DataScientest • com

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.

In [1]:

```
### Imports ###

import pandas as pd

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

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

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

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

### Handling NAs ###

#  We replace the NAs in 'prod_subcat_code' by -1
transactions['prod_subcat_code'] = transactions['prod_subcat_code'].fillna(-1).astype(int)

# We compute the mode of 'store_type'
store_type_mode = transactions['store_type'].mode()

# We replace the NAs of 'store_type' by its mode
transactions['store_type'] = transactions['store_type'].fillna(transactions['store_type'].mode()[0])

# Removal of rows where 'rate', 'tax' and 'total_amt' are all NAs
transactions = transactions.dropna(axis = 0, how = 'all', subset = ['rate', 'tax', 'total_amt'])
```

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
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

In [2]:

```
# Insert your code here
df = transactions.copy()

print('solution a', 50 * '-')

(df.head())

print('solution b', 50 * '-')

e_shop = df[(df['store_type'] == 'e-Shop') & (df['total_amt'] > 5000)]
e_shop

print('solution c', 50 * '-')

tele_shop = df.loc[(df['store_type'] == 'TeleShop') & (df['total_amt'] > 5000)]
tele_shop

print('solution d', 50 * '-')

print('e_shop shape :', e_shop.shape[0], 'tele_shop shape :', tele_shop.shape[0])
```

```
solution a --------------------------------------------------
solution b --------------------------------------------------
solution c --------------------------------------------------
solution d --------------------------------------------------
e_shop shape : 1185 tele_shop shape : 532
```

Hide solution

In [3]:

```
# Creation of e_shop et teleshop
e_shop = transactions[(transactions['store_type'] == 'e-Shop') & (transactions['total_amt'] > 5000)]


teleshop = transactions[(transactions['store_type'] == 'TeleShop') & (transactions['total_amt'] > 5000)]


# We count the number of rows of each DataFrame. Other solutions are possible.
print('Number of transactions over 5000€ for e-shop :', len(e_shop))
print('Number of transactions over 5000€ for teleshop :', len(teleshop))
```

```
Number of transactions over 5000€ for e-shop : 1185
Number of transactions over 5000€ for teleshop : 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.

In [4]:

```
# Insert your code here
print('solution e', 50 * '-')
customer = pd.read_csv('customer.csv')
customer.head()
prod_cat_info = pd.read_csv('prod_cat_info.csv')
prod_cat_info.head()
```

```
solution e --------------------------------------------------
```

Out[4]:

	prod_cat_code	prod_cat	prod_sub_cat_code	prod_subcat
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

In [5]:

```
print('solution f', 50 * '-')
mode_gender = customer['Gender'].mode()[0]
customer['Gender'].fillna(mode_gender, inplace=True)

mode_city_code = customer['city_code'].mode()[0]
customer['city_code'].fillna(mode_city_code, inplace=True)

customer.isna().sum()
```

```
solution f --------------------------------------------------
```

Out[5]:

```
customer_Id    0
DOB            0
Gender         0
city_code      0
dtype: int64
```

Hide solution

In [6]:

```
customer = pd.read_csv('customer.csv')
prod_cat_info = pd.read_csv('prod_cat_info.csv')

customer['Gender'] = customer['Gender'].fillna(customer['Gender'].mode()[0])
customer['city_code'] = customer['city_code'].fillna(customer['city_code'].mode()[0])
```

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

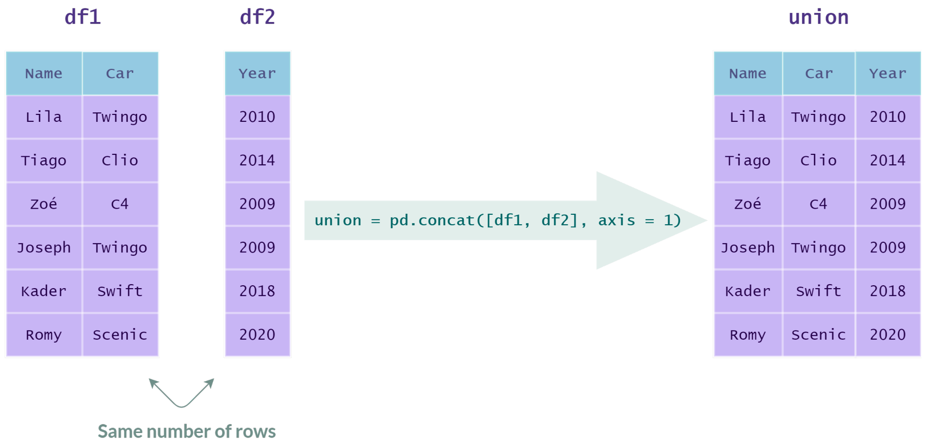
Concatenation of `DataFrames` with `concat`

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`).



In [7]:

```python
# Insert your code here

print('solution a', 50 * '-')
column_part1 = df.columns[:5]
part1 = df[column_part1]

column_part2  = df.columns[5:]
part2 = df[column_part2]

part1

print('solution b', 50 * '-')

union  = pd.concat([part1,part2], axis=1)
union

print('solution c', 50 * '-')

union_axis0 = pd.concat([part1,part2], axis=0)
union_axis0
```

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

Out[7]:

transaction_id	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type
80712190438	270351.0	28-02-2014	1.0	1.0	-5.0	NaN	NaN	NaN	NaN
29258453508	270384.0	27-02-2014	5.0	3.0	-5.0	NaN	NaN	NaN	NaN
51750724947	273420.0	24-02-2014	6.0	5.0	-2.0	NaN	NaN	NaN	NaN
93274880719	271509.0	24-02-2014	11.0	6.0	-3.0	NaN	NaN	NaN	NaN
51750724947	273420.0	23-02-2014	6.0	5.0	-2.0	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...
94340757522	NaN	NaN	NaN	NaN	NaN	1264.0	132.720	1396.720	e-Shop
89780862956	NaN	NaN	NaN	NaN	NaN	677.0	71.085	748.085	e-Shop
85115299378	NaN	NaN	NaN	NaN	NaN	1052.0	441.840	4649.840	MBR
72870271171	NaN	NaN	NaN	NaN	NaN	1142.0	359.730	3785.730	TeleShop
77960931771	NaN	NaN	NaN	NaN	NaN	447.0	46.935	493.935	TeleShop

45838 rows × 9 columns

Hide solution

In [8]:

```python
# Splitting of the transactions DataFrame
part_1 = transactions[transactions.columns[:4]]
part_2 = transactions[transactions.columns[4:]]

# Reconstitution of the transactions DataFrame by concatenation
union = pd.concat([part_1,part_2], axis = 1)

# If we were to concatenate by filling in the argument "axis = 0", we would obtain a DataFrame where half of
# the valuers are NAs
#
# This is due to the fact that the argument 'axis = 0' forces the pd.concat function to create new ROWS
# in part_1 but it cannot fill them correctly since part_1 and part_2 have no columns in common.
```

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.

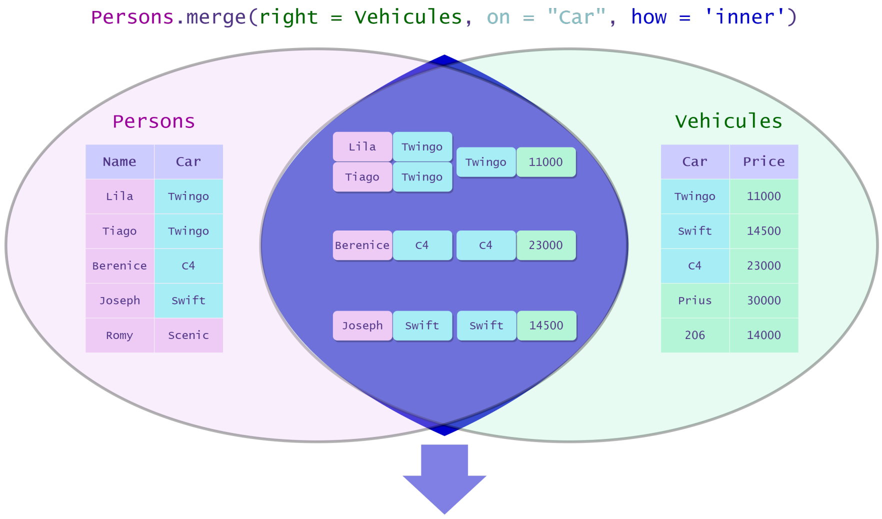
The `how` parameter can take 4 values (`'inner'`, `'outer'`, `'left'`, `'right'`) that we will illustrate on the two `DataFrames` named `Persons` and `Vehicles` below:

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**.

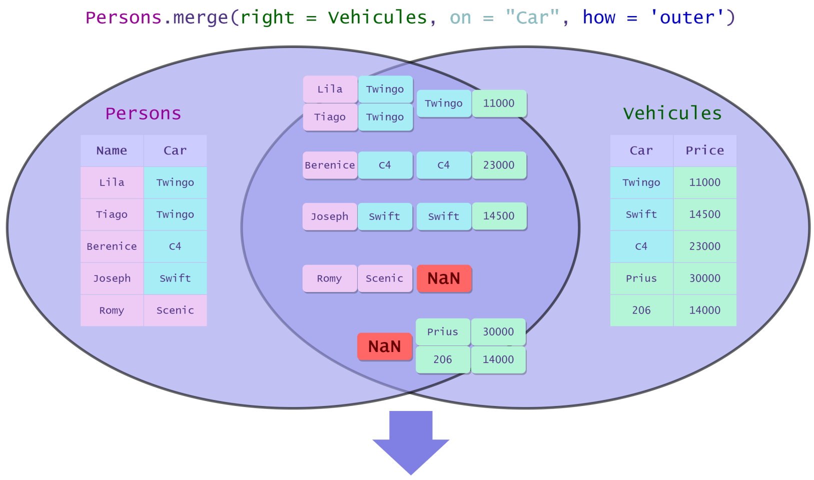
The result of the inner join `Persons.merge(right = Vehicles, on = 'Car', how = 'inner')` is shown below:



Name	Car	Price
Lila	Twingo	11000
Tiago	Twingo	11000
Berenice	C4	23000
Joseph	Swift	14500

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

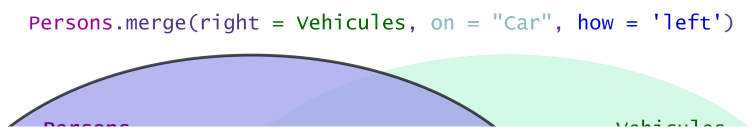
The result of the outer join `Persons.merge(right = Vehicles, on = 'Car', how = 'outer')` is shown below:

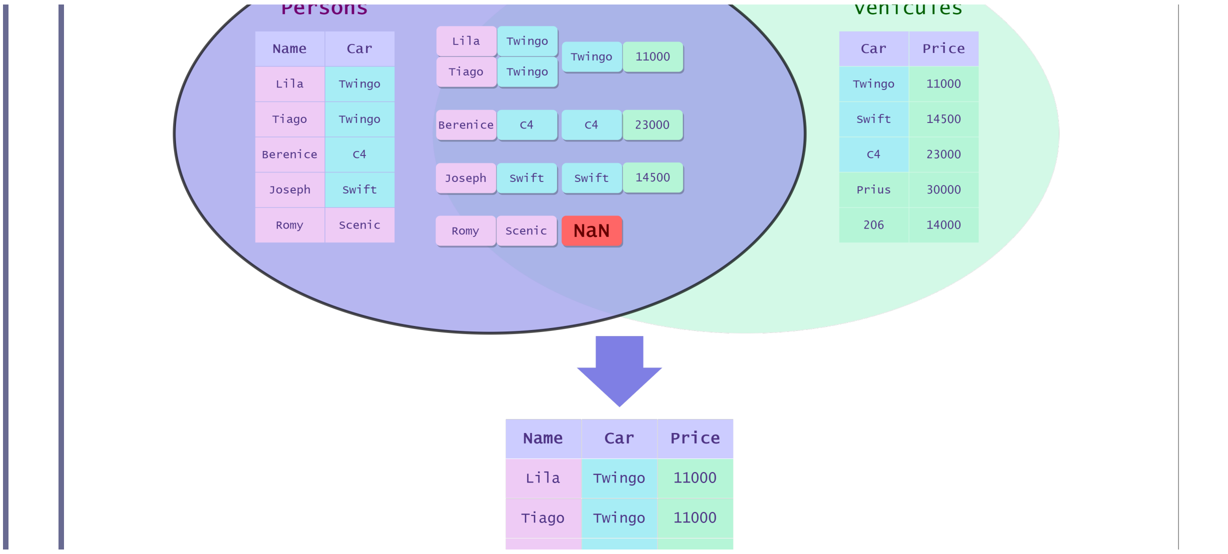


Name	Car	Price
Lila	Twingo	11000
Tiago	Twingo	11000
Berenice	C4	23000
Joseph	Swift	14500
Romy	Scenic	NaN
NaN	Prius	30000
NaN	206	14000

- `'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:





In [9]:

```python
# Insert your code here

# d)
customer = customer.rename({'customer_Id' : 'cust_id'} , axis= 'columns')
customer

# e)
fusion = transactions.merge(customer, on='cust_id', how='left')
fusion

#f)
fusion.isna().sum()

# g)
fusion.head()
```

Out[9]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	DOB	Gender	city_code
0	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	e-Shop	26-09-1981	M	5.0
1	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	e-Shop	11-05-1973	F	8.0
2	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0
3	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	e-Shop	08-06-1981	M	3.0
4	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0

Hide solution

In [10]:

```python
# We rename the 'customer_Id' column to 'cust_id' for merging
customer = customer.rename(columns = {'customer_Id': 'cust_id'})

# Left join between transactions and customer on the 'cust_id' column
fusion = transactions.merge(right = customer, on = 'cust_id', how = 'left')

# The merging did not produce NAs
fusion.isna().sum()

# The columns DOB, Gender, city_code have been added to transactions
fusion.head()
```

Out[10]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	DOB	Gender	city_code
0	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	e-Shop	26-09-1981	M	5.0
1	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	e-Shop	11-05-1973	F	8.0
2	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0
3	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	e-Shop	08-06-1981	M	3.0
4	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0

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` :

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

In [11]:

```
# Insert your code here
fusio = fusion.set_index(transactions.index)
fusion
```

Out[11]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	DOB	Gender	city_code
0	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	e-Shop	26-09-1981	M	5.0
1	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	e-Shop	11-05-1973	F	8.0
2	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0
3	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	e-Shop	08-06-1981	M	3.0
4	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0
...	...	...	...	...	...	...	...	...	...	...	...	...
22914	274550	25-01-2011	12	5	1	1264.0	132.720	1396.720	e-Shop	21-02-1972	M	7.0
22915	270022	25-01-2011	4	1	1	677.0	71.085	748.085	e-Shop	27-04-1984	M	9.0
22916	271020	25-01-2011	2	6	4	1052.0	441.840	4649.840	MBR	20-06-1976	M	8.0
22917	270911	25-01-2011	11	5	3	1142.0	359.730	3785.730	TeleShop	22-05-1970	M	2.0
22918	271961	25-01-2011	11	5	1	447.0	46.935	493.935	TeleShop	15-01-1982	M	1.0

22919 rows × 12 columns

Hide solution

In [12]:
```python
# We retrieve the index of transactions
new_index = transactions.index

# We set the new index of fusion
fusion = fusion.set_index(new_index)
fusion.head()
```

Out[12]:

	cust_id	tran_date	prod_subcat_code	prod_cat_code	qty	rate	tax	total_amt	store_type	DOB	Gender	city_code
transaction_id												
80712190438	270351	28-02-2014	1	1	-5	-772.0	405.300	-4265.300	e-Shop	26-09-1981	M	5.0
29258453508	270384	27-02-2014	5	3	-5	-1497.0	785.925	-8270.925	e-Shop	11-05-1973	F	8.0
51750724947	273420	24-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0
93274880719	271509	24-02-2014	11	6	-3	-1363.0	429.345	-4518.345	e-Shop	08-06-1981	M	3.0
51750724947	273420	23-02-2014	6	5	-2	-791.0	166.110	-1748.110	TeleShop	27-07-1992	M	8.0

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:

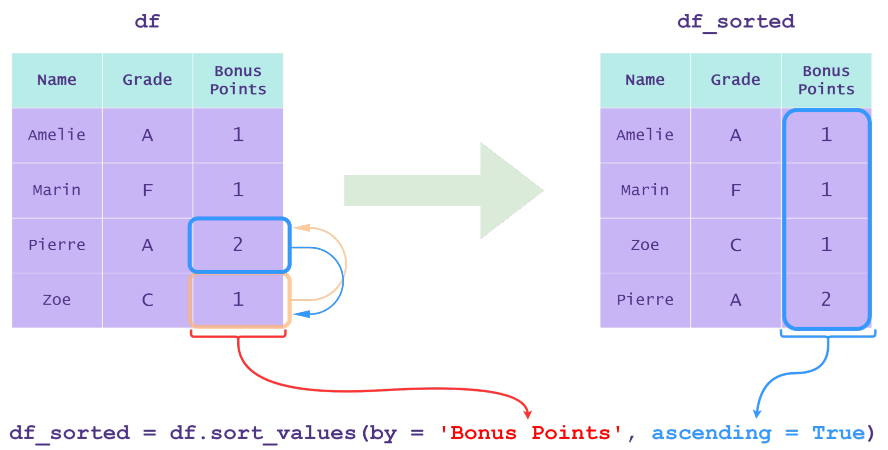
Consider the `DataFrame` `df` describing students:

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'` :

```python
# 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:

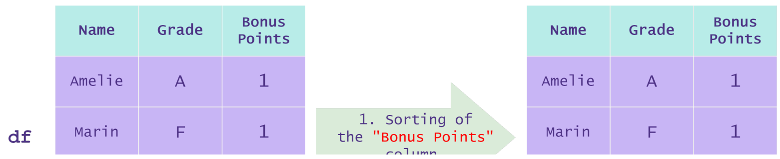


The rows of the `DataFrame` `df_sorted` are therefore sorted in **ascending order** of the **`'Bonus points'`** column. However, if we look at the column `'Grade'` , we see that it is not sorted alphabetically for the common values of `'Bonus Points'` .

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

```python
# 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:



In [13]:

```python
# Definition of the data dictionnaries
data_boats = {'boat_name'         : ['Julia', 'Siren', 'Sea Sons', 'Hercules', 'Cesar', 'Minerva'],
              'color'             : ['blue', 'green', 'red', 'blue', 'yellow', 'green'],
              'reservation_number': [2, 3, 6, 1, 4, 5],
              'n_reservations'    : [34, 10, 20, 41, 12, 16]}

data_clients = {'client_id'     : [91, 154, 124, 320, 87, 22],
                'client_name'   : ['Marie', 'Anna', 'Yann', 'Lea', 'Marc', 'Yassine'],
                'reservation_id': [1, 2, 3, 7, 9, 5]}

# Instantiation of the DataFrames
boats = pd.DataFrame(data_boats)
clients = pd.DataFrame(data_clients)
```

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.

In [14]:
```
# Insert your code here
#b)
boats.rename(columns = {'reservation_number' :'reservation_id' }, inplace=True)

#c)
boats_clients = boats.merge(clients, how='left', on='reservation_id')
boats_clients
#d)
boats_clients = boats_clients.set_index('boat_name')
boats_clients
#e)
#boats_clients.loc[boats_clients.index.isin(['Julia', 'Siren'])]
boats_clients.loc[['Julia', 'Siren']]

#f)
#boats_clients.client_name.fillna('not_reserved', inplace=True)
boats_clients[boats_clients.client_name.isna()].index

#g)
boats_clients.sort_values('n_reservations', ascending=False)
```

Out[14]:

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

Hide solution

In [15]:
```
# We rename the column 'number_reservation'
boats = boats.rename(columns = {'reservation_number': 'reservation_id'})

# We perform the left join between boats and clients
boats_clients = boats.merge(clients, on = 'reservation_id', how = 'left')

# We set the column 'boat_name' as the index of boats_clients
boats_clients = boats_clients.set_index("boat_name")

# Who reserved 'Julia' and 'Siren'?
print("The client who reserved 'Julia' is:", boats_clients.loc['Julia', 'client_name'])
print("The client who reserved 'Siren' is:", boats_clients.loc['Siren', 'client_name'])
print("\n")

# Which boats have not been reserved?
boats_not_reserved = boats_clients[boats_clients['client_name'].isna()]
print("The boats which have not been reserved are:", [boat for boat in boats_not_reserved.index])

# Which client reserved the BLUE boat with the MOST reservations to date?

boats_clients.sort_values(by = 'n_reservations', ascending = False)
# Marie
```

```
The client who reserved 'Julia' is: Anna
The client who reserved 'Siren' is: Yann


The boats which have not been reserved are: ['Sea Sons', 'Cesar']
```

Out[15]:

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

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.

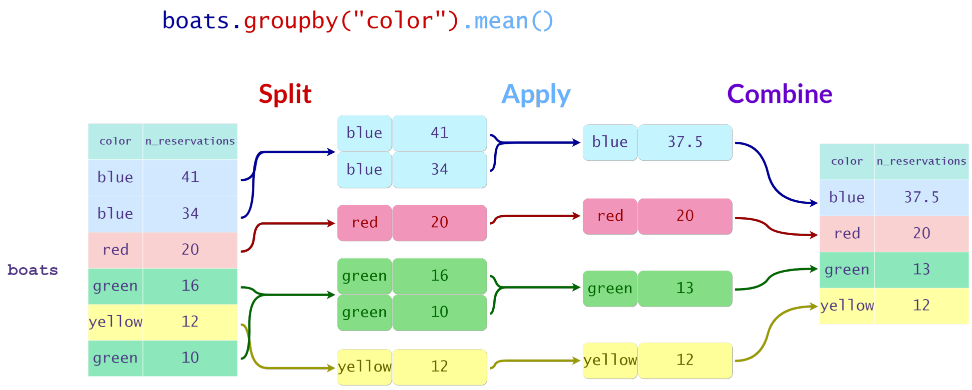
Example:

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:



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	prod_cat_code	qty	rate	tax	total_amt	store_type
80712190438	270351	28-02-2014	1	1	-5	-772	405.3	-4265.3	e-Shop
29258453508	270384	27-02-2014	5	3	-5	-1497	785.925	-8270.92	e-Shop
51750724947	273420	24-02-2014	6	5	-2	-791	166.11	-1748.11	TeleShop
93274880719	271509	24-02-2014	11	6	-3	-1363	429.345	-4518.35	e-Shop
51750724947	273420	23-02-2014	6	5	-2	-791	166.11	-1748.11	TeleShop

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_apply)
```

Which produces the following `DataFrame` :

In [16]:

```python
# Insert your code here

func = {'qty' : ['max', 'min', 'median']}

transactions.loc[transactions.qty > 0].groupby('cust_id').agg(func).head()
```

Out[16]:

	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

Hide solution

In [17]:

```python
# Maximal Quantity
max_qty = lambda qty: qty[qty > 0].max()

# Minimal Quantity
min_qty = lambda qty: qty[qty > 0].min()

# Median Quantity
median_qty = lambda qty : qty[qty > 0].median()

# Definition of the dictionnary of functions to apply
functions_to_apply = {
    'qty' : [max_qty, min_qty, median_qty]
}

# Groupby Operation
qty_groupby = transactions.groupby('cust_id').agg(functions_to_apply)

# For a better display, we can rename the columns produced by the groupby operation
qty_groupby.columns.set_levels(['max_qty', 'min_qty', 'median_qty'], level=1, inplace = True)

# Display of the first rows of the DataFrame produced by the groupby operation
qty_groupby.head()
```

```
<ipython-input-17-47f5ec3f8afd>:19: FutureWarning: inplace is deprecated and will be removed in a future version.
  qty_groupby.columns.set_levels(['max_qty', 'min_qty', 'median_qty'], level=1, inplace = True)
```

Out[17]:

	qty		
	max_qty	min_qty	median_qty
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**	**8**	**9**	**10**	**11**	**12**
1	4	1001	0	981	958	0	0	0	0	0	0	0	0
2	4	934	0	1040	1005	0	0	0	0	0	0	0	0
3	11	0	0	0	1020	950	0	0	966	976	945	0	0
4	5	993	0	0	988	0	0	0	0	0	0	0	0
5	3	0	0	1023	0	0	984	1037	0	0	998	1029	962
6	5	0	1002	0	0	0	0	0	0	0	1025	1013	1057

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 **Flagship store** **MBR** **TeleShop** **e-Shop**

In [18]:

```
# Insert your code here
#b)
df = pd.read_csv('covid_tests.csv', sep=';')
df

#c)
col1 = df.test_result
col2 = df.infected

print(pd.crosstab(col1, col2))
print(30*'-', 'normalize = 1   3/3+71 ')
#d) normalize = 0

print(pd.crosstab(col1, col2, normalize=1))

# normalize = 1
print(30*'-', 'normalize = 0     3/3+119' )
print(pd.crosstab(col1, col2, normalize=0))
```

```
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
```

Hide solution

In [19]:

```python
# Loading the dataset in 'covid_tests.csv'
covid_df = pd.read_csv("covid_tests.csv", sep = ';', index_col = 'patient_id')
covid_df.head()


# Crosstab of the test results with reality
pd.crosstab(covid_df['test_result'],
            covid_df['infected'])

# There are 3 false negatives


pd.crosstab(covid_df['test_result'],
            covid_df['infected'],
            normalize = 1)

# The false positive rate is about 5,6%
# 94,4% of healthy people are true negatives
```

Out[19]:

infected	0	1
test_result		
0	0.944444	0.040541
1	0.055556	0.959459

Conclusion and recap

In this notebook you have learned to:

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

```python
# 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.

```python
# 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.

```python
# 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.

```python
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_apply)
```

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 :)

Validate