DataScientest • com

Pandas for Data Science

Introduction to DataFrames

Introduction

The `pandas` module has been developed to provide `Python` with the tools necessary to manipulate and analyze large volumes of data.

`Pandas` introduces the **`DataFrame`** class, an array-like data structure that offers more advanced data manipulation and exploration than `NumPy` arrays.

The main features of `pandas` are:

- data recovery from files (CSV, Excel tables, etc.)
- handling this data (deletion / addition, modification, statistical visualization, etc.).

This notebook aims at:

- Understanding the format of a `DataFrame`.
- Creating a first `Dataframe`.
- Carrying out a first exploration of a dataset using the `DataFrame` class.

- **(a)** Import the `pandas` module under the name `pd`.

In [55]:

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

Show solution

1. Format of a DataFrame

A `DataFrame` is in the form of a **matrix** whose rows and columns each have an **index**. Typically, columns are indexed by name and rows by unique identifiers.

A `DataFrame` is used to store a **database**. The different **entries** in the database (individuals, animals, objects, etc.) are the different **lines** and their **features** are the different **columns**:

	Name	Gender	Height	Age
0	Robert	M	174	23
1	Mark	M	182	40
2	Aline	F	169	56

- The `DataFrame` above groups together information on **3 individuals**: the `DataFrame` therefore has **3 lines**.
- For each of these individuals, there are **4 variables** (name, gender, height and age) : therefore, the `DataFrame` has **4 columns**.

The column containing **the numbering of the lines** is called the **index** and is not managed in the same way as other columns of the `DataFrame` .

The index can be set by default (will follow the row numbering), defined with one (or several) of the columns of the `DataFrame` or even defined with a list that we specify.

Example: Default indexing (line numbering), you don't have to specify anything :

	Name	Gender	Height	Age
0	Robert	M	174	23
1	Mark	M	182	40
2	Aline	F	169	56

Example: Indexing by the column `'Name'` :

	Gender	Height	Age
Robert	M	174	23
Mark	M	182	40
Aline	F	169	56

Example: Indexing by the list `['person_1', 'person_2', 'person_3']` :

	Name	Gender	Height	Age
person_1	Robert	M	174	23
person_2	Mark	M	182	40
person_3	Aline	F	169	56

We will detail later how to define the index when creating a `DataFrame` .

The `DataFrame` class has several advantages over a `Numpy` array:

- Visually, a `DataFrame` is much more **readable** thanks to more explicit column and row indexing.
- Within the same column the elements are of the same type but from one column to another, the **type of the elements may vary**, which is not the case of `Numpy` arrays which only support data of the same type.
- The `DataFrame` class contains more methods for handling and preprocessing databases, while `NumPy` specializes instead in optimized computation.

2. Creation of a DataFrame: from a NumPy array

It is possible to directly create a `DataFrame` from a `NumPy` array using the `DataFrame()` constructor. The disadvantage of this method is that it is not very practical and the data type is necessarily the same for all the columns.

Let's take a closer look at the header of this constructor.

```
pd.DataFrame(data, index, columns, ...)
```

- The `data` parameter contains the **data** to be formatted (`NumPy` array, list, dictionary or another `DataFrame`).
- The `index` parameter, if specified, must be a **list** containing the **indices of the entries**.
- The `columns` parameter, if specified, must be a **list** containing the **name of the columns**.

For other parameters, you can consult the Python documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html).

Example:

```
# Creation of a NumPy array with 3 rows and 4 columns
array = np.array ([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# Instantiation of a DataFrame
```

In [3]:

```
# Insert your code here

dic = {'name' : ['honey', 'flour', 'wine'],
       'date' : ['08/10/2025','09/25/2024', '10/15/2023'],
       'quantity' : [100, 55, 1800],
       'price' : [2, 3, 10]}

df = pd.DataFrame(dic)

df.head()
```

Out[3]:

	name	date	quantity	price
0	honey	08/10/2025	100	2
1	flour	09/25/2024	55	3
2	wine	10/15/2023	1800	10

Show solution

4. Creation of a DataFrame: from a data file

Most often a `DataFrame` is created directly from a file containing the data of interest. The file's format can be a CSV, Excel, txt, etc.

The most common format is the CSV format, which stands for *Comma-Separated Values* and denotes a spreadsheet-like file whose values are separated by commas.

Here is an example:

```
A, B, C, D,
1, 2, 3, 4,
5, 6, 7, 8,
9, 10, 11, 12
```

In this format:

- **The first line contains the name of the columns**, but sometimes the name of the columns **is not filled in**.
- Each **line** corresponds to an **entry** in the database.
- The values are separated by a **separator character**. In this example, it is `','` but it could be a `';'`.

To import the data into a `DataFrame`, we need to use the `read_csv` function of `pandas` whose header is as follows:

```
pd.read_csv(filepath_or_buffer, sep = ',', header = 0, index_col = 0 ...)
```

The **essential arguments** of the `pd.read_csv` function to know are:

- **`filepath_or_buffer`**: The **path** of the .csv file relative to the execution environment.
  - If the file is in the same folder as the Python environment, just fill in the name of the file.
  - This path must be entered in the form of **character string**.
- **`sep`**: The character used in the .csv file to **to separate** the different columns.
  - This argument must be specified as **character**.
- **`header`**: The **number of the row** that contains the **names** of the columns.
  - If for example the column names are entered in the **first** line of the `.csv` file, then we must specify **`header = 0`**.
  - If the names are **not included**, we will put **`header = None`**.
- **`index_col`**: The **name or number of the column** containing the **indices** of the database.
  - If the database entries are indexed by the first column, you will need to fill in **`index_col = 0`**.
  - Alternatively, if the entries are indexed by a column which bears the name `"Id"`, we can specify **`index_col = "Id"`**.

This function will return an object of type `DataFrame` which contains all the data of the file.

- **(a)** Load the data contained in the file **`transactions.csv`** into a `DataFrame` named **`transactions`**:

  - The file is located in the **same folder** as the environment of this notebook.
  - Columns are separated by **commas**.
  - The names of the columns are in the **first line** of the file.
  - The rows of the database are indexed by the **"transaction_id"** column which is also the **first column**.

In [8]:

```
# Insert your code here

df = pd.read_csv('transactions.csv', sep=',', header=0, index_col=0)
df
```

Out[8]:

	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
...	...	...	...	...	...	...	...	...	...
94340757522	274550	25-01-2011	12.0	5	1	1264.0	132.720	1396.720	e-Shop
89780862956	270022	25-01-2011	4.0	1	1	677.0	71.085	748.085	e-Shop
85115299378	271020	25-01-2011	2.0	6	4	1052.0	441.840	4649.840	MBR
72870271171	270911	25-01-2011	11.0	5	3	1142.0	359.730	3785.730	TeleShop
77960931771	271961	25-01-2011	11.0	5	1	447.0	46.935	493.935	TeleShop

23053 rows × 9 columns

Show solution

We loaded the `transactions.csv` file in the `DataFrame` **`transactions`** which gathers a history of transactions carried out between 2011 and 2014. In the next section, we will study this dataset.

5. First exploration of a dataset using the `DataFrame` class

The rest of this notebook briefly presents the main **methods** of the `DataFrame` class which will allow us to do a quick analysis of our data set, that is:

- Having a brief **overview of the data** (`head` method, `columns` and `shape` attributes).
- **Selecting values** in the `DataFrame` (`loc` and `iloc` methods).
- Carrying out a quick **statistical study** of our data (`describe` and `value _counts` methods)

ℹ As a reminder, to apply a method to an object in Python (such as a DataFrame for example), you must add the method as a suffix of the object. Example: my_object.my_method()

6. Visualization of a `DataFrame`: `head` method, `columns` and `shape` attributes

- It is possible to have a preview of a dataset by displaying **only the first lines** of the `DataFrame`.

For that, we must use the **`head()`** method, specifying as an argument **the number of lines** that we want to display (by default 5).

It is also possible to preview the **last lines** using the **`tail()`** method which is applied in the same way:

```
# Display of the first 10 lines of my_dataframe
my_dataframe.head(10)
```

In [53]:

```
# Insert your code here
transactions.head(20)
```

Out[53]:

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
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
29258453508	270384	20-02-2014	5.0	3	5	1497.0	785.925	8270.925	e-Shop
25455265351	267750	20-02-2014	12.0	6	3	1360.0	428.400	4508.400	e-Shop
1571002198	275023	20-02-2014	6.0	5	4	587.0	246.540	2594.540	e-Shop
43134751727	268487	20-02-2014	3.0	2	-1	-611.0	64.155	-675.155	e-Shop
36554696014	269345	20-02-2014	3.0	5	3	1253.0	394.695	4153.695	e-Shop
56814940239	268799	20-02-2014	7.0	5	5	368.0	193.200	2033.200	e-Shop
54295803788	270787	20-02-2014	12.0	5	5	584.0	306.600	3226.600	e-Shop
25963520987	274829	20-02-2014	4.0	4	3	502.0	158.130	1664.130	Flagship store
17183929085	266863	20-02-2014	1.0	2	1	1359.0	142.695	1501.695	TeleShop
44783317894	269452	20-02-2014	3.0	1	3	825.0	259.875	2734.875	TeleShop

Show solution

- **(b)** Display the last **10 lines** of the `transactions DataFrame`.

In [52]:

```
# Insert your code here
transactions.tail(10)
```

Out[52]:

transaction_id	cust_id	tran_date	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt	Store_type
49882891062	271982	25-01-2011	10.0	5	4	1330.0	558.600	5878.600	e-Shop
14787475597	273982	25-01-2011	4.0	3	5	969.0	508.725	5353.725	e-Shop
14787475597	273982	25-01-2011	4.0	3	5	969.0	508.725	5353.725	e-Shop
40893803228	272049	25-01-2011	11.0	6	3	1077.0	339.255	3570.255	e-Shop
30856003613	266866	25-01-2011	4.0	2	2	444.0	93.240	981.240	TeleShop
94340757522	274550	25-01-2011	12.0	5	1	1264.0	132.720	1396.720	e-Shop
89780862956	270022	25-01-2011	4.0	1	1	677.0	71.085	748.085	e-Shop
85115299378	271020	25-01-2011	2.0	6	4	1052.0	441.840	4649.840	MBR
72870271171	270911	25-01-2011	11.0	5	3	1142.0	359.730	3785.730	TeleShop
77960931771	271961	25-01-2011	11.0	5	1	447.0	46.935	493.935	TeleShop

Show solution

It is possible to retrieve the **name of the columns** of a `DataFrame` thanks to its **`columns`** attribute.

```
# Creation of a DataFrame df from a dictionary
dictionary = {'A': [1, 5, 9],
              'B': [2, 6, 10],
              'C': [3, 7, 11],
              'D': [4, 8, 12]}

df = pd.DataFrame (data = dictionary, index = ['i_1', 'i_2', 'i_3'])
```

These instructions produce the same `DataFrame` as before:

In [15]:
```
# Insert your code here
print(transactions.shape)

transactions.columns[4]
```

```
(23053, 9)
```

Out[15]: 'Qty'

Show solution

7. Selecting columns from a `DataFrame`

Extracting columns from a `DataFrame` is almost identical to extracting data from a dictionary.

To extract a **column** from a `DataFrame` , all we have to do is enter **between brackets** the **name** of the column to extract. To extract **several** columns, we must enter between brackets **the list of the names** of the columns to extract:

```
# Display of the 'cust_id' column
print(transactions['cust_id'])

# Extraction of 'cust_id' and 'Qty' columns from transactions
cust_id_qty = transactions[["cust_id", "Qty"]]
```

`cust_id_qty` is a new **`DataFrame`** containing only the `'cust_id'` and `'Qty'` columns.

The display of the first 3 lines of **`cust_id_qty`** yields:

	cust_id	Qty
transactions_id		
80712190438	270351	-5
29258453508	270384	-5
51750724947	273420	-2

When we prepare a dataset for later use, it is better to **separate** the **categorical** variables from the **quantitative** variables:

- A *categorical* variable is a variable that takes only a finite **number** of *modalities.*

  The categorical variables of the `DataFrame` `transactions` are: `['cust_id', 'tran_date', 'prod_subcat_code', 'prod_cat_code', 'Store_type']` .

- A *quantitative* variable is a variable that measures a quantity that can take an **infinite** number of values.

  The quantitative variables of `transactions` are: `['Qty', 'Rate', 'Tax', 'total_amt']` .

This distinction is made because some basic operations like calculating an average only make sense for quantitative variables.

- **(a)** In a `DataFrame` named **`cat_vars`** , store the **categorical** variables of `transactions` .
- **(b)** In a `DataFrame` named **`num_vars`** , store the **quantitative** variables of `transactions` .
- **(c)** Display the first 5 lines of each `DataFrame` .

In [23]:
```
### Insert your code here

cat_vars = transactions[['cust_id', 'tran_date', 'prod_subcat_code', 'prod_cat_code', 'Store_type']]
num_vars = transactions[['Qty', 'Rate', 'Tax', 'total_amt']]

print(cat_vars.head())
print(30*'-')
print(num_vars.head())
```

```
                cust_id   tran_date  prod_subcat_code  prod_cat_code  \
transaction_id
80712190438      270351  28-02-2014               1.0              1
29258453508      270384  27-02-2014               5.0              3
51750724947      273420  24-02-2014               6.0              5
93274880719      271509  24-02-2014              11.0              6
51750724947      273420  23-02-2014               6.0              5

               Store_type
transaction_id
80712190438        e-Shop
29258453508        e-Shop
51750724947      TeleShop
93274880719        e-Shop
51750724947      TeleShop
------------------------------
                Qty    Rate      Tax  total_amt
transaction_id
80712190438      -5  -772.0  405.300  -4265.300
29258453508      -5 -1497.0  785.925  -8270.925
51750724947      -2  -791.0  166.110  -1748.110
93274880719      -3 -1363.0  429.345  -4518.345
51750724947      -2  -791.0  166.110  -1748.110
```

Show solution

8. Selecting rows of a `DataFrame` : `loc` and `iloc` methods

To extract one or more **rows** from a `DataFrame` , we use the **`loc`** method. `loc` is a very special type of method because the arguments are filled in **between square brackets** and not between parentheses. Using this method is very similar to indexing lists.

In order to retrieve the line of index `i` of a `DataFrame` , all we have to do is enter `i` as an argument of the `loc` method:

```
# We retrieve the line of index 80712190438 of the num_vars DataFrame
print(num_vars.loc[80712190438])
```

```
>>                 Rate    Tax  total_amt
>> transaction_id
>> 80712190438   -772.0  405.3    -4265.3
>> 80712190438    772.0  405.3     4265.3
```

In order to retrieve **several rows**, we can either:

- Enter a **list of indices**.
- Enter a **slice** by specifying the start and end indices of the slice. To use **slicing** with `loc` , the indices must be **unique**, which is not the case for `transactions` .

```
# We retrieve the rows at indices 80712190438, 29258453508 and 51750724947 from the transactions DataFrame
transactions.loc[[80712190438, 29258453508, 51750724947]]
```

`loc` can also take a column or **list of columns** as an argument in order to refine the data extraction:

```
# We extract the columns 'Tax' and 'total_amt' from the rows at index 80712190438 and 29258453508
transactions.loc[[80712190438, 29258453508], ['Tax', 'total_amt']]
```

This instruction produces the following `DataFrame` :

transaction_id	Tax	total_amt
80712190438	405.300	-4265.300
80712190438	405.300	4265.300
29258453508	785.925	-8270.925
29258453508	785.925	8270.925

The **`iloc`** method is used to index a `DataFrame` **exactly like a numpy array**, that is to say by only filling in the **numeric** indices of the rows and columns. This allows the use of slicing without constraint:

```
# Extraction of the first 4 rows and the first 3 columns of transactions
transactions.iloc[0:4, 0:3]
```

This instruction produces the following `DataFrame` :

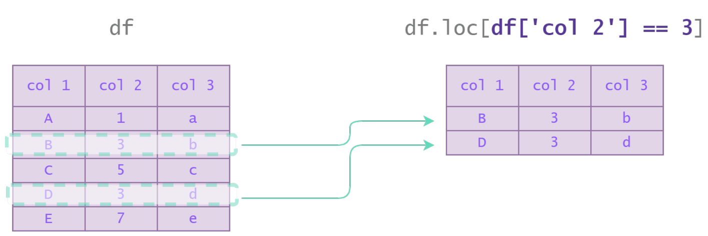
transaction_id	cust_id	tran_date	prod_subcat_code
80712190438	270351	28-02-2014	1.0
29258453508	270384	27-02-2014	5.0
51750724947	273420	24-02-2014	6.0
93274880719	271509	24-02-2014	11.0

If the row indexing is the one by default (row numbering), the `loc` and `iloc` methods are **equivalent**.

9. Conditional indexing of a `DataFrame`

As with Numpy arrays, we can use **conditional indexing** to extract rows from a `Dataframe` that meet a given condition.

In the following illustration, we select the rows of the `DataFrame` `df` **for which the column `col 2` is equal to 3.**



There are two syntaxes for conditionally indexing a `DataFrame` :

```
# We select the rows of the DataFrame df for which the column 'col 2' is equal to 3.
df[df['col 2'] == 3]

df.loc[df['col 2'] == 3]
```

If we want to **assign a new value** to these entries, we must absolutely use the **`loc`** method.

Indeed, indexing with the syntax `df[df['col 2'] == 3]` only returns a **copy** of these entries and does not provide access the memory location where the data is located.

The manager of the transactions listed in the **`transactions`** `DataFrame` wishes to have access to the **identifiers** of customers who have made an **online** purchase (i.e. in a `"e-Shop"`) as well as **the date of the corresponding transaction**.

We have the following information about the columns of `transactions` :

Column name	Description
`'cust_id'`	The **identifier** of the customer

Column name	Description
'Store_type'	The **type of store** where the transaction took place
'tran_date'	The **date** of the transaction

- **(a)** In a `DataFrame` named **`transactions_eshop`**, store the transactions that took place in an `"e-Shop"` type store.
- **(b)** In another `DataFrame` named **`transactions_id_date`**, store the **customer identifiers** and the **transaction date** of the `transactions_eshop` `DataFrame`.

In [26]:

```
# Insert your code here

transactions_eshop = transactions.loc[transactions['Store_type'] == 'e-Shop']

transactions_id_date = transactions_eshop[['cust_id', 'tran_date']]

transactions_id_date.head()
```

Out[26]:

	cust_id	tran_date
transaction_id		
80712190438	270351	28-02-2014
29258453508	270384	27-02-2014
93274880719	271509	24-02-2014
45649838090	273667	22-02-2014
50076728598	269014	21-02-2014

Show solution

Now, the manager would like to have access to the transactions carried out by the client whose identifier is `268819`.

- **(d)** In a `DataFrame` named **`transactions_client_268819`**, store all transactions with client identifier `268819`.
- **(e)** A column in a `DataFrame` can be iterated over with a loop exactly like a list (`for value in df['column']:`). Using a `for` loop on the `'total_amt'` column, compute and display the total transaction amount for the client with identifier `268819`.

In [30]:

```
# Insert your code here

transactions_client_268819 = transactions.loc[transactions.cust_id == 268819]

# First Way
total = 0
for i in transactions_client_268819['total_amt']:
    total += i
print(total)

# Second Way
transactions.loc[transactions.cust_id == 268819]['total_amt'].sum()
```

```
14911.974999999999
```

Out[30]: 14911.974999999999

Show solution

10. Quick statistical study of the data in a `DataFrame`.

> The **`describe`** method of a `DataFrame` returns a summary of the **descriptive statistics** (min, max, mean, quantiles,...) of its **quantitative** variables. It is therefore a very useful tool for a first visualisation of the type and distribution of these variables.
>
> To analyse the **categorical** variables, it is recommended to start by using the **`value_counts`** method which returns the **number of occurrences** for each modality of these variables. The `value_counts` method cannot be used directly on a `DataFrame` but only on the columns of the `DataFrame` which are objects of the **`pd.Series`** class.

- **(a)** Use the `describe` method of the `DataFrame` `transactions`.
- **(b)** The quantitative variables of `transactions` are `'Qty'`, `'Rate'`, `'Tax'` and `total_amt'`. By default, are the statistics produced by the `describe` method **only** computed on the quantitative variables?
- **(c)** Display the number of occurrences of each modality of the `Store_type` column using the `value_counts` method.

In [32]:

```
# Insert your code here

# a
transactions.describe()

# b
# No some catogerical variable data type are numerical. Because of that describe compute also this
# categorical variables
```

Out[32]:

	cust_id	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt
count	23053.000000	23021.000000	23053.000000	23053.000000	23031.000000	23031.000000	23031.000000
mean	271022.241661	6.150298	3.762721	2.434173	636.405019	248.665526	2108.007267
std	2430.830508	3.726557	1.677314	2.265703	622.053592	187.087709	2506.210476
min	266783.000000	1.000000	1.000000	-5.000000	-1499.000000	7.350000	-8270.925000
25%	268936.000000	3.000000	2.000000	1.000000	312.000000	98.175000	762.450000
50%	270981.000000	5.000000	4.000000	3.000000	710.000000	199.080000	1756.950000
75%	273114.000000	10.000000	5.000000	4.000000	1109.000000	365.820000	3569.702500
max	275265.000000	12.000000	6.000000	5.000000	1500.000000	787.500000	8287.500000

In [33]:
```
# c)
transactions.Store_type.value_counts()
```

Out[33]:
```
e-Shop            9284
MBR               4657
Flagship store    4565
TeleShop          4493
Name: Store_type, dtype: int64
```

Show solution

> The `describe` method computed statistics on the variables `cust_id`, `prod_subcat_code` and `prod_cat_code` while these are **categorical** variables.
>
> Of course, these statistics make **no sense**. The `describe` method has treated these variables as quantitative because the modalities they take are of numerical type.
>
> This is why it is necessary to pay attention to the results returned by the `describe` method and always take a step back to remember what the variables are reflecting.
>
> The manager wishes to make a quick report on the characteristics of the transactions `DataFrame`: in particular, he wants to know the **average amount spent** as well as the **maximum quantity** purchased.

- **(d)** What is the **average** total amount spent? We are interested in the `'total_amt'` column of `transactions`.
- **(e)** What is the **maximum** quantity purchased? We will look at the `'Qty'` column of `transactions`.

In [37]:
```
# Insert your code here

print('average total amount spent :', transactions['total_amt'].mean())
print('maximum quantity purchased :', transactions['Qty'].max())

transactions.describe()
```

```
average total amount spent : 2108.007266944553
maximum quantity purchased : 5
```

Out[37]:

	cust_id	prod_subcat_code	prod_cat_code	Qty	Rate	Tax	total_amt
count	23053.000000	23021.000000	23053.000000	23053.000000	23031.000000	23031.000000	23031.000000
mean	271022.241661	6.150298	3.762721	2.434173	636.405019	248.665526	2108.007267
std	2430.830508	3.726557	1.677314	2.265703	622.053592	187.087709	2506.210476
min	266783.000000	1.000000	1.000000	-5.000000	-1499.000000	7.350000	-8270.925000
25%	268936.000000	3.000000	2.000000	1.000000	312.000000	98.175000	762.450000
50%	270981.000000	5.000000	4.000000	3.000000	710.000000	199.080000	1756.950000
75%	273114.000000	10.000000	5.000000	4.000000	1109.000000	365.820000	3569.702500
max	275265.000000	12.000000	6.000000	5.000000	1500.000000	787.500000	8287.500000

Show solution

Some transactions have **negative** amounts.

These are transactions that have been cancelled and refunded to the client. These amounts will disrupt the distribution of the amounts which gives us **bad estimates** of the mean and quantiles of the variable `total_amt`.

- **(f)** What is the average amount of transactions with **positive** amounts?

In [54]:
```
transactions.to_csv('transactions.csv')
```

In [60]:
```
# Insert your code here

print('Average amount of transactions with positive amounts : ',
      transactions.loc[transactions['total_amt'] > 0]['total_amt'].mean())

transactions.loc[transactions['total_amt'] > 0]['total_amt'].describe()
```

```
Average amount of transactions with positive amounts :  2608.2052178706676
```

Out[60]:
```
count    20861.000000
mean      2608.205218
std       1963.405006
min         77.350000
25%       1029.860000
50%       2090.660000
75%       3825.510000
max       8287.500000
Name: total_amt, dtype: float64
```

Show solution

Conclusion and recap

The `DataFrame` class of the `pandas` module will be your favorite data structure when exploring, analysing and processing datasets and databases.

In this brief introduction, you have learned to:

Unvalidate