PROGRAMMING FOUNDATIONS FOR ANALYTICS

Pocket Reference: Topics 7 & 8

Dictionaries, Pandas and Data Visualisation on one sheet. Keep it open beside your Jupyter notebook — look up the syntax, copy the pattern, change the values.

- Topic 7 · Dictionaries & Pandas
- Topic 8 · Data Visualisation
- Common mistake — read twice

07A Dictionaries

key → value · get() · add & remove · items() · nested

7A.1 Key and value, kept together

A list finds things by **position**. A dictionary finds them by a **meaningful label**.

SYNTAX

```
my_dict = {key1: value1, key2: value2}
```

```
family = {'david': 1.72, 'mary': 1.58, 'mom': 1.69}
family['mary']    # 1.58 — straight to the value
```

TWO LISTS	ONE DICTIONARY
`heights[members.index('mary')]`	`family['mary']`
Breaks if either list is reordered	Label and value never separate

Keys are unique labels. **Values** can be anything — numbers, text, lists, even other dictionaries.

7A.2 Look up safely

A missing key with `[ ]` **crashes with a KeyError**. `.get()` hands back a safe default instead.

SYNTAX

```
d[key]                # key is SURE to exist
d.get(key, default)   # key MIGHT be missing
```

```
pet = {'kind': 'dog', 'owner': 'eric'}

pet['kind']                        # 'dog'
pet.get('breed', 'not recorded')   # 'not recorded'
pet.get('breed')                   # None (no default given)
'owner' in pet                     # True
```

`in` checks the **keys**, never the values.

7A.3 Add, change, remove, merge

```
contact = {'Name': 'John', 'Age': 35}

contact['Age'] = 36           # key exists → value updated
contact['Phone'] = 91234567   # key is new → pair added

del contact['Phone']          # remove a key
contact.pop('Age')            # remove AND return 36
contact.update({'Job': 'Chef'})  # merge another dict
```

YOU WANT	USE
Just delete it	`del d[key]`
Delete and keep the value	`d.pop(key)`

The same square-bracket line both **adds** and **updates** — it depends on whether the key is already there.

7A.4 Loop and aggregate

SYNTAX

```
for key, value in my_dict.items():
    # use key and value
```

```
meals = {'Breakfast': 3.80, 'Lunch': 4.60, 'Dinner': 8.60}

for meal, cost in meals.items():
    print(f"{meal}: ${cost:.2f}")

sum(meals.values())   # 17.0
len(meals)            # 3
```

YOU NEED	USE
Both label and number	`.items()`
Only the numbers	`.values()`
Only the labels	`.keys()`
How many pairs	`len(d)`

It is `len(d)`, not `d.len()` — `len` is a function, not a method.

7A.5 Nested dictionaries

A value can itself be a dictionary. The first key picks the **record**, the second picks a **detail** inside it.

```
students = {
    'Willie': {'PRGA': 83, 'DVST': 90},
    'Walter': {'PRGA': 72, 'DVST': 88},
}

students['Willie']['DVST']   # 90
```

This is how real data is shaped — a customer with an address, an order with items, a stall with a rent and a menu.

Read the brackets left to right: **which record**, then **which field**.

7A.6 List or dictionary?

	LIST	DICTIONARY
Written with	`[ ]`	`{ }`
Found by	Position (0, 1, 2…)	Key ('mary')
Order matters?	Yes	Not for lookups
Missing item gives	`IndexError`	`KeyError`

Use a **list** for a sequence of the same kind of thing. Use a **dictionary** when each value needs a name.

07B Pandas

Series · DataFrame · read_csv · loc/iloc · filter · groupby

7B.1 Import, and the Series

```
import pandas as pd   # run this FIRST, every notebook
```

A **Series** is one column: values *plus* labels (the index).

```
prices = pd.Series({'Laksa': 5.50, 'Kopi': 1.40})

prices['Laksa']   # 5.5 — by LABEL, not position
prices.mean()     # 3.45
```

LIST	SERIES
`[5.50, 1.40]`	`Laksa 5.50 · Kopi 1.40`
5.50 means… what?	Each number carries its meaning

Forget the import and every Pandas line fails with a **NameError**.

7B.2 Build a DataFrame

A **DataFrame** is a table: rows are records, columns are fields. Each dictionary key becomes a column.

```
data = {'Item':  ['Laksa', 'Kopi'],
        'Price': [5.50, 1.40],
        'Qty':   [48, 120]}

df = pd.DataFrame(data)
df.shape    # (2, 3) → 2 rows, 3 columns
```

- [ ] **Rule 1:** every list must be the **same length**, or you get a ValueError.
- [ ] **Rule 2:** keep numbers numeric. `'5.50'` in quotes is text and breaks the maths.

7B.3 Load a real CSV

```
df = pd.read_csv('hawker_sales.csv')
df.head()
```

CSV = Comma-Separated Values, a plain text file. The **first row becomes the column names**.

FileNotFoundError? The file must sit in the same folder as the notebook, and the name must match exactly — including `.csv`.

```
import os
os.getcwd()    # which folder am I in?
os.listdir()   # what files can Python see?
```

7B.4 Look before you analyse

Run these five on **every** new dataset, before any analysis.

COMMAND	SHOWS
`df.head()`	First 5 rows
`df.tail()`	Last 5 rows
`df.shape`	(rows, columns)
`df.info()`	Column names, types, missing values
`df.describe()`	count, mean, min, max, quartiles

info() = structure. describe() = numbers. A Price column showing `object` in info() is stored as **text** — fix it before doing maths. An odd max in describe() is a possible typo or outlier.

7B.5 Selecting columns

CODE	GIVES BACK
`df['Price']`	A **Series** — one column
`df[['Price']]`	A **DataFrame** — 1 column, still a table
`df[['Item','Price']]`	A DataFrame — several columns

Single brackets → Series. Double brackets → table. Use double when you want to keep the table shape.

Column names are **case-sensitive**. `'Price'` is not `'price'` — a typo gives a KeyError.

7B.6 loc vs iloc

Memory hook: **loc = Label** · **iloc = Integer location**. The comma reads as **rows, columns**.

```
df.loc[0, 'Item']    # by LABEL
df.iloc[0, 0]        # by POSITION

df.loc[0:2, ['Item','Qty']]   # rows 0,1,2 — end INCLUDED
df.iloc[0:3, 0:2]             # rows 0,1,2 — end excluded
```

The slice rules differ. `loc` **includes** the end label. `iloc` **excludes** it, like every other Python slice.

7B.7 Filtering rows

A **mask** is a True/False test run on the whole column at once. Pandas keeps only the True rows.

```
mask = df['Qty'] > 60   # one True/False per row
df[mask]

df[df['Qty'] > 60]      # the usual one-liner
```

```
# Food AND over $5
df[(df['Category'] == 'Food') & (df['Price'] > 5)]

# Drinks OR high quantity
df[(df['Category'] == 'Drinks') | (df['Qty'] > 100)]
```

Use `&` and `|`, never Python's `and` / `or`. Wrap **each** condition in its own brackets, or the order of operations goes wrong.

7B.8 Add a calculated column

Assign to a new name and Pandas fills every row at once. This is called **vectorised** — no loop needed.

```
df['Revenue'] = df['Price'] * df['Qty']

df[['Item', 'Revenue']]
# Laksa  264.0
# Kopi   168.0
```

The new column did not exist in the CSV — you created it. One line replaces the whole for-loop you would have written in Topic 5.

7B.9 Sort, summarise, group

```
df.sort_values('Revenue')                    # low → high (default)
df.sort_values('Revenue', ascending=False)   # high → low
df.sort_values('Revenue', ascending=False).head(3)
# top 3
```

```
df['Revenue'].sum()    # total takings
df['Price'].mean()     # typical price
df['Price'].max()      # dearest
df['Price'].min()      # cheapest
```

groupby splits the rows into groups, then summarises each group:

```
df.groupby('Category')['Revenue'].sum()
```

Read it as three steps: **group by** this column → **pick** that column → **apply** a summary. Perfect for totals by category, town or month.

08 Data Visualisation with Matplotlib

plot · chart furniture · legends · 5 chart types · groupby charts

8.1 The three-line recipe

```
import matplotlib.pyplot as plt
%matplotlib inline    # show charts in the notebook
import pandas as pd
```

Every chart follows the same shape: **draw**, **decorate**, **show**.

```
plt.plot(month_number, interest_paid)   # draw
plt.title('Interest Paid Each Month')   # decorate
plt.show()                              # show
```

Call `plt.plot()` twice before `plt.show()` and both lines land on the **same** chart.

> Everything between one `plt.show()` and the next belongs to one figure. `plt.show()` closes it — anything after starts a new chart.

8.2 Markers, colours, line styles

```
plt.plot(x, y, color='blue', linestyle='-',
         marker='o', markersize=10)
```

SETTING	COMMON VALUES
`marker`	`'.' 'o' '+' 'x' '*' 's' '^' 'v'`
`linestyle`	`'-' '--' ':' '-.' 'None'`
`color`	`'blue' 'red' 'green' 'black' …`
`markersize`	A number, e.g. 10

Use full colour names rather than single letters — `color='red'` reads better than `'r'`.

> Want dots only, no joining line? Set `linestyle='None'` and keep the marker.

8.3 Chart furniture

Add these **after** the plot line and **before** `plt.show()`.

LINE	SETS
`plt.figure(figsize=(10,6))`	Size in inches — put it **first**
`plt.style.use('ggplot')`	Overall look
`plt.title('…', fontsize=20)`	Chart title
`plt.xlabel / plt.ylabel`	Axis names
`plt.xticks(fontsize=10)`	Tick label size
`plt.xlim(left=0, right=11)`	Axis range
`plt.grid(axis='y', alpha=0.3)`	Grid lines

`plt.style.available` lists every style. `plt.rcdefaults()` resets everything.

> `plt.figure(figsize=…)` must come **before** the plot call, or it opens an empty extra figure.

8.4 Legends

Two steps: **label each line**, then **call the legend**.

```
plt.plot(month, interest, label='Interest Paid')
plt.plot(month, principal, label='Principal Paid')
plt.legend(loc='center left')
plt.show()
```

LOC VALUE	PUTS IT
`'best'`	Wherever it blocks least (default)
`'upper right'`	Named corners and edges
`(1.05, 0)`	Outside the plot — (0,0) is bottom-left

> No `label=` on the plot lines means `plt.legend()` shows nothing.

8.5 The five chart types

CHART	CODE	ANSWERS
Line	`plt.plot(x, y)`	How does it change over time?
Scatter	`plt.scatter(x, y)`	Are these two things related?
Bar	`plt.bar(x, y) / plt.barh(x, y)`	Which category is biggest?
Box	`plt.boxplot(values)`	How spread out is it? Any outliers?
Pie	`plt.pie(sizes, labels=names)`	What share of the whole?

```
plt.bar(['a','b','c','d'], [2, 4, 6, 2], align='center')
plt.barh(states, frequency, align='center')   # horizontal — good for long names

plt.boxplot([x1_vals, x2_vals], tick_labels=['plot1', 'plot2'])

plt.pie(frequency, labels=grades, autopct='%1.1f%%',
        explode=(0.1, 0, 0, 0), shadow=True)
```

barh reads better when the category names are long — no rotated, overlapping labels. **Pie charts** only work for a handful of slices; with many categories a bar chart is easier to compare.

8.6 Data labels with annotate()

`zip()` pairs each x with its y, then you annotate one point per round.

```
for x, y in zip(x_vals, y_vals):
    plt.annotate(f"{y:.2f}",          # the text
                 (x, y),              # the point
                 textcoords="offset points",
                 xytext=(0, 10),      # nudge up
10pt
                 ha='center')
```

`xytext=(0, 10)`	Above the point
`xytext=(0, -15)`	Below the point
`xytext=(20, 0)`	To the right — use for barh

> One loop labels **one** line. Two lines on a chart need **two** loops — a common reason half the labels go missing.

8.7 From groupby to chart

This is the pattern that ties Topics 7 and 8 together: **aggregate first, then plot the summary.**

```
profits = df.groupby('Department')['Profit'].sum()

plt.barh(profits.index, profits, align='center')
plt.title('Profits by Department')
plt.xlabel('Profits'); plt.ylabel('Department')
plt.show()
```

The Series **index** holds the category names and the Series **values** hold the numbers — exactly the two things a bar chart needs.

```
# count instead of sum, then take the top 10
freq = df.groupby('State').size() \
         .sort_values(ascending=False).head(10)
```

`.sum()` totals a column · `.count()` counts values · `.size()` counts rows per group.

09 When to use what

Pick the right tool in one glance

Which data structure?

A sequence of the same thing	`list`
Each value needs a name	`dict`
One labelled column	`pd.Series`
Rows and columns of records	`pd.DataFrame`

Once you have more than a few hundred rows, a DataFrame beats a loop every time.

Which chart answers my question?

Trend over time	`line`
Compare categories	`bar / barh`
Relationship between two numbers	`scatter`
Spread and outliers	`box`
Share of a total (few slices)	`pie`

Always turn the chart into a sentence. "Furniture earns the most profit" beats a number with no words.

Picking data out of a DataFrame

One column, as a Series	`df['Price']`
Several columns, as a table	`df[['Item','Price']]`
By row and column name	`df.loc[0,'Item']`
By row and column number	`df.iloc[0,0]`
Rows matching a test	`df[df['Qty'] > 60]`

Before you ask for help — run this checklist

- [ ] **Did I import it?** `import pandas as pd` and `import matplotlib.pyplot as plt`, in that notebook, this session.
- [ ] **Is the CSV in the same folder?** Check with `os.listdir()` and match the spelling exactly.
- [ ] **Did I spell the column right?** Names are case-sensitive — run `df.columns` to see them.
- [ ] **Did I use & and | in my filter?** Python's `and` / `or` do not work on columns.
- [ ] **Did I bracket each condition?** `(df['A']==1) & (df['B']>2)`.
- [ ] **Did I save the result?** `df.sort_values(...)` returns a new table — assign it or chain it.
- [ ] **Did I call plt.show() last?** Everything decorating a chart must come before it.
- [ ] **Is my number stored as text?** Check `df.info()` — `object` where you expected a number.

Check yourself

1 · What is returned?

```
d = {'kind': 'dog'}
d.get('breed', 'unknown')
```

'unknown' — `d['breed']` would raise a KeyError instead.

2 · What does this total?

```
m = {'a': 2.0, 'b': 3.0}
sum(m.values())
```

5.0 — `sum(m)` alone would try to add the keys and fail.

3 · What type comes back?

```
df['Price']
df[['Price']]
```

Series, then **DataFrame**. Double brackets keep the table shape.

4 · How many rows?

```
df.loc[0:2]
df.iloc[0:2]
```

3 rows, then **2 rows**. `loc` includes the end label; `iloc` does not.

5 · Why does this fail?

```
df[df['A']==1 and df['B']>2]
```

ValueError — use `&` and bracket each test: `df[(df['A']==1) & (df['B']>2)]`.

6 · Why is the legend empty?

```
plt.plot(x, y)
plt.legend()
```

No **`label=`** on the plot line, so there is nothing to list.
