Fehlwerte

by Woche 9

Wie in den vorigen Kapiteln setzen wir zunächst wieder Pandas Optionen und importieren unseren AirBnB Datensatz.

```
import numpy as np
import pandas as pd

pd.set_option('display.max_columns', 4)
pd.set_option('display.max_rows', 6)
pd.set_option('display.max_colwidth', 20)

csv_url = 'https://github.com/SchmidtPaul/ExampleData/raw/main/airbnb_open_
data/Airbnb_Open_Data.csv'
df = pd.read_csv(csv_url, dtype={25: str})

df
```

```
             id                  NAME  ...          house_rules license
0       1001254  Clean & quiet ap...  ...  Clean up and tre...     NaN
1       1002102  Skylit Midtown C...  ...  Pet friendly but...     NaN
2       1002403  THE VILLAGE OF H...  ...  I encourage you ...     NaN
3       1002755                  NaN  ...                 NaN     NaN
4       1003689  Entire Apt: Spac...  ...  Please no smokin...     NaN
...         ...                  ...  ...                 ...     ...
102594  6092437  Spare room in Wi...  ...  No Smoking No Pa...     NaN
102595  6092990  Best Location ne...  ...  House rules: Gue...     NaN
102596  6093542  Comfy, bright ro...  ...                 NaN     NaN
102597  6094094  Big Studio-One S...  ...                 NaN     NaN
102598  6094647  585 sf Luxury St...  ...                 NaN     NaN

[102599 rows x 26 columns]
```

In diesem Kapitel lernen wir, wie wir mit Fehlwerten umgehen können. Fehlwerte sind in der Praxis ein häufiges Problem. Sie können durch verschiedene Ursachen entstehen, wie z.B. fehlerhafte oder schlechtweg nicht-durchgeführte Messungen. Tatsächlich gibt es beim Umgang mit Fehlwerten in Python einige Besonderheiten, die nicht unbedingt intuitiv sind. Gleichzeitig treten sie in der Praxis sehr häufig auf und können bei der Analyse von Daten zu Problemen führen.

Fehlwerte in Python/Numpy/Pandas

In Python gibt es unterschiedliche Weisen, um Fehlwerte zu repräsentieren. Dabei stehen die Kürzel **NaN** für *Not a Number* und **NA** für *Not Available*. Mit den uns bekannten Module können euch folgende Werte begegnen:

- `None`: Dies ist ein spezielles Objekt in Python, das oft verwendet wird, um das Fehlen eines Werts zu signalisieren.
- `np.nan`: Dies steht für *Not a Number* und werden verwendet, um fehlende/undefinierte Gleitkommazahlen darzustellen.
- `pd.NA`: Dies ist ein spezieller Wert aus der pandas-Bibliothek, der für fehlende Daten über verschiedene Datentypen hinweg verwendet wird, einschließlich Integer und Strings.
- `pd.NaT`: Dies steht für *Not a Time* und wird in pandas für fehlende oder undefinierte Zeitstempel bei Datums- und Zeitdatentypen verwendet.

Ein wesentlicher Unterschied zwischen den ersten beiden ist, dass `None` ein allgemeines Python-Objekt ohne spezifischen Datentyp ist, während `np.nan` speziell für numerische (Gleitkomma-)Berechnungen in numpy gedacht ist. `np.nan` ist also (wie alles im numpy Paket) ideal für Situationen, in denen Genauigkeit und der Umgang mit Zahlen in numerischen Berechnungen kritisch sind.

```
type(None)
```

```
<class 'NoneType'>
```

```
type(np.nan)
```

```
<class 'float'>
```

In praktischen Anwendungen führt das Hinzufügen von `np.nan` zu einem Array oder einer Serie dazu, dass diese, wenn sie nicht explizit als Gleitkommazahl definiert ist, als Array vom Typ `object` behandelt wird. Dies liegt daran, dass `np.nan` als Gleitkommazahl definiert ist und somit eine Typumwandlung des gesamten Arrays erzwingt, um Datentypinkonsistenzen zu vermeiden.

Erzeugen wir einen Array mit Ganzzahlen und einem `None` Wert, so wird der gesamte Array zu einem `object`-Datentyp (also keinem numerischen Datentyp) konvertiert, da `None` keine Zahl ist und auch nicht automatisch zu einer Zahl konvertiert wird. Erzeugen wir hingegen einen Array mit Ganzzahlen und einem `np.nan` Wert, so wird der gesamte Array zu einem `float64`-Datentyp (nicht zu `integer64`) konvertiert, da `np.nan` wie gesagt vom Typ Float/Gleitkommazahl ist.

```
x = np.array([1, None, 3])
x
x.dtype
```

```
array([1, None, 3], dtype=object)
dtype('O')
```

```
x = np.array([1, np.nan, 3])
x
x.dtype
```

```
array([ 1., nan,  3.])
dtype('float64')
```

Der Wert `pd.NA` aus dem Pandas Modul ist eine relativ neue Ergänzung und soll die Verwendung von `None` und `np.nan` vereinheitlichen, da es sowohl für numerische als auch für nicht-numerische Datentypen geeignet ist. `pd.NA` ist eine Art universeller Fehlwert, welcher als `<NA>` angezeigt wird un in allen Datentypen verwendet werden kann.

```
pd.Series(
  [1, pd.NA, 3],
  dtype='Int64'
)
```

```
0       1
1    <NA>
2       3
dtype: Int64
```

```
pd.Series(
  [1.5, pd.NA, 3.5],
  dtype='Float64'
)
```

```
0     1.5
1    <NA>
2     3.5
dtype: Float64
```

```
pd.Series(
  ["Eins", pd.NA, "Drei"],
  dtype='string'
)
```

```
0    Eins
1    <NA>
2    Drei
dtype: string
```

Es ist übrigens auch ohne weiteres möglich Listen, welche `None` und/oder `np.nan` enthalten, in eine Pandas Series oder einen DataFrame zu konvertieren. Pandas wird dabei versuchen automatisch die Datentypen der Spalten erkennen und ggf. die Fehlwerte konvertieren.

```
pd.Series(
  [1, None, "Drei"]

)
```

```
0       1
1    None
2    Drei
dtype: object
```

```
pd.Series(
  [1, None, np.nan]

)
```

```
0    1.0
1    NaN
2    NaN
dtype: float64
```

```
pd.Series(
  [1, None, np.nan],
  dtype='Int64'
)
```

```
0       1
1    <NA>
2    <NA>
dtype: Int64
```

```
pd.Series(
  ["One", None, np.nan],
  dtype='string'
)
```

```
0     One
1    <NA>
2    <NA>
dtype: string
```

Letztendlich läuft es darauf hinaus, dass in DataFrame-Spalten mit Gleitzahl Datentyp `np.nan` verwendet werden sollte, in Spalten mit Datums-/Zeitangaben `pd.NaT` und in Spalten mit nicht-numerischen Datentypen `pd.NA` oder `None`.

Fehlwerte erkennen

In der Praxis ist es wichtig, Fehlwerte zu erkennen, um sie korrekt behandeln zu können. Ein einfaches Beispiel wäre es die Daten so zu filtern, dass nur die Zeilen mit Fehlwerten in einer bestimmten Spalte übrig bleiben. Tatsächlich ist eine dafür notwendige Prüfung aber nicht wie vielleicht erwartet mit dem `==` Operator möglich, sondern muss mit der Methode `isna()` durchgeführt werden.

```
pd.Series([1, None, np.nan, pd.NA]) == pd.NA
```

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

```
pd.Series([1, None, np.nan, pd.NA]).isna()
```

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

Die Methode `isna()` gibt für jeden Wert in der Serie `True` zurück, wenn der Wert ein Fehlwert ist, und `False`, wenn der Wert ein gültiger Wert ist. Das Gegenstück dazu ist die Methode `notna()`. So könnten wir also alle Zeilen aus unserem AirBnB Datensatz filtern, in denen die Spalte `host_name` fehlende Werte enthält. Allerdings erzeugen wir uns vorerst wieder einen Teildatensatz, um die Ausgabe übersichtlich zu halten.

```
df2 = df.loc[[1, 100, 2, 210, 3, 102049], ['host name', 'price']] # Wähle
Zeilen/Spalten
df2.reset_index(drop=True, inplace=True)     # Setze Index zurück
df2 = df2.assign(
  host = df2['host name'].astype('string'),  # Konvertiere in String
  price = df2['price'].str[1:].astype(float) # Entferne $ und konvertiere in
Float
)
df2.drop(columns='host name', inplace=True)  # Lösche Spalte
df2
```

```
   price   host
0  142.0  Jenna
1  739.0   <NA>
2  620.0  Elise
3    NaN   Ryan
4  368.0  Garry
5    NaN   <NA>
```

```
df2[df2['host'].isna()]
```

```
   price  host
1  739.0  <NA>
5    NaN  <NA>
```

```
df2[df2['host'].notna()]
```

```
   price   host
0  142.0  Jenna
2  620.0  Elise
3    NaN   Ryan
4  368.0  Garry
```

Es muss übrigens auch klar sein, dass die Fehlwerte beim Filtern so auch immer separat gehandhabt werden müssen. Das wird deutlich, wenn wir die Daten hier mal in vermeintlich zwei Teile trennen, indem wir einmal nach `host == Jenna` und einmal nach `host != Jenna` filtern. Man mag erwarten, dass jede Zeile entweder in der ersten oder in

der zweiten Bedingung enthalten ist, da sie ja entweder `Jenna` als `host` haben oder nicht. Wie wir aber sehen, schließt keine der beiden Bedinungen die Fehlwerte mit ein. Wir erhalten so also quasi nur zwei von drei Teilen der Daten, wobei der dritte Teil der mit den Fehlwerten ist:

```
# Teil 1
df2[df2['host'] == 'Jenna']
```

```
   price   host
0  142.0  Jenna
```

```
# Teil 2
df2[df2['host'] != 'Jenna']
```

```
   price   host
2  620.0  Elise
3    NaN   Ryan
4  368.0  Garry
```

```
# Teil 3
df2[df2['host'].isna()]
```

```
   price  host
1  739.0  <NA>
5    NaN  <NA>
```

Falls also das Ziel sein sollte alle Zeilen (auch die mit Fehlwerten) außer denen zu behalten, die nicht `Jenna` als `host` haben, dann müsste die Bedingung die Fehlwerte explizit mit einschließen. Das geht z.B. so:

```
behalten = (df2['host'] != 'Jenna') | df2['host'].isna()
df2[behalten]
```

```
   price   host
1  739.0   <NA>
2  620.0  Elise
3    NaN   Ryan
4  368.0  Garry
5    NaN   <NA>
```

> **Was ist mit `isnull()`?**
>
> Neben der hier gezeigten `.isna()` Methode gibt es auch die Methode `.isnull()`. Beide sind in Pandas identisch und können synonym verwendet werden. Gleiches gilt für die Methoden `.notna()` und `.notnull()`. In diesem Blogpost gibt es etwas mehr Details dazu warum das so ist.

Fehlwerte entfernen

Eine einfache Möglichkeit, um mit Fehlwerten umzugehen, ist es, sie einfach zu entfernen. Da wir gerade `.notna()` kennengelernt haben, könnten wir damit schlichtweg filtern. Allerdings gibt es auch eine Methode `dropna()`, die alle Zeilen mit Fehlwerten entfernt und dabei einfacher und flexibler ist. Das Argument `how=` kann entweder `'any'` sein (=Standardwert), um Zeilen mit mindestens einem Fehlwert zu entfernen, oder `'all'`, um nur Zeilen zu entfernen, die ausschließlich aus Fehlwerten bestehen. Mit dem Argument `subset=` können wir auch explizit angeben, in welchen Spalten nach Fehlwerten gesucht werden soll.

```
df2.dropna(how='any')
```

```
   price   host
0  142.0  Jenna
2  620.0  Elise
4  368.0  Garry
```

```
df2.dropna(how='all')
```

```
   price   host
0  142.0  Jenna
1  739.0   <NA>
2  620.0  Elise
3    NaN   Ryan
4  368.0  Garry
```

```
df2.dropna(subset=['host'])
```

```
   price   host
0  142.0  Jenna
2  620.0  Elise
```

```
3    NaN   Ryan
4  368.0  Garry
```

Übrigens hat auch diese Methode ein `axis=` Argument. Während also standardmäßig mit `axis=0` die Zeilen entfernt werden, können wir mit `axis=1` auch Spalten entfernen, die Fehlwerte enthalten. Schließlich gibt es sogar ein `thresh=` Argument, mit dem wir angeben können, wie viele Fehlwerte in einer Zeile/Spalte maximal enthalten sein dürfen, damit sie nicht entfernt wird. So können wir demnach auch zwischen den Extremen `how='any'` und `how='all'` noch eine Zwischenlösung finden.

Fehlwerte ersetzen

Eine andere Möglichkeit, mit Fehlwerten umzugehen, ist es, sie durch einen anderen Wert zu ersetzen. Dafür gibt es die Methode `fillna()`, die den Fehlwert durch einen anderen Wert ersetzt. Das Argument `value=` gibt an, durch welchen Wert ersetzt werden soll. Gibt man nur einen Wert an, so wird jeder Fehlwert in der Serie/Spalte/Zeile bzw. dem gesamten DataFrame durch diesen Wert ersetzt. Gibt man ein Dictionary an, so können auch unterschiedliche Werte für unterschiedliche Spalten angegeben werden.

```
df2.fillna(value='Unbekannt')



#
```

```
       price       host
0      142.0      Jenna
1      739.0  Unbekannt
2      620.0      Elise
3  Unbekannt       Ryan
4      368.0      Garry
5  Unbekannt  Unbekannt
```

```
na_ersatz = {'host': 'Unbekannt'}
df2.fillna(value=na_ersatz)



#
```

```
   price       host
0  142.0      Jenna
1  739.0  Unbekannt
2  620.0      Elise
```

```
3    NaN       Ryan
4  368.0      Garry
5    NaN  Unbekannt
```

```
na_ersatz = {
  'host': 'Unbekannt',
  'price': 0
}
df2.fillna(value=na_ersatz)
```

```
   price       host
0  142.0      Jenna
1  739.0  Unbekannt
2  620.0      Elise
3    0.0       Ryan
4  368.0      Garry
5    0.0  Unbekannt
```

Anstatt Werte vorzugeben, können wir auch die Methode `ffill()` (*frontfill*) oder `bfill()` (*backfill*) verwenden, um Fehlwerte durch den vorherigen bzw. nächsten Wert zu ersetzen. Diese Methoden können entweder auf den gesamten DataFrame angewendet werden oder auf eine spezifische Spalte.

```
df2.ffill()
```

```
   price   host
0  142.0  Jenna
1  739.0  Jenna
2  620.0  Elise
3  620.0   Ryan
4  368.0  Garry
5  368.0  Garry
```

```
df2.bfill()
```

```
   price   host
0  142.0  Jenna
1  739.0  Elise
2  620.0  Elise
3  368.0   Ryan
4  368.0  Garry
5    NaN   <NA>
```

```
df2['host'].ffill()
```

```
0    Jenna
1    Jenna
2    Elise
3     Ryan
4    Garry
5    Garry
Name: host, dtype: string
```

Praxistipp

Die oben genannten Funktionen ersetzen Fehlwerte, die bereits korrekt als solche in den Daten definiert sind. In der Praxis kommt es aber regelmäßig vor, dass statt Fehlwerten leere Strings oder Strings wie `'NA'` oder `'missing'` verwendet werden. Diese werden von den Funktionen nicht als Fehlwerte erkannt und daher auch nicht ersetzt. Um solche Werte zu erkennen und zu ersetzen, können wir die Methode `replace()` verwenden. Diese Methode ersetzt Werte in einer Serie/Spalte/Zeile bzw. dem gesamten DataFrame durch andere Werte. Das erste Argument `to_replace=` gibt an, welcher Wert ersetzt werden soll, und das zweite Argument `value=` gibt an, durch welchen Wert ersetzt werden soll.

```
dat = ['Value1', 'Value2', '', 'NA', 'missing', None, np.nan, pd.NA, 'Value3']
auchfehlwerte = ['', 'NA', 'missing']
```

```
#
pd.Series(dat)


#
```

```
0     Value1
1     Value2
2
3         NA
4    missing
5       None
6        NaN
7       <NA>
8     Value3
dtype: object
```

```
(
  pd.Series(dat)
  .fillna(value=pd.NA)
)
#
```

```
0     Value1
1     Value2
2
3         NA
4    missing
5       <NA>
6       <NA>
7       <NA>
8     Value3
dtype: object
```

```
(
  pd.Series(dat).
  replace(auchfehlwerte, pd.NA).
  fillna(value=pd.NA)
)
```

```
0    Value1
1    Value2
2      <NA>
3      <NA>
4      <NA>
5      <NA>
6      <NA>
7      <NA>
8    Value3
dtype: object
```

Mehr über Fehlwerte erfahren

Prinzipiell wissen wir nun, dass wir Fehlwerte erkennen bzw. filtern können. Wir wissen auch, wie wir sie entfernen oder ersetzen können. In der Regel ist es aber auch wichtig zu wissen, wie viele Fehlwerte in den Daten enthalten sind und wo sie sich befinden. Um dies zu erfahren kann beispielsweise schlichtweg die `.info()` Methode verwendet werden, da diese neben `Dtype` auch den `Non-Null Count` pro Spalte ausgibt. Alternativ können wir auch die `.isna()` Methode verwenden, um zu zählen, wie viele Fehlwerte in jeder Spalte enthalten sind. Versucht man nämlich die `.sum()` Methode auf boolean Werte anzuwenden, so wird `True` als `1` und `False` als `0` interpretiert. Da die `.isna()`

Methode `True` für Fehlwerte und `False` für gültige Werte zurückgibt, können wir so die Anzahl der Fehlwerte in jeder Spalte zählen.

```
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   price   4 non-null      float64
 1   host    4 non-null      string
dtypes: float64(1), string(1)
memory usage: 228.0 bytes
```

```
pd.Series([True, False, True]).sum()
```

```
np.int64(2)
```

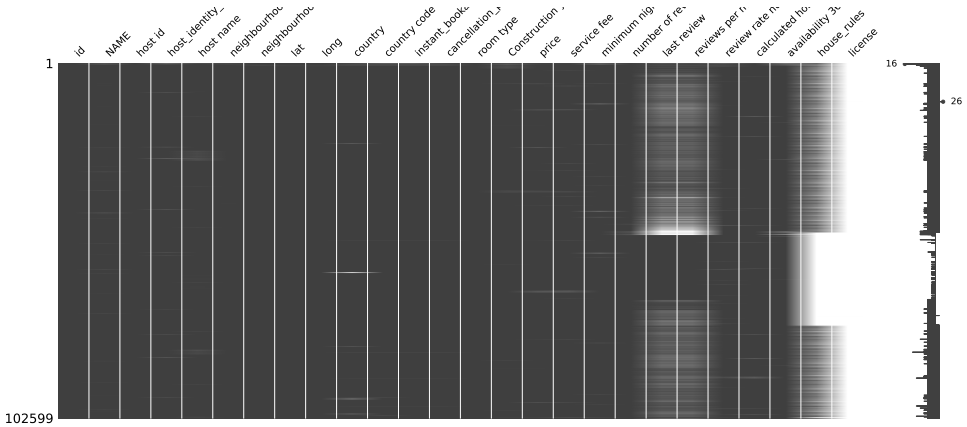
```
df2.isna().sum()
```

```
price    2
host     2
dtype: int64
```

Wer es noch genauer wissen will, kann auch das Modul `missingno` verwenden, welches speziell für die Visualisierung von Fehlwerten entwickelt wurde. Dieses Modul muss allerdings erst installiert werden (siehe Kapitel *4.A Module installieren*).

So kann beispielsweise eine sogenannte *nullity matrix* geplottet werden, die die Verteilung der Fehlwerte in einem DataFrame visualisiert. Schwarz bedeutet dabei, dass ein Wert vorhanden ist und weiß bedeutet, dass ein Wert fehlt. Auf diese Weise bekommen wir einen schnellen Überblick über die Verteilung der Fehlwerte in allen Spalten unseres großen AirBnB Datensatzes. Das Modul bietet noch weitere solcher Visualisierungen wie `msno.bar()` und `msno.heatmap()`, die wir hier aber nicht weiter behandeln.

```
import missingno as msno
msno.matrix(df)
```



Wie gehen Funktionen mit Fehlwerten um?

Es soll an dieser Stelle einmal explizit darauf hingewiesen werden, dass Pandas Fehlwerte in der Regel automatisch ignoriert. Das bedeutet, dass die meisten Funktionen, die auf Pandas-Objekten angewendet werden, standardmäßig Fehlwerte ausschließen. Das ist in den meisten Fällen auch sinnvoll, da Fehlwerte oft nicht sinnvoll interpretiert werden können und die meisten Funktionen daher nicht sinnvoll mit ihnen umgehen können. Das bedeutet aber auch, dass ggf. unbemerkt Fehlwerte in den Daten enthalten sind, die zu unerwarteten Ergebnissen führen können. Schließlich erkennt man beim Betrachten der folgenden zwei Mittelwerte nicht, dass in der zweiten Serie ein Fehlwert enthalten ist. Um sicherzustellen, dass Fehlwerte nicht ignoriert werden, können wir die Methode `skipna=False` verwenden. Diese Methode wird in vielen Pandas-Funktionen unterstützt und sorgt dafür, dass Fehlwerte nicht ignoriert werden. Das bedeutet, dass die Funktionen `np.mean()` und `np.sum()` dann `np.nan` zurückgeben, wenn Fehlwerte in den Daten enthalten sind.

```
x = pd.Series([11, 12, 13])
x.mean()
```

```
np.float64(12.0)
```

```
y = pd.Series([11, 12, np.nan])
y.mean()
```

```
np.float64(11.5)
```

```
y = pd.Series([11, 12, np.nan])
y.mean(skipna=False)
```

```
np.float64(nan)
```

> **Bei Numpy ist das anders**
>
> Bei Numpy ist das Verhalten anders. Hier wird z.B. der Mittelwert eines Arrays, der einen Fehlwert enthält, immer `np.nan` sein. Das bedeutet, dass Numpy-Funktionen standardmäßig Fehlwerte nicht ignorieren. Das ist ein wichtiger Unterschied zu Pandas. Bei `np.mean(np.array([11, 12, np.nan]))` wird also immer `np.nan` zurückgegeben. Es gibt auch kein Argument wie `skipna=False`, um dieses Verhalten zu ändern. Stattdessen gibt es die separate Funktion `np.nanmean()`, die explizit Fehlwerte ignoriert.

> **Weitere Ressourcen**
>
> - Pandas Crashkurs - fehlende Daten / Missing Values - Video 5/8 kompletter Kurs auf deutsch/german

Übungen

Berechne jeweils den Durchschnittspreis (Spalte `price`) aller AirBnB Unterkünfte in `df2` für folgende Szenarien und gib sie gerundet auf keine Nachkommastelle an.

- ___ $ ist der Durchschnittspreis.
- ___ $ ist der Durchschnittspreis, nachdem alle Zeilen gelöscht wurden, in denen der Hostname fehlt.
- ___ $ ist der Durchschnittspreis, nachdem alle Fehlwerte in der Spalte 'price' durch 0 ersetzt wurden.
- ___ $ ist der Durchschnittspreis, nachdem alle Zeilen gelöscht wurden, in denen der Hostname fehlt und alle Fehlwerte in der Spalte 'price' durch 0 ersetzt wurden.

Welche Spalte in unserem AirBnB Datensatz enthält die meisten Fehlwerte?

- Spaltenname: _____

Schau dir folgenden Python Code an und versuche durch Nachschlagen und Ausprobieren auch die angewandten Methoden zu verstehen, die wir in diesem Kurs noch nicht besprochen haben. Welche der folgenden Fragen wird mit diesem Code beantwortet?

```
x = df.isna().sum().to_frame()
x[x[0] == 0].shape[0]
```

- (A) Wie viele Spalten haben mindestens einen Fehlwert?
- (B) Wie viele Spalten haben keinen Fehlwert?
- (C) Wie viele Spalten haben genau einen Fehlwert?
- (D) Wie viele Spalten haben ausschließlich Fehlwerte?