# Data Cleaning for Customer Segmentation

Module 1 | Chapter 3 | Notebook 3

---

In the last lesson you learned about the k-Means algorithm. We want to use this to divide customers of an online retailer into different segments. However, we can't start using it immediately, because the data is not in the format we need it in yet. This lesson is about preparing and understanding the data. By the end of this lesson you will be able to:

- Aggregate data effectively with `my_df.groupby().agg()`
- Sum up DataFrames with different lengths
- Store DataFrames directly

---

## Importing and cleaning the data

**Scenario:** You work for an online retailer that sells various gift items. The company is mainly aimed at business customers. Your customer base is also very diverse. In order to better address the people using your online platform, the marketing department would like to get more insight into customer behavior. The customer base should be divided into groups based on orders they have placed to date. It is not yet clear exactly what characteristics the customers will be grouped by.

However, the customers in the groups should have something in common by the end of the project. The marketing department can then use this common ground to tailor its campaigns to the most important customer groups.

You will be provided with data containing all the orders and cancellations in a one year period, in order to gain a first insight. The data is stored in the files *online_retail_data.csv* and *online_retail_cancellations_data.csv*.

First you need to put the data in a `DataFrame` called `df`. Then print the first 5 rows of the `DataFrame` and the number of rows and columns. Remember to import `pandas`.

In [1]:
```
import pandas as pd
df = pd.read_csv('online_retail_data.csv')
df.head()
```

Out[1]:

| | InvoiceNo | StockCode | Description | Quantity | InvoiceDate | UnitPrice | CustomerID | Country |
|---|---|---|---|---|---|---|---|---|
| **0** | 536367 | 84879 | ASSORTED COLOUR BIRD ORNAMENT | 32 | 2010-12-01 08:34:00 | 1.69 | 13047.0 | United Kingdom |
| **1** | 536367 | 22745 | POPPY'S PLAYHOUSE BEDROOM | 6 | 2010-12-01 08:34:00 | 2.10 | 13047.0 | United Kingdom |
| **2** | 536367 | 22748 | POPPY'S PLAYHOUSE KITCHEN | 6 | 2010-12-01 08:34:00 | 2.10 | 13047.0 | United Kingdom |
| **3** | 536367 | 22749 | FELTCRAFT PRINCESS CHARLOTTE DOLL | 8 | 2010-12-01 08:34:00 | 3.75 | 13047.0 | United Kingdom |
| **4** | 536367 | 22310 | IVORY KNITTED MUG COSY | 6 | 2010-12-01 08:34:00 | 1.65 | 13047.0 | United Kingdom |

We have almost 400000 entries with 8 columns each. It's already possible to work out the meaning of the columns and values in them. shape

Now repeat this for the data in *online_retail_cancellations_data.csv*. Name the corresponding `DataFrame` `df_cancel`.

In [2]:
```
df_cancel = pd.read_csv('online_retail_cancellations_data.csv')
df_cancel.shape
```

Out[2]:
```
(6156, 8)
```

We have just over 6000 entries in the cancellations. They are structured in the same way as the orders, but `'Quantity'` shows negative values. That makes sense, as the items were returned.

The data shows individual ordered items. The following data dictionary explains what the columns represent:

| | Column number | Column name | Type | Description |
|---|---|---|---|---|
| 0 | | `'InvoiceNo'` | categorical/nominal | the order number |
| 1 | | `'StockCode'` | categorical/nominal | the item number |
| 2 | | `'Description'` | text ( `str` ) | Description of the item |
| 3 | | `'Quantity'` | continuous ( `int` ) | Number of ordered items (negative for cancellations) |
| 4 | | `'InvoiceDate'` | continuous ( `datetime` ) | time and date of the order |
| 5 | | `'UnitPrice'` | continuous ( `float` ) | item price in GBP |
| 6 | | `'CustomerID'` | categorical/nominal | customer number |

| Column number | Column name | Type | Description |
|---|---|---|---|
| | | ( `int` ) | |
| 7 | `'Country'` | categorical/nominal | The country the order came from |

Do the categories in the `DataFrame` match those in the data dictionary? Are there any missing values? Print the basic information of `df` to check this.

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398210 entries, 0 to 398209
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   InvoiceNo    398210 non-null  int64  
 1   StockCode    398210 non-null  object 
 2   Description  396771 non-null  object 
 3   Quantity     398210 non-null  int64  
 4   InvoiceDate  398210 non-null  object 
 5   UnitPrice    398210 non-null  float64
 6   CustomerID   264370 non-null  float64
 7   Country      398210 non-null  object 
dtypes: float64(2), int64(2), object(4)
memory usage: 24.3+ MB
```

You may have noticed that `'CustomerID'` is a `float`, but it should actually be an `int`. If an `int` column has missing values, it is treated as a `float` column, because `NaN` is a `float`, so `'CustomerID'` could contain missing values. Check both DataFrames.

In [4]:
```
df.isna().sum()
```

Out[4]:
```
InvoiceNo          0
StockCode          0
Description      1439
Quantity           0
InvoiceDate        0
UnitPrice          0
CustomerID     133840
Country            0
dtype: int64
```

There are more than 133000 missing values in `'CustomerID'`! This is not good news. Since we are interested in the customers, we cannot use the lines with the missing values in this column. In this kind of situation, it can be worthwhile to ask if the data is still available somewhere. But for now we will have to live with it. So remove the rows with the missing values in both DataFrames. Don't forget to assign the output to the variable names again. Then check if there are still values missing.

In [5]:
```
df = df.dropna(subset=['CustomerID'])
df.isna().sum()
```

Out[5]:
```
InvoiceNo      0
StockCode      0
Description    0
Quantity       0
InvoiceDate    0
UnitPrice      0
CustomerID     0
Country        0
dtype: int64
```

Now all the missing values are gone - even those that were in `'Description'`. Apparently values were only missing there, when they were also missing in `'CustomerID'`. But the categories are not stored as such in the `'DataFrame'`. If you like, you can convert them into categories in both DataFrames in the following code cell to save some memory space.

**Important:** Don't convert `'CustomerID'` into a `'category'` column. We will use this variable to group the data later on. This means that it is used for the row names of a new `DataFrame`. However, using a `'category'` as a row name means that we cannot *slice* the DataFrame. This makes it more complicated to look at the data in detail. This is why the `int` data type is much suitable for `'CustomerID'`.

In [6]:
```
df.loc[:, 'StockCode'] = df.loc[:, 'StockCode'].astype('category')
df.loc[:, 'Country'] = df.loc[:, 'Country'].astype('category')
df.loc[:, 'InvoiceDate'] = pd.to_datetime(df.loc[:, 'InvoiceDate'])
df.loc[:, 'Description'] = df.loc[:, 'Description'].astype(str)

df_cancel.loc[:, 'StockCode'] = df_cancel.loc[:, 'StockCode'].astype('category')
df_cancel.loc[:, 'Country'] = df_cancel.loc[:, 'Country'].astype('category')
df_cancel.loc[:, 'InvoiceDate'] = pd.to_datetime(df_cancel.loc[:, 'InvoiceDate'])
df_cancel.loc[:, 'Description'] = df_cancel.loc[:, 'Description'].astype(str)
```

`'Description'` has the `object` data type, as is usual for *strings* in DataFrames. We should definitely convert the `'InvoiceDate'` column. Its entries are currently being treated as `str`. So we have no access to the date functionalities. Convert them into a `datetime` in both DataFrames. Use the `pd.to_datetime()` function that you used in the last chapter.

In [7]:
```
df.head()
```

Out[7]:

| | InvoiceNo | StockCode | Description | Quantity | InvoiceDate | UnitPrice | CustomerID | Country |
|---|---|---|---|---|---|---|---|---|
| **0** | 536367 | 84879 | ASSORTED COLOUR BIRD ORNAMENT | 32 | 2010-12-01 08:34:00 | 1.69 | 13047.0 | United Kingdom |
| **1** | 536367 | 22745 | POPPY'S PLAYHOUSE BEDROOM | 6 | 2010-12-01 08:34:00 | 2.10 | 13047.0 | United Kingdom |
| **2** | 536367 | 22748 | POPPY'S PLAYHOUSE KITCHEN | 6 | 2010-12-01 08:34:00 | 2.10 | 13047.0 | United Kingdom |
| **3** | 536367 | 22749 | FELTCRAFT PRINCESS CHARLOTTE DOLL | 8 | 2010-12-01 08:34:00 | 3.75 | 13047.0 | United Kingdom |
| **4** | 536367 | 22310 | IVORY KNITTED MUG COSY | 6 | 2010-12-01 08:34:00 | 1.65 | 13047.0 | United Kingdom |

Print the data types again. Are they now what you would like them to be?

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 264370 entries, 0 to 398209
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype         
---  ------       --------------   -----         
 0   InvoiceNo    264370 non-null  int64         
 1   StockCode    264370 non-null  category      
 2   Description  264370 non-null  object        
 3   Quantity     264370 non-null  int64         
 4   InvoiceDate  264370 non-null  datetime64[ns]
 5   UnitPrice    264370 non-null  float64       
 6   CustomerID   264370 non-null  float64       
 7   Country      264370 non-null  category      
dtypes: category(2), datetime64[ns](1), float64(2), int64(2), object(1)
memory usage: 15.0+ MB
```

**Congratulations:** Now you know the data and have cleaned it. The missing values would have had a major impact on the analysis. It's good that you noticed and removed them so early in the project. Next, you will prepare the data so that we can use it for an initial analysis.

# Preparing data

The data is currently available in such a way that it describes individual items. To segment customers, we need to aggregate the data to describe individual customers. At the end we will have a `DataFrame` with rows representing customers and columns representing the following customer properties:

- Total number of orders

- Total revenue
- Average revenue per order
- Average quantity per order
- Average price per item bought

First let's calculate the turnover for each item. Multiply the `'Quantity'` and `'UnitPrice'` columns to do this. Store the result as a new column called `'Revenue'`. Do this for both DataFrames.

In [9]:
```
df.loc[:,'Revenue'] = df.loc[:,'Quantity'] * df.loc[:,'UnitPrice']
df_cancel.loc[:,'Revenue'] = df_cancel.loc[:,'Quantity'] * df_cancel.loc[:,'UnitPrice'
```

Now we have to separate the data according to the customer numbers. To do this, group `df` based on the `'CustomerID'` column. Store the resulting *groupby* object under the name `groups`.

In [10]:
```
groups = df.groupby('CustomerID')
pd.DataFrame(groups).iloc[:,1][0]
```

Out[10]:

| | InvoiceNo | StockCode | Description | Quantity | InvoiceDate | UnitPrice | CustomerID | Country | R |
|---|---|---|---|---|---|---|---|---|---|
| **48489** | 541431 | 23166 | MEDIUM CERAMIC TOP STORAGE JAR | 74215 | 2011-01-18 10:01:00 | 1.04 | 12346.0 | United Kingdom | |

Now we will aggregate the groups and reassemble them in a `DataFrame`. For this we can use the `my_groups.agg()` method. You can tell it specifically which column should be aggregated with which function. We'll pass it a `dict` to do that. The *keys* correspond to the column names and the *values* are the corresponding functions. Functions that we already know as methods for aggregating DataFrames (such as `my_df.mean()`) are passed as *strings*, and `pandas` will then recognize them correctly.

First, create three aggregated variables for each customer: total revenue, total quantity of items ordered, and number of orders per customer. To do this, you need a `dict` that uses the column names `'Revenue'`, `'Quantity'` and `'InvoiceNo'` as *keys*. The corresponding *values* are `'sum'`, `'sum'` und `'nunique'`. `pandas` calculates the total for `'sum'` the number of unique values in a column for `'nunique'`. Each different value is therefore only counted once.

Pass this `dict` to `groups.agg()`. Store the result it as a `DataFrame` named `df_customers` and print the first 5 rows.

In [11]:
```
df_customers = groups.agg({'InvoiceNo': 'nunique', 'Revenue': 'sum', 'Quantity':'sum'
df_customers.head()
```

Out[11]:

| CustomerID | InvoiceNo | Revenue | Quantity |
|---|---|---|---|
| **12346.0** | 1 | 77183.60 | 74215 |
| **12347.0** | 7 | 4310.00 | 2458 |
| **12348.0** | 4 | 1437.24 | 2332 |
| **12349.0** | 1 | 1457.55 | 630 |
| **12353.0** | 1 | 89.00 | 20 |

Each row now represents a customer. The person with customer number `12346` only placed one order with a total revenue of `77183.60` GBP. We have created a `DataFrame` with the following outline:

| Column number | Column name | Type | Description |
|---|---|---|---|
| 0 | `'Revenue'` | continuous ( `float` ) | total revenue in GBP |
| 1 | `'Quantity'` | continuous ( `int` ) | total number of items purchased |
| 2 | `'InvoiceNo'` | continuous ( `float` ) | order number |

What does this look like for cancellations? Repeat the grouping and aggregation steps to find out. Name the resulting objects `groups_cancel` and `df_cancel_customers`.

In [12]:
```
groups_cancel  = df_cancel.groupby('CustomerID')
df_cancel_customers = groups_cancel.agg({'InvoiceNo': 'nunique', 'Revenue': 'sum', 'Qu
df_cancel_customers.head()
```

Out[12]:

| CustomerID | InvoiceNo | Revenue | Quantity |
|---|---|---|---|
| **12346.0** | 1 | -77183.60 | -74215 |
| **12362.0** | 3 | -71.65 | -17 |
| **12375.0** | 1 | -2.08 | -1 |
| **12380.0** | 1 | -4.25 | -1 |
| **12384.0** | 1 | -19.11 | -5 |

It looks like the order for customer number `12346` was completely cancelled! The values in `'Revenue'` and `'Quantity'` are the same, only negative. We could also generate new features from the cancellations. However, it will be enough just to deduct the cancellations from the orders. Sum up the `'Revenue'` columns of `df_customers` and `df_cancel_customers`. We are about to run into a problem, which is why we will use `my_df.copy()` for the calculation, so we don't accidentally change our `DataFrame`. Just run the next code cell and look at the result.

In [13]:
```
add_revenues = df_customers.copy().loc[:, 'Revenue'] + df_cancel_customers.copy().loc[
add_revenues.head()
```

Out[13]:
```
CustomerID
12346.0    0.0
12347.0    NaN
12348.0    NaN
12349.0    NaN
12353.0    NaN
Name: Revenue, dtype: float64
```

As you can see, many rows could not be offset against each other and produced `NaN` as a result.

**Important:** `pandas` uses the row names for arithmetic with DataFrames and Series, to assign the values correctly. This is generally very practical. However, if row names are missing in a variable, they are created for the calculation and filled with `np.nan`. This leads to the fact that the result of the calculation is also `np.nan`! To avoid this, you have to use the `my_df.add()` method instead of `+`. If you pass it a `DataFrame`, for example `other_df`, it calculates `my_df + other_df`. But it has the useful `fill_value` parameter. You can use this to specify an alternative value for `np.nan`. Most of the time, and so here too, it makes sense to set `fill_value=0`. The values of the missing row names are then treated as 0.

Sum up the `'Revenue'` columns of `df_customers` and `df_cancel_customers`. Do the same with the `'Quantity'` columns. Store the results in the corresponding columns of `df_customers` and print the first 5 rows.

In [14]:
```
df_customers.loc[:, 'Revenue'] =  df_customers.loc[:, 'Revenue'].add(df_cancel_custome
df_customers.loc[:, 'Quantity'] = df_customers.loc[:, 'Quantity'].add(df_cancel_custom
df_customers.head()
```

Out[14]:

| CustomerID | InvoiceNo | Revenue | Quantity |
|---|---|---|---|
| **12346.0** | 1 | 0.00 | 0.0 |
| **12347.0** | 7 | 4310.00 | 2458.0 |
| **12348.0** | 4 | 1437.24 | 2332.0 |
| **12349.0** | 1 | 1457.55 | 630.0 |
| **12353.0** | 1 | 89.00 | 20.0 |

Now we can see that the turnover for customer number `12346` is exactly 0, because the whole order was cancelled. Are there negative values in the `'Revenue'` column?

In [15]:
```
mask = df_customers.loc[:, 'Revenue'] <0
df_customers.loc[mask, :]
```

Out[15]:

| CustomerID | InvoiceNo | Revenue | Quantity |
|---|---|---|---|
| **16546.0** | 2 | -95.93 | -303.0 |
| **17548.0** | 1 | -141.48 | -132.0 |

Two customer numbers have negative values. This is because at the beginning of the existing period, these customers cancelled orders that they placed before the period. To keep our analysis simple, we'll remove the special cases with values of 0 or below. Remove them and store the result as `df_customers` again. Do the same for the column `Quantity` column, because we have also summed this up with negative values.

In [18]:
```
#mask = df_customers.loc[:, 'Revenue'] <0
mask = df_customers.loc[:, 'Revenue'] > 0
df_customers = df_customers.loc[mask, :]

mask = df_customers.loc[:, 'Quantity'] > 0
df_customers = df_customers.loc[mask, :]
df_customers
```

Have the negative values now disappeared?

In [21]:
```
mask = df_customers.loc[:, 'Revenue'] <= 0
df_customers.loc[mask, :]
```

Out[21]:

| CustomerID | InvoiceNo | Revenue | Quantity |
|---|---|---|---|

Once you have cleaned the data again, we can calculate the average revenue per order, the average quantity per order and the average price per item purchased for each customer. To do this, you just need to divide the corresponding columns. Calculate these values and store them in the `'RevenueMean'`, `'QuantityMean'` and `'PriceMean'` columns in `df_customers`. Then print the first five rows again.

In [27]:
```
RevenueMean = df_customers.loc[:, 'Revenue'] / df_customers.loc[:, 'InvoiceNo']
QuantityMean = df_customers.loc[:, 'Quantity'] / df_customers.loc[:, 'InvoiceNo']
PriceMean = df_customers.loc[:, 'Revenue'] / df_customers.loc[:, 'Quantity']
df_customers.info()
print('RevenueMean', RevenueMean.head())
print('QuantityMean', QuantityMean.head())
print('PriceMean', PriceMean.head())
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 2869 entries, 12347.0 to 18281.0
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   InvoiceNo  2869 non-null   int64  
 1   Revenue    2869 non-null   float64
 2   Quantity   2869 non-null   float64
dtypes: float64(2), int64(1)
memory usage: 89.7 KB
RevenueMean CustomerID
12347.0     615.714286
12348.0     359.310000
12349.0    1457.550000
12353.0      89.000000
12354.0    1079.400000
dtype: float64
QuantityMean CustomerID
12347.0    351.142857
12348.0    583.000000
12349.0    630.000000
12353.0     20.000000
12354.0    530.000000
dtype: float64
PriceMean CustomerID
12347.0    1.753458
12348.0    0.616312
12349.0    2.313571
12353.0    4.450000
12354.0    2.036604
dtype: float64
```

The first row has the row name `12347` and the values `615.714286`, `351.142857` and `1.753458` for the columns `'RevenueMean'`, `'QuantityMean'` and `'PriceMean'`.

Each row represents one customer. Before we save our `DataFrame` in what's called a *pickle*, we should take a look at the data types in each column. You will notice that `'Quantity'` has now been converted to a decimal number because of the calculations we performed in our data analysis. This doesn't make much sense in our example: We should therefore correct the data type of `'Quantity'` before saving and convert it to an `int` by using `my_series.astype()`.

In [35]:
```
df_customers.loc[:, 'Quantity'] = df_customers.loc[:, 'Quantity'].astype('int')

df_customers.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 2869 entries, 12347.0 to 18281.0
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   InvoiceNo  2869 non-null   int64  
 1   Revenue    2869 non-null   float64
 2   Quantity   2869 non-null   int64  
dtypes: float64(1), int64(2)
memory usage: 89.7 KB
```

The following data dictionary explains what the columns represent:

| Column number | Column name | Type | Description |
|---|---|---|---|
| 0 | `'Revenue'` | continuous ( `float` ) | total revenue in GBP |
| 1 | `'Quantity'` | continuous ( `int` ) | total number of items purchased |
| 2 | `'InvoiceNo'` | continuous ( `int` ) | the order number |
| 3 | `'RevenueMean'` | continuous ( `float` ) | average revenue per order |
| 4 | `'QuantityMean'` | continuous ( `float` ) | average number of items per order |
| 5 | `'PriceMean'` | continuous ( `float` ) | average item price |

So far you have prepared the numerical features of the customers. In the following lesson you will create additional features from the dates. As there is a lot of work involved in data cleaning and preparation, it's best if you save `df_customers`. We could use the *CSV* format for this. However, then you'd have to convert the data types of the columns again after importing it again. To store and read objects in Python exactly as they are currently stored, you can use use a *pickle*. This is a file format that was created specifically for data structures in Python. This doesn't just allow you to store DataFrames in this way, but also entire, trained machine learning models.

`pandas` supports saving as a *pickle* with the method `my_df.to_pickle()`. This takes the storage path as an argument. In our case this is just the file name. The conventional file extension to use for this is `'.p'`. However, this has no technical relevance and is only for our benefit as users. `pandas` recognizes that it's a *pickle* when you import it.

Save `df_customers` as a *pickle* with the name `'customer_data.p'`.

In [36]:
```
df_customers.to_pickle('customer_data.p')
```

**Congratulations:** You have cleaned and prepared the order data. Originally, the data was only available based on the ordered items. Canceled orders had the same format, but were stored in a separate file. You aggregated and merged this data to create data based on customers. So far you have used numerical values for this purpose. In the following lessons you will create additional features based on the dates.

**Remember:**

- Cleaning and preparing data is a very important part of a data scientist's work
- Add two `pandas` objects that contain `NaN` values together with `my_Series_or_DataFrame.add()`
- Aggregate data with `my_df.groupby().agg()`.
- Store DataFrames as a *pickle* by using `my_df.to_pickle('filename')`

---

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

---

Found a mistake? Contact Support at support@stackfuel.com.

---

The data was published here first: Daqing Chen, Sai Liang Sain, and Kun Guo, Data mining for the online retail industry: A case study of RFM model-based customer segmentation using data mining, Journal of Database Marketing and Customer Strategy Management, Vol. 19, No. 3, pp. 197-208, 2012