Python for Finance: NumPy, Matplotlib, and Pandas

Amedeo Andriollo

Finance Department (DRM)
Université Paris Dauphine - PSL

Structure of the course

- Email: amedeo.andriollo@dauphine.psl.eu
- Grading: 100% final exam.
- Office hours: feel "free" to (DO) come: B612.
- Slides/materials are updated regularly. Report a typo/mistake: corrections help the class. Slides are partially based on last year's (prof. Imbet).
- This session:
  - NumPy: arrays, shapes and axes;
  - Matplotlib: figures that answer a question;
  - Pandas: labelled tables, grouping, merging, reshaping;
  - Putting it together: a small data report.
- Companion notebook: `Session2_Companion.ipynb` + `data/session2_sales.csv`.

Five questions to answer before any data operation

- Session 1: *what does this syntax do?*
  ↪ Session 2: *what will this operation do to my data?*
- Before running a line of NumPy or Pandas, with respect of an operation, ask yourself:
  1. **Shape**: which dimensions go in vs. out?
  2. **Axis/labels**: along which axis/labels does it act?
  3. **Broadcast/align**: are two objects matched by position and shape?
  4. **Copy/view/mutation**: did I just change the original/copy?
  5. **Missing values**: what happens to `NaN` here?

! Most bugs in data code raise no error: they silently return the wrong shape, labels or values.

Warm-up: slices and references

- Slices use half-open intervals `[start:stop:step]`: mind that `stop` is *excluded*.

```
prices = [10, 11, 12, 13, 14]        # list of prices
print(prices[1:3], prices[-2:], prices[::2])
```

```
[11, 12] [13, 14] [10, 12, 14]
```

- Assigning does not copy: two names can refer to the *same* object.

```
backup = prices          # the same list, under a second name (just renamed)
safe = prices.copy()     # a new, independent list
prices.append(15)        # you add on prices, and so backup
print(backup)
print(safe)
```

```
[10, 11, 12, 13, 14, 15]
[10, 11, 12, 13, 14]
```

↪ With arrays and tables, "same object or copy?" becomes a crucial.

NumPy

Arrays vs. lists

- A NumPy `ndarray` is a fixed-size block of values sharing one commond data type.
  ↪ operators act elementwise, and the loop runs in compiled code (highly optimized with respect to standard looping).

```
import numpy as np

x_list = [1, 2, 3]
x_arr = np.array([1, 2, 3])
print(x_list * 2)
print(x_arr * 2)
print(x_list + x_list)
print(x_arr + x_arr)
```

```
[1, 2, 3, 1, 2, 3]
[2 4 6]
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

! Same operator but different meaning.

- Arrays have no `append`, rather they have `np.append`, which builds a new array each time.

ndarray: shape, ndim, size, dtype

- Running example: units sold on 5 week days (rows) of 3 products (columns: coffee, tea, cake).

```
units = np.array([[12, 7, 3],
                  [15, 6, 4],
                  [11, 9, 2],
                  [14, 8, 5],
                  [13, 7, 3]])
print(units.shape, units.ndim, units.size)
print(units.dtype)
print(len(units))
```

```
(5, 3) 2 15
int64       # because they are all integer.
5
```

- `shape` is a tuple with one entry per axis: axis 0 = days, axis 1 = products.
- `len` only counts along the first axis (axis 0).

↪ The type is integer: what is one entry of the array is a float? (Next slide)

Data types are inferred

- The dtype is inferred from *all* elements: one array corresponds to one dtype.

```
print(np.array([1, 2, 3]).dtype)
print(np.array([1, 2, 3.5]).dtype)
print(np.array([1, 2, "3"]).dtype)
```

```
int64
float64      # floats
<U21         # strings
```

! Later assigning to an array never changes its original dtype.

```
counts = np.array([10, 20, 30])
counts[0] = 2.7                      # truncated, no warning
print(counts)
small = np.array([100, 120], dtype=np.int8)
print(small + small)                 # int8 holds -128..127: wraps around
```

```
[ 2 20 30]
[-56 -16]
```

↪ Integer dtypes silently truncate and overflow.

Creating and reshaping arrays

```
print(np.arange(6))     # min max values and how many values
print(np.linspace(0, 1, 5))      # min max values and how many steps
print(np.zeros((2, 3)))
grid = np.arange(12).reshape(3, -1)
print(grid.shape)
print(grid)
```

```
[0 1 2 3 4 5]
[0.   0.25 0.5  0.75 1.  ]
[[0. 0. 0.]
 [0. 0. 0.]]
(3, 4)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

- `arange` excludes its end (like `range`); `linspace` includes it.
- In `reshape`, the `-1` means "infer this size" (#entries/3=4). Total size can never change:

```
grid.reshape(5, -1)
```

```
ValueError: cannot reshape array of size 12 into shape (5,newaxis)
```

Vectorisation to replace a loop

- Suppose we need to apply a 10% discount to every price above 50, for a million prices.

```
rng = np.random.default_rng(0)              # set the seed for replicability
prices = rng.uniform(1, 100, size=1_000_000)         # random prices between 1 and 100

discounted = []                         # Session 1: a loop
for p in prices:
    discounted.append(p * 0.9 if p > 50 else p)

fast = np.where(prices > 50, prices * 0.9, prices)   # Session 2: vectorised
print(np.allclose(discounted, fast))
```

```
True
```

- Same result; the vectorised line is typically tens of times faster.

! A loop *over an array*, or built-in `sum(arr)`, is not vectorised: often slower than a list.

↪ Vectorising = describe the operation on whole arrays and let NumPy run the loop.

Indexing and slicing in two dimensions

- `arr[row, col]`: each position takes an integer, a slice, or a list of integers.

```
units = np.array([[12, 7, 3],
                  [15, 6, 4],
                  [11, 9, 2],
                  [14, 8, 5],
                  [13, 7, 3]])
```

```
print(units[1, 2])        # day 1, product 2
print(units[1])           # the whole of day 1
print(units[1:3, ::2])    # days 1-2, every other product
col = units[:, 0]         # an integer drops that axis
col_2d = units[:, [0]]    # a list keeps it
print(col.shape, col_2d.shape)
```

```
4
[15  6  4]
[[15  4]
 [11  2]]
(5,) (5, 1)
```

! `(5,)` is a 1-D array; `(5, 1)` is a 2-D column. They look similar but not quite.

- Slices obey the same half-open rule as lists, independently on each axis.

Boolean masks

- Suppose we need to count many units are above ten? And not between ten and fourteen?
- A comparison returns an array of booleans with the *same shape*.

```
busy = units > 10
print(busy[:2])
print(busy.sum(), busy.mean())     # True counts as 1
```

```
[[ True False False]
 [ True False False]]
5 0.3333333333333333
```

- Combine with & (and), | (or), ~ (not). Parentheses are required.

```
both = (units > 10) & (units < 14)
print(both.sum())
(units > 10) and (units < 14)
```

```
3
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

! mind that `and`/`or` ask for a *single* truth value vs. an array holds many.

Selecting and replacing with masks

- Indexing with a mask selects the matching elements, as a 1-D array.
- Assigning through a mask writes into the array itself.

```
print(units[units > 12])

capped = units.copy()        # a copy that will be capped
capped[capped > 12] = 12         # in place, on the copy
print(capped[:, 0])

labels = np.where(units > 10, "busy", "quiet")       # first compute units > 0 and then if true: busy
print(labels[0])         # the first row is: [12, 7, 3]
```

```
[15 14 13]
[12 12 11 12 12]
['busy' 'quiet' 'quiet']
```

- `np.where(condition, a, b)` builds a *new* array, picking from `a` or `b`.

! Both `a` and `b` are computed for every element: `np.where(x > 0, 1 / x, 0)` still divides by zero (and warns).

Reductions and the axis argument

- **Sum:** the axis you name is the axis that is summed/collapsed.

```
print(units.sum())             # everything: one number
print(units.sum(axis=0))       # collapse the days: one total per product
print(units.sum(axis=1))       # collapse the products: one total per day
print(units.mean(axis=0).shape, units.max(axis=1).shape)    # the mean across the 5 days vs. the max per each days
print(units.argmax(axis=0))    # for each product, the day with most units
```

Reductions and the axis argument

- **Sum:** the axis you name is the axis that is summed/collapsed.

```
print(units.sum())              # everything: one number
print(units.sum(axis=0))        # collapse the days: one total per product
print(units.sum(axis=1))        # collapse the products: one total per day
print(units.mean(axis=0).shape, units.max(axis=1).shape)    # the mean across the 5 days vs. the max per each days
print(units.argmax(axis=0))     # for each product, the day with most units
```

```
119
[65 37 17]
[22 25 22 27 23]
(3,) (5,)
[1 2 3]
```

- Shape `(5, 3)`: `axis=0` removes the 5, `axis=1` removes the 3.

! "`axis=0` means rows" is a trap: it means *across* the rows
↪ one result per column.

- Same rule in Pandas: `df.sum()` (default `axis=0`) gives one value per column.

Broadcasting: the shape rule

- Compare shapes **from the right**. Each pair of sizes must be equal, or one of them must be 1 (or missing).
- A size-1 or missing dimension is stretched, without copying the data.

```
units      (5, 3)          units      (5, 3)
price         (3,)         day_total     (5,)
result     (5, 3)   OK     3 vs 5  ->  error
```

```
price = np.array([2.5, 2.0, 3.5])     # one price per product
revenue = units * price               # (5, 3) * (3,)
print(revenue[:2])              # first two row should be price*units
day_total = units.sum(axis=1)         # shape (5,)
units - day_total
```

```
[[30.  14.  10.5]
 [37.5 12.  14. ]]
ValueError: operands could not be broadcast together with shapes (5,3) (5,)
```

↪ `price` is matched with the *last* axis (products), which is exactly what we want.

Broadcasting: keepdims and silent mistakes

- Task: each product's share of the day's total, so that every row sums to 1.

```
day_total = units.sum(axis=1, keepdims=True)   # (5, 1) instead of (5,)
shares = units / day_total                      # (5, 3) / (5, 1)
print(shares.shape, shares.sum(axis=1))      # sum over products is expected to be one.
```

```
(5, 3) [1. 1. 1. 1. 1.]
```

! Broadcasting can also succeed when you did not mean it to:

```
a = np.array([1, 2, 3])            # shape (3,)
b = np.array([[10], [20], [30]])   # shape (3, 1)
result = a + b
print(result.shape)
assert result.shape == a.shape, "unexpected broadcast"
```

```
(3, 3)
AssertionError: unexpected broadcast
```

↪ `assert` can help: state the shape you expect with `assert`.

Views and copies

- A **slice** (also `.T` (transpose), and `reshape` when possible) is a *view*: not indepedent data.
- A **mask** or a **list of indices** returns a *copy*.

```
week = np.array([5, 8, 6, 9, 7])
first_days = week[:3]            # slice -> view
first_days[0] = 100          # changing week as well!
busy_days = week[week > 6]       # mask -> copy
busy_days[:] = 0         # does it change week?
print(week)
print(np.shares_memory(week, first_days), np.shares_memory(week, busy_days))
```

```
[100   8   6   9   7]
True False
```

- `first_days` refers to the three entries of `week`.

↪ Need an independent array? Call `.copy()` explicitly.

↪ Want to change the original where a condition holds? One assignment: `week[week > 6] = 0`.

Missing and non-finite values

- `np.nan` ("not a number") marks a missing float; it propagates through arithmetic.

```
#nan
temps = np.array([21.5, np.nan, 23.0, 22.5])
print(temps.mean(), np.nanmean(temps))      # mean with nan vs. ignoring nan.
print(np.nan == np.nan, np.isnan(temps))        # is the nan equal to itself? No! check all elements
#indefinite
ratio = np.array([1.0, 0.0, 2.0]) / np.array([0.0, 0.0, 4.0])
print(ratio, np.isfinite(ratio))
#epsilon-precision
print(0.1 + 0.2 == 0.3, np.isclose(0.1 + 0.2, 0.3))
```

```
nan 22.333333333333332
False [False  True False False]
RuntimeWarning: divide by zero encountered in divide
RuntimeWarning: invalid value encountered in divide
[inf nan 0.5] [False False  True]
False True
```

! Never test `x == np.nan`: use `np.isnan`; `np.isfinite` also rejects `inf`.

- Division by zero on arrays gives `inf`/`nan` and a warning, not an exception.
- Compare floats with `np.isclose` (Session 1: machine epsilon).

Reproducible random numbers

- Create **one generator with a seed**, then draw everything from it.

```
rng = np.random.default_rng(42)
dice = rng.integers(1, 7, size=8)              # 7 is excluded
noise = rng.normal(loc=0, scale=1, size=(2, 3))
print(dice)
print(noise.round(2))
print(np.random.default_rng(42).integers(1, 7, size=8))   # same seed: replicability.
```

```
[1 5 4 3 3 6 1 5]
[[-1.95 -1.3   0.13]
 [-0.32 -0.02 -0.85]]
[1 5 4 3 3 6 1 5]
```

- Same seed ⇒ same numbers: results can be re-run and checked by someone else.
- ! `integers(low, high)` excludes `high` (unlike `random.randint`).
- Avoid the legacy global `np.random.seed` / `np.random.rand`: hidden shared state.

Elementwise versus matrix multiplication

- `*` is elementwise (with broadcasting). `@` is the matrix product: `(n, k) @ (k, m)` → `(n, m)`.

```
revenue_cells = units * price     # (5, 3) *  (3,) -> (5, 3)
revenue_days = units @ price      # (5, 3) @  (3,) -> (5,)
print(revenue_cells.shape, revenue_days.shape)
print(revenue_days)

X = np.ones((2, 3))
Y = np.ones((3, 2))
print((X @ Y).shape)
X * Y
```

```
(5, 3) (5,)
[54.5 63.5 52.5 68.5 57. ]
(2, 2)
ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

↪ `units @ price` is each day's revenue: the sum over products of units × price.

! For `@` the *inner* sizes must match; for `*` the shapes must broadcast.

Solving a linear system

- Task: we lost the price list, but know the units and the revenue of the first three days.
  ↪ the unknown prices p solve `A @ p = b`: 3 equations, 3 unknowns.

```
A = units[:3]                          # (3, 3): units on days 0-2
b = np.array([54.5, 63.5, 52.5])       # revenue on days 0-2
p = np.linalg.solve(A, b)
print(p)
print(np.allclose(A @ p, b))           # try always check the solution
```

```
[2.5 2.  3.5]
True
```

! Do not write `np.linalg.inv(A) @ b`: it does more work and is less accurate than `solve`.

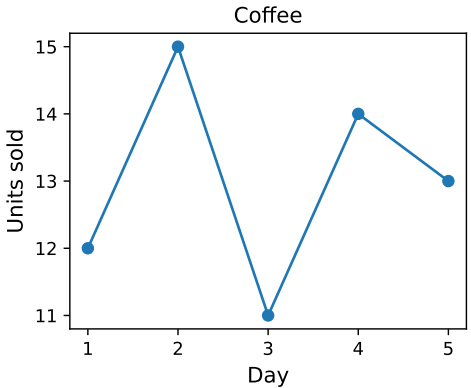
! If one day is a combination of others, `A` is (nearly) singular: `solve` may raise `LinAlgError`, or silently return huge nonsense.
↪ neither function warns reliably: check `np.allclose(A @ p, b)`; never test `det(A) == 0` (Session 1: warnings).

Matplotlib

Figure and Axes

- Two objects at once: a **Figure** (the canvas); an **Axes** (plot area, with x and y).
- Create both explicitly, then call methods on ax.

```
import matplotlib.pyplot as plt

days = np.arange(1, 6)       # x-values are the days
fig, ax = plt.subplots(figsize=(4, 3)) #physical size (inches)
ax.plot(days, units[:, 0], marker="o") # coffee sales over days
ax.set(xlabel="Day", ylabel="Units sold",
       title="Coffee")
```

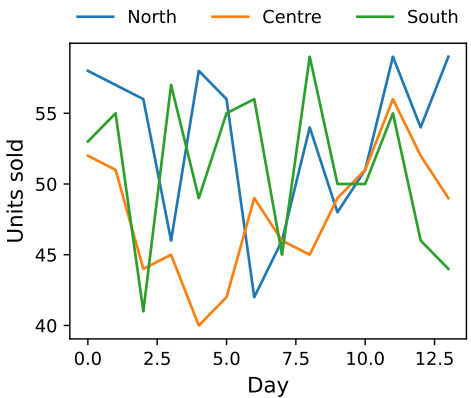


! `plt.title(...)`, `plt.xlabel(...)` act on the "current" Axes: ambiguous.
↪ prefer `ax.set_title`, `ax.set(...)`: you always know which plot you are changing.

Plotting several series

- One `plot` call per series, each with a `label`; `legend` reads the labels.

```
rng = np.random.default_rng(7)
daily = rng.integers(40, 60, size=(14, 3))   # 14 days x 3 shops
shop_names = ["North", "Centre", "South"]

fig, ax = plt.subplots(figsize=(4, 3))
for j, name in enumerate(shop_names):
    ax.plot(daily[:, j], label=name)   # column j: shape (14,)
ax.set(xlabel="Day", ylabel="Units sold")
ax.legend(ncols=3, loc="lower center",
          bbox_to_anchor=(0.5, 1.0), frameon=False)
# legend is meant to sit above the top edge of Axes
```



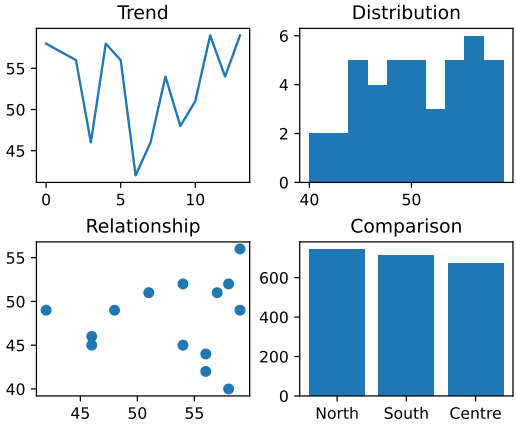
↪ Columns are series (2-D indexing).

- Label the axes with what is measured.

Choosing a chart for the question

- Start from the question, not from the type:
  - **How does it evolve?** → line;
  - **How is it distributed?** → histogram;
  - **Do two things move together?**→ scatter;
  - **Which (category) is largest?**→ sorted bars.

```
fig, axes = plt.subplots(2, 2, figsize=(4.6, 3.8),
                         layout="constrained")
axes[0, 0].plot(daily[:, 0])
axes[0, 1].hist(daily.ravel(), bins=10)
axes[1, 0].scatter(daily[:, 0], daily[:, 1])
totals = daily.sum(axis=0)
order = np.argsort(totals)[::-1]        # largest first
axes[1, 1].bar(np.array(shop_names)[order], totals[order])
titles = ["Trend", "Distribution",
          "Relationship", "Comparison"]
for ax, title in zip(axes.flat, titles):
    ax.set_title(title)
```



A chart helps visualize a problem

- Outlier: someone typed 480 instead of 48. Would the average tell you?

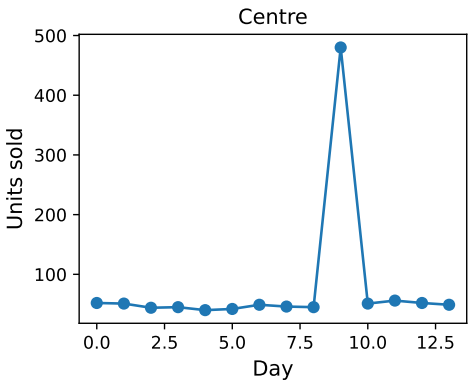
```
daily[9, 1] = 480                    # a typing error

fig, ax = plt.subplots(figsize=(4, 3))
ax.plot(daily[:, 1], marker="o")
ax.set(xlabel="Day", ylabel="Units sold", title="Centre")
print(round(daily[:, 1].mean(), 1))
```

```
78.7
```

- The plot shows one impossible day.

↪ Good rule: plot before you summarise; then repair with one mask assignment:

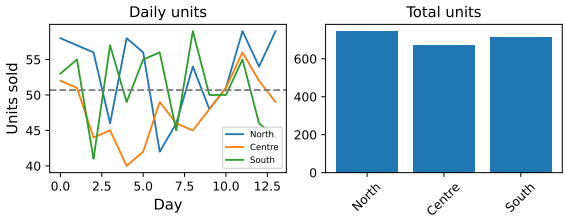
```
daily[daily > 200] = daily[daily > 200] // 10
assert daily[9, 1] == 48
```



Several panels in one figure

```
fig, axes = plt.subplots(1, 2, figsize=(6.4, 2.5),
                         layout="constrained")
assert axes.shape == (2,)        # a 1-D array of Axes
axes[0].plot(daily, label=shop_names)  # a line per column
axes[0].axhline(daily.mean(), color="gray",
                linestyle="--")  # highlight the mean
axes[0].set(xlabel="Day", ylabel="Units sold",
            title="Daily units")
```

```
axes[1].bar(shop_names, daily.sum(axis=0))
axes[1].tick_params(axis="x", labelrotation=45)
axes[1].set(title="Total units")
axes[0].legend(fontsize=7)
fig.savefig("shops.png", dpi=150)
```



! `plt.subplots(2, 2)` gives a 2-D array: `axes[1, 0]`.

- `tick_params` rotates crowded labels; `savefig` writes the file.

Pandas

Series and DataFrame

- Pandas adds **labels** and **table semantics** to array-oriented data work.
- `Series`: values with an *index* (row labels).
  vs. `DataFrame`: named columns sharing one index; each column has its own dtype.

```
import pandas as pd

products = pd.DataFrame({"product": ["coffee", "tea", "cake"],
                         "unit_price": [2.5, 2.0, 3.5]})
print(products)
print(products.index)
print(products["unit_price"].to_numpy())
```

```
  product  unit_price
0  coffee         2.5
1     tea         2.0
2    cake         3.5
RangeIndex(start=0, stop=3, step=1)
[2.5 2.  3.5]
```

↪ NumPy matches values by *position and shape*; Pandas matches them by *label* (index and column names).

Loading and inspecting a local file

- A relative path: the `data` folder next to the notebook.

```
raw = pd.read_csv("data/session2_sales.csv")
print(raw.shape)
print(raw.head(4))
print(raw.dtypes)
```

```
(490, 4)
         date  shop_id product units
0  2026-03-02        1  coffee    37
1  2026-03-02        1     tea    15
2  2026-03-02        1    cake     9
3  2026-03-02        2  coffee    44
date         str
shop_id    int64
product      str
units        str
dtype: object
```

- Always look first: `shape`, `head()`, `dtypes` (check also `info()` and `describe()`).

When numbers are read as text

! `shop_id` "001" became the integer 1; `units` is text (nan are labelled as the word `missing`); `date` is text.

```
print(raw["units"].head(3).sum())    # text: '37' + '15' + '9'
```

```
37159
```

- Repair at load time: declare the missing-value token, keep IDs as text, parse dates.

```
sales = pd.read_csv("data/session2_sales.csv",
                    na_values=["missing"],
                    dtype={"shop_id": "string"},
                    parse_dates=["date"])
print(sales.head(3))
print(sales["units"].isna().sum(), sales["units"].dtype)
```

```
        date shop_id product  units
0 2026-03-02     001  coffee   37.0
1 2026-03-02     001     tea   15.0
2 2026-03-02     001    cake    9.0
8 float64
```

↪ An integer column containing a missing value becomes `float64`.

Selecting columns

- One label is a Series (1-D). A list of labels is a DataFrame (2-D).
  ↪ compare with `(n,)` versus `(n, 1)` in NumPy.

```
one = sales["units"]
two = sales[["shop_id", "units"]]
print(type(one).__name__, one.shape)
print(type(two).__name__, two.shape)
```

```
Series (490,)
DataFrame (490, 2)
```

! Attribute access (`sales.units`) is unreliable: a column called `size` or `count` collides with an existing attribute: does it return the column `size`?

```
counts = pd.DataFrame({"size": [3, 5]})
print(counts.size, counts["size"].tolist())   # attribute vs column
```

```
2 [3, 5]
```

↪ Use brackets: `df["col"]` always means the column.

`loc` vs. `iloc`

- `.loc[rows, cols]` selects by **label**; `.iloc[rows, cols]` selects by **position**.

! `.loc` slices *include* the end label; `.iloc` slices *exclude* the end position.

```
stock = pd.DataFrame({"units": [12, 7, 3, 9, 5],
                      "price": [2.5, 2.0, 3.5, 1.5, 4.0]},
                     index=["a", "b", "c", "d", "e"])
print(stock.loc["b":"d"])
print(stock.iloc[1:3])
print(stock.loc[["a", "e"], "price"].tolist(), stock.iloc[0, 1])    # to list convert into list
```

loc vs. iloc

- `.loc[rows, cols]` selects by **label**; `.iloc[rows, cols]` selects by **position**.

! `.loc` slices *include* the end label; `.iloc` slices *exclude* the end position.

```
stock = pd.DataFrame({"units": [12, 7, 3, 9, 5],
                      "price": [2.5, 2.0, 3.5, 1.5, 4.0]},
                     index=["a", "b", "c", "d", "e"])
print(stock.loc["b":"d"])
print(stock.iloc[1:3])
print(stock.loc[["a", "e"], "price"].tolist(), stock.iloc[0, 1])    # to list convert into list
```

```
   units  price
b      7    2.0
c      3    3.5
d      9    1.5
   units  price
b      7    2.0
c      3    3.5
[2.5, 4.0] 2.5
```

`loc` vs. `iloc`: labels attached to their rows

```
orders = pd.DataFrame({"units": [8, 3, 12, 5]})
ranked = orders.sort_values("units", ascending=False)
print(ranked)
print(ranked.loc[0, "units"], ranked["units"].iloc[0])
```

```
   units
2     12
0      8
3      5
1      3
8 12
```

- Sorting reorders the rows; every row keeps its label.
  ↪ `.loc[0]` is "the row labelled 0" (now second); `.iloc[0]` is "the first row".
- `reset_index(drop=True)` gives fresh labels 0, 1, 2, ... when you want them.

! brackets can do several things: `df["col"]` (series), `df[["c1", "c2"]]` (df), `df[mask]` (filter). Try to use `.loc`/`.iloc`;
never write `series[0]` (label 0 or first position?).

Filtering rows

- A condition on columns gives a Boolean `Series`: use it as the row selector of `.loc`.

```
tea_busy = sales.loc[(sales["product"] == "tea") & (sales["units"] > 38),
                     ["date", "shop_id", "units"]]      # dropping the product column
print(tea_busy)
two_shops = sales.loc[sales["shop_id"].isin(["001", "005"])]        # filter keeping shop 001 and 005
print(len(two_shops))
```

```
          date shop_id  units
148 2026-03-14     002   41.0
227 2026-03-21     001   40.0
230 2026-03-21     002   39.0
315 2026-03-29     002   41.0
459 2026-04-10     002   42.0
471 2026-04-11     002   40.0
250
```

- Same rules as NumPy masks: `&`, `|`, `~`, and parentheses around each condition.
- `.loc[row_mask, column_list]` filters rows and selects columns in one step.

↪ Recall that `NaN` compared with `>`, `<` or `==` gives `False` by default.

Creating columns without loops

```
basket = pd.DataFrame({"units": [3, 10, 6], "unit_price": [2.5, 2.0, 3.5]})
basket["revenue"] = basket["units"] * basket["unit_price"]
basket["label"] = np.where(basket["units"] >= 5, "large", "small")
print(basket)

bigger = basket.assign(revenue_k=basket["revenue"] / 1000)
# new dataset: bigger contains basket + a new column (revenue_k), to check:
print("revenue_k" in bigger.columns, "revenue_k" in basket.columns)
```

```
   units  unit_price  revenue  label
0      3         2.5      7.5  small
1     10         2.0     20.0  large
2      6         3.5     21.0  large
True False
```

- Column arithmetic is vectorised: one expression for all rows.
- `assign` *returns* a new DataFrame; the original is unchanged unless you reassign it.

! `df.apply(func, axis=1)` calls a Python function once per row: a hidden loop (for each row: run func(row)).
↪ use it only when no (optimized) vectorised form exists.

Changing values safely

! Never rely on whether a Pandas selection is a view or a copy.

- To **modify the original**: one assignment `df.loc[rows, column] = value`.
- To get an **independent** subset: `.copy()`.

```
low = stock["units"] < 6
stock.loc[low, "units"] = 6                    # modifies stock
# Do NOT write: stock[low]["units"] = 6        (chained assignment)

cheap = stock.loc[stock["price"] < 3].copy()   # independent
cheap["price"] = 0.0                           # stock is unaffected
print(stock)
```

```
   units  price
a     12    2.5
b      7    2.0
c      5    6.0
d      9    1.5
e      6    4.0
```

! `inplace=True` (modify the existing object directly) returns `None`: write `df = df.sort_values("units")` instead.

Index alignment

- Arithmetic between two Series matches **labels**, not positions.
  A label present on one side only gives `NaN`.

```
target = pd.Series([100, 80, 90], index=["001", "002", "003"])
actual = pd.Series([95, 85, 70], index=["002", "003", "005"])
print((actual - target).to_dict())
print(actual.sub(target, fill_value=0).to_dict())  # matching by index label and convert to Py dictionary
```

```
{'001': nan, '002': 15.0, '003': -5.0, '005': nan}
{'001': -100.0, '002': 15.0, '003': -5.0, '005': 70.0}
```

- Assigning a column: a `Series` is aligned by label, an array is taken by position.

```
frame = pd.DataFrame({"units": [1, 2, 3]}, index=["x", "y", "z"])
frame["by_label"] = pd.Series([10, 20, 30], index=["z", "y", "x"])
frame["by_position"] = np.array([10, 20, 30])
print(frame)
```

```
   units  by_label  by_position
x      1        30           10
y      2        20           20
z      3        10           30
```

Missing values in tables

```
units_col = sales["units"]      # column as series
print(units_col.isna().sum(), units_col.count(), units_col.size)      # count missing values, count non-missing values,
      total
print(round(units_col.mean(), 2), units_col.to_numpy().mean())       # pandas ignores nan vs. numpy
print(len(sales), len(sales.dropna(subset=["units"]))) # total and after removing nan
print(round(units_col.fillna(0).mean(), 2))
print(pd.Series([np.nan, np.nan]).sum())        # sum of nan is 0
```

```
8 482 490
24.84 nan
490 482
24.43
0.0
```

- Pandas reductions **skip** `NaN` by default; NumPy's do not.
- `count()` = non-missing values; `size` = all rows.
- `dropna` removes rows; `fillna(0)` changes the mean. Decide deliberately, never by default.

! The sum of nothing but missing values is `0.0`, not `NaN`: a silent zero in a grouped total.

Split, apply, combine

- `groupby`: **split** the rows by key, **apply** a reduction to each group, **combine** the results into a table indexed by the keys.

```
mini = pd.DataFrame({"shop_id": ["001", "002", "001", "002", "001"],
                     "units": [10, 4, 6, 8, 2]})
print(mini.groupby("shop_id")["units"].sum())       # summing units by shops
print(sales.groupby("product")["units"].sum())
```

```
shop_id
001    18
002    12
Name: units, dtype: int64
product
cake      1948.0
coffee    6929.0
tea       3096.0
Name: units, dtype: float64
```

↪ The result has one row per group; the keys become the index.

! Rows whose key is missing are dropped from the groups by default.

Named aggregation

- Several statistics in one pass: `new_name=(column, function)`.

```
summary = (sales.groupby("shop_id")         # group all rows by shop
                .agg(total=("units", "sum"),
                     days=("date", "nunique"),
                     avg=("units", "mean"))
                .reset_index())
print(summary.round(1))
```

```
  shop_id   total  days   avg
0     001  3231.0    42  26.3
1     002  4612.0    42  37.2
2     003  2295.0    39  20.5
3     005  1835.0    42  14.9
```

- `reset_index()` turns the group keys back into an ordinary column.
- Use built-in names (`"sum"`, `"mean"`, `"nunique"`, `"count"`): vectorised and readable.

! `groupby(...).apply(lambda g: ...)` is slower and its behaviour has changed across versions.

To aggregate or to transform?

- agg: one row *per group*. `transform`: one value *per original row*, aligned on the original index.

```
totals = mini.groupby("shop_id")["units"].agg("sum")
per_row = mini.groupby("shop_id")["units"].transform("sum")
print(totals.shape, per_row.shape)
mini["share"] = mini["units"] / per_row
print(mini)
```

```
(2,) (5,)
  shop_id  units     share
0     001     10  0.555556
1     002      4  0.333333
2     001      6  0.333333
3     002      8  0.666667
4     001      2  0.111111
```

↪ Similar logic of `keepdims=True` in NumPy:
↪ keep the original shape so that a row can be divided by its group total.

Merging a table

- Many order lines per product on the left, while one price per one product on the right: **many-to-one**.

```
lines = pd.DataFrame({"product": ["tea", "cake", "tea", "coffee"],
                      "units": [3, 1, 2, 4]})
merged = lines.merge(products, on="product", how="left",
                     validate="many_to_one")
merged["revenue"] = merged["units"] * merged["unit_price"]
print(merged)
print(len(lines), len(merged))
```

```
  product  units  unit_price  revenue
0     tea      3         2.0      6.0
1    cake      1         3.5      3.5
2     tea      2         2.0      4.0
3  coffee      4         2.5     10.0
4 4
```

- `how="left"` keeps every row of the left table, matched or not.
- `validate="many_to_one"` raises an error if the right table repeats a key.

↪ A many-to-one left merge must keep the row count: check `len` before and after.

Auditing a merge

```
shops = pd.DataFrame({"shop_id": ["001", "002", "003", "004"],
                      "region": ["South", "North", "North", "North"]})
audit = sales.merge(shops, on="shop_id", how="left", indicator=True)        # each shop_id should map to one region
print(audit["_merge"].value_counts().to_dict())     # where each row came from
print(audit.loc[audit["_merge"] == "left_only", "shop_id"].unique().tolist())       # which shop IDs failed to match?
```

```
{'both': 365, 'left_only': 125, 'right_only': 0}
['005']
```

! Duplicate key in the lookup table silently multiplies rows.

```
dup = pd.concat([products, products.iloc[[1]]])     # duplicates the "tea" row
print(len(lines), len(lines.merge(dup, on="product"))) # one "tea" row may match two "tea" rows: from 4 to 6
lines.merge(dup, on="product", validate="many_to_one")
```

```
4 6
MergeError: Merge keys are not unique in right dataset; not a many-to-one merge
```

- `indicator=True` shows which rows found no partner (`left_only`): here an unknown shop.

↪ Audit every merge: row counts, `validate`, and the `_merge` column.

From long to wide

- *Long*: one row per observation. *Wide*: one row per day, one column per shop.

```
long = pd.DataFrame({"day": ["Mon", "Mon", "Mon", "Tue", "Tue"],
                     "shop": ["001", "002", "001", "001", "002"],
                     "units": [5, 7, 2, 6, 4]})
long.pivot(index="day", columns="shop", values="units")     # one cell for each (day, shop). (Mon, 001) has two values
```

```
ValueError: Index contains duplicate entries, cannot reshape
```

```
wide = long.pivot_table(index="day", columns="shop",
                        values="units", aggfunc="sum")      # aggregate duplicates
print(wide)
```

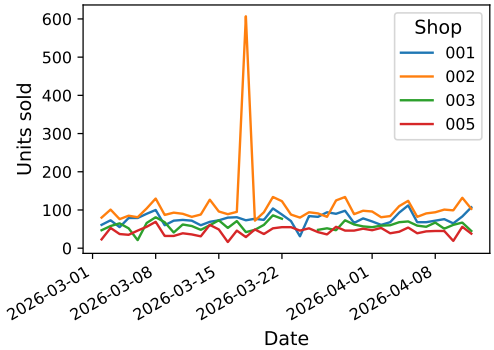
```
shop  001  002
day
Mon     7    7
Tue     6    4
```

- `pivot` requires unique (index, column) pairs; `pivot_table` aggregates duplicates with `aggfunc`.

↪ `wide.to_numpy()` is a `(days, shops)` array: back to NumPy axes.

From a table to a figure

```
daily_units = sales.pivot_table(index="date",
                                columns="shop_id",
                                values="units",
                                aggfunc="sum")
fig, ax = plt.subplots(figsize=(4.6, 3.2))
for shop in daily_units.columns:
    ax.plot(daily_units.index, daily_units[shop],
            label=shop)
ax.set(xlabel="Date", ylabel="Units sold")
ax.legend(title="Shop")
fig.autofmt_xdate()
```



↪ One glance finds the typing error (shop 002).

- `daily_units.plot(ax=ax)` draws the same lines and returns that `ax`.

Putting it together

Challenge: a weekly shop report

- Data: `data/session2_sales.csv`, plus the `products` and `shops` tables built in Python (notebook, section 5).
  1. **Load and repair**: missing-value token, IDs as text, dates.
  2. **Fix one typing error**: the only row with more than 300 units holds ten times its true value. Correct it with a single `.loc` assignment.
  3. **Revenue**: merge `products` with `validate="many_to_one"`, check the row count, compute `revenue`.
  4. **Known shops only**: merge `shops` with `indicator=True`; count the rows with no match; keep the rows found in both tables.
  5. **Weekly table**: weeks $\times$ shops of total revenue with `pivot_table`; then, in NumPy, each shop's share of every week (each row sums to 1).
  6. **Figure**: weekly revenue per shop, one line per shop, labelled axes and legend.

↪ Every step uses only today's material. Answer the five questions at each step.

Challenge: checks and hints

- Your result should pass checks like these (the notebook prints the target values):

```
assert report_sales["units"].max() == 78.0          # step 2: the typo became 58
assert n_unmatched == 125                            # step 4: rows of the unknown shop
assert weekly.shape == (6, 3)                        # step 5: 6 weeks x 3 known shops
assert np.allclose(shares.sum(axis=1), 1)            # step 5: keepdims done right
```

- Hints, by step:
  - 1: “When numbers are read as text”; 2: “Changing values safely”;
  - 3–4: “Merging a lookup table”, “Auditing a merge”;
  - 5: `sales["date"].dt.to_period("W").dt.start_time` gives the Monday of each week; “From long to wide”, then “Broadcasting: keepdims and silent mistakes”;
  - 6: “Plotting several series”, “From a table to a figure”.

Five questions, revisited

1. **Shape**: NumPy `.shape`, `(n,)` vs `(n, 1)`; Pandas `Series` vs `DataFrame`, `agg` vs `transform`.
2. **Axis / labels**: the named axis is collapsed; `.loc` is inclusive labels, `.iloc` half-open positions.
3. **Broadcast / align**: NumPy compares shapes from the right; Pandas aligns on index labels (and merges on keys).
4. **Copy / view / mutation**: NumPy slices are views, masks are copies; in Pandas use `.copy()` or one `.loc[...] = ...`.
5. **Missing values**: NumPy propagates `nan`; Pandas skips it in reductions; merges and unmatched labels create it.

↪ Check with `shape`, `len`, `isna().sum()` and `assert`, before trusting any result.

Homework (Optional but HIGHLY Recommended)

- Finish the challenge (notebook, section 5), then:
  1. Add a second panel showing each shop's weekly share.
  2. For each shop, report the week with its highest share (`argmax(axis=0)`, mapped back to the week labels).
  3. Wrap steps 1–5 in a function `weekly_report(path)` with type hints and a docstring (Session 1).
- Don't waste your time using ChatGPT, you won't have access to it during the final exam.

! Use it to learn.