Spalten selektieren

by Woche 8

Wie im vorigen Kapitel 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', 24)

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 apt ho...  ...  Clean up and treat t...     NaN
1       1002102    Skylit Midtown Castle  ...  Pet friendly but ple...     NaN
2       1002403  THE VILLAGE OF HARLE...  ...  I encourage you to u...     NaN
...         ...                      ...  ...                     ...     ...
102596  6093542  Comfy, bright room i...  ...                     NaN     NaN
102597  6094094  Big Studio-One Stop ...  ...                     NaN     NaN
102598  6094647     585 sf Luxury Studio  ...                     NaN     NaN

[102599 rows x 26 columns]
```

Spalten selektieren/sortieren

selektieren

Spalten selektieren bedeutet, dass man nur bestimmte Spalten behält und die anderen verwirft. Wir wissen bereits aus “5.2 Series & DataFrames”, dass wir eine oder mehrere Spalten selektieren können, indem wir

- Den Spaltennamen mit Punkt an den DataFrame hängen
- Den Spaltennamen oder eine Liste von Spaltennamen in eckigen Klammern an den DataFrame hängen
- Den Spaltennamen oder eine Liste von Spaltennamen hinter das Komma in `.loc[:,]` schreiben

- Den Spaltenindex oder eine Liste von Spaltenindizes hinter das Komma in `.iloc[:,]` schreiben

Dabei gibt es eine Besonderheit beim selektieren einer einzelnen Spalte: Während sowohl `df.price` als auch `df['price']` eine Series zurückgeben, gibt `df[['price']]` eine DataFrame mit nur einer Spalte/Series zurück. Das ist ein kleiner, aber feiner Unterschied, der für die weiteren Schritte wichtig sein kann.

In den folgenden Code-Beispielen sind die alternativen Befehle, die zum selben Ergebnis führen einfach untereinander geschrieben.

Eine Spalte als Series

```
df.price
df['price']
df.loc[:, 'price']
df.iloc[:, 15]
```

```
0           $966
1           $142
2           $620
           ...
102596      $988
102597      $546
102598    $1,032
Name: price, Length: 102599, dtype: object
```

Eine Spalte als DataFrame

```
#
df[['price']]
df.loc[:, ['price']]
df.iloc[:, [15]]
```

```
          price
0         $966
1         $142
2         $620
...         ...
102596    $988
102597    $546
102598  $1,032

[102599 rows x 1 columns]
```

Mehrere Spalten als DataFrame

```
#
df[['price', 'room type']]
df.loc[:, ['price', 'room type']]
df.iloc[:, [15, 13]]
```

```
          price        room type
0         $966      Private room
1         $142   Entire home/apt
2         $620      Private room
...         ...               ...
102596    $988      Private room
102597    $546   Entire home/apt
102598  $1,032   Entire home/apt

[102599 rows x 2 columns]
```

sortieren

Gleichermaßen können wir auch die Spalten gleichzeitig selektieren und ihre Reihenfolge ändern, indem wir die Spalten direkt in der gewünschten Reihenfolge in die eckigen Klammern schreiben.

```
neuer_df = df[['room type', 'price']]
neuer_df = df.loc[:, ['room type', 'price']]
neuer_df = df.iloc[:, [17, 15]]

neuer_df
```

```
        minimum nights    price
0                 10.0    $966
1                 30.0    $142
2                  3.0    $620
...                ...      ...
102596             3.0    $988
102597             2.0    $546
102598             1.0  $1,032

[102599 rows x 2 columns]
```

Wollen wir allerdings nur die Reihenfolge *einiger* Spalten ändern und die nicht-betroffenen Spalten trotzdem beibehalten, so müssen wir trotzdem alle Spalten in der gewünschten Reihenfolge explizit angeben. Das müssen wir aber nicht manuell tun, sondern können uns die Spaltennamen mit `df.columns` holen und dann z.B. mithilfe von *List Comprehension* die neue Reihenfolge als Liste speichern. Als Beispiel möchten wir die Spalten `room type` und `price` an den Anfange der Tabelle setzen, wobei alle anderen

Spalten dahinter so bleiben wie sie sind. Der Befehl `df.columns` gibt uns alle Spaltennamen als Liste zurück:

```
df.columns
```

```
Index(['id', 'NAME', 'host id', 'host_identity_verified', 'host name',
       'neighbourhood group', 'neighbourhood', 'lat', 'long', 'country',
       'country code', 'instant_bookable', 'cancellation_policy', 'room type',
       'Construction year', 'price', 'service fee', 'minimum nights',
       'number of reviews', 'last review', 'reviews per month',
       'review rate number', 'calculated host listings count',
       'availability 365', 'house_rules', 'license'],
      dtype='object')
```

Das können wir uns zu Nutze machen, indem wir eine Liste `erste_spalten` erstellen, die die gewünschten Spaltennamen enthält und dann die restlichen Spaltennamen in einer Liste `restliche_spalten` speichern. Letzteres können wir mit *List Comprehension* (siehe 2.5 If-Else & Loops) erreichen, indem wir alle Spaltennamen durchgehen und nur diejenigen behalten, die nicht in `erste_spalten` enthalten sind. Schließlich fügen wir beides zu einer gemeinsamen Liste `neue_spaltenreihenfolge` zusammen, die wir dann übergeben können.

```
erste_spalten = ['room type', 'price']
restliche_spalten = [spalte for spalte in df.columns if spalte not in
erste_spalten]
neue_spaltenreihenfolge = erste_spalten + restliche_spalten

neuer_df = df[neue_spaltenreihenfolge]
neuer_df = df.loc[:,neue_spaltenreihenfolge]
# iloc Alternative hier nicht gezeigt

neuer_df
```

```
              room type    price  ...              house_rules license
0          Private room    $966   ...  Clean up and treat t...     NaN
1       Entire home/apt    $142   ...  Pet friendly but ple...     NaN
2          Private room    $620   ...  I encourage you to u...     NaN
...                 ...      ...  ...                     ...     ...
102596     Private room    $988   ...                     NaN     NaN
102597  Entire home/apt    $546   ...                     NaN     NaN
102598  Entire home/apt  $1,032   ...                     NaN     NaN

[102599 rows x 26 columns]
```

selektiere von:bis

Manchmal möchte man auch von einer bestimmten Spalte bis zu einer anderen Spalte selektieren. Das können wir z.B. durch slicing erreichen, indem einfach die zwei Spaltennamen durch einen Doppelpunkt getrennt angeben.

```
df.loc[:, 'room type':'price']
```

```
              room type  Construction year    price
0          Private room             2020.0    $966
1       Entire home/apt             2007.0    $142
2          Private room             2005.0    $620
...                 ...                ...      ...
102596     Private room             2009.0    $988
102597  Entire home/apt             2015.0    $546
102598  Entire home/apt             2010.0  $1,032

[102599 rows x 3 columns]
```

Wollen wir allerdings zusätzlich z.B. noch eine weitere Spalte wie `id` selektieren, so geht das **nicht** einfach indem wir beispielsweise schreiben `df.loc[:, ['id', 'room type':'price']]'`. Leider erlaubt Pandas diese Kombination aus expliziten Spaltennamen und einem Slice nicht. Stattdessen könnten wir etwas umständlich (aber dennoch womöglich besser als ganz manuell) die Spaltennamen aus separaten Schritten kombinieren.

```
cols = df.loc[:, 'room type':'price'].columns
cols = ['id'] + list(cols)

df.loc[:, cols]
```

```
             id        room type  Construction year    price
0       1001254     Private room             2020.0    $966
1       1002102  Entire home/apt             2007.0    $142
2       1002403     Private room             2005.0    $620
...         ...              ...                ...      ...
102596  6093542     Private room             2009.0    $988
102597  6094094  Entire home/apt             2015.0    $546
102598  6094647  Entire home/apt             2010.0  $1,032

[102599 rows x 4 columns]
```

Möchten wir eine oder mehrere Spalten löschen, so können wir das mit dem Befehl `drop()` tun. Dabei müssen wir angeben, welche Spalten gelöscht werden sollen und ob es sich dabei um Zeilen oder Spalten handelt. Letzteres können wir mit dem Argument

`axis` tun, wobei `axis = 0` für Zeilen[1] und `axis = 1` für Spalten steht. Alternativ kann man die zu löschenden Spalten auch dem Argument `columns=` übergeben - dann wird `axis` automatisch auf 1 gesetzt.

```
neuer_df = df.loc[:, 'room type':'price']
neuer_df.drop('room type', axis = 1)
```

```
        Construction year    price
0                  2020.0    $966
1                  2007.0    $142
2                  2005.0    $620
...                   ...      ...
102596             2009.0    $988
102597             2015.0    $546
102598             2010.0  $1,032

[102599 rows x 2 columns]
```

```
neuer_df = df.loc[:, 'room type':'price']
neuer_df.drop(columns = ['room type', 'price'])
```

```
        Construction year
0                  2020.0
1                  2007.0
2                  2005.0
...                   ...
102596             2009.0
102597             2015.0
102598             2010.0

[102599 rows x 1 columns]
```

anhand von Spalteneigenschaften

Ebenso muss man manchmal Spalten selektieren/löschen, die bestimmte Eigenschaften haben. Das können z.B. Spalten sein,

- die einen bestimmten Datentyp haben
- deren Spaltenname mit einem bestimmten Buchstaben beginnt oder endet o.ä.
- die nur fehlende Werte enthalten
- die nur einen einzigen Wert enthalten

[1] Prinzipiell kann man mit `drop()` auch Zeilen löschen und deren Indices könnten ja ebenfalls als Strings wie `room type` usw. definiert worden sein. Deshalb ist es wichtig, explizit anzugeben, dass es sich um Spalten handelt, indem wir `axis = 1` setzen.

All diese Fälle haben gemeinsam, dass wir erst eine entsprechende Liste von Spaltennamen erstellen könnten und dann diese Liste mit den oben gelerente Befehlen verwenden können. Es gibt aber auch spezielle Funktionen, die uns das Leben erleichtern. Hier werden wir uns ein paar davon anschauen.

Datentyp

Möchten wir z.B. alle Spalten selektieren, die den Datentyp `float64` haben, so könnte man erst alle Spaltennamen mit dem entsprechenden Datentyp herausfinden und dann diese Spalten selektieren. Wir tun das hier wie folgt: `df.dtypes` gibt uns eine Series zurück, die die Datentypen aller Spalten enthält, wobei die Spaltennamen in dieser Series im Index stehen. Mit `df.dtypes == 'float64'` erhalten wir eine Series mit `True` und `False` Werten, die angibt, ob der Datentyp der Spalte `float64` ist. Mit `df.dtypes[df.dtypes == 'float64']` erhalten wir schließlich nur die Spaltennamen, die dabei ein `True` haben. Von dieser Series können wir dann die Indices (also die Spaltennamen) extrahieren und diese dann zum Selektieren verwenden.

```
cols = df.dtypes[df.dtypes == 'float64'].index
df[cols]
```

```
             lat      long  ...  calculated host listings count  \
0       40.64749 -73.97237  ...                          6.0
1       40.75362 -73.98377  ...                          2.0
2       40.80902 -73.94190  ...                          1.0
...          ...       ...  ...                          ...
102596  40.67505 -73.98045  ...                          1.0
102597  40.74989 -73.93777  ...                          1.0
102598  40.76807 -73.98342  ...                          1.0

        availability 365
0                  286.0
1                  228.0
2                  352.0
...                  ...
102596             342.0
102597             386.0
102598              69.0

[102599 rows x 9 columns]
```

Für speziell solche Fälle gibt es eine Hilfsfunktion names `select_dtypes()`, mit der wir direkt die Spalten selektieren, die einen bestimmten Datentyp haben. Dabei können wir auch mehrere Datentypen angeben, indem wir sie in einer Liste übergeben und außerdem zwischen `include=` und `exclude=` wählen. Da es in unserem Datensatz nur die

Datentypen `int64`, `float64` und `object` gibt, können wir also wie folgt jeweils nur die Spalten des Ganzzahlen-Typs behalten.

```
df.select_dtypes(include = 'int64')
```

```
             id      host id
0       1001254  80014485718
1       1002102  52335172823
2       1002403  78829239556
...         ...          ...
102596  6093542  69050334417
102597  6094094  11160591270
102598  6094647  68170633372

[102599 rows x 2 columns]
```

```
df.select_dtypes(exclude = ['float64', 'object'])
```

```
             id      host id
0       1001254  80014485718
1       1002102  52335172823
2       1002403  78829239556
...         ...          ...
102596  6093542  69050334417
102597  6094094  11160591270
102598  6094647  68170633372

[102599 rows x 2 columns]
```

Spaltenname

Möchten wir z.B. alle Spalten selektieren, die den String `id` enthalten, so könnten wir das wie folgt tun: `df.columns.str.contains('id')` gibt uns eine Series zurück, die `True` und `False` Werte enthält, je nachdem ob der Spaltenname den String `id` enthält oder nicht. Mit `df.columns[df.columns.str.contains('id')]` erhalten wir schließlich nur die Spaltennamen, die dabei ein `True` haben. Von dieser Series können wir dann die Indices (also die Spaltennamen) extrahieren und diese dann zum Selektieren verwenden.

```
cols = df.columns[df.columns.str.contains('id')]
df[cols]
```

```
             id      host id host_identity_verified
0       1001254  80014485718            unconfirmed
1       1002102  52335172823               verified
2       1002403  78829239556                    NaN
...         ...          ...                    ...
102596  6093542  69050334417            unconfirmed
102597  6094094  11160591270            unconfirmed
102598  6094647  68170633372            unconfirmed

[102599 rows x 3 columns]
```

Für solche Fälle gibt es wiederum eine Hilfsfunktion names `filter()`, mit der wir direkt die Spalten selektieren, die einen bestimmten String enthalten. Mit dem Argument `like=` können wir dabei den String angeben, den die Spalten enthalten sollen.

```
df.filter(like = 'id')
```

```
             id      host id host_identity_verified
0       1001254  80014485718            unconfirmed
1       1002102  52335172823               verified
2       1002403  78829239556                    NaN
...         ...          ...                    ...
102596  6093542  69050334417            unconfirmed
102597  6094094  11160591270            unconfirmed
102598  6094647  68170633372            unconfirmed

[102599 rows x 3 columns]
```

Neben `like=` kann in der `filter()` Funktion auch `regex=` verwendet werden, um die Spaltennamen anhand eines regulären Ausdrucks (siehe Ende 5.4 Strings & Method Chaining) zu filtern. Beispielsweise könnten wir so auch alle Spalten behalten, die mit `review` beginnen oder mit `id` enden.

```
# ^ steht in regex für den String-Anfang
df.filter(regex = '^review')
```

```
        reviews per month  review rate number
0                    0.21                 4.0
1                    0.38                 4.0
2                     NaN                 5.0
...                   ...                 ...
102596                NaN                 5.0
102597               0.10                 3.0
102598                NaN                 3.0
```

```
[102599 rows x 2 columns]
```

```
# $ steht in regex für das String-Ende
df.filter(regex = 'id$')
```

```
             id      host id
0       1001254  80014485718
1       1002102  52335172823
2       1002403  78829239556
...         ...          ...
102596  6093542  69050334417
102597  6094094  11160591270
102598  6094647  68170633372

[102599 rows x 2 columns]
```

Übrigens hat die `filter()` Funktion auch ein `axis=` Argument, kann also prinzipiell wie schon `drop()` auch auf Zeilen angewendet werden. Der Default ist aber, dass es sich um Spalten handelt, sodass wir das `axis=1` nicht explizit angeben müssen.

Nur ein Wert

Manchmal erhält man große, unübersichtliche Datensätze, die viele Spalten enthalten. Es kann dann nützlich sein all die Spalten zu löschen, die nur einen einzigen Wert enthalten, da diese für die Analyse ggf. keinen relevanten Mehrwert mitbringen.

Wir können das z.B. mit der Funktion `nunique()` tun, die uns die Anzahl der einzigartigen Werte je Spalte zurückgibt. `df.nunique() == 1` gibt uns eine Series zurück, die `True` und `False` Werte enthält, je nachdem ob die Spalte nur einen einzigen Wert enthält oder nicht. Mit `df.columns[df.nunique() == 1]` erhalten wir schließlich nur die Spaltennamen, die dabei ein `True` haben. Von dieser Series können wir dann die Indices (also die Spaltennamen) extrahieren und diese dann zum Selektieren verwenden.

```
cols = df.columns[df.nunique() == 1]
df[cols] # bzw. df.drop(cols, axis = 1)
```

```
             country country code license
0       United States           US     NaN
1       United States           US     NaN
2       United States           US     NaN
...               ...          ...     ...
```

```
102596  United States           US     NaN
102597  United States           US     NaN
102598  United States           US     NaN

[102599 rows x 3 columns]
```

Fehlende Werte

> **i** Fehlwerte
>
> Fehlwerte werden in einem späteren Kapitel ausführlicher behandelt. Hier ist es zunächst wichtig zu wissen, dass fehlende Werte oder Daten in Python nicht nur mit `None` markiert werden, sondern in Pandas auch mit `NA`[2], was für "not available" steht. In Pandas gibe die Funktionen `isna()` und `notna()` gibt, die uns `True` und `False` zurückgeben, je nachdem ob Werte fehlen oder vorhanden sind.

Möchten wir z.B. alle Spalten löschen, die ausschließlich Fehlwerte (also keine Werte) enthalten, so könnten wir das wie folgt tun: `df.isna().all()` gibt uns eine Series zurück, die `True` und `False` Werte enthält, je nachdem ob die Spalte nur fehlende Werte enthält oder nicht. Mit `df.columns[df.isna().all()]` erhalten wir schließlich nur die Spaltennamen, die dabei ein `True` haben. Von dieser Series können wir dann die Indices (also die Spaltennamen) extrahieren und diese dann zum Löschen verwenden.

Leider hat unser Beispieldatensatz keine Spalten, die nur fehlende Werte enthalten, sodass wir uns hier kurz einen neuen Datensatz erstellen.

```
df2 = pd.DataFrame({
  'a': [1, 2, 3],
  'b': [4, 5, pd.NA],
  'c': [None, None, None],
  'd': [pd.NA, pd.NA, pd.NA]
  })

df2
```

```
   a     b     c     d
0  1     4  None  <NA>
1  2     5  None  <NA>
2  3  <NA>  None  <NA>
```

[2] oder `NaN` oder `NaT`

```
df2.isna()
df2.isna().all()
```

```
       a      b     c     d
0  False  False  True  True
1  False  False  True  True
2  False   True  True  True
```

```
a    False
b    False
c     True
d     True
dtype: bool
```

```
cols = df2.columns[df2.isna().all()]
df2.drop(cols, axis = 1)
```

```
   a     b
0  1     4
1  2     5
2  3  <NA>
```

Es gibt aber auch hier wieder eine Hilfsfunktion namens `dropna()`, mit der wir direkt die Spalten löschen können, die nur fehlende Werte enthalten. Mit dem Argument `how=` können wir dabei angeben, ob wir die Spalten löschen wollen, die entweder nur fehlende Werte enthalten (`all`) oder mindestens einen fehlenden Wert enthalten (`any`).

```
df2.dropna(axis = 1, how = 'all')
```

```
   a     b
0  1     4
1  2     5
2  3  <NA>
```

Übungen

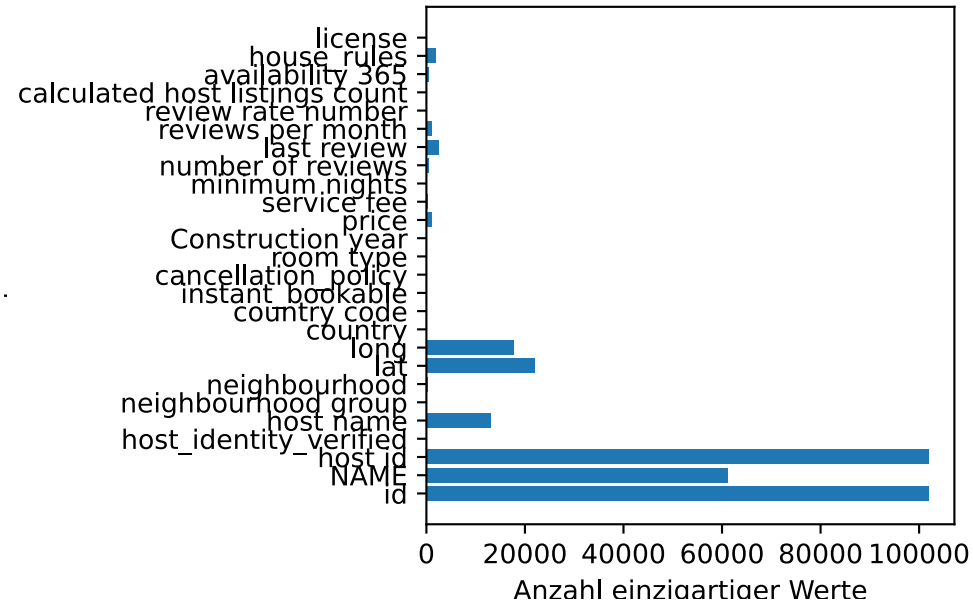
Für unseren AirBnB Datensatz können wir die Anzahl einzigartiger Werte pro Spalte wie folgt in einem Balkendiagramm darstellen. Das Balkendiagramm ist in diesem Fall horizontal, da wir `.barh()` anstelle von `.bar()` verwenden.

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

# df von oben verwenden

unique_counts = df.nunique()
```

```
plt.figure()
plt.barh(y = unique_counts.index, width = unique_counts.values)
plt.ylabel('Spalten')
plt.xlabel('Anzahl einzigartiger Werte')
plt.subplots_adjust(left=0.4)
plt.show()
```



Verändere die `0.4` in `plt.subplots_adjust(left=0.4)` in größere und kleinere Werte, und versuche zu verstehen, was passiert, wenn du die Abbildung neu erstellst.

- (A) Geschafft

Aufgrund der sehr großen Balken für die Spalten `id` und `NAME` und `host id` ist es schwer, die kleinere Balken überhaupt zu erkennen. Erzeuge die Abbildung nochmal, aber ohne die drei genannten Spalten. Erreiche das, indem du einen reduzierten DataFrame erstellst, der nur die anderen Spalten enthält. Dies sollst du auf zwei verschiedene Arten tun: Einmal indem du die drei Spalten explizit angibst, die du löschen möchtest. Einmal indem du nur Spalten behalten möchtest, die maximal 25000 einzigartige Werte enthalten.

- (A) Geschafft

Erzeuge schließlich die Abbildung noch ein drittes Mal, aber nun nur für Spalten, die maximal 10 einzigartige Werte enthalten.

- (A) Geschafft

Selektiere aus unserem AirBnB Datensatz nur die Spalten, die numerische Daten enthalten (Integer und Float). Sortiere die Spalten alphabetisch. Anschließend soll mittels `df.columns.str.lower()` alle Spaltennamen in Kleinbuchstaben umgewandelt werden. Sortiere die Spalten erneut alphabetisch. Vergleiche die Ergebnisse der beiden Sortierungen und prüfe, ob es Unterschiede gibt, die mit der Groß- und Kleinschreibung zusammenhängen.

- (A) Nein, beide Sortierungen sind identisch.
- (B) Ja, Großbuchstaben werden vor Kleinbuchstaben sortiert.
- (C) Ja, Großbuchstaben hinter Kleinbuchstaben sortiert.

Erzeuge einen Dataframe namens `mein_df` mit 5 Spalten und 5 Zeilen. Nach ausführen dieses Befehls sollte `mein_df2` genau 4 Spalten und `mein_df3` genau 2 Spalten haben.

```
mein_df2 = mein_df.dropna(axis = 1, how = 'all')
mein_df3 = mein_df2.filter(like = 'spalte')
```

- (A) Geschafft