Der ETL-Prozess mit Python

Fangen wir mit einem Beispiel an, dass in der Form wahrscheinlich hundertfach in deutschen Firmen pro Tag auftritt: Der Chef benötigt für die Aufsichtsratsitzung die aktuellsten Kennzahlen aus der CRM-Datenbank und ruft dafür beim Verfahrensverantwortlichen an. Der kann über die Oberfläche allerdings nur Standardberichte erzeugen und gibt die Anfrage an den Datenbank-Admin weiter. Der meldet sich auf der Konsole an und gibt dort den Befehl ein: "Select * from all > bericht_für_chef.csv". Etwas verschämt gibt der Verfahrensverantwortliche die 10Mb große CSV-Datei an den Chef weiter und der hält die Aufsichtsratssitzung dann ohne die aktuellsten Kennzahlen ab.

Der ETL-Prozess bildet für diesen und für ähnliche gelagerte Fälle die Brücke zwischen dem Business und den eigentlichen Daten.
ETL steht dabei für **Extract, Transform and Load** und beschreibt die Aufbereitung von Daten in eine adressatengerechte Form.

Es handelt sich dabei um ein praktisches Einsatzgebiet der deskriptiven Statistik, bei dem der Ist-Zustand der Daten betrachtet wird. Davon abzugrenzen ist die präskriptive Statistik, bei der anhand der bestehenden Daten Prognosen über die Zukunft abgehalten werden. Die präskriptive Statistik ist das klassische Einsatzgebiet der ML-Mechanismen und von Datenwissenschaftlern. Der ETL-Prozess obliegt dabei eher dem Datenanalysten und die strukturierte Speicherung der Daten dem Datenengineer.

In unserem Beispiel entspricht der Verfahrensverantwortliche bzw. der Datenbank-Admin dem Datenengineer. Beide haben sich auch mit mäßigem Erfolg als Datenanalyst versucht. Da der Chef keine Prognosen über die zukünftige Entwicklung verlangt hat, war kein Datenwissenschaftler notwendig. Zum Glück!

Bei der Beschreibung könnte man meinen, dass der Datenanalyst der kleine Bruder des Datenwissenschaftlers ist. Tatsächlich verbringen aber auch Datenwissenschaftler den größten Teil ihres Arbeitstages mit der Aufbereitung von Daten. Die Leistungsfähigkeit eines ML-Modelles hängt ja in erster Linie von der Qualität der Daten ab. Zudem muss es sich bei einem Datenanalysten oder einem Datenwissenschaftler auch nicht immer um einen dedizierten Mitarbeiter handeln. Heutzutage bringen gängige Office-Produkte schon Mechanismen zu fortgeschrittenen Datenanalysen mit. Damit können auch Verfahrensverantwortliche oder normale Mitarbeiter Tätigkeiten eines Datenanalysten wahrnehmen. Allerdings haben die am Markt verfügbaren Tools ein gemeinsames Problem: Neben den Lizenz -und Hostingkosten der entsprechenden Hard -und Software erfordern die komplexen Oberflächen eine lange Einarbeitung und trotzdem wird für fortgeschrittene Analysen auf ein Programmierbackend zurückgegriffen. Von daher kann es sinnvoll sein, sich von Anfang an der Datenanalyse über die Programmierung zu nähern. Man erhält damit die volle Flexibilität, um alle nur erdenklichen Aufgaben auszuführen. Zudem sind die entsprechenden Programmiersprachen und Entwicklungsumgebungen kostenlos verfügbar, was einen leichten Einstieg ermöglicht.

Ich würde jetzt keine Diskussion anfangen, ob Python, R oder SPSS die beste Programmiersprache für die Datenanalyse ist. Da spielen sicherlich auch subjektive Faktoren eine Rolle. Ich habe dieses Tutorial für python geschrieben, da die Sprache aus meiner Sicht den generellsten Ansatz bietet und man neben Datenanalysen auch fortgeschrittene ML-Mechanismen umsetzen kann. Neben der Standardbibliothek nutze ich dabei hauptsächlich pandas und numpy. Die meisten meiner Beispiele gehen von Daten in einer Tabellenform aus. Von daher ist das Dataframe von pandas unsere bevorzugte Spielwiese.

Ich bin ein großer Freund davon, Funktionen aus etablierten Bibliotheken zu nutzen, anstatt sie nach zu programmieren. Für das Verständnis der einzelnen Funktionen mag der alternative Ansatz sinnvoller sein. Diese Tutorial soll aber eher die Funktion eines Kochbuches erfüllen, in dem mit vielen Beispielen ohne große Erklärung einer der kürzeren Wege zum Ziel dargestellt wird. Es kann gut sein, dass meine Beispiele nicht immer den kürzesten und auch nicht immer den pythonic, sondern eher den hemdsärmeligen Weg zum Ziel abbilden. Datenanalysen sind in der Regel aber auch keine eingefleischten Programmierer, die in Codestrukturen, Klassen und Zeigern denken. Datenanalysten denken in Datenstrukturen, die Programmiersprache ist eher Mittel zum Zweck.

Durch den Fokus auf den ETL-Prozess werden auch nur wenige ML oder andere KI-Mechanismen dargestellt. Es wäre auch nicht gerecht, die mächtige und leistungsfähige scikit-learn-Bibliothek hier nur am Rand zu erwähnen. Dafür würde sich im Nachgang ein eigenes Tutorial lohnen.

Ein Warmup im ETL-Prozess

Fangen wir mit zwei kleinen aber vollständigen Beispielen zum Aufwärmen an, im wahrsten Sinne des Wortes!

```
# Zuerst importieren wir die Module, die uns im Laufe des Tutorials noch häufiger
begegnen werden.
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Beim Extrahieren von Daten werden Daten aus beliebigen Quellen in eine programmiertechnisch auswertbare Form gebracht. Nachfolgend der wahrscheinlich einfachste Fall der Datenextraktion. Eine CSV-Datei mit Klimadaten wird von einer URL in ein Dataframe geladen.

```
url="https://climate.copernicus.eu/sites/default/files/2020-
03/ts_1month_anomaly_Global_ea_2t_202002_v01.csv"
df=pd.read_csv(url,header=1)

print (df.head())
    Month  global  European
0  197901 -0.2309   -2.5160
1  197902 -0.4906   -1.4645
2  197903 -0.2922   -0.2274
3  197904 -0.3259   -1.9469
4  197905 -0.2858    0.0266
```

Die Daten sind glücklicherweise schon in einer maschinenlesbaren Form, nur die Spalte "Month" verlangt nach einer leichten Überarbeitung im Prozessschritt Datentransformation.

```
with np.nditer(df["Month"],op_flags=['readwrite'])as it:
    for x in it:
        x[...]=str(x)[:-2]
```

Beim Laden der Daten sollten wir anstreben, neue Erkenntnisse aus den Daten zu generieren. Eine einfache statistische Auswertung würde dafür schon reichen.

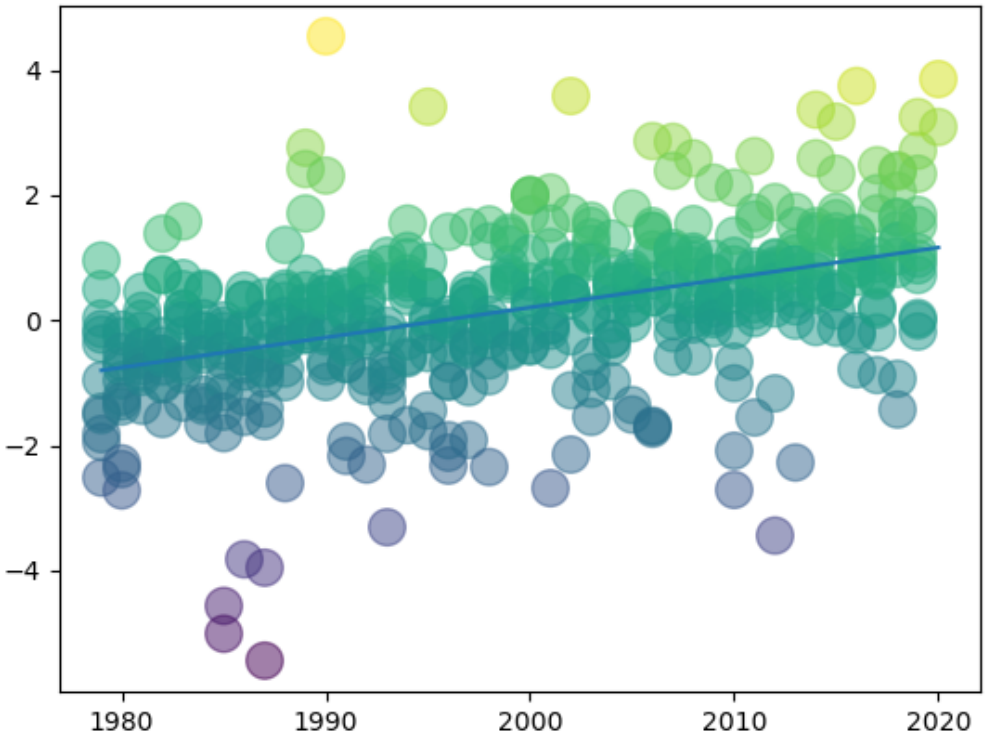
```
print (df.describe())

              Month       global     European
count   494.000000  494.000000  494.000000
mean   1999.085020    0.083307    0.160730
std      11.895286    0.269730    1.257701
min    1979.000000   -0.497600   -5.442700
25%    1989.000000   -0.120025   -0.505550
50%    1999.000000    0.062450    0.216600
75%    2009.000000    0.269800    0.870825
max    2020.000000    0.885000    4.541200
```

Quasi selbsterklärend ist die Spalte "Month", in der nach der Datentransformation nur noch die Jahreszahlen übrig geblieben sind. Natürlich haben wir hier genauso viele Datensätze wie in den anderen Spalten auch. Die Standardabweichung von fast 12 ist ebenfalls einsichtig, da jedes Jahr 12 Monate hat. Der etwas krumme Wert kommt zustande, da der Datensatz im Februar 2020 endet.

In diesem Beispiel interessiert uns aber insbesondere die Spalte "European". Es handelt sich um insgesamt 494 Temperaturwerte für jeden Monat seit 1979. Der Mittelwert von +0.16 Grad in den letzten 40 Jahren würde einen entsprechenden durchschnittlichen Temperaturanstieg suggerieren. Allerdings ist in dieser Ausgabe schon ersichtlich, dass bei den min -und max-Werten erhebliche Ausreißer dabei sind.

Visualisieren wir das Beispiel, um die Entwicklung zu verdeutlichen.

```
plt.scatter(df["Month"],df["European"],c=df["European"],s=200,alpha=0.5)

m, b = np.polyfit(df["Month"],df["European"], 1)
plt.plot(df["Month"], m*df["Month"] + b)
plt.show()
```



Die Grafik bestätigt unseren Verdacht, dass es zwar eine starke Aggregation der Datenpunkte um die Regressionslinie herum gibt, gerade um 1990 herum aber auch starke Ausreißer nach oben und nach unten vorhanden waren. Da die Regressionslinie aber schon einen leichten Aufwärtstrend verzeichnet, könnte man dieses Bild als den Beweis für den Klimawandel ansehen. Allerdings gehört die Interpretation der Ergebnisse des ETL-Prozesses nicht zu den Aufgaben des Datenanalysten. Der Datenanalyst bereitet nur die Grundlage für derartige Entscheidungen und Aussagen vor. Von daher kann aus dem vorliegenden Bild ein Klimatologe wahrscheinlich deutlich mehr herauslesen, als wir.

In einem weiteren einfachen Beispiel analysieren wir den CO2-Ausstoß von 1950-2018 über die Tabelle eines weiteren Wikipedia-Eintrages.

```
url2="https://de.wikipedia.org/wiki/Liste_der_gr%C3%B6%C3%9Ften_Kohlenstoffdioxidemittenten"

df_html2=pd.read_html(url2)
df_weltemissionen=df_html2[-3].rename(columns={"Jahr": "Jahr", "Emissionen": "Emission"}).append([df_html2[-2],df_html2[-1]],ignore_index=True)

plt.plot(df_weltemissionen["Jahr"],df_weltemissionen["Emission"])
plt.xlabel("Zeitraum",fontsize=12)
plt.ylabel("Millionen Tonnen CO2",fontsize=12)
plt.title("Emissionen in Millionen Tonnen CO2",fontsize=13)
plt.show()
```

Die Kurve ist schon prägnant genug, noch prägnanter wird es mit der Darstellung der Korrelationskoeffizienten.

```
print(df_weltemissionen.corr(method="spearman"))

              Jahr  Emission
Jahr      1.000000  0.997123
Emission  0.997123  1.000000
```

Mit 0,997 eine fast perfekte positive Korrelation zwischen den beiden Variablen Zeit und CO2-Ausstoß. Trotzdem sind die Werte nicht so eindeutig, wie sie auf den ersten Blick erscheinen und bestätigen eher den Leitsatz: "Glaube keiner Statistik, die du nicht selber gefälscht hast!"

- Korrelation ist nicht gleichbedeutend mit Kausalität. Die Daten sagen nur aus, dass der CO2-Ausstoß in den letzten 70 Jahren zugenommen hat, aber nichts zu den Ursachen und Auswirkungen.
- Die abgeschnittene Y-Achse ist ein typischer Fehler in der statistischen Darstellung. Für eine objektive Darstellung sollte die Y-Achse bei 0 beginnen.
- Es wird nur die Entwicklung der letzten 70 Jahre angezeigt, was in klimatischen Zeiträumen ein Wimpernschlag ist. Vielleicht sind solche Schwankungen normal. Das würde sich aber erst zeigen, wenn man zum Vergleich den CO2-Ausstoß der letzten 2000 Jahre betrachtet.

Ich möchte mit diesen Beispielen nicht die Glaubwürdigkeit des Klimawandels leugnen oder bestätigen. Ich möchte damit aber zeigen, wie schwer es ist, aus scheinbar völlig objektiven Daten auch objektive Schlussfolgerungen zu ziehen. Und auch wenn der Datenanalyst nicht derjenige ist, der anhand der Ergebnisse Entscheidungen trifft, beeinflusst er durch die Art der Darstellung erheblich die Entscheidung.
Dieser Verantwortung sollte sich jeder Datenanalyst bewusst sein.

Datenextraktion mit Python

Der erste Schritt im ETL-Prozess ist die Datenextraktion, also die Verbindung mit einer Datenquelle und das Laden in eine Datenstruktur innerhalb des Programmes. Ziel dieses Prozessschrittes ist, das die Daten in einem Format vorliegen, dass weiter verarbeitet werden kann. Folgende Vorgehensweisen kommen dabei regelmäßig vor:

- Laden in ein Dataframe beim Vorliegen von Daten in einer Datenbank -oder Tabellenstruktur.
- Laden in eine Series oder ein Array beim Vorliegen von 1-dimensionalen Daten.
- Das zeilenweise Einlesen beim Einlesen von Webseiten oder unstrukturierten Texten

Datenextraktion von Webseiten

Es kommt nicht oft vor, dass die Daten im CSV -oder XlSX-Format auf einer Webseite zum herunterladen angeboten werden. Häufig sind die Daten im HTML-Code versteckt und müssen von uns für die weitere Verarbeitung extrahiert werden. Im einfachsten Fall sind die Daten dabei in einer HMTL-Tabelle untergebracht und können von pandas mit wenigen Zeilen eingelesen werden.

```
# Wir laden zuerst die Tabellen von der Wikipedia-Seite ein, die die
Bevölkerungsentwicklung der Menschheit darstellt
import pandas as pd
import requests

url="https://de.wikipedia.org/wiki/Bevölkerungsentwicklung"
r=requests.get(url)
df_list=pd.read_html(r.text)
```

```
#dann schauen wir, wieviele Tabellen pandas gefunden hat
print(len(df_list))
4

#und können uns die Tabelle genauer anschauen, die uns interessiert
print(df_list[0])

     Jahr  Bevölkerung(in Mrd.)  Wachstumsrate(% pro Jahr)  Zuwachs(Mio. pro
Jahr)  Alters-durchschnitt
0    1950                   253                         18
471                  235
1    1960                   303                         19
606                  227
2    1970                   369                         20
760                  215
3    1980                   445                         18
829                  226
4    1990                   532                         15
842                  241
5    2000                   613                         12
773                  263
6    2010                   692                         12
817                  285
7   20202                   772                          9
733                  310
8   20302                   842                          7
637                  332
9   20402                   904                          6
540                  346
10  20502                   955                          5
431                  361

# Wir vergewissern uns noch einmal, ob die Daten wirklich in dem von uns
erwarteten Format vorliegen
print(type(df_list[0]))

<class 'pandas.core.frame.DataFrame'>

#Ok, schauen wir uns das Dataframe etwas genauer an
print(df_list[0].index)

RangeIndex(start=0, stop=11, step=1)

print(df_list[0].columns)

Index(['Jahr', 'Bevölkerung(in Mrd.)', 'Wachstumsrate(% pro Jahr)',
       'Zuwachs(Mio. pro Jahr)', 'Alters-durchschnitt'],
      dtype='object')
```

Ein pandas Dataframe erkennt zuverlässig die Spaltenüberschriften des Datensatzes. Es ist erkennbar, dass diese als Liste vorliegen und um den dtype kümmern wir uns im Detail im nächsten Kapitel.

Ein Dataframe erwartet aber auch immer einen Index, über den zeilenweise auf Daten zugegriffen werden kann. Wird bei der Erstellung des Dataframes kein Index angegeben, erstellt pandas automatisch einen numerischen Index über die range-Methode. Wir werden im nächsten Kapitel noch darauf eingehen, warum der range-Index fehleranfällig ist. Nach Möglichkeit sollte also einer

der bestehenden Spalten als Index verwendet werden. Das geht aber natürlich nur, wenn dort einmalige Werte stehen.

```
#Wir nehmen das von uns betrachtete Dataframe aus der Liste heraus und setzen die Jahreszahlen als Index
df_newindex=df_list[0].set_index("Jahr")

       Bevölkerung(in Mrd.)  Wachstumsrate(% pro Jahr)  Zuwachs(Mio. pro Jahr)  Alters-durchschnitt
Jahr
1950                    253                         18                     471                235
1960                    303                         19                     606                227
1970                    369                         20                     760                215
1980                    445                         18                     829                226
1990                    532                         15                     842                241
2000                    613                         12                     773                263
2010                    692                         12                     817                285
20202                   772                          9                     733                310
20302                   842                          7                     637                332
20402                   904                          6                     540                346
20502                   955                          5                     431                361

#Wir können nun über die Jahreszahl als Index auf einzelne Zeilen zugreifen. Bei der Indizierung muss aber auf den richtigen Datentyp geachtet werden.
print (df_newindex.loc["1990"])

KeyError: '1990'

print (df_newindex.index)
Int64Index([1950, 1960, 1970, 1980, 1990, 2000, 2010, 20202, 20302, 20402,
            20502],
           dtype='int64', name='Jahr')

#OK, der Index besteht aus Integerwerten
print (df_newindex.loc[1990])

Bevölkerung(in Mrd.)         532
Wachstumsrate(% pro Jahr)     15
Zuwachs(Mio. pro Jahr)       842
Alters-durchschnitt          241
Name: 1990, dtype: int64
```

Als nächstes widmen wir uns den Spaltenüberschriften. Diese werden innerhalb eines Dataframes als Liste gespeichert und lassen sich leicht anpassen. Zuerst kürzen wir die Spaltenüberschriften auf das Wesentliche. Dafür erstellen wir eine Liste mit den neuen Spaltenüberschriften und weisen sie dem Dataframe zu.

```
lst_newcolumns=['Bevölk. in Mrd.', 'Wachstum %/Jahr','Zuwachs Mio/Jahr',
'Altersdurchschnitt']
df_newindex.columns=lst_newcolumns

print(df_newindex)
```

```
       Bevölk. in Mrd.  Wachstum %/Jahr  Zuwachs Mio/Jahr  Altersdurchschnitt
Jahr
1950               253               18               471                 235
1960               303               19               606                 227
1970               369               20               760                 215
1980               445               18               829                 226
1990               532               15               842                 241
2000               613               12               773                 263
2010               692               12               817                 285
20202              772                9               733                 310
20302              842                7               637                 332
20402              904                6               540                 346
20502              955                5               431                 361
```

Nun ist ihnen wahrscheinlich aufgefallen, dass die Werte in der Spalte "Bevölk in Mrd." falsch formatiert sind. Noch leben keine 955 Milliarden Menschen auf der Welt. Für die Lösung gedulden sie sich aber bitte noch bis zum zweiten Kapitel, das ist ein typisches Thema für die Datentransformation.

Was tun wir aber nun, wenn die Daten auf der Webseite nicht in einer Tabelle, sondern roh im HTML-Code versteckt sind? Dann bleibt uns nichts anderes übrig, als den HTML-Code zu parsen und unsere eigene Datenstruktur aufzubauen. Das folgende Beispiel zeigt erst das exemplarische Vorgehen mit der Bibliothek beautifulsoup4. Wir bauen uns erst eine einfache HTML-Struktur.

```
from bs4 import BeautifulSoup
html_txt='''
<!DOCTYPE html>
<html lang="de">
  <head>
    <title>Titel der HTML-Seite</title>
  </head>
  <body>
         <h1>1. Überschrift</h1>
         <h2>2. Überschrift</h2>
            <aside>
                    <h2>Weiterführende Links</h2>
                    <ul>
                    <li><a href="link_1.html">Wiki</a></li>
                    <li><a href="link_2.html">Blog</a></li>
                    <li><a href="link_3.html">Forum</a></li>
                    </ul>
           </aside>
  </body>
</html>
'''
```

```
#Dann erstellen wir ein Beautifulsoup-Objekt
soup = BeautifulSoup(html_txt, 'html.parser')

#und können danach auf einzelne Elemente des HTML-Zweiges zugreifen
print (soup.head)

<head>
<title>Titel der HTML-Seite</title>
</head>

#oder dasselbe ohne das HTML-Gerüst
print (soup.head.text)

Titel der HTML-Seite

#damit können wir uns beliebig tief in die HTML-Struktur vorarbeiten
print (soup.body.aside.h2)

<h2>Weiterführende Links</h2>

#einfacher ist es aber, gleichartige Elemente der HTML-Struktur in eine Liste zu
laden und der Reihe nach durchzugehen
find_li=soup.find_all("li")
for li in find_li:
    print(li.text)

Wiki
Blog
Forum
```

Wir werden das anhand eines fortgeschrittenen praktischen Beispiels durchexerzieren.

Über den BSI-Grundschutz gibt es Bausteine für verschiedene Technologien, über die sich checklistenartig das Sicherheitsniveau der betroffenen Technologie nachvollziehen lässt. Die entsprechenden HTML-Seiten sind dabei zwar eingängig zu lesen, sie eignen sich aber nicht als Checkliste. Wir bauen in der nachfolgenden Übung eine vollständige Checkliste über alle Bausteine des BSI-Kompendiums

```
#Wir laden zuerst alle Bausteine im HTML-Format von folgender Seite herunter und
entpacken die Datei:
#https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/html
_kompendium2020.zip?__blob=publicationFile&v=2

#für die weitere Arbeit benötigen wir folgende Module
from bs4 import BeautifulSoup
import glob
import pandas

#dann definieren wir die Listen, aus denen nachher die Spalten generiert werden
sollen
baustein=[]
baustein_nr=[]
siegelstufe=[]
anforderungen=[]

#dann gehen wir den heruntergeladenen Ordner durch und parsen jede vorhandene
HTML-Datei
```

```
for filename in glob.glob("C:\\bausteine_und_umsetzungshinweise\\*"):    #in dem
Ordner befinden sich die HTML-Dateien
    file=open(filename,errors='ignore',encoding='utf-8')                #ohne
das encoding werden Umlaute fehlerhaft angezeigt
    soup = BeautifulSoup(file, 'html.parser')                           #pro
File bauen wir uns ein soup-objekt
#Beautifulsoup arbeitet mit Tags, die in einer Liste gespeichert werden. Wir
gehen jedes Tag durch und hangeln uns in der Hierarchie der HTML-Datei zu dem
Punkt, den wir haben wollen. Den Inhalt fügen wir dann an die eingangs erstellten
Listen an
li=soup.find_all('li')
    for child in li:
        if child.h4:
            inhalt=child.h4.text.split()
            baustein_nr.append(child.h4.text[:-3])
            anforderungen.append(child.p.text)
            siegelstufe.append(inhalt[-1][1:-1])
            baustein.append(soup.title.string.strip())
#Nun bauen wir uns das Dataframe und weisen die Listen den einzelnen Spalten zu
GSC=pandas.DataFrame(columns=
["Baustein","Nummer","Siegelstufe","Anforderungen"])
GSC["Baustein"]=baustein
GSC["Nummer"]=baustein_nr
GSC["Siegelstufe"]=siegelstufe
GSC["Anforderungen"]=anforderungen

#Wir schauen, ob überhaupt Inhalte im Dataframe gelandet sind
print (len(GSC))

1735
#und am Schluss speichern wir die Tabelle noch im Excel-Format ab. Den Index
benötigen wir dabei nicht
GSC.to_excel("C:/GSC_Kompendium_gesamt.xlsx",index=False)
```

Datenextraktion von Textdateien

Man könnte meinen, dass das Extrahieren von Daten aus reinen Textdateien, also txt, csv oder xlsx relativ einfach sein müsste. Leider sind die entsprechenden Dateien häufig nach optischen Gesichtspunkten aufgebaut und eignen sich in ihrer Rohform nicht für eine automatische Auswertung.

Wir gehen das Problem an einem Beispiel durch, bei dem der Autor Excel mit Adobe Illustrator verwechselt hat.

```
import pandas as pd

#Das folgende Beispiel stellt eine Auswertung der Verkehrsunfälle für Schleswig-
Holstein dar.
url="https://www.statistik-
nord.de/fileadmin/Dokumente/Statistische_Berichte/verkehr_umwelt_und_energie/H_I
_1_m_S/H_I_1-m2006_SH.xlsx"

#Wir öffnen das Tabellenblatt mit dem Namen "T1_1" und überspringen gleich beim
Einlesen die ersten 8 Zeilen, da sie nur einem optischen Zweck dienen.
df=pd.read_excel(url,sheet_name="T1_1",skiprows=8)

#Das Ergebnis ist trotzdem noch nicht überzeugend
```

```
          Unnamed: 0                Unnamed: 1    Unnamed: 2    Unnamed: 3
Unnamed: 4  Unnamed: 5    Unnamed: 6  Unnamed: 7    Unnamed: 8
0               2018                      Juli   8147.000000   1298.000000
143.000000    36.00000   6670.000000   17.000000   1745.000000
1                NaN                    August   7757.000000   1158.000000
143.000000    34.00000   6422.000000   10.000000   1490.000000
2                NaN                 September   7414.000000   1119.000000
144.000000    46.00000   6105.000000   12.000000   1424.000000
3                NaN                   Oktober   7817.000000   1040.000000
167.000000    38.00000   6572.000000    8.000000   1327.000000
4                NaN                  November   8111.000000    960.000000
163.000000    30.00000   6958.000000    3.000000   1196.000000
5                NaN                  Dezember   7426.000000    916.000000
187.000000    28.00000   6295.000000   13.000000   1180.000000
6              2019                     Januar   6765.000000    789.000000
164.000000    34.00000   5778.000000    5.000000   1000.000000
7                NaN                   Februar   6126.000000    702.000000
143.000000    28.00000   5253.000000    4.000000    935.000000
8                NaN                      März   6955.000000    848.000000
176.000000    39.00000   5892.000000    5.000000   1074.000000
9                NaN                     April   7901.000000    987.000000
165.000000    43.00000   6706.000000    2.000000   1288.000000
10               NaN                       Mai   8537.000000   1085.000000
133.000000    34.00000   7285.000000    7.000000   1409.000000
11               NaN                      Juni   8233.000000   1303.000000
149.000000    46.00000   6735.000000   16.000000   1758.000000
12            Summe  Juli 2018\nbis\nJuni 2019  91189.000000  12205.000000
 1877.000000   436.00000  76671.000000  102.000000  15826.000000
```

```
#Zuerst erstellen wir händisch neue Spaltenüberschriften und weisen sie dem
Dataframe zu.
lst_columns=
["Jahr","Monat","insgesamt","Personenschaden","Sachschäden","Alkoholeinfluss","Ü
brige","Getötete","Verletzte"]
df.columns=lst_columns

#Danach füllen wir die leeren Werte in der zweiten Spalte nach unten auf und
löschen Zeilen mit Nullwerten
df["Jahr"]=df["Jahr"].fillna(method="ffill")
df=df.dropna()

#Als letztes löschen wir noch Zeilen, die in der ursprünglichen Tabelle als
Zwischenüberschriften eingefügt wurden
df.drop(df[df.Jahr=="Summe"].index, inplace=True)

#und konvertieren die numerischen Spalten in Integerwerte
for column in (df.columns):
    if df[column].dtype !="object":
        df[column]=df[column].astype(int)
```

```
    Jahr      Monat  insgesamt  Personenschaden  Sachschäden  Alkoholeinfluss
 Übrige  Getötete  Verletzte
0   2018       Juli       8147             1298          143               36
 6670        17       1745
1   2018     August       7757             1158          143               34
 6422        10       1490
2   2018  September       7414             1119          144               46
 6105        12       1424
```

```
3   2018    Oktober       7817            1040          167           38
 6572         8       1327
4   2018   November       8111             960          163           30
 6958         3       1196
5   2018   Dezember       7426             916          187           28
 6295        13       1180
6   2019    Januar        6765             789          164           34
 5778         5       1000
7   2019    Februar       6126             702          143           28
 5253         4        935
8   2019     März         6955             848          176           39
 5892         5       1074
9   2019     April        7901             987          165           43
 6706         2       1288
10  2019       Mai        8537            1085          133           34
 7285         7       1409
11  2019      Juni        8233            1303          149           46
 6735        16       1758
14  2019      Juli        7740            1090          129           27
 6494        10       1416
15  2019    August        8255            1281          132           40
 6802        10       1642
16  2019  September       7735            1154          182           32
 6367         9       1445
17  2019    Oktober       8065            1004          180           45
 6836        13       1260
18  2019   November       8247            1003          203           29
 7012         6       1259
19  2019   Dezember       7650            1025          158           41
 6426        13       1359
20  2020    Januar        6785             860          182           23
 5720        12       1099
21  2020    Februar       6607             828          214           33
 5532         8       1067
22  2020     März         5049             564          119           29
 4337         7        736
23  2020     April        5175             662          102           22
 4389        11        806
24  2020       Mai        6538             894          119           24
 5501        18       1109
25  2020      Juni        7003            1150          136           23
 5694         9       1420
```

Mit der Tabelle gewinnen wir wahrscheinlich keinen Designpreis mehr, aber zumindest können wir jetzt mit den Daten arbeiten.

Daten vorbereiten mit Python

Unter Begriffen wie Datapreparation, Datashaping oder Datatransformation wird die Aufbereitung unstrukturierter Daten beim ETL-Prozess verstanden, um sie für eine nachfolgende Analyse vorzubereiten. Das Vorbereiten von Daten entspricht dabei dem "T" im ETL-Prozess.
In diesem Kapitel beschäftigen wir uns mit dem Vorbereiten von Daten, das hauptsächlich aus zwei Einzelschritten besteht:

- Daten bereinigen bzw. Datacleansing
- Daten transformieren bzw. Datashaping

Daten bereinigen

Unterschiedliche Datentypen in einer Spalte, unterschiedliche Schreibweisen einzelner Werte oder einfache Schreibfehler wie führende oder folgende Leerzeichen verhindern eine strukturierte Auswertung von Daten. Beim Bereinigen von Daten sollten folgende Grundsätze beachtet werden:

- Statistische Auswertungen oder weitergehende ML-basierte Modelle können besser mit numerischen Werten als mit Text arbeiten. Wenn nicht gerade eine Textklassifikation angestrebt wird, sollten Daten möglichst in rein numerische Werte (int, float, datetime o.ä.) übersetzt werden.
- Python ist Case-sensitive und würde damit die Wörter "Haus" und "haus" als zwei unterschiedliche Wörter ansehen. Das ist insbesondere hinderlich beim Zählen und Kategorisieren von Daten. Es ist für die Auswertung hilfreich, Textinformationen komplett in Groß -oder Kleinschreibung zu übersetzen.
- Viele Analysemethoden reagieren empfindlich auf fehlende Daten. Liefern einzelne Daten einen Null-Wert, sollte überlegt werden, diesen Datensatz komplett zu löschen oder zu maskieren.

Für die nächsten Übungen bauen wir folgendes einfaches Dataframe auf:

```
import pandas as pd

dict={"Name":pd.Series([" Hans-Peter","Karl-Heinz","Petra"," Julia  "]),
    "Beruf":pd.Series(["Beamter","Kaufmann","Lehrerin","Musikerin"]),
    "Alter":pd.Series([35,42,50,61])}

df=pd.DataFrame(dict)
```

```
          Name      Beruf  Alter
0   Hans-Peter    Beamter     35
1   Karl-Heinz   Kaufmann     42
2        Petra   Lehrerin     50
3       Julia   Musikerin     61
```

Trimmen und Kürzen von Texten

In dem Dataframe sind einige überschüssige Leer -und Sonderzeichen versteckt, was eine Auswertung erschwert. Zuerst finden wir heraus, wo unser Dataframe in der Spalte "Name" davon betroffen ist.

```
print (df.loc[df.Name.str.match(r".*\s$") | df.Name.str.match(r"^\s.*")])
```

```
          Name      Beruf  Alter
0   Hans-Peter    Beamter     35
3      Julia    Musikerin     61
```

Über die Zeichenklasse \s wird auch der Tabulator gefunden, der sich hinter Julia versteckt. Mittels der Stringfunktion strip() können die überschüssigen Leerzeichen nun entfernt werden.

```
df["Name"]=df["Name"].str.strip()
```

Das erspart bei Tabellen mit einer sechsstelligen Anzahl an Zeilen einige Sucharbeit. Warum aber nun diese Funktion nicht generell über alle Spalten laufen lassen? Versuchen wir es mal:

```
for column in df.columns:
    df[column]=df[column].str.strip()
```

```
AttributeError: Can only use .str accessor with string values!
```

Ok, irgendeine Spalte verwendet als Datentyp keine Strings. Der Verdacht würde ja auf die Spalte "Alter" fallen. Schauen wir mal:

```
print (df.info())
```

```
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    4 non-null      object
 1   Beruf   4 non-null      object
 2   Alter   4 non-null      int64
dtypes: int64(1), object(2)
```

Der Verdacht hat sich bestätigt, Stringfunktionen können natürlich nicht auf Integerwerte angewendet werden. Es bleibt also , entweder nur verdächtige Spalten zu durchsuchen und zu trimmen oder bei sehr vielen Spalten unsere Schleife auszubauen:

```
for column in df.columns:
    if df[column].dtype==object:
        df[column]=df[column].str.strip()
```

Indizieren von Datenstrukturen

In Pandas-Datenstrukturen gibt es zwei primäre Methoden zur Indizierung. .loc funktioniert labelbasiert und .iloc indexbasiert. Was heißt das jetzt konkret?

```
#Wir lassen uns die erste Zeile ausgeben. Die 0 stellt dabei das Label der
Indexspalte dar, nicht den Index der Einträge!
print (df.loc[0])

Name      Hans-Peter
Beruf        Beamter
Alter              35

#Wenn wir den Index nun auf die Namensspalte setzen, funktioniert die Label-
indizierung mit 0 nicht mehr, da der Index der erste Zeile nun "Hans-Peter"
heisst.
df=df.set_index("Name")
print (df.loc[0])
TypeError: cannot do label indexing on <class 'pandas.core.indexes.base.Index'>
with these indexers [0] of <class 'int'>

# Wir müssen also entweder mit .loc das Label konkret benennen ("Hans-Peter"),
oder mit .iloc die integerbasierte Indizierung nutzen (da wären wir wieder bei
0)
print (df.loc["Hans-Peter"])
```

```
print (df.iloc[0])
```

Das Verständnis für die unterschiedliche Arbeitsweise von .loc und .iloc ist wichtig für ein fehlerfreies Arbeiten. Generell kann man aber sagen, dass die Nutzung von .loc nur über die Indexspalte nicht empfehlenswert ist.

Bis jetzt haben wir nur einzelne Zeilen identifiziert. Wir wollen aber ja auf Zellen innerhalb einer Zeile zugreifen. Auch dafür bieten .loc und .iloc ausreichend Möglichkeiten.

```
#Das entspricht der ersten Zeile und der ersten Spalte nach dem Index
print (df.loc["Hans-Peter","Beruf"])

Beamter

#Dasselbe Ergebnis erzielen wir mit folgendem .iloc-Eintrag
print (df.iloc[0,0])
```

Um nun über alle Zeilen zu iterieren und mit den Daten der einzelnen Spalten zu arbeiten, gibt es mehrere Möglichkeiten.

```
#nicht empfehlenswert. Daten sollten nicht inline bearbeitet werden, da python in
einigen Fällen mit Kopien arbeitet
for rows in df.iterrows():
    print (rows)

#stattdessen...
#Gibt das gesamte Dataframe aus
print (df.loc[:,])
#gibt nur die Namensspalte aus
print (df.loc[:,"Name"])
#und nur den Beruf der ersten beiden Einträge
print (df.loc[0:1,"Beruf"])

#Und wer nicht auf die For-Schleife verzichten möchte, kann folgendes probieren
for zeile in range (0,len(df)):
    print(df.iloc[zeile,0],end=",")
    print (df.iloc[zeile,1],end=",")
    print (df.iloc[zeile,2])

Hans-Peter,Beamter,35
Karl-Heinz,Kaufmann,42
Petra,Lehrerin,50
Julia,Musikerin,61
```

Fortgeschrittene Suchfunktionen lassen sich am besten mit boolean-indexing umsetzen.

```
#Wer in dem Dataframe ist Beamter?
print(df[df["Beruf"]=="Beamter"])

         Name    Beruf  Alter
0  Hans-Peter  Beamter     35

#und jetzt wollen wir nur den Namen des Beamten
print(df[df["Beruf"]=="Beamter"].Name)
```

```
0    Hans-Peter
Name: Name, dtype: object

#Wer ist älter als 40 und jünger als 60?
print(df[(df.Alter > 40) & (df.Alter <60)])

         Name     Beruf  Alter
1  Karl-Heinz  Kaufmann     42
2       Petra  Lehrerin     50

#Doppelnamen sollen ohne Bindestriche geschrieben werden
df.loc[df.Name.str.contains("-"),"Name"]=df.Name.str.replace("-"," ")

         Name      Beruf  Alter
0  Hans Peter    Beamter     35
1  Karl Heinz   Kaufmann     42

#Und die Berufsbezeichnung soll noch etwas genauer sein
df.loc[df["Beruf"]=="Beamter","Beruf"]="Verwaltungsbeamter"

         Name               Beruf  Alter
0  Hans-Peter  Verwaltungsbeamter     35
```

Es ist Geschmackssache, ob man dabei über Listcomprehensions schneller ans Ziel kommt:

```
df["Geburtsjahr"]=[datetime.date.today().year-alter for alter in df["Alter"]]

         Name      Beruf  Alter  Geburtsjahr
0  Hans-Peter    Beamter     35         1985
1  Karl-Heinz   Kaufmann     42         1978
2       Petra   Lehrerin     50         1970
3       Julia  Musikerin     61         1959


#Statt des Doppelnamens soll nur der erste Namensbestandteil angezeigt werden
df["Name"]=[name.split("-")[0] for name in df["Name"]]

    Name      Beruf  Alter
0   Hans    Beamter     35
1   Karl   Kaufmann     42
2  Petra   Lehrerin     50
3  Julia  Musikerin     61

#Nun sollen die Namen groß geschrieben werden, um die spätere Auswertung zu
erleichtern
df["Name"]=[name.upper() for name in df["Name"]]

         Name      Beruf  Alter
0  HANS-PETER    Beamter     35
1  KARL-HEINZ   Kaufmann     42
2       PETRA   Lehrerin     50
3       JULIA  Musikerin     61

#Und es wird noch die Branche als zusätzliche Spalte hinzugefügt
df["Branche"]=["Öffentlicher Dienst" if (beruf=="Beamter" or beruf=="Lehrerin")
else "Freie Wirtschaft" for beruf in df["Beruf"]]
```

```
         Name      Beruf  Alter              Branche
0  Hans-Peter    Beamter     35  Öffentlicher Dienst
1  Karl-Heinz   Kaufmann     42     Freie Wirtschaft
2       Petra   Lehrerin     50  Öffentlicher Dienst
3       Julia  Musikerin     61     Freie Wirtschaft
```

```
#Natürlich lässt sich die bestehende Spalte aus inline ändern
df["Beruf"]=["Öffentlicher Dienst" if (beruf=="Beamter" or beruf=="Lehrerin")
else "Freie Wirtschaft" for beruf in df["Beruf"]]
```

```
         Name                Beruf  Alter
0  Hans-Peter  Öffentlicher Dienst     35
1  Karl-Heinz     Freie Wirtschaft     42
2       Petra  Öffentlicher Dienst     50
3       Julia     Freie Wirtschaft     61
```

Umgang mit Duplikaten

Duplikate in Datensätzen lassen sich nicht immer vermeiden und das ist häufig auch gar nicht möglich. In einem großen Datensatz mit Namen kommt es zwangsläufig vor, dass mehrere Personen denselben Namen nutzen. Problematisch wird die Situation, wenn die betroffenen Daten eine besondere Funktion in dem Datensatz einnehmen, z.B. als Index oder als Schlüsselspalte für join/merge-Operationen. Zudem benötigen mehr Daten auch mehr Speicher und mehr CPU-Zyklen. Es ist also immer eine Frage des Einzelfalles, ob das Löschen von Duplikaten unerwünschte Nebeneffekte hätte. In den seltensten Fällen macht aber ein generelles Löschen von Duplikaten über das gesamte Dataframe Sinn.

Wir nutzen wieder das Dataframe aus dem vorherigem Kapitel, bauen es aber ein wenig um, um die Sinnhaftigkeit beim Umgang mit Duplikaten darzustellen.

```
dict={"Name":pd.Series(["Hans-Peter","Karl-Heinz","Petra","Julia","Julia"]),
    "Beruf":pd.Series(["Beamter","Beamter","Lehrerin","Musikerin","Musikerin"]),
    "Alter":pd.Series([35,42,42,61,61])}

df=pd.DataFrame(dict)
```

```
         Name      Beruf  Alter
0  Hans-Peter    Beamter     35
1  Karl-Heinz    Beamter     42
2       Petra   Lehrerin     42
3       Julia  Musikerin     61
4       Julia  Musikerin     61
```

Hans-Peter und Karl-Heinz haben nun zufällig denselben Beruf und Karl-Heinz und Petra sind zufällig gleich alt. Dafür wollen wir sie natürlich nicht bestrafen und wir müssen schauen, wie wir mit dieser Art von Duplikaten im weiteren vorgehen. Auffällig sind die letzten beiden Einträgen mit dem Index 3 und 4. Diese stimmen in allen drei Spalten überein, sind also ein Duplikat im klassischen Sinne.

Wie finden wir nun heraus, ob in unserem Datensatz Duplikate vorkommen?

```
print(df.duplicated())

0    False
1    False
2    False
3    False
4     True
```

Das entspricht schon ungefähr dem von uns erwartetem Ergebnis. Der letzte Eintrag mit dem Index 4 ist identisch mit dem Index 3. Allerdings kommt es nicht zwangsläufig vor, dass doppelte Einträge untereinander stehen. Wie kriegen wir nun heraus, zu welchem zweiten (und eventuell dritten, vierten...) Duplikat der Eintrag gehört?

```
print(df.duplicated(keep=False))

0    False
1    False
2    False
3     True
4     True
```

Mit dem Parameter keep wird festgelegt, welcher der gefundenen Duplikate auf True gesetzt wird. Standardmäßig werden mit keep=first alle außer dem ersten Duplikat auf True gesetzt. Mit keep=last wird nur das letzte Duplikat nicht angezeigt (also mit False markiert). Mit keep=False werden alle Duplikate auf True gesetzt. Um zu schauen, welcher Eintrag nun wirklich davon betroffen ist, können wir uns wieder mit boolean indexing behelfen.

```
print (df.loc[df.duplicated()])
    Name      Beruf  Alter
4  Julia  Musikerin     61

#Mit dem Parameter keep=False sehen wir auch, wie oft das Duplikat vorkommt
print (df.loc[df.duplicated(keep=False)])

3  Julia  Musikerin     61
4  Julia  Musikerin     61
```

Wenn wir aber auch Duplikate in anderen Spalten finden wollen, können wir die Suche auf einzelne Spalten reduzieren.

```
print(df.duplicated(subset="Beruf",keep=False))

0     True
1     True
2    False
3     True
4     True
```

Das passt! Wir haben zwei Beamte und zwei Musikerinnen.

Wir wollen nun die Duplikate löschen. Im einfachsten Fall löschen wir nur den doppelten Eintrag mit dem Index 3 und 4.

```
print(df.drop_duplicates())

         Name      Beruf  Alter
0  Hans-Peter    Beamter     35
1  Karl-Heinz    Beamter     42
2       Petra   Lehrerin     42
3       Julia  Musikerin     61
```

Damit hätten wir paretomäßig schon mal 80% der Fälle abgedeckt. Schauen wir aber auch nochmal auf die verbleibenden 20%.

```
#Wir müssen die Anzahl an Beamten verringern!
print(df.drop_duplicates(subset="Beruf"))

         Name      Beruf  Alter
0  Hans-Peter    Beamter     35
2       Petra   Lehrerin     42
3       Julia  Musikerin     61

#oder die Diversität in der Alterstrukur erhöhen
print(df.drop_duplicates(subset="Alter"))

0  Hans-Peter    Beamter     35
1  Karl-Heinz    Beamter     42
3       Julia  Musikerin     61

#Wenn Duplikate unglaubwürdig wirken, kann man auch alle betroffenen Einträge
löschen
print(df.drop_duplicates(keep=False))

         Name     Beruf  Alter
0  Hans-Peter   Beamter     35
1  Karl-Heinz   Beamter     42
2       Petra  Lehrerin     42
```

Umgang mit NAN-Werten

Ein Ansatz zum Umgang mit fehlenden Werten ist die Verwendung des Python Objektes None, der in allen Datenstrukturen verwendet werden kann. Nachteil dieses Vorgehens ist, dass damit die betroffene Datenstruktur automatisch den dtype=object erhält und numerische Operationen wie min, max, sum auf das Array mit einem Fehler beendet werden.

```
import numpy as np
fibonacci = np.array([1,1, None,3,5,8,13,21,34])
fibonacci

array([1, 1, None, 3, 5, 8, 13, 21, 34], dtype=object)
```

Besteht eine Datenstruktur aus rein numerischen Daten, sollte daher lieber NaN (not a Number) verwendet werden, der einen speziellen Float-Wert darstellt.

```
import numpy as np
fibonacci = np.array([1,np.nan,2,3,5,8,13,21,34])

array([ 1., nan,  2.,  3.,  5.,  8., 13., 21., 34.])
```

Der dtype wird damit automatisch von int zu float64. Zu beachten ist dabei allerdings, dass damit zwar numerische Operationen ohne Fehler durchgeführt werden, aber nicht immer das erwartete Ergebnis produzieren.

```
fibonacci.sum()

nan
```

Bis jetzt haben wir das Problem von allen Seiten umkreist und bewundert. Was machen wir aber nun mit Null-Werten in Datensätzen? Zum Glück bietet uns Pandas zumindest bei den Series und Dataframes umfangreiche Möglichkeiten, mit Nullwerten zu arbeiten.

Machen wir also aus unserem Array zunächst eine Series.

```
fibonacci = np.array([1,1, None,3,5,8,13,21,34])
fibonacci_series=pd.Series(fibonacci)

0       1
1       1
2    None
3       3
4       5
5       8
6      13
7      21
8      34
```

Die Ausgabe ist ähnlich wie bei dem Array, wir haben zusätzlich allerdings noch einen Index bekommen. Über die beiden Funktionen isnull() können wir nun einfach herausbekommen, welche Datensätze fehlerhaft sind. Mit der Funktion notnull() würde das Ergebnis umgedreht werden.

```
print (fibonacci_series.isnull())

0    False
1    False
2     True
3    False
4    False
5    False
6    False
7    False
8    False
```

Über boolean-Indexing können wir nun auch auf die Zeilen filtern, die NaN-Werte enthalten.

```
print(fibonacci_series[fibonacci_series.isnull()]==True)

2    False
dtype: bool
```

Wir können jetzt händisch die fehlerhaften Werte mit einer von uns gewählten Maskierung füllen.

```
fibonacci_series.loc[fibonacci_series.isnull()]=-1

0     1
1     1
2    -1
3     3
4     5
5     8
6    13
7    21
8    34
```

Einfacher und weniger fehleranfällig sind allerdings die Standardfunktionen der Pandasbibliothek

```
#Löscht die komplette Zelle mit Nullwerten
print (fibonacci_series.dropna())

0     1
1     1
3     3
4     5
5     8
6    13
7    21
8    34

#Ergibt dasselbe Ergebnis wie bei der manuellen Bearbeitung
print (fibonacci_series.fillna(-1))

0     1
1     1
2    -1
3     3
4     5
5     8
6    13
7    21
8    34
```

Es hängt vom Einzelfall ab, ob man besser die gesamte Zeile löschen oder mit einem Füllwert maskieren soll. Folgende Aspekte müssen dabei beachtet werden:

- Wenn mit Maskierung gearbeitet wird, schränkt das den Raum möglicher Datenwerte ein. Zudem könnten Verzerrungen oder Fehler bei weitergehenden Analysen durch die Maskierung auftreten
- Wenn die Zeile komplett gelöscht wird, sind auch die Werte in weiteren Spalten verloren. Das ist insbesondere problematisch, wenn sich weitergehende Analysen insbesondere auf nicht betroffene Spalten beziehen.

Wenn also, wie in unserem Beispiel nur eine Spalte betroffen ist, ist zu empfehlen, die Zeile zu löschen. Bei Datensätzen mit vielen Spalten und NaN-Werten in für die weitere Verarbeitung unwichtigen Spalten sollten die Werte maskiert werden. Die NaN-Werte zu behalten, ist nicht zu empfehlen.

Daten transformieren

Im vorherigen Kapitel haben wir die Daten inhaltlich so weit bereinigt, dass sie zumindest theoretisch auswertbar sind. In diesem Kapitel geht es nun darum, die Daten in eine Struktur zu bringen, damit eine automatische Auswertung überhaupt funktioniert. Datenstrukturen sind häufig nach optischen Gesichtspunkten aufgebaut, damit sie durch das menschliche Auge konsumiert werden können. Menschen gelingt dabei zwar automatisch der Kontextbezug, das aber nur bei einer geringen Datenmenge. Bei großen Datenbeständen spielen Automatismen ihre Vorteile aus. Allerdings müssen die Daten dafür in einem Format vorliegen, dass keine Interpretation erfordert. Der Fokus beim Transformieren liegt auf folgenden Tätigkeiten:

- Daten in eine kontextfreie Struktur zu bringen
- Daten aus unterschiedlichen Blickwinkeln betrachten, um neue Informationen aus den Daten zu erhalten
- Daten aus mehreren Datenquellen zusammenzubringen, um die Informationsdichte zu erhöhen
- Daten zur Dimensionsreduktion aggregieren

Zuerst bauen wir uns über ein Dictionary ein einfaches Dataframe auf, mit dem wir im Laufe des Kapitels noch arbeiten werden.

Spalten tauschen, kombinieren und trennen

Das Tauschen von Spalten ist relativ einfach, da die Spaltenreihenfolge als Tuple in der Funktion DataFrame.Columns gespeichert ist. Es reicht, die Reihenfolge der Spaltenüberschriften zu ändern.

```
print(df[["Alter","Beruf","Name"]])
   Alter      Beruf        Name
0     35    Beamter  Hans-Peter
1     42   Kaufmann  Karl-Heinz
```

So ähnlich funktioniert auch das Kombinieren von Spalten. Zuerst fügen wir die Namen und den Beruf zusammen.

```
df["Name"]=df["Name"]+" "+df["Beruf"]

                 Name      Beruf  Alter
0   Hans-Peter Beamter    Beamter     35
1  Karl-Heinz Kaufmann   Kaufmann     42
2       Petra Lehrerin   Lehrerin     50
3      Julia Musikerin  Musikerin     61
```

Da die Spalte "Beruf" noch vorhanden ist, liegen die Daten nun doppelt vor. Wir müssen also noch die Spalte löschen.

```
#Wir müssen axis=1 setzen, da python standardmäßig nach dem Index (also der
Zeile) sucht. Wir wollen aber ja die Spalte löschen
print(df.drop("Beruf",axis=1))

                  Name  Alter
0   Hans-Peter Beamter     35
1  Karl-Heinz Kaufmann     42
2       Petra Lehrerin     50
3      Julia Musikerin     61
```

Beim Trennen von Spalten funktioniert ebenfalls die standardmäßige Stringoperation .split(). Der Parameter expand=True ist notwendig, da python ansonsten die Liste mit den gesplitteten Werten in die Zelle schreibt. Zudem müssen wir python noch mitteilen, auf welche neuen Spalten die neuen Werte aufgeteilt werden sollen. Ansonsten würde nur der erste Wert in die alte Spalte übernommen werden.

```
df[["1. Name","2. Name"]]=df["Name"].str.split("-",expand=True)
print (df.drop("Name",axis=1))
       Beruf  Alter 1. Name 2. Name
0    Beamter     35    Hans   Peter
1   Kaufmann     42    Karl   Heinz
2   Lehrerin     50   Petra    None
3  Musikerin     61   Julia    None
```

Und da wir weiter oben ja schon gelernt haben, dass None-Werte in Datensätzen vermieden werden sollten, maskieren wir die leeren Zellen mit einem beliebigen Füllwert.

```
print (df.fillna("-"))
         Name      Beruf  Alter 1. Name 2. Name
0  Hans-Peter    Beamter     35    Hans   Peter
1  Karl-Heinz   Kaufmann     42    Karl   Heinz
2       Petra   Lehrerin     50   Petra       -
3       Julia  Musikerin     61   Julia       -
```

Datenstrukturen transponieren

Wir nutzen für die nächsten Beispiele ein einfaches Dataframe, um die Effekte der unterschiedlichen Methoden besser darzustellen.

```
import pandas as pd
dict={"A":pd.Series(["x","y","x","y"]),
    "B":pd.Series([1,2,1,2]),
    "C":pd.Series([3,4,5,6]),
    "D":pd.Series([10,20,30,40])}

df=pd.DataFrame(dict)

   A  B  C   D
0  x  1  3  10
1  y  2  4  20
2  x  1  5  30
3  y  2  6  40
```

Im einfachsten Fall transponieren wir die Tabelle, indem wir sie um 90 Grad kippen. Aus den Spaltenüberschriften wird dadurch der Index und der vorherige Index zu den Spaltenüberschriften.


```
df_transpose=df.transpose()
print(df_transpose)
```

```
    0   1   2   3
A   x   y   x   y
B   1   2   1   2
C   3   4   5   6
D  10  20  30  40
```

Wir laden beispielhaft die Klimatabelle für die Stadt Kiel:

```
import pandas as pd
import requests

url="https://www.wetterkontor.de/de/klima/klima2.asp?land=de&stat=10046"
r=requests.get(url)
df_list=pd.read_html(r.text)
df=df_list[0]
print(df.head(3))
```

```
  Unnamed: 0  Temperatur °Cmax. Ømin. Ø  Temperatur °Cmax. Ømin. Ø.1
NiederschlagmmTage  NiederschlagmmTage.1  relativeFeuchte  Sonneh/Tag Wasser°C
0        Jan                           2                              -2
     65                    18                87          12        -
1        Feb                           3                              -2
     40                    15                84          21        -
2        Mär                           6                               0
     54                    13                81          34        -
```

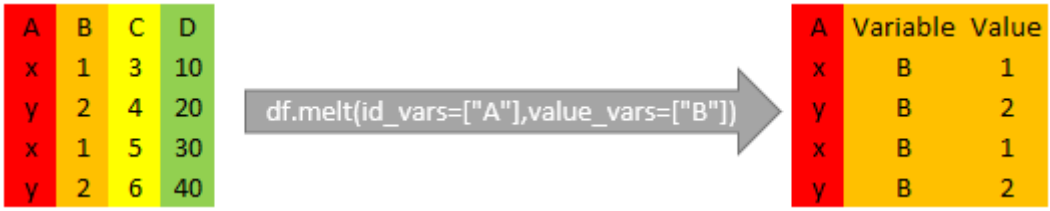
Die Tabelle ist optisch nicht besonders ansprechend. Wir erreichen eine kompaktere Form, indem wir die Tabelle transponieren:

```
df.set_index('Unnamed: 0',inplace=True) # Damit die Monatsnamen auch als
Spaltenüberschriften erscheinen
print(df.transpose())
```

```
Unnamed: 0                  Jan Feb Mär Apr Mai Jun Jul Aug Sep Okt Nov Dez
Jahr
Temperatur °Cmax. Ømin. Ø     2   3   6  11  16  20  21  21  18  13   8   4
119
Temperatur °Cmax. Ømin. Ø.1  -2  -2   0   3   7  11  12  12  10   7   3   0
51
NiederschlagmmTage           65  40  54  52  57  69  79  69  66  67  86  74
1635
NiederschlagmmTage.1         18  15  13  14  12  14  15  16  15  17  18  18
185
relativeFeuchte              87  84  81  77  74  74  76  78  81  85  68  87
81
Sonneh/Tag                   12  21  34  55  74  76  71  71  49  33  17  11
44
Wasser°C                      -   -   -   -   -   -   -   -   -   -   -   -
-
```

Datenstrukturen entpivotieren

Eine nützliche Funktion für die Datenauswertung ist das Entpivotieren von Daten. Dabei werden die Spalten einer Tabelle zusammengefasst, heißt die Anzahl an Spalten sinkt, dafür vergrößert sich die Anzahl an Zeilen. Zum Entpivotieren benötigt man eine oder mehrere ID-Spalten und eine oder mehrere Werte-Spalten. Die Werte-Spalten werden in der entpivotierten Tabelle dann in einer Variable-Spalte und einer Werte-Spalte ausgegeben. Die Variable-Spalte entspricht dabei der ehemaligen Spaltenüberschrift und der Inhalt der Werte-Spalte dem zugehörigen Zellenwert. Wenn Spalten nicht in der ID -oder der Wertespalte aufgenommen werden, erscheinen sie nachher auch nicht in der entpivotierten Spalte.



Fangen wir mit einem einfachen Beispiel an, indem wir unser eingangs angelegtes Dataframe entpivotieren:

```
import pandas as pd

dict={"A":pd.Series(["x","y","x","y"]),
    "B":pd.Series([1,2,1,2]),
    "C":pd.Series([3,4,5,6]),
    "D":pd.Series([10,20,30,40])}

df=pd.DataFrame(dict)

print (df.melt(id_vars=["A"],value_vars=["B","C","D"]))
```

```
    A variable  value
0   x        B      1
1   y        B      2
2   x        B      1
3   y        B      2
4   x        C      3
5   y        C      4
6   x        C      5
7   y        C      6
8   x        D     10
9   y        D     20
10  x        D     30
11  y        D     40
```

Wenn die ID-Spalte weggelassen wird, nimmt Python automatisch die Index-Spalte. Da die Spalte "A" nun weder als ID -noch als Wertespalte auftaucht, erscheint sie auch nicht mehr in der entpivotierten Tabelle.

```
print (df.melt(value_vars=["B","C","D"]))
```

```
   variable  value
0         B      1
1         B      2
2         B      1
3         B      2
4         C      3
5         C      4
6         C      5
7         C      6
8         D     10
9         D     20
10        D     30
11        D     40
```

Es macht Sinn, die standardmäßige Beschriftung der Variablen -und Wertespalte andere Namen zu geben:

```
print (df.melt(value_vars=["B","C","D"], var_name='Buchstabe',
value_name='Werte'))
```

```
   Buchstabe  Werte
0          B      1
1          B      2
2          B      1
```

Abschließend noch ein praktischer Anwendungsfall für das Entpivotieren von Daten aus dem Bereich der Informationssicherheit. Beim BSI-Grundschutz werden für gängige Technologien relevante Gefährdungsszenarien in Kreuzreferenztabellen zusammengefasst, um einen Anhalt für weitergehende Risikoanalysen zu bekommen. Die Kreuzrefrenztabellen sind allerdings in ihrer Ursprungsform nur einer manuellen Auswertung zugänglich. Die Tabelle für den Baustein APP.1.1 (Office-Produkte) sieht im Original folgendermaßen aus:

```
            G 0.18 G 0.19 G 0.20 G 0.21 G 0.22 G 0.28 G 0.29 G 0.37 G 0.39 G 0.45
G 0.46
APP.1.1
APP.1.1.A01    NaN    NaN      X      X    NaN    NaN    NaN    NaN      X    NaN
   NaN
APP.1.1.A02    NaN      X    NaN    NaN      X    NaN    NaN    NaN      X    NaN
   NaN
APP.1.1.A03    NaN    NaN    NaN    NaN    NaN    NaN      X    NaN      X    NaN
   NaN
APP.1.1.A04    NaN      X    NaN    NaN    NaN      X    NaN    NaN      X    NaN
     X
APP.1.1.A05      X    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
   NaN
APP.1.1.A06      X    NaN      X      X    NaN      X    NaN    NaN    NaN      X
     X
APP.1.1.A07      X    NaN      X    NaN    NaN    NaN    NaN    NaN    NaN    NaN
   NaN
APP.1.1.A08      X    NaN    NaN    NaN    NaN    NaN      X    NaN    NaN    NaN
   NaN
APP.1.1.A09    NaN      X    NaN    NaN    NaN    NaN    NaN    NaN    NaN    NaN
   NaN
APP.1.1.A10      X    NaN    NaN    NaN    NaN      X    NaN    NaN    NaN    NaN
   NaN
APP.1.1.A11    NaN    NaN      X      X    NaN      X      X    NaN      X      X
   NaN
APP.1.1.A12      X      X    NaN    NaN    NaN    NaN      X    NaN    NaN    NaN
     X
APP.1.1.A13    NaN    NaN      X    NaN    NaN      X    NaN    NaN      X    NaN
   NaN
APP.1.1.A14    NaN      X    NaN    NaN      X    NaN    NaN      X    NaN    NaN
     X
APP.1.1.A15    NaN      X      X    NaN      X    NaN    NaN      X    NaN    NaN
     X
APP.1.1.A16    NaN    NaN      X    NaN      X    NaN    NaN    NaN      X    NaN
     X
```

Dabei sind die Spaltenüberschriften die relevanten Gefährdungen für den Baustein und hinter dem Index verstecken sich die Anforderungen, mit denen der Gefährdung begegnet werden soll. Um leichter herauszufiltern, welche Gefährdung durch welche Anforderung behandelt wird, würden wir die Tabelle derart entpivotieren, dass die Anforderungen die ID-Spalte bleiben und alle weiteren Spalten werden als Werte-Spalten behandelt:

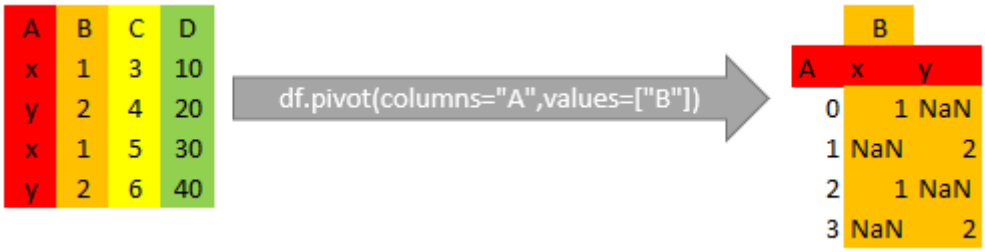
```
file="App1.1.xlsx"
df=pd.read_excel(file)
df_melt=df.melt(id_vars=df.columns[0],value_vars=df.columns[1:],
        var_name='Gefährdung', value_name='Relevanz').set_index(df.columns[0])
print (df_melt.head(10))
```

Damit wird die Tabelle deutlich kompakter und lässt sich in Excel leichter nach der Relevanz filtern.

```
           Gefährdung Relevanz
APP.1.1
APP.1.1.A01      G 0.18      NaN
APP.1.1.A02      G 0.18      NaN
APP.1.1.A03      G 0.18      NaN
APP.1.1.A04      G 0.18      NaN
APP.1.1.A05      G 0.18        X
APP.1.1.A06      G 0.18        X
APP.1.1.A07      G 0.18        X
APP.1.1.A08      G 0.18        X
APP.1.1.A09      G 0.18      NaN
APP.1.1.A10      G 0.18        X
...
```

Datenstrukturen pivotieren

Wie zu erwarten ist, handelt es sich beim Pivotieren von Daten um das Gegenteil des Entpivotierens. Eine Spalte wird um 90 Grad geschwenkt und zu Spaltenüberschriften. Die Werte für die jeweiligen Spaltenüberschriften bilden die Werte-Spalten.



Wir nehmen zum Anfang wieder unser eingangs erstelltes Dataframe und geben es zum besseren Verständnis noch einmal in Originalform aus:

```
print (df)
   A  B  C   D
0  x  1  3  10
1  y  2  4  20
2  x  1  5  30
3  y  2  6  40
```

Wir machen jetzt aus der Spalte "A" Spaltenüberschriften und nutzen die Spalte "B" für die Werte:

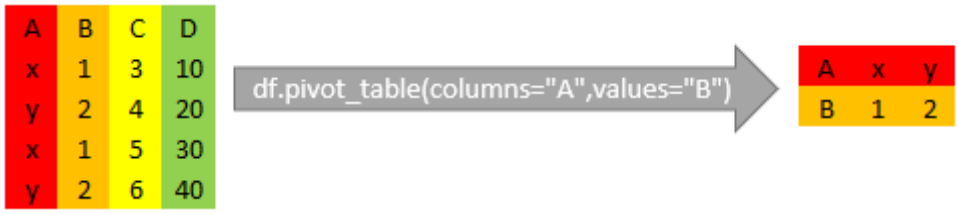
```
print (df.pivot(columns="A",values=["B"]))
     B
A    x    y
0  1.0  NaN
1  NaN  2.0
2  1.0  NaN
3  NaN  2.0
```

Aus der Spalte "A" ist nun die erste Zeile geworden, wobei python die doppelten Werte aggregiert hat. Die Werte aus der Spalte "B" hat python nun dem Index entsprechend auf die x -und y-Spalte aufgeteilt. Etwas unübersichtlich wird es allerdings, wenn wir die übrigen Spalten auch noch mit in die Pivot-Tabelle aufnehmen wollen.

```
print (df.pivot(columns="A",values=["B","C","D"]))
     B          C          D
A    x    y     x    y     x     y
0  1.0  NaN   3.0  NaN  10.0   NaN
1  NaN  2.0   NaN  4.0   NaN  20.0
2  1.0  NaN   5.0  NaN  30.0   NaN
3  NaN  2.0   NaN  6.0   NaN  40.0
```

Unsere Indexspalte "A" wird nun auf die drei Wertespalten aufgeteilt, was einer strukturierten Auswertung nicht unbedingt förderlich ist.

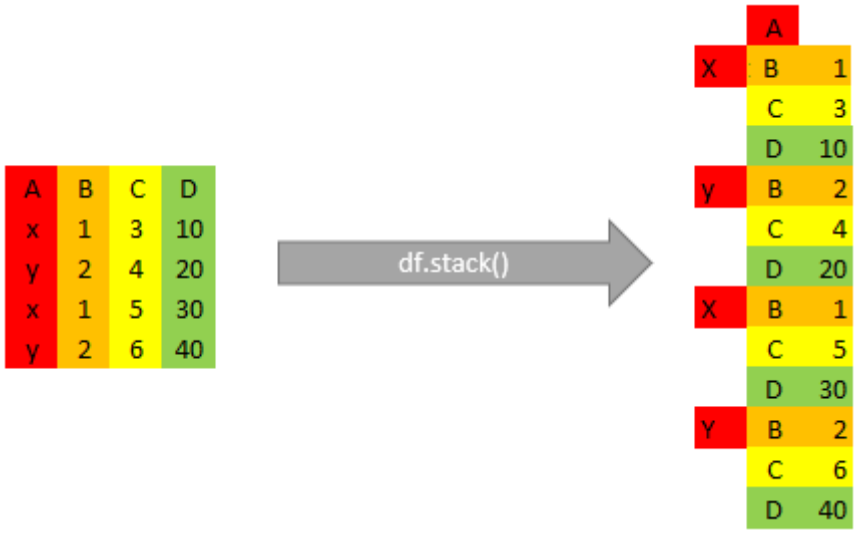
Die generelle pivot-Funktion tauscht Spalten und Zeilen einfach miteinander aus und kommt damit auch mit den Datentypen string oder object zurecht. Für die Verarbeitung von rein numerischen Informationen bietet sich die Funktion pivot_table an, die Daten ähnlich wie das von Excel bekannt ist, aggregiert. In der Standardeinstellung wird dafür der Mittelwert über numpy.mean verwendet. Über den Parameter aggfunc lassen sich aber auch weitere statistische Funktionen wie max, min, stdv oder sum verwenden.




```
print(df.pivot_table(columns=["A"],values=["B","C","D"]))
A   x   y
B   1   2
C   4   5
D  20  30
```

Datenstrukturen stapeln

Ähnlich zum Entpivotieren ist das stapeln oder "stacken". Dafür wird der bestehende Index und die Spalteneinträge zusammen zu einem Multiindex. Die verbleibenden Werte bilden dann die Werte-Spalte.

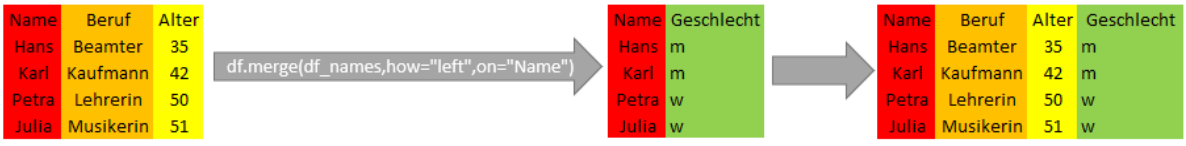


Die Besonderheit ist dabei der Multiindex, so dass man zwei Level von Labeln für den Zugriff auf einzelne Werte zur Verfügung hat.

```
print (df.stack().index)

MultiIndex([('x', 'B'),
            ('x', 'C'),
            ('x', 'D'),
            ('y', 'B'),
            ('y', 'C'),
            ('y', 'D'),
            ('x', 'B'),
            ('x', 'C'),
            ('x', 'D'),
            ('y', 'B'),
            ('y', 'C'),
            ('y', 'D')],
           names=['A', None]
#Zuerst der Zugriff auf das erste Level des Index
print (df["y"])

B     2
C     4
D    20
B     2
C     6
D    40
dtype: int64

# Dann über beide Level des Index
print (df["y","C"])

(y, C)    4
(y, C)    6

# oder wahlweise nur über den zweiten Level
print (df[:,"C"])

A
x    3
y    4
x    5
y    6
```

Daten zusammenführen

Wer schon Erfahrungen hat mit SQL, der fühlt sich beim Verknüpfen von Tabellen zu Hause. Bei SQL join genannt, bei Excel Vlookup, heißt die Funktion bei python entweder join oder merge. Es ist dabei das Ziel, Informationen mehrerer Tabellen über eine Schlüsselspalte zu einer neuen Tabelle zu verknüpfen, in der sich die Informationen aus beiden Quelltabellen finden.



Fangen wir mit einem praktischen Beispiel an. Wir reaktivieren noch einmal unser oben genutztes Dataframe mit Namen, Berufen und dem Alter.

```
import pandas as pd
dict={"Name":pd.Series(["Hans","Karl","Petra","Julia"]),
    "Beruf":pd.Series(["Beamter","Kaufmann","Lehrerin","Musikerin"]),
    "Alter":pd.Series([35,42,50,61])}
df=pd.DataFrame(dict)
```

Zur Vereinfachung haben wir die oben genutzten Doppelnamen auf den ersten Namensbestandteil gekürzt. Wir haben jetzt die Aufgabe erhalten, eine Spalte mit dem Geschlecht hinzuzufügen. Das könnten wir bei den vier Zeilen noch manuell erledigen. Allerdings ist es nicht das Ziel bei der Datenanalyse, Daten manuell zu editieren. Zumal im Produktivbetrieb selten Daten mit nur vier Einträgen über unseren Tisch laufen. Wir brauchen also einen Mechanismus, der vom Namen automatisch auf das Geschlecht kommt.

Wir nutzen dafür eine der vielen verfügbaren Namenslisten im Internet und verknüpfen die Namensspalten beider Tabellen miteinander. Bei einer Übereinstimmung wird das Geschlecht in einer zusätzlichen Spalte angezeigt.

```
#Ich habe für dieses Beispiel die Daten von https://offenedaten-
koeln.de/dataset/vornamen genutzt, die für deutsche Vornamen gute Ergebnisse
liefern

df_names=pd.read_excel(r"Vornamen_Geschlecht.xlsx",header=0)
print (df.merge(df_names,how="left",on="Name"))

    Name      Beruf  Alter Geschlecht
0   Hans    Beamter     35          m
1   Karl   Kaufmann     42          m
2  Petra   Lehrerin     50          w
3  Julia  Musikerin     61          w
```

Wir lesen zuerst die Exceltabelle ein, die aus den beiden Spalten "Name" und "Geschlecht" besteht. Die Spalte "Name" kommt in beiden Datensätzen vor und kann für die Verknüpfung genutzt werden. Der Parameter how="left" besagt, dass die Einträge im Quelldatensatz (hier das Dataframe df) führend sind. Diese werden versucht, mit den Daten des Zieldatensatzes zu verknüpfen. Das Verhalten wird deutlich, wenn wir dem Quelldatensatz einen Eintrag hinzufügen, der sich bestimmt nicht in dem Datensatz df_names findet.

```
df=df.append({"Name":"R2D2","Beruf":"Roboter","Alter":0},ignore_index=True)

    Name      Beruf  Alter Geschlecht
0   Hans    Beamter     35          m
1   Karl   Kaufmann     42          m
2  Petra   Lehrerin     50          w
3  Julia  Musikerin     61          w
4   R2D2    Roboter      0        NaN
```

R2D2 wird zwar in der Ausgabe angezeigt, allerdings ohne Geschlecht, da sich der Name nicht in der Zieltabelle findet. Mit dem Parameter how="right" können wir die Liste der Vornamen mit Geschlecht zur Quelltabelle machen, das Verhalten wird also umgedreht

```
print (df.merge(df_names,how="right",on="Name").head(10))
```

```
     Name      Beruf  Alter Geschlecht
0    Hans    Beamter   35.0          m
1    Karl   Kaufmann   42.0          m
2   Petra   Lehrerin   50.0          w
3   Julia  Musikerin   61.0          w
4   Marie        NaN    NaN          w
5  Sophie        NaN    NaN          w
6   Maria        NaN    NaN          w
7   Maria        NaN    NaN          m
8    Noah        NaN    NaN          m
9  Emilia        NaN    NaN          w
```

R2D2 ist nun wieder verschwunden, da er nicht in der "neuen" Quelltabelle (df_names) auftaucht. Beruf und Alter tauchen jetzt auch nur noch in den Einträgen auf, die sich in unserer "alten" Quelltabelle (df) befinden, da es in df_names keine derartigen Informationen gibt. Mit dem Parameter .head(10) werden nur die ersten 10 Einträge angezeigt. Die neue Quelltabelle df_names würde ansonsten über einige Seiten weiterlaufen.

Probieren wir noch die weiteren Verküpfungsoptionen durch:

```
# Mit inner werden nur die Daten angezeigt, die in beiden Datensätzen vorkommen
print (df.merge(df_names,how="inner",on="Name"))

    Name      Beruf  Alter Geschlecht
0   Hans    Beamter     35          m
1   Karl   Kaufmann     42          m
2  Petra   Lehrerin     50          w
3  Julia  Musikerin     61          w

#Mit outer werden alle Daten angezeigt und nur zugeordnet, wenn möglich
print (df.merge(df_names,how="outer",on="Name").head(10))

     Name      Beruf  Alter Geschlecht
0    Hans    Beamter   35.0          m
1    Karl   Kaufmann   42.0          m
2   Petra   Lehrerin   50.0          w
3   Julia  Musikerin   61.0          w
4    R2D2    Roboter    0.0        NaN
5   Marie        NaN    NaN          w
6  Sophie        NaN    NaN          w
7   Maria        NaN    NaN          w
8   Maria        NaN    NaN          m
9    Noah        NaN    NaN          m
```

Wer aus der SQL-Welt kommt, kennt eher den join-Befehl. Wo liegt also der Unterschied zwischen join und merge bei python? Beim merge lässt sich das Quell-Dataframe nur mit einem Ziel-Dataframe - oder Series verknüpfen. Dafür ist man aber bei der Auswahl der zu verknüpfenden Spalten flexibler. Beim join lassen sich mehrere Ziel-Dataframes mit dem Quell-Dataframe verknüpfen. Allerdings ist man bei den Ziel-Dataframes immer auf den Index angewiesen.

Schauen wir uns das bei einem einfachen Beispiel an, bei dem am Schluss dasselbe Ergebnis herauskommen soll, wie im vorangegangenen Abschnitt. Zuerst bauen wir uns drei Dataframes auf, die nur in der Spalte "Name" übereinstimmen.

```
import pandas as pd

dict1=
{"Name":pd.Series(["Hans","Karl","Petra","Julia"]),"Beruf":pd.Series(["Beamter",
"Kaufmann","Lehrerin","Musikerin"])}
dict2=
{"Name":pd.Series(["Hans","Karl","Petra","Julia"]),"Alter":pd.Series([35,42,50,6
1])}
dict3=
{"Name":pd.Series(["Hans","Karl","Petra","Julia"]),"Geschlecht":pd.Series(["m","
m","w","w"])}

df1=pd.DataFrame(dict1)
df2=pd.DataFrame(dict2)
df3=pd.DataFrame(dict3)
```

Im Ergebnis soll in der Ausgabe pro Zeile der Name, der Beruf, Alter und Geschlecht ausgegeben werden. Tasten wir uns langsam an dieses Ziel heran.

```
print (df1.join([df2,df3]))

ValueError: Indexes have overlapping values: Index(['Name'], dtype='object')
```

Ok, python scheint ein Problem damit zu haben, dass in der Spalte "Name" überlappende Werte zu finden sind. Versuchen wir also, die Spalte "Name" als Schlüsselspalte zu verwenden.

```
print (df1.join([df2,df3],on="Name"))

ValueError: Joining multiple DataFrames only supported for joining on index
```

Zum Glück sind die Fehlermeldungen aussagekräftiger, als bei manch einem Betriebssystem. Setzen wir also den Index auf die Spalte "Name" und starten noch einen Versuch.

```
df1=pd.DataFrame(dict1).set_index("Name")
df2=pd.DataFrame(dict2).set_index("Name")
df3=pd.DataFrame(dict3).set_index("Name")

print (df1.join([df2,df3]))

           Beruf  Alter Geschlecht
Name
Hans     Beamter     35          m
Karl    Kaufmann     42          m
Petra   Lehrerin     50          w
Julia  Musikerin     61          w
```

Das war das Ergebnis, dass wir erwartet hatten.

Es bleibt also festzuhalten, dass die merge-Funktion flexibler ist, wenn nur zwei Datensätzen im Spiel sind. Die join-Funktion kann auch mit mehreren Datensätzen umgehen, dafür allerdings nur indexbasiert.

Datentypen harmonisieren

Pandas stellt den dtype pro Spalte in einem Dataframe ein. Dieses Verhalten ist vergleichbar mit Excel, bei dem auch pro Spalte ein Datentyp festgelegt wird. Eine differenzierte Festlegung pro Spalte ist wichtig, da weitergehende Operationen von dem Vorliegen eines bestimmten dtypes abhängig sind. Stringoperatoren funktionieren nicht bei numerischen Datentypen und arithmetische Funktionen nicht (immer) bei Strings. Excel und python teilen dabei dasselbe Problem. Sie schätzen den dtype anhand der Zeileninhalte pro Spalte. Sind einzelne Werte nicht eindeutig, wird ein generellerer dtype genutzt. Der generellste dtype ist dabei "object", den wir leider auch am häufigsten in Dataframes finden.

Wir verändern einige Einträge in unserem Dataframe, um das Problem deutlich zu machen und geben den dtype pro Spalte aus.

```
dict={"A":pd.Series(["x","y","x",None]),
    "B":pd.Series([1.0,2,1,2]),
    "C":pd.Series(["3",4,5,6]),
    "D":pd.Series([10,20,30,40])}

print (df.dtypes)

A     object
B    float64
C     object
D      int64
```

Dadurch ergeben sich ganz offensichtliche Probleme

```
print(df.C.sum())
TypeError: can only concatenate str (not "int") to str
```

Erst wenn wir einen bestimmten dtype erzwingen, funktionieren numerische Operationen.

```
#Entweder bei der Initialisierung der Series
"C":pd.Series(["3",4,5,6],dtype="int"),
#oder im Nachhinein innerhalb des Dataframes
df["C"]=df["C"].astype(int)

print (df.dtypes)

A     object
B    float64
C      int32
D      int64

print(df.C.sum())

18
```

Im Idealfall geben wir den dtype bei der Initialisierung vor. Das erzwingt einen einheitlichen Datentyp pro Spalte. Allerdings sollte auch immer hinterfragt werden, warum fehlerhafte Werte in einer Spalte vorkommen, unabhängig von reinen Tippfehlern.

Dazu ein Beispiel eines excelverwöhnten Anwenders.

```
Kosten=pd.Series(["10€","15€","18€"])
print (Kosten.dtype)

object

print (Kosten.sum())
10€15€18€   #nicht ganz das erwartete Ergebnis. Einen Currency-Datentyp wie in
Excel gibt es halt in python nicht

#Versuchen wir es mit der Brechstange
Kosten.astype(int)

ValueError: invalid literal for int() with base 10: '10€'

#Es bleibt also nur, das €-Zeichen manuell zu entfernen, und den Datentyp zu
casten
Kosten=Kosten.str.extract(r'(\d*)(€)')[0]
print (Kosten.astype(int).sum())

43
```

Der Datentyp mit dem wahrscheinlich größten Fehlerpotential sind Datumswerte. Das liegt insbesondere dann verschiedenen möglichen Schreibweisen eines Datums und den vielfältigen Berechnungsmöglichkeiten mit Datumswerten.

```
import pandas as pd

dict={"heterogene Datumswerte":pd.Series(["19.09.2006","13.12.1973
20:13","12.06.97","01/01/2000","20-09-2010"])}
df=pd.DataFrame(dict)


  heterogene Datumswerte
0             19.09.2006
1       13.12.1973 20:13
2               12.06.97
3             01/01/2000
4             20-09-2010
```

Grundsätzlich sind die verschiedenen Schreibweisen für das menschliche Auge lesbar. Python versieht die Einträge aber erst einmal mit den dtype "object", es wären also nur Stringoperationen möglich. Damit wir mit den verschiedenen Datumswerten arbeiten können, müssen wir python erst einmal sagen, dass es sich bei den Einträge um den dtype "datetime" handelt.

```
df["homogene Datumswerte"]=pd.to_datetime(df["heterogene Datumswerte"])

  heterogene Datumswerte homogene Datumswerte
0             19.09.2006  2006-09-19 00:00:00
1       13.12.1973 20:13  1973-12-13 20:13:00
2                12.06.97  1997-12-06 00:00:00
3             01/01/2000  2000-01-01 00:00:00
4             20-09-2010  2010-09-20 00:00:00

print (df.dtypes)
heterogene Datumswerte            object
homogene Datumswerte      datetime64[ns]
```

Wir können nun mit der Spalte "homogene Datumswerte" Datumsberechnungen durchführen. Dabei wird unterschieden in skalare- und Listenoperationen, sowie in Zeitreihen -und Zeitpunktberechnungen. Am besten gehen wir die verschiedenen Möglichkeiten mit einigen Beispielen durch.

```
from datetime import datetime as dt
#Wir fügen ein paar Spalten mit Detailinfos hinzu und kürzen die Spalte "homogene
Datumswerte"

df["Wochentag"]=[dt.strftime(x,"%A") for x in df["homogene Datumswerte"]]
df["Monat"]=[dt.strftime(x,"%B") for x in df["homogene Datumswerte"]]
df["Jahr"]=[dt.strftime(x,"%Y") for x in df["homogene Datumswerte"]]
df["homogene Datumswerte"]=[dt.strftime(x,"%d-%m-%Y") for x in df["homogene
Datumswerte"]]

  heterogene Datumswerte homogene Datumswerte Wochentag      Monat  Jahr
0             19.09.2006           19-09-2006   Tuesday  September  2006
1       13.12.1973 20:13           13-12-1973  Thursday   December  1973
2                12.06.97           06-12-1997  Saturday   December  1997
3             01/01/2000           01-01-2000  Saturday    January  2000
4             20-09-2010           20-09-2010    Monday  September  2010

#Aber Achtung! Durch strftime werden die Werte wieder zu einem object gecastet.
Für weitergehende Berechnungen muss wieder umgewandelt werden.
df["Vergangene Tage"]=pd.to_datetime(df["homogene Datumswerte"]) -
pd.to_datetime('today')

  heterogene Datumswerte homogene Datumswerte Wochentag      Monat  Jahr
Vergangene Tage
0             19.09.2006           19-09-2006   Tuesday  September  2006  -5099
days +06:03:40.282310
1       13.12.1973 20:13           13-12-1973  Thursday   December  1973 -17067
days +06:03:40.282310
2                12.06.97           06-12-1997  Saturday   December  1997  -8485
days +06:03:40.282310
3             01/01/2000           01-01-2000  Saturday    January  2000  -7552
days +06:03:40.282310
4             20-09-2010           20-09-2010    Monday  September  2010  -3637
days +06:03:40.282310

df["Vergangene Tage"]=df["Vergangener Zeitraum"].dt.days
  heterogene Datumswerte homogene Datumswerte Wochentag      Monat  Jahr
Vergangener Zeitraum  Vergangene Tage
```

```
0             19.09.2006           19-09-2006   Tuesday  September  2006  -5099
days +06:00:52.243508           -5099
1       13.12.1973 20:13           13-12-1973  Thursday   December  1973 -17067
days +06:00:52.243508          -17067
2               12.06.97           06-12-1997  Saturday   December  1997  -8485
days +06:00:52.243508           -8485
3             01/01/2000           01-01-2000  Saturday    January  2000  -7552
days +06:00:52.243508           -7552
4             20-09-2010           20-09-2010    Monday  September  2010  -3637
days +06:00:52.243508           -3637
```

```
#Am Schluss können wir die Ausgabe noch auf Deutsch umstellen
import locale
locale.setlocale(locale.LC_ALL,"de_DE.utf8")
```

```
  heterogene Datumswerte homogene Datumswerte   Wochentag      Monat  Jahr
  Vergangener Zeitraum  Vergangene Tage
0             19.09.2006           19-09-2006    Dienstag  September  2006
 -5100 days +09:34:21.840195           -5100
1       13.12.1973 20:13           13-12-1973  Donnerstag   Dezember  1973
-17068 days +09:34:21.840195          -17068
2               12.06.97           06-12-1997     Samstag   Dezember  1997
 -8486 days +09:34:21.840195           -8486
3             01/01/2000           01-01-2000     Samstag     Januar  2000
 -7553 days +09:34:21.840195           -7553
4             20-09-2010           20-09-2010      Montag  September  2010
 -3638 days +09:34:21.840195           -3638
```

```
#Nun können wir mit den Datumswerten rechnen. Da durch strftime die betroffenen
Spalten wieder auf den dtype "object" gecastet wurden, sind eingangs nur
Stringoperationen möglich.
print (df[df["Monat"]=="December"])
```

```
  heterogene Datumswerte homogene Datumswerte Wochentag     Monat  Jahr
Vergangener Zeitraum  Vergangene Tage
1       13.12.1973 20:13           13-12-1973  Thursday  December  1973 -17067
days +05:35:37.460442          -17067
2               12.06.97           06-12-1997  Saturday  December  1997  -8485
days +05:35:37.460442           -8485
```

```
#Das sorgt bei Nichtbeachtung für seltsame Ergebnisse
print(df[df["homogene Datumswerte"] < "01.01.2000"])
```

```
  heterogene Datumswerte homogene Datumswerte Wochentag   Monat  Jahr
 Vergangener Zeitraum  Vergangene Tage
3             01/01/2000           01-01-2000   Samstag  Januar  2000 -7553 days
+09:12:19.276894           -7553
```

```
#Python vergleicht hier die Stringwerte miteinander. Und da ist nur "01-01-2000"
kleiner als "01.01.2000". Wenn das Datum verglichen werden soll, muss also wieder
in ein Datetime-Object gecastet werden.
print(df[pd.to_datetime(df["homogene Datumswerte"]) < "01.01.2000"])
```

```
  heterogene Datumswerte homogene Datumswerte   Wochentag     Monat  Jahr
  Vergangener Zeitraum  Vergangene Tage
1       13.12.1973 20:13           13-12-1973  Donnerstag  Dezember  1973 -17068
days +09:09:20.101490          -17068
```

```
2              12.06.97          06-12-1997     Samstag  Dezember  1997  -8486
days +09:09:20.101490           -8486

#Damit ist auch die Filterung nach Datumsbereichen möglich
print(df[(pd.to_datetime(df["homogene Datumswerte"]) < "01.01.2000") &
(pd.to_datetime(df["homogene Datumswerte"]) > "01.01.1990")])

  heterogene Datumswerte homogene Datumswerte Wochentag     Monat  Jahr
 Vergangener Zeitraum  Vergangene Tage
2              12.06.97          06-12-1997   Samstag  Dezember  1997 -8486
days +09:07:09.998526           -8486
```

Daten laden mit Python

Wir haben in den ersten beiden Prozessschritten Daten aus einer Quelle extrahiert und so aufbereitet, dass sie überhaupt auswertbar sind. Im dritten und letzten Schritt geht es nun darum, die Daten so darzustellen oder sie in einem neuen Blickwinkel zu betrachten, dass damit neue Erkenntnisse erlangt werden können. Folgende Methoden kommen beim Laden von Daten zum Zuge:

- Statistische Kennzahlen wie mean, stdv, min, max ...
- grafische oder numerische Darstellung der Verteilung der Daten, zb. über die Standardnormalverteilung
- Visualisierung von Daten, um auch die rechte Gehirnhälfte des Betrachters anzuregen

Mit Methoden wie der Regressionsanalyse oder Entscheidungsbäumen kratzen wir damit auch an der Grenze der künstlichen Intelligenz.

Laden von quantitativen Daten

Hier zeigt sich nun der Vorteil, wenn wir es im letzten Kapitel bei der Transformation der Daten geschafft haben, möglichst viele Werte numerisch, also als int, float oder datetime abzubilden. Zahlen lassen sich statistisch deutlich einfacher verarbeiten, als Texte.

Wir nehmen noch einmal die statistische Auswertung der Verkehrsunfälle in Schleswig-Holstein aus dem letzten Kapitel und lassen uns die statistischen Informationen zu den Daten anzeigen.

```
import pandas as pd

url="https://www.statistik-
nord.de/fileadmin/Dokumente/Statistische_Berichte/verkehr_umwelt_und_energie/H_I
_1_m_S/H_I_1-m2006_SH.xlsx"
df=pd.read_excel(url,sheet_name="T1_1",skiprows=8)
lst_columns=
["Jahr","Monat","insgesamt","Personenschaden","Sachschäden","Alkoholeinfluss","Ü
brige","Getötete","Verletzte"]
df.columns=lst_columns
df["Jahr"]=df["Jahr"].fillna(method="ffill")
df=df.dropna()
df.drop(df[df.Jahr=="Summe"].index, inplace=True)
for column in (df.columns):
    if df[column].dtype !="object":
        df[column]=df[column].astype(int)

print (df.describe())
```

```
         insgesamt  Personenschaden  Sachschäden  Alkoholeinfluss       Übrige
Getötete    Verletzte
count    24.000000        24.000000    24.000000        24.000000    24.000000
 24.000000    24.000000
mean   7334.916667       988.333333   155.541667        33.500000  6157.541667
9.500000  1268.500000
std     939.301355       196.627803    27.791114         7.494926   769.261531
4.283436   264.873391
min    5049.000000       564.000000   102.000000        22.000000  4337.000000
2.000000   736.000000
25%    6780.000000       857.000000   135.250000        28.000000  5713.500000
6.750000  1092.750000
50%    7692.500000      1003.500000   153.500000        33.500000  6394.500000
9.500000  1274.000000
75%    8076.500000      1126.750000   177.000000        39.250000  6713.250000
 12.250000  1421.000000
max    8537.000000      1303.000000   214.000000        46.000000  7285.000000
 18.000000  1758.000000
```

Schon mit dieser einfachen Ausgabe erhalten wir zusätzliche Infos zu unseren Daten, die in vielen Fällen schon weiterhelfen. Die statistischen Funktionen lassen sich natürlich auch einzeln aufrufen.

```
print (df.mean())

Jahr               2019.000000
insgesamt          7334.916667
Personenschaden     988.333333
Sachschäden         155.541667
Alkoholeinfluss      33.500000
Übrige             6157.541667
Getötete              9.500000
Verletzte          1268.500000
```

Durch statistische Funktionen werden die Daten auf eine Kennzahl aggregiert. Wir werden jetzt aber mit der Frage konfrontiert, ob die Jahreszeit eine Auswirkung auf die Anzahl der Verkehrsunfälle hat. Da bietet sich eher eine visuelle Auswertung an. Wir nehmen dafür für dieses Beispiel die beiden Spalten "Monat" und die Gesamtanzahl der Verkehrsunfälle in der Spalte "insgesamt". Nun haben wir aber die Herausforderung, dass die Monate mehrfach(einmal pro Jahr) in der Spalte auftauchen. Eine direkte grafische Ausgabe würde also keinen Mehrwert bieten. Also aggregieren wir zuerst die Werte pro Monat und arbeiten mit dem Mittelwert. Dafür haben wir im letzten Kapitel eine sinnvolle Funktion kennengelernt, die wir nun praktisch anwenden können.

```
df_mean=df.pivot_table(columns="Monat",values="insgesamt")

Monat        April  August  Dezember  Februar  Januar    Juli    Juni     Mai
 März  November  Oktober  September
insgesamt  6538.0  8006.0    7538.0   6366.5  6775.0  7943.5  7618.0  7537.5
 6002.0    8179.0   7941.0     7574.5
```

Die Monatszeilen sind nun zu Spaltenüberschriften geworden und die doppelten Werte über ihren Mittelwert aggregiert. Schon jetzt ist bei den Spaltenüberschriften ein Problem erkennbar, um das wir uns gleich noch kümmern müssen. Zuerst entpivotieren wir die Tabelle aber wieder, damit die für uns relevanten Informationen in einer Spalte stehen.

```
df_melt=df_mean.melt()

        Monat   value
0       April  6538.0
1      August  8006.0
2    Dezember  7538.0
3     Februar  6366.5
4      Januar  6775.0
5        Juli  7943.5
6        Juni  7618.0
7         Mai  7537.5
8       März  6002.0
9    November  8179.0
10    Oktober  7941.0
11  September  7574.5
```

Das entspricht schon ungefähr dem von uns gewünschten Ergebnis. Nun sticht das vorhin schon angesprochene Problem aber noch deutlicher ins Auge: Die Monatsnamen sind nicht mehr aufsteigend sortiert. Die Tabelle ließe sich in der Form zwar visualisieren, aber mit der Reihenfolge der Monate wird immer noch nicht viel Aussagekraft erzeugt. Zum sortieren der Monate müssen wir allerdings etwas tiefer in die Trickkiste greifen.
Zuerst erstellen wir ein Dictionary mit der Zuordnung der Monate zu einer numerischen Reihenfolge.

```
dict_monate = {"Januar":1, "Februar":2,"März":3,"April":4,"Mai":5,"Juni":6,

 "Juli":7,"August":8,"September":9,"Oktober":10,"November":11,"Dezember":12}
```

Danach erstellen wir eine neue Spalte und ordnen die Monatsnamen ihrer numerischen Reihenfolge zu.

```
df_melt["Monatsindex"]=[dict_monate.get(monat) for monat in df_melt["Monat"]]

        Monat   value  Monatsindex
0       April  6538.0            4
1      August  8006.0            8
2    Dezember  7538.0           12
3     Februar  6366.5            2
4      Januar  6775.0            1
5        Juli  7943.5            7
6        Juni  7618.0            6
7         Mai  7537.5            5
8       März  6002.0            3
9    November  8179.0           11
10    Oktober  7941.0           10
11  September  7574.5            9
```

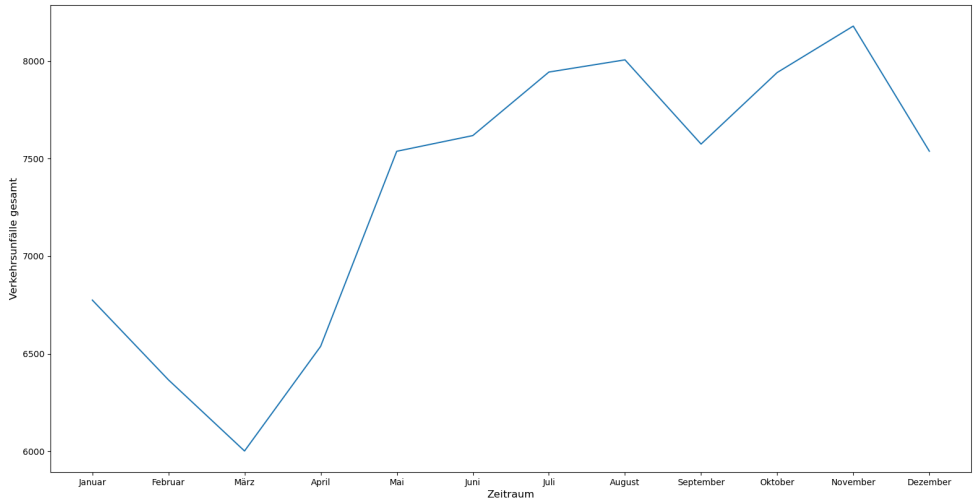
Nun ist das Ziel in greifbarer Nähe. Wir machen aus der Spalte "Monatsindex" nun den Index der Tabelle und sortieren sie.

```
df_melt=df_melt.set_index("Monatsindex").sort_index()

                Monat   value
Monatsindex
```

```
1              Januar  6775.0
2             Februar  6366.5
3               März  6002.0
4              April  6538.0
5                Mai  7537.5
6               Juni  7618.0
7               Juli  7943.5
8             August  8006.0
9          September  7574.5
10           Oktober  7941.0
11          November  8179.0
12          Dezember  7538.0
```

Nun können wir in einem Plot die Monate auf der X-Achse und die Anzahl der Verkehrsunfälle auf der Y-Achse unterbringen.

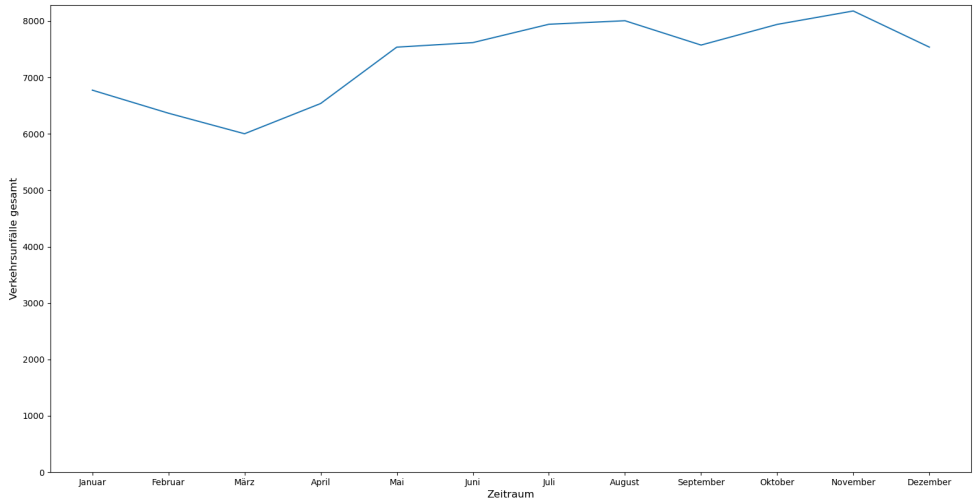
```
plt.plot(df_melt["Monat"],df_melt["value"])
plt.xlabel("Zeitraum",fontsize=12)
plt.ylabel("Verkehrsunfälle gesamt",fontsize=12)
plt.show()
```



Man kann jetzt deutlich den Zusammenhang sehen zwischen der Anzahl der Verkehrsunfälle und der Jahreszeit. Statistisch gesprochen, wir haben eine Korrelation entdeckt. Das Herauslesen der Kausalität ist aber wiederum nicht die Aufgabe der Datenanalysten.

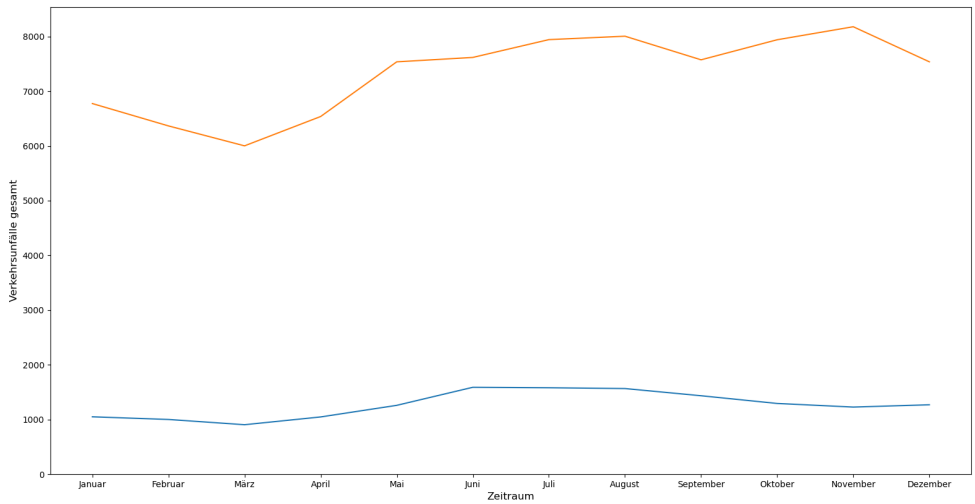
Allerdings tritt bei der Ausgabe wieder das Problem der abgeschnittenen Y-Achse auf. Dadurch wird das Ergebnis verzerrt, da es auf den ersten Blick so erscheint, als würden im März kaum Verkehrsunfälle passieren. Um unser Ergebnis etwas objektiver zu bekommen, lassen wir die Y-Achse bei Null beginnen.

```
plt.yticks(np.arange(0, 9000, step=1000))
```



Ich schätze, damit wurde das erste Ergebnis schon reichlich relativiert. Bringen wir jetzt noch zusätzlich die Anzahl der Verletzten mit ins Spiel.

```
df_mean_verletzt=df.pivot_table(columns="Monat",values="Verletzte")
df_melt_verletzt=df_mean_verletzt.melt()
df_melt_verletzt["Monatsindex"]=[dict_monate.get(monat) for monat in
df_melt_verletzt["Monat"]]
df_melt_verletzt=df_melt_verletzt.set_index("Monatsindex").sort_index()

plt.plot(df_melt_verletzt["Monat"],df_melt_verletzt["value"])
plt.plot(df_melt["Monat"],df_melt["value"])
plt.xlabel("Zeitraum",fontsize=12)
plt.ylabel("Verkehrsunfälle gesamt",fontsize=12)
plt.yticks(np.arange(0, 9000, step=1000))
plt.show()
```

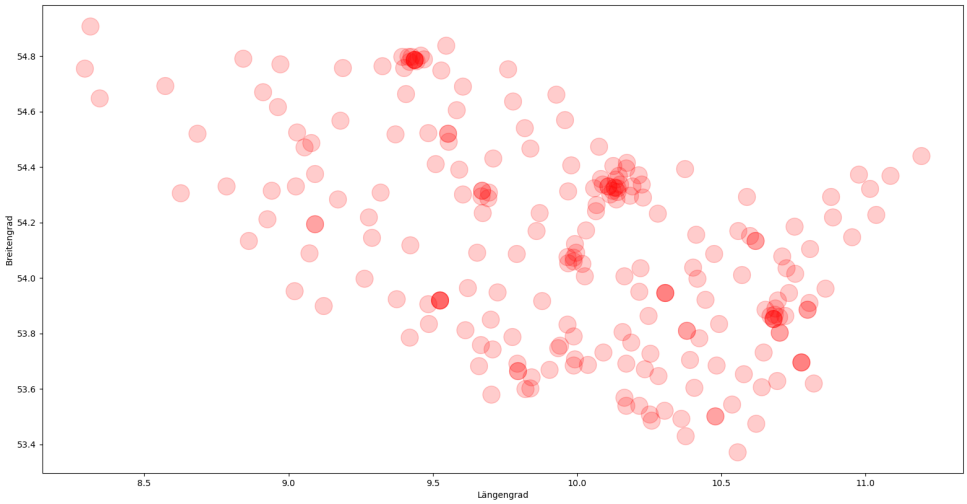


In einem nächsten Beispiel wollen wir geographische Informationen visuell abbilden. Dafür holen wir uns zuerst die Koordinaten aller Polizeidienststellen in Schleswig-Holstein und löschen leere Einträge in den Spalten "longitude" und "latitude". Denn auf die kommt es im weiteren an.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

url="https://opendata.zitsh.de/data/zufish/polizei_2020-06-07.csv"
df=pd.read_csv(url,error_bad_lines=False,delimiter="\t")
df=df.dropna(subset=["longitude","latitude"])
```

Wenn wir die Datenpunkte in ein Scatterdiagramm eintragen, erkennen wir mit ein wenig Phantasie schon die Umrisse von Schleswig-Holstein, auf jeden Fall aber die Ballungsräume Flensburg, Kiel und Lübeck.

```
plt.scatter(df["longitude"],df["latitude"],s=400,alpha=0.2,c="r")
print (plt.axis())
plt.xlabel("Längengrad")
plt.ylabel("Breitengrad")
plt.show()
```



Es ist aber natürlich nicht Sinn der Sache, zu raten, was das Bild uns sagen will. Von daher würden wir die Daten mit etwas mehr Details anfüttern. Wir hinterlegen die Grafik mit einem Hintergrundbild von Schleswig-Holstein und tragen die größten Städte ein.

Wir laden zuerst eine Liste der größten Städte in Schleswig-Holstein.

```
url_städte="https://de.wikipedia.org/wiki/Liste_der_gr%C3%B6%C3%9Ften_St%C3%A4dte_in_Schleswig-Holstein"
df=pd.read_html(url_städte)
df=df[0]
```

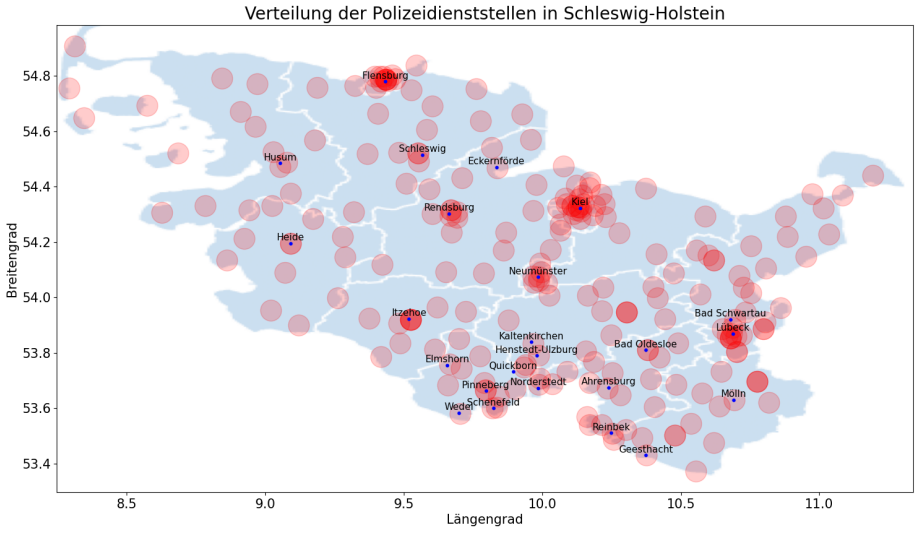
Wir gehen jetzt die Liste der Städte der Reihe nach durch und suchen bei Bingmaps nach den entsprechenden GPS-Koordinaten.

```
lst_städte=[]
lst_latitude=[]
lst_longitude=[]
for stadt in (df["Stadt"]):
    lst_städte.append(stadt)
```

```
    url_teil1="http://dev.virtualearth.net/REST/v1/Locations/"
    url_teil2=",sh?o=xml&key={mein geheimer Bing-Key}"
    r=requests.get(url_teil1+stadt+url_teil2)
    xml=r.text
    soup=BeautifulSoup(xml,"xml")
    lst_latitude.append(soup.Point.Latitude.string)
    lst_longitude.append(soup.Point.Longitude.string)

df_städte=pd.DataFrame(columns=["Stadt","Latitude","Longitude"])
df_städte["Stadt"]=lst_städte
df_städte["Latitude"]=lst_latitude
df_städte["Longitude"]=lst_longitude
df_städte["Latitude"]=df_städte["Latitude"].astype("float")
df_städte["Longitude"]=df_städte["Longitude"].astype("float")
```

Zum Schluss fügen wir die Koordinaten der Städte noch dem Scatterdiagramm hinzu und beschriften die Einträge.

```
img=plt.imread(r"C:\map_sh.png")
#Das Hintergrundbild muss dieselben Ausmaße haben, wie unsere Koordinaten.
Ansonsten wird die Grafik verzerrt angezeigt.
plt.imshow(img,extent=[8.247308456715733, 11.33849532271057, 53.296954406000005,
54.984367474])
plt.scatter(df_städte["Longitude"],df_städte["Latitude"],s=10,c="b")
for i,txt in enumerate(df_städte["Stadt"]):
    plt.annotate(txt,xy=
(df_städte.iloc[i,2],df_städte.iloc[i,1]),ha="center",xytext=
(df_städte.iloc[i,2],df_städte.iloc[i,1]+0.01))
```



Der gesamte Quellcode sieht inzwischen folgendermaßen aus.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from bs4 import BeautifulSoup
import requests

url_städte="https://de.wikipedia.org/wiki/Liste_der_gr%C3%B6%C3%9Ften_St%C3%A4dt
e_in_Schleswig-Holstein"
```

```
df=pd.read_html(url_städte)
df=df[0]
lst_städte=[]
lst_latitude=[]
lst_longitude=[]
for stadt in (df["Stadt"]):
    lst_städte.append(stadt)
    url_teil1="http://dev.virtualearth.net/REST/v1/Locations/"
    url_teil2=",sh?o=xml&key={hier ist mein Api-Key von Bing}"
    r=requests.get(url_teil1+stadt+url_teil2)
    xml=r.text
    soup=BeautifulSoup(xml,"xml")
    lst_latitude.append(soup.Point.Latitude.string)
    lst_longitude.append(soup.Point.Longitude.string)

df_städte=pd.DataFrame(columns=["Stadt","Latitude","Longitude"])
df_städte["Stadt"]=lst_städte
df_städte["Latitude"]=lst_latitude
df_städte["Longitude"]=lst_longitude
df_städte["Latitude"]=df_städte["Latitude"].astype("float")
df_städte["Longitude"]=df_städte["Longitude"].astype("float")

url="https://opendata.zitsh.de/data/zufish/polizei_2020-06-07.csv"
df=pd.read_csv(url,error_bad_lines=False,delimiter="\t")
df=df.dropna(subset=["longitude","latitude"])
img=plt.imread(r"C:\map_sh.png")
plt.imshow(img,extent=[8.247308456715733, 11.33849532271057, 53.296954406000005,
54.984367474],alpha=0.2)

plt.scatter(df["longitude"],df["latitude"],s=600,alpha=0.2,c="r")
plt.scatter(df_städte["Longitude"],df_städte["Latitude"],s=10,c="b")
plt.xlabel("Längengrad",fontsize=15)
plt.ylabel("Breitengrad",fontsize=15)
plt.title("Verteilung der Polizeidienststellen in Schleswig-
Holstein",fontsize=20)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
for i,txt in enumerate(df_städte["Stadt"]):
    plt.annotate(txt,xy=
(df_städte.iloc[i,2],df_städte.iloc[i,1]),ha="center",xytext=
(df_städte.iloc[i,2],df_städte.iloc[i,1]+0.01),fontsize=13)
plt.show()
```

Textklassifikation mit Python

Wir haben uns im Laufe dieses Tutorials hauptsächlich mit HTML-Seiten, CSV - oder Excel-Dateien beschäftigt. Dabei lag der Fokus auf dem Extrahieren und Transformieren der Daten. Das nachfolgende Beispiel werden wir abwechslungsweise eine reine Textdatei extrahieren, dann aber fortgeschrittenere Transformationen und Auswertungen über diesen Text fahren. Wir berühren damit den Bereich der Textklassifikation bzw. dem NLP (Natural Language Processing), was schon zu den ML-Mechanismen gehört.

Fangen wir damit an, Goethes Faust 1 als Textdatei in eine Liste zu laden.

```
import re
import requests

url="http://www.datashaping.de/faust.txt"
fh=requests.get(url)
zeilen=fh.text.split("\n")
```

Damit wären wir mit dem Prozessschritt "Extrahieren von Daten" schon fertig!

Der nächste Prozessschritt ist nicht ganz so schnell erledigt. Wir müssen die Daten in eine Struktur bringen, die später auswertbar ist. Es bietet sich dabei an, die vorgegebene Struktur des Buches zu nutzen. Unser Ziel wäre es dabei, dass die Informationen in folgender Datenstruktur vorliegen.

```
["Kapitel","Sprecher","Vers","Stimmung"]
```

Damit können wir später jeden Satz einem Sprecher und/oder einem Kapitel zuordnen. Da die Kapitelüberschriften dabei keiner besonderen Systematik folgen, würde ich dort eine Abkürzung nehmen, und mir die Auswertung der Kapitelüberschriften etwas zurechtschummeln.

```
lst_kapitel=[
"Zueignung.\r",
"Vorspiel auf dem Theater\r",
"Prolog im Himmel.\r",
"Vor dem Tor\r",
"Studierzimmer\r",
"Auerbachs Keller in Leipzig\r",
"Hexenküche.\r",
"Straße (I)\r",
"Abend. Ein kleines reinliches Zimmer\r",
"Spaziergang\r",
"Der Nachbarin Haus\r",
"Straße (II)\r",
"Garten\r",
"Ein Gartenhäuschen\r",
"Wald und Höhle\r",
"Gretchens Stube.\r",
"Marthens Garten\r",
"Am Brunnen\r",
"Zwinger\r",
"Nacht. Straße vor Gretchens Türe\r",
"Dom\r",
"Walpurgisnacht.\r",
"Walpurgisnachtstraum\r",
"Trüber Tag. Feld\r",
"Nacht, offen Feld\r",
"Kerker\r"
]

#Danach gehen wir jede Zeile durch, und ordnen die Sprecher mit ihrem Redeanteil
den einzelnen Kapiteln zu.
for zeile in zeilen:
    #Handelt es sich bei der Zeile um ein Kapitel?
    if zeile in lst_kapitel:
        kapitel=zeile
        dict_faust[kapitel]=[]
```

```
    #Handelt es sich um einen Sprecher?
    elif re.match(r"^ [A-ZÜÖÄ\s].*:",zeile):
            sprecher=re.split(r"[(:]",zeile)[0].strip()
            dict_faust[kapitel].append (sprecher)

#und Ausgabe des ersten und letzten Kapitels
print(dict_faust.get("Zueignung.\r"))
print(dict_faust.get("Kerker\r"))
```

```
[]
[' MARGARETE ', ' FAUST ', ' MARGARETE ', ' FAUST', ' MARGARETE ', ' FAUST', '
MARGARETE', ' FAUST ', ' MARGARETE ', ' FAUST ', ' MARGARETE ', ' FAUST', '
MARGARETE', ' FAUST ', ' MARGARETE', ' FAUST', ' MARGARETE', ' FAUST', '
MARGARETE ', ' FAUST', ' MARGARETE', ' FAUST', ' MARGARETE', ' MARGARETE', '
FAUST', ' MARGARETE', ' FAUST', ' MARGARETE', ' FAUST', ' MARGARETE', ' FAUST',
' MARGARETE', ' FAUST', ' MARGARETE', ' FAUST', ' MARGARETE', ' FAUST', '
MARGARETE', ' FAUST', ' MEPHISTOPHELES ', ' MARGARETE', ' FAUST', ' MARGARETE',
' MEPHISTOPHELES ', ' MARGARETE', ' MEPHISTOPHELES', ' STIMME ', '
MEPHISTOPHELES ', ' STIMME ']
```

Im ersten Kapitel gibt es keine Sprecher, von daher ist die Liste leer. Für die anderen Kapitel haben wir aber eine Liste erhalten, wann welcher Sprecher zu Wort kam. Wir kümmern uns später darum, was er sagte.

Als nächstes wollen wir herausfinden, wer die Dialoge dominierte. Dabei interessieren uns vorerst nur die Anzahl der Dialoge (nicht die Länge) pro Sprecher. Dafür kopieren wir unser gerade erstelltes Dictionary, da wir mit dem Original später noch weiterarbeiten wollen und zählen über eine Counter-Instanz die Anzahl der Dialoge.

```
from collections import Counter

dict_count_speaker=dict_faust
for key in dict_count_speaker.keys():
    dict_count_speaker[key] = dict(Counter(dict_count_speaker[key]))
```

```
{'Zueignung.\r': Counter(), 'Vorspiel auf dem Theater\r': Counter({' DIREKTOR':
4, ' DICHTER': 4, ' LUSTIGE PERSON': 3, ' Direktor. Theatherdichter. Lustige
Person': 1}), 'Prolog im Himmel.\r': Counter({' FAUST': 12, ' MEPHISTOPHELES':
7, ...
```

Danach ermitteln wir die Anzahl der Dialoge pro Sprecher über alle Kapitel und extrahieren sie in ein neues Dictionary.

```
dict_most_common_speaker={}

for item in dict_count_speaker.items():
    for key in item[1].keys():
        alte_anzahl=dict_most_common_speaker.get(key)
        neue_anzahl=item[1].get(key)
        if alte_anzahl:
            dict_most_common_speaker[key]=alte_anzahl+neue_anzahl
        else:
            dict_most_common_speaker[key]=neue_anzahl
```

```
{'Direktor. Theatherdichter. Lustige Person': 1, 'DIREKTOR': 4, 'DICHTER': 4,
'LUSTIGE PERSON': 3, 'RAPHAEL': 1,...
```

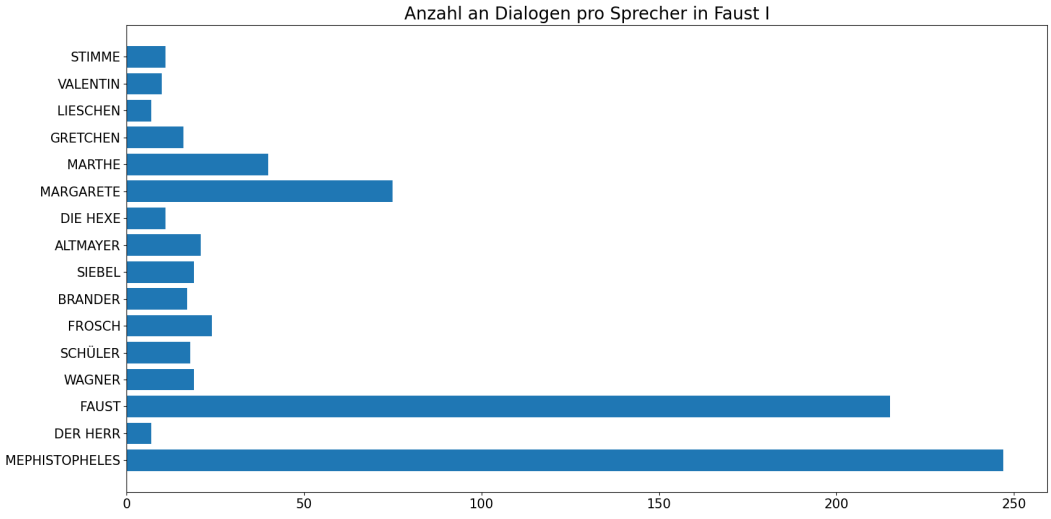
Da in dem Dictionary nun viele Sprecher mit nur einem Dialog auftauchen, reduzieren wir das Dictionary auf eine Mindestanzahl von 5 Dialogen. Da das Dictionary beim iterieren nicht inline geändert werden kann, müssen wir die Ergebnisse wiederum in ein neues Dictionary kopieren.

```
dict_most_common_speaker_red={}

for key in dict_most_common_speaker:
    if dict_most_common_speaker.get(key)>5:
        dict_most_common_speaker_red[key]=dict_most_common_speaker.get(key)
```

```
{'MEPHISTOPHELES': 247, 'DER HERR': 7, 'FAUST': 215, 'WAGNER': 19, 'SCHÜLER':
18, 'FROSCH': 24, 'BRANDER': 17, 'SIEBEL': 19, 'ALTMAYER': 21, 'DIE HEXE': 11,
'MARGARETE': 75, 'MARTHE': 40, 'GRETCHEN': 16, 'LIESCHEN': 7, 'VALENTIN': 10,
'STIMME': 11}
```

Am Schluss geben wir das Ergebnis noch grafisch aus.

```
labels = list(dict_most_common_speaker_red.keys())
data = np.array(list(dict_most_common_speaker_red.values()))
plt.barh(labels,data)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("Anzahl an Dialogen pro Sprecher in Faust I",fontsize=20)
plt.show()
```



Das Faust und Mephisto als die beiden Hauptpersonen den größten Redeanteil haben, war zu erwarten. Die Dominanz von Mephisto ist allerdings deutlich erkennbar. Mehr hatte ich eigentlich auch von Gretchen erwartet, da sie ja eigentlich die dritte Hauptperson im Bunde ist.

Wir haben bis jetzt analysiert, wieviel jeder Protagonist in dem Buch gesprochen hat. Im nächsten Schritt wollen wir nun eine inhaltliche Analyse der Dialoge vornehmen. Wir führen dafür eine Sentinent-Analyse der Dialog pro Sprecher durch, um zu sehen, ob sich die Stimmung des Protagonisten innerhalb des Buches (also von Kapitel zu Kapitel) verändert. Dazu nutzen wir die Textblob-de Bibliothek. Damit kann die Wortwahl eines Satzes analysiert werden, um darauf Rückschlüsse auf die Stimmung des Sprechers zu ziehen. Die Stimmung wird dabei auf einer Skala von -1 (sehr schlechte Stimmung) über 0 (neutral) bis +1 (sehr gute Stimmung) dargestellt. Ich zeige das Vorgehen exemplarisch an ausgewählten Dialogen.

```
from textblob_de import TextBlobDE as TextBlob

Mephisto= '''
Wie traurig steigt die unvollkommne Scheibe
Des roten Monds mit später Glut heran
Und leuchtet schlecht, daß man bei jedem Schritte
Vor einen Baum, vor einen Felsen rennt!
'''
Margarete='''
Sie schlief, damit wir uns freuten.
Es waren glückliche Zeiten!
'''
blob_mephisto = TextBlob(Mephisto)
blob_margarete=TextBlob(Margarete)

print("Stimmung von Mephisto: "+ str(blob_mephisto.sentiment))
print("Stimmung von Margarete: "+ str(blob_margarete.sentiment))

Stimmung von Mephisto: Sentiment(polarity=-1.0, subjectivity=0.0)
Stimmung von Margarete: Sentiment(polarity=0.5, subjectivity=0.0)
```

Eigentlich hätte ich vom Teufel eine bessere Laune erwartet!

Diese Auswertung funktioniert am besten bei kurzen Sätzen mit prägnanten Adjektiven. Wir könne aber auch eine Auswertung über den gesamten Text von Faust I durchführen.

```
from textblob_de import TextBlobDE as TextBlob
import requests

url="http://www.datashaping.de/faust.txt"
fh=requests.get(url)

blob_faust=TextBlob(fh.text)

print("Die grundsätzliche Stimmung in Faust: "+ str(blob_faust.sentiment))

Die grundsätzliche Stimmung in Faust: Sentiment(polarity=0.06586225144590556,
subjectivity=0.03041068720102671)
```

Das Ergebnis ist eine minimal positive Grundstimmung und eine ebenfalls minimale Subjektivität, d.h. es werden eher Gefühle und Emotionen dargestellt, als Fakten.

Jetzt müssen wir noch einen Schritt nachholen, den wir anfangs übersprungen haben. Unsere eigentliche Zielarchitektur der Daten sah folgendermaßen aus.

```
["Kapitel","Sprecher","Vers","Stimmung"]
```

Wir sind bisher aber nur so weit gekommen.

```
{Kapitel:[Sprecher]}
```

Die von den Sprechern gesprochenen Verse fehlen noch in unserer Datenstruktur. Da wir die für die Sentinent-Analyse benötigen, fügen wir die Verse noch hinzu.

```
kapitel="Leer"
```

```
sprecher="Leer"

lst_faust=[]

for zeile in zeilen:
    #Handelt es sich bei der Zeile um ein Kapitel?
    if zeile in lst_kapitel:
        kapitel=zeile.strip()
    #Handelt es sich um einen Sprecher?
    elif re.match(r"^ [A-ZÜÖÄ\s].*:",zeile):
            sprecher=re.split(r"[(:]",zeile)[0].strip()
    #Ansonsten sollte es ein reiner Vers sein Ab hier fügen wir die Infos einer
Liste hinzu
    else:
        blob_zeile=TextBlob(zeile)
        lst_faust.append([kapitel,sprecher,zeile.strip(),blob_zeile.polarity])

print (lst_faust[0])

['Zueignung.', 'Leer', 'Ihr naht euch wieder, schwankende Gestalten,', 0.0]
```

Die erste Zeile gehört zwar zum Kapitel "Zueignung", ist aber keinem Sprecher zugeordnet. Der erste Satz ist auch nicht außergewöhnlich emotional, von daher ist der Sentinent-Wert auch 0.0.

Damit wir langsam wieder in vertraute Fahrwasser kommen, werden wir unsere Liste in eine Datenstruktur konvertieren, die uns aus den letzten Kapiteln bekannt sein müsste.

```
df_faust=pd.DataFrame(lst_faust,columns=
["Kapitel","Sprecher","Vers","Stimmung"])

         Kapitel        Sprecher
 Vers  Stimmung
0     Zueignung.            Leer      Ihr naht euch wieder, schwankende
Gestalten,      0.00
1     Zueignung.            Leer     Die früh sich einst dem trüben Blick
gezeigt.      0.00
2     Zueignung.            Leer      Versuch ich wohl, euch diesmal
festzuhalten?      0.00
3     Zueignung.            Leer       Fühl ich mein Herz noch jenem Wahn
geneigt?      0.00
4     Zueignung.            Leer  Ihr drängt euch zu! nun gut, so mögt ihr
walten,      0.15
```

Jetzt können wir alles Erlernte aus den letzten Kapiteln anwenden, um mehr Informationen über den Text zu erhalten.

```
#Wer verbreitet die schlechteste Stimmung?
print (df_faust.loc[df_faust.Stimmung ==-1.0])

                     Kapitel        Sprecher
            Vers  Stimmung
13                  Zueignung.            Leer               Des Lebens
labyrinthisch irren Lauf,      -1.0
69    Vorspiel auf dem Theater         DICHTER         Verschlingt des wilden
Augenblicks Gewalt.      -1.0
```

```
94    Vorspiel auf dem Theater        DIREKTOR       Die Masse könnt Ihr nur
durch Masse zwingen,      -1.0
103   Vorspiel auf dem Theater         DICHTER  Ihr fühlet nicht, wie schlecht
ein solches Han...      -1.0
116   Vorspiel auf dem Theater        DIREKTOR  Man eilt zerstreut zu uns, wie
zu den Maskenfe...      -1.0
...                          ...             ...
             ...       ...

#Statistische Kennzahlen zur Person Faus
print (df_faust.loc[df_faust["Sprecher"]=="FAUST"].describe())

          Stimmung
count  1262.000000
mean      0.033763
std       0.427679
min      -1.000000
25%       0.000000
50%       0.000000
75%       0.000000
max       1.000000

#und als Vergleich Gretchen

print (df_faust.loc[df_faust["Sprecher"]=="GRETCHEN"].describe())

         Stimmung
count  143.000000
mean    -0.057226
std      0.370512
min     -1.000000
25%      0.000000
50%      0.000000
75%      0.000000
max      1.000000

#wo tauchen die bekannten Verse auf
print (df_faust.loc[df_faust.Vers.str.contains("des Pudels Kern")])

               Kapitel Sprecher                            Vers  Stimmung
1340  Studierzimmer (I)    FAUST  Das also war des Pudels Kern!       0.0

#auch unter der Vewendung regulärer Ausdrücke
print (df_faust.loc[df_faust.Vers.str.contains(r"[bB]in weder [Ff]räulein")])

         Kapitel   Sprecher                                Vers  Stimmung
2706  Straße (I)  MARGARETE  Bin weder Fräulein, weder schön,       1.0

#Wie viele Zeilen fallen pro Sprecher an
print (df_faust.loc[df_faust.Sprecher=="FROSCH"].count())

Kapitel     51
Sprecher    51
Vers        51
Stimmung    51
dtype: int64

#und wie viele pro Kapitel
```

```
print (df_faust.loc[df_faust.Kapitel=="Kerker"].count())

Kapitel     224
Sprecher    224
Vers        224
Stimmung    224
dtype: int64
```

Für die Sentinent-Analyse konzentrieren wir uns im ersten Schritt auf die beiden Protagonisten Faust und Mephisto. Wir brauchen dafür nur die durchschnittliche Stimmung pro Kapitel.

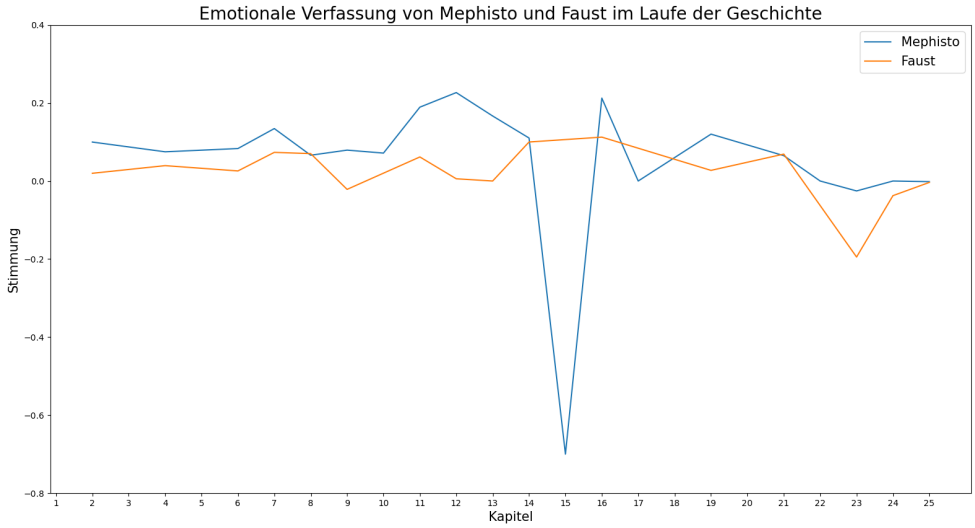
```
#Wir aggregieren Faust und Mephisto in jeweils ein eigenes Dataframe
df_sprecher_faust=df_faust[df_faust["Sprecher"]=="FAUST"]
df_sprecher_Mephisto=df_faust[df_faust["Sprecher"]=="MEPHISTOPHELES"]

#und pivotieren danach das Dataframe. Dadurch wird die durchschnittliche Stimmung
pro Kapitel gebildet

df_sprecher_faust_pivot=df_sprecher_faust.pivot_table(columns=
["Kapitel"],values=["Stimmung"])
df_sprecher_Mephisto_pivot=df_sprecher_Mephisto.pivot_table(columns=
["Kapitel"],values=["Stimmung"])

print (df_sprecher_faust_pivot)

Kapitel   Abend. Ein kleines reinliches Zimmer  Ein Gartenhäuschen    Garten
 Hexenküche.    Kerker  Marthens Garten ...
Stimmung                                0.070213                 0.0  0.005714
0.025806 -0.003571           0.1125 ...

#Danach entpivotieren wir wieder die Daten
df_sprecher_faust_melt=df_sprecher_faust_pivot.melt()
df_sprecher_Mephisto_melt=df_sprecher_Mephisto_pivot.melt()

#Es erscheinen in der Ausgabe nicht mehr alle Kapitel, da nicht jeder Protagonist
in jedem Kapitel auftritt
print (df_sprecher_faust_melt)

                                Kapitel     value
0   Abend. Ein kleines reinliches Zimmer  0.070213
1                     Ein Gartenhäuschen  0.000000
2                                 Garten  0.005714
3                           Hexenküche.  0.025806
4                                Kerker -0.003571
5                          Marthens Garten  0.112500
6                        Nacht, offen Feld -0.037500
7       Nacht. Straße vor Gretchens Türe  0.027273
8                        Prolog im Himmel.  0.019818
9                              Spaziergang -0.021429
10                              Straße (I)  0.073437
11                             Straße (II)  0.061667
12                       Studierzimmer (I)  0.039403
13                        Trüber Tag. Feld -0.194828
14                             Vor dem Tor  0.029577
15                          Wald und Höhle  0.100000
16                          Walpurgisnacht.  0.069017
```

Nun ist aber wieder ein Problem aufgetreten, dass wir von weiter oben schon kennen. Die Kapitel sind nicht mehr aufsteigend sortiert. Eine mögliche Lösung können wir uns ein paar Seiten vorher abschauen.

```
# Wir bauen uns aus der Liste der Kapitel ein Dictionary mit der Reihenfolge
dict_kapitel_sort=dict()
i=0
for kapitel in lst_kapitel:
    dict_kapitel_sort[kapitel]=i
    i+=1

print (dict_kapitel_sort)

{'Zueignung.\r': 0, 'Vorspiel auf dem Theater\r': 1, 'Prolog im Himmel.\r': 2,
'Vor dem Tor\r': 3, 'Studierzimmer (I)\r': 4, ...

# und fügen danach das Dictionary als neue Spalte hinzu und machen aus der Spalte
den neuen sortierten Index
df_sprecher_faust_melt["Neuer Index"]=[dict_kapitel_sort.get(kapitel) for
kapitel in df_sprecher_faust_melt["Kapitel"]]
df_sprecher_Mephisto_melt["Neuer Index"]=[dict_kapitel_sort.get(kapitel) for
kapitel in df_sprecher_Mephisto_melt["Kapitel"]]

df_sprecher_faust_melt=df_sprecher_faust_melt.set_index("Neuer
Index").sort_index()
df_sprecher_Mephisto_melt=df_sprecher_Mephisto_melt.set_index("Neuer
Index").sort_index()

print (df_sprecher_Mephisto_melt)

                                        Kapitel     value
Neuer Index
2                                Prolog im Himmel.  0.099848
4                                Studierzimmer (I)  0.074897
6                                      Hexenküche.  0.083223
7                                       Straße (I)  0.134420
8            Abend. Ein kleines reinliches Zimmer  0.066111
9                                       Spaziergang  0.079167
10                              Der Nachbarin Haus  0.071579
11                                     Straße (II)  0.189286
12                                          Garten  0.226667
13                              Ein Gartenhäuschen  0.166667
14                                  Wald und Höhle  0.110196
15                                Gretchens Stube. -0.700000
16                                 Marthens Garten  0.212500
17                                      Am Brunnen  0.000000
19               Nacht. Straße vor Gretchens Türe  0.120139
21                                  Walpurgisnacht.  0.065000
22                             Walpurgisnachtstraum  0.000000
23                                 Trüber Tag. Feld -0.025439
24                                Nacht, offen Feld  0.000000
25                                          Kerker -0.001667
```

Nun können wir uns daran machen, die Ergebnisse auszugeben. Wir erstellen dafür ein Plot, dass auf der X-Achse die Kapitel und auf der Y-Achse die Stimmung wiedergibt.

```
plt.plot(df_sprecher_Mephisto_melt.index,
df_sprecher_Mephisto_melt["value"],label="Mephisto")
plt.plot(df_sprecher_faust_melt.index,
df_sprecher_faust_melt["value"],label="Faust")
plt.yticks(np.arange(-1,0.5,0.2))
plt.xticks(np.arange(1,26))
plt.legend()
plt.xlabel("Kapitel")
plt.ylabel("Stimmung")
plt.title("Emotionale Verfassung von Mephisto und Faust im Laufe der
Geschichte")
plt.show()
```



Wenn ich exemplarisch noch Margarete in die Auswahl nehme, ist ersichtlich, welch schlechten Einfluss die beiden Hauptdarsteller auf die junge Frau haben.