DataScientest • com

Pandas for Data Science

Data Cleaning and Missing Values Management

Introduction

> **Data cleaning** and **missing values management** (called **NaN** or **NA**) are two essential steps before any analysis on a database.
>
> The objective of this notebook is to detail each of these two steps in order to obtain a clean and easily usable `DataFrame`. Indeed, databases very often present this kind of problem.
>
> For this, we are going to use the `DataFrame` **`transactions`** imported in the previous exercise.

- **(a)** Import the `pandas` module under the name `pd` and load the file `"transactions.csv"` in a `DataFrame` named **`transactions`**. The values in the CSV file are separated by **commas** and the column containing the identifiers is **`'transaction_id'`**.
- **(b)** Display the first 10 rows of `transactions.csv` with the `head` method.

In [2]:

```python
# Insert your code here
import pandas as pd

df = pd.read_csv('transactions.csv', sep=',', index_col='transaction_id')
df1 = df.copy()
df.head(10)
```

Out[2]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt	Store_type
transaction_id									
80712190438	270351	28-02-2014	1.0	1	-5	-772.0	405.300	-4265.300	e-Shop
29258453508	270384	27-02-2014	5.0	3	-5	-1497.0	785.925	-8270.925	e-Shop
51750724947	273420	24-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
93274880719	271509	24-02-2014	11.0	6	-3	-1363.0	429.345	-4518.345	e-Shop
51750724947	273420	23-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
97439039119	272357	23-02-2014	8.0	3	-2	-824.0	173.040	-1821.040	TeleShop
45649838090	273667	22-02-2014	11.0	6	-1	-1450.0	152.250	-1602.250	e-Shop
22643667930	271489	22-02-2014	12.0	6	-1	-1225.0	128.625	-1353.625	TeleShop
79792372943	275108	22-02-2014	3.0	1	-3	-908.0	286.020	-3010.020	MBR
50076728598	269014	21-02-2014	8.0	3	-4	-581.0	244.020	-2568.020	e-Shop

Hide solution

In [3]:

```python
# Import of the pandas module under the name pd
import pandas as pd

# Loading of the transactions database
transactions = pd.read_csv("transactions.csv", sep = ',', index_col = "transaction_id")

# Display of the first 10 rows of transactions
transactions.head(10)
```

Out[3]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt	Store_type
transaction_id									
80712190438	270351	28-02-2014	1.0	1	-5	-772.0	405.300	-4265.300	e-Shop
29258453508	270384	27-02-2014	5.0	3	-5	-1497.0	785.925	-8270.925	e-Shop
51750724947	273420	24-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
93274880719	271509	24-02-2014	11.0	6	-3	-1363.0	429.345	-4518.345	e-Shop
51750724947	273420	23-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
97439039119	272357	23-02-2014	8.0	3	-2	-824.0	173.040	-1821.040	TeleShop
45649838090	273667	22-02-2014	11.0	6	-1	-1450.0	152.250	-1602.250	e-Shop
22643667930	271489	22-02-2014	12.0	6	-1	-1225.0	128.625	-1353.625	TeleShop
79792372943	275108	22-02-2014	3.0	1	-3	-908.0	286.020	-3010.020	MBR
50076728598	269014	21-02-2014	8.0	3	-4	-581.0	244.020	-2568.020	e-Shop

1. Cleaning up a dataset

> In this part we will introduce the methods of the `DataFrame` class that are essential to clean a dataset. These methods can be grouped into three different categories :
>
> > - **Duplicates management** (`duplicated` and `drop_duplicates` methods)
> > - **Modification of the elements** of a `DataFrame` (`replace` , `rename` and `astype` methods)
> > - **Operations** on the values of a `DataFrame` (`apply` method and `lambda` functions)

Managing duplicates (`duplicated` and `drop_duplicates` methods)

Duplicates are identical entries that appear **more than once** in a dataset.

When we first discover a dataset it is very important to **check up front** that there are no duplicates. The presence of duplicates will generate **errors** in the computation of statistics or the plotting of graphs.

Let **`df`** be the following `DataFrame` :

	Age	Gender	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

The presence of duplicates is checked using the **`duplicated`** method of a `DataFrame` :

```
# We identify the rows containing duplicates
df.duplicated()

>>> 0 False
>>> 1 False
>>> 2 False
>>> 3 True
```

This method returns `Series` object from `pandas` , which is equivalent to the column of a `DataFrame` . The `Series` object tells us for each row wether it is a duplicate.

In this example, the result of the `duplicated` method informs us that **the row with index 3 is a duplicate**. Indeed, it is the **exact copy** of the row with index 1.

Since the `duplicated` method returns an object of the `Series` class, we can apply the **`sum`** method to it in order to count the number of duplicates:

```
# To calculate the sum of boolean values, we consider that True is worth 1 and False is worth 0.
print(df.duplicated().sum())
>>> 1
```

The method of the `DataFrame` class used to remove duplicates is **`drop_duplicates`** . Its header is as follows:

```
drop_duplicates(subset, keep, inplace)
```

- The `subset` parameter indicates the column(s) to consider in order to identify and remove duplicates. By default, **`subset = None`** namely we consider **all** the columns of the `DataFrame` .
- The `keep` parameter indicates which entry should be kept :
  - **`'first'`** : We keep the **first** occurrence.
  - **`'last'`** : We keep the **last** occurrence.
  - **`False`** : We do not keep **any** occurrence.
  - By default, **`keep = 'first'`** .
- The **`inplace`** parameter (very common in the methods of the `DataFrame` class), specifies whether you modify **directly** the `DataFrame` (in this case `inplace = True`) or if the method returns a **copy** of the `DataFrame` (`inplace = False`). A method applied with the argument `inplace = True` is **irreversible**. By default, `inplace = False` .

You have to be very careful when using the `inplace` parameter. A good practice is to forget this parameter and assign the `DataFrame` returned by the method to a **new** `DataFrame` .

The `keep` parameter is the one that is most often specified. Indeed, a database can have duplicates created on different dates. We will then specify the value of the `keep` argument to keep only the most recent entries, for example.

Let us go back to the `df` `DataFrame` :

	Age	Gender	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

We illustrate `df` with the following illustration :

	Age	Sex	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

We illustrate in the following examples the entries that are **deleted** by the `drop_duplicates` method depending on the value of the `keep` parameter:

```
# We keep only the first occurrence of the duplicate
df_first = df.drop_duplicates(keep = 'first')
```

	Age	Sex	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

```python
# We keep only the last occurrence of the duplicate
df_last = df.drop_duplicates(keep = 'last')
```

	Age	Sex	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

```python
# We keep no duplicates
df_false = df.drop_duplicates(keep = False)
```

	Age	Sex	Height
Robert	56	M	174
Mark	23	M	182
Alina	32	F	169
Mark	23	M	182

In [8]:

```python
# Insert your code here

df.duplicated()
df.duplicated().sum()
```

Out[8]: 112

Hide solution

In [9]:

```python
# Counting the number of duplicates
duplicates = transactions.duplicated().sum()

print ("There are", duplicates, "duplicates in transactions.")
```

```
There are 112 duplicates in transactions.
```

The transactions were recorded in anti-chronological order, i.e. the **first rows** contain the most **recent** transactions and the last rows the oldest transactions.

- **(b)** Eliminate duplicates from the database by keeping only the first occurrence, i.e. the most recent transaction.
- **(c)** Using the **`subset`** and **`keep`** parameters of the `drop_duplicates` method of `transactions`, display the **most recent** transaction for **each category of `prod_cat_code`**. To do this, you can remove all the duplicates from the `prod_cat_code` column by keeping only the first occurrence.

In [13]:

```python
# Insert your code here

#b) Keep First
df_first = df.drop_duplicates(keep='first')
df_first.duplicated().sum()

#c)
df_prod = df.drop_duplicates(subset=['prod_cat_code'], keep='first')
df_prod
```

Out[13]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt	Store_type
transaction_id									
80712190438	270351	28-02-2014	1.0	1	-5	-772.0	405.300	-4265.300	e-Shop
29258453508	270384	27-02-2014	5.0	3	-5	-1497.0	785.925	-8270.925	e-Shop
51750724947	273420	24-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
93274880719	271509	24-02-2014	11.0	6	-3	-1363.0	429.345	-4518.345	e-Shop
43134751727	268487	20-02-2014	3.0	2	-1	-611.0	64.155	-675.155	e-Shop
25963520987	274829	20-02-2014	4.0	4	3	502.0	158.130	1664.130	Flagship store

Hide solution

In [14]:

```python
transactions = transactions.drop_duplicates(keep = 'first')

transactions.drop_duplicates(subset = ['prod_cat_code'], keep = 'first')
```

Out[14]:

transaction_id	cust_id	tran_date	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt	Store_type
80712190438	270351	28-02-2014	1.0	1	-5	-772.0	405.300	-4265.300	e-Shop
29258453508	270384	27-02-2014	5.0	3	-5	-1497.0	785.925	-8270.925	e-Shop
51750724947	273420	24-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
93274880719	271509	24-02-2014	11.0	6	-3	-1363.0	429.345	-4518.345	e-Shop
43134751727	268487	20-02-2014	3.0	2	-1	-611.0	64.155	-675.155	e-Shop
25963520987	274829	20-02-2014	4.0	4	3	502.0	158.130	1664.130	Flagship store

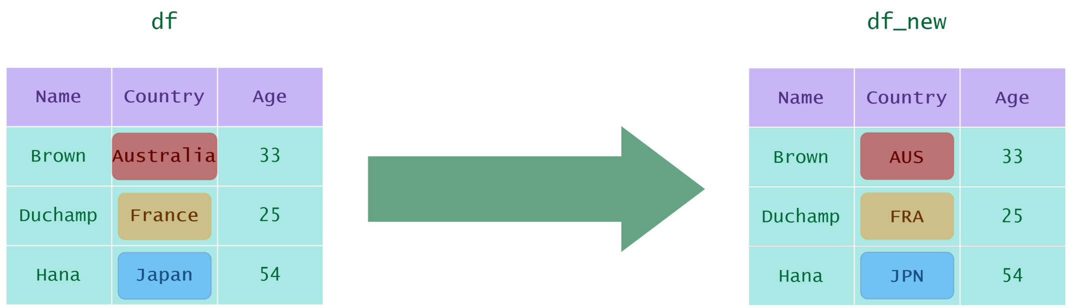
Modification of the elements of a `DataFrame` (`replace` , `rename` and `astype` methods)

The **`replace`** method allows to **replace** one or more values of a column of a `DataFrame` .

Its header is as follows:

```python
replace(to_replace, value, ...)
```

- The `to_replace` parameter contains the value or the list of values **to be replaced**. It can be a list of integers, strings, booleans, etc.
- The `value` parameter contains the value or the list of the substitute **values**. It can also be a list of integers, strings, booleans, etc.




```python
df_new = df.replace(to_replace=['Australia','France','Japan'], value=['AUS', 'FRA', 'JPN'])
```

In addition to modifying the elements of a `DataFrame` , it is possible to **rename** its columns.

This is possible thanks to the **`rename`** method which takes as argument a **dictionary** whose **keys** are the **old** names and the **values** are the **new** names. You must also fill in the argument **`axis = 1`** to specify that the names to rename are those of the columns.

```python
# Creation of the dictionary associating the old names with the new column names
dictionary = {'old_name1': 'new_name1',
              'old_name2': 'new_name2'}

# We rename the variables using the rename method
df = df.rename(dictionary, axis = 1)
```

It is sometimes necessary to modify not only the name of a column but also its **type**.

For example, it is possible that when importing a database, a variable is of type string when in fact it is a numerical variable. Whenever one of the entries in the column is incorrectly recognized, `pandas` will consider that this column is of type string.

This is possible thanks to the **`astype`** method.

The types that we will see most often are:

- `str` : Character string (`'Hello'`).
- `float` : Floating point number (`1.0` , `1.14123`).
- `Int` : Integer (`1` , `1231`)

As for the **`rename`** method, **`astype`** can take as argument a dictionary whose **keys** are the **names of the columns whose type should be modified** and the **values** are the **new types** to assign. This is useful if you want to change the type of several columns at once.

Most often, we will directly select the column whose type should be modified and overwrite it by applying the **`astype`** method to it.

```python
# Method 1: Creation of a dictionary then call to the astype method of the DataFrame
dictionary = {'col_1': 'int',
              'col_2': 'float'}
df = df.astype(dictionary)

# Method 2: Selection of the column and then calling the astype method of a Series
df['col_1'] = df['col_1'].astype('int')
```

❗ These methods also have the `inplace` parameter to perform the operation directly on the `DataFrame` . To be used with great caution.

- If you make a mistake in the next exercise, you can re-import and redo the preprocessing by running the following cell.

In [15]:

```
# Data import
transactions = pd.read_csv("transactions.csv", sep = ',', index_col = "transaction_id")

# Removal of duplicates
transactions = transactions.drop_duplicates(keep = 'first')
```

- **(d)** Import the `numpy` module under the name `np`.
- **(e)** Replace the modalities `['e-Shop', 'TeleShop', 'MBR', 'Flagship store', np.nan]` of the `Store_type` column by the modalities `[1, 2, 3, 4, 0]`.

> The `np.nan` value is the one that encodes a missing value. We will replace this value with `0`.

- **(f)** Convert the type of the columns `Store_type` and `prod_subcat_code` to type `'int'`.
- **(g)** Rename the `'Store_type'`, `'Qty'`, `'Rate'` and `'Tax'` columns with `'store_type'`, `'qty'`, `'rate'` and `'tax'`.

In [54]:

```
# Insert your code here
import numpy as np

#e)
df_first = df.drop_duplicates(keep = 'first')
df_repStore = df_first.replace(['e-Shop', 'TeleShop', 'MBR', 'Flagship store', np.nan], [1, 2, 3, 4, 0])
print(df_repStore.dtypes)
print(80*'-')
#f)
dict1 = {'Store_type' : 'int',
        'prod_subcat_code' : 'int'}

df_repStore.astype(dict1).dtypes
print(df_repStore.dtypes)

#g) First Way (need inplace or imputation)
df_repStore.rename(columns={'Store_type' : 'store_type',
                    'Qty' : 'qty',
                    'Rate' : 'rate',
                    'Tax' : 'tax'}).head()

#--- Second Way
#list1 = df_repStore.columns.str.lower()

#df_repStore.rename(columns = list1)
```

```
cust_id              int64
tran_date           object
prod_subcat_code   float64
prod_cat_code        int64
Qty                  int64
Rate               float64
Tax                float64
total_amt          float64
Store_type           int64
dtype: object
--------------------------------------------------------------------------------
cust_id              int64
tran_date           object
prod_subcat_code   float64
prod_cat_code        int64
Qty                  int64
Rate               float64
Tax                float64
total_amt          float64
Store_type           int64
dtype: object
```

Out[54]:

```
Index(['cust_id', 'tran_date', 'prod_subcat_code', 'prod_cat_code', 'qty',
       'rate', 'tax', 'total_amt', 'store_type'],
      dtype='object')
```

Hide solution

In [55]:

```python
# Data import
transactions = pd.read_csv("transactions.csv", sep = ',', index_col = "transaction_id")

# Removal of duplicates
transactions = transactions.drop_duplicates(keep = 'first')

## Exercise

import numpy as np

# Replacement of values
transactions = transactions.replace(to_replace = ['e-Shop', 'TeleShop', 'MBR', 'Flagship store', np.nan],
                                    value = [1, 2, 3, 4, 0])

# Conversion of column types
new_types = {'Store_type'       : 'int',
             'prod_subcat_code' : 'int'}

transactions = transactions.astype(new_types)

# Renaming the columns
new_names = {'Store_type'   : 'store_type',
              'Qty'         : 'qty',
              'Rate'        : 'rate',
              'Tax'         : 'tax'}

transactions = transactions.rename(new_names, axis = 1)

# Display of the first rows of transactions
transactions.head()
```

Out[55]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type
transaction_id									
80712190438	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	1
29258453508	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	1
51750724947	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	2
93274880719	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	1
51750724947	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	2

Operations on the values of a `DataFrame` (`apply` method and `lambda` functions)

It is often interesting to modify or aggregate the information of the columns of a `DataFrame` using an operation or a function.

These operations can be any type of function **which takes a column** as argument. Thus, the **numpy module is perfectly suited** to perform operations on this type of object.

The method used to perform an operation on a column is the **`apply`** method of a `DataFrame` whose header is:

```
apply(func, axis, ...)
```

where:

- **`func`** is the function to apply to the column.
- **`axis`** is the dimension on which the operation must be applied.

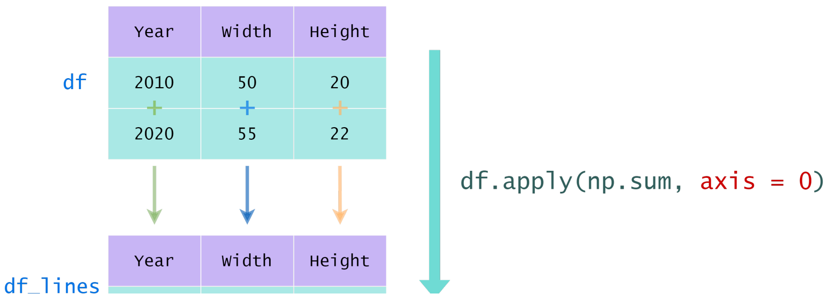
Example: `apply` and `np.sum`

For each column with numerical values, we want to calculate the **sum of all rows**. The `sum` function of `numpy` does this, so we can use it with the `apply` method.

Since we are going to perform an operation on the **rows**, we must therefore specify the argument **`axis = 0`** in the `apply` method.

```
# Sum of the ROWS for each column of df
df_lines = df.apply(np.sum, axis = 0)
```

The result is the following:



In [61]:
```
transactions.tran_date.isna().sum()
```

Out[61]: 0

In [85]:
```
# Insert your code here

#h)

date = '28-02-2014 '
def get_day(X):
    return X.split('-')[0]

def get_month(X):
    return X.split('-')[1]

def get_year(X):
    return X.split('-')[2]


days = transactions['tran_date'].apply(get_day)
months = transactions['tran_date'].apply(get_month)
years = transactions['tran_date'].apply(get_year)

transactions['day'] = days
transactions['month'] = months
transactions['year'] = years

transactions.head()
```

Out[85]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	day	month	year
transaction_id												
80712190438	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	1	28	02	2014
29258453508	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	1	27	02	2014
51750724947	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	2	24	02	2014
93274880719	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	1	24	02	2014
51750724947	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	2	23	02	2014

Hide solution

In [ ]:

```python
# Definition of the functions to apply to the 'tran_date' column
def get_day(date):
    """
    Takes a date as a string argument.

    The date must have the format 'DD-MM-YYYY'.

    This function returns the day (DD).
    """

    # Splitting the string on the '-' character
    splits = date.split('-')

    # We return the first element of the breakdown (day)
    day = splits[0]
    return day

def get_month(date):
    return date.split('-')[1]

def get_year(date):
    return date.split('-')[2]


# Retrieving the day, month and year of each transaction
days = transactions['tran_date'].apply(get_day)
months = transactions['tran_date'].apply(get_month)
years = transactions['tran_date'].apply(get_year)

# Creation of new columns
transactions['day'] = days
transactions['month'] = months
transactions['year'] = years

# Displaying first rows of transactions
transactions.head()
```

The **`apply`** method is very powerful when combined with a **`lambda`** function.

In Python, the keyword **`lambda`** is used to define an **anonymous** function: a function declared without name.

A function **`lambda`** can take any number of arguments, but can only have one expression.

Here is its syntax:

```python
lambda arguments: expression
```

`Lambda` functions allow you to define functions with a very short syntax :

```python
# Example 1
x = lambda a: a + 2
print(x(3))
>>> 5

# Example 2
x = lambda a, b : a * b
print(x(2, 3))
>>> 6

# Example 3
x = lambda a, b, c : a - b + c
print(x(1, 2, 3))
>>> 2
```

Although syntactically different, **`lambda`** functions behave in the same way as regular functions that are declared using the **`def`** keyword.

The classic definition of a function is done with the **`def`** keyword:

```python
def increment(x):
    return x + 1
```

It is also possible to define a function with the keyword **`lambda`** :

```python
increment = lambda x: x + 1
```

The first method is very clean but the advantage of the second is that it can be defined on-the-fly directly **within** the **`apply`** method.

Thus, the previous exercise can be done with a very compact syntax:

```python
transactions['day'] = transactions['tran_date'].apply(lambda date: date.split('-')[0])
```

This kind of syntax is very practical and very often used for cleaning databases.

The `prod_subcat_code` column of `transactions` depends on the `prod_cat_code` column because it identifies a **subcategory** of product. It would make more sense to have the category and subcategory of a product in the same variable.

To do this, we will merge the values of these two columns:

- We will first convert the values of these two columns into strings using the **`astype`** method.
- Then, we will concatenate these strings to have a unique code representing both the category and sub-category. This can be done in the following way:

```python
string1 = "I think"
string2 = "therefore I am."

# Concatenation of the two strings by separating them with a space
print (string1 + " " + string2)
>>> I think therefore I am.
```

To apply a lambda function to an entire row, you must specify the argument **`axis = 1`** in the `apply` method. In the function itself, the columns of the row can be accessed as on a `DataFrame` :

```python
# Computation of the unit price of a product
transactions.apply(lambda row: row['total_ amt']/row['qty'], axis = 1)
```

- **(m)** Using a `lambda` function applied to `transactions` , create a column **`'prod_cat'`** in `transactions` containing the concatenation of the values of `prod_cat_code` and `prod_subcat_code` separated by a hyphen `'-'` . Remember to convert the values to strings.

Displaying this column should yield:

```
transaction_id
80712190438     1-1
29258453508     3-5
51750724947     5-6
93274880719     6-11
51750724947     5-6
               ...
94340757522     5-12
89780862956     1-4
85115299378     6-2
72870271171     5-11
77960931771     5-11
```

In [57]:

```python
import numpy as np
df['prod_subcat_code'].isna().sum()
```

Out[57]: 32

In [68]:

```python
transactions['prod_subcat_code'] = transactions['prod_subcat_code'].replace(['e-Shop', 'TeleShop', 'MBR', 'Flagship store', np.nan]
                                                                         , [1, 2, 3, 4, 0])
transactions['prod_subcat_code'] = transactions['prod_subcat_code'].astype(int)
transactions['prod_subcat_code']
```

Out[68]:

```
transaction_id
80712190438     1
29258453508     5
51750724947     6
93274880719    11
51750724947     6
               ..
94340757522    12
89780862956     4
85115299378     2
72870271171    11
77960931771    11
Name: prod_subcat_code, Length: 22941, dtype: int64
```

In [69]:

```python
# Insert your code here
transactions['prod_cat'] = transactions.apply(lambda row : str(row['prod_cat_code']) + '-' + str(row['prod_subcat_code']),
                                           axis=1)
transactions.head()
```

Out[69]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	prod_cat
transaction_id										
80712190438	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	e-Shop	1-1
29258453508	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	e-Shop	3-5
51750724947	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	5-6
93274880719	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	e-Shop	6-11
51750724947	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	5-6

Hide solution

In [70]:

```python
transactions['prod_cat'] = transactions.astype('str').apply(lambda row: row['prod_cat_code']+'-'+row['prod_subcat_code'],
                                                             axis = 1)

print(transactions['prod_cat'])
```

```
transaction_id
80712190438     1-1
29258453508     3-5
51750724947     5-6
93274880719    6-11
51750724947     5-6
               ...
94340757522    5-12
89780862956     1-4
85115299378     6-2
72870271171    5-11
77960931771    5-11
Name: prod_cat, Length: 22941, dtype: object
```

2. Dealing with missing values

A **missing value** is either:

- An unspecified value.
- A value that does not exist. In general, they result from mathematical calculations having no solution (a division by zero for example).

A missing value appears under the name **NaN** ("**N**ot **a** **N**umber") in a `DataFrame` .

In this part, we will see several methods to:

- **Detect** missing values (`isna` and `any` methods)
- **Replace** these values (`fillna` method)
- **Delete** missing values (`dropna` method)

In one of the previous exercises, we used the `replace` method of `transactions` to replace missing values with `0` . This approach is not rigorous and should not be done in practice.

For this reason, we are going to re-import the raw version of `transactions` to undo the steps we did in the previous exercises.

- **(a)** Run the cell below to re-import `transactions` , remove duplicates and rename its columns.

In [71]:

```python
# Data import
transactions = pd.read_csv("transactions.csv", sep = ',', index_col = "transaction_id")

# Duplicates removal
transactions = transactions.drop_duplicates(keep = 'first')

# Renaming the columns
new_names = {'Store_type'  : 'store_type',
             'Qty'         : 'qty',
             'Rate'        : 'rate',
             'Tax'         : 'tax'}

transactions = transactions.rename(new_names, axis = 1)

transactions.head()
```

Out[71]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type
transaction_id									
80712190438	270351	28-02-2014	1.0	1	-5	-772.0	405.300	-4265.300	e-Shop
29258453508	270384	27-02-2014	5.0	3	-5	-1497.0	785.925	-8270.925	e-Shop
51750724947	273420	24-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop
93274880719	271509	24-02-2014	11.0	6	-3	-1363.0	429.345	-4518.345	e-Shop
51750724947	273420	23-02-2014	6.0	5	-2	-791.0	166.110	-1748.110	TeleShop

Detecting missing values (`isna` and `any` methods)

The `isna` method of a `DataFrame` detects its missing values. This method does not take any arguments.

This method returns the same `DataFrame` whose values are:

- `True` if the original table cell is a missing value (`np.nan`)
- `False` otherwise.

df

Name	Country	Age
NaN	Australia	NaN
Duchamp	France	25

`df_new = df.isna()`

df_new

Name	Country	Age
True	False	True
False	False	False

In [172]:

```python
# Insert your code here

print(transactions.isna().sum())
print('c', 50 * '*')
print(transactions.isna().any(axis = 1).sum())
print('total null values in data frame',21 * '*')
print(transactions.isna().sum(axis = 1).sum())
print('d 1', 50 * '*')
print(transactions.isna().sum().idxmax())
print('d 2' , 50 * '*')
print(transactions.isna().sum()[transactions.isna().sum() == transactions.isna().sum().max()])
print('d 3' , 50 * '*')
df1 = transactions.isna().sum().reset_index(name='null_num')
print(df1[['index', 'null_num']][df1.null_num == df1['null_num'].max()])
print('e', 50 * '*')

transactions[transactions[['rate', 'tax', 'total_amt']].isna().any(axis = 1)]
```

```
cust_id              0
tran_date            0
prod_subcat_code    32
prod_cat_code        0
qty                  0
rate                22
tax                 22
total_amt           22
store_type          53
dtype: int64
c **************************************************
107
total null values in data frame *********************
151
d 1 **************************************************
store_type
d 2 **************************************************
store_type    53
dtype: int64
d 3 **************************************************
        index  null_num
8  store_type        53
e **************************************************
```

Out[172]:

transaction_id	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type
27576087298	270419	9-2-2014	11.0	6	-2	NaN	NaN	NaN	MBR
6472413088	272105	31-01-2014	11.0	6	2	NaN	NaN	NaN	e-Shop
13300797307	272662	12-1-2014	6.0	5	4	NaN	NaN	NaN	TeleShop
63096895453	270519	3-1-2014	3.0	1	1	NaN	NaN	NaN	Flagship store
77032870309	271120	28-11-2013	10.0	6	5	NaN	NaN	NaN	e-Shop
72504986680	270550	18-10-2013	3.0	2	3	NaN	NaN	NaN	Flagship store
47049265058	269588	24-08-2013	8.0	3	3	NaN	NaN	NaN	Flagship store
141709406	266910	7-8-2013	2.0	6	3	NaN	NaN	NaN	Flagship store
37780749229	272135	5-2-2013	4.0	1	2	NaN	NaN	NaN	e-Shop
82258282007	274767	31-01-2013	11.0	6	1	NaN	NaN	NaN	TeleShop
55032993738	268992	12-11-2012	5.0	3	5	NaN	NaN	NaN	TeleShop
52063651015	274590	29-07-2012	1.0	4	4	NaN	NaN	NaN	TeleShop
3065420704	268056	12-7-2012	7.0	5	2	NaN	NaN	NaN	MBR
54589211549	273721	28-02-2012	11.0	6	3	NaN	NaN	NaN	e-Shop
95464182713	272323	9-11-2011	4.0	1	3	NaN	NaN	NaN	MBR
12647081503	267464	7-9-2011	11.0	5	2	NaN	NaN	NaN	e-Shop
75579128061	267116	2-8-2011	3.0	5	4	NaN	NaN	NaN	TeleShop
10720099148	268144	3-7-2011	10.0	3	-2	NaN	NaN	NaN	e-Shop
2851810788	271201	24-04-2011	3.0	5	4	NaN	NaN	NaN	TeleShop
84507124783	269291	2-4-2011	9.0	3	1	NaN	NaN	NaN	e-Shop
37686564846	273928	8-3-2011	3.0	1	3	NaN	NaN	NaN	TeleShop
46557552371	269451	23-02-2011	10.0	3	5	NaN	NaN	NaN	MBR

Hide solution

In [173]:

```python
# Which columns contain NaNs
columns_na = transactions.isna().any(axis = 0)

print(columns_na.sum(), "columns of transactions contain NaNs. \n")

# Which rows contain NaNs
rows_na = transactions.isna().any(axis = 1)

print(rows_na.sum(), "rows of transactions contain NaNs. \n")

# Number of NaNs per column
columns_nbna = transactions.isna().sum(axis = 0)

print ("The column containing the most NaNs is:",  columns_nbna.idxmax())

# Display the first 10 entries containing at least one NaN in 'rate', 'tax' or 'total_amt'
transactions[transactions[['rate', 'tax', 'total_amt']].isna().any(axis = 1)].head(10)

# The three variables are still missing together.
```

```
5 columns of transactions contain NaNs.

107 rows of transactions contain NaNs.

The column containing the most NaNs is: store_type
```

Out[173]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type
transaction_id									
27576087298	270419	9-2-2014	11.0	6	-2	NaN	NaN	NaN	MBR
6472413088	272105	31-01-2014	11.0	6	2	NaN	NaN	NaN	e-Shop
13300797307	272662	12-1-2014	6.0	5	4	NaN	NaN	NaN	TeleShop
63096895453	270519	3-1-2014	3.0	1	1	NaN	NaN	NaN	Flagship store
77032870309	271120	28-11-2013	10.0	6	5	NaN	NaN	NaN	e-Shop
72504986680	270550	18-10-2013	3.0	2	3	NaN	NaN	NaN	Flagship store
47049265058	269588	24-08-2013	8.0	3	3	NaN	NaN	NaN	Flagship store
141709406	266910	7-8-2013	2.0	6	3	NaN	NaN	NaN	Flagship store
37780749229	272135	5-2-2013	4.0	1	2	NaN	NaN	NaN	e-Shop
82258282007	274767	31-01-2013	11.0	6	1	NaN	NaN	NaN	TeleShop

Replacing missing values (`fillna` method)

> The `fillna` method allows you to replace the missing values of a `DataFrame` by **values you want**.
>
> ```python
> # We replace all the NaNs of the DataFrame by zeros
> df.fillna(0)
>
> # We replace the NaNs of each numerical column by the average on this column
> df.fillna(df.mean()) # df.mean() can be replaced by any statistical method.
> ```
>
> It is common to replace missing values of a column containing **numerical** values with **statistics** like:
>
> > - The **mean**: `.mean`
> > - The **median**: `.median`
> > - The **minimum / maximum**: `.min` / `.max` .
>
> For categorical type columns, replace the missing values with:
>
> > - The **mode**, i.e. the most frequent modality: `.mode` .
> > - A **constant** or arbitrary category: `0` , `-1` .
>
> To avoid making replacement errors, it is very important to **select the correct columns** before using the `fillna` method.

If you make mistakes in the following exercise, you can re-import `transactions` using the following cell:

In [193]:

```python
# Data import
transactions = pd.read_csv("transactions.csv", sep = ',', index_col = "transaction_id")

# Removal of duplicates
transactions = transactions.drop_duplicates(keep = 'first')

# Renaming the columns
new_names = {'Store_type' : 'store_type',
             'Qty'        : 'qty',
             'Rate'       : 'rate',
             'Tax'        : 'tax'}

transactions = transactions.rename(new_names, axis = 1)
```

- **(f)** Replace the missing values in **`prod_subcat_code`** column of `transactions` with `-1` .
- **(g)** Determine **the most frequent modality** (the mode) of the **`store_type`** column of `transactions` .
- **(h)** Replace the missing values of the `store_type` column by this modality. The value of this modality is accessed **at index 0** of the `Series` returned by `mode` .
- **(i)** Check that the `prod_subcat_code` and `store_type` columns of `transactions` no longer contain missing values.

In [191]:

```
# Insert your code here

print('f', 50 * '*')
transactions['prod_subcat_code'].fillna(-1, inplace=True)
print(transactions['prod_subcat_code'].isna().sum())


print('g', 50 * '*')
print("The most frequent mode of 'store_type' is:", transactions['store_type'].mode())

print('h', 50 * '*')
store_type_mode_value = transactions['store_type'].mode().values[0]

transactions['store_type'].fillna(store_type_mode_value, inplace=True)

print('i', 50 * '*')
transactions[['prod_subcat_code', 'store_type']].isna().sum()
```

```
f **************************************************
0
g **************************************************
0    e-Shop
dtype: object
h **************************************************
i **************************************************
```

Out[191]:

```
prod_subcat_code    0
store_type          0
dtype: int64
```

Hide solution

In [194]:

```
# Replacing the NaNs of 'prod_subcat_code' by -1
transactions['prod_subcat_code'] = transactions['prod_subcat_code'].fillna(-1)

# Determining the mode of 'store_type'
store_type_mode = transactions['store_type'].mode()
print ("The most frequent mode of 'store_type' is:", store_type_mode[0])

# Replacing the NaNs of 'store_type' by its mode
transactions['store_type'] = transactions['store_type'].fillna(transactions['store_type'].mode()[0])

# Checking that these two columns no longer contain NANs
transactions[['prod_subcat_code', 'store_type']].isna().sum()
```

```
The most frequent mode of 'store_type' is: e-Shop
```

Out[194]:

```
prod_subcat_code    0
store_type          0
dtype: int64
```

Removing missing values (**dropna** method)

The `dropna` method allows you to remove rows or columns containing missing values.

The header of the method is as follows:

```
dropna(axis, how, subset, ..)
```

- The **axis** parameter specifies whether to delete rows or columns (**0** for rows, **1** for columns).
- The **how** parameter lets you specify how the rows (or columns) are deleted:
  - **how = 'any'** : We delete the row (or column) if it contains **at least one** missing value.
  - **how = 'all'** : We delete the row (or column) if it contains **only** missing values.
- The **subset** parameter is used to specify the columns/rows on which the search for missing values is carried out.

Example:

```
# We delete all the rows containing at least one missing value
df = df.dropna(axis = 0, how = 'any')

# We delete the empty columns
df = df.dropna(axis = 1, how = 'all')

# We remove the rows with missing values in the 3 columns 'col2', 'col3' and 'col4'
df.dropna(axis = 0, how = 'all', subset = ['col2', 'col3', 'col4'])
```

As with the other methods of replacing values of a `DataFrame` , the `inplace` argument can be used with great care to perform the modification directly without reassignment.

Transaction data for which the transaction amount is not provided is of no interest to us. For this reason:

- **(j)** Delete the `transaction` entries for which the **rate** , **tax** and **total_amt** columns are **all** empty.
- **(k)** Check that the columns of `transactions` **no longer contain missing values.**

In [197]:

```
# Insert your code here

transactions.dropna(axis = 0 , how = 'all', subset = ['rate', 'tax', 'total_amt'], inplace=True)
transactions.isna().sum()
```

Out[197]:

```
cust_id             0
tran_date           0
prod_subcat_code    0
prod_cat_code       0
qty                 0
rate                0
tax                 0
total_amt           0
store_type          0
dtype: int64
```

Hide solution

In [ ]:

```
transactions = transactions.dropna(axis = 0, how = 'all', subset = ['rate', 'tax', 'total_amt'])

transactions.isna().sum(axis = 0)
```

Conclusion and recap

In this chapter we have seen the essential methods of the `pandas` module in order to clean up a dataset and manage missing values (`NaN`).

This step of preparing a dataset is **always** the first step of a data project.

Regarding **data cleaning**, we have learned how to:

- Identify and delete duplicates of a `DataFrame` using the **`duplicated`** and **`drop_duplicates`** methods.
- Modify the elements of a `DataFrame` and their type using the **`replace`** , **`rename`** and **`astype`** methods.
- Apply a function to a `DataFrame` with the **`apply`** method and the **`lambda`** clause.

Regarding the **management of missing values**, we have learned to:

- **Detect** them using the **`isna`** method followed by the **`any`** and **`sum`** methods.
- **Replace** them using the **`fillna`** method and the **statistical methods**.
- **Delete** them using the **`dropna`** method.

In the following notebook, you will see other manipulations of `DataFrames` for a more advanced **exploration** of data.

Validate