfernen von Ausreißern

```
# Kostenüberschreitung absolut
ka1 = df["kostenüberschreitung_abs"].quantile(0.01)
ka99 = df["kostenüberschreitung_abs"].quantile(0.99)
# Kostenüberschreitung relativ
kr1 = df["kostenüberschreitung_rel"].quantile(0.01)
kr99 = df["kostenüberschreitung_rel"].quantile(0.99)
# Verzögerung absolut
va1 = df["verzögerung_abs"].quantile(0.01)
va99 = df["verzögerung_abs"].quantile(0.99)
# Verzögerung relativ
vr1 = df["verzögerung_rel"].quantile(0.01)
vr99 = df["verzögerung_rel"].quantile(0.99)
### REMOVE OUTLIERS
df = df.query("kostenüberschreitung_abs > @ka1 and kostenüberschreitung_abs < @ka99")
df = df.query("kostenüberschreitung_rel > @kr1 and kostenüberschreitung_rel < @kr99")
df = df.query("verzögerung_abs > @va1 and verzögerung_abs < @va99")
df = df.query("verzögerung_rel > @vr1 and verzögerung_rel < @vr99")
```

Exkurs: Funktion schreiben für Entfernen von Ausreißern

Funktion schreiben: Wir schreiben eine Funktion, die die Entfernung von Ausreißern für eine beliebige Spalte ermöglicht.

```
def remove_outliers(df, col, q1=0.01, q99=0.99):
    q1 = df[col].quantile(q1)
    q99 = df[col].quantile(q99)
    df = df.query(f"{col} > @q1 and {col} < @q99")
    return df
```

Funktion anwenden: Wir wenden die Funktion auf eine der vier Spalten an.

```
df = remove_outliers(df, "kostenüberschreitung_abs")
```

Exkurs: Funktion schreiben für Entfernen von Ausreißern

Problem der Funktion: bei mehreren Spalten werden die Perzentile immer auf Basis eines neuen und bereits bereinigtem Datensatz berechnet. Dies kann dazu führen, dass sehr viele Beobachtungen entfernt werden.

Lösung: Wir schreiben eine Funktion, die die Entfernung von Ausreißern für mehrere, beliebige Spalte ermöglicht. Die Perzentile werden für jede Spalte auf Basis des ursprünglichen Datensatzes berechnet.

```
1  def remove_outliers(df, cols, p_low=0.01, p_high=0.99):
2
3      # Calculate percentile cutoff values
4      cutoffs = df[cols].quantile([p_low, p_high])
5
6      # Remove outliers
7      for col in cols:
8          df = df.query(f"{col} > {cutoffs.loc[p_low, col]} and {col} < {cutoffs.loc[p_high,c
9
10     return df
11
12 cols = ["kostenüberschreitung_abs", "kostenüberschreitung_rel", "verzögerung_abs", "verzöge
13 df = remove_outliers(df, cols)
```

Analyse der Daten

Nachdem wir nun (i) relevante Daten hinzugefügt und (ii) Ausreißer entfernt haben, können wir uns nun mit der Analyse der Daten beschäftigen.

Wir könnten folgenden Fragen nachgehen:

1. Ausgangssituation bestätigen: gibt es tatsächlich Verzögerungen und Kostenüberschreitungen?

2. Hypothesen zur Erklärung der Verzögerungen und Kostenüberschreitungen aufstellen und prüfen

   - Haben sich die Verzögerungen und Kostenüberschreitungen im Laufe der Zeit verändert?
   - Gibt es einen Zusammenhang zwischen den Verzögerungen und den Kostenüberschreitungen?
   - Sind die Verzögerungen und Kostenüberschreitungen abhängig vom Team?
   - Sind die Verzögerungen und Kostenüberschreitungen abhängig von der Art des Projekts?
   - Sind die Verzögerungen und Kostenüberschreitungen abhängig von der Größe des Projekts?

Gruppieren und Aggregieren

Warum?

- Gruppieren und Aggregieren von Daten ist eine leistungsfähige Methode zur Analyse und Zusammenfassung großer Datenmengen.
- Sie ist nützlich für die deskriptive Statistik, da sie es ermöglicht, Muster, Trends und Zusammenhänge in den Daten zu erkennen.
- Der Ansatz funktioniert, indem man die Daten zunächst nach bestimmten Kriterien in Gruppen einteilt (gruppieren) und dann innerhalb dieser Gruppen verschiedene statistische Funktionen anwendet (aggregieren), wie zum Beispiel Mittelwert, Median, Summe oder Anzahl.

Gruppieren und Aggregieren (cont'd)

In Python

```
1  df.groupby("spalte_1").agg({"spalte_2": "<Aggregationsfunktion>"})
```

- `df.groupby("spalte_1")`: Teilt den DataFrame df in Gruppen basierend auf den einzigartigen Werten in der Spalte `"spalte_1"`.
- `.agg()`: Wendet eine Aggregationsfunktion auf die Gruppen an.
- `{"spalte_2": "<Aggregationsfunktion>"}`: Gibt an, dass für die Spalte "spalte_2" die `<aggregationsfunktion>` angewendet werden soll.
- `<Aggregationsfunktion>`: Eine Aggregationsfunktion, die auf die Gruppen angewendet werden soll. Zum Beispiel: `mean`, `sum`, `count`, `median`, `min`, `max`, `std`, `var`, `quantile`.

Beispiel

```
1  df.groupby("spalte_1").agg({"spalte_2": "mean"})
```

Gruppieren und Aggregieren (cont'd)

Beispiel: durchschnittliche absolute Verzögerung je Projektteam

- Gruppieren: Wir gruppieren die Daten nach `team`
- Aggregieren: wir nehmen die Spalte `verzögerung_abs` und berechnen den Mittelwert

```
1  df.groupby("team").agg({"verzögerung_abs": "mean"})
```

team	verzögerung_abs
Team 1	8.580801
Team 2	8.717090
Team 3	11.773603
Team 4	6.025735

Aufgabe: Gruppieren und Aggregieren

Aufgabe:

1. Berechne den Median der relative Verzögerung je Projektteam.

2. Berechnen Sie die Anzahl der Projekte je Projektteam.

Lösung: Gruppieren und Aggregieren

Lösung 1:

```
1  df.groupby("team").agg({"verzögerung_rel": "median"})
```

team	verzögerung_rel
Team 1	2.777778
Team 2	8.888889
Team 3	21.637427
Team 4	3.703704

Lösung 2:

```
1  df.groupby("team").agg({"id": "count"})
```

team	id
Team 1	2073
Team 2	2089
Team 3	2129

Gruppieren und Aggregieren: Datum

`pd.Grouper` ermöglicht es, Daten nach Zeitintervallen zu gruppieren

Beispiel: Wir gruppieren die Daten für jeweils 9 Monate und berechnen den Mittelwert der absoluten Verzögerung.

```
1  df.groupby(pd.Grouper(key="beginn", freq="9M")).agg({"verzögerung_abs": "mean"})
```

	verzögerung_abs
beginn	
2013-02-28	10.508475
2013-11-30	8.353675
2014-08-31	8.235784
2015-05-31	9.001344
2016-02-29	10.205479
2016-11-30	8.201794
2017-08-31	8.638571
2018-05-31	8.739521
2019-02-28	9.261773
2019-11-30	7.726008
2020-08-31	7.508772
2021-05-31	9.358273
2022-02-28	9.754331

Aufgabe: Gruppieren und Aggregieren mit Datum

Analysen Sie, ob sich die absolute Kostenüberschreitung im Laufe der Zeit verändert hat.

Lösungsskizze: Gruppieren und Aggregieren mit Datum

Lösungsskizze:

- Wir gruppieren die Daten nach Zeitintervallen von z.B. 3 Monaten.
- Basis könnte der Projektbeginn sein
- Wir berechnen den Median der absoluten Kostenüberschreitung, weil wir die Ergebnisse nicht durch noch verbliebene Extreme verfälschen wollen.

```
1 data = df.groupby(pd.Grouper(key="beginn", freq="3M")).agg({"kostenüberschreitung_abs": "me
2 data.head() # Zeige Auszug der Daten
```

	kostenüberschreitung_abs
beginn	
2013-02-28	12618.310
2013-05-31	14794.530
2013-08-31	13354.175
2013-11-30	15960.120
2014-02-28	16450.080

Visualisierung

Warum?

- Visualisierung ist eine sehr effektive Methode, um Daten zu verstehen und zu analysieren.
- Visualisierungen können Muster, Trends und Zusammenhänge in den Daten aufzeigen, die mit deskriptiven Statistiken nicht so leicht zu erkennen sind, da durch die Aggregation der Daten Informationen verloren gehen.

In Python

- Wir werden die Bibliothek `seaborn` verwenden, um die Daten zu visualisieren.
- `seaborn` ist eine Bibliothek, die auf `matplotlib` aufbaut und die Visualisierung von Daten mit `pandas`-DataFrames vereinfacht.
- Dokumentation: *https://seaborn.pydata.org/index.html*

Visualisierung: Arten

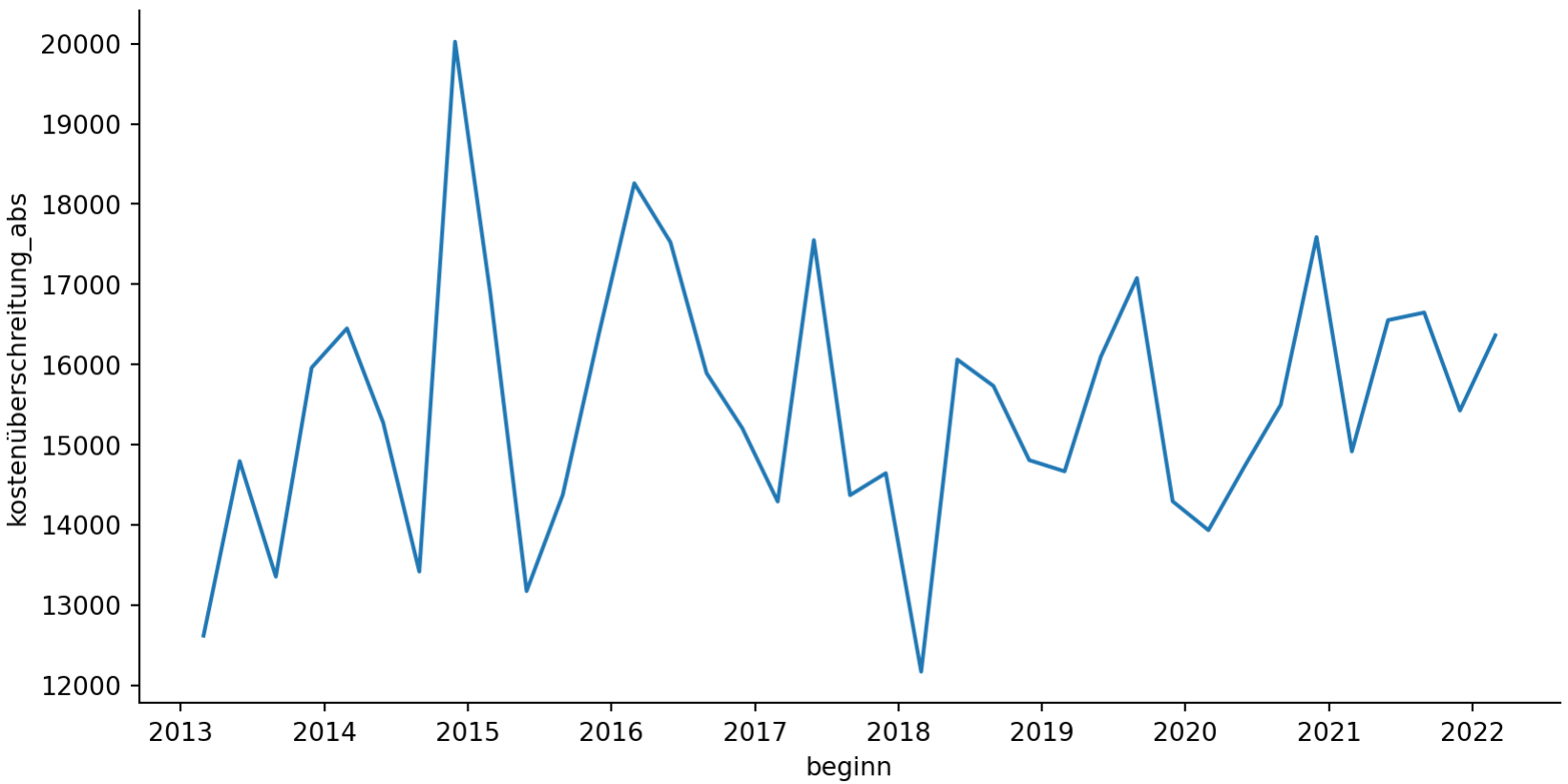
Typische Arten von Visualisierungen:

1. **Beziehungen:** Scatterplots, Liniendiagramme, Streudiagramme (*Tutorial*)
2. **Verteilungen:** Histogramme, Boxplots (*Tutorial*)
3. **Kategorien:** Balkendiagramme, obige Diagramme nach Kategorien (*Tutorial*)

> **Seaborn** bietet eine Vielzahl von Funktionen, um diese Diagramme zu erstellen.

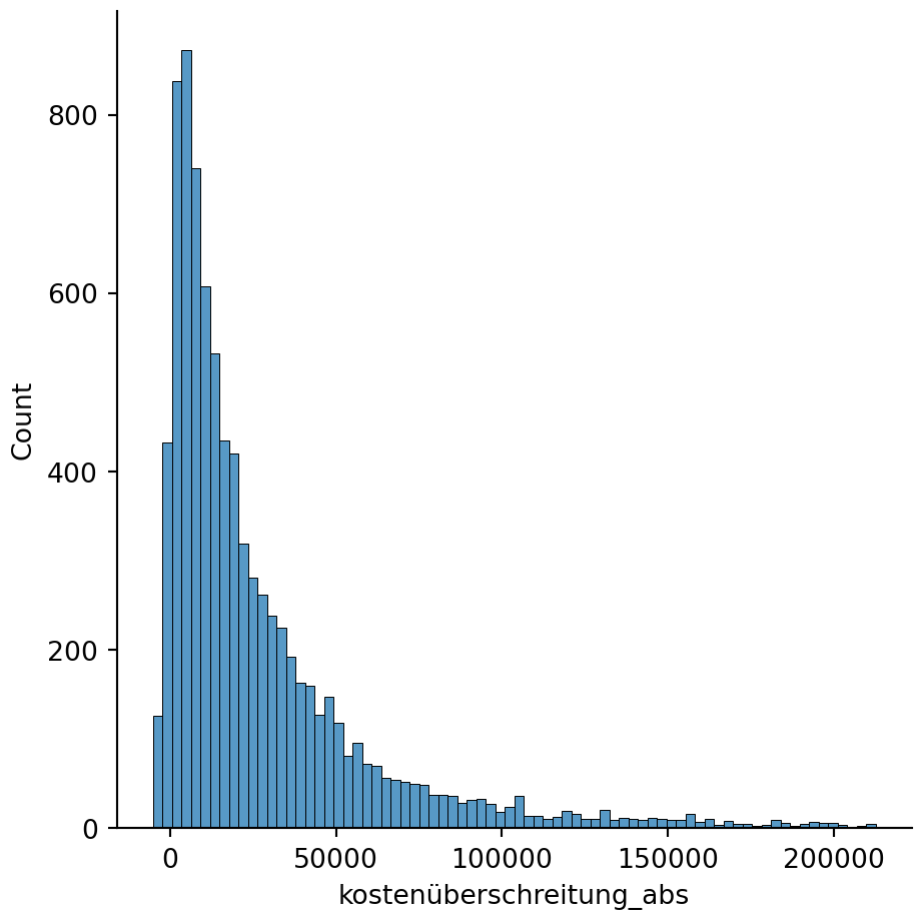
Visualisierung von Beziehungen: Beispiel “Liniendiagramm”

```
1  import seaborn as sns
2
3  data = df.groupby(pd.Grouper(key="beginn", freq="3M")).agg({"kostenüberschreitung_abs": "me
4  sns.lineplot(data=data, x="beginn", y="kostenüberschreitung_abs");
```



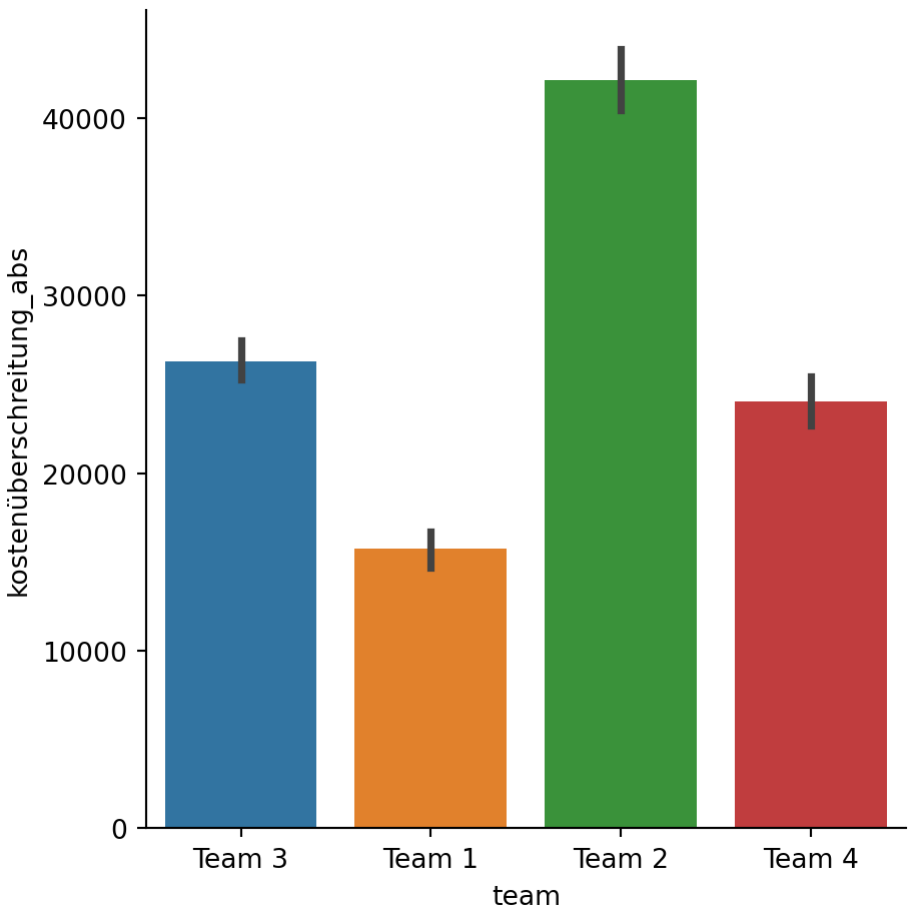
Visualisierung von Verteilungen: Beispiel “Histogramm”

```
1  sns.displot(data=df, x="kostenüberschreitung_abs", kind="hist");
```



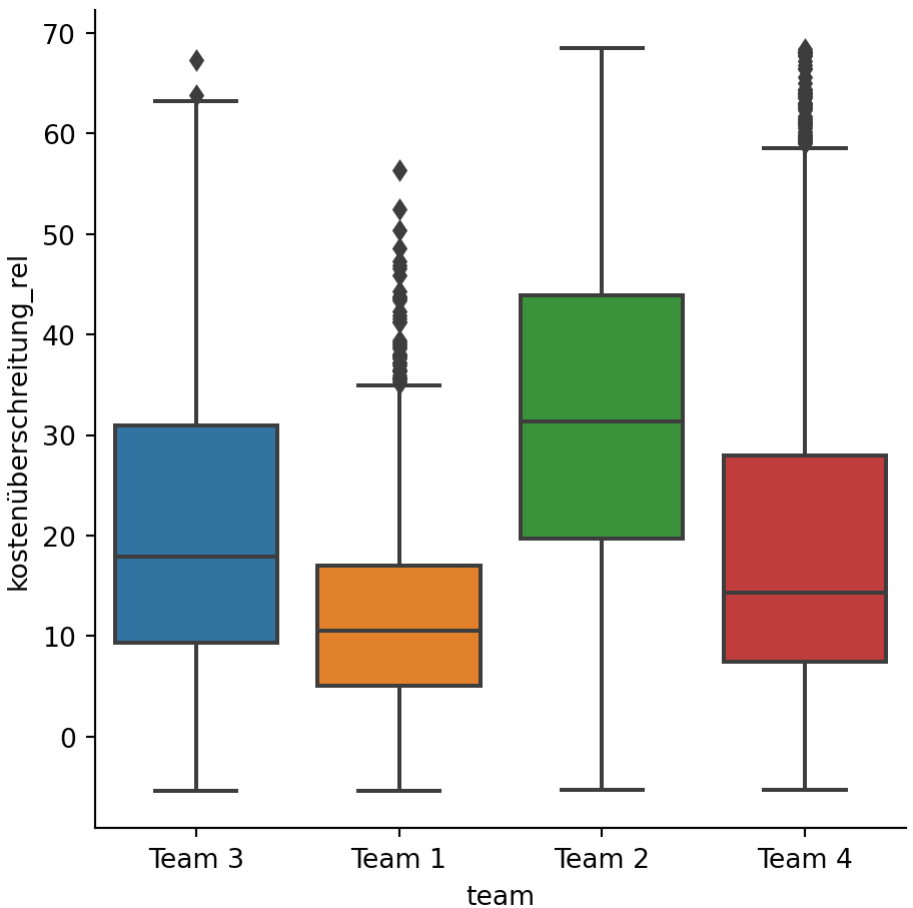
Visualisierung von Kategorien: Beispiel “Balkendiagramm”

```
1  sns.catplot(data=df, x="team", y="kostenüberschreitung_abs", kind="bar");
```



Visualisierung von Kategorien: Beispiel “Boxplot” je Projektteam

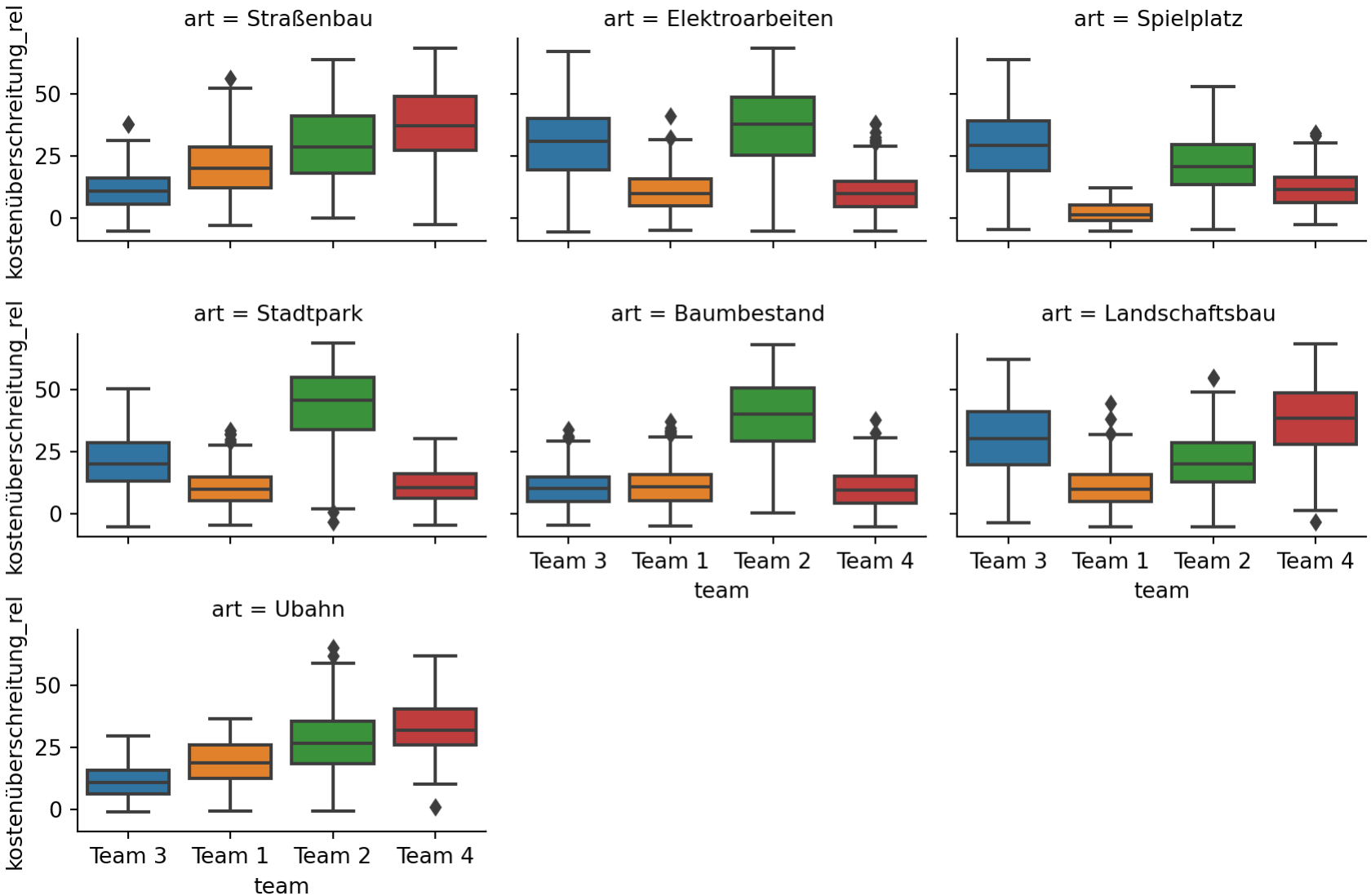
```
1  sns.catplot(data=df, x="team", y="kostenüberschreitung_rel", kind="box");
```



Deskriptive Analyse

Visualisierung von Kategorien: Beispiel “Boxplot” je Projektteam und Projektart

```
sns.catplot(data=df, x="team", y="kostenüberschreitung_rel", col="art",
            col_wrap=3, kind="box",
            height=2, aspect=1.5); # nur für Darstellung
```


art = Straßenbau
art = Elektroarbeiten
art = Spielplatz
art = Stadtpark
art = Baumbestand
art = Landschaftsbau
art = Ubahn
kostenüberschreitung_rel
0
25
50
Team 3
Team 1
Team 2
Team 4
team

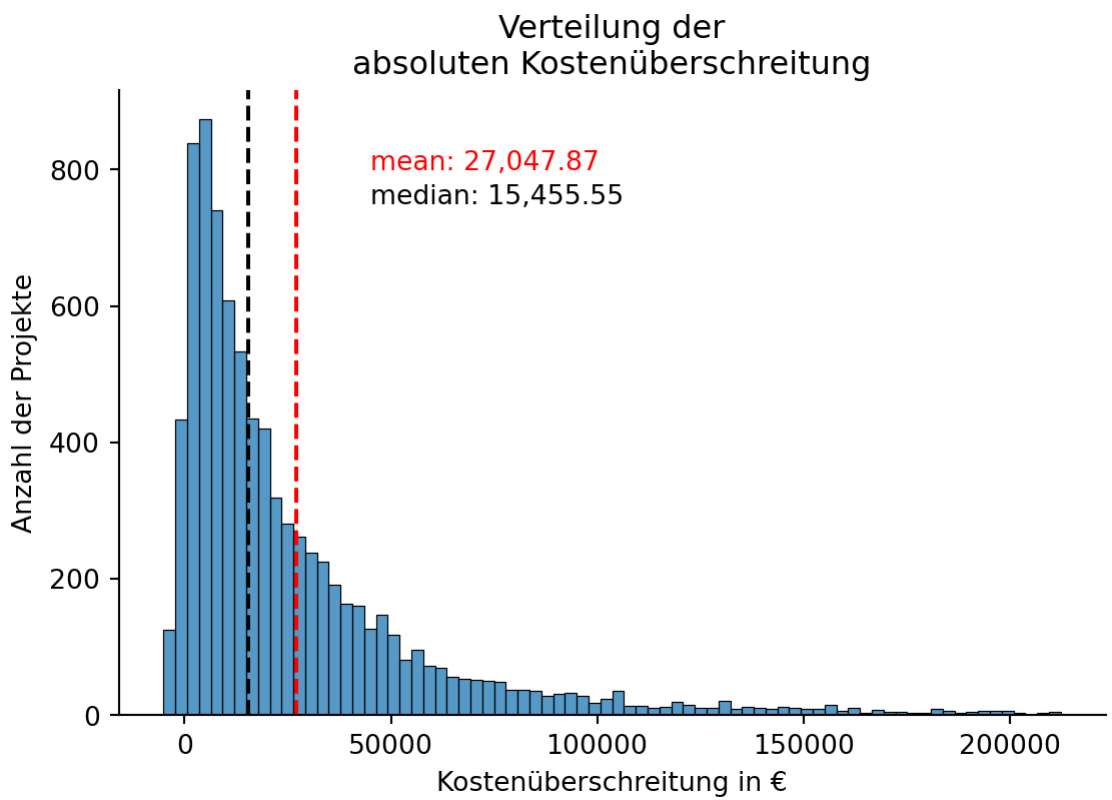
Visualisierung: Formatieren von Diagrammen

Formatierung von Diagrammen:

- Titel, Achsenbeschriftungen, Legende, Farben, Größe, etc.
- prinzipiell etwas *komplizierter* als in z.B. in Excel, dafür aber in Summe *flexibler* und *mehr Möglichkeiten*.
- nicht Fokus dieses Kurses!
- `matplotlib` ist Basis für `seaborn` und bietet viele Möglichkeiten zur Formatierung von Diagrammen.

Visualisierung: Formatieren von Diagrammen

Plot | Code



Quellen