Data Acquisition, Processing and Mining for AI

Week 01 — Introduction & Python for Data

Silviu Maniu

Welcome

Who / what / when

- **Course:** Data Acquisition, Processing and Mining for AI
- **Instructor:** Silviu Maniu first.last@univ-grenoble-alpes.fr
- **Format:** 8 weeks $\times$ (1.5 h lecture + 1.5 h lab)
- **Materials:** slides, lab notebooks and project documents on IM2AG Moodle
  `https://im2ag-moodle.univ-grenoble-alpes.fr/course/view.php?id=721`
- **Prerequisites:** basic Python and basic terminal

A title ending in (*) marks optional material: context or an extension, dropped first if we run short.

Grading

Project (50 %)

- in pairs, incremental over the 8 weeks
- a full data pipeline on a real data source
- notebook + short report + data card + 5-minute defense

Read the handout!

Written exam (50 %)

- covers lecture material
- focused on understanding
- reading/writing SQL, finding leakage, running algorithms by hand, …

Project milestones

	Due	You submit on Moodle	Grading
M0	before wk-3 lab	registration form: source + scope + accounts	pass/fail
M1	before wk-4 lab	acquisition log + the script + Parquet shapes	pass/fail
M2	before wk-6 lab	data card + EDA notebook	pass/fail
M3	wk 8 + 1	final pipeline + report + data card	graded

- Milestone submissions: due 20h00 the day before. Missing: $-1/20$ on the grade
- Will be discussed with the pairs in the lab, if needed
- M3 = 5-minute defense, in the Week 8 lab: a week before the final submission.
- every lab ends with "*now do this on your data*" (to help with your project)

Why a course about *data for AI*?

- The model is a small part of a real AI system → most of the surrounding machinery is data collection, verification, preparation[1]
- Folklore says practitioners spend "80 % of their time on data"; the measured figure is more like ∼38 % on data preparation and cleaning[2] → still the single largest slice
- Most methods assume the data *after* the `DataFrame` is clean… this course is about everything before that
- The field has a name: **data-centric AI** → hold the model fixed and improve the data

[1] Sculley et al., *Hidden Technical Debt in Machine Learning Systems*, NeurIPS 2015.

[2] Anaconda, *State of Data Science* 2022: 37.75 % of working time, preparation and cleansing combined.

The pipeline

acquire → understand → clean → mine

Wk	Topic
1	tooling / scripting (Python, NumPy, Pandas)
2	relational data & advanced SQL (DuckDB)
3	Web APIs: REST, authentication, pagination
4	web scraping, ethics & legality, data formats
5	EDA & visualization for data quality
6	cleaning, transformation, data leakage
7	clustering (k-means)
8	frequent itemsets & association rules (Apriori)

Data and its formats

What does data look like?

- This course's focus: **tabular data** → rows = observations, columns = variables
- Running example: NYC yellow-taxi trips → one row per trip: pickup/dropoff time and place, distance, fare, tip, …
- But data *arrives* in many shapes: text files, nested structures, API responses, HTML pages → getting it into a table

Why and where do you meet tabular data

- Machine learning: `fit(X, y)` takes a matrix of n rows (samples) $\times$ p columns (features) $\rightarrow$ a table
- Databases: every relational table, every SQL result set
- Business & science: spreadsheet exports, sensor and server logs, transaction records, survey responses, clinical data
- The Web: API responses and HTML tables (after flattening)
- Even text and images can become tables: one row per document, columns = embedding (vector) dimensions

Note that on tabular data, deep learning still cannot truly beat tree ensembles[3]

[3] Grinsztajn, Oyallon, Varoquaux, *Why do tree-based models still outperform deep learning on typical tabular data?*, NeurIPS 2022.

Example: NYC yellow-taxi trips

pickup	dropoff	pass.	dist.	PULoc	fare	total
2025-01-03 10:10:08	2025-01-03 10:31:04	1	2.32	141	18.40	26.88
2025-01-30 09:06:13	2025-01-30 09:28:05	1	2.18	236	19.80	29.75
2025-01-26 08:45:52	2025-01-26 09:20:36	1	18.00	211	70.00	85.95
2025-01-28 17:36:51	2025-01-28 17:52:33	1	1.30	90	12.80	20.05

- January 2025 alone: 3.5 M trips $\times$ 20 columns (we use a 200 000-row sample)
- One row = one trip; columns are heterogeneous: timestamps, counts, distances, money, codes
- Example: `PULocationID` is a code, not a quantity – 141 is a pickup zone $\rightarrow$ its mean means nothing
- Public, messy, and large enough to hide “bad things”

CSV: a classic tabular format

```
tpep_pickup_datetime,passenger_count,trip_distance,fare_amount
2025-01-14 08:12:03,1,2.4,14.2
2025-01-14 08:15:47,2,0.9,7.9
```

- **CSV** = **c**omma-**s**eparated **v**alues: plain text, one row per line, first line often the header
- Deceptively simple but tricky to decode: what if a value contains a comma? a newline? Is `;` the separator (like in French Excel)? What *encoding*?
- No types: `"2.4"` is text until *something* decides it is a number

XML: verbose, but checkable and readable

```xml
<trip pickup="2025-01-14T08:12:03">
  <passengers>1</passengers>
  <fare currency="USD">14.2</fare>
</trip>
```

- **XML**: a tree of elements carrying attributes (somewhat displaced by JSON, but still widely used)
- Real advantage: an XSD schema declares types and required fields, and a parser *rejects* documents that violate it (JSON does not have this)
- Far from dead: RSS feeds, sitemaps, Office/OpenDocument files, government open data, older APIs, GPS traces
- Navigate it with **XPath** (`//trip/fare`)

JSON: nested and self-describing

```
{"trip": {"pickup": "2025-01-14T08:12:03",
          "passengers": 1,
          "fare": {"amount": 14.2, "currency": "USD"}}}
```

- **JSON**: objects ({ }), arrays ([ ]), strings, numbers, booleans, `null`, and arbitrary nesting
- The native format of Web APIs → you will see a lot of it (and not only in this course)
- Tables are *flat* but JSON is a tree ⇒ annoying to flatten nested data into rows and columns

Parquet: typed, columnar, compressed

row-oriented (CSV)

one row's fields together → reading one column touches every byte

columnar (Parquet)

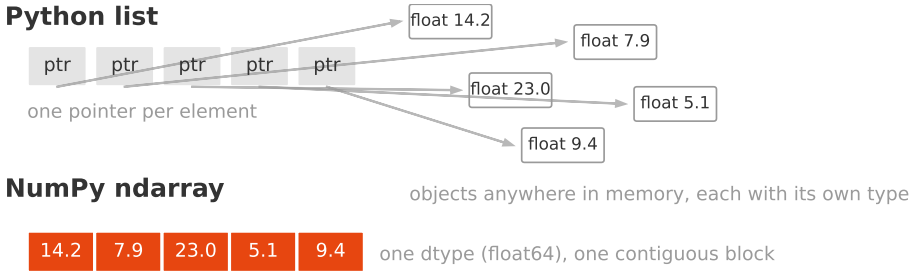
- Binary, with the schema stored inside the file: types survive the round-trip — exactly what CSV loses
- **Columnar** → read only the columns you asked for, also compresses much better
- The lab sample: 5.4 MB as Parquet vs 21.7 MB as CSV (for 200 000 rows)

NumPy: arrays and vectorization

What NumPy is for

- **Python**: the de-facto default language for data work, but can be slow for large datasets
- Has no real array: a list is a vector of pointers to boxed objects of any type, scattered over the heap
- **NumPy** (numerical library for Python) adds the missing piece, the `ndarray`: one dtype, one contiguous block, operations run in compiled C/Fortran (BLAS)
- The foundation of the scientific Python stack — pandas, scikit-learn, SciPy and matplotlib all use `ndarray`
- Why you want it: array expressions instead of loops — shorter code, and one to two orders of magnitude faster

Why not just (Python) lists?



- Summing a list: each `+=` follows a pointer, checks a type, unboxes a float — 200 000 times
- Summing an `ndarray`: one compiled loop over one block of bytes, all known to be `float64`
- Rule of thumb for data work: if you wrote a `for` loop over data, look for the vectorized form

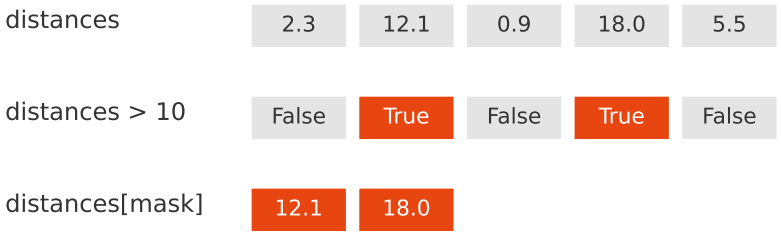
The ndarray

```
import numpy as np
a = np.array([14.2, 7.9, 23.0])
a.dtype         # dtype('float64')  -- ONE type for the whole array
a.shape         # (3,)
a * 1.09        # array([15.478,  8.611, 25.07 ])  -- no loop!
```

- **dtype**: every element has the same type; increases speed
- Arithmetic is **vectorized**: `a * 1.09` multiplies all elements in compiled code
- Works element-wise between arrays too: `fares + tips`

Selecting with boolean masks

```
mask = distances > 10      # array([False, True, False, True, False])
distances[mask]              # array([12.1, 18. ])
```



- A comparison yields a **boolean mask**; indexing with it filters
- Combine masks with `&` and `|` — never `and`/`or` — and parenthesize each one
- The same idiom returns in Pandas, and as SQL's `WHERE`

How much faster? (order-of-magnitude thinking)

- Summing $n = 2 \times 10^5$ floats: pure-Python loop vs `a.sum()` $\rightarrow$ typically $\sim$100$\times$ faster (we will measure it in the lab)
- Same $\mathcal{O}(n)$ complexity, different *constant factors*: interpreted vs compiled, element-at-a-time vs contiguous
- **Data locality**: values used together sit together $\rightarrow$ one cache operation brings the next ones along. $\mathcal{O}(\cdot)$ hides the dominating factors
- The principle is known as **data-oriented design**: array-of-structs $\rightarrow$ struct-of-arrays. NumPy, pandas and Parquet all use it
- Also why data work stays interactive: a 2-second step run 100 times = 3 minutes (too long); 200 ms keeps you interactive

PyTorch: the same idea, on a GPU (*)

```
import torch
t = torch.from_numpy(a)        # same buffer, no copy
t = t.to("cuda")               # same code, now on the GPU
(t * 1.09).mean()              # same expression, same semantics
```

- A **torch.Tensor** is an `ndarray` with two extras: a device (CPU/GPU) and autograd. Shapes, dtypes, broadcasting, slicing = same ideas
- This means array-shaped code is portable: written as whole-array expressions, it moves CPU $\rightarrow$ GPU naturally. A Python `for` loop does not
- Same data-oriented design: contiguous typed blocks, moved once, operated on in batches
- This is why we use NumPy: it is the model underneath pandas, scikit-learn and PyTorch alike

Pandas: labeled tables

What pandas is for

- NumPy arrays are homogeneous and unlabeled: one dtype for everything, columns identified by position (`a[:, 7]` — good luck)
- Real tables are heterogeneous: timestamps next to floats next to categorical codes, with missing values sprinkled in
- **pandas** = labeled, mixed-type tables on top of NumPy, plus boilerplate: readers for a lot of formats, missing-data rules, dates, strings, grouping, joins, etc.
- The data frame is an old idea from S/R, brought to Python by Wes McKinney (2008) for financial time series
- Principle: pure numerics → NumPy; a table with names and types → pandas

Series and DataFrame

- **Series** = a 1-D NumPy array + labels (an index)
- **DataFrame** = a table: named columns, each a Series sharing one row index
- The “model” for the course: DataFrame $\approx$ SQL table $\approx$ one CSV file

Reading data

```python
import pandas as pd
df = pd.read_parquet("taxi_2025_01_sample.parquet")
df = pd.read_csv("taxi.csv", parse_dates=["tpep_pickup_datetime"])
```

- `read_csv`, `read_parquet`, `read_json`, …: one call resulting in one DataFrame
- CSV needs help: `parse_dates=`, `sep=`, `dtype=` → CSV carries no types
- After loading always do: `df.head()`, `df.shape`, `df.info()`

Inspecting: what did I just load?

```
df.shape          # (200000, 20)
df.dtypes         # fare_amount float64, tpep_pickup_datetime dt64...
df.describe()     # count / mean / std / min / quartiles / max
```

- `describe()` is your first data-quality instrument: a negative `min` on `fare_amount`? a `max` trip distance of 100 000 miles? Real data has both.

Selecting columns and rows

```
df["fare_amount"]                    # one column -> Series
df[["fare_amount", "tip_amount"]]    # list of columns -> DataFrame
df.loc[42]                            # row by index LABEL
df.iloc[0:5]                          # rows by POSITION
```

- **loc** = by label, **iloc** = by integer position — mixing them up is the classic week-1 bug
- `df.loc[rows, cols]` does both at once: `df.loc[df.fare_amount < 0, "fare_amount"]`

Filtering: boolean masks again

```
busy = df[df["passenger_count"] >= 4]
weird = df[(df["trip_distance"] == 0) & (df["fare_amount"] > 50)]
```

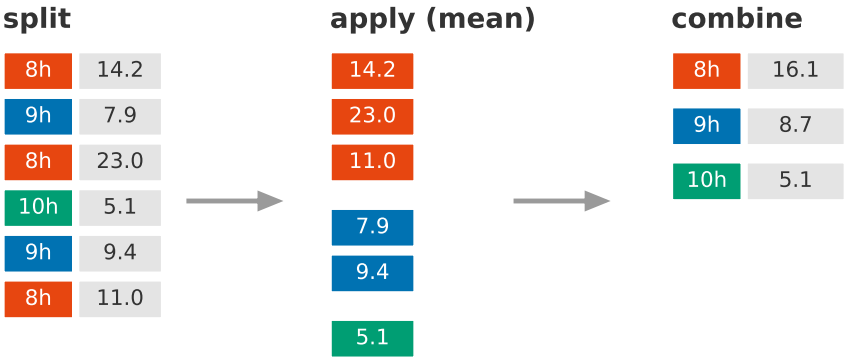
- Same as NumPy: mask, then index
- `weird` above is not hypothetical — you will count these trips in today's lab (zero distance, 50 $ fare… data quality issue!)
- Same operation in SQL: `WHERE trip_distance = 0 AND fare_amount > 50`

Making new columns

```
dur = df["tpep_dropoff_datetime"] - df["tpep_pickup_datetime"]
df["duration_min"] = dur.dt.total_seconds() / 60
df["pickup_hour"] = df["tpep_pickup_datetime"].dt.hour
```

- Assignment to a new name adds a column — vectorized, no loop
- **.dt** accessor: hour, weekday, date arithmetic on datetime columns, very useful for temporal analysis
- Derived columns are the first step of feature construction

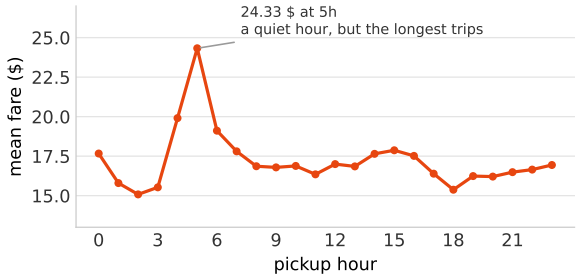
groupby: split → apply → combine



- "*mean fare per pickup hour*" in one line:
  `df.groupby("pickup_hour")["fare_amount"].mean()`
- Same as SQL `GROUP BY` (same keyword even!)

Plotting: matplotlib in one slide

```
ax = df.groupby("pickup_hour")["fare_amount"].mean().plot()
ax.set_xlabel("pickup hour"); ax.set_ylabel("mean fare ($)")
```



- `.plot` on a Series is a thin wrapper over **matplotlib**
- The 5 h spike is the point of plotting: one of the quietest hours, but the longest trips = airport runs
- A plot can be a data-quality instrument
- Label both axes, with units

Lab and next week

Today's lab (right after the break)

- Environment check: everyone's `verify_setup.py` green
- Pandas on 200 000 real NYC taxi trips: load Parquet → inspect → filter → derive columns → guided `groupby` → one (simple) plot
- Plus a first taste of data quality: negative fares, zero-distance trips (they exist!)

Before next week

- Read the project handout (Moodle) and browse the source list
- Look at your source before you register it: open one endpoint in a browser and see what actually comes back — all but TMDB and Twitch answer without credentials
- M0 is due before the week-3 lab, as form on Moodle:
  - create a GitHub account (need it for API tokens)
  - TMDB key or Twitch app (with 2FA) *if* you pick those sources
- Next lecture on relational data & advanced SQL: joins that build datasets, sampling, and why SQL is **not** dead

References

- Sculley et al., *Hidden Technical Debt in Machine Learning Systems*, NeurIPS 2015. https://papers.nips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html
- Anaconda, *State of Data Science Report* 2022. https://www.anaconda.com/state-of-data-science-report-2022
- Grinsztajn, Oyallon, Varoquaux, *Why do tree-based models still outperform deep learning on typical tabular data?*, NeurIPS 2022. https://arxiv.org/abs/2207.08815
- Zha et al., *Data-centric Artificial Intelligence: A Survey*, 2023. https://arxiv.org/abs/2303.10158
- Documentation: https://pandas.pydata.org — https://numpy.org — https://matplotlib.org
- Apache Parquet: https://parquet.apache.org — Fabian, *Data-oriented Design*: https://www.dataorienteddesign.com/dodbook/
- NYC TLC Trip Record Data: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page