Zeilen filtern

by Woche 8

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

```
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
...         ...                  ...  ...                 ...     ...
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 Abschnitt konzentrieren wir uns auf das Auswählen von Zeilen. Zuvor sollten wir jedoch die verwendeten Begriffe klären, insbesondere den Unterschied zwischen “Zeilen filtern” und “Spalten selektieren”. Die Unterscheidung ist nicht immer eindeutig, da man eigentlich auch Zeilen selektieren oder Spalten herausfiltern kann. Trotzdem hat sich durch spezifische Funktionen in Programmiersprachen wie z.B. SQL (`SELECT`/`FILTER`) oder R (`select()`/`filter()`) ein gewisser Zusammenhang zwischen den erstgenannten Begriffspaaren etabliert. Die Begriffe lassen sich jedoch, wenn überhaupt, nur grob voneinander abgrenzen: “filtern” impliziert in der Regel, dass eine Bedingung erfüllt sein muss, während “selektieren” eher darauf hindeutet, dass eine explizite, bekannte Auswahl getroffen wird. In der Praxis muss man Zeilen häufiger filtern und Spalten eher selektieren.

Zeilen selektieren

Der Vollständigkeit halber wollen wir also zuerst doch kurz darauf eingehen, wie man auch Zeilen einfach selektieren kann. Speziell in Python und Pandas ist dies nämlich dem Spalten selektieren prinzipiell ähnlich, da die Indices von Zeilen in gewisser Weise gehandhabt werden wie die Indices von Spalten. Für die direkte Vergleichbarkeit zum vorangegangen Kapitel zur Spaltenselektion sind hier dieselben vier Stichpunkte analog für Zeilen:

Für die ersten beiden Punkte gibt es in Pandas also keine direkte Entsprechung für Zeilen. Ebenfalls gilt es wieder zu unterscheiden ob man beim Selektieren einer einzelnen Zeile eine Series oder einen Dataframe mit einer Zeile erhält.

Eine Zeile als Series

```
df.loc[0]
df.iloc[0]
```

```
id                              1001254
NAME                Clean & quiet ap...
host id                        80014485718
                               ...
availability 365                    286.0
house_rules         Clean up and tre...
license                               NaN
Name: 0, Length: 26, dtype: object
```

Eine Zeile als DataFrame

```
df.loc[[0]]
df.iloc[[0]]
```

```
        id                 NAME  ...          house_rules license
0  1001254  Clean & quiet ap...  ...  Clean up and tre...     NaN

[1 rows x 26 columns]
```

Mehrere Zeilen als DataFrame

```
df.loc[[0, 1]]
df.iloc[[0, 1]]
```

```
        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 rows x 26 columns]
```

Was für etwas Verwirrung sorgen könnte ist die Tatsache, dass hier sowohl in `.loc[]` als auch in `.iloc[]` die Zeilenindizes in eckigen Klammern stehen. Der Unterschied sollte ja sein, dass `.loc[]` die Zeilen anhand ihres Namens auswählt, während `.iloc[]` die Zeilen anhand ihres Index auswählt. Da wir standardmäßig und so eben auch hier aber keine expliziten Zeilennamen haben, ist also der Zeilenname gleich dem Index und wir können die beiden Methoden hier gleich verwenden.

Zeilenindices bearbeiten

Jetzt ist also ein guter Zeitpunkt um mal die Zeilenindices zu verändern und ihnen tatsächlich Namen/Label zu geben. Das geht z.B. mit der Methode `.set_index()`. Hierbei wird eine im Dataframe vorhandene Spalte als Index festgelegt, also zum Index konvertiert, sodass die Zeilen dann anhand der Einträge in dieser Spalte ausgewählt werden können. In unserem Fall bietet sich die Spalte `id` an, da diese eindeutig und somit als Index geeignet ist. Die Spalte `id` wird also anstelle der Standard-Indices als Index festgelegt und ist dann auch nicht mehr als “normale Spalte” im Dataframe.

```
df.set_index('id', inplace=True)
df
```

```
                        NAME      host id  ...          house_rules license
id                                         ...
1001254  Clean & quiet ap...  80014485718  ...  Clean up and tre...     NaN
1002102  Skylit Midtown C...  52335172823  ...  Pet friendly but...     NaN
1002403  THE VILLAGE OF H...  78829239556  ...  I encourage you ...     NaN
...                      ...          ...  ...                 ...     ...
6093542  Comfy, bright ro...  69050334417  ...                 NaN     NaN
6094094  Big Studio-One S...  11160591270  ...                 NaN     NaN
6094647  585 sf Luxury St...  68170633372  ...                 NaN     NaN

[102599 rows x 25 columns]
```

Nun wäre die erste Zeile zwar weiterhin mit `df.iloc[[0]]`, aber eben mit `df.loc[[1001254]]` zu selektieren:

```
df.loc[[1001254]]
```

```
                        NAME      host id  ...          house_rules license
id                                         ...
1001254  Clean & quiet ap...  80014485718  ...  Clean up and tre...     NaN

[1 rows x 25 columns]
```

Dabei kann der Index zum Einen auch ein String sein und zum Anderen mit `index_col=` auch direkt beim Import festgelegt werden.

```
df = pd.read_csv(
  csv_url,
  dtype={25: str},
  index_col='NAME'
  )

df
```

```
                           id      host id  ...          house_rules license
NAME                                        ...
Clean & quiet apt...  1001254  80014485718  ...  Clean up and tre...     NaN
Skylit Midtown Ca...  1002102  52335172823  ...  Pet friendly but...     NaN
THE VILLAGE OF HA...  1002403  78829239556  ...  I encourage you ...     NaN
...                       ...          ...  ...                 ...     ...
Comfy, bright roo...  6093542  69050334417  ...                 NaN     NaN
Big Studio-One St...  6094094  11160591270  ...                 NaN     NaN
585 sf Luxury Studio  6094647  68170633372  ...                 NaN     NaN

[102599 rows x 25 columns]
```

```
df.loc[['Clean & quiet apt home by the park']]
```

```
                           id      host id  ...          house_rules license
NAME                                        ...
Clean & quiet apt...  1001254  80014485718  ...  Clean up and tre...     NaN

[1 rows x 25 columns]
```

```
df.iloc[[0]] # alternativ weiterhin mit IndexNr
```

```
                           id      host id  ...          house_rules license
NAME                                        ...
Clean & quiet apt...  1001254  80014485718  ...  Clean up and tre...     NaN
```

```
[1 rows x 25 columns]
```

Für den Moment setzen wir den Index mit `.reset_index()` wieder zurück, um uns nun auf das Filtern zu konzentrieren.

```
df = df.reset_index()
df
```

```
                        NAME       id  ...          house_rules license
0       Clean & quiet ap...  1001254  ...  Clean up and tre...     NaN
1       Skylit Midtown C...  1002102  ...  Pet friendly but...     NaN
2       THE VILLAGE OF H...  1002403  ...  I encourage you ...     NaN
...                     ...      ...  ...                 ...     ...
102596  Comfy, bright ro...  6093542  ...                 NaN     NaN
102597  Big Studio-One S...  6094094  ...                 NaN     NaN
102598  585 sf Luxury St...  6094647  ...                 NaN     NaN

[102599 rows x 26 columns]
```

Zeilen filtern

Nun wollen wir uns dem eigentlichen Thema widmen: dem Filtern. Hierbei wird eine Bedingung definiert, die für jede Zeile überprüft wird. Wenn die Bedingung erfüllt ist, wird die Zeile beibehalten, ansonsten verworfen.

An dieser Stelle ist es sinnvoll, dass wir uns vorest einen übersichtlichen Teildatensatz erzeugen um mit diesem zu arbeiten. Dazu selektieren wir folgende 6 Zeilen und 4 Spalten:

```
df2 = df.loc[0:5, ['room type', 'minimum nights', 'Construction year']]
df2
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
1  Entire home/apt            30.0             2007.0
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

Wie man prüft, also einen boolschen Wert `True` oder `False` erhält, haben wir schon früh in Kapitel *2.2 Datentypen* gelernt. So können wir prüfen ob eine Zahl z.B. gleich 2005 ist, indem wir den == Operator verwenden. Praktischerweise, können wir mit Pandas auch direkt eine ganze Spalte auf diese Weise prüfen. Es wird also nicht geprüft ob eine

bestimmte Spalte dasselbe ist wie 2005, sondern ob jede einzelne Zelle in der Spalte gleich 2005 ist. Das Ergebnis ist dann entsprechend eine Spalte mit `True` und `False` Werten.

```
x = 2005 # definiere x als 2005
x == 2005 # prüfe ob x gleich 2005
```

```
True
```

```
df2['Construction year'] == 2005
```

```
0    False
1    False
2     True
3     True
4    False
5    False
Name: Construction year, dtype: bool
```

Diese Spalte können wir dann übergeben und behalten nur die Zeilen, die `True` sind. Die Daten wurden also entsprechend der Bedinung gefiltert. Dabei kann die Bedingung entweder direkt übergeben oder vorher in einer Variable gespeichert werden. Letzteres wird ggf. als übersichtlicher empfunden und ergibts spätestens dann Sinn, wenn die Bedingung komplex ist und/oder mehrfach verwendet wird.

```
#
df2[df2['Construction year'] == 2005]
```

```
         room type  minimum nights  Construction year
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
```

```
gebaut_2005 = df2['Construction year'] == 2005
df2[gebaut_2005]
```

```
         room type  minimum nights  Construction year
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
```

Bedingung umkehren

Manchmal ist es nötig oder intuitiver eine Bedingung umzukehren. Dies kann mit dem ~ Operator erreicht werden. Dabei werden dann dementsprechend alle `True` Werte zu `False` und umgekehrt.

```
gebaut_2005
```

```
0    False
1    False
2     True
3     True
4    False
5    False
Name: Construction year, dtype: bool
```

```
~gebaut_2005
```

```
0     True
1     True
2    False
3    False
4     True
5     True
Name: Construction year, dtype: bool
```

Mehrere Bedingungen

Mehrere Bedingungen können mit den logischen Operatoren `&` (und) und `|` (oder) verknüpft werden. Dabei ist es wichtig, die Bedingungen in Klammern zu setzen, um die Reihenfolge der Operationen zu steuern.

& (und)

Wollen wir beispielsweise alle Zeilen behalten, die den Zimmertyp Ganze Wohnung/ Haus haben **und** weniger als 15 Nächte Mindestaufenthalt aufweisen, so können wir dies wie folgt umsetzen:

```
behalten = (df2['room type'] == 'Entire home/apt') & (df2['minimum nights'] < 15)
df2[behalten]
```

```
         room type  minimum nights  Construction year
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

| (oder)

Wollen wir stattdessen alle Zeilen behalten, die den Zimmertyp Privatzimmer **oder** Baujahr 2005 haben, so können wir dies so umsetzen:

```
behalten = (df2['room type'] == 'Private room') | (df2['Construction year'] ==
2005)
df2[behalten]
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
```

Beachte, dass | in Python OR und nicht XOR bedeutet. XOR bedeutet, dass wirklich nur eine der beiden Bedingungen erfüllt sein darf. In diesem Fall wäre die mittlere, gefilterte Zeile dementsprechend nicht enthalten, da sie sowohl den Zimmertyp Privatzimmer als auch das Baujahr 2005 hat. Der XOR Operator ist ^.

Klammern

Schließlich muss klar sein, dass die Klammern in der Bedingung nicht nur für die Lesbarkeit, sondern auch für die korrekte Ausführung notwendig sind. Um dies zu demonstrieren wollen wir folgende drei Bedingungen vorbereiten und sie dann mal in verschiedenen Kombinationen ausführen, wobei die Operatoren & und | aber uvnerändert bleiben:

```
bed1 = df2['Construction year'] > 2005
bed2 = df2['room type'] == 'Private room'
bed3 = df2['minimum nights'] > 15
```

```
df2[bed1 & bed2 | bed3]
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
```

```
1  Entire home/apt            30.0             2007.0
3  Entire home/apt            30.0             2005.0
```

```
df2[bed1 & (bed2 | bed3)]
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
1  Entire home/apt            30.0             2007.0
```

Es zeigt sich, dass die unterschiedliche Setzung von Klammern auch zu unterschiedlich gefilterten Ergebnissen führen. Ohne Klammern wird zuerst `bed1 & bed2` berechnet und dann mit `bed3` verknüpft. Mit Klammern wird zuerst `bed2 | bed3` berechnet und dann mit `bed1` verknüpft.

Die `.isin()` Methode

Eine weitere Möglichkeit, mehrere Bedingungen zu verknüpfen, ist die `.isin()` Methode. Diese Methode prüft, ob ein Wert in einer Liste von Werten enthalten ist. Sie prüft also in gewisser Hinsicht mehrere Bedingungen gleichzeitig. Hier der Vergleich um mit und ohne `.isin()` zu filtern ob das Baujahr 2020, 2009 oder 2005 ist:

```
ist_2020 = df2['Construction year'] == 2020
ist_2009 = df2['Construction year'] == 2009
ist_2005 = df2['Construction year'] == 2005

df2[ist_2020 | ist_2009 | ist_2005]
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
4  Entire home/apt            10.0             2009.0
```

```
ok_jahre = [2020, 2009, 2005]
ist_ok = df2['Construction year'].isin(ok_jahre)


df2[ist_ok]
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
```

```
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
4  Entire home/apt            10.0             2009.0
```

Die `.between()` Methode

Die `.between()` Methode ist eine weitere Möglichkeit, um Bedingungen zu formulieren. Sie prüft, ob ein Wert zwischen zwei anderen Werten liegt. Hier der Vergleich um mit und ohne `.between()` zu filtern ob das Baujahr zwischen 2006 und 2015 liegt:

```
ist_nach_2008 = df2['Construction year'] > 2006
ist_vor_2015 = df2['Construction year'] < 2015

df2[ist_nach_2008 & ist_vor_2015]
```

```
         room type  minimum nights  Construction year
1  Entire home/apt            30.0             2007.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

```
zwi = df2['Construction year'].between(2006, 2015)


df2[zwi]
```

```
         room type  minimum nights  Construction year
1  Entire home/apt            30.0             2007.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

String-Methoden

Wie schon im letzten Kapitel zur Selektion von Spalten anhand ihres Namens, können natürlich auch hier beim Filtern String-Methoden
wie `.str.contains()`, `.str.startswith()`, `.str.endswith()` zum Filtern von Text/String-Spalten eingesetzt werden. Als Beispiel könnten wir wie folgt feststellen, dass 800 unserer >100,000 Bezeichnung von AirBnB Unterkünften mit “Clean” beginnen:

```
beginnt_mit_Clean = df['NAME'].str.startswith('Clean', na=False)
df[beginnt_mit_Clean]
```

```
                        NAME        id  ...          house_rules license
0       Clean & quiet ap...   1001254  ...  Clean up and tre...     NaN
37      Clean and Quiet ...   1021771  ...  NO Shoes in the ...     NaN
180     Clean and Cozy H...   1100750  ...  We live on the p...     NaN
...                     ...       ...  ...                 ...     ...
102384  Clean cozy overn...  20361660  ...  We have a no sho...     NaN
102468  Clean, Cozy Home...  20408053  ...  All guests are e...     NaN
102564  Clean & Cozy- Pr...   6075868  ...                 NaN     NaN

[800 rows x 26 columns]
```

> **i** na=False
>
> Wie schon im letzten Kapitel erwähnt, gehen wir auf Fehlwerte erst in einem folgenden Kapitel ein. Dennoch muss in der obigen Funktion `.str.startswith()` der Parameter `na=False` gesetzt werden, um Fehlwerte zu ignorieren. Ohne dieses Argument würde die Funktion standardmäßig einen Fehler werfen, da in der Spalte `NAME` Fehlwerte enthalten sind und diese nich ohne weiteres verarbeitet werden.

Die `.query()` Methode

Man kann auch alternativ die `.query()` Methode zum Filtern von Zeilen verwenden. In diese Methode können wir prinzipiell dieselben Bedingungen schreiben wie in den vorherigen Beispielen, allerdings als String mit leicht abgeänderter Syntax. Hier eine direkt Gegenüberstellung:

```
df[df['NAME'] == 'Great Location for NYC']
```

```
                   NAME       id  ...          house_rules license
89  Great Location f...  1050491  ...  I just ask that ...     NaN

[1 rows x 26 columns]
```

```
df.query('NAME == "Great Location for NYC"')
```

```
                   NAME       id  ...          house_rules license
89  Great Location f...  1050491  ...  I just ask that ...     NaN

[1 rows x 26 columns]
```

Allerdings müssten Spaltennamen, die Leerzeichen enthalten, in der `.query()` Methode in diese ` Anführungszeichen gesetzt werden.

```
df2[df2['Construction year'] == 2005]
```

```
         room type  minimum nights  Construction year
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
```

```
df2.query('`Construction year` == 2005')
```

```
         room type  minimum nights  Construction year
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
```

Die `.query()` Methode ist also schlichtweg eine alternative Schreibweise, die in manchen Fällen übersichtlicher sein kann. Darüber hinaus kann die `.query()` Methode in bestimmten Fällen schneller als die herkömmliche Methode sein.

Duplikate entfernen

Zuletzt wollen wir noch auf das Entfernen von Duplikaten eingehen, das letztendlich auch eine Form des Filterns ist. Hierbei wird jede Zeile mit einer anderen Zeile verglichen und entfernt, wenn sie identisch ist. Die Methode `.duplicated()` gibt eine Spalte mit `True` und `False` Werten zurück, die angibt, ob eine Zeile bereits vorher vorkam. Erzeugen wir uns zunächst einen noch kleineren Teildatensatz `df3`, in welchem identische Zeilen, also Duplikate vorliegen:

```
df3 = df2[['room type', 'minimum nights']]
df3
```

```
         room type  minimum nights
0     Private room            10.0
1  Entire home/apt            30.0
2     Private room             3.0
3  Entire home/apt            30.0
4  Entire home/apt            10.0
5  Entire home/apt             3.0
```

```
#
df3.duplicated()
```

```
0    False
1    False
```

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

Es wird also nur die erste auftretende Zeile als nicht dupliziert betrachtet. Die Kombination aus `room type` `Entire home/apt` und `minimum nights` `30.0` taucht also in der zweiten Zeile erstmals auf und ist dort somit nicht als Duplikat markiert. In der vierten Zeile taucht sie dann aber erneut auf und ist somit als Duplikat markiert. Um Duplikate zu entfernen könnten wir nun die durch `.duplicated()` erzeugte Series verwenden und sie mittels `~` umkehren. Allerdings gibt es auch die noch einfachere Variante `.drop_duplicates()`, die direkt die Duplikate entfernt:

```
df3[~df3.duplicated()]
```

```
         room type  minimum nights
0     Private room            10.0
1  Entire home/apt            30.0
2     Private room             3.0
4  Entire home/apt            10.0
5  Entire home/apt             3.0
```

```
df3.drop_duplicates()
```

```
         room type  minimum nights
0     Private room            10.0
1  Entire home/apt            30.0
2     Private room             3.0
4  Entire home/apt            10.0
5  Entire home/apt             3.0
```

Schließlich soll noch erwähnt sein, dass die Methode `.drop_duplicates()` auch die Möglichkeit bietet, bei der Einstufung was ein Duplikat ist, nur einige der vorhandenen Spalten zu berücksichten. Dazu wird der Parameter `subset=` verwendet, der eine Liste von Spaltennamen erwartet. Wir könnten also hier zur selben Aussortierung von Duplikaten wie soeben mit `df3` kommen, auch wenn `df2` zugrundeliegt. Dessen zusätzliche Spalte `Construction year` würde eigentlich dafür sorgen, dass die zweite und vierte Zeile nicht als Duplikate betrachtet werden, da sie ja unterschiedliche Baujahre vorweisen. Da wir diese Spalte aber nicht in die Überprüfung einbeziehen, werden sie dennoch als Duplikate betrachtet und entfernt:

```
df2.drop_duplicates()
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
1  Entire home/apt            30.0             2007.0
2     Private room             3.0             2005.0
3  Entire home/apt            30.0             2005.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

```
df2.drop_duplicates(subset=['room type', 'minimum nights'])
```

```
         room type  minimum nights  Construction year
0     Private room            10.0             2020.0
1  Entire home/apt            30.0             2007.0
2     Private room             3.0             2005.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
```

Zeilen sortieren

Zum Abschluss wollen wir noch kurz auf das Sortieren von Zeilen eingehen. Dies kann mit der Methode `.sort_values()` erreicht werden. Diese Methode erwartet den Namen der Spalte, nach der sortiert werden soll. Mit dem Parameter `ascending=` kann die Sortierreihenfolge festgelegt werden, wobei es standardmäßig auf `True` gesetzt ist, sodass die Werte von oben nach unten aufsteigen. Hier ein Beispiel, wie wir den Teildatensatz `df2` nach der Spalte `minimum nights` sortieren:

```
df2.sort_values('minimum nights')
```

```
         room type  minimum nights  Construction year
2     Private room             3.0             2005.0
5  Entire home/apt             3.0             2013.0
4  Entire home/apt            10.0             2009.0
0     Private room            10.0             2020.0
3  Entire home/apt            30.0             2005.0
1  Entire home/apt            30.0             2007.0
```

```
df2.sort_values('minimum nights', ascending=False)
```

```
         room type  minimum nights  Construction year
1  Entire home/apt            30.0             2007.0
3  Entire home/apt            30.0             2005.0
0     Private room            10.0             2020.0
4  Entire home/apt            10.0             2009.0
2     Private room             3.0             2005.0
5  Entire home/apt             3.0             2013.0
```

Natürlich kann auch nach mehreren Spalten sortiert werden, indem eine Liste von Spaltennamen übergeben wird. Die Sortierreihenfolge wird dabei von links nach rechts festgelegt. Hier ein Beispiel, wie wir den Teildatensatz `df2` zuerst nach `room type` (aufsteigend, alphabetisch) und dann (also innerhalb desselben room types) nach `minimum nights` (absteigend) sortieren:

```
df2.sort_values(
  by=['room type', 'minimum nights'],
  ascending=[True, False]
  )
```

```
         room type  minimum nights  Construction year
1  Entire home/apt            30.0             2007.0
3  Entire home/apt            30.0             2005.0
4  Entire home/apt            10.0             2009.0
5  Entire home/apt             3.0             2013.0
0     Private room            10.0             2020.0
2     Private room             3.0             2005.0
```

Es gibt auch eine analoge Methode `.sort_index()`, die den Index des Dataframes sortiert. Auch hier kann der Parameter `ascending=` verwendet werden.

Thematisch passend ist auch die Funktion `.rank()`, die die Ränge der Zeilen bestimmt. Das heißt, dass die Daten nicht sortiert zurückgegeben werden, sondern die Ränge der Zeilen in der ursprünglichen Reihenfolge. Hier ein Beispiel, wie wir den Teildatensatz `df2` nach der Spalte `Construction year` ranken. Wann immer es ein Unentschieden gibt, wird der Durchschnittsrang vergeben. Man kann aber auch `.astype(int)` anhängen, um die Ränge in Ganzzahlen zu konvertieren.

Hinweis: Wir erzeugen hier eine neue Spalte `rank`, die die Ränge enthält. Eigentlich behandeln wir das erzeugen und bearbeiten von Spalten aber erst im nächsten Kapitel.

```
df5 = df2[['Construction year']]
df5['rank'] = df5['Construction year'].rank()
df5
```

```
   Construction year  rank
0             2020.0   6.0
1             2007.0   3.0
2             2005.0   1.5
3             2005.0   1.5
4             2009.0   4.0
5             2013.0   5.0
```

```
df5 = df2[['Construction year']]
df5['rank'] = df5['Construction year'].rank().astype(int)
df5
```

```
   Construction year  rank
0             2020.0     6
1             2007.0     3
2             2005.0     1
3             2005.0     1
4             2009.0     4
5             2013.0     5
```

Übungen

In einem vorangegangenen Kapitel wurden bereits die Hilfsfunktionen `.head()` und `.tail()` vorgestellt, die die ersten bzw. letzten Zeilen eines Dataframes ausgeben. Wie könnte man mit `.iloc[]` zum selben Ergebnis kommen?

- `df.head()` entspricht `df.iloc[ ___ ]`
- `df.tail()` entspricht `df.iloc[ ___ ]`

Die Spalte `review rate number` enthält die durchschnittliche Bewertung einer Unterkunft. Wie hoch ist der Durchschnitt der Bewertungen für die Unterkünfte, deren Name mit “Clean” beginnt im Vergleich zu den anderen Unterkünften? Hinweis 1: Du kannst z.B. `.mean()` direkt an einer (extrahierten) Series verwenden. Hinweis 2: Das Filtern von Unterkünften, deren Name mit “Clean” beginnt, wurde bereits oben im Text gezeigt.

Auf zwei Nachkommastellen gerundet haben Unterkünfte, deren Name

- mit “Clean” beginnt ein durchschnittliches Rating von ____
- nicht mit “Clean” beginnt ein durchschnittliches Rating von ____

Erzeuge zunächst den Teildatensatz `df_ex`, indem du folgenden Code ausführst:

```
pd.set_option('display.max_rows', 20)
```

```
df_ex = df[['neighbourhood', 'number of reviews', 'country code','host name']]
df_ex = df_ex[df_ex['host name'].str.startswith('Mari', na=False)]
df_ex = df_ex.head(20)
df_ex
```

```
          neighbourhood  number of reviews country code      host name
489         Fort Greene               35.0           US          Maria
590             Chelsea               32.0           US          Maria
1073        Sunset Park               28.0           US    Maria Luiza
1187    Lower East Side               57.0           US        Mariana
1302       East Village              115.0           US       Marianna
1351    Upper East Side               12.0           US   Marie-Jeanne
1425        Tottenville               59.0           US         Marina
1470    Upper West Side              108.0           US         Mariko
1508    Mariners Harbor               48.0           US          Maria
1517      East New York               13.0           US  Maria Daniela
1558            Concord               59.0           US       Marianne
1570            Concord               36.0           US       Marianne
1622           Arrochar                1.0           US         Marina
1680    Upper West Side                9.0           US         Mariko
1684        East Harlem               24.0           US         Mariko
1714             Harlem               97.0           US          Marie
1741       Williamsburg              121.0           US        Mariana
1824       West Village               38.0           US         Marina
1864            Chelsea                2.0           US          Marie
1944  Greenwich Village               17.0           US          Mario
```

Sorge nun dafür, dass von `df_ex` nur die Zeilen gefiltert werden, in denen der Host “Maria” heißt ohne aber die Spalte `host name` in deinem Code zu benutzen. Anders ausgedrückt: Tu für deine Vorgehensweise so als wäre die Spalte mit der Bezeichnunge `host name` nicht im Datensatz, filte aber dennoch so, dass nur die Zeilen gefiltert werden, in denen der Host “Maria” heißt.

- (A) Geschafft

Erzeuge nun selbst basierend auf dem vollen Datensatz `df` einen Teildatensatz, welcher (i) nur die Spalten `NAME`, `neighbourhood` und `host name` enthält und (ii) nur die Zeilen, in denen der Host “Peter” heißt. Entferne schließlich Duplikate so, dass pro Neighbourhood nur eine Zeile übrig bleibt.

- Der resultierende Datensatz hat __ Zeilen. Demnach gibt es entsprechend viele verschiedene Neighbourhoods mit mindestens einem Host names Peter.