DS105W – Data for Data Science

Week 07

✍ Mini Project 2 Launch: Focus on Methodology, Then Set Up the Tech

Dr Jon Cardoso-Silva
LSE Data Science Institute

📅 05 Mar 2026

Hour 1: Methodology clinic (NB01)

Let’s talk about the new challenge ahead: ✍️ **Mini-Project 2**.

1. Discuss what makes a strong approach to your mini-project
2. Draft your own plan
3. Give a peer some feedback
4. Submit a single Mentimeter entry (Scope, Confidence, Risk)

Your next challenge

How does travelling between Inner London and Outer London change when it happens during peak hours versus off-peak hours?

What's already set in the question:

- We focus on the journey between two points: Inner London and Outer London.
- We care about the time of day: peak and off-peak.
- We want to compare how journey times differ between these two parts of the day.

Decisions you must make

How does travelling between Inner London and Outer London change when it happens during peak hours versus off-peak hours?

What you decide:

- Which time period will you study?
  (Next week? Next month? Only weekdays? Only weekends?)
- How will you define **Inner London**?
- Which **Outer London** destinations will you choose, and why?
- How will you define **peak** and **off-peak** travel?
- What aspect will you measure?
  Time in minutes? Walking distance in meters? Cost in pounds? Number of transfers?
- How will you compare values numerically?

Groupby recap

In this section, we will:

1. Look at how to Group, Apply, and Combine data (with clear visuals)
2. Explore the shape of a nested JSON
3. See how `json_normalize()` turns that structure into a flat table

What happens when you use `groupby()` with a dataframe

Imagine you have this dataframe:

```
df = pd.DataFrame({
    "route_id": ["R1", "R2", "R3", "R4", "R5"],
    "time_band": ["peak", "off-peak", "peak", "off-peak", "pe
    "duration_min": [51, 43, 56, 40, 49],
})

df
```

If you group data by `time_band`, you produce this:

```
df.groupby("time_band")
```

BEFORE GROUPING

route_id	time_band	duration_min
R1	peak	51
R2	off-peak	43
R3	peak	56
R4	off-peak	40
R5	peak	49

AFTER GROUPING

group label	rows inside
peak	R1, R3, R5
off-peak	R2, R4

The same rows are now split by `time_band`.

Now I need to do something with each group ☞

What `groupby().agg()` does

We can summarize each group using built-in aggregation functions. `agg()` lets us do this for specific columns:

```
df.groupby("time_band").agg(
    mean_duration=("duration_min", "mean"),
    median_duration=("duration_min", "median"),
    n_routes=("route_id", "count"),
)
```

BEFORE summarizing

group	values
peak	51, 56, 49
off-peak	43, 40

AFTER summarizing

time_band	mean_duration	median_duration	n_routes
peak	52.0	51.0	3
off-peak	41.5	41.5	2

What `groupby().apply()` lets you do

Use `pd.Series.apply()` when you want to decide exactly how to calculate something for each group. `groupby().apply()` lets you run your own function on every group:

```
def spread_ratio(g):
    return g["duration_min"].max() / g["duration_min"].min()

df.groupby("time_band").apply(spread_ratio)
```

BEFORE summarizing

group	values
peak	51, 56, 49
off-peak	43, 40

AFTER groupby().apply()

time_band	spread_ratio
peak	1.14
off-peak	1.08

Using `pd.DataFrame.apply(..., axis=1)`

Apply a custom function to each row by using `pd.DataFrame.apply(..., axis=1)`. Here, you can combine columns into something new for every row. The `axis=1` argument means: “run the function once per row”.

```python
def route_signature(row):
    return f"{row['from_zone']} -> {row['to_zone']} ({row['time_band']})"

df["route_signature"] = df.apply(route_signature, axis=1)
df[["from_zone", "to_zone", "time_band", "route_signature"]].head()
```

BEFORE

from_zone	to_zone	time_band
Inner	Outer	peak
Inner	Outer	off-peak

AFTER row-wise apply

from_zone	to_zone	time_band	route_signature
Inner	Outer	peak	`Inner -> Outer (peak)`
Inner	Outer	off-peak	`Inner -> Outer (off-peak)`

Comparing groups with `groupby().apply(...)`

Here, we compare each route pair (`from_zone`, `to_zone`) by passing each group to our function:

```
grouped_signature = (
  df
  .groupby(["from_zone", "to_zone"])
  .apply(compare_peak_vs_off_peak, axis=1)
)

grouped_signature.head()
```

BEFORE

from_ zone	to_ zone	time_ band	time
Inner	Outer	peak	51
Inner	Outer	off-peak	43
Inner	Outer	peak	56
Inner	Outer	off-peak	40
…	…	…	…

AFTER grouped apply

from_ zone	to_ zone	peak_ mean	off_peak_ mean	delta_ min
Inner	Outer	52.3	41.5	10.8
Inner	Suburban	47.0	39.2	7.8

Note: The values here are made up so you can see that in the end, you produce just one row per group.

Nested JSON data

Here's something new. Most APIs send nested data structures, but to analyse the data, you usually want a flat table.

NESTED JSON

```
{
  "requestContext": {
    "from": "WC2A 2AE",
    "to": "SE1 9JA"
  },
  "journeys": [
    {"duration": 42, "fare": {"totalCost": 2.90}},
    {"duration": 37, "fare": {"totalCost": 2.40}}
  ]
}
```

TABULAR TARGET

duration	fare_totalCost	from	to
42	2.90	WC2A 2AE	SE1 9JA
37	2.40	WC2A 2AE	SE1 9JA
39	2.60	WC2A 2AE	SW11 1AA
45	3.10	WC2A 2AE	E15 2EE
…	…	…	…

You probably faced this in ✍️ **Mini Project 1** when unpacking nested dictionaries into columns.

Manual route first: what we would do without `json_normalize()`

This route works, but it is fragile when nested structure changes.

```
journeys_df = pd.DataFrame(journey_response["journeys"])
journeys_df["from"] = journey_response["requestContext"]["from"]
journeys_df["to"] = journey_response["requestContext"]["to"]
journeys_df.head()
```

BEFORE manual extraction

response object
`{"requestContext": {...}, "journeys": [...]}`

AFTER manual extraction

duration	fare	from	to
42	`{...}`	WC2A 2AE	SE1 9JA
37	`{...}`	WC2A 2AE	SE1 9JA
...	...	...	...

We still have some nested structures in the `fare` column.

Pandas has a function that does this in one consistent pattern: `json_normalize()`

Use `json_normalize()` when your main task is flattening nested response structures into analysis-ready tables.

You could use it alone: `pd.json_normalize(journey_response)` or you could make use of its parameters:

- `record_path`: the path to the nested list to expand
- `meta`: the path to the parent object to keep
- `sep`: the separator to use for the column names

Parameter effect 1: `record_path`

```
# option A: keep journeys nested in one row
pd.json_normalize(journey_response)

# option B: one row per journey
pd.json_normalize(journey_response, record_path="journeys")
```

`record_path` choice	What happens	Typical row unit
not set	nested journeys stay in one cell	one response object
`"journeys"`	journeys are expanded to multiple rows	one journey

Visual comparison: `record_path` output

WITHOUT `record_path`

requestContext.from	requestContext.to	journeys
WC2A 2AE	SE1 9JA	`[{...}, {...}, ...]`

All the data is here, but most of it is packed inside one cell.

WITH `record_path="journeys"`

duration	fare.totalCost
42	2.90
37	2.40
39	2.60
45	3.10
...	...

We lost the data outside the `journeys` key but at least we have one row per journey!

Parameter effect 2: `meta`

Once rows are expanded from `journeys`, we often need route context copied onto each row. `meta` does that.

```
pd.json_normalize(
    journey_response,
    record_path="journeys",
    meta=[["requestContext", "from"], ["requestContext", "to"]],
)
```

`meta` choice	Effect on flattened rows
not set	journey rows lose route context columns
include origin/destination paths	each row keeps route context and journey values

Visual comparison: `meta` context preservation

WITH `record_path`
BUT WITHOUT `meta`

duration	fare.totalCost
42	2.90
37	2.40
39	2.60
45	3.10
...	...

WITH `record_path`
AND WITH `meta`

duration	fare.totalCost	requestContext.from	requestContext.to
42	2.90	WC2A 2AE	SE1 9JA
37	2.40	WC2A 2AE	SE1 9JA
39	2.60	WC2A 2AE	SW11 1AA
45	3.10	WC2A 2AE	E15 2EE
...	...	...	...

Both pieces of information is preserved!! Useful for a `groupby()` operation later if I have multiple routes in the same DataFrame.

Parameter effect 3: `sep`

This does not change the data values. It changes column naming grammar for readability and downstream coding.

```
df = pd.json_normalize(
    journey_response,
    record_path="journeys",
    sep="__",
)
```

`sep` value	Example output column
default `"."`	`fare.totalCost`
`"__"`	`fare__totalCost`

What's Next?

- **In the lab**, your class teacher will walk you through calling the TfL API with a working notebook, then guide you through flattening the nested JSON response with `json_normalize()`.
- **By the end of lab**, you should have your ✍️ **Mini Project 2** repository cloned, at least one real API call in your own `NB01`, and the raw JSON saved to `data/raw/`.
- **Before W08 Lecture**, have a look at the ONS Postcode Directory data dictionary in the ✍️ **Mini Project 2** spec. W08 is about databases and merging data from multiple sources, and you will want to know what columns are available before you get there.