NUMPY

Python → doesn't have built-in support for arrays (have lists, but...)

Numpy → provides support for arrays and a wide range of mathematical functions

```
import numpy as np

arr1D = np.array([1,2,3,4,5])
zeroes = np.zeroes(5)   ※ [0. 0. 0. 0. 0.]
ones = np.ones(5)   ※ [1. 1. 1. 1. 1.]
empty = np.empty(5)   ※ random initialization
sequence = np.arange(0,10,2)   ※ [ 0 2 4 6 8]

sales_month1 = np.array([120, 150, 90])
sales_month2 = np.array([130, 160, 80])
tot_sales = sales_month1 + sales_month2
          ※  [ 260, 310, 170]
```

+ | - | * | / fanno le operazioni uno a uno

```
array1 = np.array([1,2,3])
array2 = np.array([4,5,6])
dot_product = np.dot(array1, array2)
    ※ out: 32   (1*4 + 2*5 + 3*6)
```

Indexing and slicing are as for lists

toys = np.array(['teddy bear', 'robot', 'doll', 'ball', 'yo-yo'])

toys.shape → '(5,)'] = dimensione

↳ senza () (è una proprietà)

toys_boxes = np.array(['teddy bear', 'robot', 'car'],
['doll', 'ball', 'yo-yo'])

toys_boxes.shape → = '(2,3)'

* toys = np.array(['teddy bear', 'robot', 'doll',
'ball', 'yo-yo', 'car'])

toys.reshape(3,2)

※ toys :

[['teddy bear', 'robot'],
['doll', 'ball'],
['yo-yo', 'car']]

---

data = np.array([12, 43, 36, 32, 51, 18, 79, 7])

※ Boolean indexing : boolean mask

* bool_array = data > 30 ※ [False True True True ...]

filtered_data = data[bool_array]

↳ ※ [43, 36, 32, 79]

↳ we can include boolean conditions (...) & (....) | (....)

---

grades = np.array([18, 30, 25])

np.mean(grades) ※media

np.median(grades)

np.mode(grades) ※moda

np.var(grades) ※varianza

np.std(grades) ※deviazione standard

Ripassino:

* varianza: misura quanto i dati sono dispersi rispetto alla media. Si calcola facendo la media dei quadrati delle differenze tra ogni valore e la media
* deviazione standard: radice quadrata della varianza indica, in media, quanto i valori si discostano dalla media (quanto deviano dallo standard)

PANDAS

```
import pandas as pd

data_list = ['apple', 'banana', 'cherry']
df_list = pd.DataFrame(data_list, columns=['Fruit'])

data_dict = {
    "Fruits": ['apple', 'banana', 'cherry'],
    "count": [10, 20, 15]
}
data_df = pd.DataFrame(data_dict)

dict = {
    "name_of_column": [...values...], ...}

df.head() → first 5 rows
df.tail() → last 5 rows
df.info() → overview on df

data1 = {"Fruit": ['apple', 'banana'],
```

```
                count": [10, 20] }
df1 = pd.DataFrame(data1)
data2 = { "Fruit": ['cherry', 'date'],
          "count": [15, 25] }
df2 = pd.DataFrame(data2)
df_combined = pd.concat([df1, df2])
```

	Fruit	count
0	apple	10
1	banana	20
0	cherry	15
1	date	25

> notice that indices are preserved 2

to avoid this

```
df_combined = pd.concat([df1, df2], ignore_index = True)
```

	Fruit	count
0	apple	10
1	banana	20
2	cherry	15
3	date	25

In a DataFrame -> each column is a Series object

essentially a list of values with an associate label (index) for each value

```
fruits = ['apple', 'banana', 'cherry']
series = pd.Series(fruits)
```

↳ Series also have methods like head(), tail(), describe()

* info() → informazioni generali sul DataFrame (tipo di dati / numero di valori non nulli / nomi delle colonne)
* describe() → statistiche descrittive

By default → pandas assigns integer labels to the rows but we can set any column as the index

```
df = pd.DataFrame ( {
    "Name" : [ "Alice" , "Bob", "John"],
    "Age": [25, 22, 30],
    "City": [ "New York ", "Los Angeles", "Chicago"]
})
```

```
df. set_index ("Name ", inplace = True)
```

Accessing data using the index is performed with loc[] method for label based indexing and iloc[] method for integer-based indexing.

↳ to reset indexing back to default:

```
df. reset_index (inplace = True)
```

Per rinominare le colonne:

```
df. rename ( { "Name": "Student Name", "Age" : "Student Age"},
                                        inplace = True )
```

↳ Let's suppose we set "Name" as index:

```
print ( df.loc [ ["Alice", "John"], ["Age", "City"]]
  # output:
  #          Age      City
  # Name
  # Alice    25       New York
  # John     30       Chicago
```

FILTERING

```
data = {
    'name' : ['Alice' ,'Bob' ,'charlie', 'Dave' , 'Eve'],
    'age' : [12, 13, 14, 13, 12],
    'grade_level ' : [6,7,8,7,6]
}
students_df = pd.DataFrame(data)
#Filter 7th grade students
grade_seven_students = students_df [students_df [ 'grade_level')==7]
```

Boolean mask

if we wanted 6th and 7th grade:

```
students_df [ 'grade_level').isin ([6,7])
```

```
data = { 'Name' : [ 'TOM' , 'Mik', 'Jay', 'LOl', 'Min' , 'Arg' ],
         'Type' : [ 'Dog' , 'Dog' , 'cat' , 'cat', 'Dog', 'Bird' ],
         'Weight' : [ 12, 15, 8, 9, 14, 17 ]}
pets_df = pd.Dataframe (data)
print ( pets_df ['Type' ] .value_counts())
```

count of each 'Type'

```
Dog    3
Cat    2
```

Bird 1

```
pets_df.groupby('Type').agg({'Weight': 'mean'})
```

↓

Bird	1.0
Cat	8.5
Dog	13.67

```
sorted_df = pets_df.sort_values('Weight')
```

↓

	Name	Type	Weight
5	Arg	Bird	1
2	Mik	Cat	8
	...		

`where()` method → useful when we want to select data but replace the data that doesn't satisfy the condition with a custom value rather than simply discarding them.

Es:

```
print(scores_df['score'].where(scores_df['score'] > 85, other='Fail'))
```

↓

gli score <= 85 sono sostituiti dal valore 'Fail'

Missing Data

```
data = {'Name': ['Anna', 'Bob', 'Charlie', 'David', None],
        'Score': [85, 88, None, 92, 90]}
```

df = pd.DataFrame(data)

method `isnull()` → allows us to identify missing values.

print(df.isnull())

	Name	Score
0	False	False
1	False	False
2	False	True
3	False	False
4	True	False

`notnull()` works the same, but returns the exact opposite.

How to handle missing values? → common strategy: remove rows with None values using `dropna()`

print(df.dropna())

	Name	Score
0	Anna	85.0
1	Bob	88.0
3	David	92.0

We can use the `subset` argument to specify which columns to check for missing values

print(df.dropna(subset=['Score']))

	Name	Score
0	Anna	85.0
1	Bob	88.0
3	David	92.0
4	None	90.0

Another strategy is to fill missing value with a specific value such as the mean, median, mode.

↳ `fillna()`

{ bfill = backward-fill
ffill = forward-fill

```
print (df.fillna(0))
```

	Name	Score
0	Anna	85.0
1	Bob	88.0
2	Charlie	0.0
3	David	92.0

```
print (df.fillna(method = 'bfill'))
```

↓ takes value after

	Name	Score
0	Anna	85.0
1	Bob	88.0
2	Charlie	92.0
3	David	92.0

```
mean_score = df['Score'].mean()
print (df.fillna (mean_score)
```

CREATING NEW COLUMNS

```
df = pd.DataFrame ( {
        "Item" : ["Apples", "Bananas", "oranges"],
        "Price" : [1.5, 0.5, 0.75],
        "Quantity" : [10, 20, 30] })

*Create a new column "Total" which is Price * Quantity
df["Total"] = df["Price"] * df["Quantity"]
*Add "Location" column with static value
df["Location"] = "New York"
```

```
df = pd.DataFrame ( {
        "Student" : ["Alice", "Bob", "Charlie"],
        "Score" : [42, 87, 56] })
* Create a new column "Status" that is "Pass" if
* "Score" > 40 else "Fail"
df["Status"] = np.where (df["Score"] > 40, "Pass", "Fail")
```

print (df)

	Student	Score	Status
0	Alice	42	Pass
1	Bob	37	Fail
2	Charlie	56	Pass

DATA CLEANING AND TRANSFORMATION

Recognizing & dealing with inconsistencies in Data

```
sizes = ['XL', 'S', 'M', 'X-L', 'xl', 'S', 'L', 'XL', 'M']
df = pd.DataFrame(sizes, columns = ['size'])
#use value_counts() to spot inconsistent values
print(df['size'].value_counts())
```

↳

XL	2
X-L	1
xl	1
S	2
M	2
L	1

problem

```
df.replace(['X-L', 'xl'], 'XL', inplace = True)
print(df['size'].value_counts())
```

↳

XL	4
S	2
M	2
L	1

↳ NEXT STEP → SCANNING FOR OUTLIERS (= valori anomali)

One common method to detect outliers is using **Interquartile Range (IQR)**

→ IQR method suggests that any value **below** $Q_1 - 1.5 \cdot IQR$ and **above** $Q_3 + 1.5 \cdot IQR$ are considered to be **outliers**. Where:

- Q_1 is the first quartile
- Q_3 is the third quartile
- IQR is the inter-quartile range

n.b. un quartile divide i dati ordinati in 4 parti uguali

interquartile range = $Q_3 - Q_1$

Let's use **IQR method** to identify and filter out outliers in a dataset:

```
import pandas as pd

data = [1, 1.2, 1.1, 1.05, 1.5, 1.4, 9]   # 9 = outlier
df = pd.DataFrame(data, columns = ['values'])

Q1 = df['values'].quantile(0.25)
Q3 = df['values'].quantile(0.75)

IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

no_outliers_df = df[(df['values'] >= lower_bound) &
                    df['values'] <= upper_bound)]
```

DATA TRANSFORMATION

↳ required when data needs adjustment to suit a specific analysis or model-building exercise.

↓

The need might be to bring skewed data to normality or harmonize different scales of variables.

↓

For this purpose the `scikit-learn` library comes in handy

↓

We can create a `StandardScaler` object → then use the `fit_transform` method

↓

We select the column of the df with double square brackets `[['Feature2']]` to ensure it's treated as a data-frame (required by scikit-learn transformers) rather than a Series.

↓

Quello che fa lo scaler è trasformare i dati per avere media 0 e deviazione standard 1.

↓

Serve perché molti algoritmi di Machine-Learning funzionano meglio se le variabili sono sulla stessa scala (per esempio per evitare che una variabile con numeri grandi "schiacci" le altre)

↓

I valori "scalati" mantengono le stesse relazioni tra loro (le distanze relative non cambiano) ma sono espressi in una nuova unità.

```
scaler = StandardScaler()
```

```
df['scaledSize'] = scaler.fit_transform(df[['FeatureSize']])
```

↳ crea un nuovo Serie nel df 'scaledSize'

Se invece di StandardScaler() usiamo MinMaxScaler() dopo lo scalamento tutti i valori saranno in un range tra 0 e 1

ADVANCED DATA ANALYSIS WITH PANDAS

pd.merge()

```
students = pd.DataFrame({
    'student_id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [15, 16, 17]
})

performance = pd.DataFrame({
    'student_id': [1, 3, 4],
    'grade': ['A', 'B', 'A'],
    'attendance': [95, 85, 100]
})

students_merged = pd.merge(students, performance, on='student_id', how='left')
```

↳

	student_id	name	age	grade	attendance
0	1	Alice	15	A	95.0
1	2	Bob	16	NaN	NaN
2	3	Charlie	17	B	85.0

```
students_merged = pd.merge(students, performance, on='student_id', how='right')
```

↳

	student_id	name	age	grade	attendance
0	1	Alice	15.0	A	95

1	3	Charlie	17.0	B	85
2	4	NaN	NaN	A	100

```
students_merged = pd.merge(students, performance, on='student_id', how='inner')
```

	student_id	name	age	grade	attendance
0	1	Alice	15	A	95
1	3	Charlie	17.0	B	85

```
students_merged = pd.merge(students, performance, on='student_id', how='outer')
```

	student_id	name	age	grade	attendance
0	1	Alice	15.0	A	95.0
1	2	Bob	16.0	NaN	NaN
2	3	Charlie	17.0	B	85.0
3	4	NaN	NaN	A	100.0

Grouping data = organizing it by common values in one or more columns

groupby : groups data by specific values in a column

ex: `grouped = df.groupby('Representative')`

↓

The result of the operation -grouped- is a special object that contains our data in a proper grouped format.

↓

If we print this object, we will see something like

<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1169eb820>

because this obj doesn't have the __repr__ method.

Applying functions to Groups

* summing sales

```
total_sales = df.groupby('Representative')['Sales'].sum()
```

↳ per ogni representative calcola la somma dei suoi sales

* counting entries

```
count_sales = df.groupby('Representative')['sales'].count()
```

↳ how many sales entries exist for each representative

* average sales

```
average_sales = df.groupby('Representative')['sales'].mean()
```

groupby: splits data into groups based on some criteria

agg: applies one or more aggregation functions to those groups

Let's have a dataframe like this:

store	product	units_sold	price
...	...	...	...

1) Create the aggregation dictionary

```
agg_funcs = {'units_sold': 'max', 'price': 'mean'}
```

2) Group data by store and apply the aggr. functions

```
result = df.groupby('store').agg(agg_funcs)
```

↓

result:

store	units_sold	price
...	max/store	avg/store

Correlation: statistical measure that describes how much two variables change together

Two main types of correlation:

1) Positive correlation: when one variable increases, the other tends to increase
2) Negative correlation: the opposite.

↓

TO calculate correlations → corr() method

We have a Dataframe like this:

Price	Size	Bedrooms	Age
...	...	..	...

houses

correlation_matrix = houses.corr()

	Price	Size	Bedrooms	Age
Price	1	0.99	0.97	-0.87
Size	..	1	...	..
Bedrooms	..	..	1	..
Age	..	..	...	1

shows the correlation coefficients between each pair of variables

Interpretation of result

* (1) = perfect positive correlation
* (-1) = perfect negative correlation
* (0) = no correlation

If we are interested in the correlation between just two columns instead of the entire dataset, we can use corr() directly on those columns

```
correlation_price_size = houses['Price'].corr(houses['Size'])
```

↳ example result: 0.98...

⚠ Important: HANDLE MISSING DATA BEFORE FINDING CORRELATIONS.

Categorical data → can be divided into groups or categories

↓

In Pandas categorical data can make computations faster and save

memory.

To convert DataFrame columns to categorical types → method .astypes()

Example:

```
import pandas as pd          # Titanic dataset
import seaborn as sns

#Load titanic dataset
titanic = sns.load_dataset('titanic')
print(titanic.info())        # how many columns, which ones, details...

#convert 'sex' and 'class' columns to categorical types
titanic['sex'] = titanic['sex'].astype('category')
titanic['class'] = titanic['class'].astype('category')
print(titanic.info())
```

↓

after conversion 'sex' and 'class' changed from object to category

faster and more efficient

★ Sometimes we must convert categorical data to numeric codes

for machine learning models → LABEL ENCODING

Simplest encoding: replace categories with some numbers

?

For example for sex column → encoding 'male' with 0

and 'female' with 1

```
#Label encoding the 'sex' column
titanic['sex_code'] = titanic['sex'].cat.codes
```

→ curioso output di print(titanic.head())

cat.codes = an attribute of Pandas' Categorical type that returns the codes corresponding to the categories in the categorical data.

One-hot encoding → turns a column with categories into several new columns, one for each category → in each new column we put 1 if the row belongs to that category, otherwise 0.

※ one-hot encoding the 'class' column

titanic_class_dummies = pd.get_dummies(titanic['class'], prefix='class')

↓

creates a separate dataframe with encoded values, performing the one-hot encoding

---

Date info → often comes as text → not very useful for analysis

↓

we want to convert this information in datetime format

pd.to_datetime() → converts different date formats correctly

Example:

```
data = {
    'order_date': ['2023-10-01', '10-01-2023', 'October 3 2023',
                   '2023.10.04']
}
sales = pd.DataFrame(data)
※ convert 'order_date' to datetime
sales['order_date'] = pd.to_datetime(sales['order_date'], format='mixed')
```

= format will be inferred for each element individually

Extracting components from Datetime

```
※ Extract year, month and day from datetime
sales['year'] = sales['order_date'].dt.year
sales['Month'] = sales['order_date'].dt.month
```

```
sales['day'] = sales['order_date'].dt.day
```

Basic Datetime Operations

```
from datetime import datetime
#calculate time delta
sales['time_since_order'] = datetime.now() - sales['order_date']
#today's date
today = pd.to_datetime('today')
```

Sorting data

```
import seaborn as sns
#Load the titanic dataset
titanic = sns.load_dataset('titanic')
#sort by fare in descending order
titanic_sorted = titanic.sort_values(by='fare', ascending=False)
```

Ranking data

We have this DataFrame:

student	score
...	...

```
#Rank students by their score
students['score_rank'] = students['score'].rank(method='average',
                                                ascending=True)
```

lowest score -> rank = 1

if Bob and David share ranks 3 and 4

average rank = 3.5 for both

There are different methods of sorting ties in ranking:

- average
- min (smallest rank assigned to all ties)
- max

- first (ranks assigned in the order they appear)
- dense (like min, but the rank of the next group is just one more than the previous group)

PIVOT TABLES → powerful tools that allow us to summarize, analize and explore data in different ways

↓

They are commonly used in data analysis for generating insights from data by arranging, sorting and aggregating it.

Pandas library → provides function `pivot_table()`

Important parameters of the function:

1) index → the columns to group by (like aisles in a store)
2) columns → The column whose distinct values will form the columns of the pivot table
3) values → The columns containing the data we want to aggregate (like toy prices)
4) aggfunc → the function used to aggregate the data (e.g. mean)

Example

Let's suppose we have data about different products, and we want to see the average price of each product category

df →

Product	Category	Price
...	...	...

```
#create a pivot table
pivot_table = df.pivot_table(index='Product', values='Price', aggfunc='mean')
```

result:

Product	Price
Book	9.5

Electronic	100.0
Toy	12.5

Example 2

From the titanic dataset → we want to analyse the average fare and survival rate based on the class of the passenger and their gender.

```
import seaborn as sns
titanic = sns.load_dataset('titanic')
# create a simplified pivot table
pivot = titanic.pivot_table(index='class', columns='sex',
        values=['survived', 'fare'], aggfunc='mean')
print(pivot)
```

	fare		survived	
sex	female	male	female	male
class				
First	106.13	67.22	0.97	0.37
Seond	21.97	19.74	0.92	0.16
Third	16.12	12.66	0.50	0.14

Example 3

df : | Toy | Store | Sales |
|---|---|---|
| ... | ... | ... |

↳ we want to create the pivot table that shows the sum of toy sales by 'Toy' type and 'Store'

```
pivot_table = df.pivot_table(index='Toy', columns='Store',
```

values = 'Sales' , aggfunc = 'sum')

↓

Result →

Store	A	B
Toy		
Blocks	300.0	200.0
Car	100.0	50.0
Doll	150.0	NaN

COMPREHENSIVE DATA WRANGLING AND ANALYSIS WITH PANDAS AND NUMPY

Complex groupby operations

Recall of the basic groupby

df →

student	subject	score
...	...	...

```
grouped = df.groupby('student')['score'].mean()
```

↳

Student	
Alice	86.5
Bob	91.5
Charlie	96.0

Transition to complex groupby

*Detailed grouping with multiple aggregations

```
grouping_details = titanic.groupby(['class', 'embark_town'],
                                   observed = True).agg({
    'fare': ['mean', 'max', 'min'],
```

age : ['mean', 'std', 'count']

3)

print (grouped_details)

Setting observed = True ensures the result only includes the combinations observed in the data

→ per esempio abbiamo:

class : First, Second, Third

embark_town : Cherbourg, Queenstown, Southampton

↳ in teoria ci sono 3×3 = 9 combinazioni. Ma magari nel dataset nessun passeggero di First class è partito da Queenstown ↴

con observed = True una combinazione del genere (possibile, ma non presente nei dati) viene esclusa :)

titanic.groupby(['class', 'embark_town'])

we first group data by 'class' and 'embark_town'

→ this means we'll have a separate group for each combination of class and embarkation town

.agg({...}) → Apply multiple aggregations

↓

inside we specify the columns and the aggregation functions we want to apply

Result:

		fare			age		
		mean	max	min	mean	std	count
class	embark_town						
First	Cherbourg						
	Queenstown						
	Southampton						

Second	Cherbourg
	Queenstown
	Southampton
Third	Cherbourg
	Queenstown
	Southampton

Piccola nota

* groupby → più flessibile e permette operazioni complesse su più colonne
* pivot_table → pensata per creare tabelle riassuntive in stile Excel, con righe e colonne etichettate

---

STANDARD SCALING

↓

it's like leveling the playing field for our data.

It transforms our data so it has a mean of 0 and a standard deviation (how spread out numbers are) of 1.

This is especially useful when we want our data to follow a standard normal distribution.

The formula for standard scaling is:

$$z = \frac{(X - \mu)}{\sigma}$$

- X = original value
- μ = mean of the values
- σ = standard dev of the values.

Applying standard Scaling:

```
import pandas as pd
import seaborn as sns
```

```
titanic = sns.load_dataset('titanic')

# calculate mean and std dev for 'age' and 'fare'
age_mean = titanic['age'].mean()
age_std = titanic['age'].std()
fare_mean = titanic['fare'].mean()
fare_std = titanic['fare'].std()

# standard scaling
titanic['age_standard'] = (titanic['age'] - age_mean)/age_std
titanic['fare_standard'] = (titanic['fare'] - fare_mean)/fare_std
```

MIN-MAX SCALING

↳ it adjusts the scale of our data to fit within a specific range (typically between 0 and 1).

Formula: $$X' = \frac{(X - X_{min})}{(X_{max} - X_{min})}$$

```
# calculate min and max for 'age' and 'fare'
age_min = titanic['age'].min()
age_max = titanic['age'].max()
fare_min = titanic['fare'].min()
fare_max = titanic['fare'].max()

# MinMax scaling
titanic['age_minMax'] = (titanic['age'] - age_min)/
                        (age_max - age_min)
titanic['fare_minMax'] = (titanic['fare'] - fare_min)/
                         (fare_max - fare_min)
```

DATA PREPROCESSING

↳ prepare a dataset for analysis

1) Drop unnecessary columns

Example :

```
columns_to_drop = ['deck', 'embark_town', 'alive' ...]
titanic = titanic.drop(columns = columns_to_drop)
```

2) Handle missing values

Example :

```
* Fill missing values in 'age' with the median value
titanic['age'] = titanic['age'].fillna(titanic['age'].median())
* Fill missing values in 'embarked' with the mode value
titanic['embarked'] = titanic['embarked'].fillna(
                      titanic['embarked'].mode()[0])
* Fill missing values in 'fare' with the median value
titanic['fare'] = titanic['fare'].fillna(titanic['fare'].
                  median())
```

```
* check for any remaining missing values
print(titanic.isnull().sum())
```

↓

returns a dataframe of the same size containing True instead of the missing values and False instead of the present values.

output :

```
1   survived    0
2   pclass      0
3   sex         0
```

4 age 0

...

3) Encode categorical values

```
#Encode categorical values
titanic = pd.get_dummies(titanic, columns = ['sex',
                                             'embarked'], dtype = 'int')
```

4) Scale numerical values

```
#scale numerical values
titanic['age'] = (titanic['age'] - titanic['age'].mean(1)/ titanic['age'].std(1
```

...

COMPREHENSIVE ANALYSIS WITH MULTIPLE TECHNIQUES

goal: learn how to combine groupby, merge and pivot tables :)

Comprehensive example on titanic dataset

1) Combining groupby and aggregation

```
import seaborn as sns
import pandas as pd
import numpy as np

titanic = sns.load_dataset('titanic')

# Group by 'class' and 'sex' with observed = True to specify
# the exact behaviour expected
class_sex_grouping = titanic.groupby(['class', 'sex'], observed=True).agg({
    'survived': 'mean'
```

'fare' : 'mean',

'age' : ['mean', 'std']

3). reset_index()

Using reset_index here is necessary to convert the multi-level index (created by the groupby operation) back into regular columns of the DataFrame.

→ Without resetting the index, the result DataFrame would have 'class' and 'sex' as index levels, which can complicate further data manipulation and readability

2) Combining groupby and aggregation pt2

After grouping we'll simplify the multi level columns for readability

※ Simplify multi-level columns

```
class_sex_grouping.columns = ['class', 'sex', 'survived_man', 'fare_mean',
                              'age_mean', 'age_std']
```

↓

before this the dataframe is like this :

class	sex	survived	fare	age	
		mean	mean	mean	std
...	...	...	...	...	...

↳ after that instruction :

class	sex	survived_mean	fare_mean	age_mean	age_std
...	...	...	...	...	...

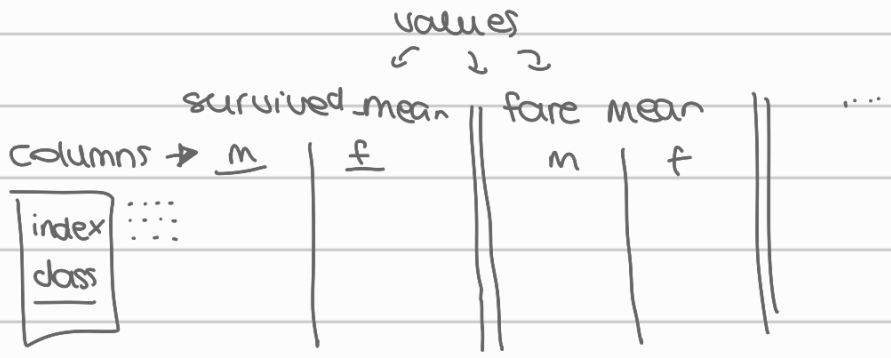
3) Creating a Pivot Table

※ Pivot table with observed = True for grouping to avoid FutureWarnings

```
pivot_table = class_sex_grouping.pivot_table(
```

```
    index = 'class',
    columns = 'sex',
    values = ['survived_mean', 'fare_mean', 'age_mean', 'age_std'],
    observed = True
)
```

PIVOT TABLE



4) Adding a conditional column

Well' add a new column to indicate whether a passenger is a child

```
** Adding a child column : whether the passenger is a child (age < 18)
titanic['is_child'] = titanic['age'] < 18
```

5) Analysis of survival rates by class and age group

Let's analyse survival rates by class and whether the passenger is a child

```
** Analyze survival rates by class and whether the passenger is a
**child or not
survival_by_class_child = titanic.pivot_table(
    'survived', index='class', columns='is_child', agg_func='mean',
    observed = True
)
```

is_child	False	True
class		
First	...	...
Second	...	...
Third	...	...

6) Merging Datasets for Comprehensive View

```
# Merge the pivot table with the original grouped data
# for comprehensive view
comprehensive_view = pd.merge(
        class_sex_grouping,
        survival_by_class_child,
        on = 'class',
        how = 'left'
)
```

result:

Survival by class

class	sex	survived_mean	....	age_std	False	True

```
# Rename columns for clarity
comprehensive_view.rename(columns = {
                False: 'adult_survival_rate',
                True: 'child_survival_rate'},
                inplace = True)
```