train.datascientest.com

pandas_03_processing_english - Jupyter Notebook

28–36 dakika

¶

Pandas for Data Science¶

Data processing ¶

Data processing can be reduced to the use of 4 essential operations: **filtering**, **merging**, **ordering** and **grouping**.

If the `DataFrame` class has prevailed in the domain of data manipulation, it is because it is often sufficient to **repeat or combine** these four operations.

In this exercise, you will learn how to use these 4 operations of data processing.

- Before starting this notebook, **run the following cell** in order to retrieve the work done in the previous exercises.

1. Filtering a `DataFrame` with binary operators.¶

Filtering consists in **selecting** a subset of **rows** of a DataFrame which meet a **condition**. Filtering corresponds to what was called conditional indexing until now, but the term "filtering" is the one that is used most in database management.

We cannot use the logical operators `and` and `or` to filter on multiple conditions. Indeed, these operators create **ambiguities** that pandas is unable to handle for filtering.

The operators suitable for filtering on several conditions are the binary **operators**:

- The 'and' operator: &.
- The 'or' operator: `|`.
- The 'not' operator: –.

These operators are similar to logical operators but their evaluation methods are not the same.

The 'and' operator: &.¶

The & operator is used to filter a `DataFrame` on several conditions which must be verified **simultaneously**.

Example:

Consider the following `DataFrame`` df` that contains information on apartments in Paris:

	district	year	surface
0	'Champs-Elysées'	1979	70
1	'Europe'	1850	110
2	'Père-Lachaise'	1935	55

	district	year	surface
3	'Bercy'	1991	30

If we want to find an apartment dating from 1979 **and** with a surface area greater than 60 squared meters, we can filter the lines of `df` with the following code:

```
# Filtering of the DataFrame on the 2 previous conditions
print(df[(df['year'] == 1979) & (df['surface']> 60)])
```

```
>>>         district  year  surface
>>> 0  Champs-Elysées  1979       70
```

The conditions must be written **between parentheses** to eliminate any ambiguity on the **order of evaluation** of the conditions. Indeed, if the conditions are not properly separated, we will get the following error:

```
print(df[df['year'] == 1979 & df['surface']> 60])
```

```
>>> ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

The 'or' operator: |.¶

The operator `|` is used to filter a `DataFrame` on several conditions of which **one at least** must be verified.

Example:

Consider the same `DataFrame` `df`:

	district	year	surface
0	'Champs-Elysées'	1979	70
1	'Europe'	1850	110
2	'Père-Lachaise'	1935	55
3	'Bercy'	1991	30

If we want to find an apartment that dates after 1900 **or** is located in the Père-Lachaise district, we can filter the lines of `df` with the following code:

```
# Filtering of the DataFrame on the 2 previous conditions
print(df[(df['year']> 1900) | (df['district'] == 'Père-Lachaise')])
```

```
>>>          district  year  surface
>>> 0  Champs-Elysées  1979       70
>>> 2  Père-Lachaise   1935       55
>>> 3  Bercy           1991       30
```

The 'not' operator: –.¶

The operator – is used to filter a`DataFrame` on a condition which must **not** be true, i.e. whose **negation** must be verified.

Example:

Consider the same `DataFrame`` df`:

	district	year	surface
0	'Champs-Elysées'	1979	70
1	'Europe'	1850	110
2	'Père-Lachaise'	1935	55

	district	year	surface
3	'Bercy'	1991	30

If we want to retrieve apartments **not** located in the Bercy district then we filter `df` as follows:

```
# Filtering of the DataFrame on the the negation of a condition
print(df[~(df['district'] == 'Bercy')])
```

```
>>> district year surface
>>> 0 Champs-Elysées 1979 70
>>> 1 Europe 1850 110
>>> 2 Père-Lachaise 1935 55
```

- **(a)** Display the first 5 lines of the `transactions` `DataFrame`.
- **(b)** From **`transactions`**, create a `DataFrame` named **`e_shop`** containing only the transactions carried out in stores of type **`'e-Shop'`** with a total amount **greater than** than 5000 (`'store_type'` and `'total_ amt'` columns).
- **(c)** Similarly, create a `DataFrame` named **`teleshop`** which contains the transactions made in stores of type **`'TeleShop'`** with a total amount of more than 5000.
- **(d)** Which of the two types of store has the most transactions over € 5,000?

  solution a ----------
  solution b ----------
  solution c ----------
  solution d ----------
  e_shop shape : 1185 tele_shop shape : 532
- **(e)** Import into two `DataFrames` named respectively **`customer`** and **`prod_cat_info`** the data contained in the files **`'customer.csv'`** and **`'prod_cat_info.csv'`**.
- **(f)** The `Gender` and `city_code` columns of **`customer`** contain two missing values each. Replace them with their mode using the `fillna` and `mode` methods.

  Out[21]:

	prod_cat_code	prod_cat	prod_sub_cat_code	prod_subc
0	1	Clothing	4	Mens
1	1	Clothing	1	Women
2	1	Clothing	3	Kids
3	2	Footwear	1	Mens
4	2	Footwear	3	Women

solution f ----------

Out[26]:

customer_Id 0
DOB 0
Gender 0
city_code 0
dtype: int64

2. Joining `Dataframes`: `concat` function and `merge` method.¶

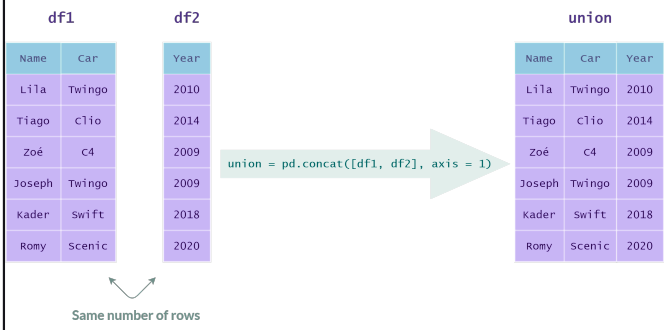
Concatenation of `DataFrames` with `concat`¶

The `concat` function of the`pandas` module allows you to concatenate several `DataFrames`, i.e. juxtapose them horizontally or vertically.

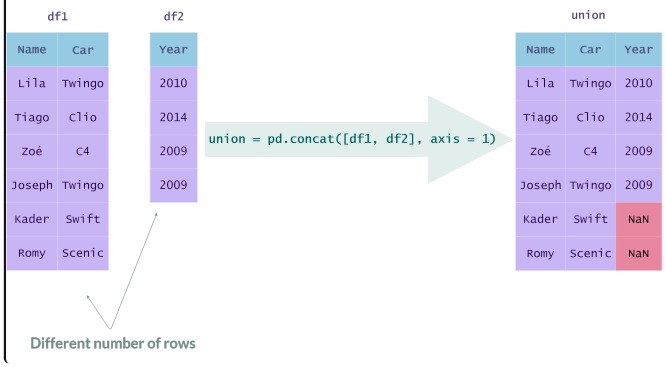
The header of this function is as follows:

`pandas.concat (objs, axis ..)`

- The `objs` parameter contains the list of `DataFrames` to concatenate.
- The `axis` parameter specifies whether to concatenate vertically (`axis = 0`) or horizontally (`axis = 1`).



When the number of rows or columns of the `DataFrames` does not match, the`concat` function fills the empty cells with NaN, as shown in the illustration below.



- **(a)** Split the **columns** of the **`transactions`** `DataFrame` in half with half of the columns in a `DataFrame` named **`part_1`** and the second half in a `DataFrame` named **`part_2`**.
- **(b)** Reconstitute **`transactions`** in a `DataFrame` named **`union`** by concatenating **`part_1`** and **`part_2`** horizontally.
- **(c)** What happens if we concatenate `part_1` and `part_2` by filling in the argument **`axis = 0`**?

solution a ----------
solution b ----------
solution c ----------

Out[46]:

	cust_id	tran_date	prod_subcat_code	pro
transaction_id				
80712190438	270351.0	28-02-2014	1.0	1.0
29258453508	270384.0	27-02-2014	5.0	3.0
51750724947	273420.0	24-02-2014	6.0	5.0
93274880719	271509.0	24-02-2014	11.0	6.0
51750724947	273420.0	23-02-2014	6.0	5.0
...	...	...	...	...
94340757522	NaN	NaN	NaN	Na
89780862956	NaN	NaN	NaN	Na
85115299378	NaN	NaN	NaN	Na
72870271171	NaN	NaN	NaN	Na
77960931771	NaN	NaN	NaN	Na

45838 rows × 9 columns

Merging `DataFrames` with the `merge` method¶

Two `DataFrames` can be merged if they have a column in common. This is done thanks to the `merge` method of the `DataFrame` class whose header is as follows:

```
merge(right, on, how, ...)
```

- The **`right`** parameter is the `DataFrame` to merge with the one calling the method.
- The **`on`** parameter is the name of the columns of the `DataFrame` which will be used as reference for the merge. They must be **common** to both `DataFrames`
- The **`how`** parameter allows you to choose the **type of join** to perform for merging the `DataFrames`. The values for this parameter are based on **SQL** syntax joins.

The how parameter can take 4 values (`'inner'`, `'outer'`, `'left'`, `'right'`) that we will illustrate on the two `DataFrames` named `Persons` and `Vehicles` below:

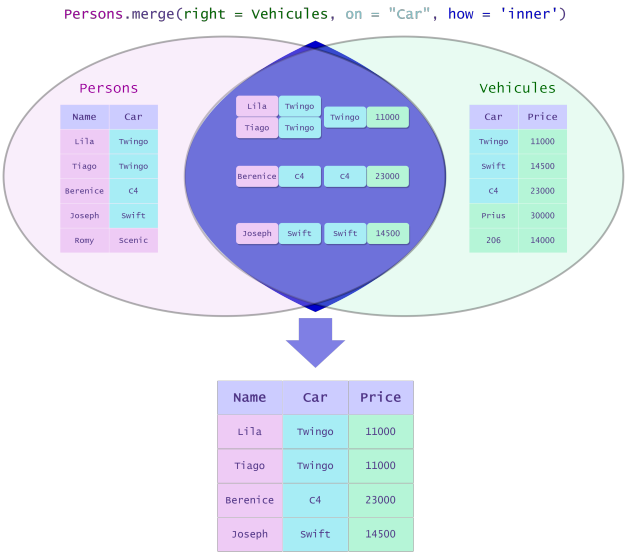
Name	Car
Lila	Twingo
Tiago	Clio
Berenice	C4 Cactus
Joseph	Twingo
Kader	Swift
Romy	Scenic

Car	Price
Twingo	11000
Swift	14500
C4 Cactus	23000
Clio	16000
Prius	30000

- **`'inner'`**: The inner join returns the rows whose values in the common columns are **present in the two `DataFrames`**. This type of join is often **not recommended** because it can lead to the loss of many entries. However, the inner join does not produces **NAs**.

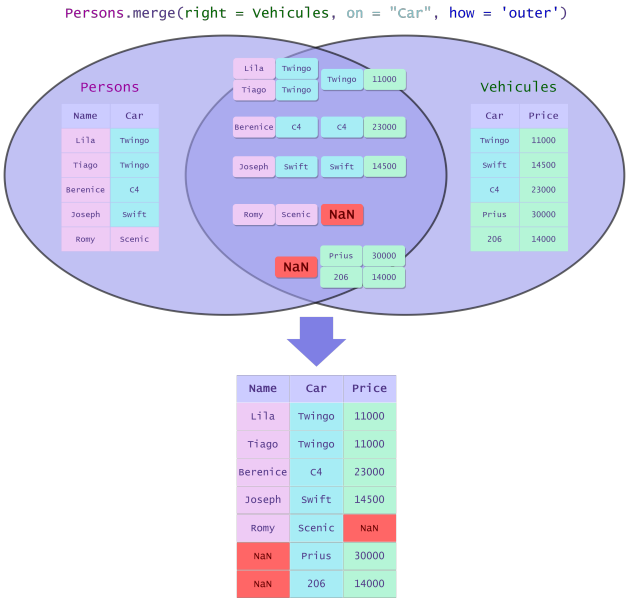
  The result of the inner join `Persons.merge(right =`

`Vehicles, on = 'Car', how = 'inner')` is shown below:



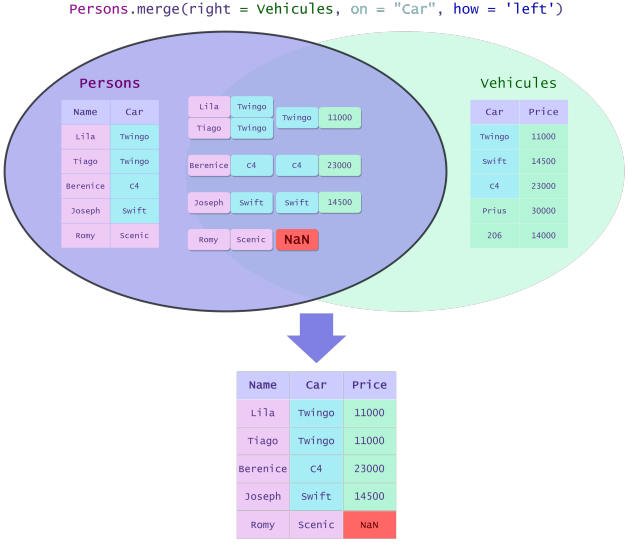
- **`'outer'`**: The outer join Persons the two `DataFrames` **in their entirety**. No row will be deleted. This method can generate **a lot of NAs**.

  The result of the outer join `Persons.merge(right = Vehicles, on = 'Car', how = 'outer')` is shown below:



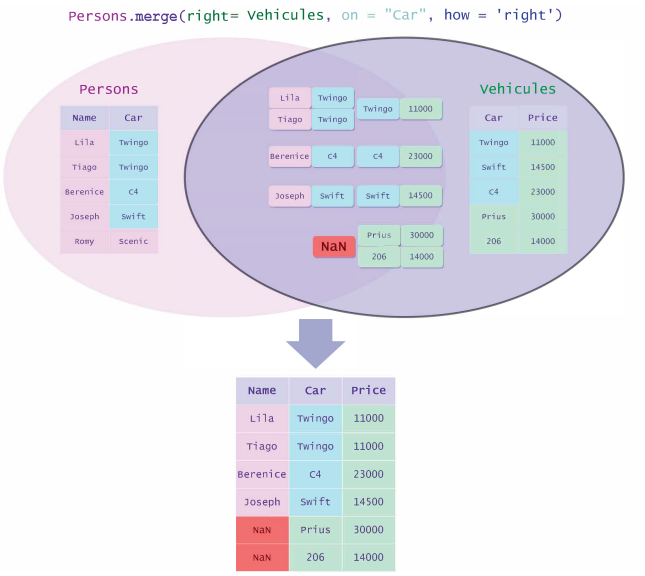
- **`'left'`**: The left join returns **all the rows** of the `DataFrame` on the **left** (i.e. the one calling the method), and completes them with the rows of the second `DataFrame` which coincide according to the values of the common column. This is the **default value for the how** parameter.

  The result of the left join `Persons.merge(right = Vehicles, on = 'Car', how = 'left')` is shown below:



- `'right'`: The right join returns **all the rows** of the `DataFrame` on the **right**, and complete them with the rows of the left `DataFrame` which coincide according to the values of the common column.

  The result of the right join `Persons.merge(right = Vehicules, on = 'Car', how = 'right')` is shown below:



  Doing a left join, right join, or outer join followed by a `dropna(how = 'any')` is equivalent to an inner join.

  The **`customer`** `DataFrame` contains information about customers in the `'cust_id'` column of `transactions`.

  The **`'customer_Id'`** column of the `customer` `DataFrame` will be used to make the join between `transactions` and `customer`. This will enrich the `transactions` `DataFrame` with additional information.

- **(d)** Using the `rename` method and a dictionary, rename the **`'customer_Id'`** column of the **`customer`** `DataFrame` to **`'cust_id'`**.
- **(e)** Using the `merge` method, perform the **left join** between the `DataFrames` **`transactions`** and **`customer`** on the `'cust_id'` column. Name the created `DataFrame`**`fusion`**.
- **(f)** Did the merging produce NAs?
- **(g)** Display the first lines of **`fusion`**. What are the new columns?

  Out[43]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	q
0	270351	28-02-2014	1	1	-5
1	270384	27-02-2014	5	3	-5

	cust_id	tran_date	prod_subcat_code	prod_cat_code	q
2	273420	24-02-2014	6	5	-1
3	271509	24-02-2014	11	6	-3
4	273420	23-02-2014	6	5	-1

The merging went well and produced no NaNs. However, the index of the `DataFrame` is no longer the column **`transaction_id'`** and has been reset with the default index (`0`,1, 2 , ...).

It is possible to re-define the index of a `DataFrame` using the **`set_index`** method.

This method can take as argument:

- The **name** of a column to use as indexing.
- A Numpy `array` or pandas `Series` with the same number of rows as the `DataFrame` calling the method.

Example:

Let `df` be the following `DataFrame`:

	Name	Car
0	Lila	Twingo
1	Tiago	Clio
2	Berenice	C4 Cactus
3	Joseph	Twingo
4	Kader	Swift
5	Romy	Scenic

We can set the column `'Name'` as being the new index:

```
df = df.set_index('Name')
```

This will produce the following `DataFrame`:

Name	Car
Lila	Twingo
Tiago	Clio
Berenice	C4 Cactus
Joseph	Twingo
Kader	Swift
Romy	Scenic

We can also define the index from a Numpy array, from a `Series`, etc:

```
# New index to use
new_index = ['10000' + str(i) for i in range(6)]
print(new_index)
>>> ['100000', '100001', '100002', '100003',
'100004', '100005']

# Using an array or a Series is equivalent
index_array = np.array(new_index)
index_series = pd.Series(new_index)


df = df.set_index(index_array)
df = df.set_index(index_series)
```

This will produce the following `DataFrame`:

	Name	Car
100000	Lila	Twingo

	Name	Car
100001	Tiago	Clio
100002	Berenice	C4 Cactus
100003	Joseph	Twingo
100004	Kader	Swift
100005	Romy	Scenic

To return to the default numeric indexing, we use the **`reset_index`** method of the `DataFrame`:

```
df = df.reset_index()
```

The indexing column that was used **is not deleted**. A new column will be created containing the old index:

	index	Name	Car
0	100000	Lila	Twingo
1	100001	Tiago	Clio
2	100002	Berenice	C4 Cactus
3	100003	Joseph	Twingo
4	100004	Kader	Swift
5	100005	Romy	Scenic

Merging `transactions` and`customer` removed the index of `transactions`.

The index of a `DataFrame` can be retrieved using its `.index` attribute.

- **(h)** Take the index from `transactions` and use it to index `fusion`.

  Out[85]:

	cust_id	tran_date	prod_subcat_code	proc
transaction_id				
80712190438	270351	28-02-2014	1	1
29258453508	270384	27-02-2014	5	3
51750724947	273420	24-02-2014	6	5
93274880719	271509	24-02-2014	11	6
51750724947	273420	23-02-2014	6	5
...	...	...	...	...
94340757522	274550	25-01-2011	12	5
89780862956	270022	25-01-2011	4	1
85115299378	271020	25-01-2011	2	6
72870271171	270911	25-01-2011	11	5
77960931771	271961	25-01-2011	11	5

22919 rows × 12 columns

3. Sort and order the values of a `DataFrame`: `sort_values` and `sort_index` methods.¶

The `sort_values` method allows you to sort the rows of a `DataFrame` according to the values of one or more columns.

The header of this method is as follows:

```
sort_values(by, ascending, ...)
```

- The by parameter allows you to specify on which column(s) the sort is performed.

- The `ascending` parameter is a boolean value (`True` or `False`) determining whether the sorting order is ascending or descending. By default this parameter is set to `True`.

Example:

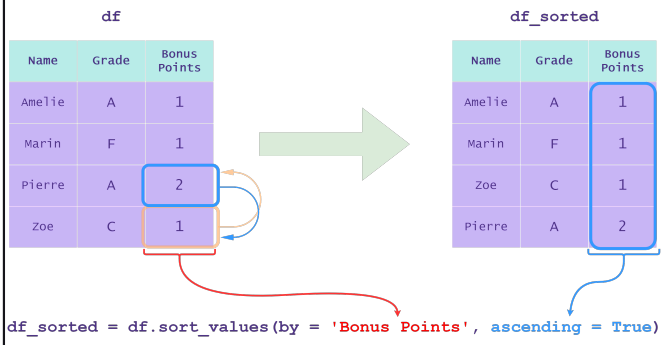
Consider the `DataFrame` `df` describing students:

Name	Grade	Bonus points
'Amelie'	A	1
'Marin'	F	1
'Pierre'	A	2
'Zoe'	C	1

First of all, we will sort the rows on a single column, for example the column `'Bonus Points'`:

```
# We sort the DataFrame df on the column 'Bonus Points'
df_sorted = df.sort_values(by ='Bonus Points', ascending = True)
```

We obtain the following result:

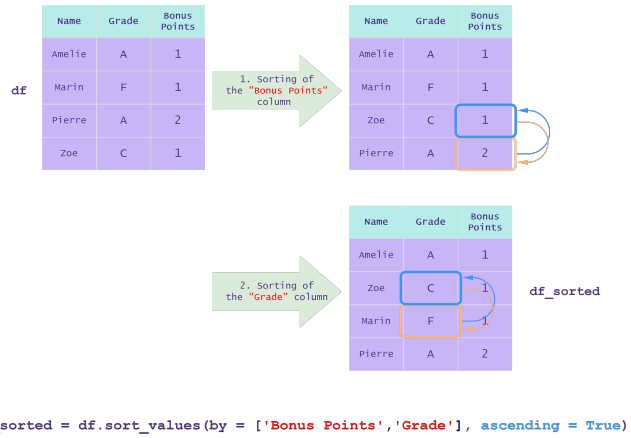


The rows of the `DataFrame` `df_sorted` are therefore sorted in **ascending order** of the **`'Bonus points'`** column. However, if we look at the column `'Grade'`, we see that it is not sorted alphabetically for the common values of `'Bonus Points'`.

This can be remedied by also sorting by the `'Grade'` column:

```
# We first sort the DataFrame df by the column 'Bonus Points' then in case of equality, by the column 'Grade'.
df_sorted = df.sort_values(by = ['Bonus Points', 'Grade'], ascending = True)
```

We obtain the following result:



The **`sort_index`** method allows you to sort a `DataFrame` according to its index.

If the index is the default one (numerical), this method is not particularly interesting.

However, it can often be combined with the `set_index` method of

a DataFrame that we have just seen.

Example:

```
# We define the column 'Grade' as the index of df
df = df.set_index('Grade')

# We sort the DataFrame df according to its index
df = df.sort_index()
```

This produces the following DataFrame:

Grade	Name	Bonus points
A	'Amelie'	1
A	'Peter'	2
C	'Zoe'	1
F	'Sailor'	1

Consider the two following DataFrames containing boat rental data.

Below are the boats DataFrame:

	boat_name	color	reservation_number	n_reservations
0	Julia	blue	2	34
1	Siren	green	3	10
2	Sea Sons	red	6	20
3	Hercules	blue	1	41
4	Cesar	yellow	4	12
5	Minerva	green	5	16

And the clients DataFrame:

	client_id	client_name	reservation_id
0	91	Marie	1
1	154	Anna	2
2	124	Yann	3
3	320	Lea	7
4	87	Marc	9
5	22	Yassine	10

- **(a)** Run the following cell to instantiate these DataFrames.

  We want to easily determine which customer has reserved the boats of the boats DataFrame. To do this, we can simply merge the DataFrames.

- **(b)** Rename the `'reservation_number'` column from boats to `'reservation_id'` using the rename method.
- **(c)** In a DataFrame named **boats_clients**, perform the **left join** between boats (left) and clients (right).
- **(d)** Set the column `'boat_name'` as index of the boats_clients DataFrame.
- **(e)** Using the loc method, find who reserved the boats `'Julia'` and `'Siren'`.
- **(f)** Using the isna method applied to the client_name column, determine the boats that have not been reserved.
- **(g)** The number of times a boat has been reserved so far is indicated by the column `'n_reservations'`. Using the **sort_values** method, determine the name of the customer who reserved the **blue** boat with the most reservations to date.

Out[120]:

	color	reservation_id	n_reservations	client_id
boat_name				
Hercules	blue	1	41	91.0
Julia	blue	2	34	154.0
Sea Sons	red	6	20	NaN
Minerva	green	5	16	22.0
Cesar	yellow	4	12	NaN
Siren	green	3	10	124.0

4. Grouping the elements of a `DataFrame`: `groupby`, `agg` and `crosstab` methods.¶

The **`groupby`** method allows you to **group the rows** of a `DataFrame` which share a **common** value on a given column.

This method does not return a `DataFrame`. The object returned by the `groupby` method is an object of the **`DataFrameGroupBy`** class.

This class is used to perform operations such as calculating statistics (sum, average, maximum, etc.) for each modality of the column on which the rows are grouped.

The general structure of a **`groupby` operation** is as follows:

- **Split** the data.
- **Apply** a function.
- **Combine** the results.

Example:

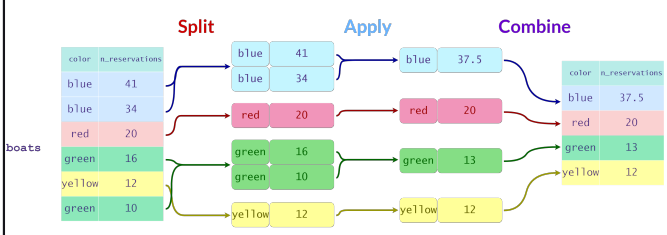
It is assumed that the boats in the `boats` `DataFrame` are all identical and have the same age. We want to determine if the color of a boat has an influence on its number of reservations. For this, we will calculate for each color the average number of reservations per boat.

It is therefore necessary to:

- **Split** the boats by color.
- **Apply** the **`mean`** function to compute the average number of reservations.
- **Combine** the results in a `DataFrame` to easily compare them.

Therefore, we can use the **`groupby`** method followed by the **`mean`** method to get the result:

```
boats.groupby("color").mean()
```



All the usual statistical methods (`count`,`mean`, `max`, etc.) can be used as a suffix of the `groupby` method. These will only be applied on columns of compatible type.

It is possible to specify for each column which function must be used in the **Apply** step of a `groupby` operation. For that, we use the **agg** method of the `DataFrameGroupBy` class by giving it a **dictionary** where each **key** is the **name** of a column and the **value** is the **function** to apply.

Example:

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

transaction_id	cust_id	tran_date	prod_subcat_code	proc
80712190438	270351	28-02-2014	1	1
29258453508	270384	27-02-2014	5	3
51750724947	273420	24-02-2014	6	5
93274880719	271509	24-02-2014	11	6
51750724947	273420	23-02-2014	6	5

We want to determine, **for each customer** (`cust_id`), the **minimum**, **maximum** and the **total amount** spent from the `total_amt` column. We also want to know **how many types of stores** the customer has made a transaction in (`store_type` column).

We can perform these calculations using a **`groupby`** operation:

- **Split** the transactions by the **customer identifier**.
- For the **`total_amt`** column, calculate the minimum (`min`), maximum (`max`) and the sum (`sum`). For the **`store_type`** column, count the **number of unique modalities taken**.
- **Combine** the results in a `DataFrame`.

To find the number of unique modalities taken by the `store_type` column, we will use the following **`lambda`** function:

```
import numpy as np

n_modalities = lambda store_type:
len(np.unique(store_type))
```

- The `lambda` function must take as argument a **column** and return a **number**.
- The **`np.unique`** function determines the unique **modalities** that appear in a sequence.
- The **`len`** function counts the number of elements in a sequence, i.e. its length.

Thus, this function will allow us to determine the number of unique modalities for the `store_type` column.

To apply these functions in the `groupby` operation, we'll use a dictionary whose **keys** are the **columns** to process and the **values** the **functions** to use.

```
functions_to_apply = {
# Classic statistical methods can be entered with
# strings
'total_amt': ['min', 'max', 'sum'],
'store_type': n_modalities
}
```

This dictionary can now be fed into the **`agg`** method to perform the `groupby` operation:

```
transactions.groupby('cust_id').agg(functions_to_a
```

Which produces the following `DataFrame`:

cust_id	total_amt min	max	sum	store_type <lambda>
266783	-5838.82	5838.82	3113.89	2
266784	442	4279.66	5694.07	3
266785	-6828.9	6911.77	21613.8	3
266788	1312.74	1927.12	6092.97	3

cust_id	total_amt min	max	sum	store_type <lambda>
266794	-135,915	4610.06	27981.9	4

- **(a)** Using a `groupby` operation, determine for each customer from the quantity of items purchased in a transaction (**`qty`** column):
- The maximum quantity.
- The minimum quantity
- The median quantity.

  You will have to filter the transactions whose quantity is negative. For this, you can use **conditional indexing** (`qty[qty > 0]`) over the column in a `lambda` function.

  Out[126]:

	qty		
	max	**min**	**median**
cust_id			
266783	4	1	2.5
266784	5	2	3.0
266785	5	2	5.0
266788	4	1	1.5
266794	4	1	3.0

Another way of grouping and summarizing data is to use the `crosstab` function of `pandas` which, as its name suggests, is used to crosstab the data in the columns of a `DataFrame`.

A crosstab allows us to visualize the **appearance frequency** of **pairs of modalities** in a `DataFrame`.

Example :

In the `transactions` `DataFrame`, we want to know which are the most frequent category and subcategory pairs (`prod_cat_code` and `prod_subcat_code` columns)

The `crosstab` function of pandas gives us this result:

```
column1 = transactions['prod_cat_code']
column2 = transactions['prod_subcat_code']
pd.crosstab(column1, column2)
```

This instruction produces the following `DataFrame`:

prod_subcat_code prod_cat_code	-1	1	2	3	4	5	6	7
1	4	1001	0	981	958	0	0	0
2	4	934	0	1040	1005	0	0	0
3	11	0	0	0	1020	950	0	0
4	5	993	0	0	988	0	0	0
5	3	0	0	1023	0	0	984	103
6	5	0	1002	0	0	0	0	0

The `(i, j)` cell of the resulting `DataFrame` contains the number of rows of the `DataFrame` having the modality `i` for column 1 and the modality `j` for column 2.

Thus, it is easy to determine, for example, that **the dominant subcategories** of the category **4** are 1 and 4.

The **`normalize`** argument of `crosstab` allows to display frequencies as a percentage.

Thus, the argument **`normalize = 1`** normalizes the table over the axis 1 of the crosstab, i.e. its **columns**:

```
We recover the year of the transaction.
column1 = transactions['tran_date'].apply(lambda
x: x.split('-')[2]).astype(int)

column2 = transactions[store_type]

pd.crosstab(column1,
            column2,
            normalize = 1)
```

This produces the following `DataFrame`:

store_type tran_date	Flagship store	MBR	TeleShop	e-Shop
2011	0.291942	0.323173	0.283699	0.306947
2012	0.331792	0.322093	0.336767	0.322886
2013	0.335975	0.3115	0.332512	0.320194
2014	0.0402906	0.0432339	0.0470219	0.0499731

This `DataFrame` allows us to say that **33.5975%** of the transactions made in a **`'Flagship store'`** took place in **2013**.

Conversely, by entering the argument **`normalize = 0`**, the crosstab is normalized over each **row**:

store_type tran_date	Flagship store	MBR	TeleShop	e-Shop
2011	0.191121	0.21548	0.182617	0.410781
2012	0.20096	0.198693	0.20056	0.399787
2013	0.205522	0.194074	0.2	0.400404
2014	0.173132	0.189215	0.198675	0.438978

Normalizing over the rows allows us to deduce that the transactions made in an **`'e-Shop'`** account for **41.0781%** of the transactions of the year **2011**.

In the `covid_tests.csv` file, we have a dataset of 200 COVID-19 tests. The columns of this dataset are as follows:

- `'patient_id'`: ID of the patient tested.
- `'test_result'`: Result of the test. Equals 1 if the patient is tested positive and 0 otherwise.
- `'infected'`: Equals 1 if the patient was actually infected and `0` otherwise.

- **(b)** Load the dataset contained in the `covid_tests.csv` file. The values are separated by the character `';'`.
- **(c)** Use the `pd.crosstab` function to determine the number of **False Negatives** produced by this test. (A false negative occurs when the test determines that the patient is not infected when they actually are.)
- **(d)** What is the false positive rate of the test? The false positive rate is the **proportion** of false positives in relation to the number of people that are not infected. (A false positive occurs when the test determines the patient is infected when they are not.)

infected 0 1
test_result
0 119 3
1 7 71
---------- normalize = 1 3/3+71

infected 0 1
test_result
0 0.944444 0.040541
1 0.055556 0.959459
---------- normalize = 0 3/3+119
infected 0 1
test_result
0 0.975410 0.024590
1 0.089744 0.910256

Conclusion and recap¶

In this notebook you have learned to:

- **Filter** the rows of a `DataFrame` with **multiple conditions** using the binary operators **&**, **|** and **~**:

```
# Year equal to 1979 and surface area greater
than 60
df[(df['year'] == 1979) & (df['surface'] > 60)]

Year greater than 1900 or neighborhood equal to
'Père-Lachaise'.
df[(df['year'] > 1900) | (df['neighborhood'] ==
'Père-Lachaise')]
```

- **Merge** `DataFrames` using the **`concat`** function and the **`merge`** method.

```
# Vertical concatenation
pd.concat([df1, df2], axis = 0)

# Horizontal concatenation
pd.concat([df1, df2], axis = 1)

# Different types of joins
df1.merge(right = df2, on = 'column', how =
'inner')
df1.merge(right = df2, on = 'column', how =
'outer')
df1.merge(right = df2, on = 'column', how =
'left')
df1.merge(right = df2, on = 'column', how =
'right')
```

- **Sort** and **order** the values of a `DataFrame` with the **`sort_values`** and **`sort_index`** methods.

```
# Sorting a DataFrame by 'column' in ascending
order
df.sort_values(by = 'column', ascending = True)
```

- Perform a complex **`groupby` operation** using `lambda` functions and the **`groupby`** and **`agg`** methods.

```
functions_to_apply = {
'column1': ['min', 'max'],
'column2' : [np.mean, np.std],
'column3' : lambda x: x.max() - x.min()
}

df.groupby('column_to_group_by').agg(functions_to
```

In this introductory module to `Python` for Data Science, you have learned how to create, clean and manipulate a dataset with Python using the **`numpy`** and **`pandas`** modules.

You now have all the tools to approach more advanced Data Science notions such as Machine Learning or Data Visualization :)