The first thing to do in any data analysis task is to plot the data. Graphs enable many features of the data to be visualised, including patterns, unusual observations, changes over time, and relationships between variables. The features that are seen in plots of the data must then be incorporated, as much as possible, into the forecasting methods to be used. Just as the type of data determines what forecasting method to use, it also determines what graphs are appropriate. But before we produce graphs, we need to set up our time series in Python.

2.1 `DataFrame` objects

A time series is a list of numbers (the observations) with information about when those numbers were recorded (the index). Sometimes, there are additional columns with more details about the observations, known as covariates or exogenous variables. This two-dimensional data structure is represented as a `DataFrame`. In Python, tables are often created as Pandas `DataFrame` objects and more recently as Polars `DataFrame` objects. If the `DataFrame` is too large for memory, parallel, distributed versions like Dask, Spark, and Ray can be used. We will focus on the Pandas `DataFrame` in this book.

The index variable

Suppose you have annual observations for the last few years:

	Year	Observation
0	2015	123
1	2016	39
2	2017	78
3	2018	52
4	2019	110

We can store the information using an `DataFrame` object using the `pd.DataFrame()` function:

```
x = [123, 39, 78, 52, 110]
yr = list(range(2015, 2020))
df = pd.DataFrame({"Year": yr, "Observation": x})
df.set_index("Year", inplace=True)
```

We have set the time series `index` to be the `Year` column, which associates the observations with the time of recording (`Year`). Make sure the time column (in this case index) is of the datetime data type.

For observations that are more frequent than once per year, we need to use a time class function on the index. For example, suppose we have a monthly dataset `z`:

	Month	Observation
0	2019-01-01	50
1	2019-02-01	23
2	2019-03-01	34
3	2019-04-01	30
4	2019-05-01	25

We can convert the time column using the following code:

```
z["Month"] = z["Month"].dt.strftime("%Y %b")
z.set_index("Month", inplace=True)
z
```

Month	Observation
2019 Jan	50
2019 Feb	23
2019 Mar	34
2019 Apr	30
2019 May	25

First, the `Month` column is being converted from date to month with `dt.strftime()`. We then set the index of the dataframe as the `Month` column.

Other time class functions can be used depending on the frequency of the observations.

Frequency	Function
Annual	`.dt.to_period('Y')` or `.dt.strftime("%Y")`
Quarterly	`.dt.to_period('Q')`
Monthly	`.dt.to_period('M)`
Weekly	`.dt.to_period('W')` or `.dt.strftime("%Y-%W")`
Daily	`.dt.strftime("%Y-%m-%d")`
Sub-daily	`.dt.strftime("%Y-%m-%d %H:%M:%S")`

Note that `.dt.to_period('W')` and `.dt.strftime("%Y-%W")` assume that the week starts on Monday. If you want the week to start on Sunday, use `.dt.to_period('W-SUN')` or `.dt.strftime("%Y-%W%U")`.

The key variables

A `DataFrame` also allows multiple time series to be stored in a single object. Suppose you are interested in a dataset containing the fastest running times for women's and men's track races at the Olympics, from 100m to 10000m:

```
olympic_running = pd.read_csv("../data/olympic_running_unparsed.csv")
olympic_running.head(10)
```

	Year	Length	Sex	Time
0	1896	100	men	12.0
1	1900	100	men	11.0
2	1904	100	men	11.0
3	1908	100	men	10.8
4	1912	100	men	10.8
5	1916	100	men	NaN
6	1920	100	men	10.8
7	1924	100	men	10.6
8	1928	100	men	10.8
9	1932	100	men	10.3

This is a `DataFrame` object, which contains 312 rows and 4 columns. Alongside this, the data is recorded every four years, and there are 14 separate time series in this dataframe. A preview of the first 10 observations is also shown, in which we can see a missing value occurs in 1916. This is because the Olympics were not held during World War I.

The 14 time series in this object are uniquely identified by the keys: the `Length` and `Sex` variables. The `unique()` function can be used to show the categories of each variable or even combinations of variables:

```
print(olympic_running["Sex"].unique())
```

```
['men' 'women']
```

Working with timeseries dataframes

We can use `pandas` functions such as `assign()`, `query()`, `filter()` and `agg()` to work with `DataFrame` objects. To illustrate these, we will use the `PBS`, containing sales data on pharmaceutical products in Australia.

```
pbs = pd.read_csv("../data/PBS_unparsed.csv")
pbs["Month"] = pd.to_datetime(pbs["Month"])
```

```
pbs.head()
```

	Month	Concession	Type	ATC1	ATC1_desc	ATC2	ATC2_desc	Scripts	Cost
0	1991-07-01	Concessional	Co-payments	A	Alimentary tract and metabolism	A01	STOMATOLOGICAL PREPARATIONS	18228	67877.0
1	1991-08-01	Concessional	Co-payments	A	Alimentary tract and metabolism	A01	STOMATOLOGICAL PREPARATIONS	15327	57011.0
2	1991-09-01	Concessional	Co-payments	A	Alimentary tract and metabolism	A01	STOMATOLOGICAL PREPARATIONS	14775	55020.0
3	1991-10-01	Concessional	Co-payments	A	Alimentary tract and metabolism	A01	STOMATOLOGICAL PREPARATIONS	15380	57222.0
4	1991-11-01	Concessional	Co-payments	A	Alimentary tract and metabolism	A01	STOMATOLOGICAL PREPARATIONS	14371	52120.0

This contains monthly data on Medicare Australia prescription data from July 1991 to June 2008. These are classified according to various concession types, and Anatomical Therapeutic Chemical (ATC) indexes. For this example, we are interested in the `Cost` time series (total cost of scripts in Australian dollars).

We can use the `query()` function to extract the A10 scripts:

```
a10 = pbs.query('ATC2 == "A10"')
```

This allows rows of the dataframe to be selected. Next we can simplify the resulting object by selecting the columns we will need in subsequent analysis.

```
a10 = a10.filter(["Month", "Concession", "Type", "Cost"])
a10
```

	Month	Concession	Type	Cost
1524	1991-07-01	Concessional	Co-payments	2092878.0
1525	1991-08-01	Concessional	Co-payments	1795733.0
1526	1991-09-01	Concessional	Co-payments	1777231.0
1527	1991-10-01	Concessional	Co-payments	1848507.0
1528	1991-11-01	Concessional	Co-payments	1686458.0
...	...	...	...	...
52339	2008-02-01	General	Safety net	530709.0
52340	2008-03-01	General	Safety net	51773.0
52341	2008-04-01	General	Safety net	36289.0
52342	2008-05-01	General	Safety net	101233.0
52343	2008-06-01	General	Safety net	193179.0

816 rows × 4 columns

The `filter()` method of Pandas allows us to select particular columns, while `query()` method allows us to keep particular rows.

Another useful method is `agg()` which allows us to combine data across keys. For example, we may wish to compute total cost per month regardless of the `Concession` or `Type` keys.

```
total_cost_df = a10.groupby("Month", as_index=False).agg({"Cost": "sum"})
total_cost_df.rename(columns={"Cost": "TotalC"}, inplace=True)
total_cost_df
```

	Month	TotalC
0	1991-07-01	3526591.0
1	1991-08-01	3180891.0
2	1991-09-01	3252221.0
3	1991-10-01	3611003.0
4	1991-11-01	3565869.0
...	...	...
199	2008-02-01	21654285.0
200	2008-03-01	18264945.0
201	2008-04-01	23107677.0
202	2008-05-01	22912510.0
203	2008-06-01	19431740.0

204 rows × 2 columns

The new column `TotalC` represents the sum of all `Cost` values for each month.

We can create new variables using the `assign()` function. Here we change the units from dollars to millions of dollars:

```
total_cost_df = total_cost_df.assign(
    Cost = round(total_cost_df["TotalC"] / 1e6, 2)
)
total_cost_df
```

	Month	TotalC	Cost
0	1991-07-01	3526591.0	3.53
1	1991-08-01	3180891.0	3.18
2	1991-09-01	3252221.0	3.25
3	1991-10-01	3611003.0	3.61
4	1991-11-01	3565869.0	3.57
...	...	...	...
199	2008-02-01	21654285.0	21.65
200	2008-03-01	18264945.0	18.26
201	2008-04-01	23107677.0	23.11
202	2008-05-01	22912510.0	22.91
203	2008-06-01	19431740.0	19.43

204 rows × 3 columns

Next, we will save the resulting dataframe to a CSV file for use in examples later in this chapter. This allows us to easily reload the data without having to recompute it. If you are running this code make sure to specify the correct path to save the file in your own working directory.

```
total_cost_df.to_csv("../data/total_cost_df.csv", index=False)
```

Read a csv file

The data used in this book is stored as pandas `DataFrame` objects. In real world scenarios, data is sometimes stored in databases, Spreadsheets, csv or tsv files. So often, the first step in creating a DataFrame is to read in the data.

For example, suppose we have a quarterly data of the prison population in Australia stored in a comma separated values file (.csv) file. This data set provides information on the size of the prison population in Australia, disaggregated by state, gender, legal status and indigenous status. (Here, ATSI stands for Aboriginal or Torres Strait Islander.)

	Date	State	Gender	Legal	Indigenous	Count
0	2005-03-01	ACT	Female	Remanded	ATSI	0
1	2005-03-01	ACT	Female	Remanded	Non-ATSI	2
2	2005-03-01	ACT	Female	Sentenced	ATSI	0
3	2005-03-01	ACT	Female	Sentenced	Non-ATSI	5
4	2005-03-01	ACT	Male	Remanded	ATSI	7
5	2005-03-01	ACT	Male	Remanded	Non-ATSI	58
6	2005-03-01	ACT	Male	Sentenced	ATSI	5
7	2005-03-01	ACT	Male	Sentenced	Non-ATSI	101
8	2005-03-01	NSW	Female	Remanded	ATSI	51
9	2005-03-01	NSW	Female	Remanded	Non-ATSI	131

In Python we can read (or load) the file as a `DataFrame` by calling the `pd.read_csv()` method. The resulting object is a `DataFrame` where the first column is the index. The original file stored the dates as individual days, although the data is actually quarterly, so we need to convert the `Date` variable to quarters.

```
prison = pd.read_csv("../data/prison_population.csv")
prison["Date"] = pd.to_datetime(prison["Date"])
prison["Quarter"] = prison["Date"].dt.to_period("Q")
prison = prison.drop(columns=["Date"])
prison.set_index("Quarter", inplace=True)
prison.sort_values(by=["State", "Gender", "Legal", "Indigenous"])
prison.head()
```

Quarter	State	Gender	Legal	Indigenous	Count
2005Q1	ACT	Female	Remanded	ATSI	0
2005Q1	ACT	Female	Remanded	Non-ATSI	2
2005Q1	ACT	Female	Sentenced	ATSI	0
2005Q1	ACT	Female	Sentenced	Non-ATSI	5
2005Q1	ACT	Male	Remanded	ATSI	7

This dataframe contains 64 unique time series corresponding to the unique combinations of the 8 states, 2 genders, 2 legal statuses and 2 indigenous statuses. Each of these series is 48 observations in length, from 2005 Q1 to 2016 Q4.

For a time series dataframe to be valid, each unique combination of keys must have a unique index.

The seasonal period

Seasonalities are particularly important for time series analysis. Some graphics and some models rely on the repeated seasonal period of the data. The seasonal period is the number of observations before a seasonal pattern repeats. In most cases, this will be automatically detected using the time index variable.

Some common periods for different time intervals are shown in the table below:

Data	Minute	Hour	Day	Week	Year
Quarters					4
Months					12
Weeks					52
Days				7	365.25
Hours			24	168	8766
Minutes		60	1440	10080	525960
Seconds	60	3600	86400	604800	31557600

For quarterly, monthly and weekly data, there is only one seasonal period — the number of observations within each year. Let be noted, that actually there are not 52 weeks in a year, but 365.25/7 = 52.18 on average, allowing for a leap year every fourth year. Approximating seasonal periods to integers can be useful as many seasonal terms in models only support integer seasonal periods.

If the data is observed more than once per week, then there is often more than one seasonal pattern in the data. For example, data with daily observations might have weekly (period =7) or annual (period =365.25) seasonal patterns. Similarly, data that are observed every minute might have hourly (period =60), daily (period =24\times60=1440), weekly (period =24\times60\times7=10080) and annual seasonality (period =24\times60\times365.25=525960).

More complicated (and unusual) seasonal patterns can be specified using the `seasonal_decompose()` function in the `statsmodels` package.

2.2 Time plots

For time series data, the obvious graph to start with is a time plot. That is, the observations are plotted against the time of observation, with consecutive observations joined by straight lines. Figure 2.1 shows the weekly economy passenger load on Ansett airlines between Australia's two largest cities (Melbourne and Sydney).

```
ansett = pd.read_csv("../data/ansett.csv")
ansett["ds"] = pd.to_datetime(ansett["ds"])
melsyd_economy = ansett.query(
        'Airports == "MEL-SYD" & Class == "Economy"'
    ).copy()
melsyd_economy["y"] = melsyd_economy["y"] / 1000

plot_series(df=melsyd_economy,
            id_col="Airports",
            time_col="ds",
            target_col="y",
            ylabel="Passengers ('000)",
            xlabel="Week [1W]",
            title="Ansett airlines economy class: Melbourne-Sydney"
           )
```

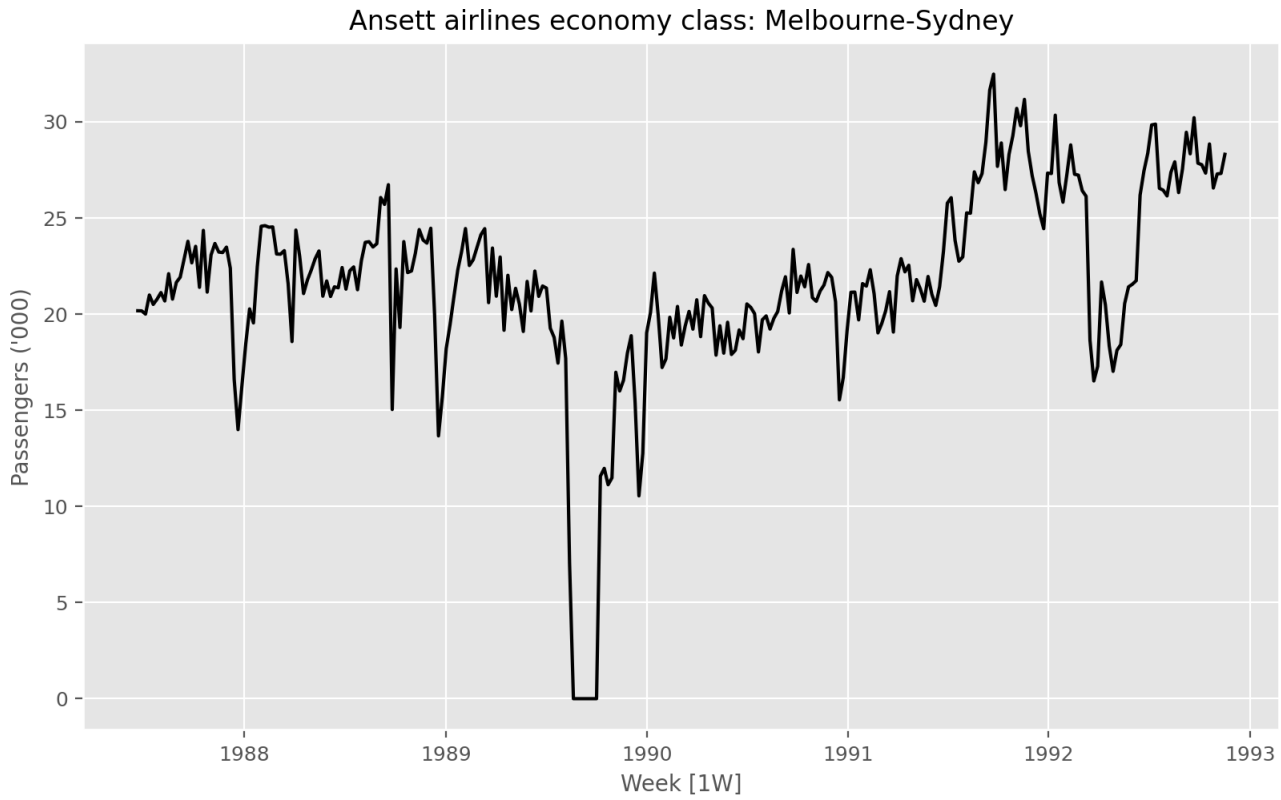


Figure 2.1: Weekly economy passenger load on Ansett Airlines.

For plotting series, we will often use the `plot_series()` function from the `utilsforecast` library. The function needs these parameters:

- `df`: Input dataframe (aka pandas DataFrame)
- `id_col`: Series identifier column (aka unique id)
- `time_col`: Timestamp (aka index)
- `target_col`: Target variable column (aka measurement)

For further details, see the utilsforecast documentation.

This plot reveals some interesting features.

- There was a period in 1989 when no passengers were carried — this was due to an industrial dispute.
- There was a period of reduced load in 1992. This was due to a trial in which some economy class seats were replaced by business class seats.
- A large increase in passenger load occurred in the second half of 1991.
- There are some large dips in load around the start of each year. These are due to holiday effects.
- There is a long-term fluctuation in the level of the series which increases during 1987, decreases in 1989, and increases again through 1990 and 1991.

Any model will need to take all these features into account in order to effectively forecast the passenger load into the future.

A simpler time series is shown in Figure 2.2, using the `total_cost_df` data saved earlier. In this case, given that we aggregated the data before, we have just one unique time series and we don't have a `unique_id` column so we need to create one.

```
total_cost_df["unique_id"] = "total_cost" # Create a unique id column
plot_series(total_cost_df,
            id_col="unique_id",
            time_col="Month",
            target_col="Cost",
            xlabel="Month [1M]",
            ylabel="$ (millions)",
            title="Australian antidiabetic drug sales")
```

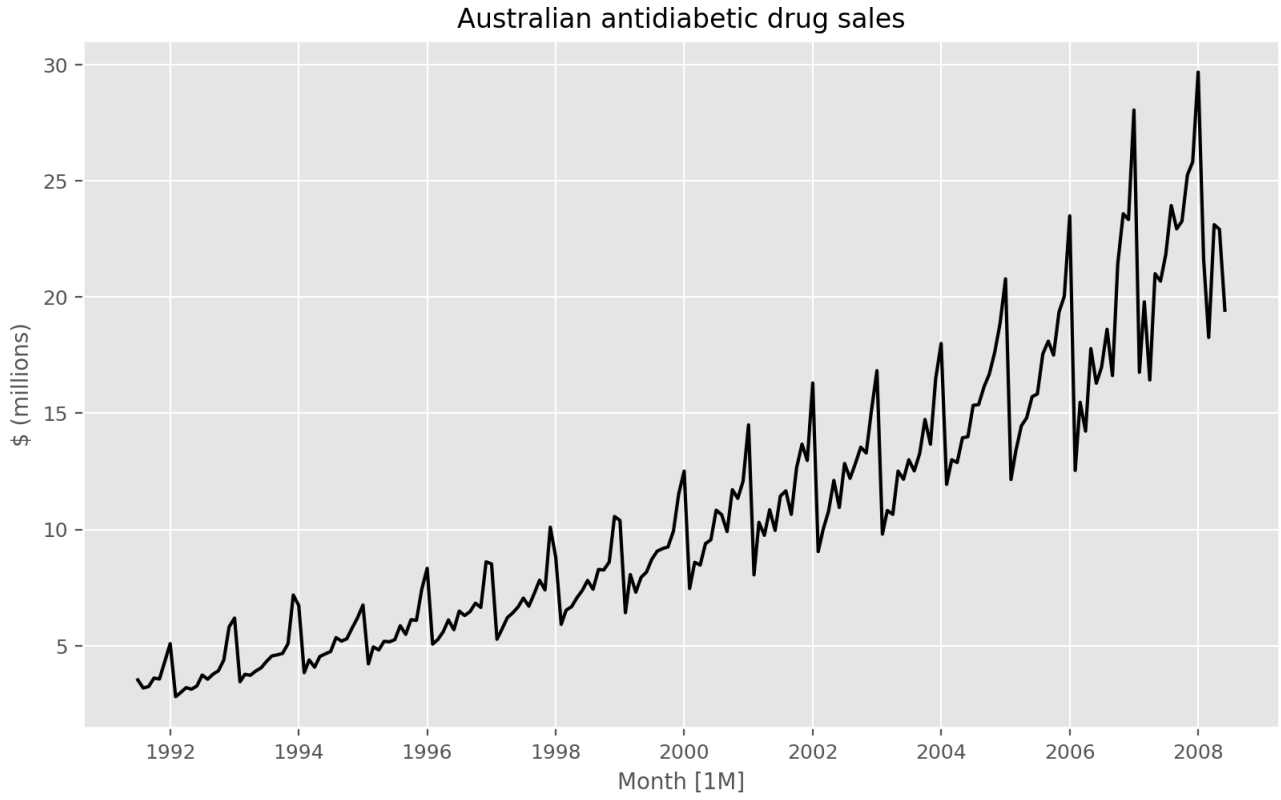


Figure 2.2: Monthly sales of antidiabetic drugs in Australia.

Here, there is a clear and increasing trend. There is also a strong seasonal pattern that increases in size as the level of the series increases. The sudden drop at the start of each year is caused by a government subsidisation scheme that makes it cost-effective for patients to stockpile drugs at the end of the calendar year. Any forecasts of this series would need to capture the seasonal pattern, and the fact that the trend is changing slowly.

2.3 Time series patterns

In describing these time series, we often use terms like “trend” and “seasonal” which require further clarification.

Trend

A *trend* exists when there is a long-term increase or decrease in the data. It does not have to be linear. Sometimes we will refer to a trend as “changing direction”, when it might go from an increasing trend to a decreasing trend. There is a trend in the antidiabetic drug sales data shown in Figure 2.2.

Seasonal

A *seasonal* pattern occurs when a time series is affected by seasonal factors such as the time of the year, the day of the week or the hour of the day. Seasonality is always of a fixed and known period. The monthly sales of antidiabetic drugs (Figure 2.2) shows seasonality which is induced partly by the change in the cost of the drugs at the end of the calendar year. (Note that one series can have more than one seasonal pattern.)

Cyclic

A *cycle* occurs when the data exhibit rises and falls that are not of a fixed frequency. These fluctuations are usually due to economic conditions, and are often related to the “business cycle”. The duration of these fluctuations is usually at least 2 years.

Many people confuse cyclic behaviour with seasonal behaviour, but they are really quite different. If the fluctuations are not of a fixed frequency then they are cyclic; if the frequency is unchanging and associated with some aspect of the calendar, then the pattern is seasonal. In general, the average length of cycles is longer than the length of a seasonal pattern, and the magnitudes of cycles tend to be more variable than the magnitudes of seasonal patterns.

Many time series include trend, cycles and seasonality. When choosing a forecasting method, we will first need to identify the time series patterns in the data, and then choose a method that is able to capture the patterns properly.

The examples in Figure 2.3 show different combinations of these components.

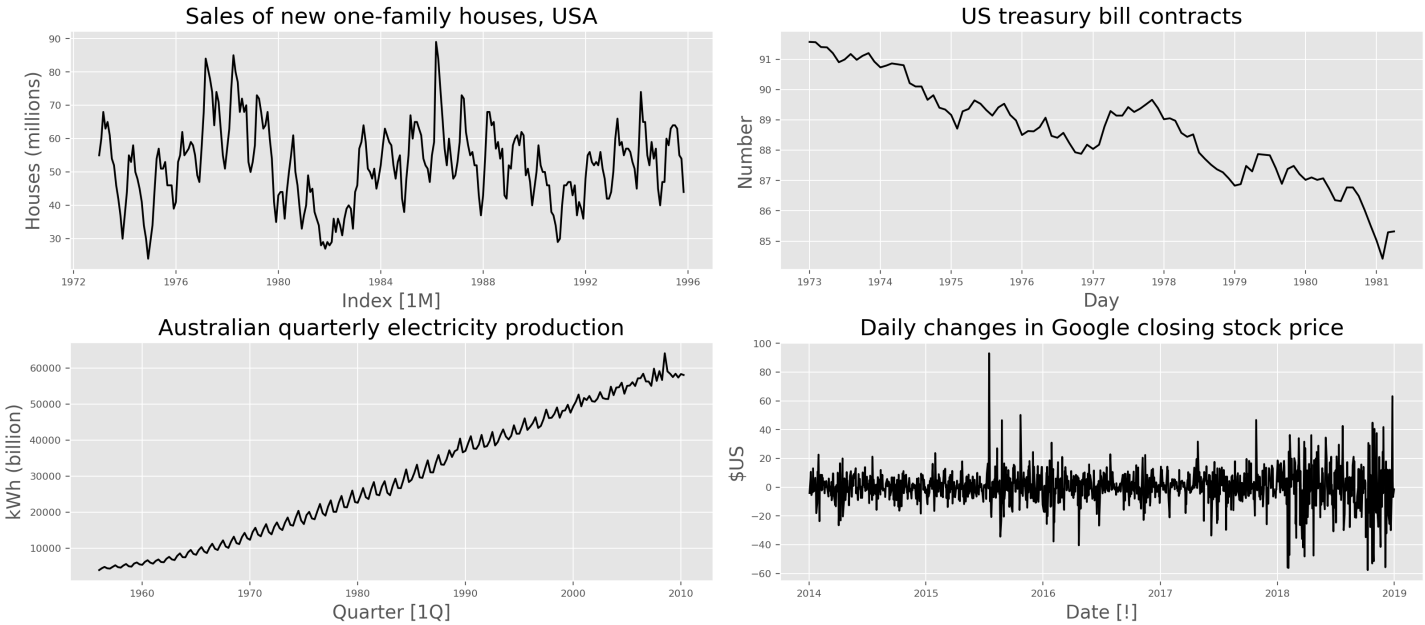


Figure 2.3: Four examples of time series showing different patterns.

1. The monthly housing sales (top left) show strong seasonality within each year, as well as some strong cyclic behaviour with a period of about 6–10 years. There is no apparent trend in the data over this period.
2. The US treasury bill contracts (top right) show results from the Chicago market for 100 consecutive trading days in 1981. Here there is no seasonality, but an obvious downward trend. Possibly, if we had a much longer series, we would see that this downward trend is actually part of a long cycle, but when viewed over only 100 days it appears to be a trend.
3. The Australian quarterly electricity production (bottom left) shows a strong increasing trend, with strong seasonality. There is no evidence of any cyclic behaviour here.
4. The daily change in the Google closing stock price (bottom right) has no trend, seasonality or cyclic behaviour. There are random fluctuations which do not appear to be very predictable, and no strong patterns that would help with developing a forecasting model.

2.4 Seasonal plots

A seasonal plot is similar to a time plot except that the data are plotted against the individual “seasons” in which the data were observed. An example is given in Figure 2.4 showing the antidiabetic drug sales.

```
total_cost_df["Month_name"] = total_cost_df["Month"].dt.strftime("%b")
total_cost_df["Year"] = total_cost_df["Month"].dt.year
total_cost_df["Month_num"] = total_cost_df["Month"].dt.month

unique_years = total_cost_df["Year"].unique()
year_palette = sns.color_palette("husl", n_colors=len(unique_years))

fig, ax = plt.subplots()
sns.lineplot(
    data=total_cost_df,
    x="Month_num",
    y="Cost",
    hue="Year",
    palette=year_palette,
    legend=False,
    ax=ax,
)
ax.set_title("Seasonal Plot: Antidiabetic Drug Sales")
ax.set_xlabel("Month")
ax.set_ylabel("$ (millions)")
ax.set_xticks(
    ticks=range(1, 13),
    labels=[
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    ],
)

min_year = unique_years.min()
for year, subset in total_cost_df.groupby("Year"):
    ax.text(
        subset["Month_num"].iloc[-1],
        subset["Cost"].iloc[-1],
        str(year),
        fontsize=9,
        weight="bold",
        color=year_palette[year - min_year],
    )

fig.show()
```

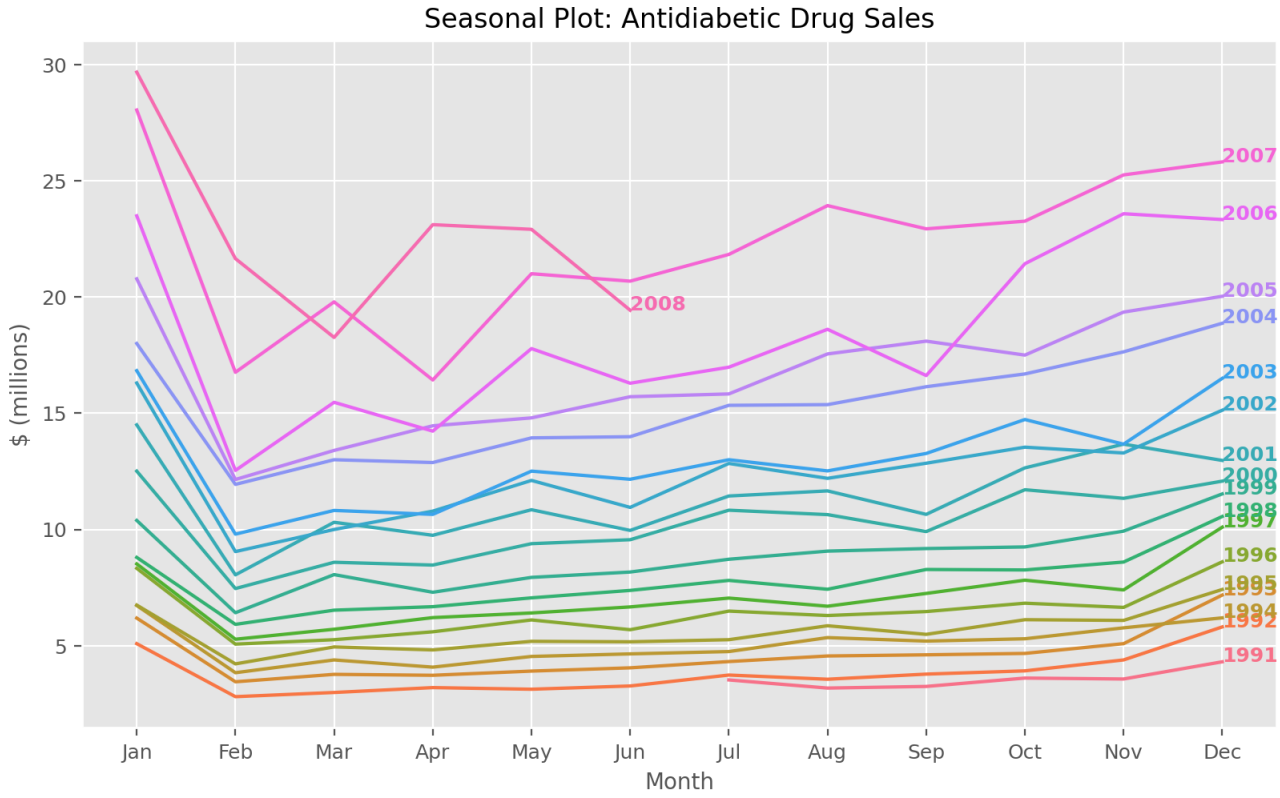


Figure 2.4: Seasonal plot of monthly antidiabetic drug sales in Australia.

This is the same data shown earlier, but now the data from each season (year) is overlapped. A seasonal plot shows the underlying seasonal pattern more clearly and helps identify years when the pattern changes.

There is a large jump in sales in January each year. These are probably sales in late December as customers stockpile before the end of the calendar year, but the sales are not registered with the government until a week or two later. The graph also shows an unusually small number of sales in March 2008 (most other years show an increase between February and March). The small number of sales in June 2008 is probably due to incomplete counting of sales when the data were collected.

Multiple seasonal periods

Where the data has more than one seasonal pattern, we can group the data according to different seasonalities and plot. The `vic_elec` data contains half-hourly (`freq='30T'`) electricity demand for the state of Victoria, Australia. We can plot the daily pattern, weekly pattern or yearly pattern by grouping according to date, week and year as shown in Figures 10.14–2.7.

In the first plot, the three days with 25 hours are when daylight saving ended in each year and so these days contained an extra hour. There were also three days with only 23 hours each (when daylight saving started) but these are hidden beneath all the other lines on the plot.

```
vic_elec_df = pd.read_csv("../data/vic_elec.csv")
vic_elec_df["ds"] = pd.to_datetime(vic_elec_df["ds"])
vic_elec_demand = vic_elec_df[vic_elec_df["unique_id"] == "Demand"].copy()
vic_elec_demand["hour-minute"] = \
  vic_elec_demand["ds"].dt.strftime("%H:%M:%S")
vic_elec_demand["day"] = vic_elec_demand["ds"].dt.date

fig, ax = plt.subplots()
sns.lineplot(
    data=vic_elec_demand,
    x="hour-minute",
    y="y",
    hue="day",
    palette="husl",
    legend=False,
    ax=ax,
)
unique_ticks = vic_elec_demand["hour-minute"].unique()
ticks_to_plot = unique_ticks[::2]
ax.set_xticks(ticks=range(0, len(unique_ticks), 2), labels=ticks_to_plot,
    rotation=45)
ax.set_title("Electricity Demand: Victoria")
ax.set_xlabel("Time")
ax.set_ylabel("MWh")
fig.show()
```

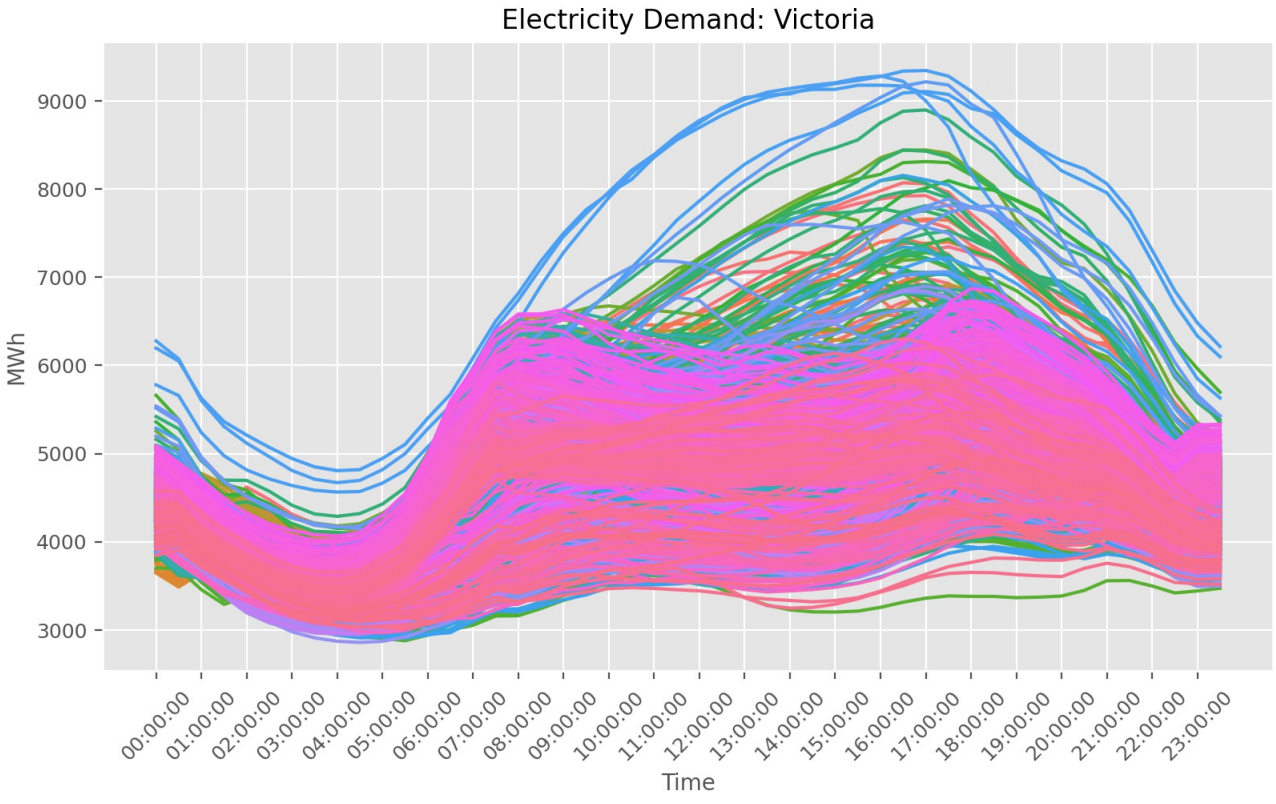


Figure 2.5: Seasonal plot showing daily seasonal patterns for Victorian electricity demand.

```
vic_elec_demand["day_of_week"] = vic_elec_demand["ds"].dt.day_name()
vic_elec_demand = vic_elec_demand[
    (vic_elec_demand["ds"] >= "2012-01-02") &
    (vic_elec_demand["ds"] < "2014-12-29")
].copy()

weeks = vic_elec_demand["ds"].dt.to_period("W-SUN").dt.start_time
unique_weeks = weeks.unique()
palette = sns.color_palette("husl", n_colors=len(unique_weeks))
color_map = dict(zip(unique_weeks, palette))

groups = vic_elec_demand["ds"].dt.to_period("W-SUN").dt.start_time
fig, ax = plt.subplots()
for df_week in vic_elec_demand.groupby(groups):
    week, df_w = df_week
    df_w.plot(
        y="y",
        x="day_of_week",
        ax=ax,
        color=color_map[week],
        label=str(week.date())
    )
ax.get_legend().remove()
ax.set_title("Electricity Demand: Victoria")
ax.set_ylabel("MWh")
ax.set_xlabel("Time")
fig.show()
```

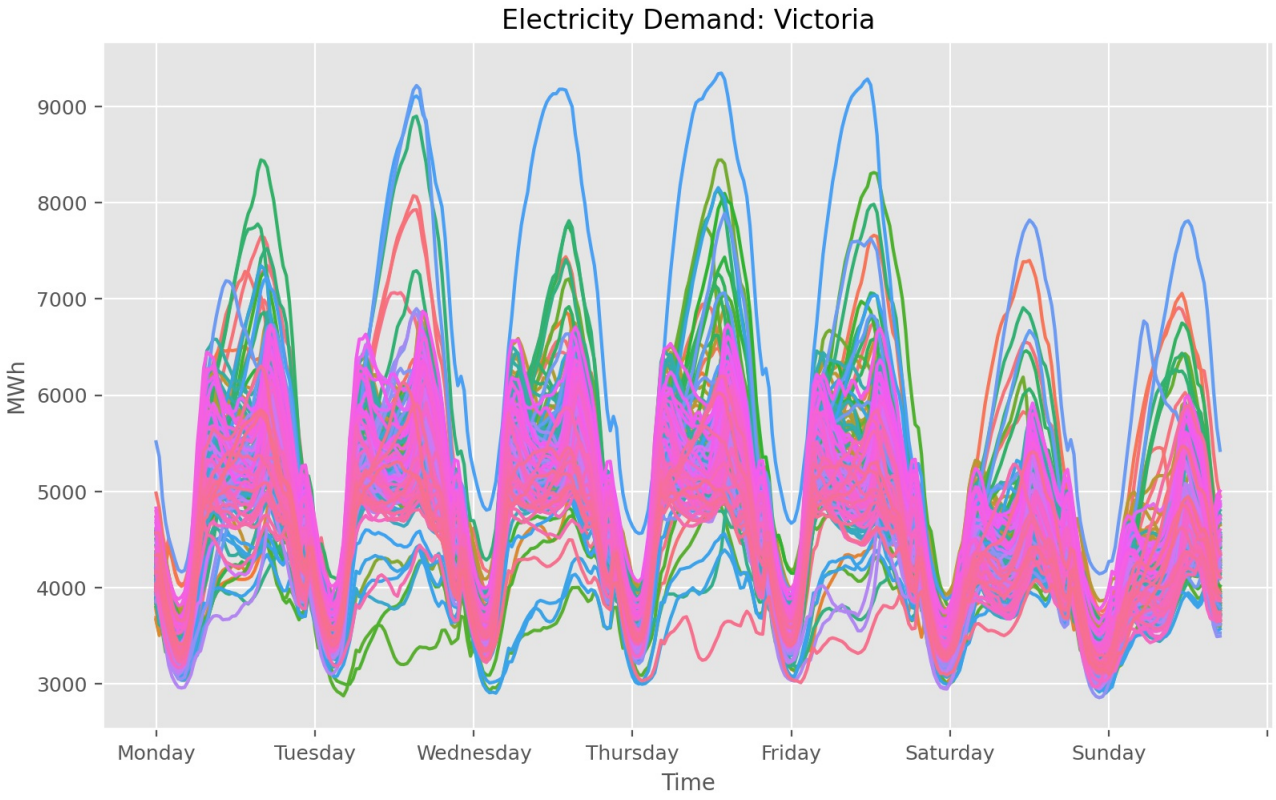


Figure 2.6: Seasonal plot showing weekly seasonal patterns for Victorian electricity demand.

```
vic_elec_demand["day_of_year"] = vic_elec_demand["ds"].dt.strftime("%m-%d")

unique_years = vic_elec_demand["ds"].dt.year.unique()
palette = sns.color_palette("husl", n_colors=len(unique_years))
color_map = dict(zip(unique_years, palette))

fig, ax = plt.subplots()
for df_year in vic_elec_demand.groupby(vic_elec_demand["ds"].dt.year):
    year, df_y = df_year
    df_y.plot(
        y="y",
        x="day_of_year",
        ax=ax,
        label=str(year),
        color=color_map[year]
    )
ax.set_title("Electricity Demand: Victoria")
ax.set_ylabel("MWh")
ax.set_xlabel("Time")
fig.show()
```

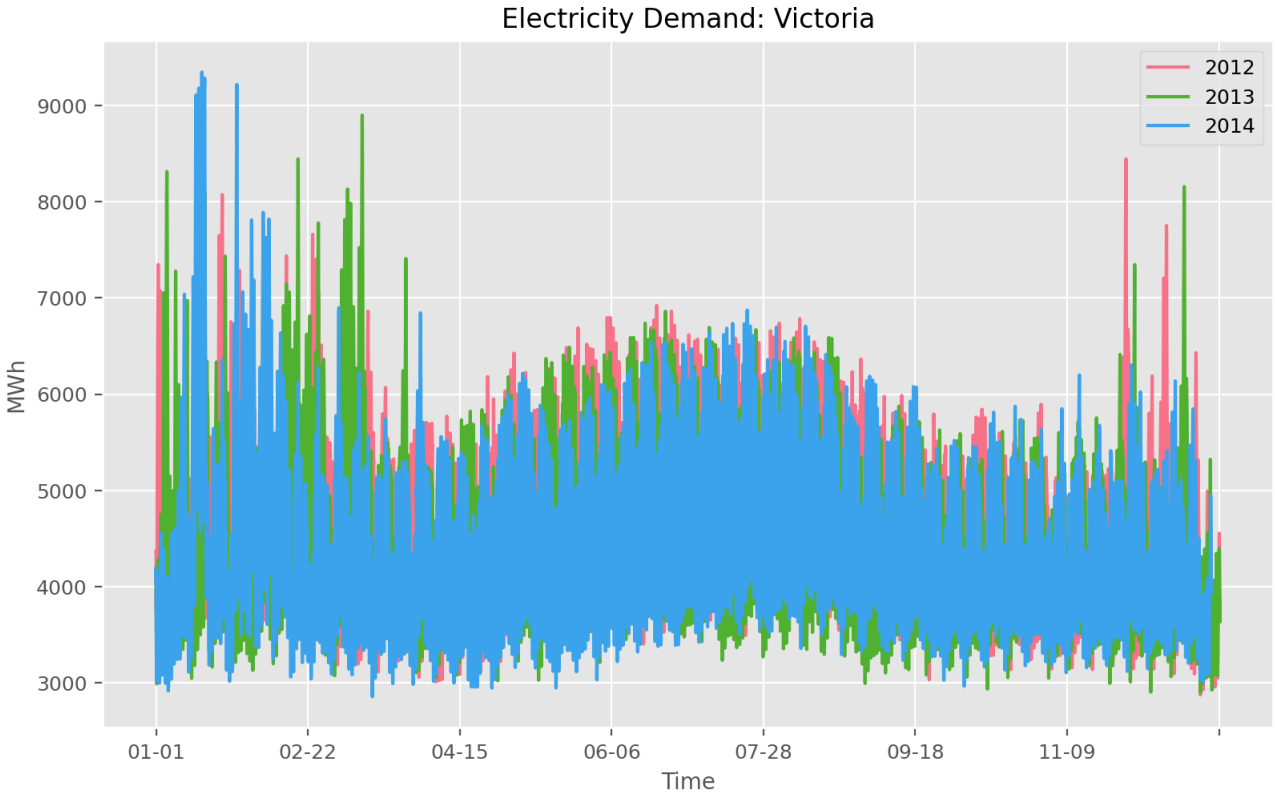


Figure 2.7: Seasonal plot showing yearly seasonal patterns for Victorian electricity demand.

2.5 Seasonal subseries plots

An alternative plot that emphasises the seasonal patterns is where the data for each season are collected together in separate mini time plots.

```
total_cost_df
```

	Month	TotalC	Cost	unique_id	Month_name	Year	Month_num
0	1991-07-01	3526591.0	3.53	total_cost	Jul	1991	7
1	1991-08-01	3180891.0	3.18	total_cost	Aug	1991	8
2	1991-09-01	3252221.0	3.25	total_cost	Sep	1991	9
3	1991-10-01	3611003.0	3.61	total_cost	Oct	1991	10
4	1991-11-01	3565869.0	3.57	total_cost	Nov	1991	11
...	...	...	...	...	...	...	...
199	2008-02-01	21654285.0	21.65	total_cost	Feb	2008	2
200	2008-03-01	18264945.0	18.26	total_cost	Mar	2008	3
201	2008-04-01	23107677.0	23.11	total_cost	Apr	2008	4
202	2008-05-01	22912510.0	22.91	total_cost	May	2008	5
203	2008-06-01	19431740.0	19.43	total_cost	Jun	2008	6

204 rows × 7 columns

```
total_cost_df["Month"] = pd.to_datetime(total_cost_df["Month"])
total_cost_df["year"] = total_cost_df["Month"].dt.year
total_cost_df["month"] = total_cost_df["Month"].dt.strftime("%B")
total_cost_df["month"] = pd.Categorical(
  total_cost_df["month"],
  categories=[
    "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December",
  ],
  ordered=True,
)

fig, axes = plt.subplots(nrows=1, ncols=12, sharey=True)
for i, month in enumerate(total_cost_df["month"].cat.categories):
    month_data = total_cost_df.query("month == @month")
    mean_cost = month_data["Cost"].mean()
    axes[i].plot(month_data["year"], month_data["Cost"], color="black")
    axes[i].axhline(
        mean_cost, color="blue", linestyle="-", linewidth=1, label="Average"
    )
    axes[i].set_title(month[:3])
    axes[i].set_xlabel("")
    axes[i].tick_params(axis='x', rotation=90)
    if i == 0:
        axes[i].set_ylabel("$(millions)")
    else:
        axes[i].set_yticklabels([])

fig.suptitle("Australian Antidiabetic Drug Sales")
fig.text(0.5, -0.05, "Month", ha="center")
fig.subplots_adjust(wspace=0.2)
fig.show()
```

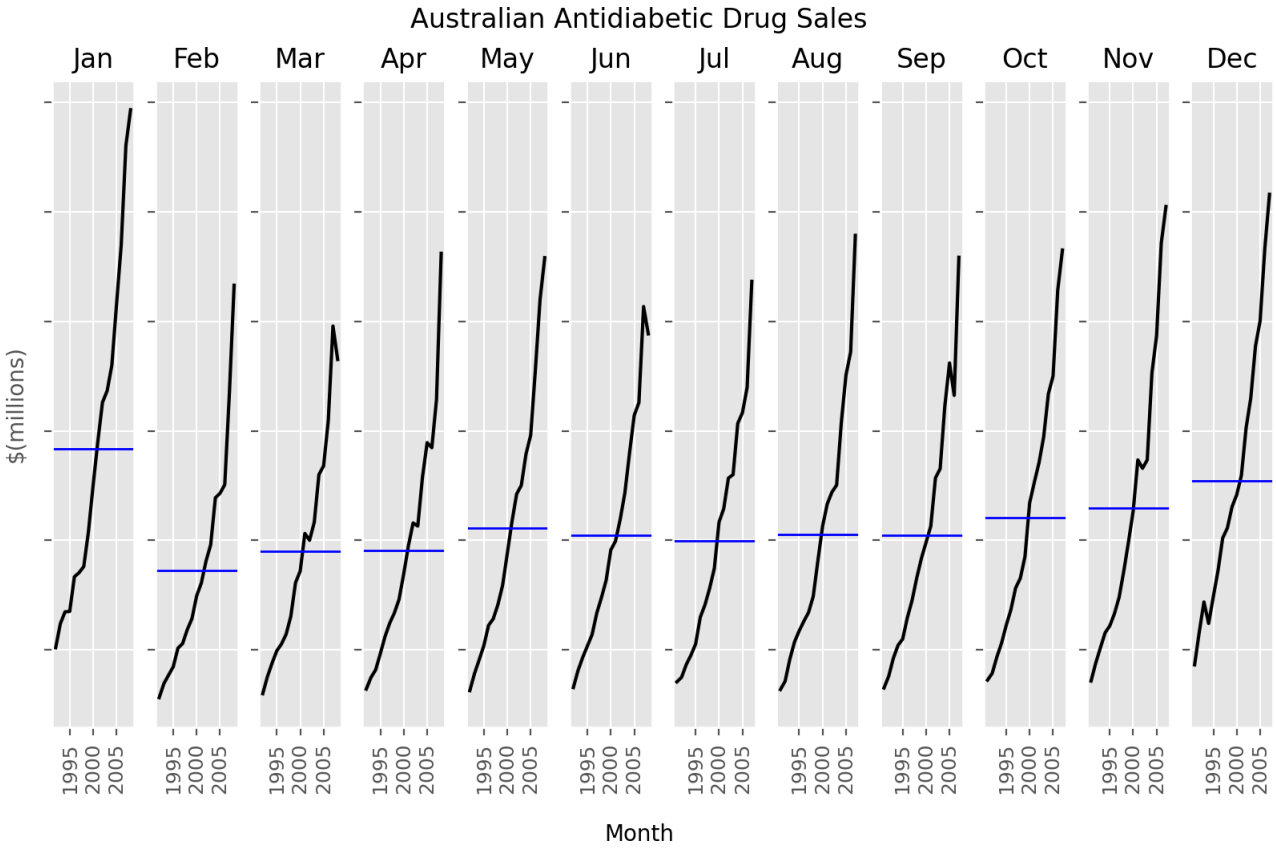


Figure 2.8: Seasonal subseries plot of monthly antidiabetic drug sales in Australia.

The blue horizontal lines indicate the means for each month. This form of plot enables the underlying seasonal pattern to be seen clearly, and also shows the changes in seasonality over time. It is especially useful in identifying changes within particular seasons. In this example, the plot is not particularly revealing; but in some cases, this is the most useful way of viewing seasonal changes over time.

Example: Australian holiday tourism

Australian quarterly vacation data provides an interesting example of how these plots can reveal information. First we need to extract the relevant data from the `tourism` dataframe. All the usual `pandas` wrangling functions apply. To get the total visitor nights spent on Holiday by State for each quarter (i.e., ignoring Regions) we can use the following code.

```
tourism = pd.read_csv("../data/tourism.csv")
tourism["ds"] = pd.to_datetime(tourism["ds"])
tourism["Quarter"] = tourism["ds"].dt.to_period("Q").astype(str)
tourism_sub = tourism.query('Purpose == "Holiday"')
trips = tourism_sub.groupby(["State", "ds"], as_index=False)["y"].sum().round(2)
```

```
trips.head(10)
```

	State	ds	y
0	ACT	1998-01-01	196.22
1	ACT	1998-04-01	126.77
2	ACT	1998-07-01	110.68
3	ACT	1998-10-01	170.47
4	ACT	1999-01-01	107.78
5	ACT	1999-04-01	124.64
6	ACT	1999-07-01	177.95
7	ACT	1999-10-01	217.66
8	ACT	2000-01-01	158.41
9	ACT	2000-04-01	154.81

Time plots of each series show that there is strong seasonality for most states, but that the seasonal peaks do not coincide.

```
states = trips["State"].unique()
colors = sns.color_palette("husl", len(states))
fig, ax = plt.subplots()
for state, color in zip(states, colors):
    state_df = trips.query("State == @state")
    state_df.plot(y="y", x="ds", ax=ax, label=state, color=color)
ax.set_title("Australian domestic holidays")
handles, labels = ax.get_legend_handles_labels()
ax.get_legend().remove()
fig.legend(
    handles, labels, loc="center left", bbox_to_anchor=(1.02, .5),
    frameon=False, borderaxespad=0,
)

ax.set_ylabel("Overnight trips ('000)")
ax.set_xlabel("Quarter [1Q]")
fig.show()
```

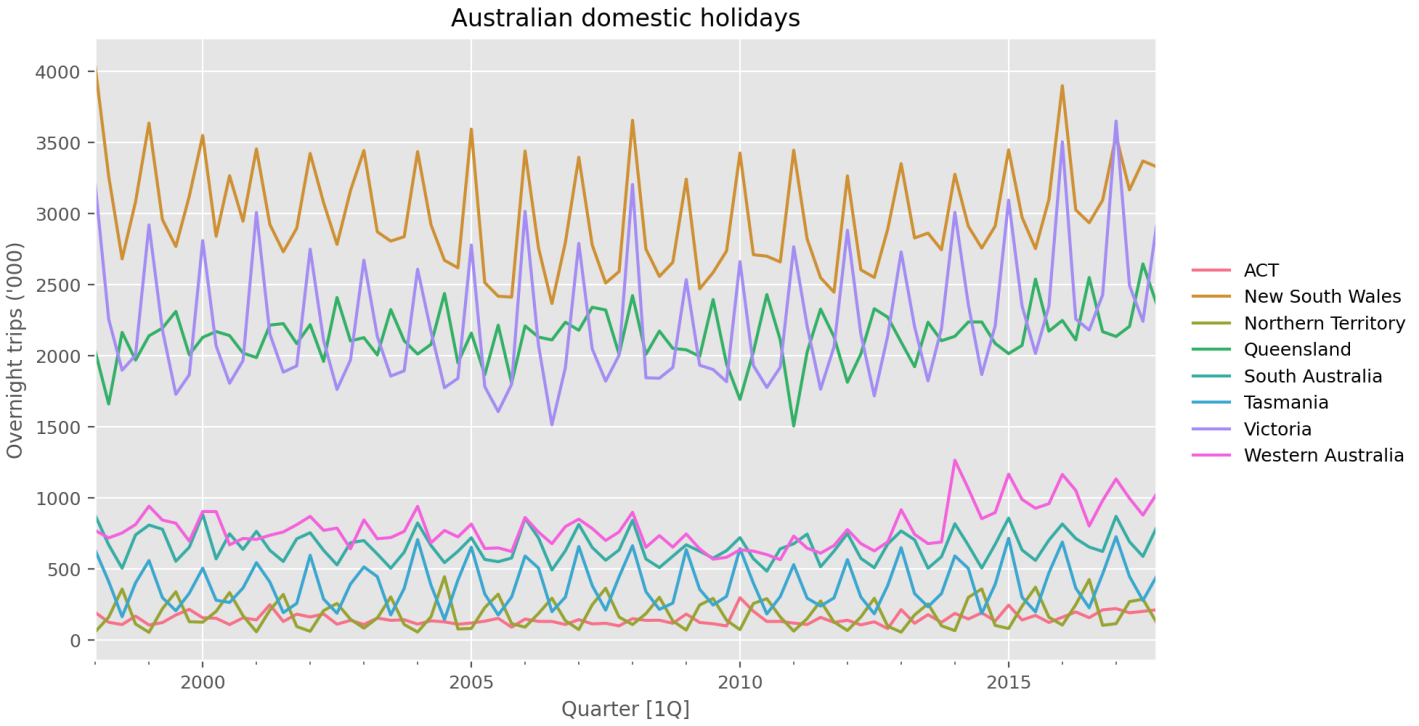


Figure 2.9: Time plots of Australian domestic holidays by state.

To see the timing of the seasonal peaks in each state, we can use a season plot. Figure 2.10 makes it clear that the southern states of Australia (Tasmania, Victoria and South Australia) have strongest tourism in Q1 (their summer), while the northern states (Queensland and the Northern Territory) have the strongest tourism in Q3 (their dry season).

```
trips["Quarter"] = "Q" + trips["ds"].dt.quarter.astype(str)
trips["Year"] = trips["ds"].dt.year
years = sorted(trips["Year"].unique())
palette = sns.color_palette("husl", len(years))
year_to_color = dict(zip(years, palette))

n_states = trips["State"].nunique()

fig, axs = plt.subplots(n_states, 1, sharex=True, figsize=(8, 10))
for ax, (state, df_state) in zip(axs, trips.groupby("State")):
    pivot_data = df_state.pivot(index="Quarter", columns="Year", values="y")
    pivot_data.plot(ax=ax, color=[year_to_color[year] for year in pivot_data.columns])
    ax.get_legend().remove()
    ax.text(1.02, 0.5, state, va="center", ha="right", rotation=270,
        fontsize=10, transform=ax.transAxes)

handles = [plt.Line2D([0], [0], color=year_to_color[year], lw=4) for year in years]
labels = [str(year) for year in years]
fig.legend(handles, labels, title="Year", loc="center left", bbox_to_anchor=(1.05, 0.5), frameon=F


fig.supylabel("Overnight trips ('000)")
fig.suptitle("Australian domestic holidays")
fig.show()
```

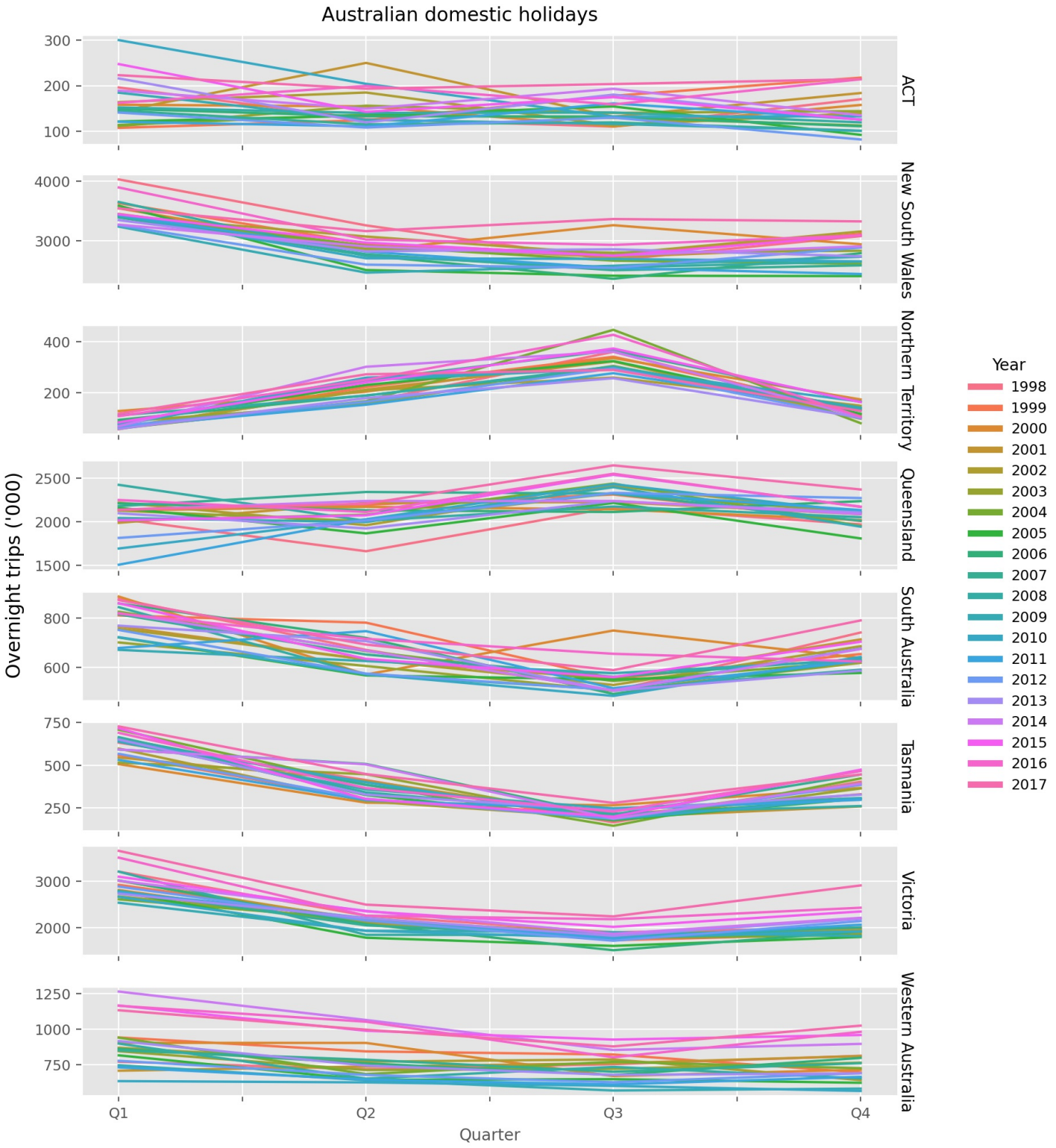


Figure 2.10: Seasonal plot of Australian domestic holidays by state.

The corresponding subseries plots are shown in Figure 2.11.

```
fig, axs = plt.subplots(n_states, 4, sharex=True, figsize=(8, 10))
axs = axs.flatten()
quarters = sorted(trips["Quarter"].unique())
states = sorted(trips["State"].unique())
for idx, (state_i, df_state) in enumerate(trips.groupby("State")):
    y_state_min, y_state_max = df_state["y"].min(), df_state["y"].max()
    for jdx, (quart_i, df_state_quart) in \
            enumerate(df_state.groupby("Quarter")):
        axi = axs[idx * 4 + jdx]
        df_state_quart.plot(y="y", x="ds", label=None, ax=axi,
            color="black")
        axi.axhline(df_state_quart["y"].mean(), color="blue")
        axi.set_ylim(y_state_min, y_state_max)
        axi.set_xlabel("")
        axi.tick_params(axis="x", rotation=90)
        axi.get_legend().remove()
for j, quarter in enumerate(quarters):
    axs[j].set_title(quarter)
for i, state in enumerate(states):
    axs[i * 4 + 3].text(
        1.02, 0.5, state, va="center", ha="left", rotation=270,
        transform=axs[i * 4 + 3].transAxes
    )
fig.supylabel("Overnight trips ('000)", va="center", rotation=90)
fig.supxlabel("Quarter", ha="center")
fig.suptitle("Australian domestic holidays")
fig.show()
```

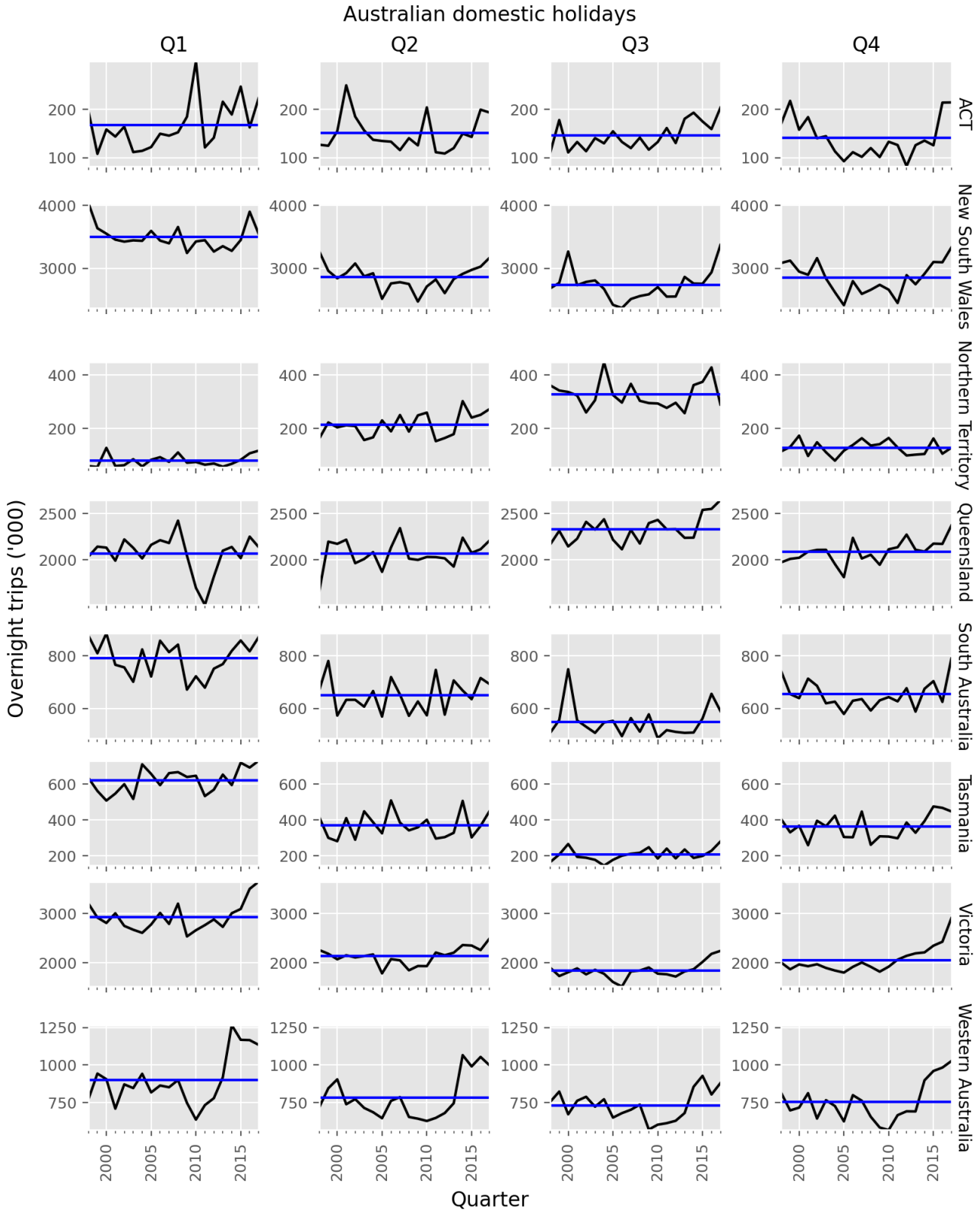


Figure 2.11: Seasonal subseries plot of Australian domestic holidays by state.

This figure makes it evident that Western Australian tourism has jumped markedly in recent years, while Victorian tourism has increased in Q1 and Q4 but not in the middle of the year.

2.6 Scatterplots

The graphs discussed so far are useful for visualising individual time series. It is also useful to explore relationships *between* time series.

Figures 2.12 and 2.13 show two time series: half-hourly electricity demand (in Gigawatts) and temperature (in degrees Celsius),

for 2014 in Victoria, Australia. The temperatures are for Melbourne, the largest city in Victoria, while the demand values are for the entire state.

```
plot_series(vic_elec_df, ids=["Demand"],
            max_insample_length=2 * 24 * 365,
            xlabel="Time [30m]",
            ylabel="GW",
            title="Half-hourly electricity demand: Victoria")
```

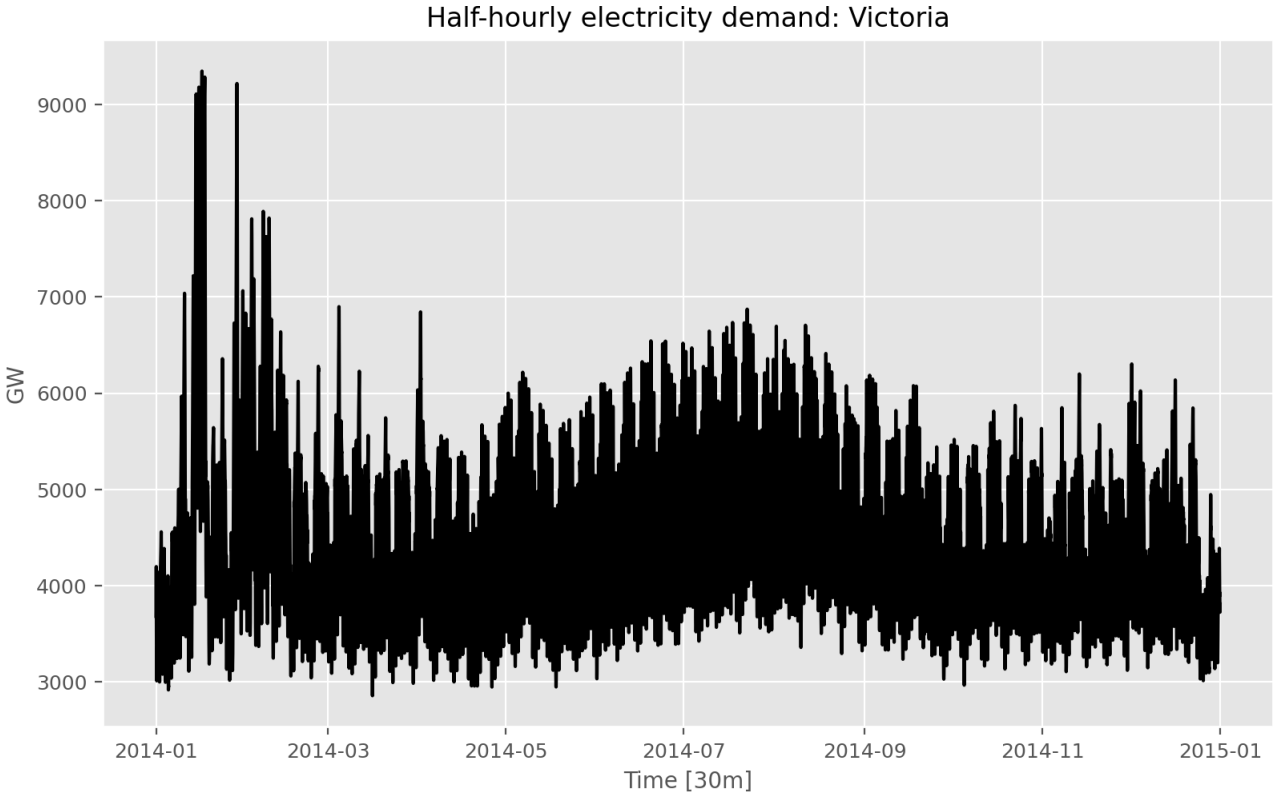


Figure 2.12: Half hourly electricity demand in Victoria, Australia, for 2014.

```
plot_series(vic_elec_df, ids=["Temperature"],
            max_insample_length=2 * 24 * 365,
            xlabel="Time [30m]",
            ylabel="Degrees Celsius",
            title="Half-hourly temperature: Melbourne, Australia")
```

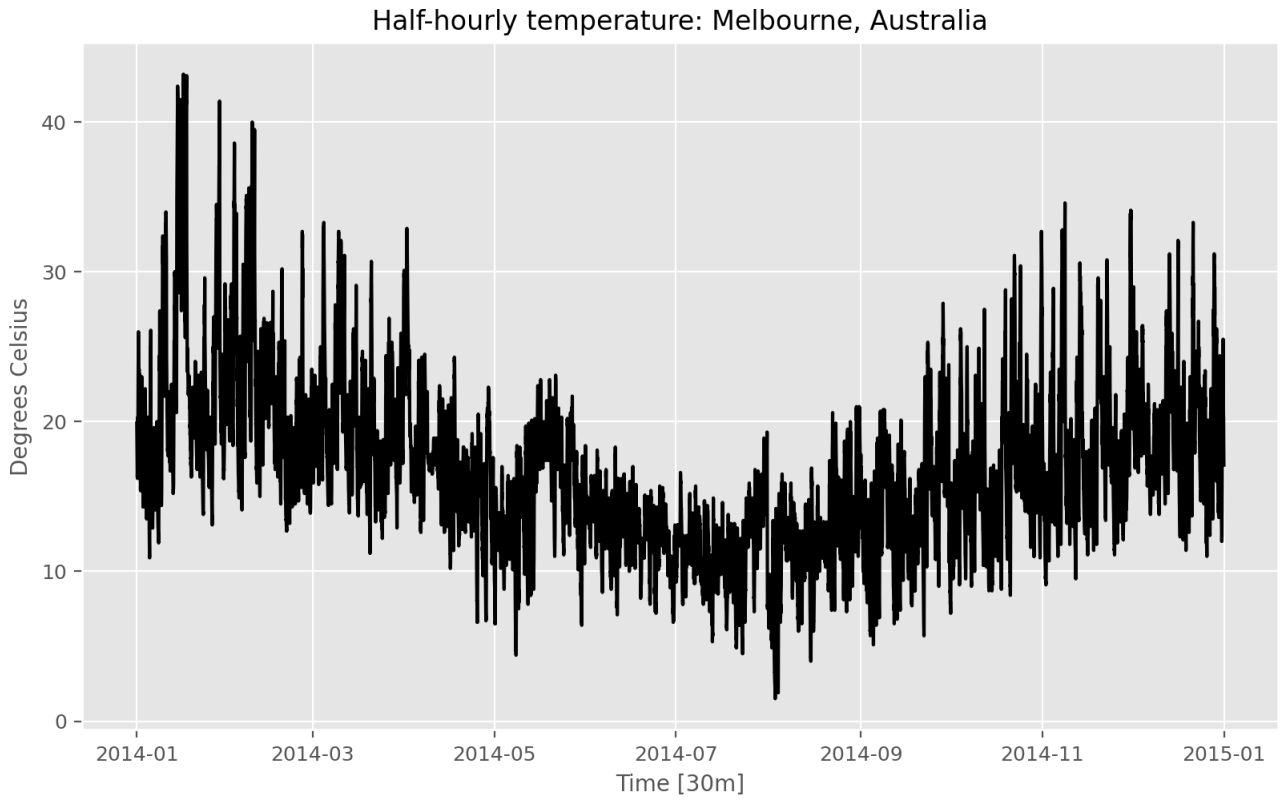


Figure 2.13: Half hourly temperature in Melbourne, Australia, for 2014.

We can study the relationship between demand and temperature by plotting one series against the other.

```
elec_2014 = vic_elec_df.query('ds >= "2014"')
elec_2014_pivot = elec_2014.pivot(
    index=["ds", "Holiday"], columns="unique_id", values="y"
).reset_index()
fig, ax = plt.subplots()
sns.scatterplot(data=elec_2014_pivot, x="Temperature", y="Demand", ax=ax)
ax.set_title("Scatter Plot of Demand vs. Temperature")
ax.set_xlabel("Temperature (degrees Celsius)")
ax.set_ylabel("Electricity demand (GW)")
fig.show()
```

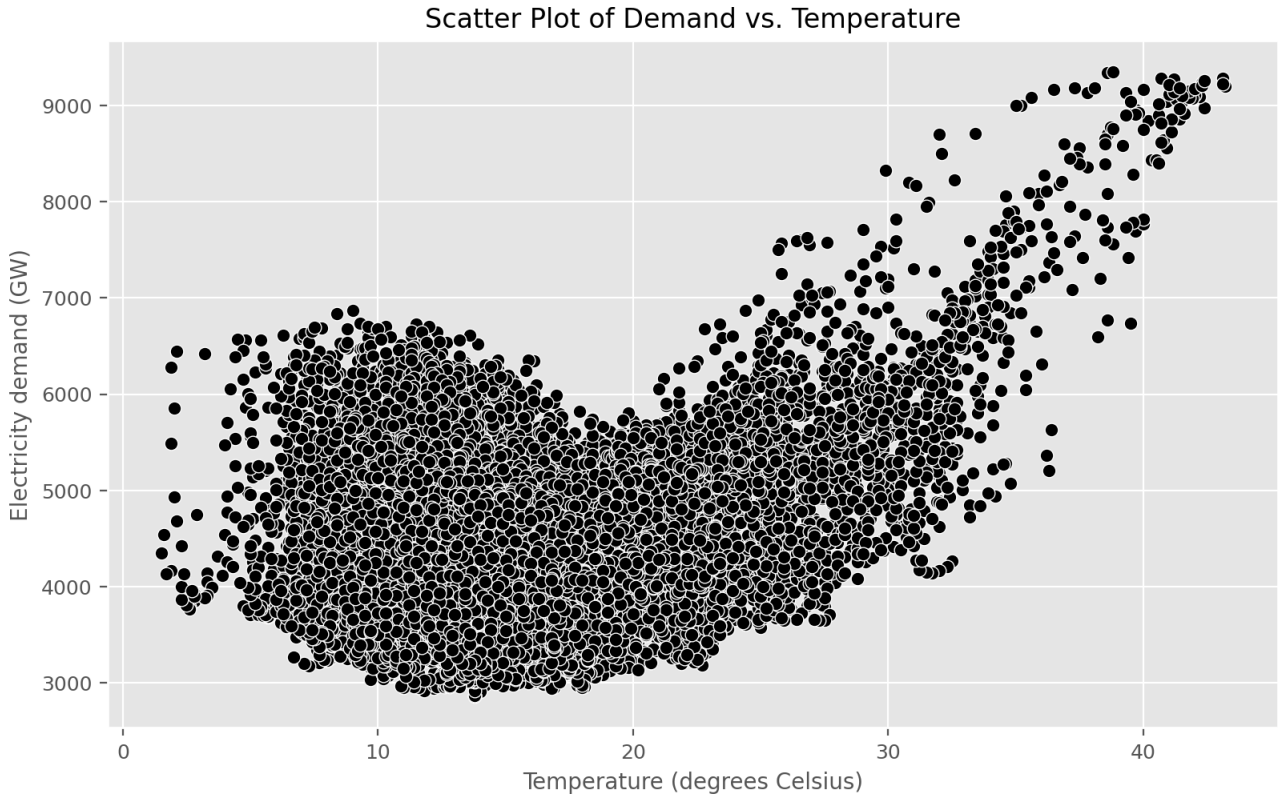


Figure 2.14: Half-hourly electricity demand plotted against temperature for 2014 in Victoria, Australia.

This scatterplot helps us to visualise the relationship between the variables. It is clear that high demand occurs when temperatures are high due to the effect of air-conditioning. But there is also a heating effect, where demand increases for very low temperatures.

Correlation

It is common to compute *correlation coefficients* to measure the strength of the linear relationship between two variables. The correlation between variables x and y is given by r = \frac{\sum (x_{t} - \bar{x})(y_{t}-\bar{y})}{\sqrt{\sum (x_{t} - \bar{x})^2} \sqrt{\sum (y_{t} - \bar{y})^2}}. The value of r always lies between -1 and 1 with negative values indicating a negative relationship and positive values indicating a positive relationship. The graphs in Figure 2.15 show examples of data sets with varying levels of correlation.

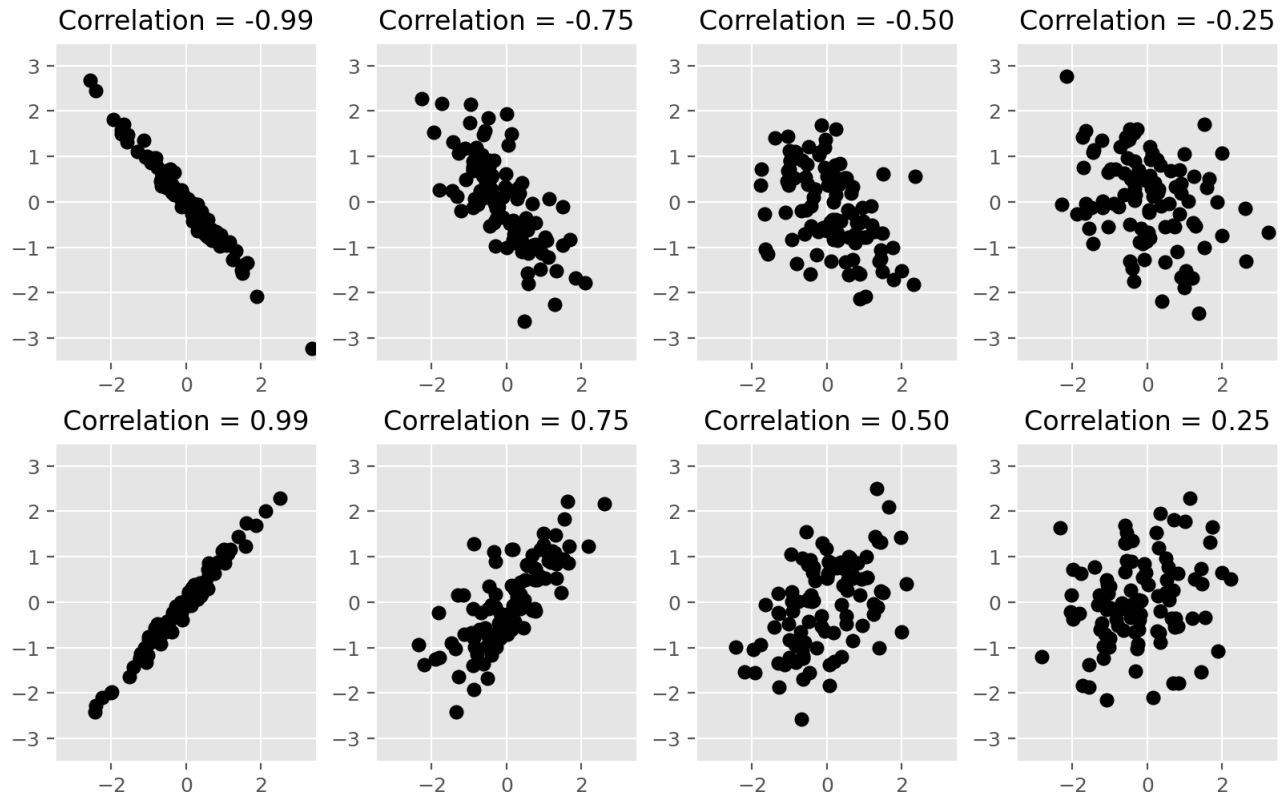


Figure 2.15: Examples of data sets with different levels of correlation.

The correlation coefficient only measures the strength of the *linear* relationship between two variables, and can sometimes be

misleading. For example, the correlation for the electricity demand and temperature data shown in Figure 2.14 is 0.28, but the *non-linear* relationship is stronger than that.

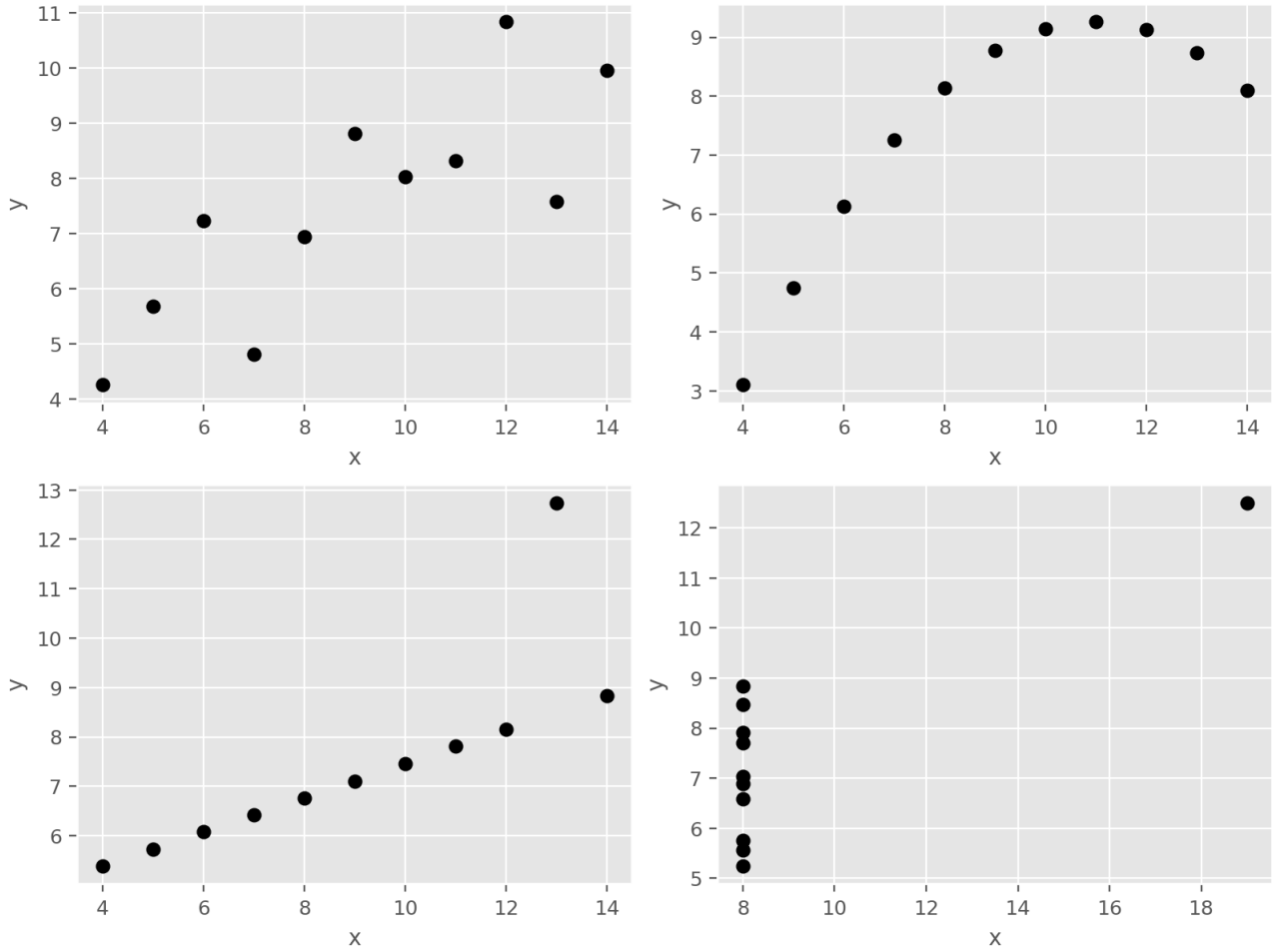


Figure 2.16: Each of these plots has a correlation coefficient of 0.82. Data from Anscombe (1973).

The plots in Figure 2.16 all have correlation coefficients of 0.82, but they have very different relationships. This shows how important it is to look at the plots of the data and not simply rely on correlation values.

Scatterplot matrices

When there are several potential predictor variables, it is useful to plot each variable against each other variable. Consider the eight time series shown in Figure 2.17, showing quarterly visitor numbers across states and territories of Australia.

```
visitors = tourism.groupby(["State", "ds"], as_index=False)["y"].sum()
n_states = visitors["State"].nunique()
fig, axs = plt.subplots(n_states, 1, sharex=True, figsize=(8, 10))
for ax, (state, df_state) in zip(axs, visitors.groupby("State")):
    ax.plot(df_state["ds"], df_state["y"])
    ax.grid(True, linestyle="--", alpha=0.6)
    ax.text(1.02, 0.5, state, va="center", ha="right",
        rotation=270, transform=ax.transAxes)
fig.suptitle("Australian domestic tourism")
fig.supylabel("Overnight trips ('000)", va="center", rotation=90)
fig.supxlabel("Quarter", ha="center")
fig.show()
```

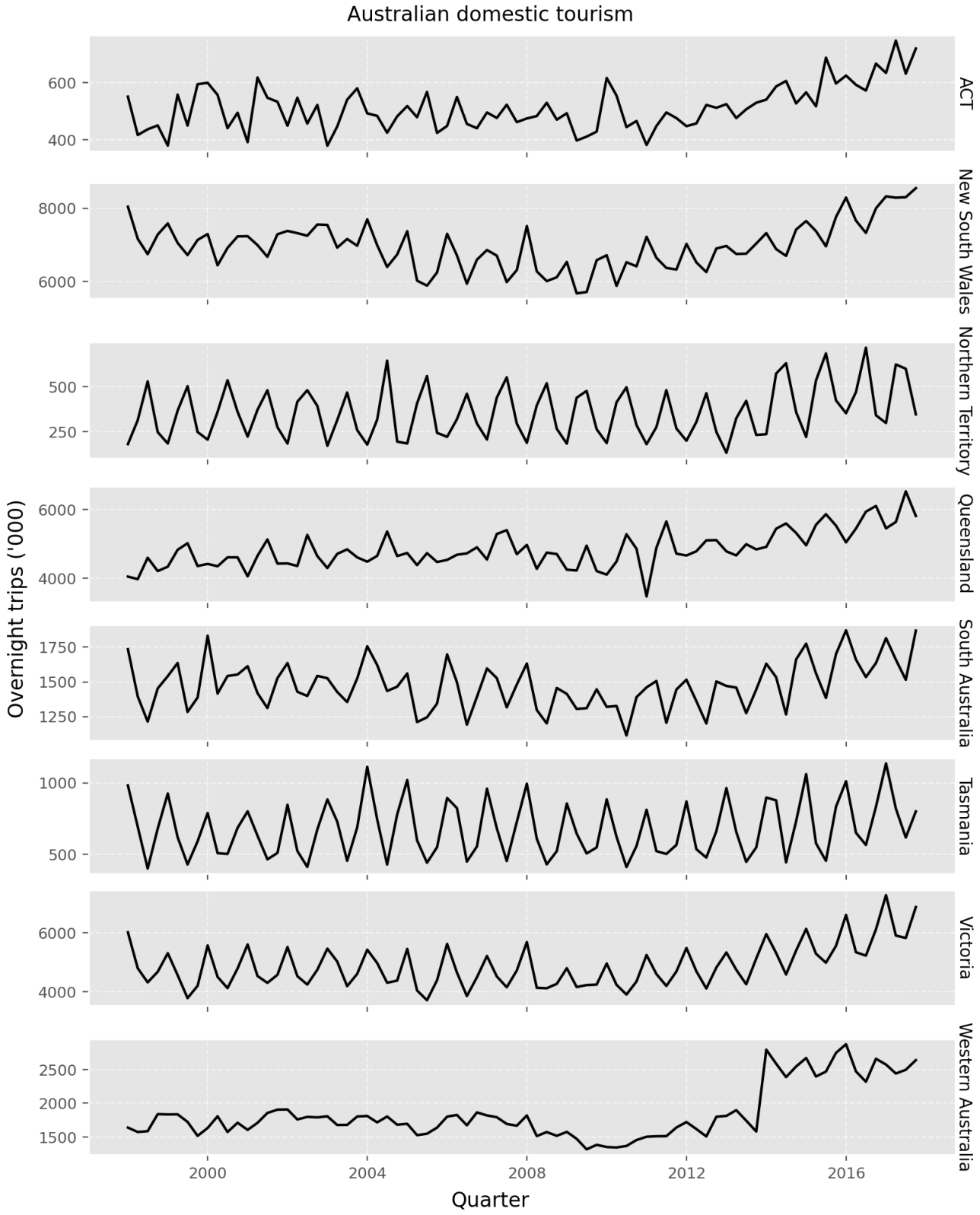


Figure 2.17: Quarterly visitor nights for the states and territories of Australia.

To see the relationships between these eight time series, we can plot each time series against the others. These plots can be arranged in a scatterplot matrix, as shown in Figure 2.18. (This plot requires the `seaborn` package to be installed.)

```
visitors_pivot = \
    visitors.pivot(index="ds", columns="State", values="y").reset_index()
df_for_plot = visitors_pivot.drop(columns=["ds"])

def corrfunc(x, y, **kws):
    r, pvalue = pearsonr(x, y)
    ax = plt.gca()
    ax.annotate(
        f"Corr: \n{r:.3f}{'***' if pvalue < 0.05 else ''}",
        xy=(0.5, 0.5),
        xycoords="axes fraction",
        ha="center",
        va="center",
        fontsize=12,
    )


g = sns.PairGrid(df_for_plot, height=1.6)
g.map_lower(sns.scatterplot)
g.map_upper(corrfunc)
g.map_diag(sns.kdeplot, lw=2)

# Remove default axis labels
g.set(xlabel="")

# Move y-axis labels to the top
for i, col in enumerate(df_for_plot.columns):
    g.axes[0, i].set_title(col, fontsize=12)

fig.show()
```

```
/home/hyndman/git/Books/fpppy/.venv/lib/python3.10/site-packages/seaborn/axisgrid.py:123: UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

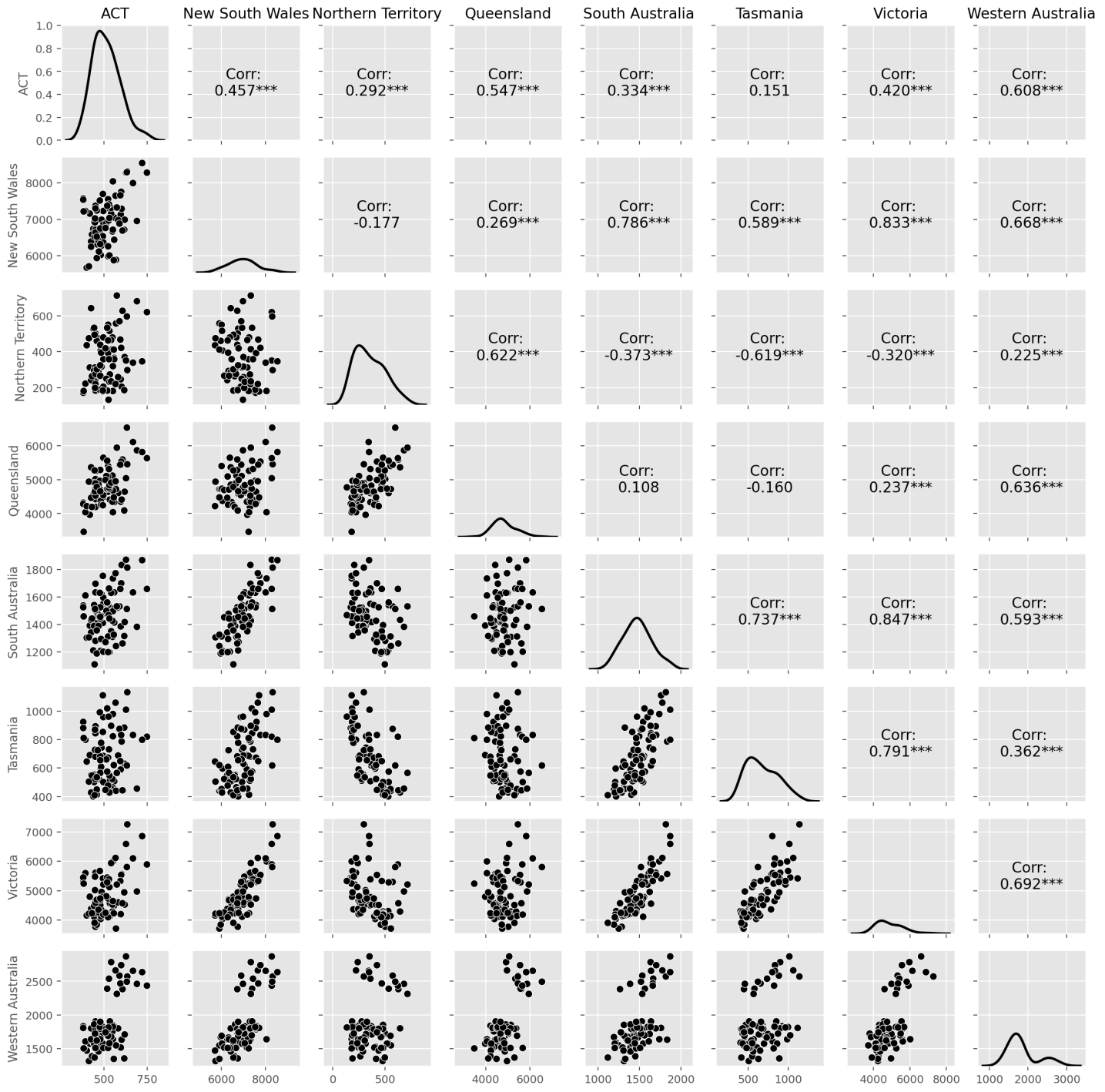


Figure 2.18: A scatterplot matrix of the quarterly visitor nights in the states and territories of Australia.

For each panel, the variable on the vertical axis is given by the variable name in that row, and the variable on the horizontal axis is given by the variable name in that column. There are many options available to produce different plots within each panel. In the default version, the correlations are shown in the upper right half of the plot, while the scatterplots are shown in the lower half. On the diagonal are shown density plots.

The value of the scatterplot matrix is that it enables a quick view of the relationships between all pairs of variables. In this example, mostly positive relationships are revealed, with the strongest relationships being between the neighbouring states located in the south and south east coast of Australia, namely, New South Wales, Victoria and South Australia. Some negative relationships are also revealed between the Northern Territory and other regions. The Northern Territory is located in the north of Australia famous for its outback desert landscapes visited mostly in winter. Hence, the peak visitation in the Northern Territory is in the July (winter) quarter in contrast to January (summer) quarter for the rest of the regions.

2.7 Lag plots

Figure 2.19 displays scatterplots of quarterly Australian beer production (introduced in Figure 1.1), where the horizontal axis shows lagged values of the time series. Each graph shows y_{t} plotted against y_{t-k} for different values of k.

```
recent_production = aus_production[["ds", "Beer"]].query("ds >= 2000")
recent_production.rename(columns={"Beer": "y"}, inplace=True)
recent_production["ds"] = pd.to_datetime(recent_production["ds"])
recent_production["Quarter"] = recent_production["ds"].dt.quarter

for lag in range(1, 10):
    recent_production[f"lag_{lag}"] = recent_production["y"].shift(lag)
lags = [f"lag_{i}" for i in range(1, 10)]
lims = [
    np.min([recent_production[lag].min() for lag in lags] +
        [recent_production["y"].min()]),
    np.max([recent_production[lag].max() for lag in lags] +
        [recent_production["y"].max()]),
]
fig, axes = plt.subplots(3, 3)
for ax, lag in zip(axes.flatten(), lags):
    ax.scatter(
        recent_production[lag],
        recent_production["y"],
        c=recent_production["Quarter"],
        cmap="viridis"
    )
    ax.plot(lims, lims, "grey", linestyle="--", linewidth=1)
    ax.set_title(lag)
unique_quarters = sorted(recent_production["Quarter"].unique())
colors = plt.cm.viridis(np.linspace(0, 1, len(unique_quarters)))
handles = [plt.Line2D([0], [0], color=color, lw=4) for color in colors]
labels = [f"Q{q}" for q in unique_quarters]
fig.legend(handles, labels, title="Season", loc="center left", bbox_to_anchor=(1.02, .5),
    frameon=False, borderaxespad=0,)
fig.supxlabel("lag(Beer, k)")
fig.supylabel("Beer")
fig.show()
```

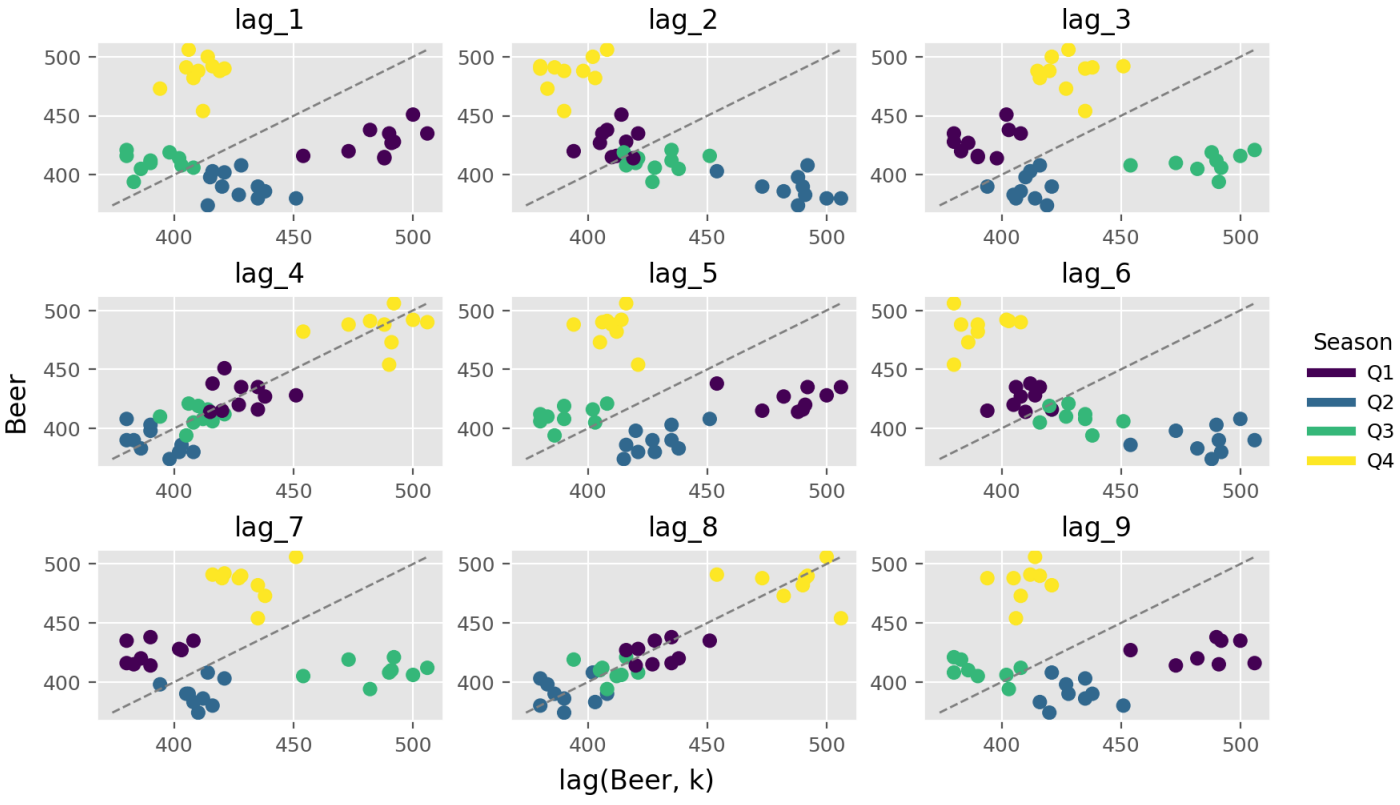


Figure 2.19: Lagged scatterplots for quarterly beer production.

Here the colours indicate the quarter of the variable on the vertical axis. The relationship is strongly positive at lags 4 and 8,

reflecting the strong seasonality in the data. The negative relationship seen for lags 2 and 6 occurs because peaks (in Q4) are plotted against troughs (in Q2)

2.8 Autocorrelation

Just as correlation measures the extent of a linear relationship between two variables, autocorrelation measures the linear relationship between *lagged values* of a time series.

There are several autocorrelation coefficients, corresponding to each panel in the lag plot. For example, r_{1} measures the relationship between y_{t} and y_{t-1}, r_{2} measures the relationship between y_{t} and y_{t-2}, and so on.

The value of r_{k} can be written as r_{k} = \frac{\sum\limits_{t=k+1}^T (y_{t}-\bar{y})(y_{t-k}-\bar{y})} {\sum\limits_{t=1}^T (y_{t}-\bar{y})^2}, where T is the length of the time series. The autocorrelation coefficients make up the *autocorrelation function* or ACF.

The autocorrelation coefficients for the beer production data can be computed using the `acf()` function.

```
acf_df = pd.DataFrame(
    {"Lag": range(10), "ACF": sm.tsa.acf(recent_production["y"], nlags=9,
        fft=False, bartlett_confint=False)}
).set_index("Lag")
acf_df[1:]
```

Lag	ACF
1	-0.052981
2	-0.758175
3	-0.026234
4	0.802205
5	-0.077471
6	-0.657451
7	0.001195
8	0.707254
9	-0.088756

The values in the `acf` column are r_1,\dots,r_9, corresponding to the nine scatterplots in Figure 2.19. We usually plot the ACF to see how the correlations change with the lag k. The plot is sometimes known as a *correlogram*.

```
recent_production["ds"] = pd.to_datetime(recent_production["ds"])
fig, ax = plt.subplots()
plot_acf(recent_production["y"], lags=16,
         ax=ax,
         zero=False, bartlett_confint=False, auto_ylims=True)

ax.set_title("Autocorrelation Function for Beer")
ax.set_xlabel("Lags")
ax.set_ylabel("ACF")

fig.show()
```

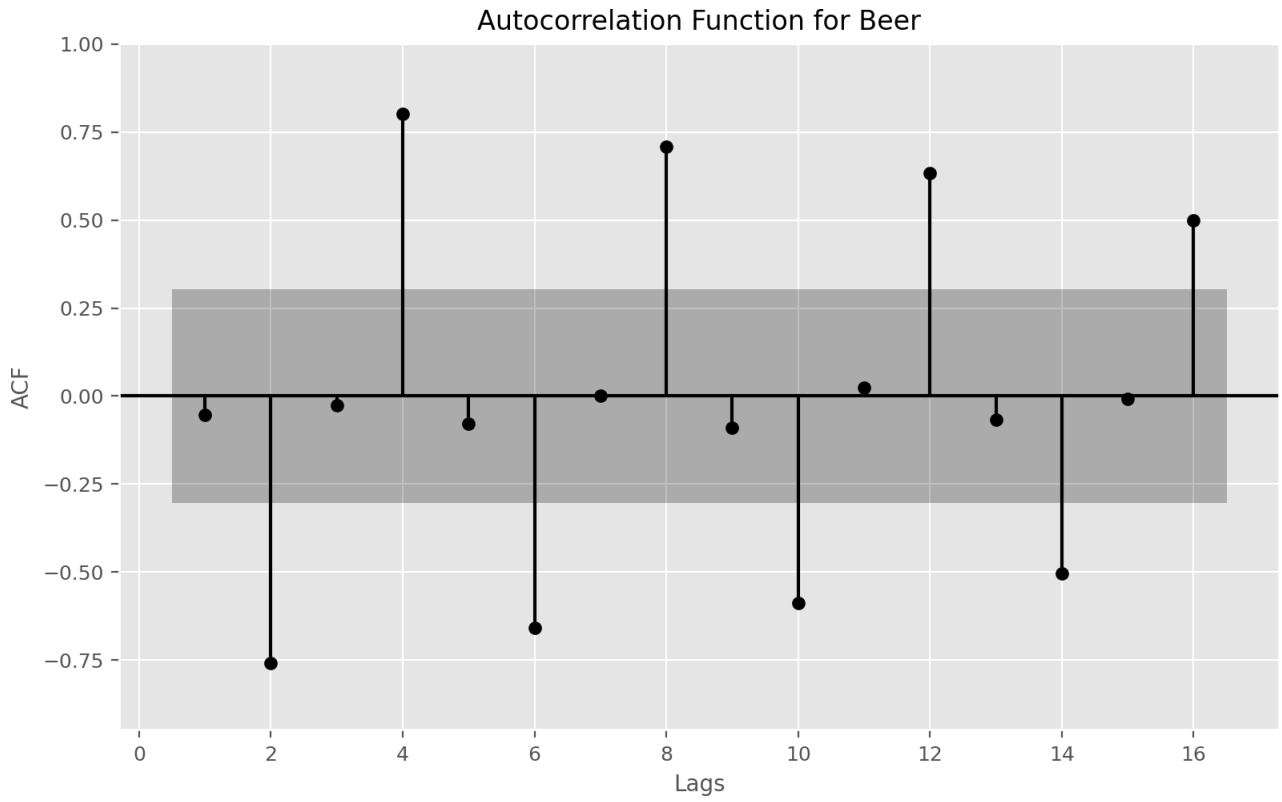


Figure 2.20: Autocorrelation function for quarterly beer production.

In this graph:

- r_{4} is higher than for the other lags. This is due to the seasonal pattern in the data: the peaks tend to be four quarters apart and the troughs tend to be four quarters apart.
- r_{2} is more negative than for the other lags because troughs tend to be two quarters behind peaks.
- The dashed blue lines indicate whether the correlations are significantly different from zero (as explained in Section 2.9).

Trend and seasonality in ACF plots

When data have a trend, the autocorrelations for small lags tend to be large and positive because observations nearby in time are also nearby in value. So the ACF of a trended time series tends to have positive values that slowly decrease as the lags increase.

When data are seasonal, the autocorrelations will be larger for the seasonal lags (at multiples of the seasonal period) than for other lags.

When data are both trended and seasonal, you see a combination of these effects. The `total_cost_df` data plotted in Figure 2.2 shows both trend and seasonality. Its ACF is shown in Figure 2.21. The slow decrease in the ACF as the lags increase is due to the trend, while the “scalloped” shape is due to the seasonality.

```
total_cost_df["Month"] = pd.to_datetime(total_cost_df["Month"])
fig, ax = plt.subplots()
plot_acf(total_cost_df["Cost"], lags=48, ax=ax,
         zero=False, bartlett_confint=False, auto_ylims=True)

ax.set_title("Australian antidiabetic drug sales")
ax.set_xlabel("Lags")
ax.set_ylabel("ACF")
ax.set_ylim(bottom=-0.3)

fig.show()
```

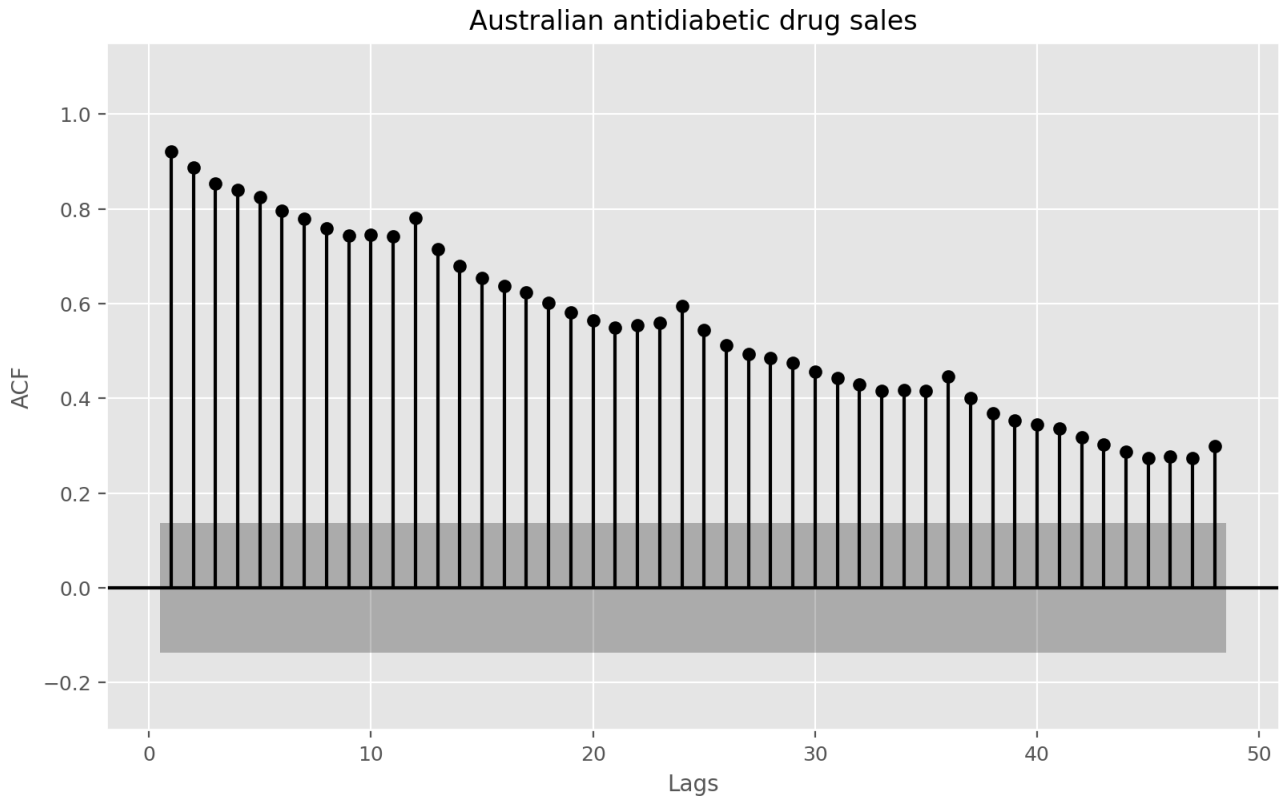


Figure 2.21: Autocorrelation function for monthly antidiabetic drug sales in Australia.

2.9 White noise

Time series that show no autocorrelation are called **white noise**. Figure 2.22 gives an example of a white noise series.

```
np.random.seed(30)
y = pd.DataFrame(
    {"wn": np.random.normal(0, 1, 50), "ds": np.arange(1, 51),
        "unique_id": "wn"}
)
plot_series(y, target_col="wn",
            xlabel="sample [1]",
            title = "White noise")
```

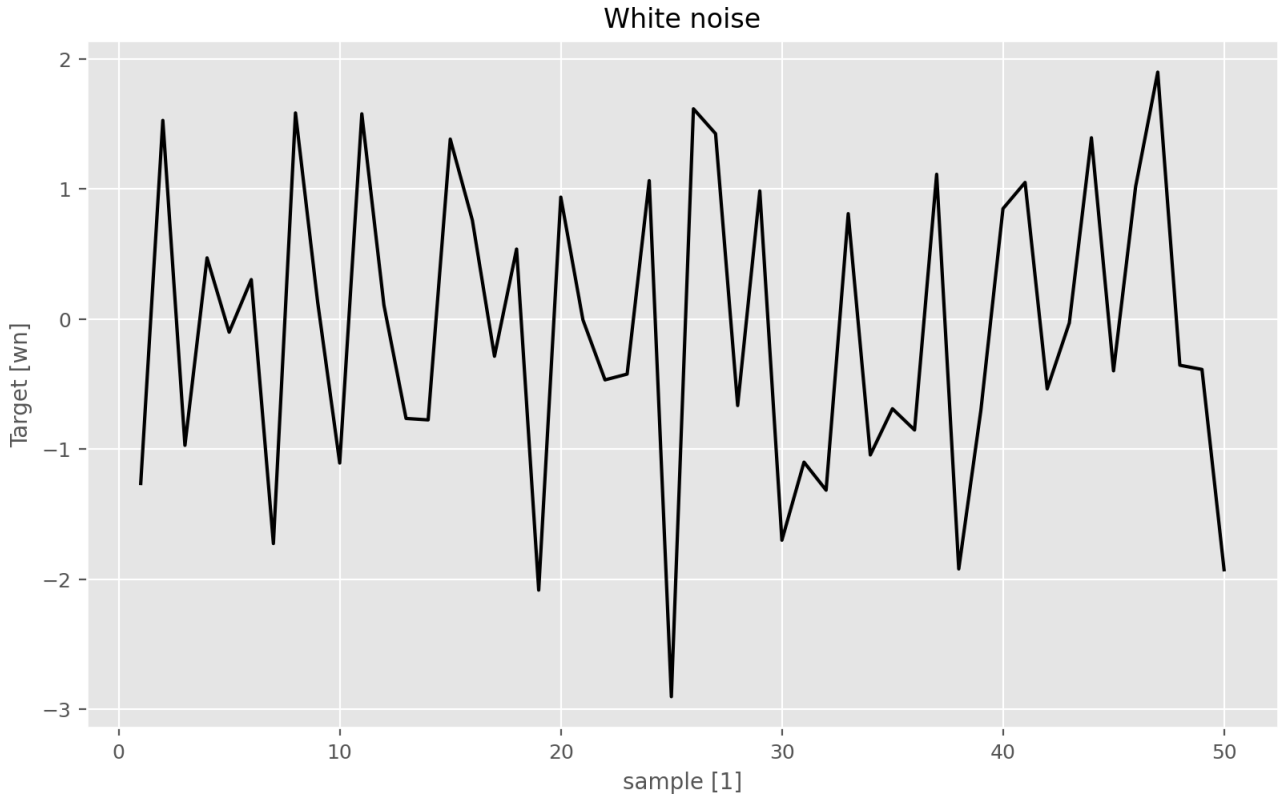


Figure 2.22: White noise series.

```
fig, ax = plt.subplots()
plot_acf(y["wn"], lags=16, ax=ax,
         zero=False, bartlett_confint=False,
         auto_ylims=True)
ax.set_title("White Noise")
ax.set_xlabel("Lag[1]")
ax.set_ylabel("ACF")
#ax.set_ylim(bottom=-0.3)
fig.show()
```

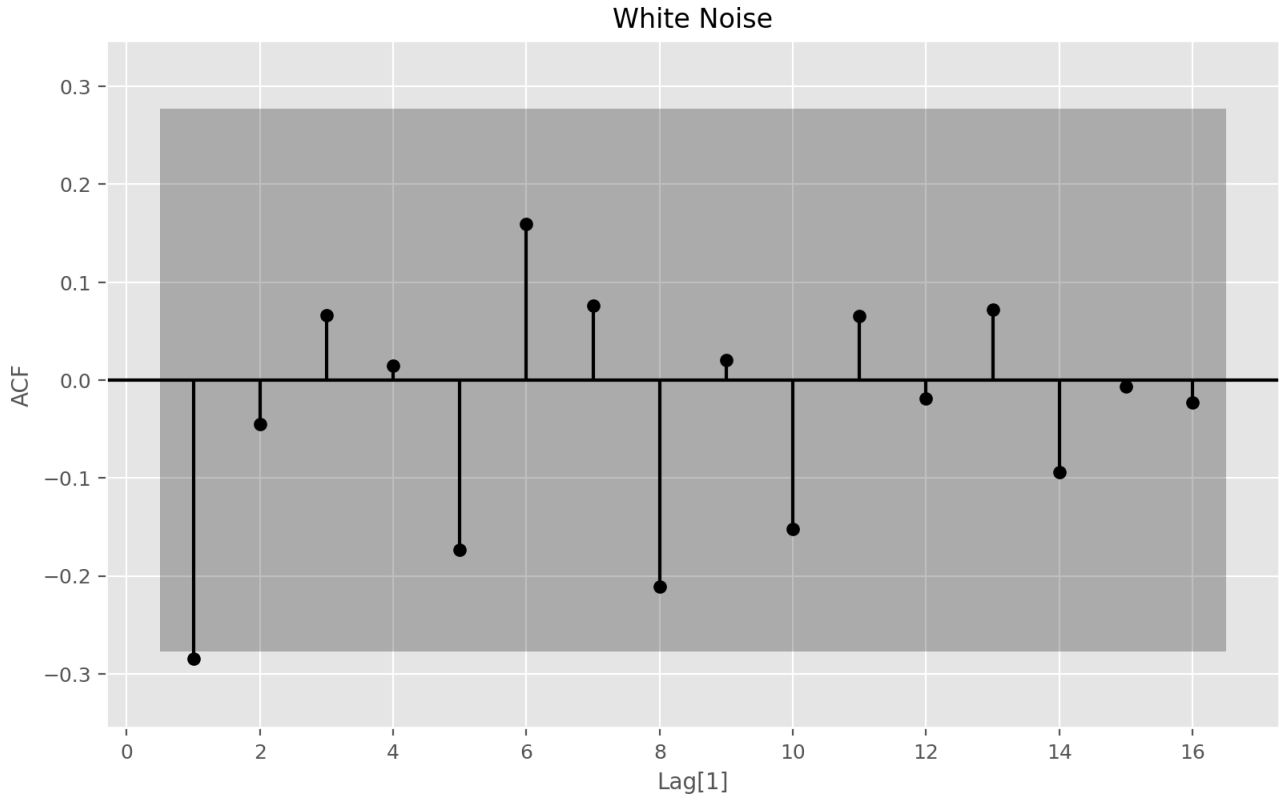


Figure 2.23: Autocorrelation function for the white noise series.

For white noise series, we expect each autocorrelation to be close to zero. Of course, they will not be exactly equal to zero as there is some random variation. For a white noise series, we expect 95% of the spikes in the ACF to lie within \pm 1.96/\sqrt{T} where T is the length of the time series. It is common to plot these bounds on a graph of the ACF (the blue dashed lines above). If one or more large spikes are outside these bounds, or if substantially more than 5% of spikes are outside these bounds, then the series is probably not white noise.

In this example, T=50 and so the bounds are at \pm 1.96/\sqrt{50} = 0.28. All of the autocorrelation coefficients lie within these limits, confirming that the data are white noise.

2.10 Exercises

1. Explore the following four time series: `Bricks` from `aus_production`, `Lynx` from `pelt`, `GOOG_Close` from `gafa_stock`, `Demand` from `vic_elec`.
    - Use `info()` to find out about the data in each series.
    - What is the time interval of each series?
    - Use `plot_series()` to produce a time plot of each series.
    - For the last plot, modify the axis labels and title.
2. Use `query()` to find what days corresponded to the peak closing price for each of the four stocks in `gafa_stock`.
3. Download the file `tute1.csv` from the book website, open it in Excel (or some other spreadsheet application), and review its contents. You should find four columns of information. Columns B through D each contain a quarterly series, labelled Sales, AdBudget and GDP. Sales contains the quarterly sales for a small company over the period 1981-2005. AdBudget is the advertising budget and GDP is the gross domestic product. All series have been adjusted for inflation.
    a. You can read the data into Python with the following script:

    ```
    tute1 = pd.read_csv('../data/tute1.csv')
    tute1.head()
    ```

    b. Convert the data to time series

    ```
    tute1['ds'] = pd.to_datetime(tute1['ds'])
    ```

c. Construct time series plots of each of the three series

```
plot_series(tute1)
```

4. The `us_total.csv` contains data on the demand for natural gas in the US.

   a. Download `us_total.csv` from the book website read in the csv file using `pd.read_csv()`.
   b. Create a dataframe from `us_total` with year as the index.
   c. Plot the annual natural gas consumption by state for the New England area (comprising the states of Maine, Vermont, New Hampshire, Massachusetts, Connecticut and Rhode Island).

5. a. Download `tourism.xlsx` from the book website and read in it using `pd.read_excel()`.
   b. Create a dataframe using `tourism.xlsx`.
   c. Find what combination of `Region` and `Purpose` had the maximum number of overnight trips on average.
   d. Create a new dataframe which combines the Purposes and Regions, and just has total trips by State.

6. The `aus_arrivals` data set comprises quarterly international arrivals to Australia from Japan, New Zealand, UK and the US.

   - Use `plot_series()` to visualize the data.
   - Use `seaborn` or `matplotlib` to create seasonal and subseries plots to compare the differences between the arrivals from these four countries and identify any unusual observations.

7. Monthly Australian retail data is provided in `aus_retail`. Select one of the time series as follows (but choose your own seed value):

```
np.random.seed(12345678)
random_series_id = \
    np.random.choice(aus_retail['Series ID'].unique(), 1)[0]
myseries = aus_retail.query('`Series ID` == @random_series_id')
```

   Explore your chosen retail time series using the following functions:

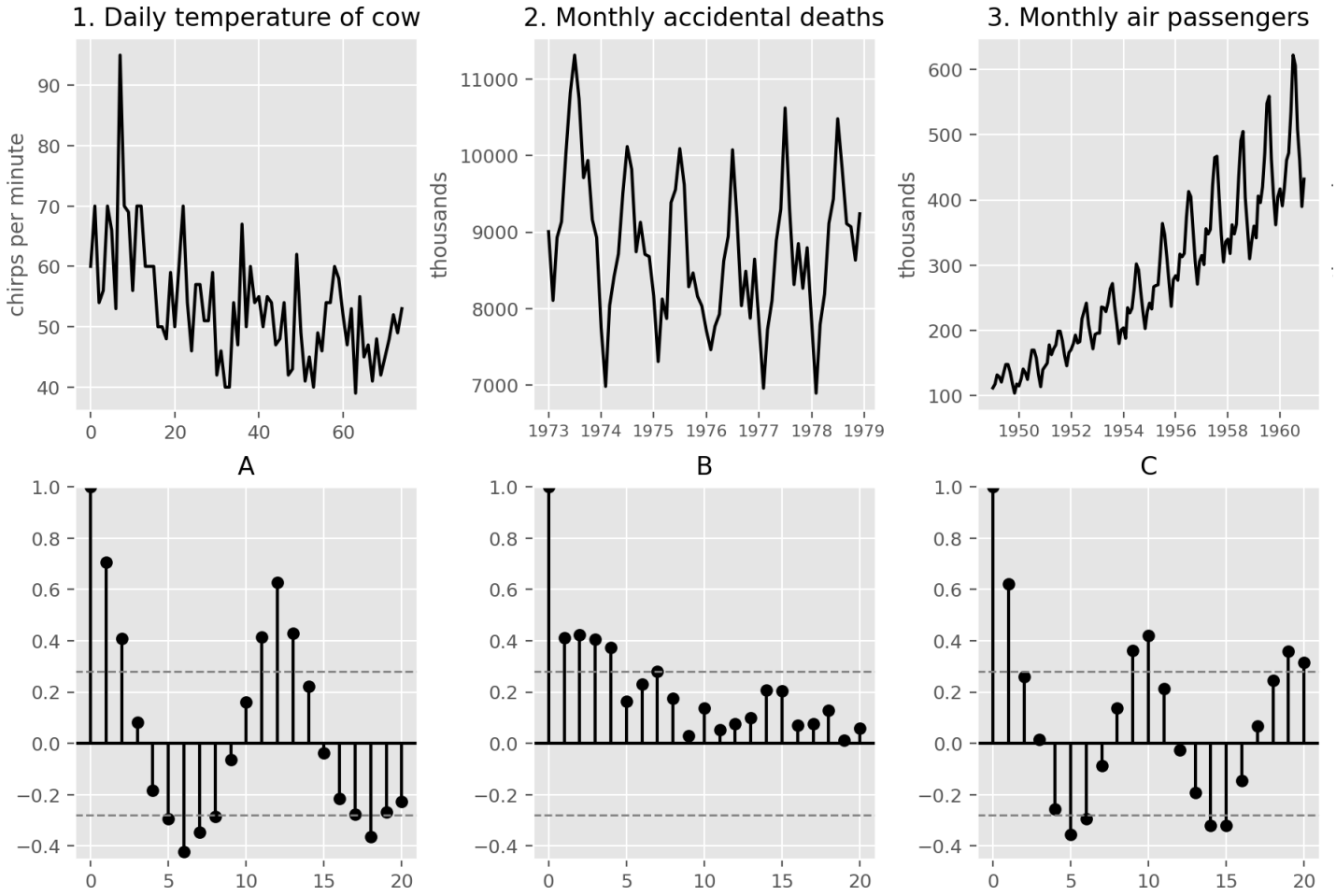
   `plot_series()`, `seasonal_decompose()`, `lag_plot()`, `plot_acf()`

   Can you spot any seasonality, cyclicity and trend? What do you learn about the series?

8. Use the following graphics functions: `plot_series()`, `seasonal_decompose()`, `lag_plot()`, `plot_acf()` and explore features from the following time series: “Total Private” `Employed` from `us_employment`, `Bricks` from `aus_production`, `Hare` from `pelt`, “H02” `Cost` from `PBS`, and `Barrels` from `us_gasoline`.

   - Can you spot any seasonality, cyclicity and trend?
   - What do you learn about the series?
   - What can you say about the seasonal patterns?
   - Can you identify any unusual years?

9. The following time plots and ACF plots correspond to four different time series. Your task is to match each time plot in the first row with one of the ACF plots in the second row.



10. The `aus_livestock` data contains the monthly total number of pigs slaughtered in Victoria, Australia, from Jul 1972 to Dec 2018. Use `query()` to extract pig slaughters in Victoria between 1990 and 1995. Use `plot_series()` and `plot_acf()` for this data. How do they differ from white noise? If a longer period of data is used, what difference does it make to the ACF?

11. a. Use the following code to compute the daily changes in Google closing stock prices.

```
dgoog =  stock.query('unique_id == "GOOG_Close" & ds >= "2018"')
dgoog['trading_day'] = np.arange(1, len(dgoog) + 1)
dgoog['diff'] = dgoog['y'].diff()
dgoog.set_index('trading_day', inplace=True)
```

    b. Why was it necessary to re-index the dataframe?

    c. Plot these differences and their ACF.

    d. Do the changes in the stock prices look like white noise?

2.11 Further reading

- Cleveland (1993) is a classic book on the principles of visualisation for data analysis. While it is more than 30 years old, the ideas are timeless.
- Unwin (2015) is a modern introduction to graphical data analysis using R. It does not have much information on time series graphics, but plenty of excellent general advice on using graphics for data analysis.

2.12 Used modules and classes

StatsForecast

- `StatsForecast` class - Core forecasting engine
- `AutoETS` model - For automatic exponential smoothing

UtilsForecast

- `plot_series` utility - For creating time series visualizations

← Chapter 1 Getting started

Chapter 3 Time series decomposition →