Spalten erzeugen & bearbeiten

by Woche 9

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 Kapitel lernen wir, wie wir in einem DataFrame neue Spalten erzeugen und vorhandene Spalten bearbeiten können. In Gewisser Hinsicht ist die Handhabung dieser beiden Punkte sehr ähnlich, da wir beim “bearbeiten” einer Spalte im Endeffekt auch eine neue Spalte erzeugen, die die alte Spalte ersetzt/überschreibt.

Auch hier wollen wir uns zunächst einen übersichtlichen Teildatensatz erzeugen um die Beispiele besser nachvollziehen zu können.

```
rows = range(10, 16)
cols = ['NAME', 'price', 'last review']

df2 = df.loc[rows, cols]
df2
```

```
                   NAME  price last review
10  Cute & Cozy Lowe...  $319     6/9/2019
11  Beautiful 1br on...  $606    6/22/2019
12  Central Manhatta...  $714    6/23/2019
13  Lovely Room 1, G...  $580    6/24/2019
14  Wonderful Guest ...  $149     7/5/2019
15  West Village Nes...  $578   10/31/2018
```

Spalten erzeugen

Wir haben mehreren Möglichkeiten eine neue Spalte zu erzeugen:

- Mit eckigen Klammern und einem neuen Spaltennamen
- Mit der `.assign()` Methode
- Mit der `.insert()` Methode

Um eine neue Spalte zu erzeugen, können wir einfach einen Wert einem Spaltennamen zuweisen, den es nicht gibt. Dieser Wert steht dann in allen Zellen der neuen Spalte:

```
df2['Neu'] = 'Hallo!'
df2
```

```
                   NAME  price last review     Neu
10  Cute & Cozy Lowe...  $319     6/9/2019  Hallo!
11  Beautiful 1br on...  $606    6/22/2019  Hallo!
12  Central Manhatta...  $714    6/23/2019  Hallo!
13  Lovely Room 1, G...  $580    6/24/2019  Hallo!
14  Wonderful Guest ...  $149     7/5/2019  Hallo!
15  West Village Nes...  $578   10/31/2018  Hallo!
```

```
df2['Neu'] = round(number = 2/3, ndigits = 4)
df2
```

```
                   NAME  price last review     Neu
10  Cute & Cozy Lowe...  $319     6/9/2019  0.6667
11  Beautiful 1br on...  $606    6/22/2019  0.6667
12  Central Manhatta...  $714    6/23/2019  0.6667
13  Lovely Room 1, G...  $580    6/24/2019  0.6667
14  Wonderful Guest ...  $149     7/5/2019  0.6667
15  West Village Nes...  $578   10/31/2018  0.6667
```

```
df2 = df2.drop(columns='Neu') # Lösche Spalte 'Neu' wieder
```

Deutlich relevanter ist es natürlich, wenn wir eine neue Spalte erzeugen, die von den Werten einer oder mehrerer anderer Spalten abhängt. Eine angebrachte Anwendung wäre z.B. eine neue Version der Spalte `price` zu erzeugen, die nicht ein String beginnen mit “$” ist, sondern ein numerischer Wert. Wir erzeugen also eine neue Spalte `price_num`, indem wir das Dollarzeichen von den Werten der Spalte `price` entfernen und den Rest in einen numerischen Wert umwandeln. Hier sind zwei Möglichkeiten gezeigt um dies zu erreichen: Entweder entfernen wir einfach immer das erste Zeichen, oder wir ersetzen explizit das Dollarzeichen durch einen leeren String, sodass danach `.astype(float)` angewendet werden kann:

```
df2['price_num'] = (
  df2['price']   # Wähle Spalte 'price'
  .str[1:]       # Behalte ab zweitem Zeichen
  .astype(float) # Wandle in numerischen Wert um
)

df2
```

```
                   NAME  price last review  price_num
10  Cute & Cozy Lowe...  $319     6/9/2019      319.0
11  Beautiful 1br on...  $606    6/22/2019      606.0
12  Central Manhatta...  $714    6/23/2019      714.0
13  Lovely Room 1, G...  $580    6/24/2019      580.0
14  Wonderful Guest ...  $149     7/5/2019      149.0
15  West Village Nes...  $578   10/31/2018      578.0
```

```
df2['price_num'] = (
  df2['price']          # Wähle Spalte 'price'
  .str.replace('$', '') # Ersetze '$' durch ''
  .astype(float)        # Wandle in numerischen Wert um
)

df2
```

```
                   NAME  price last review  price_num
10  Cute & Cozy Lowe...  $319     6/9/2019      319.0
11  Beautiful 1br on...  $606    6/22/2019      606.0
12  Central Manhatta...  $714    6/23/2019      714.0
13  Lovely Room 1, G...  $580    6/24/2019      580.0
14  Wonderful Guest ...  $149     7/5/2019      149.0
15  West Village Nes...  $578   10/31/2018      578.0
```

Erst jetzt könnten wir sinnvolle Analysen über die Preise durchführen, z.B. den Durchschnitt berechnen oder eine Abbildung erstellen.

```
df2['price_num'].mean()
```

```
np.float64(491.0)
```

```
df2['price'].mean()
```

```
TypeError: Could not convert string '$319 $606 $714 $580 $149 $578 ' to
numeric
```

> **i** Zusätzlicher Hinweis
>
> Die Art und Weise wie wir die Spalte `price` hier konvertieren funktioniert. Nachdem wir das Dollarzeichen entfernt haben, bleiben nur noch Zahlen im String übrig, sodass eine Konvertierung in einen numerischen Wert möglich ist. Derselbe Befehl würde allerdings für den gesamten Datensatz `df` nicht ausreichen/funktionieren. Das liegt daran, dass dort Preise höher als $1000 vorkommen, bei denen dann im String ein Komma als Tausendertrennzeichen verwendet wird - also z.B. `1,000`. In diesem Fall müssten wir das Komma ebenfalls entfernen, bevor wir den String in einen numerischen Wert umwandeln können. Dort bräuchten wir dann also eine zusätzliche `.str.replace(',', '')` Methode.

Die `.assign()` Methode

Die `.assign()` Methode ist eine weitere Möglichkeit, eine neue Spalte zu erzeugen. Hierbei wird ein neues DataFrame zurückgegeben, das die alte Spalte und die neue Spalte enthält. Das ursprüngliche DataFrame bleibt unverändert. Im Gegensatz zu den eckigen Klammern können wir hier auch mehrere Spalten auf einmal erzeugen.

```
temps = pd.DataFrame({'temp_C': [10.0, 25.0]})

temps.assign(
  temp_F = temps['temp_C'] * 9 / 5 + 32,
  temp_K = temps['temp_C'] + 273.15
  )
```

```
   temp_C  temp_F  temp_K
0    10.0    50.0  283.15
1    25.0    77.0  298.15
```

Die `insert()` Methode

Die `insert()` Methode ist noch eine Möglichkeit, eine neue Spalte zu erzeugen. Hierbei wird die neue Spalte an einer bestimmten Position eingefügt. Im Gegensatz zur `.assign()` Methode wird das ursprüngliche DataFrame verändert.

```
temps = pd.DataFrame({'temp_C': [10.0, 25.0]})

temps.insert(loc = 0, column = 'temp_F', value = temps['temp_C'] * 9 / 5 + 32)
temps
```

```
   temp_F  temp_C
0    50.0    10.0
1    77.0    25.0
```

Spalten bearbeiten

Tatsächlich ist das Bearbeiten einer Spalte nicht viel anders als das Erzeugen einer neuen Spalte. Wir können einfach den Wert einer Spalte überschreiben, indem wir einen neuen Wert zuweisen. Wir wollen also die die Spalte `price_num` löschen und stattdessen `price` direkt in numerische Werte umwandeln:

```
df2 = df2.drop(columns='price_num')

df2['price'] = (
  df2['price']   # Wähle Spalte 'price'
  .str[1:]       # Behalte ab zweitem Zeichen
  .astype(float) # Wandle in numerischen Wert um
)

df2
```

```
                   NAME  price last review
10  Cute & Cozy Lowe...  319.0    6/9/2019
11  Beautiful 1br on...  606.0   6/22/2019
12  Central Manhatta...  714.0   6/23/2019
13  Lovely Room 1, G...  580.0   6/24/2019
14  Wonderful Guest ...  149.0    7/5/2019
15  West Village Nes...  578.0  10/31/2018
```

Die `.case_when()` Methode

Manchmal ist es nützlich, eine neue Spalte zu erzeugen, die von den Werten einer anderen Spalte abhängt, aber nicht einfach durch eine einfache Transformation erreicht

werden kann. In diesem Fall können wir die (noch relativ neue) `.case_when()` Methode verwenden. Wie der Name schon andeutet, können wir hier eine Fallunterscheidung definieren, die für jeden Wert der ursprünglichen Spalte eine andere Transformation durchführt. Im einfachsten Fall können wir eine Liste von Bedingungen und auszugebenen Werten angeben, also hier beispielsweise Preiskategorien:

```
caselist=[
  (df2['price'] < 200, 'günstig'),
  (df2['price'] < 580, 'mittel'),
  (df2['price'] >= 580, 'teuer')
]

df2['price_cat'] = df2['price'].case_when(caselist)

df2
```

```
                  NAME  price last review price_cat
10  Cute & Cozy Lowe...  319.0    6/9/2019    mittel
11  Beautiful 1br on...  606.0   6/22/2019     teuer
12  Central Manhatta...  714.0   6/23/2019     teuer
13  Lovely Room 1, G...  580.0   6/24/2019     teuer
14  Wonderful Guest ...  149.0    7/5/2019   günstig
15  West Village Nes...  578.0  10/31/2018    mittel
```

Es sei darauf hingewiesen, dass die Bedingungen nacheinander überprüft werden. Das heißt, dass nur die erste Bedingung, die erfüllt ist, angewendet wird. Das wird deutlich, wenn wir die Reihenfolge der Bedingungen von eben ändern, da so nichts mehr als “günstig” eingestuft werden kann:

```
caselist=[
  (df2['price'] < 580, 'mittel'),
  (df2['price'] < 200, 'günstig'),
  (df2['price'] >= 580, 'teuer')
]

df2['price_cat'] = df2['price'].case_when(caselist)

df2
```

```
                  NAME  price last review price_cat
10  Cute & Cozy Lowe...  319.0    6/9/2019    mittel
11  Beautiful 1br on...  606.0   6/22/2019     teuer
12  Central Manhatta...  714.0   6/23/2019     teuer
13  Lovely Room 1, G...  580.0   6/24/2019     teuer
```

```
14  Wonderful Guest ...  149.0    7/5/2019    mittel
15  West Village Nes...  578.0  10/31/2018    mittel
```

Außerdem gilt, dass wenn keine Bedingung erfüllt ist, der Wert nicht verändert wird. Dies kann zu unbemerkten Problemen führen, da man manchmal davon ausgeht, dass alle Werte transformiert wurden, obwohl dies nicht der Fall ist. Hier ein Beispiel, in welchem wir versehentlich den Fall übersehen, dass ein Preis genau $580 ist und so auf unserer Grenze liegt, für die hier keine der Bedingungen zutrifft.

Für solche Fälle können wir quasi als Fallnetz eine Default-Bedingung angeben, die für alle Werte gilt, die nicht durch die vorherigen Bedingungen abgedeckt sind - z.B. indem wir eine Bedingung angeben, die möglichst immer zutrifft wie `df2['price']==df2['price']`. Anschließend könnte man dann nämlich nach dem entsprechenden Fallnetz-Wert (siehe unten `'--------'`) filtern um zu prüfen, ob alles erwartungsgemäß geklappt hat.

```
caselist=[
  (df2['price'] < 200, 'günstig'),
  (df2['price'] < 580, 'mittel'),
  (df2['price'] > 580, 'teuer')
]


df2['price_cat'] = df2['price'].case_when(caselist)

df2
```

```
                  NAME  price last review price_cat
10  Cute & Cozy Lowe...  319.0    6/9/2019    mittel
11  Beautiful 1br on...  606.0   6/22/2019     teuer
12  Central Manhatta...  714.0   6/23/2019     teuer
13  Lovely Room 1, G...  580.0   6/24/2019     580.0
14  Wonderful Guest ...  149.0    7/5/2019   günstig
15  West Village Nes...  578.0  10/31/2018    mittel
```

```
caselist=[
  (df2['price'] < 200, 'günstig'),
  (df2['price'] < 580, 'mittel'),
  (df2['price'] > 580, 'teuer'),
  (df2['price']==df2['price'], '--------')
]

df2['price_cat'] = df2['price'].case_when(caselist)
```

```
df2
```

```
                   NAME  price last review price_cat
10  Cute & Cozy Lowe...  319.0    6/9/2019    mittel
11  Beautiful 1br on...  606.0   6/22/2019     teuer
12  Central Manhatta...  714.0   6/23/2019     teuer
13  Lovely Room 1, G...  580.0   6/24/2019  --------
14  Wonderful Guest ...  149.0    7/5/2019   günstig
15  West Village Nes...  578.0  10/31/2018    mittel
```

```
df2 = df2.drop(columns='price_cat')
```

Weitere Datentypen

In den vorigen Kapiteln haben wir bereits gesehen, dass es in Pandas verschiedene Datentypen gibt, die wir für unsere Spalten verwenden können. Die wichtigsten sind:

- `int` für Ganzzahlen
- `float` für Fließkommazahlen
- `str` für Zeichenketten
- `bool` für Wahrheitswerte
- `category` für kategorische Variablen
- `datetime` für Zeitstempel
- `object` für beliebige Python Objekte

Die meisten dieser Datentypen sind uns schon bekannt, doch wir wollen hier noch auf `datetime` und `category` eingehen.

`datetime`

Der `datetime` Datentyp repräsentiert Zeitstempel, also einen bestimmten Zeitpunkt. Dieser Datentyp ist sehr mächtig, da wir damit nicht nur einen Zeitpunkt, sondern auch Zeitdifferenzen und Zeitintervalle darstellen können. Aktuell ist die Spalte `last review` in unserem `df2` noch ein String. Wir können das Datum also lesen, doch wie auch mit dem `price` Spalte vor der Umwandlung in eine numerische Spalte könnten wir so keine sinnvollen Analysen durchführen. Das wird spätestens dann klar, wenn man nach der Spalte sortieren will und merkt, dass die Sortierung nicht chronologisch, sondern alphabetisch ist.

```
df2.sort_values('last review')
```

```
                   NAME  price last review
15  West Village Nes...  578.0  10/31/2018
11  Beautiful 1br on...  606.0   6/22/2019
12  Central Manhatta...  714.0   6/23/2019
13  Lovely Room 1, G...  580.0   6/24/2019
10  Cute & Cozy Lowe...  319.0    6/9/2019
14  Wonderful Guest ...  149.0    7/5/2019
```

Dies wiederum führt z.B. dann zu Problemen, wenn man die Werte chronologisch auf einer Achse eine Abbildung darstellen will.

Um die Spalte in einen `datetime` Datentyp umzuwandeln, können wir die `pd.to_datetime()` Funktion verwenden. Diese Funktion ist sehr mächtig und kann viele verschiedene Datums-Formate erkennen. Da die Daten hier in einem Standardformat vorliegen, reicht es einfach die Spalte zu übergeben.

```
df2['last_review_date'] = pd.to_datetime(df2['last review'])
df2.sort_values('last_review_date')
```

```
                   NAME  price last review last_review_date
15  West Village Nes...  578.0  10/31/2018       2018-10-31
10  Cute & Cozy Lowe...  319.0    6/9/2019       2019-06-09
11  Beautiful 1br on...  606.0   6/22/2019       2019-06-22
12  Central Manhatta...  714.0   6/23/2019       2019-06-23
13  Lovely Room 1, G...  580.0   6/24/2019       2019-06-24
14  Wonderful Guest ...  149.0    7/5/2019       2019-07-05
```

Nun funktioniert auch direkt die Sortierung. Es fällt auf, dass das Datum nun als Jahr-Monat-Tag angegeben ist, wo es doch vorher im String als Monat/Tag/Jahr geschrieben war. Das liegt daran, dass Pandas das ISO Format bevorzugt, welches international einheitlich ist und so keine Verwechslungen zulässt[1]. Beeindruckenderweise hat `pd.to_datetime()` das vorliegende Format automatisch erkannt und umgewandelt. In der Praxis wird dies allerdings nicht immer so reibungslos funktionieren, da es viele verschiedene Datumsformate gibt. In solchen Fällen können wir das Format auch explizit angeben:

```
df2['last_review_date'] = pd.to_datetime(
  df2['last review'],
  format='%m/%d/%Y'
)
```

[1] Ein weiterer Vorteil des ISO-Formats ist, dass es sich auch als String alphabetisch sinnvoll sortieren lässt, da die Zahlen in der Reihenfolge der Größe angeordnet sind.

```
df2.sort_values('last review')
```

```
                   NAME  price last review last_review_date
15  West Village Nes...  578.0  10/31/2018       2018-10-31
11  Beautiful 1br on...  606.0   6/22/2019       2019-06-22
12  Central Manhatta...  714.0   6/23/2019       2019-06-23
13  Lovely Room 1, G...  580.0   6/24/2019       2019-06-24
10  Cute & Cozy Lowe...  319.0    6/9/2019       2019-06-09
14  Wonderful Guest ...  149.0    7/5/2019       2019-07-05
```

Im Argument `format` können wir verschiedene Codes verwenden, um das Format zu spezifizieren. Der String enthält also die zwei `/` Zeichen, die wir in den Daten haben, und die Codes `%m`, `%d` und `%Y`, die für Monat, Tag und Jahr stehen. Die Codes sind in der Dokumentation aufgelistet. So steht `%Y` z.B. für ein vierstelliges Jahr, `%y` für ein zweistelliges Jahr, `%m` für einen zweistelligen Monat, `%M` aber wiederum für Minuten. Auch dies ist wie schon Regex zum Arbeiten mit Strings ein Beispiel für eine Art separater Sprache, die man nicht unbedingt lernen, aber zumindest nachschlagen muss, um effektiv mit Pandas arbeiten zu können.

`category`

Der `category` Datentyp ist ein spezieller Datentyp, der für kategorische Variablen verwendet wird. Kategorische Variablen sind Variablen, die nur eine begrenzte Anzahl von diskreten Werten annehmen können. Ein Beispiel wäre z.B. die Spalte `room_type` in unserem AirBnB Datensatz. Diese Spalte hat nur vier verschiedene Werte: `Entire home/apt`, `Hotel room`, `Private room` und `Shared room`.

```
room_types = df['room type'].unique()
room_types
```

```
array(['Private room', 'Entire home/apt', 'Shared room', 'Hotel room'],
      dtype=object)
```

Um die Spalte in einen kategorischen Datentyp umzuwandeln, können wir die `astype()` Methode verwenden. Der Vorteil von kategorischen Variablen ist, dass sie weniger Speicherplatz benötigen und schneller zu verarbeiten sind. Das liegt daran, dass Pandas die Werte als Zahlen speichert und eine separate Tabelle mit den zugehörigen Werten führt. So wird z.B. `Entire home/apt` als `0`, `Hotel room` als `1`, `Private room` als `2` und `Shared room` als `3` gespeichert. Das bedeutet demnach auch, dass diese diskreten Stufen/Level nun immer in der entsprechenden Reihenfolge sortiert werden, was z.B. bei einer Abbildung von Vorteil sein kann.

```
#
df['room type']
```

```
0            Private room
1         Entire home/apt
2            Private room
               ...
102596       Private room
102597    Entire home/apt
102598    Entire home/apt
Name: room type, Length: 102599, dtype: object
```

```
df['room type'] = df['room type'].astype('category')
df['room type']
```

```
0            Private room
1         Entire home/apt
2            Private room
               ...
102596       Private room
102597    Entire home/apt
102598    Entire home/apt
Name: room type, Length: 102599, dtype: category
Categories (4, object): ['Entire home/apt', 'Hotel room', 'Private room',
'Shared room']
```

Wie man sieht hat sich der Inhalt der Spalte nicht wirklich verändert, doch der Datentyp ist nun `category`. Außerdem wird deshalb auch `Categories (4, object): ['Entire home/apt', 'Hotel room', 'Private room', 'Shared room']` angezeigt, was bedeutet, dass es vier Kategorien gibt und diese in der genannten Reihenfolge sortiert sind. Wie man sieht, werden die Stufen standardmäßig alphabetisch sortiert. Wollen wir eine andere Reihenfolge, können wir dies wie folgt mit `.cat.set_categories()` tun. Außerdem gibt es die Möglichkeit, die Kategorien mit `ordered=True` als geordnet zu definieren, was bedeutet, dass die Reihenfolge der Kategorien eine signifikante Rolle spielt:

```
meine_stufen = ['Shared room', 'Private room', 'Hotel room', 'Entire home/
apt']
```

```
df['room type'] = df['room type'].cat.set_categories(
  meine_stufen,
  ordered=False
)
```

```
df['room type']
```

```
0            Private room
1         Entire home/apt
2            Private room
               ...
102596       Private room
102597    Entire home/apt
102598    Entire home/apt
Name: room type, Length: 102599, dtype: category
Categories (4, object): ['Shared room', 'Private room', 'Hotel room', 'Entire
home/apt']
```

```
df['room type'] = df['room type'].cat.set_categories(
  meine_stufen,
  ordered=True
)

df['room type']
```

```
0            Private room
1         Entire home/apt
2            Private room
               ...
102596       Private room
102597    Entire home/apt
102598    Entire home/apt
Name: room type, Length: 102599, dtype: category
Categories (4, object): ['Shared room' < 'Private room' < 'Hotel room' <
'Entire home/apt']
```

Wenn man `ordered=True` setzt, erklärt man, dass die Reihenfolge der Kategorien eine signifikante Rolle spielt. Man erkennt das im Output daran, dass zwischen den Stufen ein `<` anstelle eines `,` steht. Es bedeutet, dass die Kategorien nicht nur eine Reihe von verschiedenen Werten darstellen, sondern dass zwischen den Werten eine spezifische, sinnvolle Ordnung existiert. Diese wäre hier z.B. der Fall, aber auch bei anderen Variablen wie z.B. Schulnoten oder Kleidergrößen. Nicht der Fall wäre es z.B. bei Farben (wenn man nicht gerade auf Wellenlängen oder Helligkeit abzielt), Geschlechtern oder bei den Namen von Ländern.

So oder so werden die Kategorien nun in der Reihenfolge angezeigt, die wir angegeben haben, wenn wir jedoch zusätzlich `ordered=True` setzen, können wir auch mit den

Kategorien rechnen. So können wir z.B. Operationen wie < oder > auf den kategorischen Spalten durchführen, was bei nicht-geordneten Kategorien nicht möglich ist.

Man kann übrigens auch direkt beim Umwandeln in den kategorischen Datentyp die Reihenfolge angeben, indem man die Kategorien als Liste übergibt und `pd.Categorical()` nutzt:

```
farben = ['Rot', 'Gelb', 'Blau']
farb_df = pd.DataFrame({'Farbe': farben})

# Umwandlung in eine kategorische Variable ohne Ordnung
farb_df['Farbe_unordered'] = pd.Categorical(
  farb_df['Farbe'],
  categories=farben,
  ordered=False
)

# Umwandlung in eine kategorische Variable mit Ordnung
farb_df['Farbe_ordered'] = pd.Categorical(
  farb_df['Farbe'],
  categories=farben,
  ordered=True
)

farb_df
```

```
  Farbe Farbe_unordered Farbe_ordered
0   Rot             Rot           Rot
1  Gelb            Gelb          Gelb
2  Blau            Blau          Blau
```

```
farb_df[farb_df['Farbe_ordered'] > 'Rot']
```

```
  Farbe Farbe_unordered Farbe_ordered
1  Gelb            Gelb          Gelb
2  Blau            Blau          Blau
```

```
farb_df[farb_df['Farbe_unordered'] > 'Rot']
```

```
TypeError: Unordered Categoricals can only compare equality or not
```

Im Endeffekt sollten wir also immer dann, wenn wir wissen, dass eine Spalte nur eine begrenzte Anzahl von diskreten Werten annehmen kann, diese in einen kategorischen

Datentyp umwandeln. Dies ist nicht nur effizienter, sondern auch sicherer, da wir so sicherstellen, dass keine falschen Werte eingegeben werden können.

Weitere Ressourcen

- Pandas Tutorial #17 - DateTime (Python für Data Science)
- Pandas Tutorial #18 - Datum und Uhrzeit als Index (Python für Data Science)
- Pandas Tutorial #11 - Mehr zu Kategorien (Python für Data Science)
- Die Beispiele in der `pandas.Series.cat` Dokumentation

Übungen

Füge dem AirBnB Datensatz eine neue Spalte `servicegebuehr_eur` hinzu, die die Servicegebühr der Unterkunft (=Spalte `service fee`) als numerischen Wert und in Euro angibt. Nimm dazu an, dass 1,00 Euro genau 1,07 Dollar entspricht. Filtere daraufhin nur die Unterkünfte mit einer Servicegebühr von höchstens 15 Euro heraus.

- Von den insgesamt 102599 Unterkünften haben ____ eine Servicegebühr von maximal 15 Euro.

Erzeuge eine neue Spalten `preiskategorie`, die die Servicegebühren der Unterkünfte in die Kategorien “günstig”, “mittel” und “teuer” einteilt. Die Kategorien sollen dabei folgendermaßen definiert sein:

- “günstig” für Servicegebühren unter 20 Euro
- “mittel” für Servicegebühren ab 20 und bis 50 Euro
- “teuer” für Servicegebühren über 50 Euro

Die erzeugte Spalte soll am Ende als kategorischer Datentyp vorliegen und die Stufen dabei in der Reihenfolge “günstig”, “mittel”, “teuer” sortiert sein.

- (A) Geschafft

Für ein Szenario wie das obige, in dem wir Kategorien basierend auf Schwellenwerten definieren, gibt es auch eine spezielle Methode in Pandas, die `.cut()` Methode. Diese Methode ist sehr mächtig und kann sogar auch mit Zeitstempeln umgehen. Mache dich selbst mit dieser Methode vertraut und versuche die obige Übung damit - anstelle mit `case_when()` - zu lösen. Ein guter Startpunkt sind die Beispiele in der Dokumentation hier und hier oder auch dieses Youtube Video.

- (A) Geschafft