Lecture 17 | Matplotlib

Create readable visual explanations of computed results



What should a chart answer?

Matplotlib turns Python data into figures: the basic workflow is to create plotting objects, draw data, add labels and display or save the result

In this lesson we use the object-oriented interface: `fig, ax = plt.subplots()` creates a Figure and one Axes

Call a plotting method on ax, for example `.plot()`, `.bar()`, `.scatter()`, `.hist()` or `.boxplot()`

Plotting methods receive the values to draw and optional arguments controlling their appearance

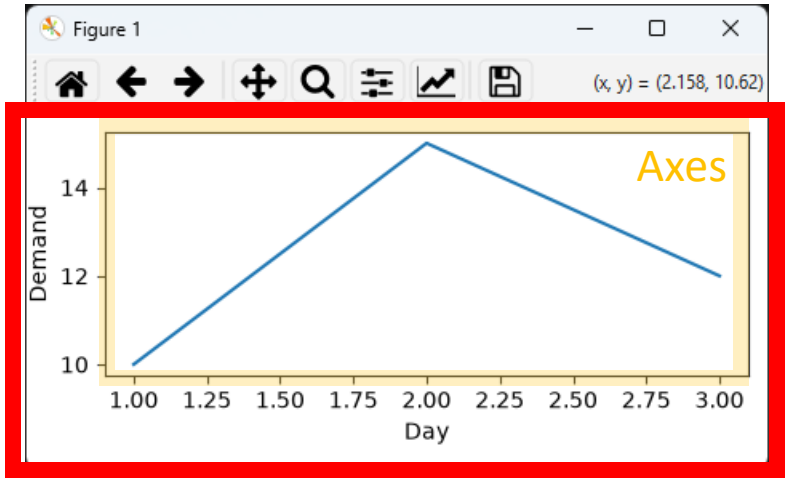
The chart type depends on the data: ordered values → line; categories → bars; paired measurements → scatter; one distribution → histogram or box plot

Your first Matplotlib program

- Import `matplotlib.pyplot as plt`: pyplot provides functions to create and display Matplotlib figures
- `fig, ax = plt.subplots()` creates two objects:
  - `fig` is the whole ***Figure***;
  - `ax` is the ***Axes*** where data are plotted;
- a Figure may contain **multiple axes for plots**
- These objects are used to build the plot: `ax.plot(x, y)` uses the first sequence for horizontal coordinates and the second for vertical coordinates; values are paired by position
- `ax.set_xlabel(...)`, `ax.set_ylabel(...)` and `ax.set_title(...)` add text to the Axes
- `plt.show()` displays the figure; creating it and drawing on it are separate steps

```python
import matplotlib.pyplot as plt

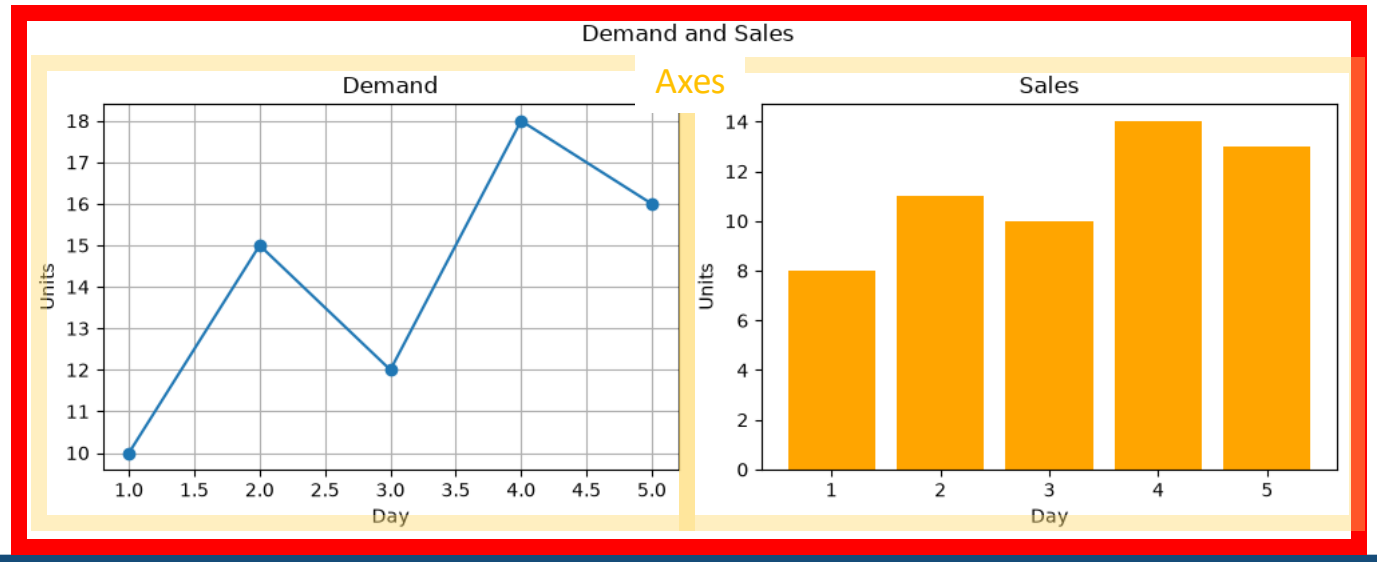
fig, ax = plt.subplots()
ax.plot([1, 2, 3], [10, 15, 12])
ax.set_xlabel("Day")
ax.set_ylabel("Demand")
plt.show()
```



Figure

Figure and Axes: not the same object

- A **Figure** is the complete image or canvas; an **Axes** is one plotting area inside the Figure (despite its plural name)
- Use ax to add graphical elements and configure one plot: `.plot()`, `.bar()`, `.set_title()`, `.set_xlabel()` and others
- Use `fig` for figure-level operations, especially `.savefig()` and arranging multiple Axes
- **One Figure can contain several Axes:** `plt.subplots()` without row/column arguments creates a single Axes
- The variable names `fig` and `ax` are conventions, not required keywords; the objects they refer to matter




```python
import matplotlib.pyplot as plt

# One figure with two axes side by side.
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

days = [1, 2, 3, 4, 5]

ax1.plot(days, [10, 15, 12, 18, 16], marker="o")
ax1.set_title("Demand")
ax1.set_xlabel("Day")
ax1.set_ylabel("Units")
ax1.grid(True)

ax2.bar(days, [8, 11, 10, 14, 13], color="orange")
ax2.set_title("Sales")
ax2.set_xlabel("Day")
ax2.set_ylabel("Units")

fig.suptitle("Demand and Sales")
fig.tight_layout()
plt.show()
```

Common methods to customize a plot

Axes methods are called on `ax`; `pyplot` functions such as `plt.xticks()` act on the current Axes

Method / function	Useful parameters	Effect	Example in Python
`.set_title()`	`label, fontsize, loc`	Adds or positions the plot title	`ax.set_title('Revenue', loc='left')`
`.set_xlabel() / .set_ylabel()`	`xlabel or ylabel, fontsize`	Labels the horizontal or vertical axis	`ax.set_xlabel('Month')`
`.set_xlim() / .set_ylim()`	`left/right or bottom/top`	Restricts the range displayed on an axis	`ax.set_ylim(0, 120)`
`.set_xticks() / .set_yticks()`	`ticks, labels (optional)`	Selects tick positions and optionally replaces their labels	`ax.set_xticks([1, 2], ['Jan', 'Feb'])`
`plt.xticks() / plt.yticks()`	`ticks, labels, rotation, fontsize`	Pyplot functions for tick positions, labels and their appearance	`plt.xticks(rotation=45)`
`.grid()`	`visible, axis, linestyle, alpha`	Shows and styles reference grid lines	`ax.grid(True, axis='y', alpha=0.4)`
`.legend()`	`loc, title, fontsize`	Displays labels given to plotted series	`ax.legend(loc='upper right')`
`.axhline() / .axvline()`	`y or x, color, linestyle`	Adds a horizontal or vertical reference line	`ax.axhline(0, color='gray', linestyle='--')`
`.annotate()`	`text, xy, xytext, arrowprops`	Adds text pointing to a specific data point	`ax.annotate('Peak', xy=(3, 90))`

Example setup: `fig, ax = plt.subplots()` | Docs: matplotlib.org/stable/api/axes_api.html

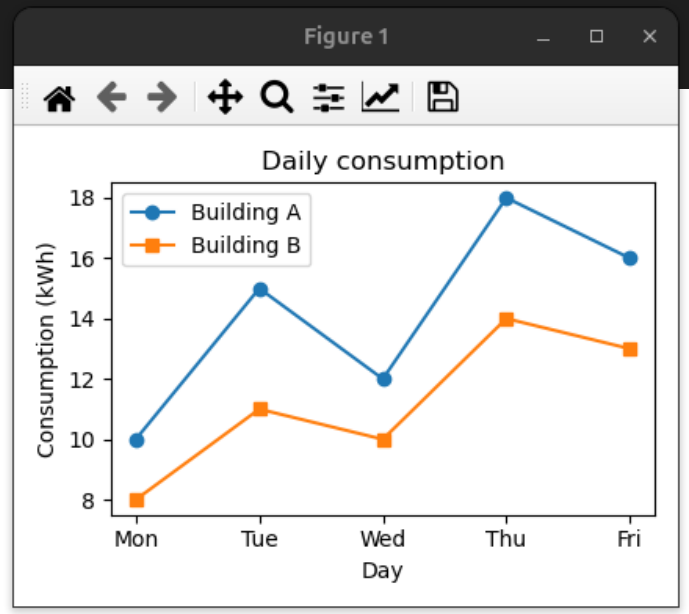
Line plots: trends and evolution

- Use `ax.plot(x, y)` to draw a line through pairs of corresponding observations in their given sequence order
- x contains the horizontal coordinates (for example day numbers); y contains the corresponding measurements
- Call `.plot()` again on the same ax to overlay another series rather than create another figure; Every call adds a new Line2D object to the existing Axes
- Its x and y sequences must have equal lengths
- Useful options:
  - `marker="o"` shows each observation with a circular marker; other options are `"^", "v", "x"`...
  - `linestyle="--"` changes the connecting line; `"-" solid, ":" dotted, "--" dashed, "-." dashdot...`
  - `linewidth=2` controls thickness
  - alpha=0.7 sets the trasparency
  - `label="Building A"` set a label for the line
- A line is appropriate when the x order represents a meaningful sequence, such as time or simulation steps

```python
fig, ax = plt.subplots()

ax.plot(days, building_a, marker="o", label="Building A")
ax.plot(days, building_b, marker="s", label="Building B")

ax.set_title("Daily consumption")
ax.set_xlabel("Day")
ax.set_ylabel("Consumption (kWh)")
ax.legend()
plt.show()
```



Line plots for simulation trajectories

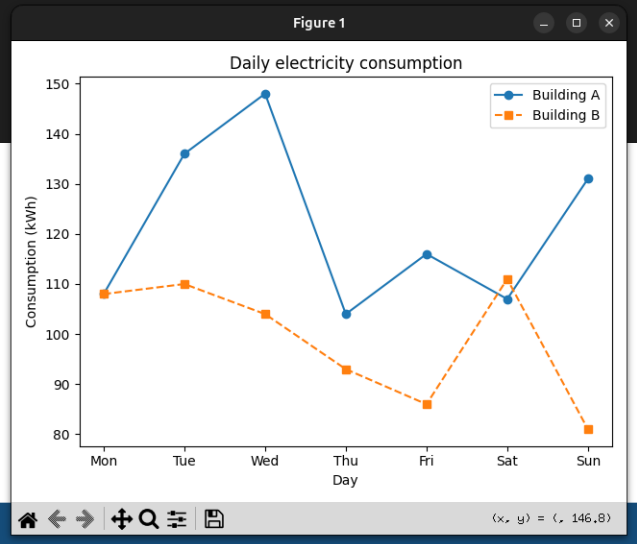
- Example: a simulation may store values in a Python list during a loop; Matplotlib can draw this list
- Use a second list for simulation steps or periods, and pass both lists to `ax.plot(steps, values)`
- If the horizontal coordinates are simply 0, 1, 2, …, `ax.plot(values)` is also accepted and uses positions by default
- Add `ax.set_xlabel("Period")` and `ax.set_ylabel(...)` so the meaning and units of the sequences are clear
- Generate the data first, then plot the stored history

```python
import random
import matplotlib.pyplot as plt

random.seed(1)  # reproducibility
days = range(1, 8)

# simulate consumption (kWh) of each building
building_a = [random.randint(100, 150) for day in days]
building_b = [random.randint(80, 120) for day in days]

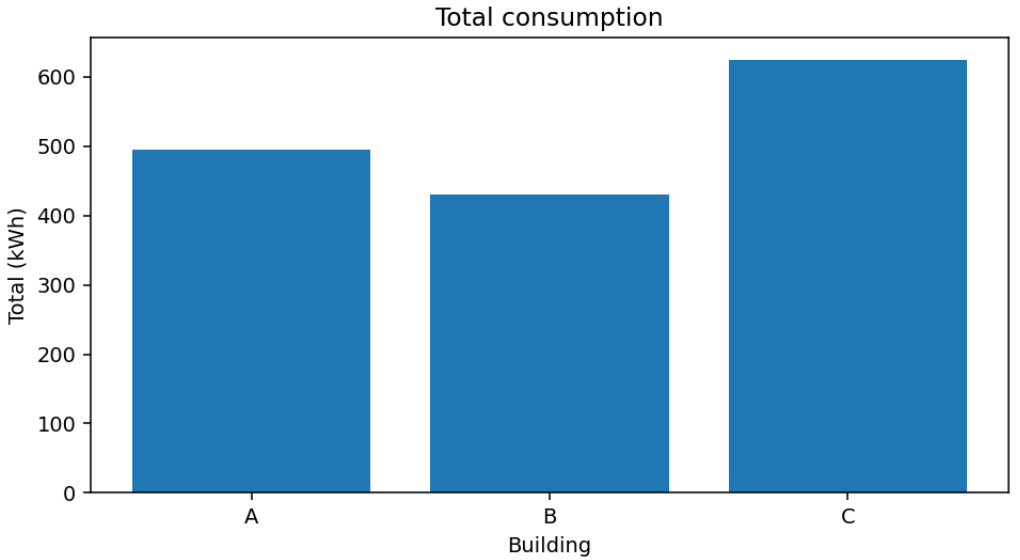
plt.plot(days, building_a, "o-", label="Building A")
plt.plot(days, building_b, "s--", label="Building B")
plt.title("Daily electricity consumption")
plt.xlabel("Day")
plt.ylabel("Consumption (kWh)")
plt.xticks(days, ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"])
plt.legend()
plt.tight_layout()
plt.show()
```



Bar plots: comparing categories

- Use `ax.bar(categories, heights)` when each position identifies a **category** and each height is its **numerical value**
- `categories` may be **strings** such as building names; `heights` (numerial) must correspond to them in the same order
- Optional arguments include `color="steelblue"`, `width=0.6` (bar width), and `alpha=0.8` (transparency)
- `ax.barh(categories, widths)` draws horizontal bars; use `.set_xlabel()` for the numerical measure in that case
- Typical use: compare category totals, counts or averages already calculated from the data

```
categories = ["A", "B", "C"]
totals = [495, 430, 625]
fig, ax = plt.subplots()
ax.bar(categories, totals)
ax.set_ylabel("Consumption (kWh)")
plt.show()
```



Bar plots: code and orientation

- `ax.bar(x, height)` draws one vertical bar for every pair of category label and numerical height
- `ax.barh(y, width)` uses the same data but places categories on the vertical axis and the values horizontally
- Both arguments can be Python lists, NumPy arrays or pandas Series; their lengths must match
- `color=` selects the fill; `width=` controls vertical bar width; `edgecolor=` draws bar outlines
- For vertical bars label the y-axis with the measured units; for horizontal bars label the x-axis

```
categories = ["A", "B", "C"]
totals = [495, 430, 625]
fig, ax = plt.subplots()
ax.bar(categories, totals)
ax.set_ylabel("Consumption (kWh)")
plt.show()
```

Replace ax.bar with ax.barh to get horizontal bars

Horizontal bars are useful for longer category names

From pandas records to a bar chart

- We return to pandas to prepare the data that Matplotlib will draw
- Example: the table contains electricity consumption records from three buildings; a building may appear on several days
- Goal: compare the total recorded consumption per building, with the largest total first
- Before plotting, we must calculate one total for each building

Example: input DataFrame df

Building	Consumption_kWh
A	120
B	80
C	95
A	150
C	125
B	110
C	90

Seven records, but only three categories

Aggregate and rank the categories

- First use `.groupby("Building")` to collect the records belonging to the same building
- Select `["Consumption_kWh"]` and apply `.sum()` to obtain one total per building
- The result is a Series: its index contains building labels and its values contain the totals
- Finally, `.sort_values(ascending=False)` orders the Series by total, from highest to lowest

Preparation before plotting

```
summary = df.groupby("Building")["Consumption_kWh"].sum()
ranked = summary.sort_values(ascending=False)
```

After .sum() — summary

Building	Total kWh
A	270
B	190
C	310

After sorting — ranked

Building	Total kWh
C	310
A	270
B	190

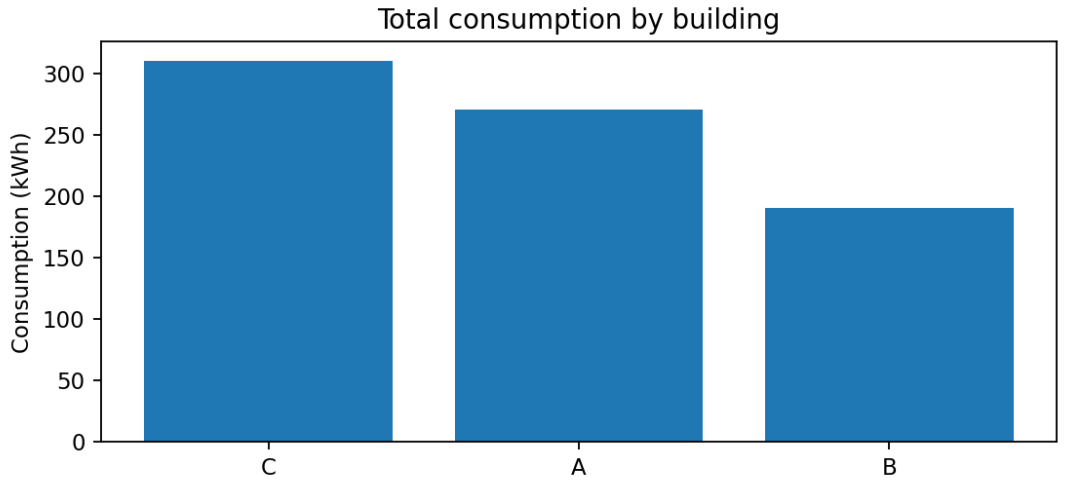
index values

Draw the ranked bar chart

- The data are now ready: `ranked.index` provides the ordered category labels, while `ranked.values` provides the matching numerical heights
- Pass both sequences to `ax.bar(x, height)`; Matplotlib draws one bar for each corresponding pair
- Use the Axes methods to label the plot, then display it with `plt.show()`

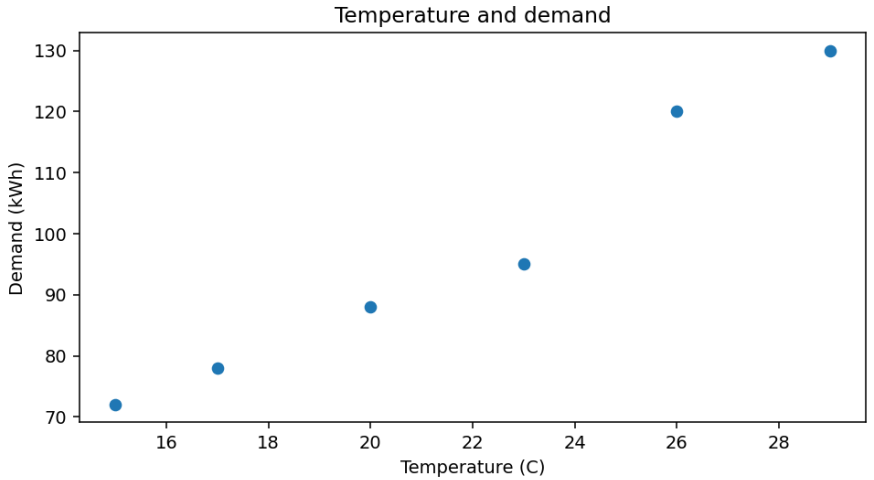
Plotting the prepared Series

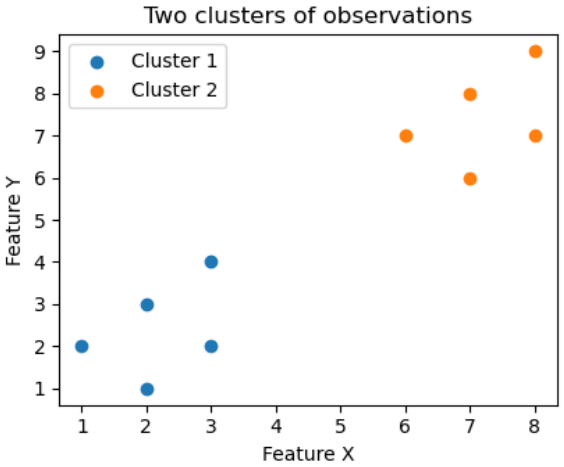
```
fig, ax = plt.subplots()
ax.bar(ranked.index, ranked.values)
ax.set_ylabel('Consumption (kWh)')
ax.set_title('Total consumption by building')
plt.show()
```



Scatter plots: two numerical variables

- Use `ax.scatter(x, y)` to draw a separate point for every pair of numerical coordinates
- x and y must contain matching observations and have the same length; the first x value pairs with the first y value
- Unlike `.plot()`, `.scatter()` does not connect successive observations with a line
- Optional arguments: `s=40` sets marker area in points squared; `color="tab:orange"` chooses a color; `alpha=0.6` reduces overlap
- Typical use: inspect the relationship between two measurements, such as temperature and electricity demand; clustering...

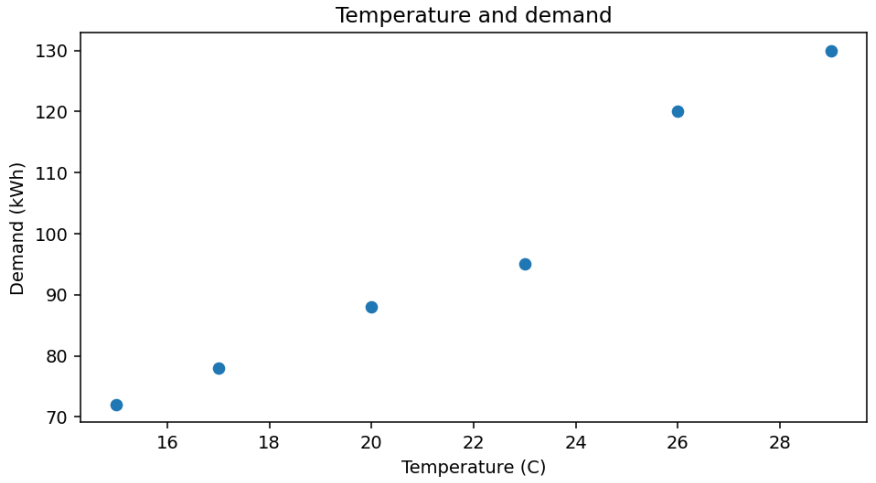




Scatter plot: sample code

- `ax.scatter(temperature, demand)` places one dot at each pair `(temperature[i], demand[i])`
- The two sequences must match position by position; they need not be sorted
- Optional `s=60` controls marker size; `marker="x"` changes the symbol; `alpha=0.7` helps when dots overlap
- Use `.set_xlabel()` and `.set_ylabel()` or the compact `.set(xlabel=..., ylabel=...)` syntax

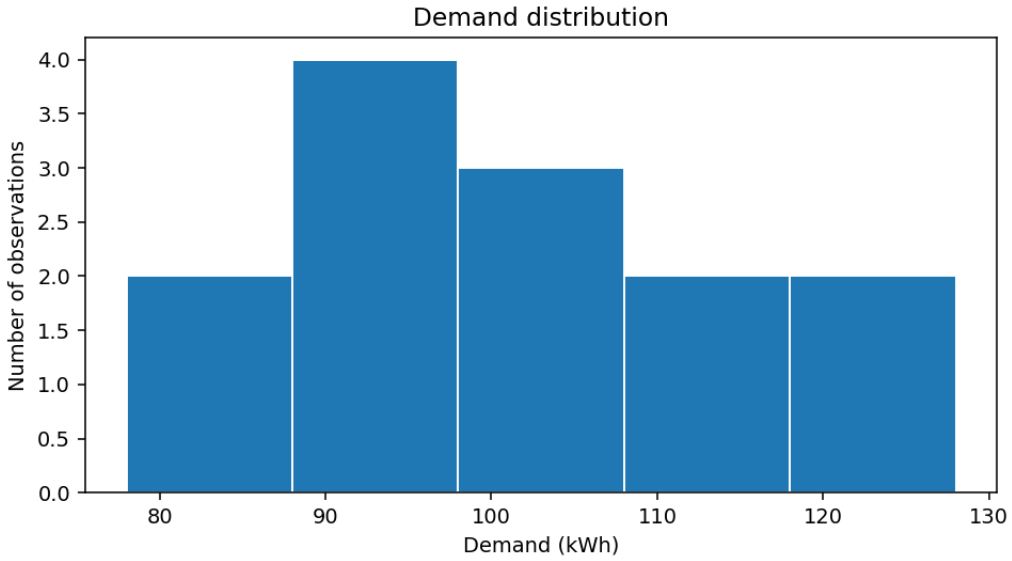
```
temperature = [15, 17, 20, 23, 26, 29]
demand = [72, 78, 88, 95, 120, 130]
fig, ax = plt.subplots()
ax.scatter(temperature, demand)
ax.set(xlabel="Temperature (C)", ylabel="Demand (kWh)")
plt.show()
```



Histograms: understanding the bins

- An histogram counts how many numerical observations fall inside each interval (`bin`)
- Example: 13 energy-demand measurements, from 78 to 128 kWh, using five bins

Demand interval (kWh)	Observed values	Count
[78, 88)	78, 83	2
[88, 98)	89, 92, 94, 97	4
[98, 108)	99, 103, 107	3
[108, 118)	111, 116	2
[118, 128]	121, 128	2



Each interval includes its left boundary; the final interval also includes 128

Histogram: code, bins and frequencies

- **Single input sequence**: unlike a line or scatter plot, `.hist()` takes one set of numeric observations; the frequencies are computed automatically
- `bins=5`: Divides the range 78–128 into five **equal-width intervals** of 10 kWh for this dataset
- How values are counted: 88 belongs to [88, 98), not [78, 88); only the last interval includes its right endpoint
- `edgecolor='white'`: Draws thin boundaries between the adjacent bars
- Other options: `color=` sets the fill; `density=True` replaces counts with normalized density

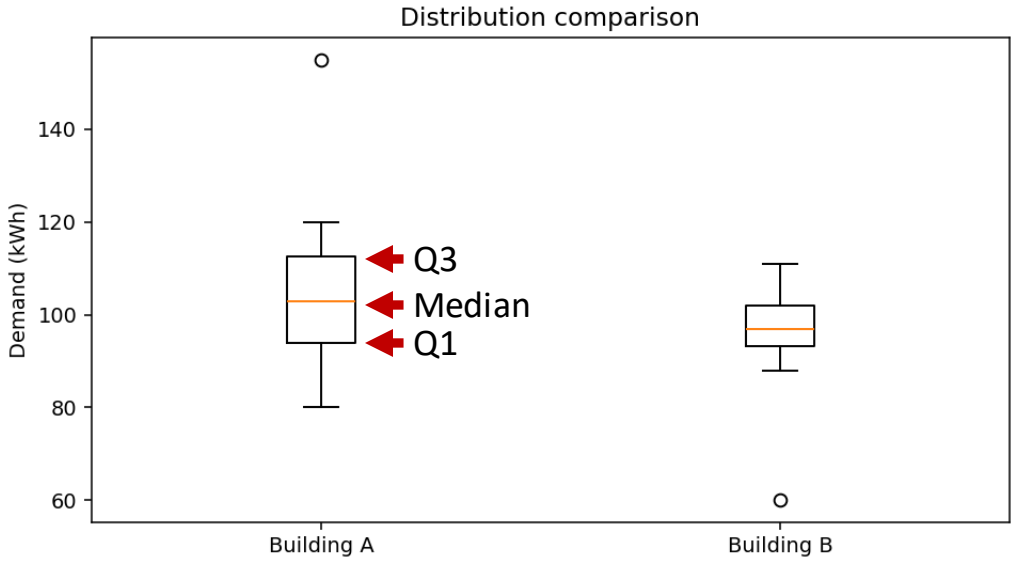
```
import matplotlib.pyplot as plt

observations = [78, 83, 89, 92, 94, 97,
                99, 103, 107, 111, 116, 121, 128]
fig, ax = plt.subplots()
ax.hist(observations, bins=5,
        edgecolor="white")
ax.set_title("Demand distribution")
ax.set_xlabel("Demand (kWh)")
ax.set_ylabel("Number of observations")
plt.show()
```

Bar heights, from left to right: 2, 4, 3, 2, 2

Box plots: a compact distribution summary

- `ax.boxplot(values)` creates a compact statistical summary of one numerical sequence
- The box extends from the first quartile (Q1) to the third quartile (Q3); the internal line marks the median
- Whiskers normally extend to the most extreme observations within 1.5 × IQR* of the box; more distant points may be shown separately as outliers
- Use `showmeans=True` to display the mean as well; `patch_artist=True` enables filled boxes that can be styled
- Typical use: compare medians, spread and possible outliers, especially across several groups



*Interquartile Range = $Q3 - Q1$

Box plot: compare groups

Pass a list of groups to `ax.boxplot([group_a, group_b])` to draw **one box for each numerical sequence**

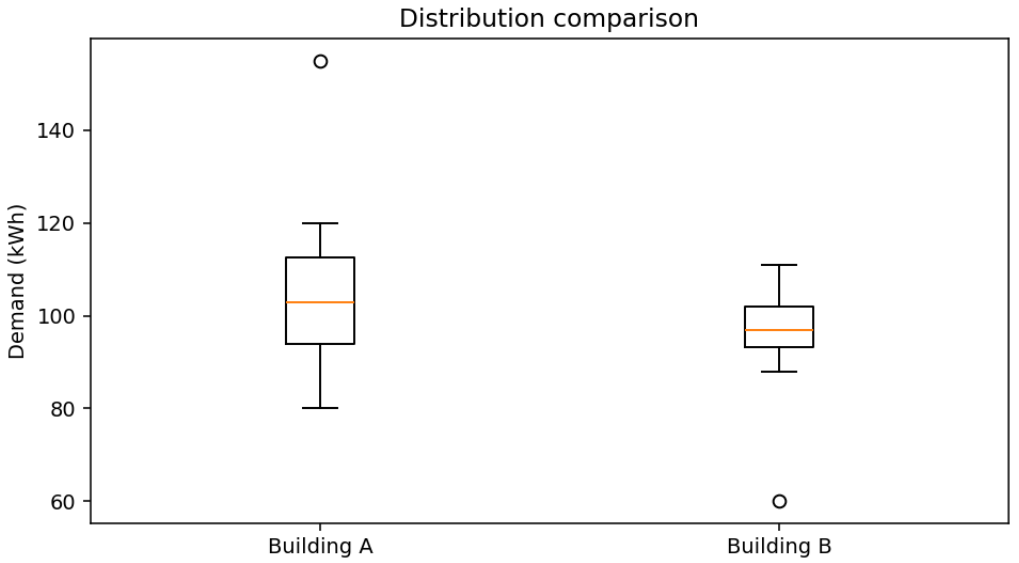
`tick_labels=["A", "B"]` supplies the names under the boxes (in recent Matplotlib versions)

Use `showmeans=True` if the average should also be displayed; `showfliers=False` hides outlier markers without removing data

`ax.set_ylabel("Demand (kWh)")` gives the units of the numerical axis

Each group **can contain a different number of observations**; the method calculates the box statistics separately

```
group_a = [80, 91, 95, 102, 110, 120, 155]
group_b = [60, 88, 95, 98, 105, 111]
fig, ax = plt.subplots()
ax.boxplot([group_a, group_b], tick_labels=["A", "B"])
ax.set_ylabel("Demand (kWh)")
plt.show()
```

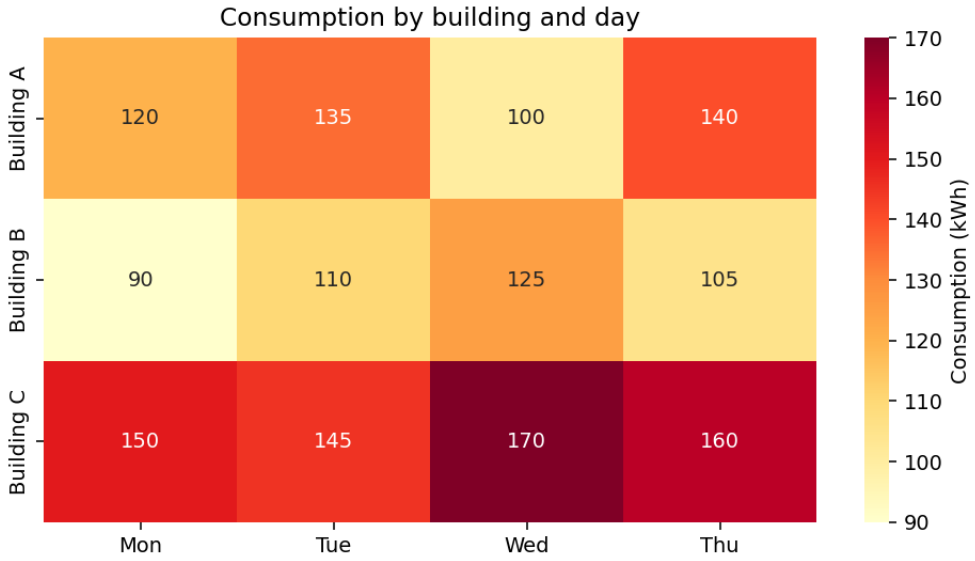


Histogram, box plot and scatter: different questions

- `.scatter(x, y)` receives **two paired numerical sequences** and creates one point per pair
- `.hist(values, bins=...)` **receives one numerical sequence** and counts values inside intervals
- `.boxplot(values)` **receives one numerical sequence**, or a list of group sequences, and summarizes quartiles
- Display options differ: scatter uses `s=` for point size; histogram uses `bins=`; box plot uses `tick_labels=` for groups
- Choose based on the output needed: relationship, frequency distribution or compact distribution comparison

Heatmaps: add a color dimension

- A **heatmap** displays a two-dimensional table using a colored cell for every row/column combination
- Input should be a **matrix or DataFrame**: rows and columns identify the cells; numeric entries determine the colors
- **Seaborn** is another library that offers `.heatmap(table, ...)` to avoid manually drawing and configuring every cell
- With a DataFrame, the row index and column labels are used as axis labels automatically
- Typical use: inspect values across two dimensions, such as building × weekday or asset × period; in general when you need to visualize a third dimension variating with respect to two parameters



Why introduce Seaborn?

- Seaborn is a plotting library built on Matplotlib: import seaborn as sns
- Its functions create common statistical graphics with fewer commands, especially from pandas DataFrames
- For a heatmap, `sns.heatmap(data, ax=ax, ...)` draws into an Axes that can still be configured using Matplotlib
- We retain the same workflow: create fig and ax, call a plotting function, add titles or labels, then `plt.show()`
- You only need the basic `sns.heatmap()` interface here; Seaborn contains many other optional chart types

From individual records to a heatmap

INPUT · long-format table (DataFrame)

Each row contains one recorded consumption for a building on a day. Multiple records may share the same Building–Day pair.

Building	Day	Consumption_kWh
A	Mon	82
A	Tue	88
A	Wed	95
B	Mon	105
B	Tue	108
B	Tue	**114**
B	Wed	109
C	Mon	91
C	Tue	98
C	Wed	104

Notice: building B has TWO observations on Tue: 108 and 114 kWh

They cannot occupy a single heatmap cell until we decide how to combine them.

```
import pandas as pd

df = pd.DataFrame({
  'Building': ['A','A','A','B','B',
               'B','B','C','C','C'],
  'Day': ['Mon','Tue','Wed','Mon',
          'Tue','Tue','Wed','Mon',
          'Tue',      'Wed'],
  'Consumption_kWh': [82, 88, 95,105,
                      108,114,109, 91,
                       98,    104]
})
```

Build the wide table with .pivot_table()

TRANSFORMATION · one cell per Building–Day pair

Set `index='Building'` to create rows, `columns='Day'` to create columns, and `values='Consumption_kWh'` to select the numbers to summarize

```
table = df.pivot_table(
    index='Building', columns='Day',
    values='Consumption_kWh',
    aggfunc='mean'
)
table = table[['Mon', 'Tue', 'Wed']]
print(table)
```

RESULT · wide DataFrame

Building	Mon	Tue	Wed
A	82	88	95
B	105	**111**	109
C	91	98	104

For B on Tue, `.pivot_table()` calculates (108 + 114) / 2 = 111. Every other pair has only one record, so its value remains unchanged.

From the wide table to colored cells

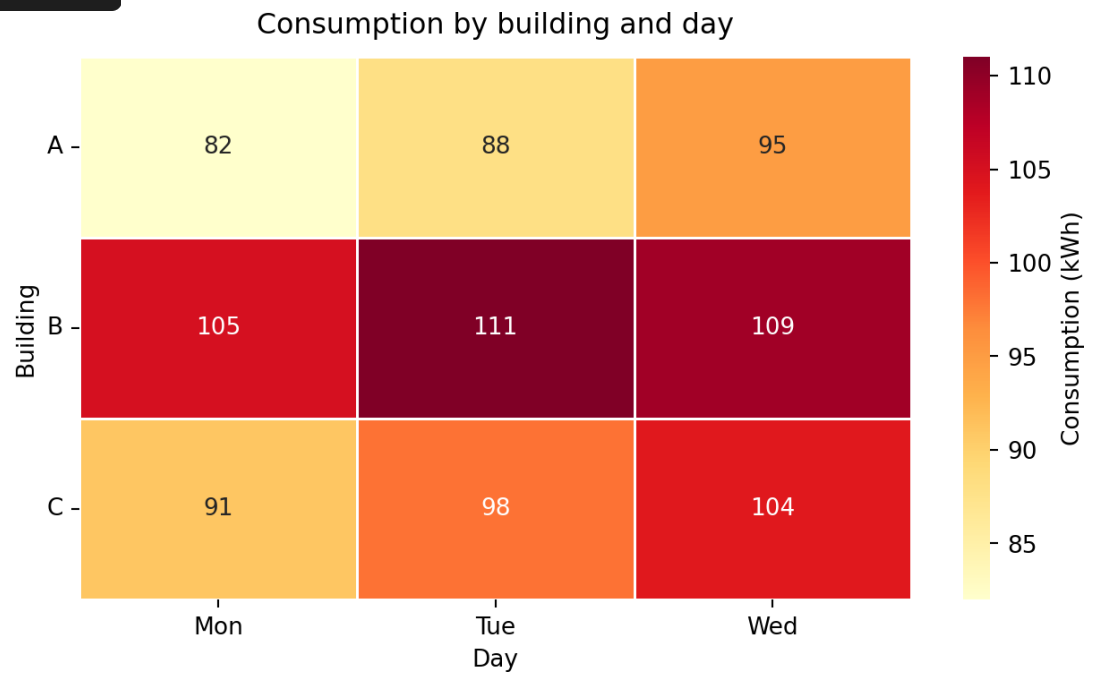
```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.heatmap(table, annot=True, fmt='.0f', cmap='YlOrRd', ax=ax)
plt.show()
```

The **index labels** of the DataFrame identify heatmap **rows**, **column labels** identify its **columns**, and each numeric value determines the color of one cell

WIDE DATAFRAME

Building	Mon	Tue	Wed
A	82	88	95
B	105	**111**	109
C	91	98	104

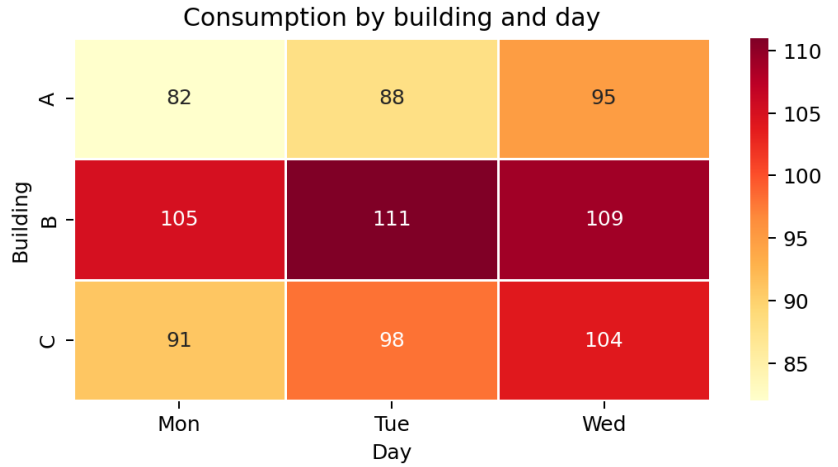


Heatmap using Seaborn

- `sns.heatmap(table, ax=ax)` accepts a two-dimensional numeric DataFrame or array and draws one colored cell per value; To interact with a pyplot figure, an axes is given as argument
- `annot=True` prints values inside cells; fmt="d" displays integers, while fmt=".1f" displays one decimal place
- `cmap="YlOrRd"` selects the color palette; `vmin=` and `vmax=` can fix the displayed color range for comparisons
- `cbar=True (default)` displays the color scale; linewidths=0.5 draws visible cell boundaries
- If a Matplotlib Axes is supplied through `ax=ax`, use `ax.set_title(...)` and `plt.show()` as usual

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.heatmap(table, annot=True, fmt='.0f', cmap='YlOrRd', ax=ax)
plt.show()
```



Make a chart readable

- The plot call draws data; the Axes methods add the information needed to read it
- `ax.set_title("...")` creates a title; `ax.set_xlabel("...")` and `ax.set_ylabel("...")` name the axes and units
- `ax.legend()` displays series names previously assigned with `label=` in `.plot()` or other plotting calls
- `ax.grid(True)` displays a reference grid; `alpha=` in a plotting call can reduce the visual weight of data marks
- Choose a chart type only after knowing whether the input consists of ordered values, categories, pairs or distributions

A well-labeled plot

- Create the Figure with `plt.subplots(figsize=(7, 4))`; figsize specifies width and height in inches
- Plot with `ax.plot(days, consumption, marker="o", label="A")` to draw the series and register its legend label
- Use `.set_title()`, `.set_xlabel()` and `.set_ylabel()` to explain the data and units
- Call `ax.legend()` to show the line label; call `ax.grid(True)` if reference lines help reading numerical values
- Finish with `plt.show()` for an interactive window or `fig.savefig(...)` for a file

```
fig, ax = plt.subplots(figsize=(7, 4))
ax.plot(days, consumption, marker="o", label="A")
ax.set_title("Daily energy consumption")
ax.set_xlabel("Day")
ax.set_ylabel("Energy (kWh)")
ax.legend()
ax.grid(True)
plt.show()
```

Save the figure as a report asset

- `fig.savefig("summary.png", dpi=150)` saves the entire Figure to the specified output file
- The filename extension selects the format: `.png` for a raster image, `.pdf` or `.svg` for vector output
- `dpi=150` sets pixel density for raster images; `bbox_inches="tight"` can include labels with less extra margin
- The output directory must already exist (you may create it with `.mkdir()`); use a simple filename in the current directory for basic scripts
- Call `.savefig()` after plotting and labeling; `.show()` displays the figure in an interactive window, and does not itself save a file

```python
from pathlib import Path
output_dir = Path(__file__).resolve().parent / "reports"
output_dir.mkdir(exist_ok=True)

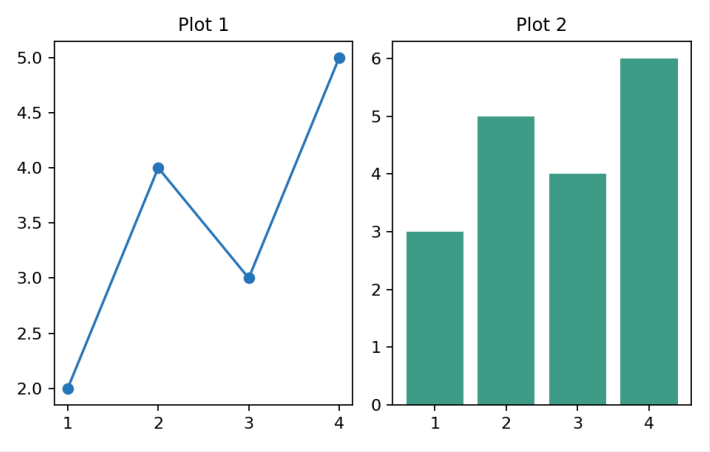
fig, ax = plt.subplots()
ax.bar(["A", "B"], [10, 15])
fig.savefig(output_dir / "summary.png", dpi=150)
```

Arranging multiple plots with plt.subplots()

`plt.subplots(nrows, ncols)` creates one Figure containing a grid of Axes (individual plots)

1 row × 2 columns

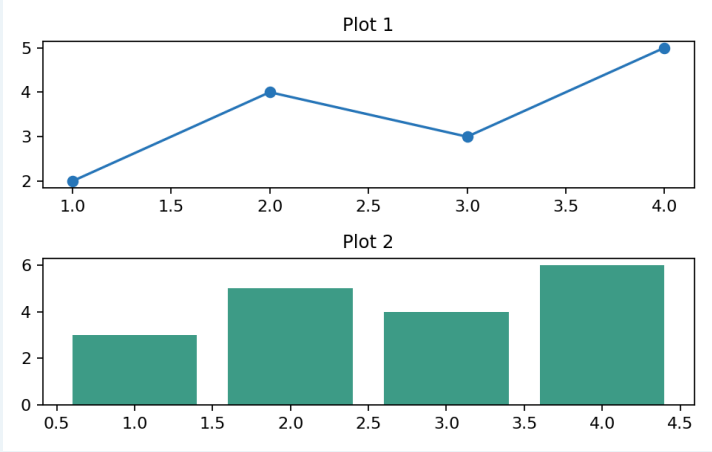
Side-by-side panels



plt.subplots(1, 2)

2 rows × 1 column

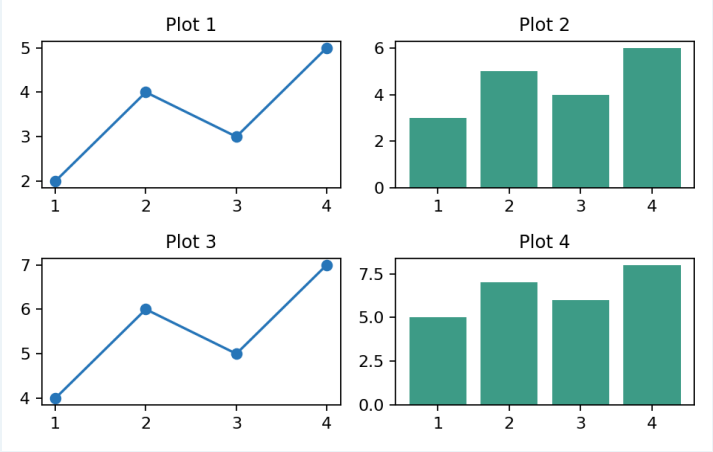
Vertically stacked panels



plt.subplots(2, 1)

2 rows × 2 columns

Four panels in a grid



plt.subplots(2, 2)

Rows are counted from top to bottom; columns are counted from left to right

How to access the Axes in a subplot grid

The object returned as ax (or axes) depends on the number and arrangement of panels

One row or one column: a 1D array of Axes

```
fig, axes = plt.subplots(1, 2)
axes[0].plot(x, y)
axes[1].bar(x, y)
```

For a 2 × 1 grid, the same indexing applies: axes[0] is the upper plot and axes[1] the lower plot

```
fig, axes = plt.subplots(2, 1)
axes[0].plot(x, y)
axes[1].bar(x, y)
```

Several rows and columns: a 2D array

```
fig, axes = plt.subplots(2, 2)
axes[0, 0].plot(x, y)
axes[0, 1].bar(x, y)
axes[1, 0].scatter(x, y)
axes[1, 1].hist(y)
```

The first index selects the row, the second the column. For example, axes[1, 0] is the bottom-left panel

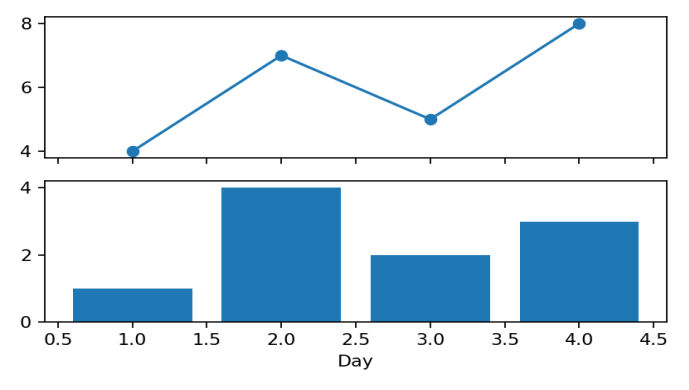
For these snippets, let x = [1, 2, 3, 4] and y = [2, 4, 3, 5]. With one panel, ax is a single Axes object

Sharing axes and controlling the layout

Optional parameters control alignment, figure dimensions and spacing; panels remain independently configurable

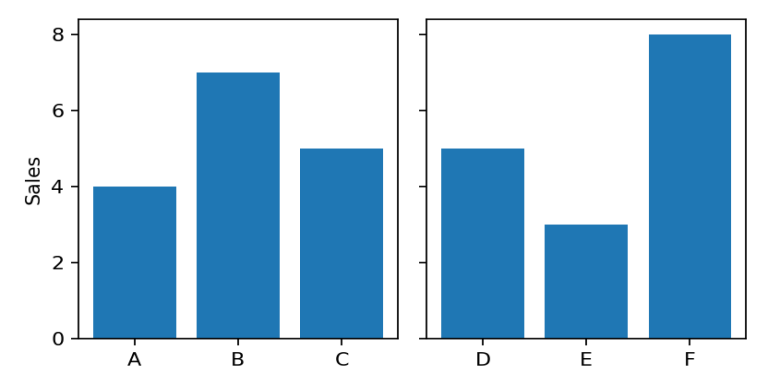
sharex=True: common horizontal scale

```
fig, axes = plt.subplots(2, 1, sharex=True)
```



sharey=True: common vertical scale

```
fig, axes = plt.subplots(1, 2, sharey=True)
```



Other useful controls

figsize=(9, 4) sets the Figure size in inches | fig.tight_layout() adjusts panel spacing

axes[0].set_title(...) or axes[1, 0].set_ylabel(...) configures an individual panel

Common visualization mistakes

- **Input mismatch**: `.plot(x, y)` or `.scatter(x, y)` needs **sequences of matching lengths** and corresponding observations
- **Wrong object**: `call .plot()` or `.set_title()` on `ax`; call `.savefig()` on `fig`
- **Missing legend**: `label=` in the plotting call does not display anything until `ax.legend()` is called
- **Wrong output**: `plt.show()` opens the figure, whereas `fig.savefig("file.png")` writes it to disk
- **Misleading parameters**: check `bins=` for histograms, category order for bars, and labels/units for every plot

Which chart fits a project?

Loan balance over time: `ax.plot(periods, balances, marker="o")`

Expenses per category: `ax.bar(categories, totals)` or `ax.barh(categories, totals)`

Distribution of measured values: `ax.hist(values, bins=10)` or `ax.boxplot(values)`

Relationship between two measurements: `ax.scatter(x_values, y_values, alpha=0.7)`

Two-dimensional numeric summary: `sns.heatmap(table, annot=True, fmt=".1f")`

What to remember

- The basic Matplotlib pattern is `fig, ax = plt.subplots()` → draw with `ax` → add labels → `plt.show()` or `fig.savefig(...)`
- `.plot(x, y)`, `.bar(categories, heights)`, `.scatter(x, y)`, `.hist(values, bins=n)` and `.boxplot(values)` cover the lesson
- The most useful optional arguments include `label=`, `marker=`, `color=`, `alpha=`, `bins=` and `figsize=` (for `plt.subplots`)
- All drawing calls use existing values from Python lists, NumPy arrays or pandas objects
- Use official API references when a plot needs a parameter or customization not shown in the examples

- `plt.subplots` creates Figure and Axes
- `ax.plot`, `ax.bar`, `ax.scatter` and `ax.hist` cover many fundamental tasks
- `ax.boxplot` and `sns.heatmap` illustrate distribution summaries and two-dimensional colored tables
- `fig.savefig` writes an image for the final project report
- `plt.show` opens an interactive window with the plot

Reliable reference documentation

Matplotlib Axes API (methods and parameters): https://matplotlib.org/stable/api/axes_api.html

Matplotlib `.plot()` and line options:
https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html

Matplotlib gallery (complete working examples): https://matplotlib.org/stable/gallery/index.html

Matplotlib `.subplots()` and Figure objects:
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html

Seaborn `.heatmap()` parameters: https://seaborn.pydata.org/generated/seaborn.heatmap.html