# Module 06 - Working with *real* datasets

# Real world data

So far, we have use *dummy* data. But, now we would like to explore data from the *real world* and write *useful* programs!

For that, we need to be able to *read files* containing data.

Data might be written, for example, in the commonly used *Comma-Separated Values*(CSV) format or some other formats.

# Why using CSV files is useful?

Because all of your programs have *access* to the *outside world through* CSV files now!

Almost *every software in the world* (Excel or any spreadsheet software, banks, CRMs, web apps, SQL databases, Google Sheets) *can export data as a CSV*!

Once you have a dataset in a CSV format file, you can write small Python programs (scripts) to:

- *Read* and *clean* data
- *Analyze* data with Numpy or any useful Python library
- *Output* the *results* into a brand new CSV summary file or save generated charts (with Matplotlib) directly as image files
- etc.

# Advantage of using a Python program instead of importing data into Excel (or any spreadsheet software) 1/2

- Python's CSV `reader` read the file **line by line**! Which means, **you can work with (very) large files and datasets** (several GB of data!). If you try to load these huge datasets into Excel it will break! (because Excel, or any spreadhseet, will load *everything* in memory at once!)
- **No (Row) Limits**! Excel has a strict hard limit of $1,048,576$ rows and $16,384$ columns. If your dataset surpasses this limit, Excel will literally cut off and delete the extra data without warning! Python has no such artificial limit, **a Python program can process tens or hundreds of millions of rows!**

# Advantage of using a Python program instead of importing data into Excel (or any spreadsheet software) 2/2

- *Repeatability*! In Excel, every time you get a new weekly or monthly CSV file, you have to manually open it, click around, apply filters, and *copy-paste* formulas all over again. With Python, you *write the script once* and can rerun it instantly with a single command whenever new data arrives.
- *Audit*: your programs act as *clear*, *step-by-step recipes* of how your data was *cleaned* and *analyzed*. In Excel, it is often difficult to remember which manual filter or hidden formula was applied to which cell, making it harder to spot mistakes or explain your work.

# Be careful, real-world data is *never* clean

For example, look at this `sales_messy.csv` file provided by the sales team:

```
salesperson,sales_euros
  John ,100
john,5000
JANE,9000
jane,1200
eve,N/A
EVE ,3000
,500
```

**Inspect** the data. What's *wrong* here?

> What's wrong with them?!

# Classic problems in messy (real world) data

What's *wrong* here?

- **Inconsistent** casing: "John" and "john" or "JANE" and "jane" should count as the same person.
- Extra **whitespaces**: Names have accidental spaces (like " John " or "EVE ").
- **Invalid** or **missing** data: One row has "N/A" instead of a number, and last row is missing a name entirely!

# You need to clean datasets

*Proper data cleaning* is often said to take up 80% of a data scientist's or analyst's time!

> It's a big part of the job

Following a *systematic* approach ensures your results are accurate and *reliable*.

**Here are some recommandations**

# How to clean datasets 1/2

- Inspect data *first*: have a look at the raw data *before* writing anything!
- Check the *schema*: know what *type* of data each column *should* be (e.g. text, integer, float, date).
- *Standardize*:
  - Strip whitespaces: Accidental leading or trailing spaces (" John " vs "John") will break aggregations and lookups
  - Fix casing to all lowercase or uppercase ("John", "john", and "JOHN" must be treated as the same entity)

# How to clean datasets 2/2

- Handle *missing* or *corrupt* data:
  - Define *how* to treat blanks: decide whether missing values should be *ignored* or imputed (replaced with an average, median, or placeholder like 0 or "Unknown").
  - Catch *invalid placeholders*: real-world data often uses text strings like "N/A", "NULL", "?", or "-" where a number is expected. Your program needs to *handle* or *skip* these safely instead of crashing.
- Remove *duplicates*: rows that are exact copies can *skew* counts and sums! Drop them.
- *Validate* data types and validity ranges:
  - Ensure numbers are *actually* stored as numbers, dates have a correct format (e.g `YYYY-mm-dd`), etc.
  - Look for *logical impossibilities* or *abnormal data*. For example: a negative age

# Cleaning data with a Python program

Never clean data manually by *deleting* rows in the source CSV file directly!

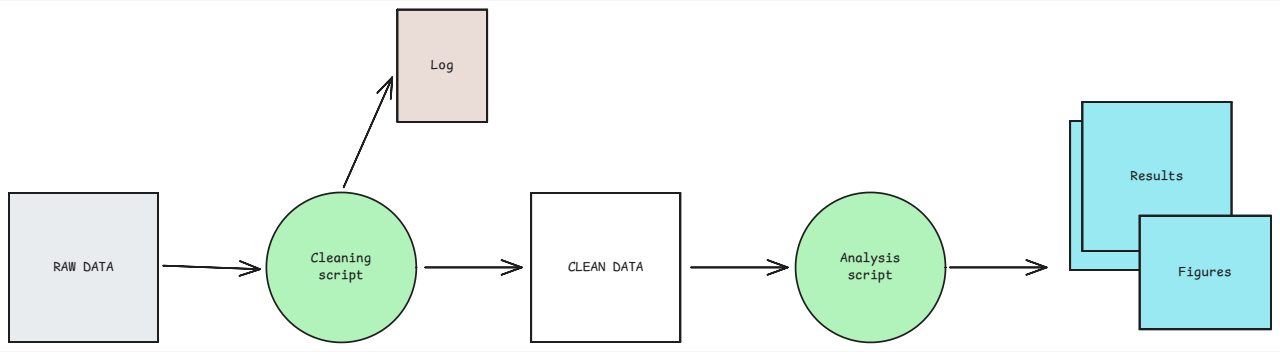
Instead, **write a dedicated cleaning Python script**. *Once cleaned*, always export the result into **a new CSV file**, leaving your **source data untouched**!

This ensures your data transformations are fully **repeatable**, **transparent** and **traceable**.

# Your data pipeline

Keep data cleaning *separate from* data analysis!

> If you drop or modify rows (e.g., dropping rows with missing names), it's good practice to *log* somewhere how many rows were dropped so you know how much data you lost.



# Introducing the Pandas library

Pandas is a fast, powerful, and flexible open-source data analysis and manipulation library for Python, *built on top* of NumPy.

It allows to work with *labeled data* easily and intuitively. It is like *bringing (Excel) spreadsheets* directly *into* Python.

It is noticably really **useful to work with CSV files** (loading and writing tabular data). Indeed, pandas **supports the integration with many file formats or data sources out of the box** (csv, excel, sql, json, etc.).

# Pandas key datastructures

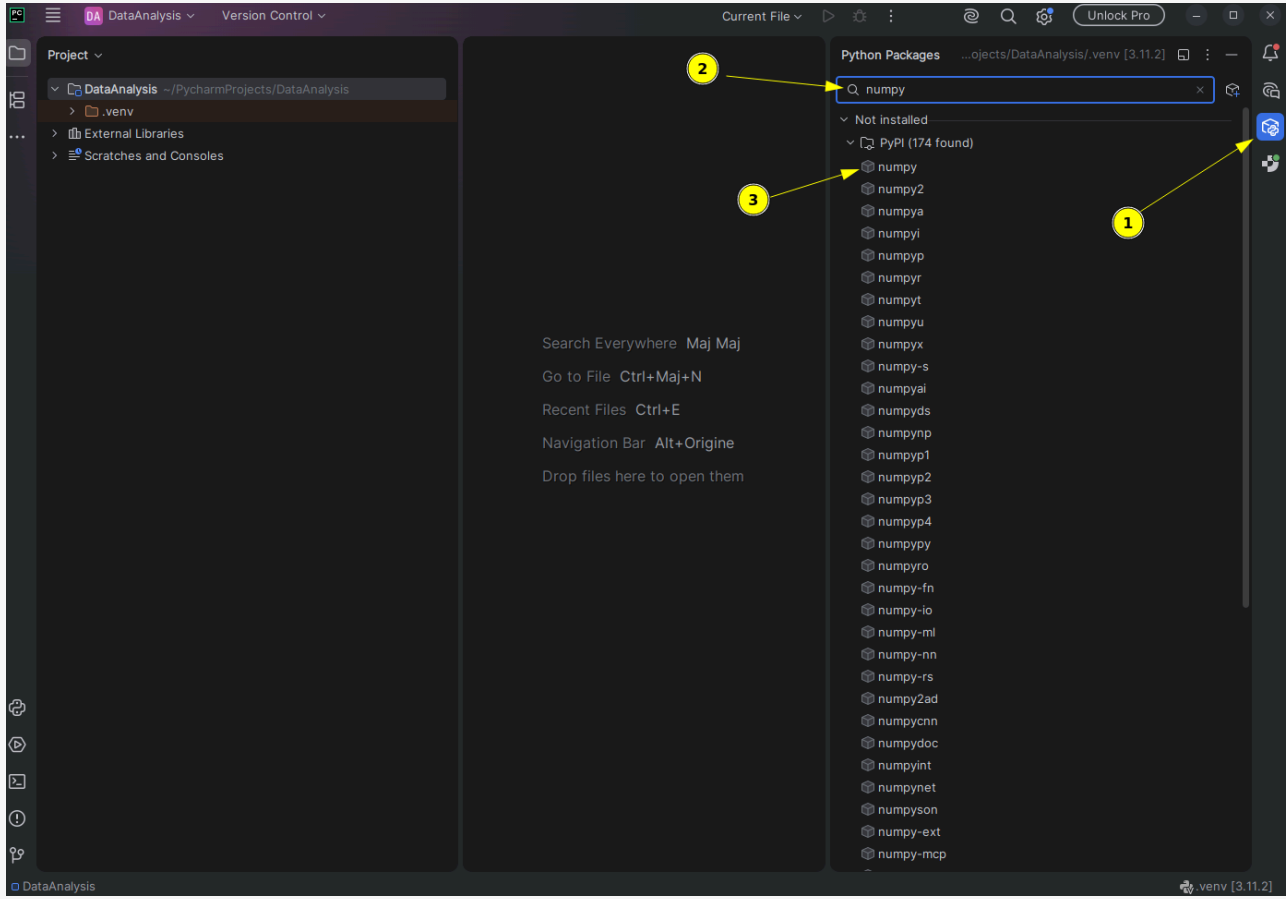
Pandas brings **two new datastructures** to the table:

- DataFrame: A two-dimensional labeled data structure with rows and columns (a complete table).
- Series: A one-dimensional labeled array (essentially a single column).

> Each of these datastructures offer useful functionalities to work with tabular labelled data.

# Install pandas package

1. **Open** the *Python Packages Menu*. You should see only `pip` (the package manager) for now
2. **Search** and **install** the following packages :
   `pandas`



# Use pandas in your program

To load the pandas package and start working with it, import the package. The community agreed alias for `pandas` is `pd`, so loading `pandas as pd` is assumed *standard practice*:

```
import pandas as pd
# your code...
```

# Prepare a simple CSV file with headers

In your current PyCharm project, **create** a text file `tickets_sold.csv` containing the data from the local cinema problem (see Problem 1: Arrays and sales):

```
Adult,Children,Senior
120, 45, 30
95,  30, 25
110, 50, 40
150, 60, 35
```

# Read the CSV file

pandas provides the `read_csv()` function to read data stored as a csv file into a pandas `DataFrame`:

```
import pandas as np
dataset = pd.read_csv("tickets_sold.csv")
print(dataset)
```

# Export the data as a numpy array

You can convert pandas DataFrame to numpy array easily with the `to_numpy()` function:

```
import pandas as np
dataset = pd.read_csv("tickets_sold.csv")
sales = dataset.to_numpy()
print(sales, type(sales))
```

> `sales` is now a numpy array! You can use everything you know about numpy!

# Learning pandas working with a real dataset

For code examples we will use the real-world dataset of Titanic passenger data, provided by pandas and used in their tutorials.

```
import pandas as pd
# Load a CSV file from your local directory into a DataFrame
df = pd.read_csv('titanic.csv')
```

# Inspect your data

Once loaded, it's best practice to **check** *first* what the data *looks like*:

- `df.head()`: Displays the first 5 rows (you can pass a number like `df.head(10)`).
- `df.info()`: Shows column names, non-null counts, and data types.
- `df.describe()`: Provides basic statistical summaries (mean, min, max, etc.) for numerical columns.

```
print(df.head())
print(df.info())
print(df.describe())
```

> You should *always* inspect your data before going further!

# Useful Parameters in `read_csv`

As we already said earlier, real-world data *is messy*! `read_csv()` has *many* built-in parameters to handle formatting issues:

- `sep` : Define a custom separator if it's not a comma (e.g., sep=';' for semicolon-separated files).
- `header` : Specify which row number to use as column names (e.g., header=0 or header=None).
- `skiprows` : Skip header rows
- `na_values` : Define custom strings that should be treated as missing/NaN values (e.g., `na_values=['N/A', 'Unknown']` ).
- `usecols` : Load only specific columns to save memory (e.g., `usecols=['Name', 'Age']` ).
- `index_col` : Which column from your CSV file should be used as the row labels (the index) of the `DataFrame` .

> With pandas, you don't have to handle manually corrupted or missing data like we have to do before!

# Exemple

```
titanic = pd.read_csv('titanic.csv', sep=',', usecols=['Name','Age', 'Pclass'], na_values=['N/A'])
```

> We load only 3 columns 'Name','Age' and 'Pclass'

# Selecting columns

To select a *single* column from a `DataFrame`, use bracket notation with the name of the column inside quotes. This extracts the data as a pandas `Series` (a single 1D column with a label index):

```
ages = titanic["Age"]
print(type(ages))  # Output: <class 'pandas.core.series.Series'>
print(ages.head())
```

# Selecting multiple columns

If you need to select *multiple columns at the same time*, **pass a list of column names inside the brackets**. Note the double brackets (`[[]]`):

- the outer brackets specify that you are indexing the `DataFrame`
- the inner brackets contain the list of column names. This returns a new, smaller `DataFrame`

```
subset = titanic[["Name", "Age", "Sex"]]
print(type(subset))  # Output: <class 'pandas.core.frame.DataFrame'>
print(subset.head())
```

# Filtering rows

To filter rows *based on specific conditions*, you use *boolean indexing*. You pass a conditional expression (predicate) inside the `DataFrame` brackets, and pandas *evaluates that condition for every single row*, returning only the rows where the condition evaluates to `True`.

For example, to find all passengers older than 35:

```
above_35 = titanic[titanic["Age"] > 35]
print(above_35.head())
```

# Filtering rows with multiple conditions

You can combine multiple conditions using bitwise operators (`&` for "AND", `|` for "OR"). Each condition must be wrapped in parentheses:

```
# Passengers in 1st class who are older than 50
seniors_class1 = titanic[(titanic["Age"] > 50) & (titanic["Pclass"] == 1)]
```

# Filtering with `.isin()`

If you want to check if values belong to a *specific list of options*, use the `.isin()` method:

```
# Passengers who embarked in Cherbourg ('C') or Queenstown ('Q')
port_filter = titanic[titanic["Embarked"].isin(["C", "Q"])]
```

# Selecting rows and columns with `loc` and `iloc`

Besides bracket notation, pandas provides two *indexers* to select rows **and** columns at the same time:

- `.loc[]`: selection by **label** (index values and column names)
- `.iloc[]`: selection by **integer position** (0-based, like Python lists)

Both use the syntax `df.loc[rows, columns]`.

# Setting up an index

To use labels, the `DataFrame` needs a *meaningful index*. We can choose one at loading time with `index_col` parameter:

```
titanic = pd.read_csv('titanic.csv', index_col='PassengerId')
print(titanic.index)  # 1, 2, 3, ... 891
```

> Without `index_col`, pandas creates a default index `0, 1, 2, ...`. In that case, labels and positions are identical, which is a frequent source of confusion.

# `loc` : selection by label

```
titanic.loc[1]                          # one row (returns a Series)
titanic.loc[[1, 5, 10]]                 # several rows (returns a DataFrame)
titanic.loc[1:5]                        # rows 1 to 5 (end INCLUDED)

titanic.loc[1, "Name"]                  # one value
titanic.loc[1:5, ["Name", "Age"]]       # rows and columns
titanic.loc[:, "Name":"Sex"]            # all rows, columns from Name to Sex
```

# `iloc`: selection by position

```
titanic.iloc[0]                          # first row
titanic.iloc[[0, 4, 9]]                  # rows at positions 0, 4 and 9
titanic.iloc[0:5]                        # positions 0 to 4 (end EXCLUDED)

titanic.iloc[0, 3]                       # first row, 4th column
titanic.iloc[0:5, [2, 3]]                # rows 0 to 4, columns at positions 2 and 3
titanic.iloc[-1]                         # last row
```

> Remember: `iloc` means *i*ndex *loc*ation

# Remark on slicing: `loc` vs `iloc`

The two indexers do not treat the end of a slice the same way:

```
titanic.loc[1:3]    # PassengerId 1, 2, 3  --> 3 rows (end included)
titanic.iloc[1:3]   # positions 1, 2       --> 2 rows (end excluded, as we have seen with range)
```

# `loc` with conditions

`loc` accepts the boolean conditions seen previously, and lets you choose the **columns** to keep:

```
# Select only passenger names and ages of passengers older than 35
titanic.loc[titanic["Age"] > 35, ["Name", "Age"]]

# Select only passenger names of female passengers in 1st class
titanic.loc[(titanic["Pclass"] == 1) & (titanic["Sex"] == "female"), "Name"]
```

# Summary: `loc` vs `iloc`

| | `loc` | `iloc` |
|---|---|---|
| Selection by | labels | integer positions |
| Slice end | **included** | **excluded** |
| Accepts a boolean condition | yes | only with a list/array (not a `Series`) |

# When to use which?

- Use `loc` when you know **names** or when filtering with **conditions**
- Use `iloc` when you need **positions** (first rows, last row, "the 3rd column")

> Prefer labels (`loc`) whenever possible because the code will stay correct if the column order changes

# Statistics with pandas

Pandas provides many built-in statistical methods to quickly analyze your data *without writing manual loops*!

These methods are applied directly to a `DataFrame` column (aka a `Series`) and automatically ignore missing (NaN) values:

- .mean(): Computes the average
- .median(): Computes the middle value
- .sum(): Adds up all values (great for binary flags or counts)
- .min() / .max(): Finds the minimum and maximum values
- .count(): Counts non-null entries

# Examples

```
import pandas as pd

titanic = pd.read_csv('titanic.csv')

# Average age of all passengers
print("Mean Age:", titanic["Age"].mean())

# Median age
print("Median Age:", titanic["Age"].median())

# Total number of survivors (since Survived is 1 or 0)
print("Total Survivors:", titanic["Survived"].sum())

# Minimum and maximum fare paid
print("Lowest Fare:", titanic["Fare"].min())
print("Highest Fare:", titanic["Fare"].max())
```

# Sorting data with sort_values()

To order your rows based on the values of one or more columns, use .sort_values():

- `by` : Specifies the column name (or a list of column names) to sort by.
- `ascending` : Set to True (default) for lowest-to-highest, or False for highest-to-lowest.

```
# Sort passengers by age in ascending order (youngest first)
youngest_first = titanic.sort_values(by="Age")
print(youngest_first[["Name", "Age"]].head())

# Sort passengers by fare in descending order (highest price first)
expensive_first = titanic.sort_values(by="Fare", ascending=False)
print(expensive_first[["Name", "Fare"]].head())
```

# Combining methods call (chaining) to express complex operations

Because Pandas methods return a Series or a DataFrame, you can *chain* multiple operations together in a single readable line:

```
# Find the name of the Oldest survivor
oldest_survivor = (
    titanic[titanic["Survived"] == 1]
    .sort_values(by="Age", ascending=False)
    .iloc[0][["Name", "Age"]]
)
```

print("Youngest survivor:", youngest_survivor)

# Practice

**Write** Python instructions (directly in the console) with pandas to find the:

1. list of survivors
2. list of survivor's names only
3. number of survivors
4. number of survivors who were female
5. average age of survivors
6. median age of survivors
7. name and age of the youngest survivor
8. name, age and class of the 5 youngest survivor

We suppose that passenger's data are stored in the DataFrame `titanic`.

# Correction

```
#1. List of survivors
titanic[titanic["Survived"] == 1]

#2. Names of survivors
titanic.loc[titanic["Survived"] == 1, "Name"]

#3. Number of survivors
titanic["Survived"].sum()

#4. Number of survivors who were female
((titanic["Survived"] == 1) & (titanic["Sex"] == "female")).sum()

#5. Average survivor age
titanic.loc[titanic["Survived"] == 1, "Age"].mean()

#6. Median survivor age
titanic.loc[titanic["Survived"] == 1, "Age"].median()

#7. Name and age of the youngest survivor
titanic[titanic["Survived"] == 1].sort_values("Age").iloc[0][["Name", "Age"]]

#8. Name and age of the youngest survivor
titanic[titanic["Survived"] == 1].sort_values(by="Age")[["Name", "Age", "Pclass"]].head(5)
```

# Counting occurrences with `value_counts()`

To count how many times *each unique value appears in a column*, use `.value_counts()`:

```
# Count passengers by class
passenger_counts = titanic["Pclass"].value_counts()
print(passenger_counts)
```

To get **percentages** instead of *raw* counts, pass `normalize=True` to the function (it rescales counts between 0 and 1) and multiply by 100:

```
# Get the percentage split of passengers by sex
passenger_pct = titanic["Sex"].value_counts(normalize=True) * 100
print(passenger_pct)
```

# Aggregating data with `groupby()`

To summarize data by categories, use `.groupby()`. It follows the **Split-Apply-Combine** pattern:

1. **Split** the data into *groups* based on category values.
2. **Apply** a function (like `.mean()`, `.sum()`, or `.count()`) to each group.
3. **Combine** the results into a clean summary table.

# Aggregating examples

```
# Calculate the average age for each passenger class
avg_age_by_class = titanic.groupby("Pclass")["Age"].mean()
print(avg_age_by_class)

# Calculate average survival rate (%) by class
survival_by_class = titanic.groupby("Pclass")["Survived"].mean() * 100
print(survival_by_class)
```

> **Why `.mean()` for survival?** Since `Survived` contains binary values (`1` for survived, `0` for died), taking the mean calculates the exact proportion of survivors!

# Visualizing data with Pandas and Matplotlib

Pandas comes with *built-in* plotting capabilities powered by **Matplotlib**, making it *effortless* to turn your `DataFrames` and `Series` into *plots* with a single method call: `.plot()`

```
import pandas as pd
import matplotlib.pyplot as plt # import!

# Plot a simple bar chart from a value count
titanic["Sex"].value_counts().plot(kind="bar", color=["#3498db", "#e74c3c"])

plt.title("Titanic Passengers by Sex")
plt.xlabel("Sex")
plt.ylabel("Count")
plt.show()
```

Everything you know about Matplotlib can be reused!

> To render plots with pandas, always import `matplotlib.pyplot`!

# Common plot types

You can change the type of visualization by modifying the `kind` parameter:

- `kind="bar"`: Vertical bar chart (ideal for comparing categories)
- `kind="barh"`: Horizontal bar chart
- `kind="hist"`: Histogram (ideal for showing distributions of continuous variables like Age)

```
# Plot an age distribution histogram with 20 bins
titanic["Age"].plot(kind="hist", bins=20, color="#2ecc71", edgecolor="white")

plt.title("Age Distribution of Titanic Passengers")
plt.xlabel("Age")
plt.ylabel("Frequency")
plt.show()
```

> Always use `plt.show()` at the end of your plotting block to display the canvas window!

# First steps with pandas

Follow the *How do I read and write tabular data?* tutorial

> You can also check 10 minutes to pandas, a short introduction to pandas, geared for new users

# Practice

- DataCamp: Introduction to Data Science in Python. Do *at least* Chapter 1 and Chapter 2
- DataCamp: Course - Data Manipulation with pandas
- Problem 5

# Extra Information/Going further

# Handling CSV files without pandas

To handle CSV files, we can use the built-in `csv` module.

**Create** the file `sample.csv` with the following content:

```
salesperson,sales_euros
john,100
john,5000
jane,9000
jane,1200
eve,100
eve,3000
```

# Read the CSV file

```
import csv

with open('sample.csv', newline='') as csvfile:
    #'reader' is an iterable over the file "rows"
    reader = csv.reader(csvfile, delimiter=',', quotechar='|')
    # iterate over the reader to access each row, one by one
    # Skip the header row
    header = next(reader)
    for row in reader:
        #Each row is a list of strings (one column = one string)
        print(row[0], row[1])
```

> In Python, `with open('sample.csv', newline='') as csvfile:` is a *context* manager. It handles the setup and teardown of resources, in this case, opening and closing a file.

# Reader's parameters

```
reader = csv.reader(csvfile, delimiter=',', quotechar='|')
```

- `csvfile` : name of the file open with the `with` instruction
- `delimiter` : character to delimit columns. Comma by default ( `,` )
- `quotechar` : defines the character used to wrap fields that contain special characters (such as commas or newlines).

# Write a CSV file

```
import csv
# open a new file with 'w' (write permission)
with open('total.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    writer.writerow(["Some data"] * 5 + ["Other data"])
    writer.writerow(["Some data"] * 2 + ["Extra data"] * 3)
```

> Check the `total.csv` file in your project

# Practice: Cleaning datasets

1. **Write a Python program** `clean_dataset.py` (using the built-in csv module) to clean the dataset stored in `sales_messy.csv` and output the clean dataset into `clean_sales.csv`. Drop lines with incomplete data and standardize text in lowercase.
2. **Write a Python program** `analyze_dataset.py` that reads the clean CSV file `clean_sales.csv` and aggregates the total sales by person (like before).

# Practice: Cleaning datasets, suggestion 1/2

```
# clean_dataset.py
import csv

cleaned_rows = []

with open('sales_messy.csv', mode='r', encoding='utf-8') as infile:
  reader = csv.reader(infile)
  header = next(reader) # Skip header
  for row in reader:
    name = row[0]
    sales_str = row[1]

    # Skip rows with missing or empty names
    if not name or not name.strip():
      continue

    # Clean the name: remove extra spaces and lowercase it
    person = name.strip().lower()

    # Skip rows with invalid sales numbers (like 'N/A')
    if not sales_str.strip().isdigit():
      continue

    cleaned_rows.append([person, sales_str])

# 2. Write clean rows into a new CSV file sales_clean.csv
with open('sales_clean.csv', mode='w', newline='', encoding='utf-8') as outfile:
  writer = csv.writer(outfile)
  writer.writerow(header)
  writer.writerows(cleaned_rows)

print('Cleaning complete! Saved to sales_clean.csv')
```

# Practice: Cleaning datasets, suggestion 2/2

```
# analyze_dataset.py

import csv

sales_by_person = {}

# Read the clean file generated by Program 1
with open('sales_clean.csv', mode='r', encoding='utf-8') as file:
  reader = csv.reader(file)

  header = next(reader)

  for row in reader:
    person = row[0]
    sales = int(row[1])

    if person in sales_by_person:
      sales_by_person[person] += sales
    else:
      sales_by_person[person] = sales

print('Total Sales by person:', sales_by_person)
```

# Why pandas is useful

Working *manually* with real-world data requires *a lot of work* and *precautions*!

Maybe we can use some Python library to help us?

> (Pandas enters the room)