Step 1: Importing the Required Libraries

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[193] ✓ 0.0s Python

M↓ ## Load the CSV into df (file name: DataOnly) ···

The analysis begins by importing the key Python libraries needed to load, analyse, and visualise the dataset.

Step 2: Loading and Inspecting the Dataset

```python
#b Read the CSV file into a DataFrame called df
df = pd.read_csv("DataOnly.csv")
```

[248] ✓ 0.0s Python

```python
# Quick check: show the first 5 rows
df.head()
```

[249] ✓ 0.0s Python

...

	yyyy	mm	tmax degC	tmin degC	af days	rain mm	sun hours
0	1941	1	NaN	NaN	NaN	74.7	NaN
1	1941	2	NaN	NaN	NaN	69.1	NaN
2	1941	3	NaN	NaN	NaN	76.2	NaN
3	1941	4	NaN	NaN	NaN	33.7	NaN
4	1941	5	NaN	NaN	NaN	51.3	NaN

I loaded the dataset and performed an initial inspection to confirm structure and identify missing values.

Step 3: Preparing Measurement Columns for Cleaning

Convert measurement columns to text (strings)

Some measurement columns can contain extra symbols like *, #, blanks, or ---. Converting them to strings first (and trimming spaces) makes the cleaning and flag checks reliable and prevents type errors.

```python
# These columns may contain markers like '*', '#', '---'
# Convert to string so we can detect and remove symbols safely
value_cols = ["tmax degC", "tmin degC", "af days", "rain mm", "sun hours"]

for c in value_cols:
    if c in df.columns:
        df[c] = df[c].astype("string").str.strip()
```

[256] ✓ 0.0s Python

I converted measurement columns to text to safely identify special symbols and avoid data type errors during cleaning.

Create flag columns BEFORE removing symbols

```python
# Estimated flags: True if the cell contains '*'

df["tmax degC_estimated"] = df["tmax degC"].str.contains(r"\*", na=False)
df["tmin degC_estimated"] = df["tmin degC"].str.contains(r"\*", na=False)
df["af days _estimated"]  = df["af days"].str.contains(r"\*", na=False)
df["rain mm _estimated"]  = df["rain mm"].str.contains(r"\*", na=False)
df["sun hours _estimated"]= df["sun hours"].str.contains(r"\*", na=False)

# 2) Sunshine source flag: True if sunshine value contains '#'
df["sun hours_kipp_zonen"] = df["sun hours"].str.contains(r"#", na=False)
```

[257] ✓ 0.0s Python

```python
df.head()
```

[258] ✓ 0.0s Python

	yyyy	mm	tmax degC	tmin degC	af days	rain mm	sun hours	tmax degC_estimated	tmin degC_estimated	af days _estimated	rain mm _estimated	sun hours _estimated	hours_kip
0	1941	1	<NA>	<NA>	<NA>	74.7	<NA>	False	False	False	False	False	
1	1941	2	<NA>	<NA>	<NA>	69.1	<NA>	False	False	False	False	False	
2	1941	3	<NA>	<NA>	<NA>	76.2	<NA>	False	False	False	False	False	

Step 4: Creating Flags for Estimated and Source-Specific Values

Convert --- and blanks to missing values, remove * and #, then convert to numbers

```python
def clean_numeric_series(s: pd.Series) -> pd.Series:

    s = s.astype("string").str.strip()

    # Convert known missing markers to NA
    s = s.replace(["---", ""], pd.NA)

    # Remove special markers (we already saved them in flag columns)
    s = s.str.replace("*", "", regex=False)
    s = s.str.replace("#", "", regex=False)

    # Convert to numeric; anything invalid becomes NaN
    return pd.to_numeric(s, errors="coerce")

# Apply cleaning to each measurement column
for c in value_cols:
    df[c] = clean_numeric_series(df[c])
```

[259] ✓ 0.0s Python

I created flag columns before cleaning to retain information about estimated values and changes in data collection methods.

Step 5: Creating a Proper Monthly Date Variable

Create a proper monthly date

```python
# Ensure year and month are numeric
df["yyyy"] = pd.to_numeric(df["yyyy"], errors="coerce").astype("Int64")
df["mm"]   = pd.to_numeric(df["mm"], errors="coerce").astype("Int64")
```

[261] ✓ 0.0s Python

```python
# Create a date column as the first day of each month
df["date"] = pd.to_datetime(
    dict(year=df["yyyy"].astype(int), month=df["mm"].astype(int), day=1),
    errors="coerce"
)
```

[262] ✓ 0.0s Python

+ Code + Markdown

I created a standard monthly date variable to support accurate time-series analysis and forecasting.

Step 6: Creating Derived Metrics for Analysis and Reporting

Create derived metrics for clearer reporting

```python
# Average temperature
df["tmean degC"] = df[["tmax degC", "tmin degC"]].mean(axis=1)

# Temperature range
df["trange degC"] = df["tmax degC"] - df["tmin degC"]

# year and month fields
df["year"] = df.index.year
df["month"] = df.index.month
```

[265] ✓ 0.0s Python

+ Code + Markdown

I created derived metrics to summarise temperature behaviour and support clearer trend and seasonal analysis.

Step 7: Data Quality Checks for Missing Values

Data quality checks (missing values + marker flags)

```python
# % missing in each key metric
missing_pct = df[["tmax degC","tmin degC","tmean degC","af days","rain mm","sun hours"]].isna().mean() * 100
missing_pct.sort_values(ascending=False)
```

[269] ✓ 0.0s Python

```
af days       18.823529
tmax degC      1.764706
sun hours      1.372549
tmin degC      1.176471
tmean degC     1.176471
rain mm        0.000000
dtype: float64
```

+ Code + Markdown

I quantified missing values to assess data quality and identify variables that required cautious interpretation.

Step 8: Interpretation of Sunshine Data Source Flags

Interpretation

290 rows of sunshine values contain # and 730 rows do not contain #.

```python
COLORS = {
    "green": "#2E7D32",
    "gold_orange": "#F9A825",
    "brown": "#6D4C41",
    "milk": "#FFF8E1",
    "blue": "#1E88E5",
    "pink": "#D81B60"
}
```

[272] ✓ 0.0s Python

I identified changes in sunshine measurement methods and used consistent visual styling to clearly communicate these differences.

Step 9: Analysing Seasonal Temperature Patterns

Shows the typical seasonal cycle by averaging daily mean temperature across all years for each month.

```
import matplotlib.pyplot as plt

month_names = ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"]

seasonal_temp = df.groupby("month")["tmean degC"].mean()

plt.figure(figsize=(10,4))
bars = plt.bar(
    seasonal_temp.index,
    seasonal_temp.values,
    color=COLORS["gold_orange"],
    edgecolor=COLORS["brown"]
```

[273]

I averaged temperatures by month across all years to clearly show the typical seasonal cycle.

Step 10: Analysing Long-Term Annual Temperature Trends

Annual temperature trend over time

```
# Yearly average of daily mean temperature (tmean). This removes seasonality and highlights long-term change.
```

[274] ✓ 0.0s Python

```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Create annual averages (one value per year)mj ú,kmji9
annual_temp = df.groupby("year")["tmean degC"].mean().sort_index()

# Quick check: confirm the last year in the series (should be 2025 if present)
print("Last year in annual_temp:", int(annual_temp.index.max()))
print(annual_temp.tail(5))

# Step 2: Smooth short-term ups/downs (optional but useful for storytelling)
rolling_5y = annual_temp.rolling(5, min_periods=1).mean()

# Step 3: Fit a linear trend safely (polyfit fails if NaN/inf exist)
```

[275]

I averaged temperatures by year and applied a rolling mean to clearly highlight long-term trends.

Step 11: Analysing Long-Term Rainfall Trends

Annual Total Rainfall Trend (Sum of Daily Rainfall)

```python
import numpy as np
import matplotlib.pyplot as plt

# Sums daily rainfall to annual totals to show long-term wet/dry variability and highlight unusually wet or dry years.
annual_rain = df.groupby("year")["rain mm"].sum().sort_index()

# Rolling mean to smooth year-to-year swings
rolling_5y = annual_rain.rolling(5, min_periods=1).mean()

# Long-term average (baseline)
long_run_mean = annual_rain.mean()

# Identify wettest and driest years
wettest_year = int(annual_rain.idxmax())
driest_year  = int(annual_rain.idxmin())

plt.figure(figsize=(14, 4))

# Annual totals
plt.plot(
```

I aggregated rainfall annually and applied a rolling average to understand long-term wet and dry trends.

Step 12: Analysing the Relationship Between Temperature and Sunshine

Temperature vs Sunshine Relationship

```python
# Scatter plot of daily mean temperature against daily sunshine hours to show whether warmer days tend to be sunnier.
plot_data = df[["tmean degC", "sun hours"]].dropna()

x = plot_data["tmean degC"].to_numpy(dtype=float)
y = plot_data["sun hours"].to_numpy(dtype=float)

# Correlation (Pearson)
r = np.corrcoef(x, y)[0, 1]

plt.figure(figsize=(8, 5))

# Scatter
plt.scatter(x, y, color=COLORS["pink"], alpha=0.35, s=18, edgecolors="none", label="Daily values")

# Trend line (simple linear fit)
m, b = np.polyfit(x, y, 1)
x_line = np.linspace(x.min(), x.max(), 200)
y_line = m * x_line + b
plt.plot(x_line, y_line, color=COLORS["brown"], linewidth=2, label="Linear fit")
```

I compared temperature and sunshine using correlation and a trend line to confirm that warmer periods tend to be sunnier.

Step 13: Creating a Simple Seasonal Forecast

```python
forecast_horizon = 12

# Future monthly dates (month-start)
future_index = pd.date_range(
    start=ts.index.max() + pd.offsets.MonthBegin(1),
    periods=forecast_horizon,
    freq="MS"
)

# Seasonal naive: repeat the last 12 observed monthly values
last_12 = ts.iloc[-12:]
forecast = pd.Series(last_12.values, index=future_index, name="tmean degC_forecast")

forecast
```

[284] ✓ 0.0s Python

I used a simple seasonal approach by repeating the most recent year of data to produce a transparent short-term forecast.