

Business Analytics

03 | Deskriptive Analyse - Part 1

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

Inhaltsverzeichnis

(1) Programmierkonzept: "Funktionen"

(2) BA-Prozess: Einlesen und Aufbereiten von Daten

Lernziele

- 2 Programmierkonzept: **“Funktionen”**
- 1 Einlesen und Aufbereiten von Daten
- 3 Arbeiten mit Python-Modul `pandas`

(1) Programmierkonzept: "Funktionen"

Ausgangssituation

- Wir haben bereits erste Python-Programme geschrieben
- Einführende Fallstudie griff auf Module **pandas** (Dataframes) und **seaborn** (Visualisierungen) zurück
- Um Fallstudie zu bearbeiten haben wir Funktionen¹ aus diesen Modulen verwendet

Frage: Was sind eigentlich Funktion?

1. Hinweis: teilweise muss es präziser heißen: Klassen und Methoden. Für unsere Zwecke ist dies aber zunächst nicht weiter relevant.

Was sind Funktionen: Analogie "Toaster"



Was sind Funktionen: Analogie "Toaster"

- Funktionen in der Programmierung sind wie Toaster: klar definierte Aufgabe (d.h. Funktion)
- Sie nehmen Eingabewerte entgegen, verarbeiten sie und geben Ergebnisse zurück.
- Analogie: Brot (Eingabewerte) -> Rösten (Verarbeitung) -> geröstetes Brot (Ergebnis).

Was sind Funktionen: Analogie "Excel"

Fett

fx

	B	C	D	E
	Werte			
	0,706793		Anzahl Werte	?
	0,447353			
	0,855576			
	0,102575			
	0,989661			
	0,561972			
	0,656805			
	0,043952			
	0,299101			
	0,085281			

Was sind Funktionen: Analogie "Excel"

<div><div>↕</div><div>✖</div><div>✓</div><div><i>f</i>_x</div><div>=+ANZAHL(B3:B12)</div></div>				
	B	C	D	E
	Werte			
	0,706793		Anzahl Werte	10
	0,447353			
	0,855576			
	0,102575			
	0,989661			
	0,561972			
	0,656805			
	0,043952			
	0,299101			
	0,085281			

Was sind Funktionen: Analogie "Excel"

- **ANZAHL ()** ist eine Excel-Funktion, die die Anzahl der Elemente in einer Spalte zählt
- **B3:B12** sind die Eingabewerte
- **=ANZAHL (B3 : B12)** verarbeitet die Eingabewerte und zählt die Elemente
- **10** ist der Rückgabewert, d.h. das Ergebnis der Funktion

Wofür benötige ich Funktionen?

- **Wiederverwendbarkeit:** Funktionen ermöglichen die Wiederverwendung von Code, wodurch redundante Codestellen reduziert und die Effizienz verbessert wird.
- **Modularität:** Funktionen fördern die Aufteilung eines Programms in kleinere, unabhängige Bausteine, die leichter zu verstehen und zu warten sind.
- **Lesbarkeit:** Funktionen mit aussagekräftigen Namen verbessern die Lesbarkeit des Codes, da sie die Absicht und Funktionalität des Codeblocks klar kommunizieren.
- **Fehlerbehebung:** Durch die Verwendung von Funktionen können Fehler leichter identifiziert und isoliert werden, da der Code in überschaubaren Einheiten organisiert ist.

Schreiben von Funktionen in Python

Syntax:

- In Python werden Funktionen mit dem Schlüsselwort `def` definiert, gefolgt vom Funktionsnamen und Klammern `()`, die die Parameter enthalten.
- Jeder Funktionsname sollte aussagekräftig sein und die Aktion oder Berechnung, die sie ausführt, beschreiben.
- Die Parameter sind die Eingabewerte, mit denen die Funktion arbeitet.
- Nach der Parameterliste folgt ein Doppelpunkt `:`, der den Beginn des Funktionskörpers markiert.
- Der Funktionskörper sollte eingerückt sein, um den Code innerhalb der Funktion vom restlichen Code abzugrenzen.
- Eine Funktion kann Werte mit dem Schlüsselwort `return` zurückgeben, gefolgt vom Wert oder Ausdruck, der zurückgegeben werden soll.

Beispiel:

```
1 def berechne_gewinn(umsatz, kosten):    # Definition der Funktion
2     gewinn = umsatz - kosten           # Verarbeitung / Funktionskörper
3     return gewinn                     # Rückgabe des Ergebnisses
```

Ausführen von Funktionen in Python

- Um eine Funktion in Python aufzurufen, verwenden Sie den Funktionsnamen, gefolgt von Klammern () und den Argumenten, die an die Parameter der Funktion weitergegeben werden.
- Argumente sind die konkreten Werte, die an die Parameter einer Funktion übergeben werden, wenn sie aufgerufen wird.
- Die Reihenfolge der Argumente sollte der Reihenfolge der Parameter in der Funktionsdefinition entsprechen.
- Wenn die Funktion einen Wert zurückgibt, kann dieser Wert in einer Variable gespeichert oder direkt verwendet werden.

Beispiel: Funktion ausführen und Ergebnis direkt verwenden

```
1 umsatz = 10000
2 kosten = 5000
3 gewinn = berechne_gewinn(umsatz, kosten)
4 gewinn
```

Aufgabe 1

Aufgabenstellung:

Erstellen Sie eine Funktion, die den **Break-Even-Punkt** berechnet, an dem die Gesamtkosten eines Unternehmens gleich den Gesamteinnahmen sind. Die Funktion soll den Break-Even-Punkt in Form der Anzahl der verkauften Einheiten zurückgeben.

Anforderungen:

- Definieren Sie eine Funktion mit einem sinnvollen Namen, die drei Parameter akzeptiert: `fixkosten`, `variable_kosten_pro_einheit` und `preis_pro_einheit`.
- Geben Sie das Ergebnis als Ganzzahl zurück, da es sich um eine Anzahl von Einheiten handelt. Verwenden Sie die Funktion `round()` oder `int()` in Python, um das Ergebnis auf die nächste ganze Zahl zu runden bzw. die Nachkommastellen abzuschneiden.
- Rufen Sie die Funktion mit Werten Ihrer Wahl auf und speichern Sie das Ergebnis in einer Variable.

Lösung Aufgabe 1

Aufgabe 2

Aufgabenstellung:

Erstellen Sie eine Funktion, die einen Betrag in einer Ausgangswährung in einen Betrag in einer Zielwährung umrechnet, basierend auf dem gegebenen Wechselkurs.

Anforderungen:

- Definieren Sie eine Funktion mit dem Namen `umrechnung_waehrung`, die zwei Parameter akzeptiert: `betrag`, `wechselkurs`
- Geben Sie das Ergebnis als Fließkommazahl mit zwei Nachkommastellen zurück. Verwenden Sie die Funktion `round()` in Python, um das Ergebnis auf zwei Nachkommastellen zu runden.
- Rufen Sie die Funktion mit einem Betrag und einem Wechselkurs Ihrer Wahl auf und speichern Sie das Ergebnis in einer Variable.
- Geben Sie das Ergebnis mithilfe einer `print`-Anweisung aus, gefolgt von der Zielwährung.

(2) BA-Prozess: Einlesen und Aufbereiten von Daten

Großteil der Datenanalyse besteht aus der Aufbereitung der Daten

Warum Aufbereitung?

- erheblicher Teil der Arbeit bei der Datenanalyse entfällt auf die Vorverarbeitung und Bereinigung der Daten.
- Einige Schätzungen gehen davon aus, dass 50% bis 80% der Zeit, die für ein Datenanalyseprojekt aufgewendet wird, der Datenvorverarbeitung gewidmet ist.¹

Einige Gründe für den hohen Zeitaufwand bei der Vorverarbeitung sind:

- **Datenqualität:** Daten aus der realen Welt sind oft "dreckig" bzw. enthalten fehlende Werte, Inkonsistenzen und Fehler.
- **Datenintegration:** Das Zusammenführen von Daten aus mehreren Quellen erfordert häufig das Lösen von Konflikten bei Formaten, Einheiten oder Datenstrukturen.
- **Datentransformation:** Daten müssen oft transformiert, normalisiert oder aggregiert werden, um für die Analyse oder Modellierung geeignet zu sein.
- **Merkmalsbildung:** Das Erstellen neuer Merkmale aus den Rohdaten kann ein zeitaufwändiger Prozess sein, der Fachwissen und Kreativität erfordert.

1. siehe z.B. Dasu und Johnson (2003)



- Wir werden im Rahmen unserer Analyse(n) verschiedene Module / Bibliotheken und Funktionalitäten Pythons kennenlernen.
- Eine der wichtigsten Bibliotheken ist **pandas**. Wir bezeichnen **pandas** als das Schweizer Taschenmesser der Datenanalyse, weil es ein mächtiges und vielseitiges Werkzeug für die Datenanalyse und -verarbeitung in Python ist.
- Pandas bietet für nahezu alle gängigen Anforderungen - sei es Datenimport, Datenbereinigung, Datentransformation und -analyse oder sogar Visualisierung - nützliche Funktionalitäten.
- Die Palette an Funktionalitäten ist so umfangreich, dass wir diese unmöglich alle vorstellen können. Wir glauben auch nicht, dass dies didaktisch sinnvoll ist. Stattdessen werden wir geeignete Funktionen immer dann vorstellen, wenn wir sie für unsere Analyse benötigen. Viele der vorgestellten Module bieten Funktionalitäten an, die wir nicht nutzen. Es ist deshalb sinnvoll, dass Sie sich die Dokumentation der Funktionen anschauen, wenn Sie diese für Ihre Analyse benötigen.
- Die Dokumentation finden Sie *hier*.

Einleitung: Fallstudie

Die **Bau und Werken GmbH** wurde 1967 in Recklinghausen gegründet und hat dort bis heute ihren Sitz. Ursprünglich konzentrierte sich das Unternehmen auf kleinere kommunale Bauprojekte, entwickelte sich aber im Laufe der Zeit zu einem wichtigen Akteur in der kommunalen Bau- und Instandhaltungsbranche. Mit einem starken Netzwerk an Geschäftspartnern und Lieferanten führt das Unternehmen eine breite Palette von Projekten durch, vom Tiefbau bis hin zu städtischen U-Bahnen. Dabei legt es Wert auf Qualität und Kundenzufriedenheit, um einen ausgezeichneten Ruf in der Branche zu sichern.

Allerdings kämpft auch die Bau und Werken GmbH mit Projekten, die länger dauern und mehr kosten als geplant, was die Profitabilität beeinträchtigt. Geschäftsführer Maximilian Müller ist sich dieser Herausforderung bewusst und möchte das Problem lösen. Er erkennt, dass Bauverzögerungen und Kostenüberschreitungen die Profitabilität mindern, obwohl das Unternehmen grundsätzlich profitabel wirtschaftet. Müller plant, vergangene Projekte zu analysieren, um wiederkehrende Muster oder Faktoren zu identifizieren, die Verzögerungen und Kostenüberschreitungen verursachen.

Die Lösung dieser Herausforderung ist entscheidend für die Zukunft des Unternehmens, und Maximilian ist entschlossen, die Bau und Werken GmbH noch erfolgreicher und profitabler zu gestalten.

Einleitung: Datensatz

Der Datensatz beinhaltet Informationen zu den Projekten, die das Unternehmen in den letzten Jahren durchgeführt hat. Die Daten wurden von einem Mitarbeiter der Bau und Werken GmbH in einer Excel-Tabelle erfasst und anschließend in das CSV-Format exportiert. Der Datensatz beinhaltet folgende Informationen:

- **Project_ID**: eindeutige Identifikationsnummer des Projekts
- **Name Projekt**: Art des Projektes // Ort des Projektes (z.B. "Stadtspark // Eberhardtallee")
- **projekt_Beginn**: Baubeginn bzw. Beginn der Instandhaltungsarbeiten
- **Plan Bau fertig**: geplantes Bauende bzw. Instandhaltungsende
- **Fertig_IST**: tatsächliches Bauende bzw. Instandhaltungsende
- **Kosten Plan**: budgetierte Gesamtkosten des Projektes
- **Ist_Kosten**: tatsächliche Gesamtkosten des Projektes
- **Project_team**: internes Team, welches das Projekt bearbeitet hat

Link zum Datensatz: *Construction.csv*

Schritt 1: Problemstellung

- Was sollte Ziel unserer Analyse sein?
- Welche Fragestellungen wollen wir beantworten?

Wichtig:

- Formulieren der Problemstellung hilft, den Überblick zu behalten und die Analyse zu strukturieren.
- Problemstellung sollte immer der erste Schritt der Analyse sein.

Daten Einlesen und Überblick verschaffen

- Beginnen wir mit der Analyse, in dem wir die Daten zunächst einlesen und aufbereiten. Dazu verwenden wir `read_csv()`. Bei `read_csv()` handelt es sich um eine **Funktion**, der verschiedene Parameter übergeben werden können.
- Der wichtigste (und einzige zwingend notwendige) Parameter ist der **Pfad zur Datei**, die eingelesen werden soll.
- Der Pfad der Datei kann als Link oder als lokaler Dateipfad übergeben werden

Einlesen: CSV-Dateien können via `read_csv()` eingelesen werden

Einlesen via Link:

```
1 import pandas as pd
2
3 path = "https://www.dropbox.com/s/ov6mnmgzrydquie/Construction.csv?dl=1"
4 df = pd.read_csv(path)
```

Einlesen: CSV-Dateien können via `read_csv()` eingelesen werden

Einlesen via lokalem Pfad:

```
1 import pandas as pd
2
3 path = "_data/Construction.csv"
4 df = pd.read_csv(path)
```

Wichtig:

- Der Dateipfad ist relativ zum Ordner, in dem sich die Jupyter-Notebook-Datei befindet.
- Datei liegt im Beispiel im Ordner `_data`
- Ordner `_data` liegt im gleichen Ordner wie die Jupyter-Notebook-Datei

Daten Einlesen und Überblick verschaffen: grundsätzliches Vorgehen

- Pandas-Bibliothek importieren: Um Dateien mit Pandas einzulesen, muss zunächst die Pandas-Bibliothek in Python importiert werden: `import pandas as pd`
- Einlesefunktion verwenden: Verwenden Sie eine der spezifischen Einlesefunktionen `read_...()` von Pandas, um Dateien unterschiedlicher Formate einzulesen.

Funktionen für verschiedene Dateiformate:

- CSV-Dateien: `pd.read_csv('dateiname.csv')` - Liest eine CSV-Datei ein und gibt sie als DataFrame zurück.
- Excel-Dateien: `pd.read_excel('dateiname.xlsx')` - Liest eine Excel-Datei ein und gibt sie als DataFrame zurück. Benötigt die zusätzlichen Bibliotheken `openpyxl` und `xlrd`.
- JSON-Dateien: `pd.read_json('dateiname.json')` - Liest eine JSON-Datei ein und gibt sie als DataFrame zurück.
- HTML-Dateien: `pd.read_html('dateiname.html')` - Liest eine HTML-Datei ein, extrahiert Tabellen aus dem HTML und gibt sie als Liste von DataFrames zurück. Benötigt die zusätzliche Bibliothek `lxml`.
- SQL-Datenbanken: `pd.read_sql('sql_query', connection_object)` - Führt eine SQL-Abfrage auf einer Datenbank aus und gibt das Ergebnis als DataFrame zurück.
- ...

Ersten Überblick verschaffen

Vier Funktionen für den ersten Überblick:

- `head()`: Zeigt die ersten Zeilen des DataFrames an.
- `tail()`: Zeigt die letzten Zeilen des DataFrames an.
- Anzahl Zeilen anpassen: z.B. `df.head(10)` für die ersten zehn Zeilen
- `info()`: Zeigt Informationen zum DataFrame an, z.B. Anzahl Zeilen, Anzahl Spalten, Datentypen, ...
- `describe()`: Zeigt eine Zusammenfassung der numerischen Spalten an, z.B. Mittelwert, Standardabweichung, Minimum, Maximum, ...

Aufgabe 1: Ersten Überblick verschaffen

- Lese die Datei `Construction.csv` ein und speichere sie in der Variable `df`.
- Zeige die ersten 10 Zeilen des DataFrames an.
- Zeige die letzten 3 Zeilen des DataFrames an.
- Führen Sie die Funktion `info()` auf den DataFrame `df` aus. Was stellen Sie fest?
- bestimmen Sie die durchschnittlichen Kosten über alle Projekte

Lösung Aufgabe 1: Ersten Überblick verschaffen

Siehe Jupyter Notebook in der Veranstaltung.

Aufbereitung der Daten

Natürlich gibt es nicht *den* richtigen Weg, um Daten aufzubereiten. Jeder Datensatz ist unterschiedlich und nicht jede Analyse hat dieselben Anforderungen. Dennoch gibt es typische Aufgaben, die wir bei der Aufbereitung von Daten immer wieder vorfinden.

Wir werden uns im Folgenden einige dieser Aufgaben ansehen und die entsprechenden Lösungen diskutieren.

1. Variablennamen anpassen

2. Datentypen anpassen

3. Daten bereinigen

- Fehlende Werte
- Falsche Werte

Variablennamen anpassen

Warum?

- Nicht zwingend notwendig, Variablen umzubenennen
- Vorhandene Bezeichnungen: nicht intuitiv, inkonsistent, unklar
- Sinnvolle Umbenennung: schnellere Erkennung, intuitive Auswahl
- Subjektive Komponente bei Umbenennung
- Uneinheitliche Spaltennamen: Groß-/Kleinbuchstaben, Deutsch/Englisch, Leerzeichen/Unterstrich
- Erschwert intuitive Auswahl und Analyse

Datensatz:

Welche Veränderungen könnten wir vornehmen, um die Spaltennamen zu vereinheitlichen?

Variablennamen anpassen

Beispielhafte Anpassungen:

- alle Spalten in Kleinbuchstaben umwandeln
- alle Leerzeichen durch `_` ersetzen
- alle Begriffe `project` durch `projekt` ersetzen
- wo sinnvoll kürzere Begriffe verwenden
- Begriffe wie `Plan` und `Ist` einheitlich verwenden

Python:

Wir können die Spalten mit der Funktion `rename()` umbenennen. Die Funktion `rename()` erwartet als Parameter ein sog. `Dictionary`, in dem wir die alten Spaltennamen als Schlüssel und die neuen Spaltennamen als Werte angeben.

```
1 df.rename(columns={'alter_spaltenname': 'neuer_spaltenname'})
```

Aufgabe 2: Variablennamen anpassen

- passen Sie die Spaltennamen des DataFrames `df` an.
- nehmen Sie dabei die Anpassungen auf der vorherigen Folie vor.

Lösung Aufgabe 2: Spaltennamen anpassen

► Code

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

Datentypen anpassen

Wir haben bereits gesehen, dass unser Datensatz nach dem Einlesen zwei Datentypen enthält: `object` und `float64`. **Sind Datentypen für unsere Analyse passend?**

Diskussion der Datentypen:

- `id`: die Spalte beinhaltet Buchstaben, Zeichen und Zahlen (z.B. P-62602). Der Datentyp `object` ist also passend.
- `name`: die Spalte beinhaltet Text. Der Datentyp `object` ist also passend.
- `beginn`: die Spalte beinhaltet Datumswerte. Der Datentyp `object` ist nicht passend. Wir sollten den Datentyp in ein Datumsformat umwandeln.
- `ende_plan`: die Spalte beinhaltet Datumswerte. Der Datentyp `object` ist nicht passend. Wir sollten den Datentyp in ein Datumsformat umwandeln.
- `ende_ist`: die Spalte beinhaltet Datumswerte. Der Datentyp `object` ist nicht passend. Wir sollten den Datentyp in ein Datumsformat umwandeln.
- `kosten_plan`: die Spalte beinhaltet Zahlen. Der Datentyp `float64` erscheint passend.
- `kosten_ist`: die Spalte beinhaltet Zahlen. Der Datentyp `float64` erscheint passend.
- `team`: die Spalte beinhaltet Text. Der Datentyp `object` ist also passend.

Fazit:

Anpassen der Datumsspalten in ein Datumsformat erscheint sinnvoll.

Datentypen anpassen: warum Datumsformate?

- **Zeitreihenanalyse:** Konvertierung in Datumsformat ermöglicht einfache Analysen von Zeitreihendaten.
- **Datumsarithmetik:** Berechnungen mit Datums- und Zeitangaben werden erleichtert, z.B.:

```
1 df['differenz'] = df['enddatum'] - df['startdatum']
```

- **Zeitbezogene Funktionen:** Zugriff auf spezielle Funktionen, die auf Datums- und Zeitangaben anwendbar sind, z.B.:

```
1 df['spalte'].dt.month
```

Datentypen anpassen in Python

Wie können wir Datentypen in Python anpassen?

Es gibt verschiedene Wege, z.B.:

1 `astype()`: Verwenden Sie die `astype()` Methode, um den Datentyp einer Spalte zu ändern.

```
1 df['spalte'] = df['spalte'].astype('neuer_datentyp')`
```

Für `neuer_datentyp` können Sie z.B. `int64`, `float64`, `datetime64` oder `category` verwenden.

2 `to_datetime()`: Konvertieren Sie Spalten, die Datums- und Zeitangaben enthalten, in den `datetime64`-Datentyp.

```
1 df['spalte'] = pd.to_datetime(df['spalte'])
```

Aufgabe 3: Spalten in Datumsformat umwandeln

Wandeln Sie die Spalten `beginn`, `ende_plan` und `ende_ist` in das Datumsformat um.

Lösung Aufgabe 3: Spalten in Datumsformat umwandeln

► Code

Mit der Funktion `info()` können wir uns die Datentypen der Spalten anzeigen lassen.

```
1 df.info()
```

Alternativ können wir die Datentypen der Spalten mit der Funktion `dtypes` anzeigen lassen.

```
1 df.dtypes
```

Daten bereinigen

Wir schauen uns im Folgenden einige Bereinigungsmaßnahmen an, die wir an unserem Datensatz vornehmen können.

- **Fehlende Werte** identifizieren und behandeln
- **Duplikate** identifizieren und behandeln
- **Fehlerhafte Werte** identifizieren und behandeln

Daten bereinigen: Fehlende Werte

- wir haben bereits gesehen (siehe `df.info()`), dass nicht alle Spalten mit Werten gefüllt sind.
- wir wissen wir nicht, wo sich diese fehlenden Werte befinden.
- Information ist aber wichtig, um zu entscheiden, wie wir mit den fehlenden Werten umgehen:
 - Beobachtungen mit fehlenden Werten entfernen?
 - fehlende Werte durch andere Werte ersetzen?
 - sind fehlende Werte zufällig verteilt oder gibt es einen Zusammenhang mit anderen Variablen?
 - sind fehlende Werte überhaupt relevant für die Analyse?

- ## Beispiel

[illegible]

	id	name	beginn	ende_plan	ende_ist	kosten_plan	kosten_ist	team
10018	False	False	False	False	False	False	False	False

10019 rows × 8 columns

Daten bereinigen: Fehlende Werte

Ein typisches Analysemuster ist es, alle Zeilen (d.h. Beobachtungen) zu selektieren, die fehlende Werte enthalten.

1. Wir identifizieren alle Elemente, die fehlende Werte enthalten, indem wir die Funktion `isna()` auf den gesamten Datensatz anwenden.
2. Wir selektieren alle Zeilen, die fehlende Werte enthalten, indem wir die Funktion `any()` auf die Ergebnisse der Funktion `isna()` anwenden.

```
1 mask = df.isna().any(axis=1) # mask = True, wenn Zeile fehlende Werte enthält
2 mask.head()
```

```
0    False
1    False
2    False
3    False
4    False
dtype: bool
```

Daten bereinigen: Fehlende Werte

```
1 df[mask].head()
```

	id	name	beginn	ende_plan	ende_ist	kosten_plan	kosten_ist	team
103	TH-593312	Elektroarbeiten // Börnerplatz	2018-05-07	2018-07-11	2018-07-17	NaN	69375.23	Team 3
740	NL-673150	Baumbestand // Loosplatz	2016-12-30	NaT	2016-12-31	16349.93	17892.16	Team 3
896	LM-748355	Baumbestand // Martha-Bohlander-Ring	2019-10-17	2019-10-22	2019-10-24	NaN	40637.75	Team 4
996	AY-257302	Elektroarbeiten // Eimerstr.	2014-02-03	2014-02-28	2014-03-01	NaN	112867.21	Team 2
1059	RR-446242	Elektroarbeiten // Florentine-Kambs-Allee	2013-06-16	2013-07-24	2013-07-23	NaN	164730.94	Team 1

Daten bereinigen: Fehlende Werte

Da wir keine weiteren Informationen zu den fehlenden Daten haben und auch keinen systematischen Fehler entdecken können, der zu den fehlenden Daten führt, werden wir die fehlenden Werte einfach entfernen. Dies können wir mit der Funktion `dropna()` erreichen.

```
1 df = df.dropna()
2 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9998 entries, 0 to 10018
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  -
 0   id              9998 non-null   object
 1   name            9998 non-null   object
 2   beginn          9998 non-null   datetime64[ns]
 3   ende_plan       9998 non-null   datetime64[ns]
 4   ende_ist        9998 non-null   datetime64[ns]
 5   kosten_plan     9998 non-null   float64
 6   kosten_ist      9998 non-null   float64
 7   team            9998 non-null   object
dtypes: datetime64[ns](3), float64(2), object(3)
memory usage: 703.0+ KB
```


Daten bereinigen: Duplikate

Beispieldatensatz mit Duplikaten

```
1 # Erstelle Beispiel-Datensatz
2 example = pd.DataFrame({'Spalte A': [1, 2, 3, 2, 4, 5],
3                           'Spalte B': ['Hund', 'Katze', 'Vogel', 'Katze', 'Fisch', 'Vogel'],
4                           'Spalte C': [19, 43, 1, 43, 127, 21]})
5 example
```

	Spalte A	Spalte B	Spalte C
0	1	Hund	19
1	2	Katze	43
2	3	Vogel	1
3	2	Katze	43
4	4	Fisch	127
5	5	Vogel	21

Daten bereinigen: Duplikate

- können die Methode `duplicate()` nutzen, um Duplikate in Zeilen eines DataFrames zu **identifizieren**.
- gibt eine Boolesche Reihe (True, False, True etc.) zurück, die angibt, ob eine Zeile ein Duplikat ist oder nicht
- Standardmäßig werden dabei alle Spalten berücksichtigt, jedoch kann auch eine Teilmenge von Spalten angegeben werden, die bei der Suche nach Duplikaten berücksichtigt werden soll (via `subset`)
- `keep` gibt an, welche Duplikate in der Ergebnisreihe enthalten sein sollen. Standardmäßig werden alle Duplikate außer dem ersten entfernt.

Variante 1 Variante 2 Variante 3 Variante 4

```
1 example.duplicated()
```

```
0    False
1    False
2    False
3     True
4    False
5    False
dtype: bool
```

Daten bereinigen: Duplikate

Wir müssen nun entscheiden, ob wir diesen Eintrag entfernen wollen oder nicht. In diesem Fall scheint es sinnvoll, den doppelten Eintrag zu entfernen, da wir in der Regel davon ausgehen, dass die Daten nur einmal erfasst wurden. Wir können dies mit der Methode `drop_duplicates()` tun. Diese entfernt standardmäßig die zweite Zeile einer doppelten Beobachtung. Wir können dies jedoch ebenfalls mit dem Parameter `keep` ändern (d.h. alle Duplikate entfernen oder nur die erste oder letzte Zeile).

```
1 example.drop_duplicates()
```

Spalte A Spalte B Spalte C			
0	1	Hund	19
1	2	Katze	43
2	3	Vogel	1
4	4	Fisch	127
5	5	Vogel	21

Daten bereinigen: Duplikate

Lassen Sie uns nun analysieren, ob es überhaupt Duplikate in unserem Datensatz gibt und dafür alle Spalten berücksichtigen. Mit der Funktion `sum()` können wir die Anzahl der Duplikate ermitteln.

```
1 df.duplicated().sum()
```

19

Fazit

Es gibt 19 Duplikate in unserem Datensatz. Schauen wir uns diese an.

Aufgabe 4: Duplikate anzeigen

- Schreiben Sie einen Code, der alle Duplikate in unserem Datensatz anzeigt.
- lassen Sie die ersten 6 Zeilen ausgeben (d.h. die ersten 3 Duplikate)

Lösung 4: Duplikate anzeigen

Unsortiert

Sortiert

► Code

	id	name	beginn	ende_plan	ende_ist	kosten_plan	kosten_ist	team
34	KN-987191	Landschaftsbau // Helga-Krause-Allee	2016-04-01	2016-04-04	2016-04-06	114862.28	125114.70	Team 1
447	SW-908572	Elektroarbeiten // Karl-Heinrich-Ring-Allee	2020-03-26	2020-06-07	2020-06-13	43298.09	60629.52	Team 3
590	ZE-101640	Landschaftsbau // Henry-Steinberg-Gasse	2013-10-25	2013-11-20	2013-11-26	99605.32	118141.92	Team 3
1337	DH-600607	Straßenbau // Tröstplatz	2015-01-06	2015-01-25	2015-01-26	189536.10	210407.91	Team 1
1356	JI-333627	Ubahn // Karl-Friedrich-Eigenwillig-Platz	2014-12-19	2015-03-18	2015-08-18	322147.45	331683.54	Team 1
1436	VV-651307	Stadtpark // Reza-Cichorius-Straße	2017-10-18	2018-01-02	2018-03-21	195746.98	203686.56	Team 1

Daten bereinigen: Duplikate

In unserem Falle erscheint es sinnvoll, die Duplikate zu entfernen. Wir können dies mit der Funktion `drop_duplicates()` tun. Diese Funktion entfernt standardmäßig die zweite Zeile einer doppelten Beobachtung. Wir können dies jedoch ebenfalls mit dem Parameter `keep` ändern (d.h. alle Duplikate entfernen oder nur die erste oder letzte Zeile).

```
1 df = df.drop_duplicates()
```

Wir können nun nochmals überprüfen, ob es noch Duplikate gibt und stellen fest, dass unsere Daten nun auf Dopplungen bereinigt sind.

```
1 df.duplicated().any()
```

`False`

Deskriptive Analyse

Falsche Werte

Ein wichtiger Schritt bei der Datenbereinigung ist die Überprüfung der Daten auf (offensichtlich) falsche Werte. Nicht immer ist bereits bei der Aufbereitung der Daten erkennbar, ob ein Wert plausibel ist oder nicht. Häufig werden falsche Werte erst bei der Analyse der Daten sichtbar. Jedoch können und sollten einige Plausibilitätsprüfungen bereits bei der Aufbereitung der Daten durchgeführt werden.

Im vorliegenden Datensatz können wir z.B. folgende **Plausibilitätsprüfungen** durchführen:

- 1 Ist der Wert der Spalte `id` eindeutig?
- 2 Sind die Werte für Kosten (`kosten_plan` und `kosten_ist`) plausibel, d.h. sind die Kosten positiv (bzw. haben alle das gleiche Vorzeichen)?
- 3 Sind die Werte für die Datumsspalten (`beginn`, `ende_plan` und `ende_ist`) plausibel, d.h. (i) liegen die Werte in der Vergangenheit und (ii) ist das Enddatum nach dem Startdatum?

Falsche Werte: Eindeutigkeit der ID

Wie können wir überprüfen, ob die `id` eindeutig ist?

Falsche Werte: Eindeutigkeit der ID

Vorgehen:

- gib die Anzahl der eindeutigen Werte der Spalte `id` aus
- vergleiche die Anzahl der eindeutigen Werte mit der Anzahl der Zeilen des Datensatzes

In Python:

```
1 unique_ids = df['id'].unique()
```

Wir können nun überprüfen, ob die Anzahl der eindeutigen Werte der Anzahl der Zeilen entspricht.

```
1 len(unique_ids) == len(df)
```

`True`

Da der Wert `True` zurückgegeben wird, ist die `id` eindeutig und wir können uns sicher sein, dass keine Projekt-ID mehrfach vergeben wurde.

Falsche Werte: Kosten

Sind die Werte für Kosten (`kosten_plan` und `kosten_ist`) plausibel, d.h. sind die **Kosten positiv** (bzw. haben alle das gleiche Vorzeichen)?

Variante 1

```
1 # Mit "mask" arbeiten
2 mask = df['kosten_plan'] < 0
3 df[mask]
4
5 # Ohne "mask" arbeiten
6 df[df['kosten_plan'] < 0]
```

Variante 2

```
1 # Datensatz nach Bedingung filtern
2 df.query('kosten_plan < 0')
```

Falsche Werte: Kosten

Der einfachste Weg, die Zeilen mit negativen Vorzeichen zu eliminieren, ist es den Datensatz mit `query` zu filtern. Wir können die Bedingung `kosten_plan >= 0` nutzen, um alle Zeilen auszugeben, die einen positiven Kostenwert haben.

```
1 df = df.query('kosten_plan >= 0')
```

Wir könnten nun die gleiche Überprüfung für die Spalte `kosten_ist` durchführen. Stattdessen filtern wir aber alle Zeilen, die einen negativen Kostenwert haben, aus dem Datensatz heraus.

```
1 df = df.query('kosten_ist >= 0')
```

Wir hätten die beiden letzten Schritte auch in einem Schritt durchführen können, indem wir die Bedingungen mit einem `&` verknüpft hätten.

```
1 df = df.query('kosten_plan >= 0 & kosten_ist >= 0')
```

Falsche Werte: Datum

Bei der nun folgenden Überprüfung kommt uns zugute, dass wir die Spalten bereits in den Datentyp `datetime` konvertiert haben. Wir können nun die Werte für die Spalten `beginn`, `ende_plan` und `ende_ist` miteinander vergleichen.

Wir können zwei Überprüfungen durchführen:

1. Sind alle Projekte bereits beendet?
2. Liegt das Datum für `ende_ist` nach dem Datum für `beginn`?

Falsche Werte: Datum

1. Sind alle Projekte bereits beendet?

```
1 df['ende_ist'].max()
```

Timestamp('2023-03-09 00:00:00')

Das letzte Projekt wurde in der Vergangenheit beendet, d.h. wir können sicher sein, dass alle Projekte bereits beendet sind (und somit annehmen, dass auch alle Kosten bereits berücksichtigt wurden).

Deskriptive Analyse

Falsche Werte: Datum

2. Liegt das Datum für ende_ist nach dem Datum für beginn?

```
1 df.query('ende_ist <= beginn').head()
```

	id	name	beginn	ende_plan	ende_ist	kosten_plan	kosten_ist	team
1251	TG-362669	Baumbestand // Cornelia-Schönland-Gasse	2017-08-16	2017-08-16	2017-08-16	55507.48	69877.32	Team 2
1584	CF-360814	Baumbestand // Adlerallee	2014-10-18	2014-10-19	2014-10-18	80258.30	80779.37	Team 4
2456	JV-632251	Baumbestand // Gabriel-Gude-Platz	2014-11-03	2014-11-03	2014-11-03	31973.47	35215.93	Team 3
2764	WS-799566	Stadtpark // Hesseplatz	2014-04-21	2014-04-22	2014-04-21	175422.45	270521.71	Team 2
3319	QC-047279	Stadtpark // Heiko-Pohl-Straße	2020-06-24	2020-07-12	2020-06-24	92345.10	117155.99	Team 3

Falsche Werte: Datum

Findings

- es handelt sich ausschließlich um Projekte, die am gleichen Tag begonnen und beendet wurden.
- prinzipiell nicht ausgeschlossen, jedoch erscheint dies - zumindest für einige Projekte - sehr unwahrscheinlich, da gleichzeitig hohe Kosten veranschlagt wurden, was auf eine längere Projektlaufzeit schließen lässt.

Was tun?

- da wir uns in diesem Fall nicht sicher sind, ob die Daten korrekt sind, werden wir die Projekte nicht eliminieren.
- aber: füge eine neue Spalte hinzu, die die Dauer des Projektes in Tagen angibt. Wir können dann im Rahmen unserer Analyse die Projekte mit einer Projektdauer von 0 Tagen herausfiltern und analysieren, ob dies unsere Ergebnisse beeinflusst.

```
1 df['dauer'] = df['ende_ist'] - df['beginn']
```


One more thing: Index zurücksetzen

Index des Datensatzes via `reset_index()` zurücksetzen

- Saubere Indizierung: `reset_index()` stellt eine fortlaufende und eindeutige Indizierung sicher, insbesondere nach Sortierung, Filterung oder Gruppierung.
- Vermeidung von Fehlern: Durch das Zurücksetzen des Index werden mögliche Fehler vermieden, die durch inkonsistente oder fehlende Indexwerte entstehen können.

In Python:

```
1 df = df.reset_index(drop=True)
```

- `drop=True` sorgt dafür, dass die alte Index-Spalte nicht mit in den Datensatz übernommen wird.

Deskriptive Analyse

Bereinigter Datensatz

1 df.head()

	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

Übersicht genutzter Funktionen

Wir haben nun einige wesentliche und offensichtliche Aufbereitungsschritte durchlaufen und können mit der eigentlichen Analyse der Daten beginnen.

Es wurden folgende Funktionen und Methoden genutzt:

- `read_csv()`: Einlesen der Daten
- `head()`: Anzeigen der ersten Zeilen
- `tail()`: Anzeigen der letzten Zeilen
- `info()`: Anzeigen der Datentypen und der Anzahl der nicht fehlenden Werte
- `rename()`: Umbenennen der Spalten
- `to_datetime()`: Konvertieren der Spalten in den Datentyp `datetime`
- `isna()`: Überprüfen, ob Werte fehlen
- `any()`: Überprüfen, ob Spalte oder Zeile `False` enthält
- `dropna()`: Eliminieren von Zeilen mit fehlenden Werten
- `duplicated()`: Überprüfen, ob es Duplikate gibt
- `drop_duplicates()`: Eliminieren von Duplikaten
- `unique()`: Ermitteln der einzigartigen Werte
- `query()`: Filtern von Zeilen nach Bedingung
- `max()`: Ermitteln des höchsten Wertes

Quellen

Dasu, Tamraparni, und Theodore Johnson. 2003. *Exploratory data mining and data cleaning*. Wiley series in probability and statistics. New York: Wiley-Interscience.

