Lecture 1: Introduction to Time Series

Contents

UBC Master of Data Science

DSCI 574
Spatial & Temporal Models

Credit: Marvel

Lecture Outline

Imports

```
import calendar
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
from statsmodels.tsa.tsatools import detrend
from statsmodels.tsa.forecasting.stl import STLForecast
from statsmodels.tsa.seasonal import seasonal_decompose, STL
from statsmodels.graphics.tsaplots import plot_acf, seasonal_plot, month_plot
import matplotlib as mpl
import matplotlib.pyplot as plt
import plotly.express as px
plt.style.use("ggplot")
plt.rcParams.update({"font.size": 14, "axes.labelweight": "bold", "figure.figs
from scripts.plotting import *
```

Lecture Learning Objectives

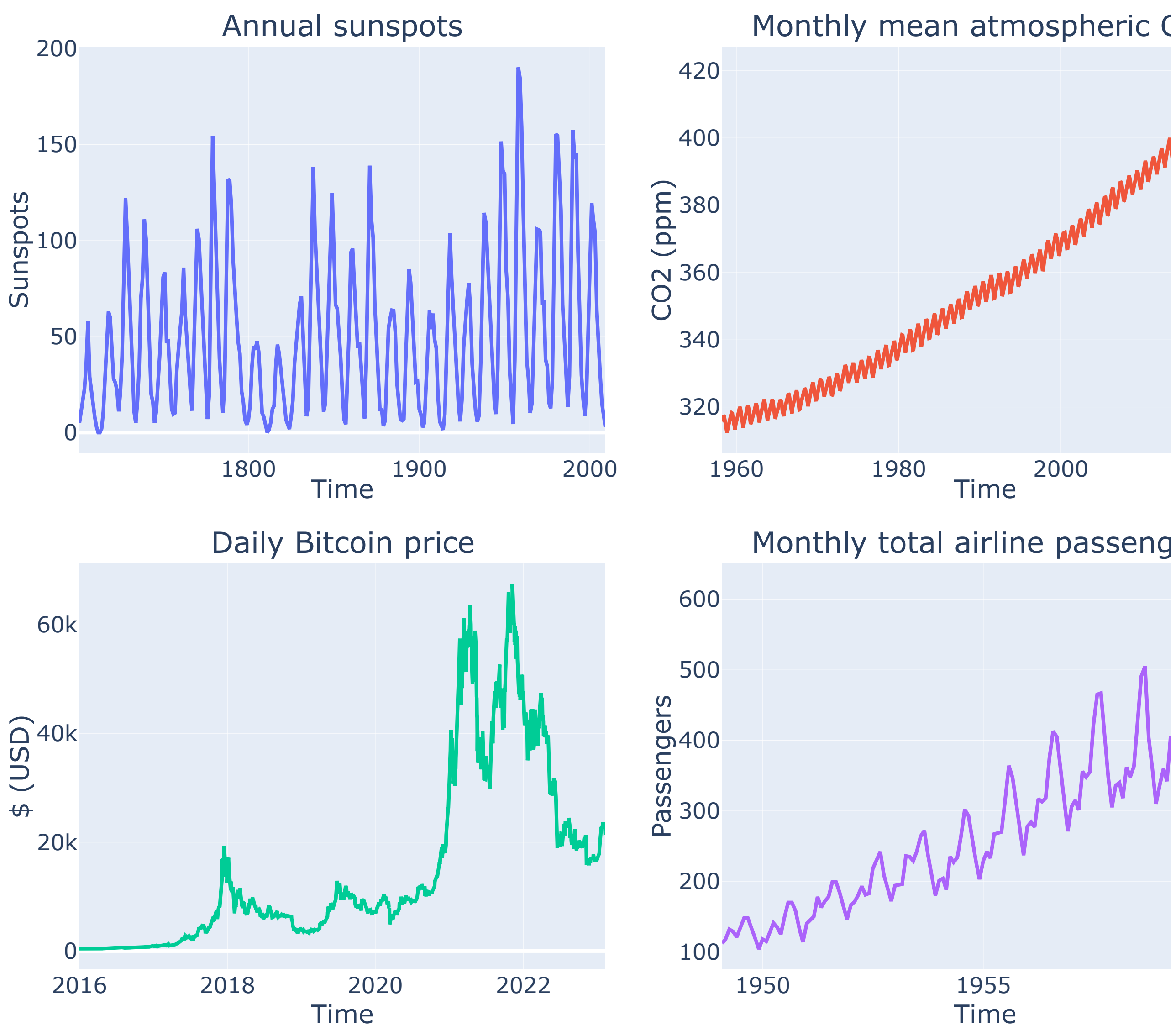
- Define and explain terminology used to describe time series - including “trend”, “seasonal effects”, “cyclical effects”, and “white noise”.
- Explain what is the (sample) autocorrelation function and a correlogram.
- Describe the behaviour of a correlogram for series that alternate (vary above and below the mean), have a trend or show seasonal fluctuations.
- Describe the difference between and additive and multiplicative series.
- Estimate the trend-cycle component of a series by curve-fitting or applying a moving average smoother (centering if necessary) using `statsmodels` and `pandas`.
- Decompose a time series into trend-cycle, seasonal, and remainder components using the `statsmodels` functions `seasonal_decompose` and `STL`.

1. What is a time series?

- A time series is simply a collection of observations recorded sequentially in time
- Time series are everywhere:
  - Physical sciences (meteorology, hydrology, geophysics, ecology, etc.)
  - Economics, finance, commerce
  - Engineering (electricity consumption, concrete curing time, etc.)

- Social science and epidemiology
- Sport science
- Etc.

```
plot_examples()
```



> Note: all the datasets used in this course have been put together via web scraping or using some API. We've included the code for obtaining all datasets in Appendix A so feel free to check that out if you're interested.

1.1 Data format in time series

- **Univariate**: A time series that consists of a single variable recorded sequentially over time (e.g., bike sale over time from a single store)

store	time	unit_sale
A	Jan	120
A	Feb	122
A	Mar	123

- **Multivariate**: Consists of multiple variables recorded sequentially over time (e.g., bike sale and profit over time from a single store)

store	time	unit_sale	profit
A	Jan	120	$12,000
A	Feb	122	$13,000
A	Mar	123	$14,000

- **Hierachical**: Consists of multiple variables from multiple observations, nested within the larger group categories, recorded sequentially over time (e.g., bike sale and profit from multiple stores over time)

store	time	unit_sale	profit
A	Jan	120	$12,000
A	Feb	122	$13,000
A	Mar	123	$14,000
B	Jan	120	$11,000
B	Feb	222	$33,000
B	Mar	143	$24,000

Note

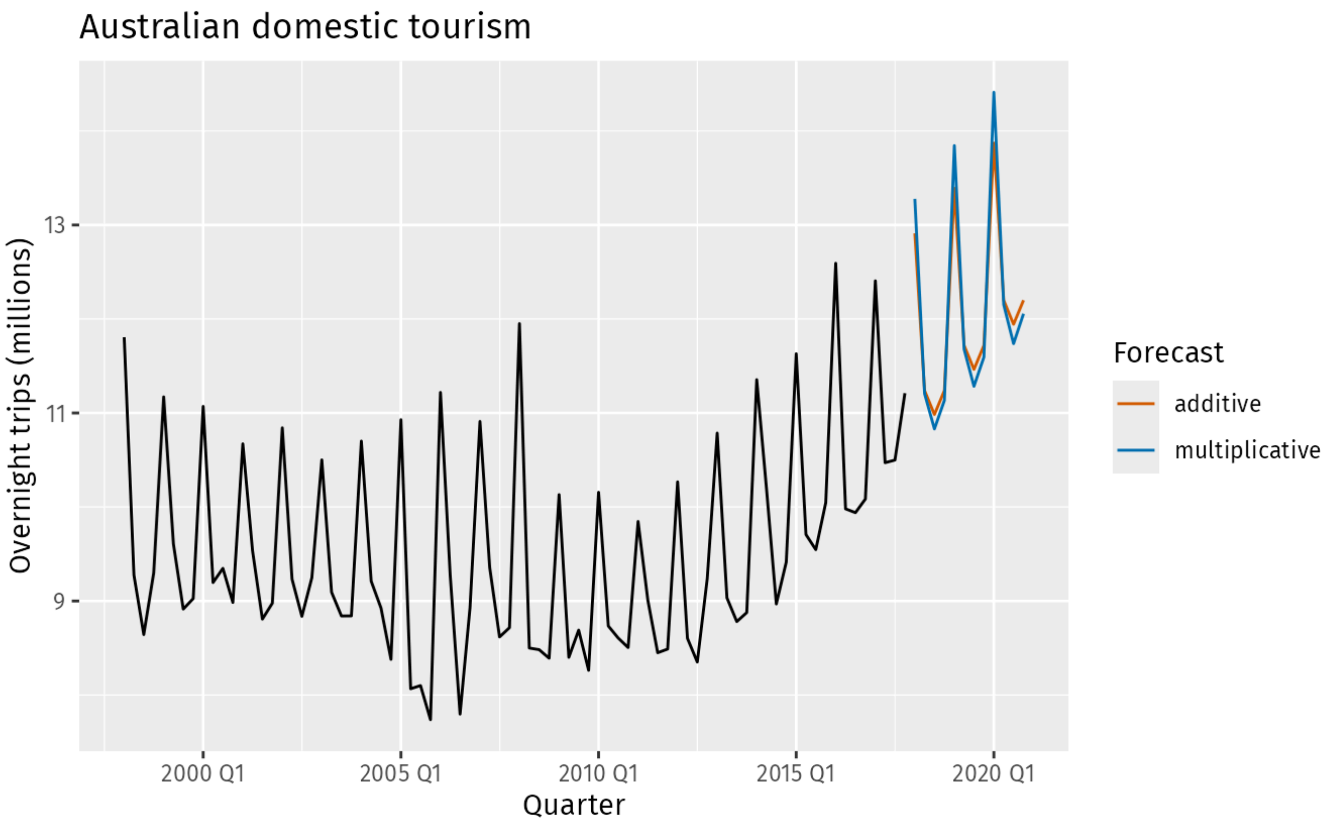
Observations in a series may be evenly spaced (**regular time series**) or unevenly space (**irregular time series**)

Dealing with irregular time series is tricky and we'll be focussing on regular time series in this course. If you encounter an irregular time series, it is typical to aggregate it to a regular interval, and/or to impute missing values. We'll explore both of those techniques in this course.

1.2 Different tasks in time series analysis

Prediction/forecasting (supervised)

- Example: Predicting the number of tourists in the upcoming quarters

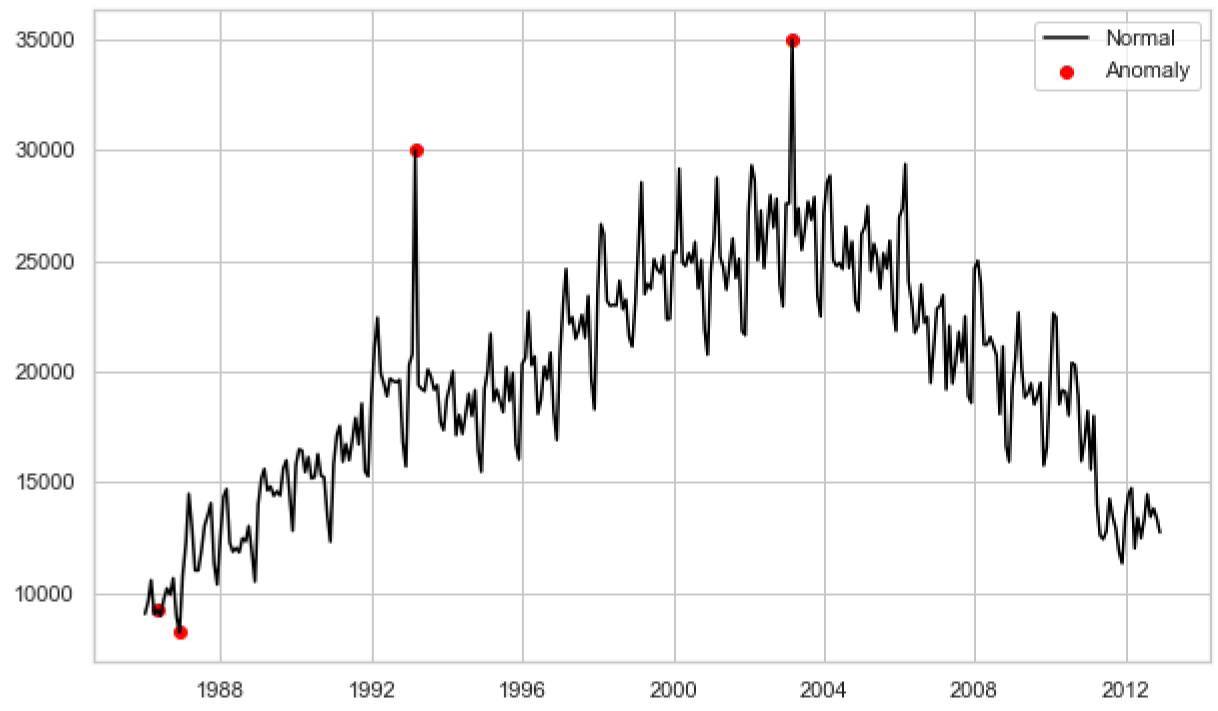


- **Approach**: Statistical models (exponential smoothing, ARIMA, etc..) or ML/DL models (CNN,RNN,LightGBM, etc...)

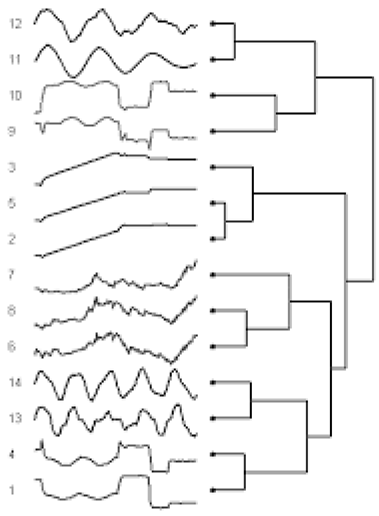
This will be the main focus in this course

Clustering/Anomaly detection (unsupervised)

- Example 1: Detect earthquakes



- Example 2: Detect groups of similar time series



Credit: https://robjhyndman.com/papers/wang.pdf

Approach: Start with some dimensionality reduction/transformation, compute similarity measures (euclidean, DTW), and apply unsupervised learning models (k-means, DBSCAN, etc.)

Here's a [good read](#) on unsupervised time-series if you are interested

1.3 Why predicting the future is difficult?

What makes something easy/difficult to forecast?

- Weather tomorrow
- Weather a month from now
- Stock price in the next 3 days
- The number of COVID cases at the beginning of the pandemic
- Stock price during FED's announcement
- Stock price of a company that will be affected by the new congress's bill tomorrow
- DOGE coin's price after Elon Musk's tweet! #tothemoon

Factors that affect forecasting

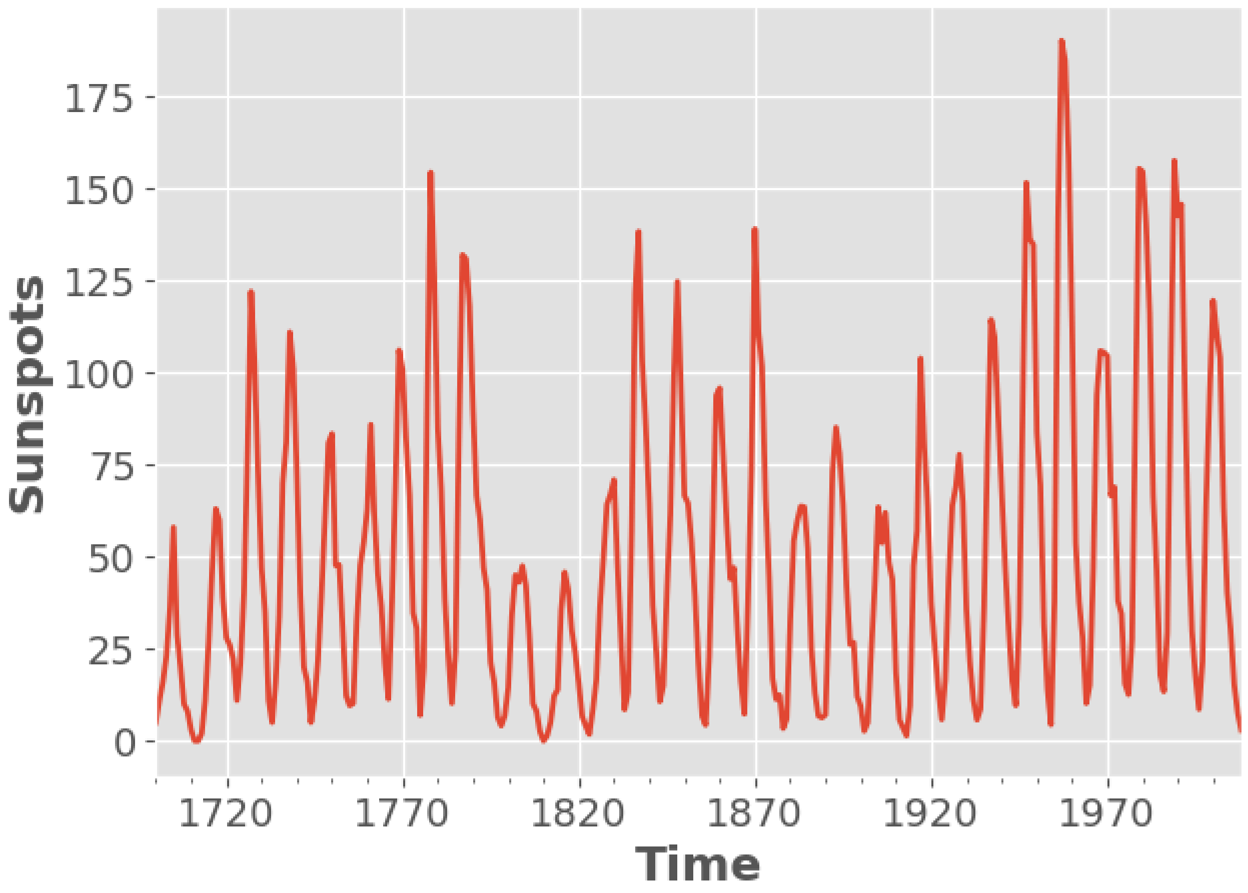
- How much (historical) data we have
- How far into the future do we forecast
- How similar historical patterns are compared to the future
- How 'stable' the environment is
- How much domain knowledge/information you possess

2. Features of a time series

2.1. Visualizing

- The first step in any time series analysis should be a time plot! Which is: time on the x-axis, and the series values on the y-axis.
- We'll use `pandas` for plotting our time series data in this lecture because it is quick and easy and a good skill to know
- Let's take a look at our dataset of annual sunspots:

```python
sun = pd.read_csv("data/sunspots.csv", index_col=0, parse_dates=True)
sun.plot.line(xlabel="Time", ylabel="Sunspots", legend=False);
```



2.2. Temporal dependence

- What makes time series different to data we've seen previously is that they include temporal dependence - observations close in time tend to be correlated
- That means that the value of the series at time t often depends in some way on past values, y_{t-1}, y_{t-2}, y_{t-3}, etc.
- We can quantify this dependence by looking at the correlation of a time series with "lagged" values of itself. We call this **autocorrelation**
- We can easily "lag" (or "shift") a time series in Pandas with the `.shift()` method:

```
sun["sunspots (lag=1)"] = sun["sunspots"].shift(1)
sun
```

	sunspots	sunspots (lag=1)
year		
1700-12-31	5.0	NaN
1701-12-31	11.0	5.0
1702-12-31	16.0	11.0
1703-12-31	23.0	16.0
1704-12-31	36.0	23.0
...	...	...
2004-12-31	40.4	63.7
2005-12-31	29.8	40.4
2006-12-31	15.2	29.8
2007-12-31	7.5	15.2
2008-12-31	2.9	7.5

309 rows × 2 columns

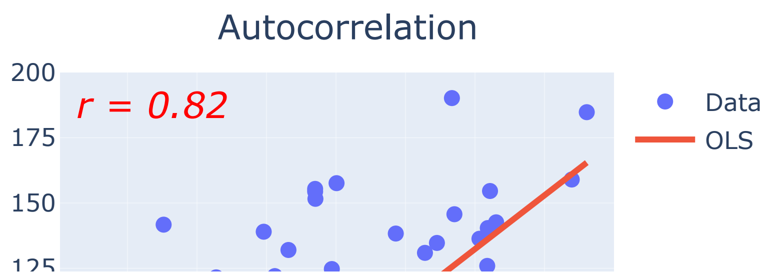
- The autocorrelation between `sunspots` and `sunspots at lag 1` is:

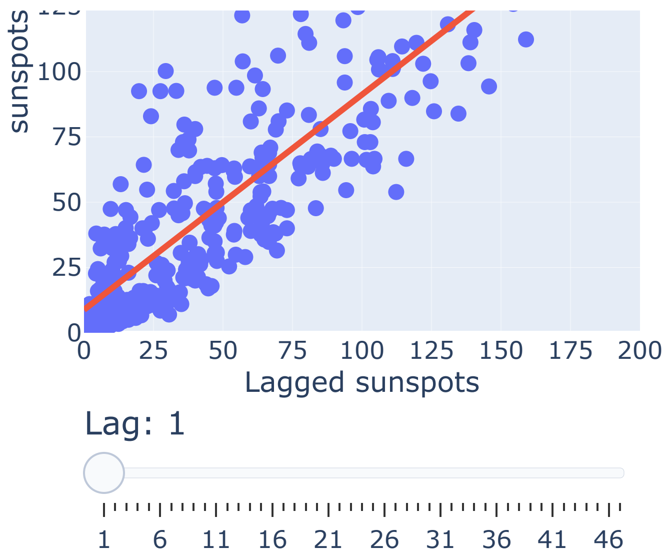
```
sun["sunspots"].corr(sun["sunspots (lag=1)"])
```

```
0.823628883717728
```

- Let's make an interactive scatter plot of lagged values and their autocorrelation for different lags:

```
plot_autocorrelation(sun, label="sunspots", dlag=1, max_lag=48)
```



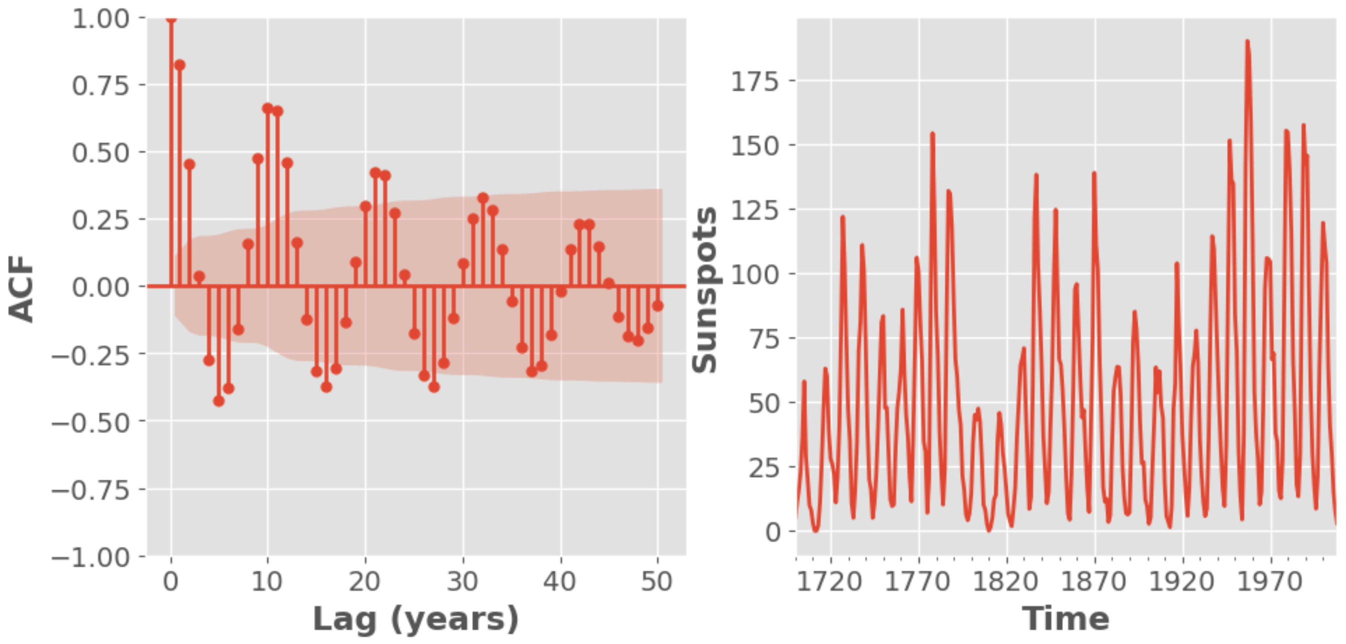


- Autocorrelation is positive at lags of about 10-11 (the solar cycle!) and negative at lags of 5-6.
- Autocorrelation also reduces towards 0 as the lag increases which make intuitive sense.

Correlogram

- We have a simpler plot to visualise all this information called a **correlogram**. A correlogram plots the autocorrelation function (ACF) on the y-axis and lags on the x-axis (we call it the ACF because it is a *function* of lag)
- We can make one easily using `statsmodels.graphics.tsaplots.plot_acf()`:

```
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(11, 5))

plot_acf(sun["sunspots"], lags=50, title=None,ax=ax1)
ax1.set_xlabel("Lag (years)")
ax1.set_ylabel('ACF')
sun.sunspots.plot.line(xlabel="Time", ylabel="Sunspots", legend=False,ax=ax2);
```



- Now we can clearly see that ~11 year cycle! We see positive correlation at 11 year periods and negative at 5-6 years
- correlations reducing as lags increase
- The red shading indicates whether the correlations are significantly different from zero. They are calculated using:

$$CI = \pm z_{1-\alpha/2} SE(r_h)$$

- Where r_h is the estimated autocorrelation at lag h, $z_{1-\alpha/2}$ is usually taken as 1.96 (the 95% quantile for a normal distribution)

Side note on SE's formula…

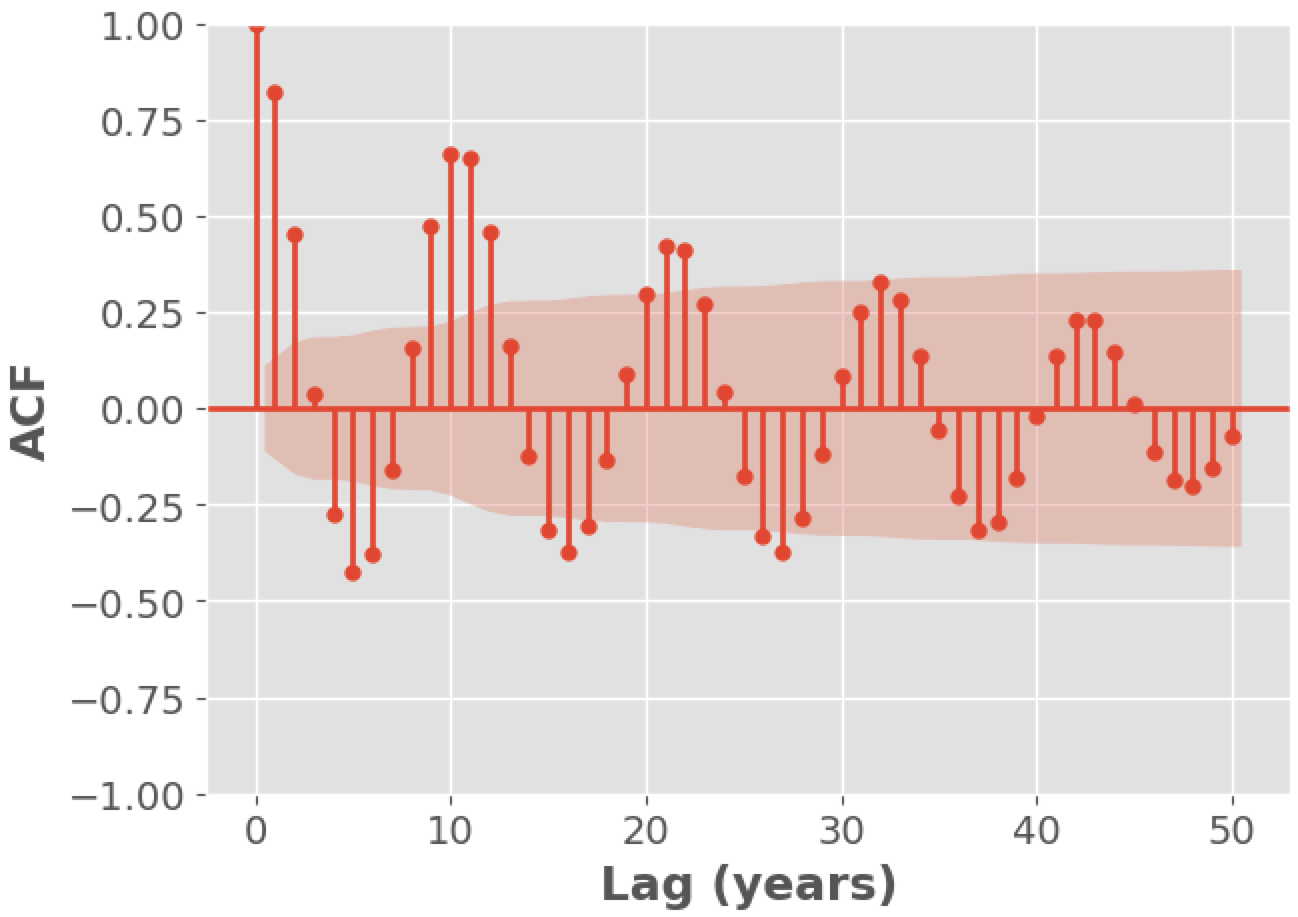
- The standard error is typically calculated using one of two methods:
  1. Assuming a white noise sampling distribution: $SE(r_h) = \frac{1}{\sqrt{T}}$
  2. Bartlett's formula: $SE(r_h) = \sqrt{\frac{1+2\sum_{i=1}^{h-1} r_i^2}{T}}$

The formula used for standard error depends upon the situation.

- If the autocorrelations are being used for residuals diagnostics as part of the ARIMA routine, the standard errors are determined assuming the residuals are white noise (1).
- If the autocorrelations are being used as part of the ACF of the raw data as a model selection tool, then we should use the Bartlett's formula (2).

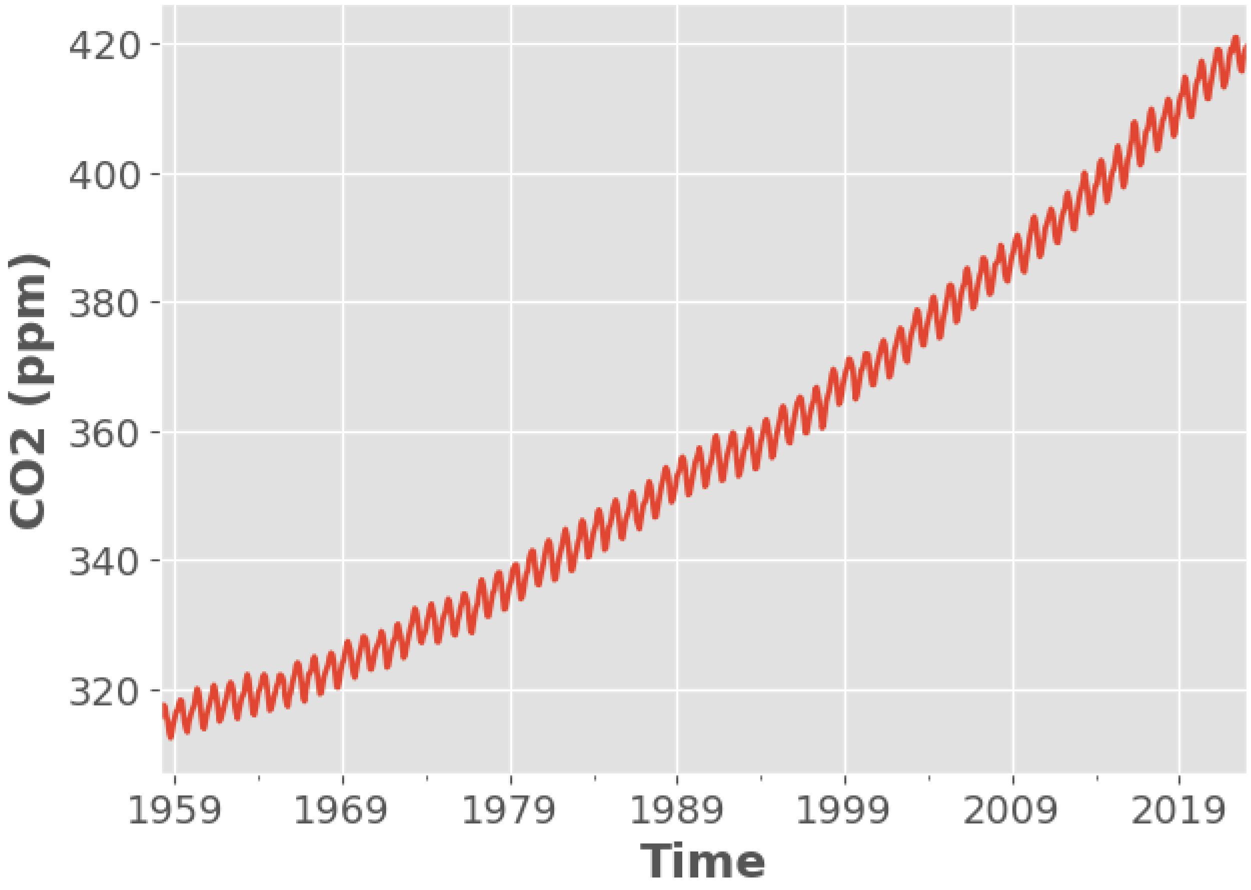
You can specifiy the formula to compute the SE via the `bartlett_confint` argument

```
fig = plot_acf(sun["sunspots"], lags=50, title=None, bartlett_confint=True)
plt.ylabel("ACF")
plt.xlabel("Lag (years)");
```



- Anyway, we'll revisit those in later lectures. For now, let's revisit our monthly atmospheric CO2 dataset and look at plotting a correlogram.

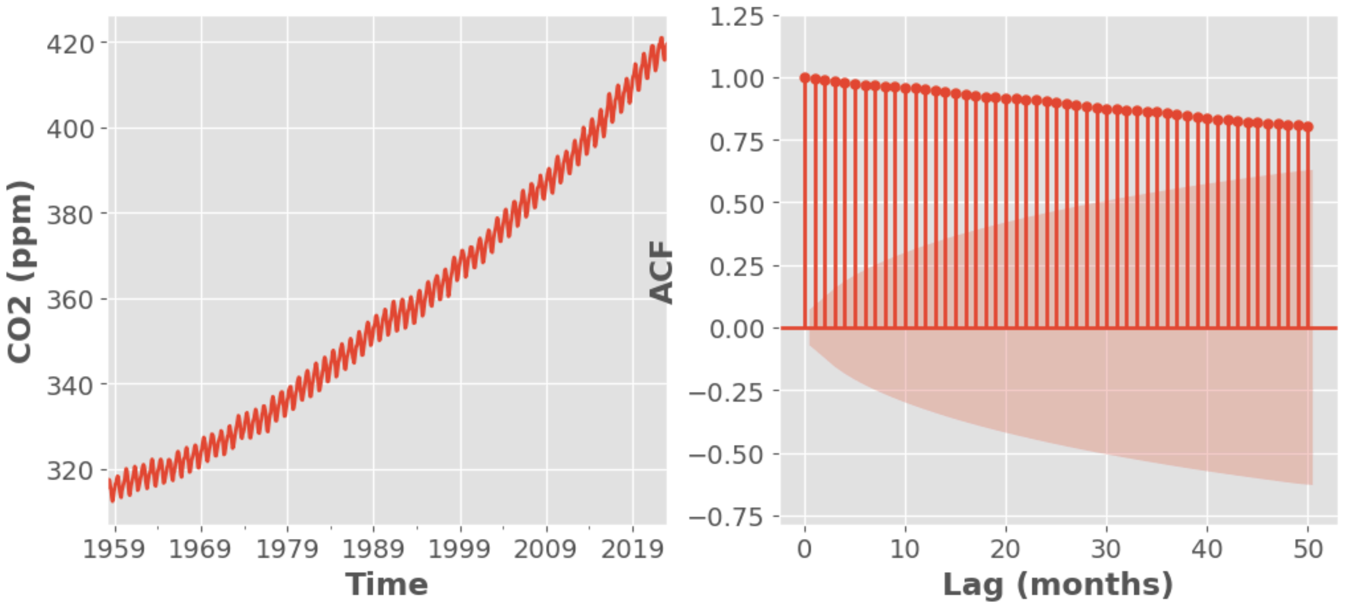
```
co2 = pd.read_csv("data/co2.csv", index_col=0, parse_dates=True)
co2.plot.line(xlabel="Time", ylabel="CO2 (ppm)", legend=False);
```



What do you notice about the CO2 time-series?

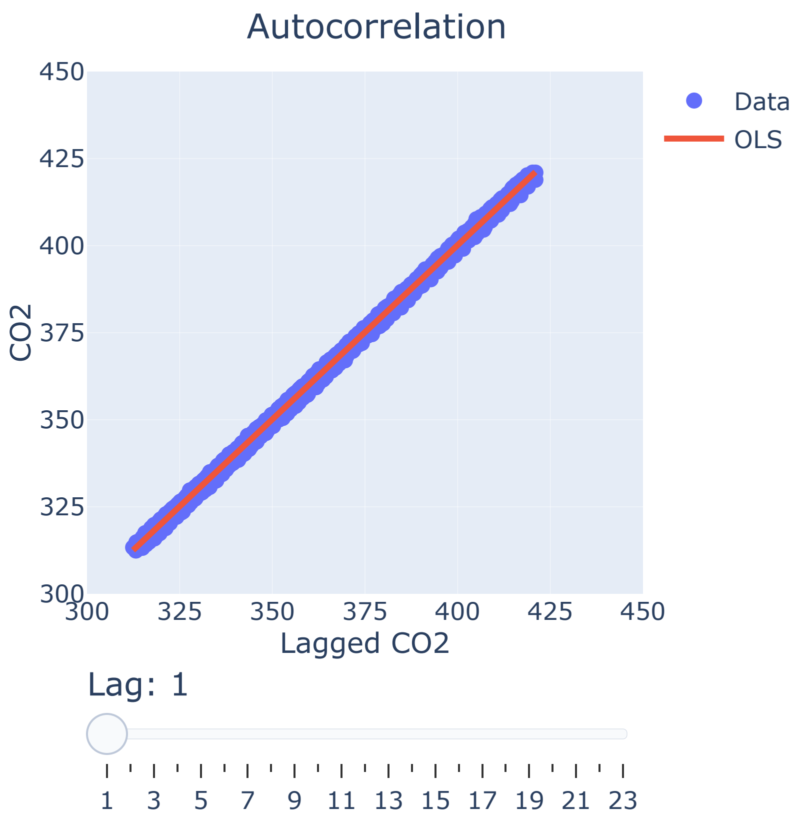
- Are the any seasonality patterns?
- Is there any trend?

```
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

plot_acf(co2, lags=50, title=None,auto_ylims=True,ax=ax2)
ax2.set_xlabel("Lag (months)")
ax2.set_ylabel('ACF')
#plt.ylim(0.75,1.05) #Did you notice the ACF oscillate every 12 months?
co2.plot.line(xlabel="Time", ylabel="CO2 (ppm)", legend=False,ax=ax1);
```



- Um... what's going on here? Why does this ACF plot of C02 look different from the sunspots data?
- Well there's a strong trend in our data. When data have a trend, the autocorrelations for smaller lags tend to be large because observations nearby in time are very similar (because of the trend):

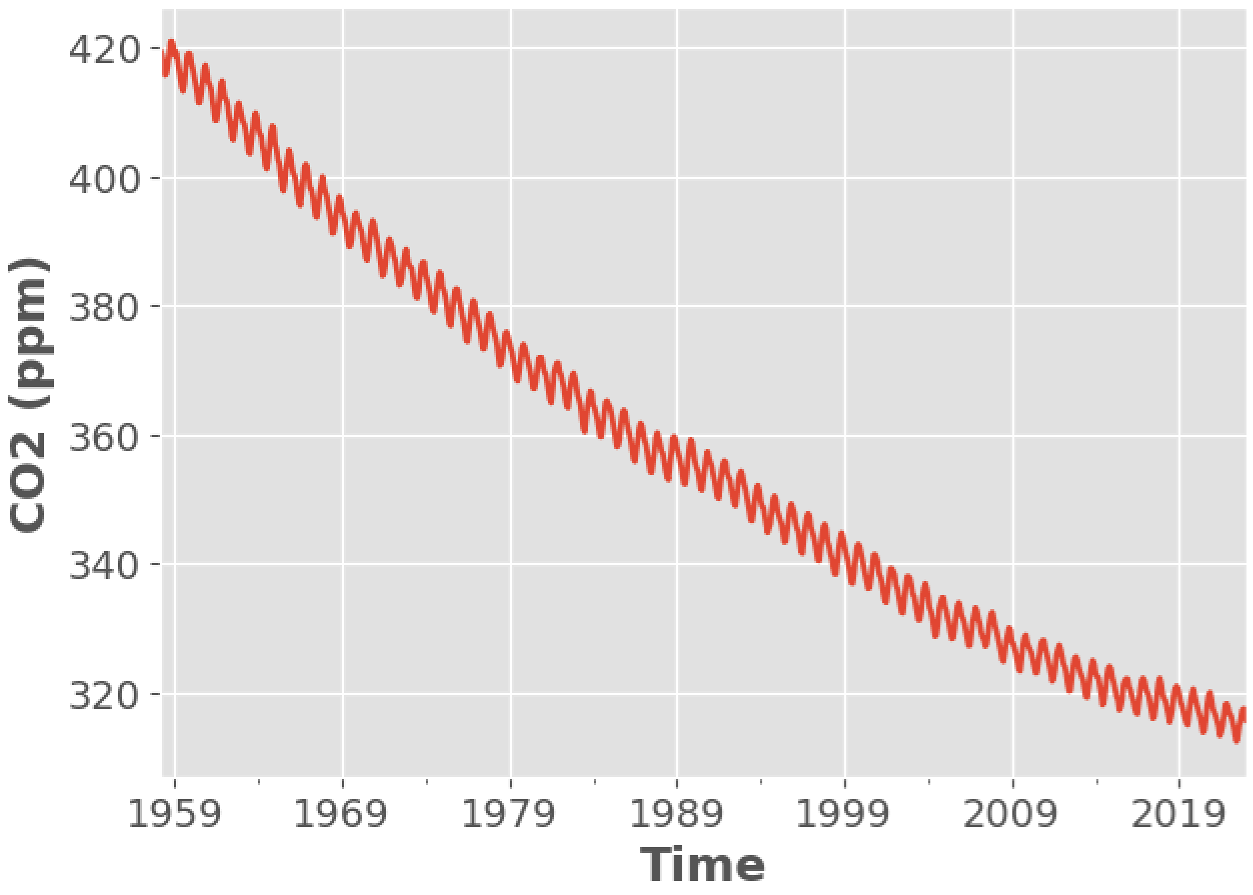
```
plot_autocorrelation(co2, label="CO2", dlag=1, max_lag=24, axis_limits=[300, 4
```



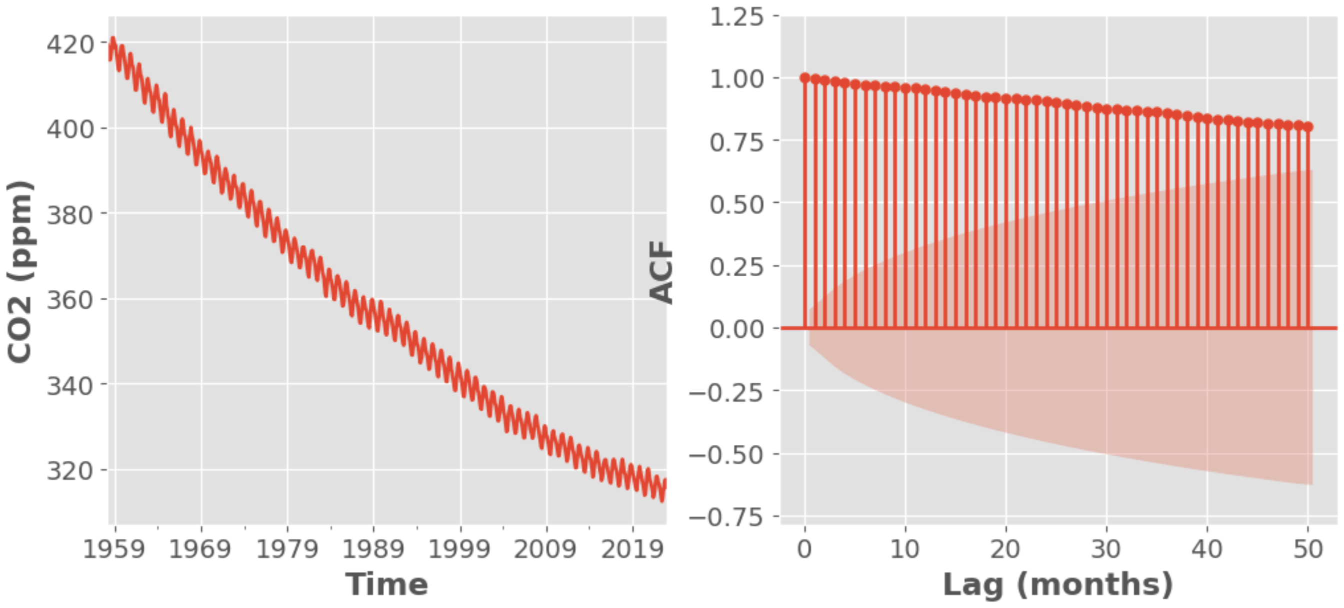
- Notice the strong correlation even at large lags in the interactive plot above

Let’s reverse things- how will this affect the ACF?

```
co2 = pd.read_csv("data/co2.csv", index_col=0, parse_dates=True)
co2['CO2'] = co2.CO2.values[::-1]
co2.plot.line(xlabel="Time", ylabel="CO2 (ppm)", legend=False);
```



```
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

plot_acf(co2, lags=50, title=None,auto_ylims=True,ax=ax2)
ax2.set_xlabel("Lag (months)")
ax2.set_ylabel('ACF')
co2.plot.line(xlabel="Time", ylabel="CO2 (ppm)", legend=False,ax=ax1);
```



- We'll explore this notion of trends more later, but for now, some key observations abouts correlograms:

1. The ACF will almost always decay with the lag (observations farther apart in time are less correlated)
2. If a series alternates (i.e., consecutive values tend to be on the opposite sides of the mean, like our sunspots data), then the ACF alternates too.
3. If a series has seasonal or cyclical fluctuations, the ACF will oscillate at the same frequency.
4. If the series has a trend, the ACF will have a very slow decay due to high correlation of the consecutive values (which tend to lie on the same side of the mean)
5. In general, experience is required to glean much from an ACF plot. We will use the correlogram as a model selection tool later in the course.

2.2. Time Series Patterns

- There are 3 main patterns of a time series you should be aware of:
    1. **Trend**: long term increases or decreases in the series.
    2. **Seasonality**: regular variation in the series at some fixed interval, e.g., month, day of week, time of day, etc.

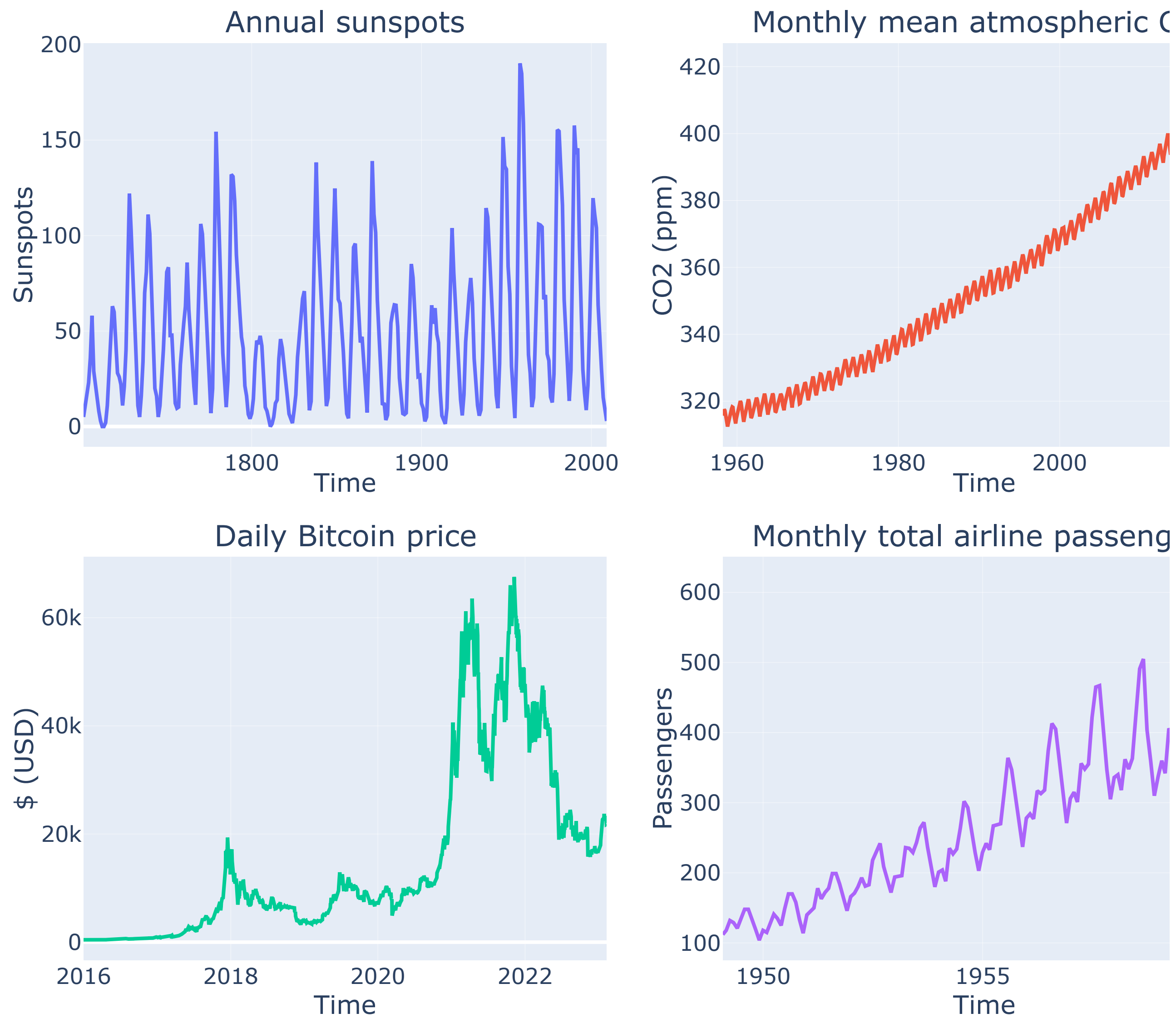
3. **Cyclicity**: variations in the series that repeat with some regularity but of unknown and changing period.

In-Class Exercise

From the time plots of the four examples previously plotted, discuss the **trend**, **seasonality** and **cyclicity** in each with a partner.

Be prepared to share your findings with the class!

```
plot_examples()
```



- So what patterns do we see above?

1. **Sunspots**: no apparent trend, but strong seasonality with a period 11 years (this is known as the "solar cycle")
2. **CO2**: strong positive trend, seasonal pattern with 12 month period
3. **Bitcoin**: seems to have an irregular, positive trend but if we had a longer series we might see that this is not a trend but part of some longer cyclic pattern. No obvious seasonality but there could potentially be some weekly pattern - we'll see how to investigate this more later.
4. **Airline passengers**: increasing trend and a seasonal pattern of 12 months. Note in particular that variation in the seasonal pattern increases with the value of the series.

iClicker Question

Which of the following statements is true regarding seasonality vs. cyclicity?

A. Seasonal behaviour usually occurs over a larger timeframe than cyclical behaviour, but at a fixed frequency

B. Cyclical behaviour usually occurs over a larger timeframe than seasons, and not at a fixed frequency

C. Seasonal behaviour usually occurs over a smaller timeframe than cyclical behaviour, but not at a fixed frequency

D. It is impossible to have cyclical behaviour shorter than seasonal behaviour

2.3. White Noise

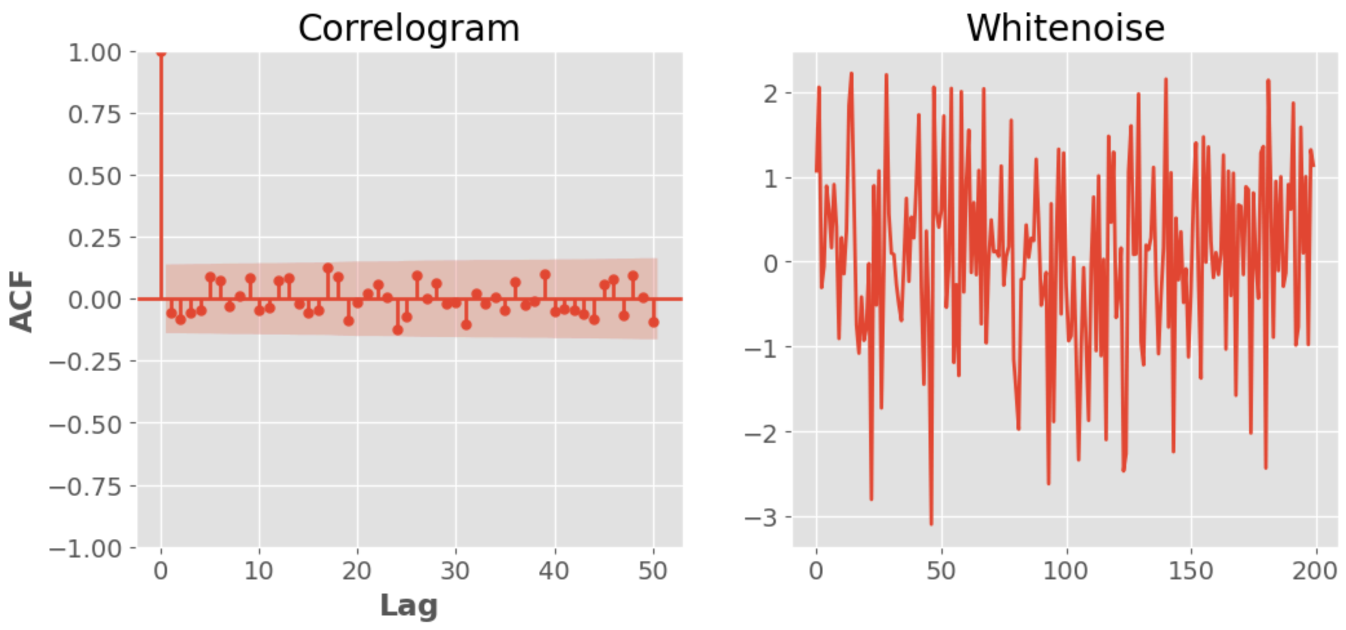
Time series are called white noise if:

- It has zero mean
- It has constant variance
- It has no autocorrelation

It is often further assumed that the white noise is i.i.d and Gaussian: $w_t \sim \mathcal{N}(0, \sigma^2)$

As a result, we expect the ACF of a white noise series to be close to 0 (no correlation) for all lags:

```python
fig, (ax1,ax2) = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

plot_acf(np.random.normal(size=200), lags=50, title="Correlogram",ax=ax1)
ax1.set_xlabel("Lag")
ax1.set_ylabel('ACF')
ax2.plot(np.random.normal(size=200))
ax2.set_title("Whitenoise");
```



Why do we care about white noise?

- **Predictability**: If your time series is white noise then that means it is a series of random numbers and cannot be predicted.
- **Residuals Diagnostics**: The residuals/errors from a time series for a forecast model should resemble white noise, implying that we have extracted all the useful information from the time-series.

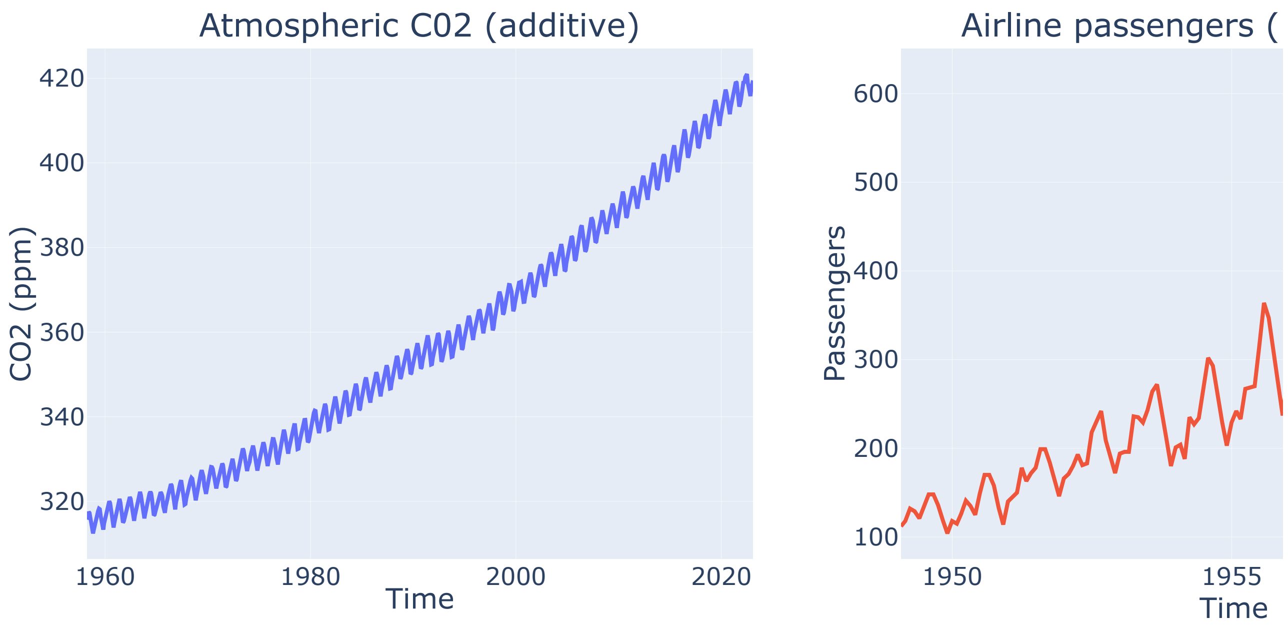
3. Time series decomposition

- Decomposing a time series into the components above can help us study them
- When we decompose a time series, we usually split it into 3 components:
    1. **Trend-cycle** (T). We typically combine the trend and cycle of a time series as they are patterns that don't repeat with a regular period. I'll often refer to this simply as the "trend".
    2. **Seasonal** (S)

3. **Remainder** (R) (also called the “residual”)

- There are two main ways we can combine/decompose these components to make up a time series:
  1. **Additive**: $y_t = S_t + T_t + R_t$. Appropriate if the magnitude of the seasonal fluctuations, or the variation around the trend-cycle, does not vary with the value of the series.
  2. **Multiplicative**: $y_t = S_t \times T_t \times R_t$. Appropriate if variation in the seasonal pattern, or the variation around the trend-cycle, appears to be proportional to the value of series.
- We’ve already seen examples of both kinds of models already:

```
plot_add_mult()
```



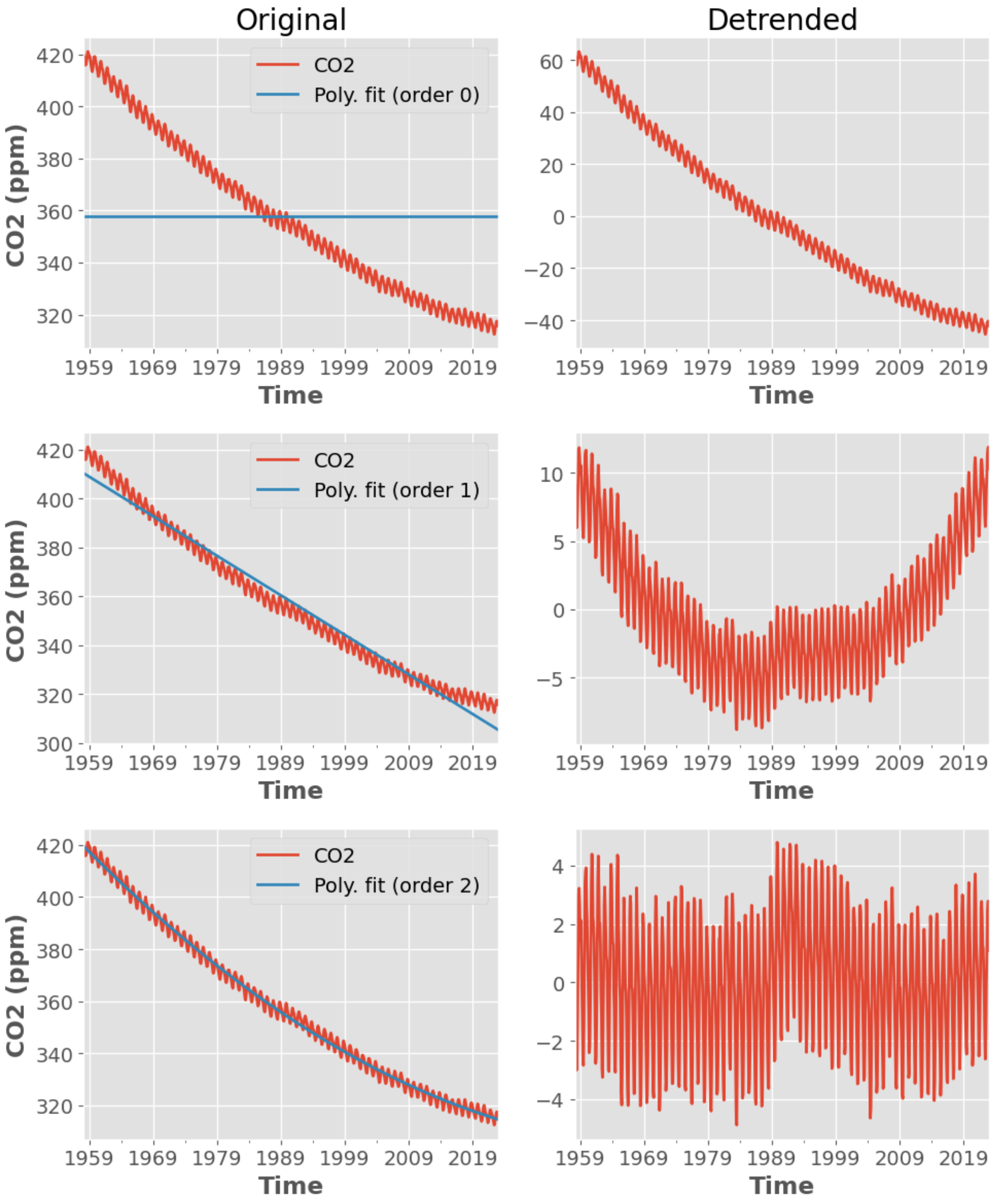
3.1. Estimating the trend-cycle

- There are two main approaches to estimating the trend component of a time series (if it exists):
  1. Curve fitting
  2. Moving average

Curve-fitting

- The function `statsmodels.tsa.tsatools.detrend()` can help us easily calculate and remove a simple polynomial curve from our data
- Let’s try this out on the CO2 time series

```
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(10, 12))

for order in range(3):
    fit_label = f"Poly. fit (order {order})"
    co2[fit_label] = co2["CO2"] - detrend(co2["CO2"], order=order)
    co2[["CO2", fit_label]].plot.line(xlabel="Time", ylabel="CO2 (ppm)", ax=ax
    (co2["CO2"] - co2[fit_label]).plot.line(xlabel="Time", legend=False, ax=ax
axes[0, 0].set_title("Original")
axes[0, 1].set_title("Detrended")
plt.tight_layout();
```

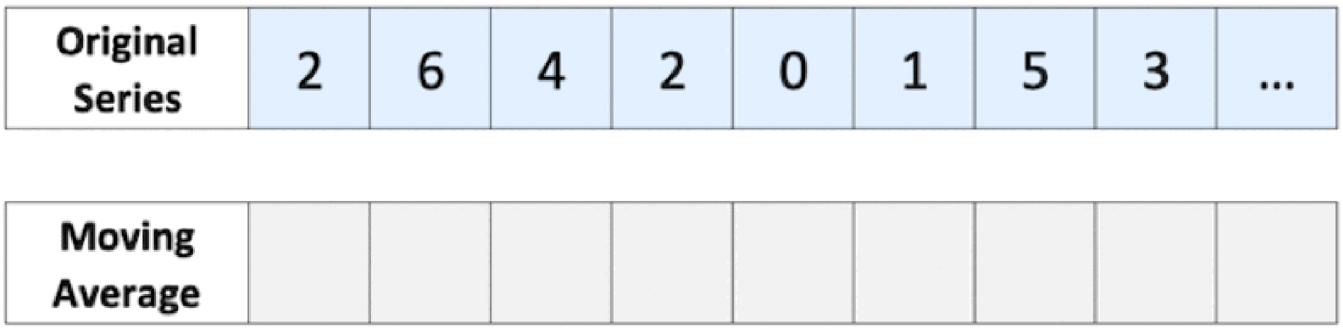


- We can see that a polynomial curve of order 2 fits the data okay, but it’s certainly not perfect and there’s still some large-scale changes in the level of our series
- In cases where you can capture the trend in a time series by just fitting a line, you often don’t need to do any more analysis! (unfortunately this is rarely the case)

- The curve fitting method assumes that the same curve fits the data at all time points, but this is usually not the case

Moving average

- Instead, we usually estimate the trend using a moving average which just means, passing a `window` of some size over the data and calculating the average as we go:



Original Series	2	6	4	2	0	1	5	3	...
Moving Average									

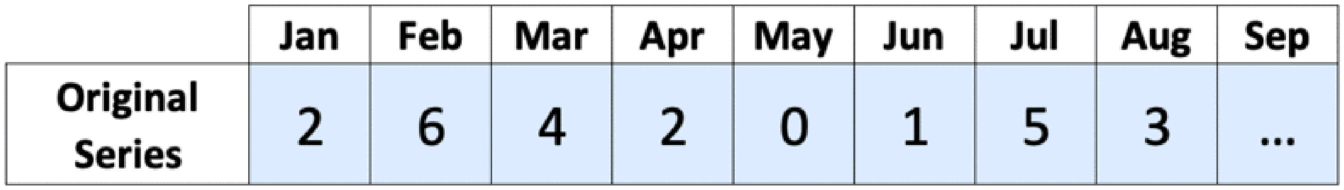
- We can calculate a moving average easily using the `.rolling()` method in Pandas. We also want to specify `center=True` to align the index

```
window_size = 3
df = pd.DataFrame([2, 6, 4, 2, 0, 1, 5, 3],
                  index=["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug

df.rolling(window=window_size, center=True).mean()
```

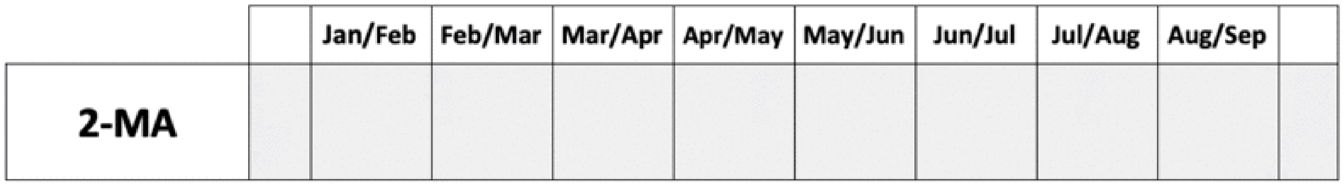
	0
Jan	NaN
Feb	4.0
Mar	4.0
Apr	2.0
May	1.0
Jun	2.0
Jul	3.0
Aug	NaN

- But what window size should you use? Well it's often dictated by the the nature of your data, your expert knowledge, and what you are trying to achieve.

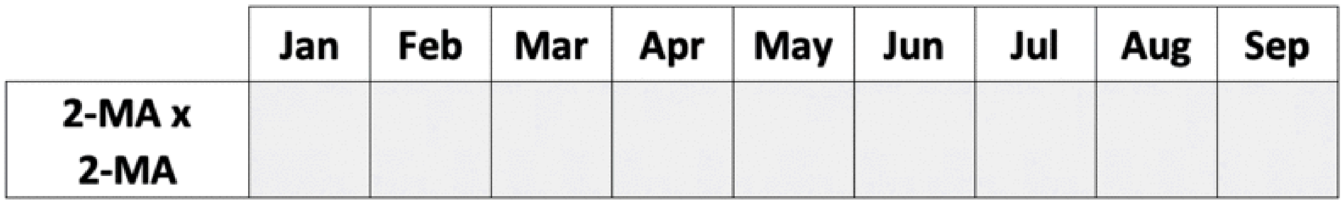
- If there's seasonality in the data, you want to use a window equal to the seasonal period. For example, in the monthly CO2 dataset, we observed an annual seasonal pattern. We should use a moving average with a window of 12 months (denoted `12-MA`). This will average out seasonal variation and allow us to focus on the trend-cycle. Using a period other than 12 may contaminate the trend-cycle with seasonality.
- Unfortunately, this leads to one minor problem, an *even* window size means our moving average doesn't line up with the index of the original series
- To rectify this, we usually take an extra `2-MA` of the result of a MA with an even window:



	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep
Original Series	2	6	4	2	0	1	5	3	...



		Jan/Feb	Feb/Mar	Mar/Apr	Apr/May	May/Jun	Jun/Jul	Jul/Aug	Aug/Sep	
2-MA										



	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep
2-MA x 2-MA									

Problem with MA of even window size

```
window_size = 2
df = pd.DataFrame([2, 6, 4, 2, 0, 1, 5, 3],
                        index=["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug
df.rolling(window=window_size, center=True).mean()
```

	0
Jan	NaN
Feb	4.0
Mar	5.0
Apr	3.0
May	1.0
Jun	0.5
Jul	3.0
Aug	4.0

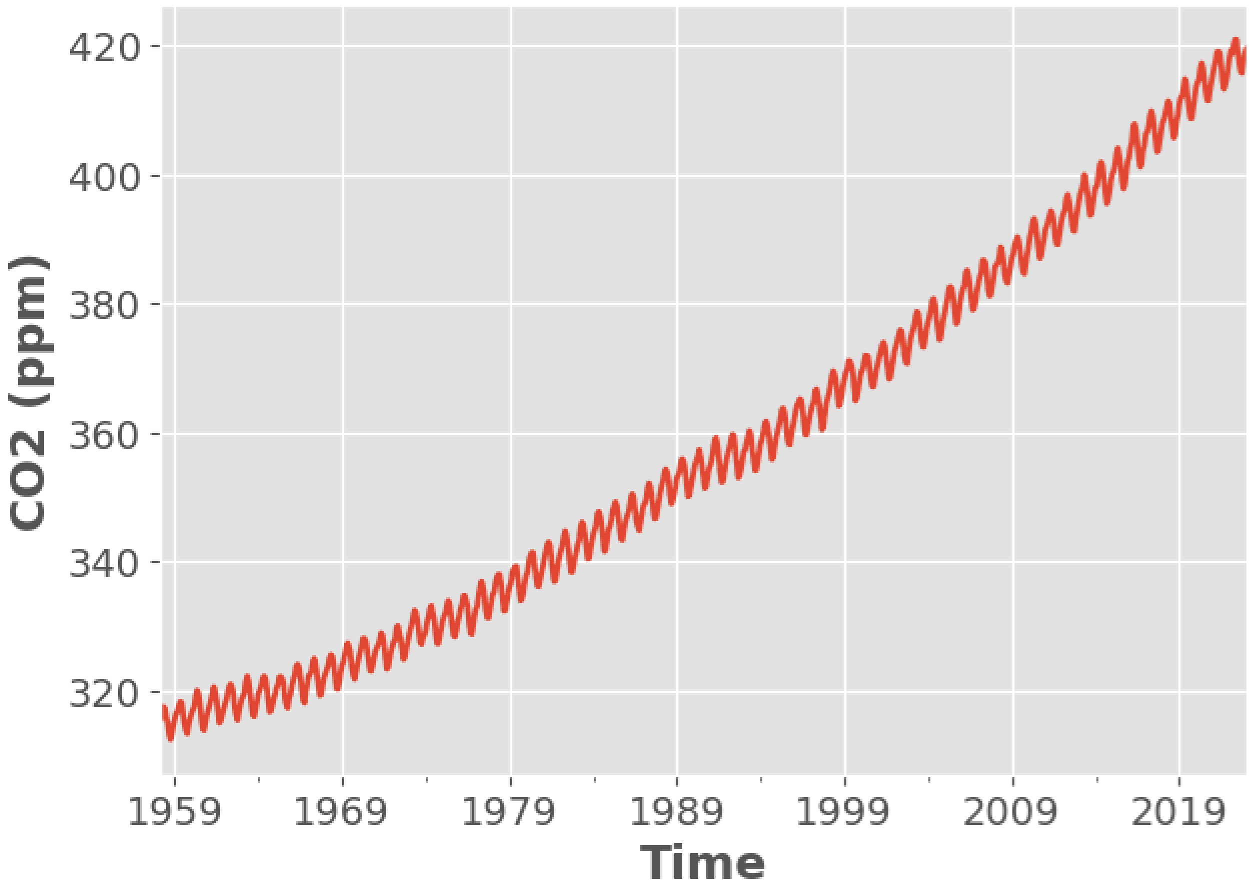
- This is how we do the above in Pandas:

```
df = pd.DataFrame(
    [2, 6, 4, 2, 0, 1, 5, 3],
    index=["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug"],
)

window = 2
df.rolling(window).mean().rolling(2).mean().shift(-window // 2)
```

	0
Jan	NaN
Feb	4.50
Mar	4.00
Apr	2.00
May	0.75
Jun	1.75
Jul	3.50
Aug	NaN

- Surprisingly difficult right? Most people don't worry about this and would just use `df.rolling(window=2, center=True)`, but that does not technically give the right result. `center=True` only relabel the index
- Let's do a `12-MA` on the the CO2 data:

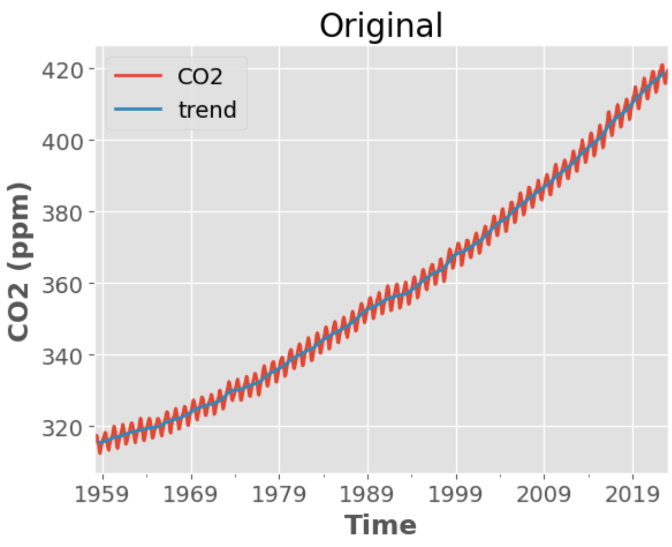
```
co2 = pd.read_csv("data/co2.csv", index_col=0, parse_dates=True)
co2.plot.line(xlabel="Time", ylabel="CO2 (ppm)", legend=False);
```

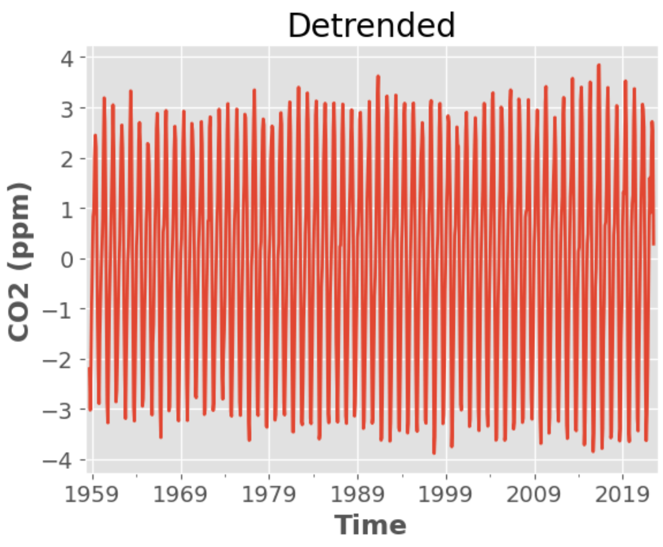


```
# Compute 2 x 12-MA
co2["trend"] = co2["CO2"].rolling(window=12).mean().rolling(2).mean().shift(-1

# Compute detrended data by substracting raw data with trend
co2["detrended"] = co2["CO2"] - co2["trend"]

co2.head(10)
```

	CO2	trend	detrended
Time			
1958-03-31	315.70	NaN	NaN
1958-04-30	317.45	NaN	NaN
1958-05-31	317.51	NaN	NaN
1958-06-30	317.24	NaN	NaN
1958-07-31	315.86	NaN	NaN
1958-08-31	314.93	NaN	NaN
1958-09-30	313.20	315.404583	-2.204583
1958-10-31	312.43	315.455417	-3.025417
1958-11-30	313.33	315.499167	-2.169167
1958-12-31	314.67	315.569583	-0.899583

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))
co2[["CO2", "trend"]].plot.line(xlabel="Time", ylabel="CO2 (ppm)", title='Orig
co2["detrended"].plot.line(xlabel="Time", ylabel="CO2 (ppm)", title='Detrended
plt.tight_layout();
```





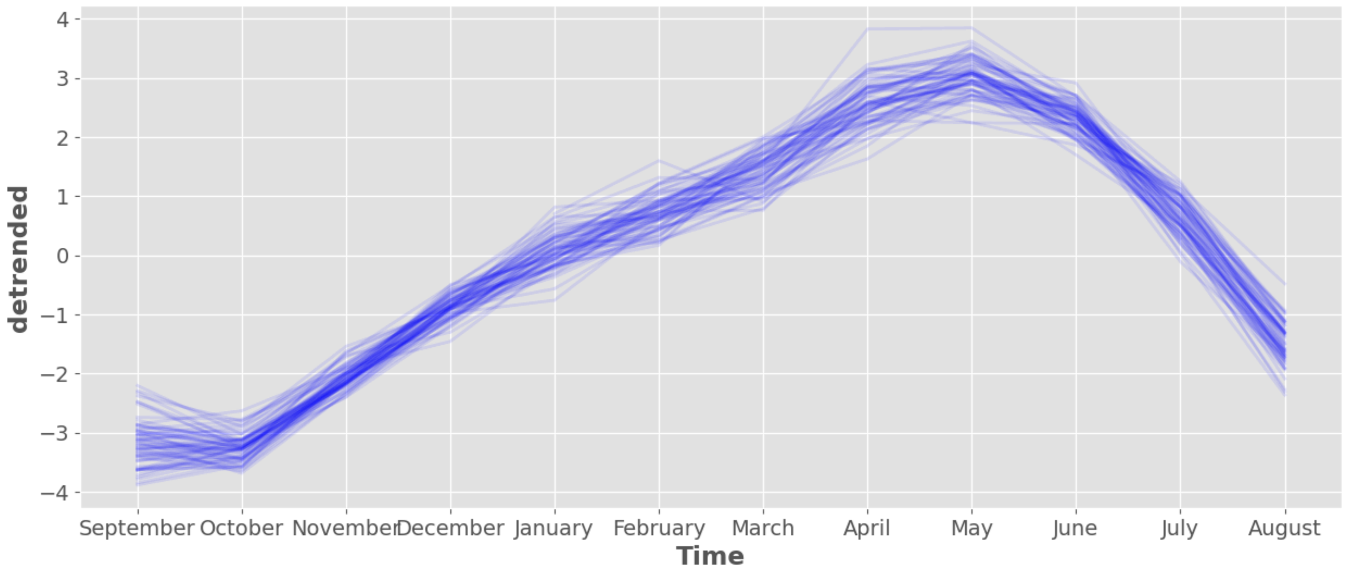
- Now we've done a *much* better job of removing the trend-cycle!

3.2. Estimating seasonality

- To compute the seasonal component we can simply:
    1. Remove the trend-cycle component from the data (we did this above and call this the "detrended" data)
    2. Estimate the seasonal component by averaging

```
df2 = co2[['detrended']].dropna()
import seaborn as sns
plt.figure(figsize=(15,6))
sns.lineplot(df2,x=df2.index.month_name(), y='detrended', hue=df2.index.year,
```

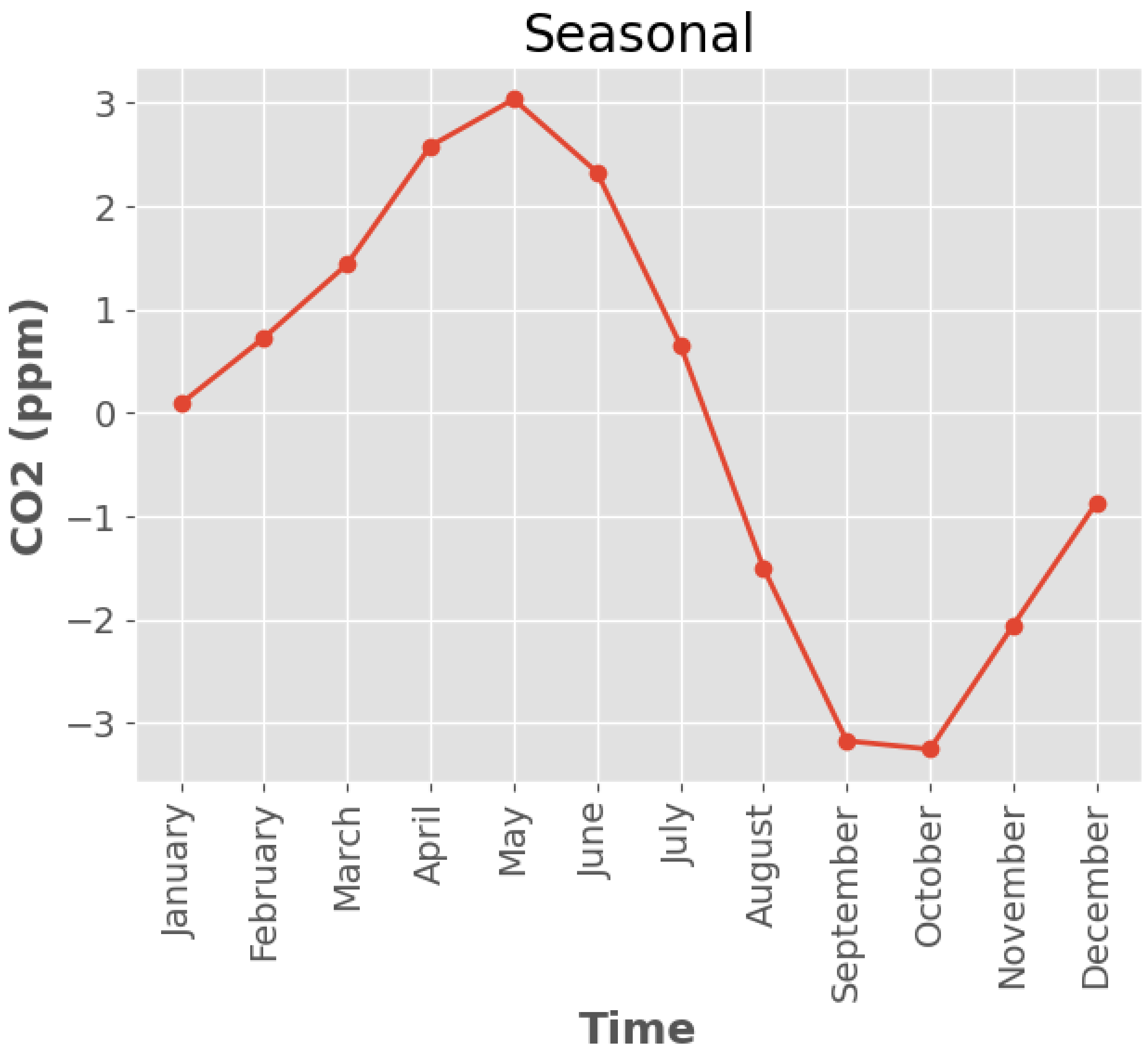
```
<Axes: xlabel='Time', ylabel='detrended'>
```



```
seasonal = co2["detrended"].groupby(co2.index.month).mean()
seasonal
```

```
Time
1     0.085898
2     0.725521
3     1.436426
4     2.579121
5     3.030885
6     2.326107
7     0.659642
8    -1.501336
9    -3.173262
10   -3.254805
11   -2.055964
12   -0.870410
Name: detrended, dtype: float64
```

```
ax = seasonal.plot.line(xlabel="Time", ylabel="CO2 (ppm)", title="Seasonal", m
ax.set_xticks(range(1, 13))
ax.set_xticklabels(calendar.month_name[1:], rotation='vertical');
```



> We can make a similar plot to the above using the `statsmodels` function `month_plot()`, but I don't think it looks as good and it was easy to make our own with Pandas:

```
month_plot(co2[['detrended']].dropna());
```

- Let's add the seasonal pattern as a column in our dataframe:

```
co2["seasonal"] = co2.index.month.map(lambda x: seasonal.to_dict()[x])
co2.head()
```

	CO2	trend	detrended	seasonal
Time				
1958-03-31	315.70	NaN	NaN	1.436426
1958-04-30	317.45	NaN	NaN	2.579121
1958-05-31	317.51	NaN	NaN	3.030885
1958-06-30	317.24	NaN	NaN	2.326107
1958-07-31	315.86	NaN	NaN	0.659642

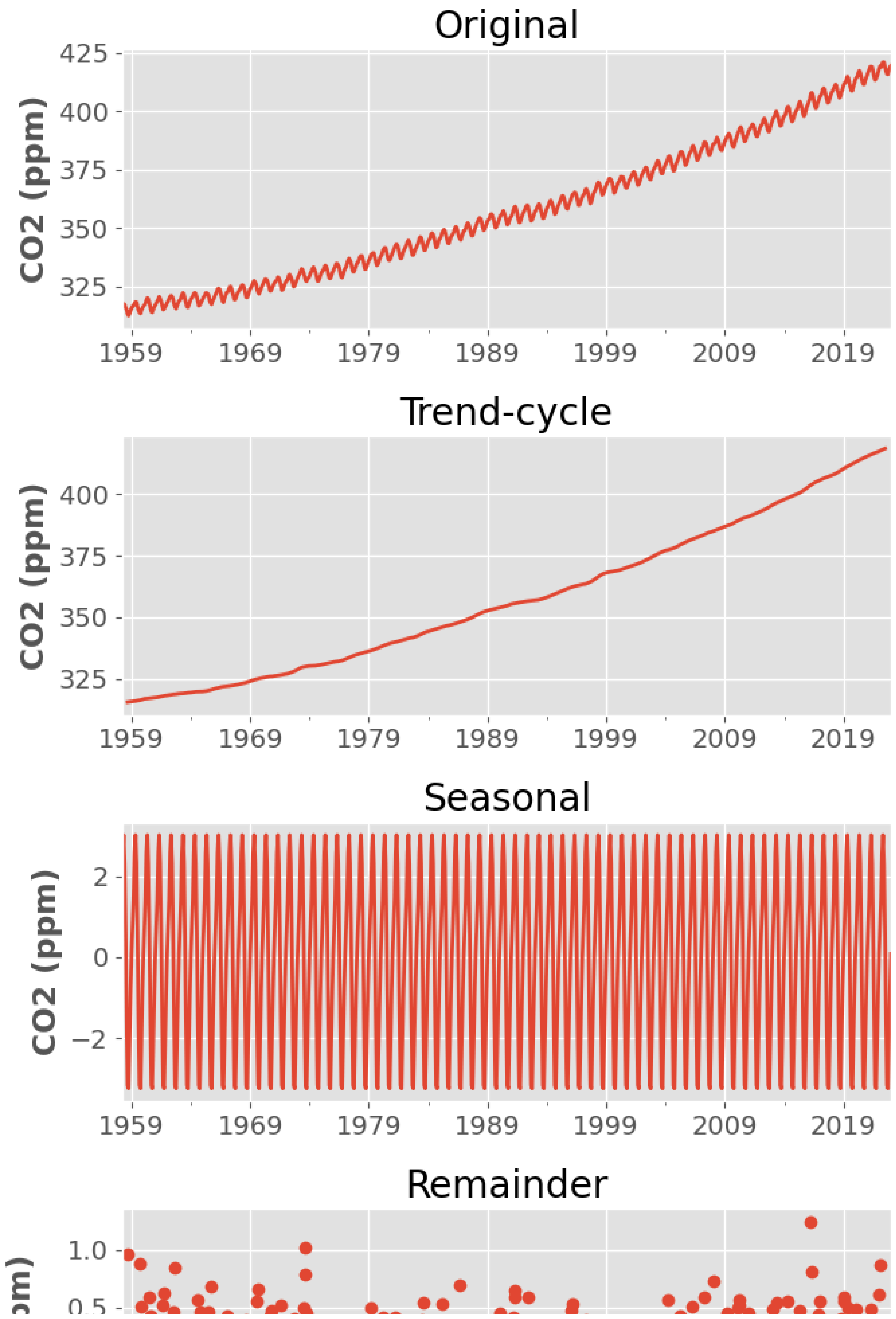
3.3. Estimating the remainder

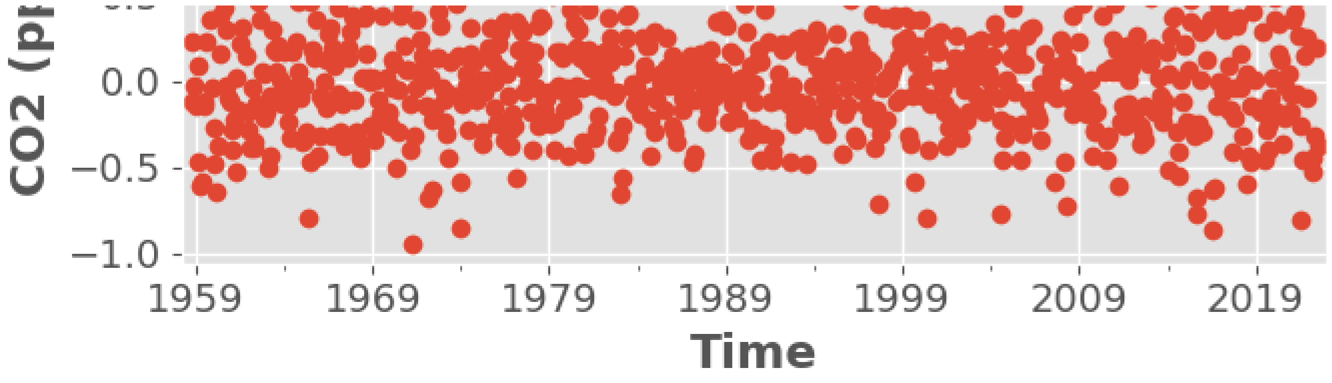
- To estimate the remainder, we simply subtract the trend-cycle and seasonal components from the original series:

```
co2["remainder"] = co2["CO2"] - co2["trend"] - co2["seasonal"]
co2.head(12)
```

	CO2	trend	detrended	seasonal	remainder
Time					
1958-03-31	315.70	NaN	NaN	1.436426	NaN
1958-04-30	317.45	NaN	NaN	2.579121	NaN
1958-05-31	317.51	NaN	NaN	3.030885	NaN
1958-06-30	317.24	NaN	NaN	2.326107	NaN
1958-07-31	315.86	NaN	NaN	0.659642	NaN
1958-08-31	314.93	NaN	NaN	-1.501336	NaN
1958-09-30	313.20	315.404583	-2.204583	-3.173262	0.968678
1958-10-31	312.43	315.455417	-3.025417	-3.254805	0.229388
1958-11-30	313.33	315.499167	-2.169167	-2.055964	-0.113203
1958-12-31	314.67	315.569583	-0.899583	-0.870410	-0.029173
1959-01-31	315.58	315.635833	-0.055833	0.085898	-0.141732
1959-02-28	316.48	315.658750	0.821250	0.725521	0.095729

```
fig, axes = plt.subplots(nrows=4, ncols=1, figsize=(7, 12))
co2["CO2"].plot.line(ylabel="CO2 (ppm)", title="Original", xlabel="", ax=axes[
co2["trend"].plot.line(ylabel="CO2 (ppm)", title="Trend-cycle", xlabel="", ax=
co2["seasonal"].plot.line(ylabel="CO2 (ppm)", title="Seasonal", xlabel="", ax=
co2["remainder"].plot.line(ylabel="CO2 (ppm)", title="Remainder", ax=axes[3],
plt.tight_layout();
```


Original
CO2 (ppm)
425
400
375
350
325
1959
1969
1979
1989
1999
2009
2019
Trend-cycle
CO2 (ppm)
400
375
350
325
1959
1969
1979
1989
1999
2009
2019
Seasonal
CO2 (ppm)
2
0
−2
1959
1969
1979
1989
1999
2009
2019
Remainder
1.0
0.5

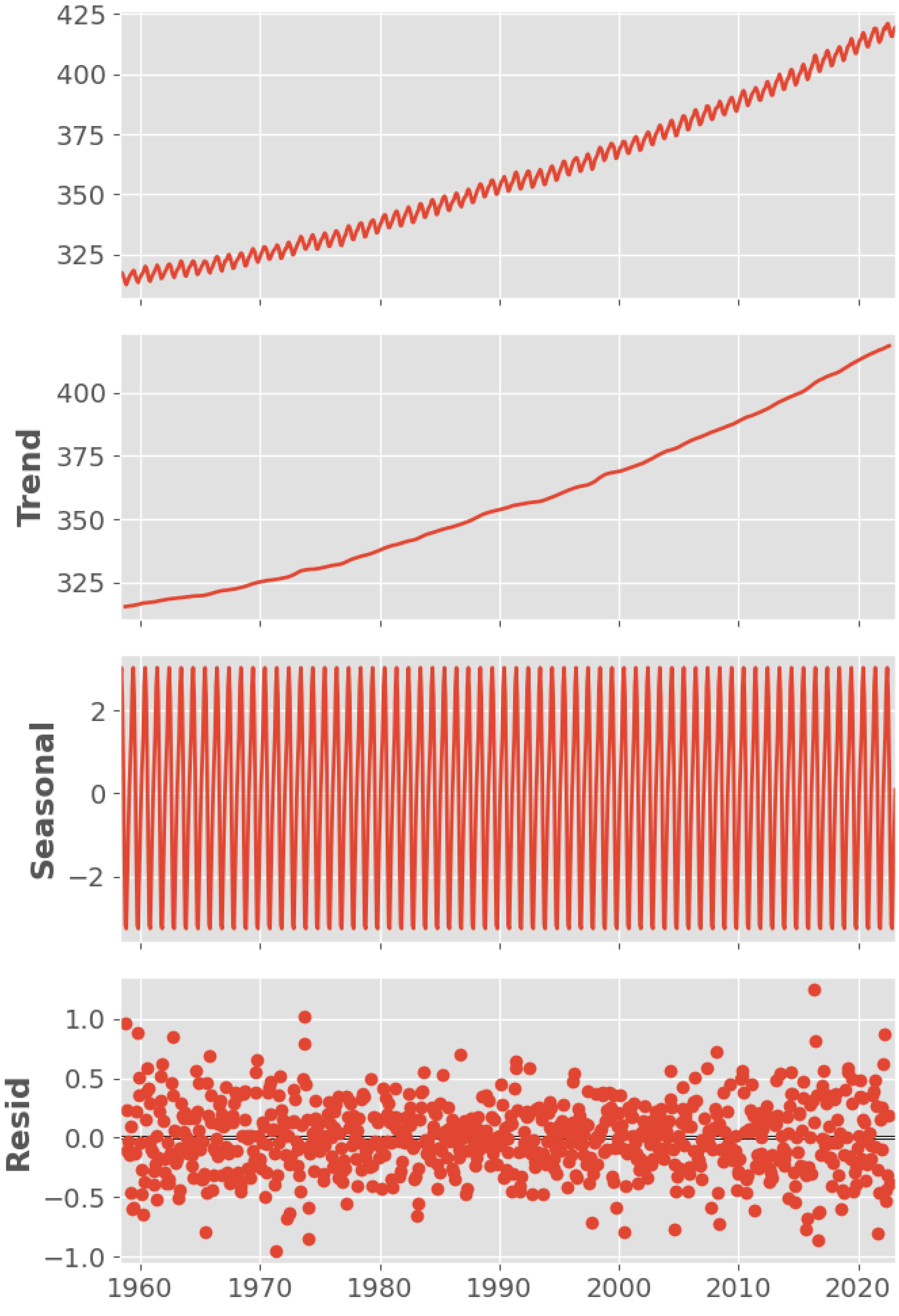


- What we did above is an “additive decomposition”. Multiplicative decomposition is similar, except that the subtractions we did are replaced by divisions
- We also call this kind of “classical decomposition” naive because it assumes that the seasonal component remains constant over time. We’ll look at an more advanced alternative decomposition method later on.

3.4. Putting it all together

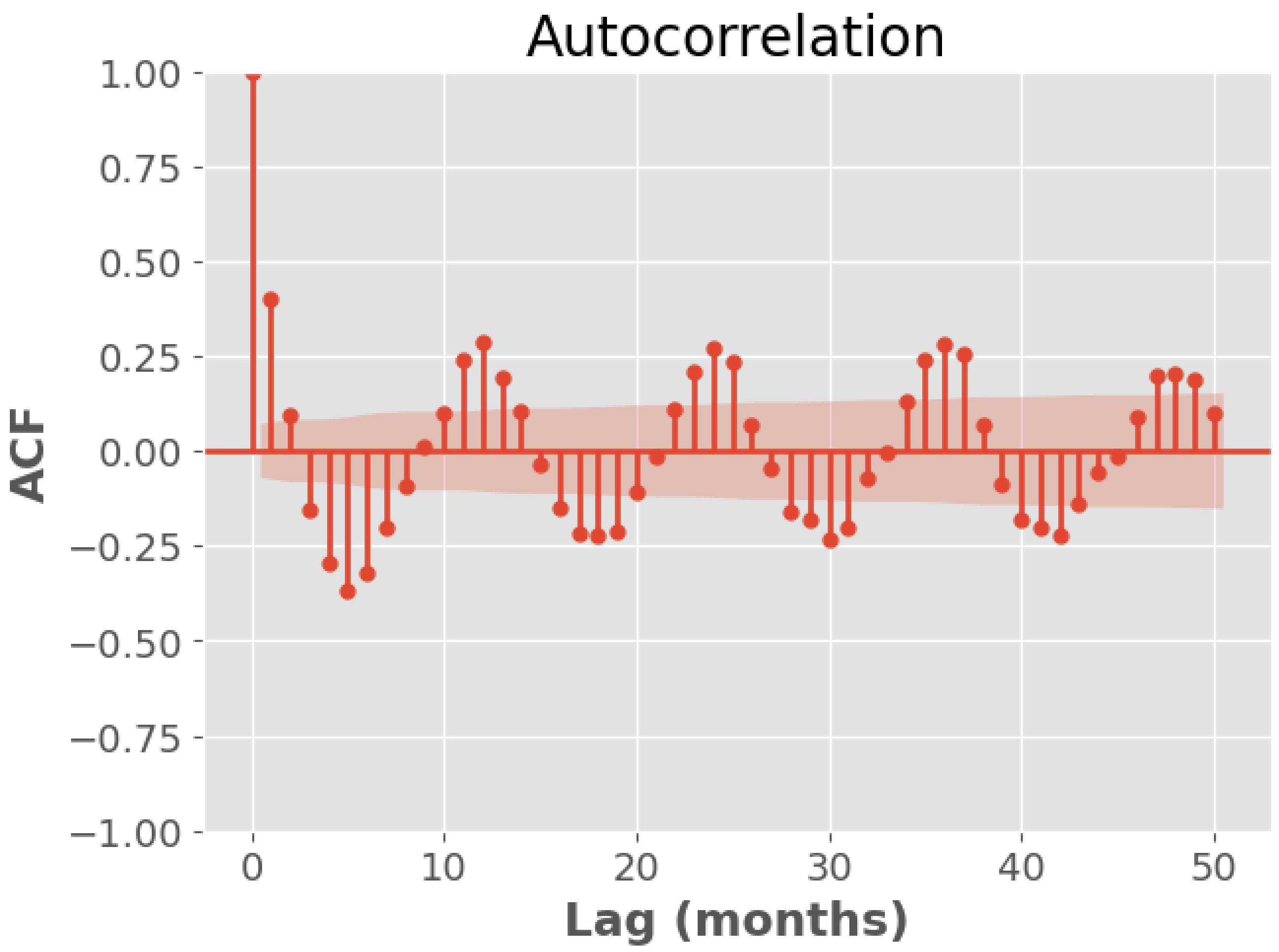
- Luckily, we don’t need to do this by hand ever again, there are convenient functions out there for us!
- For example, `statsmodels.tsa.seasonal.seasonal_decompose()`:

```
model = seasonal_decompose(co2[["CO2"]], model="additive", period=12)
with mpl.rc_context():  # this context manager help adjust the size of the plo
    mpl.rc("figure", figsize=(7, 10))
    model.plot()
    plt.tight_layout()
```

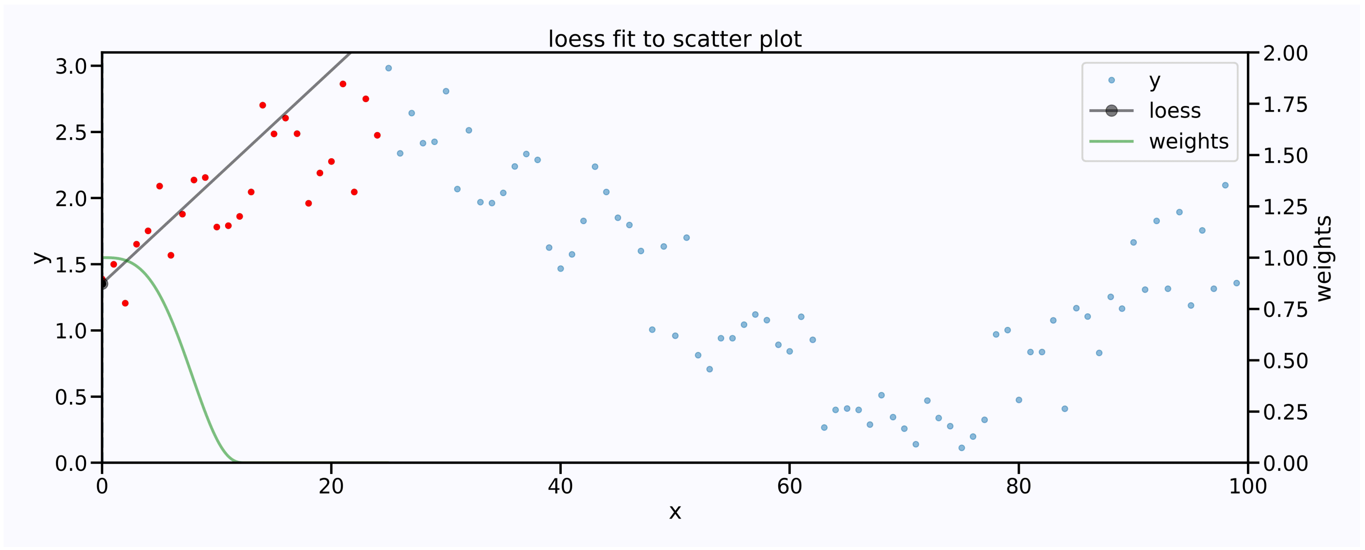

425
400
375
350
325
Trend
Seasonal
2
0
−2
Resid
1.0
0.5
0.0
−0.5
−1.0
1960
1970
1980
1990
2000
2010
2020

- Ideally, the remainder doesn't contain any obvious pattern or seasonality
- Let's check the residuals now with a correlogram:

```
fig = plot_acf(model.resid.dropna(), lags=50)
plt.ylabel("ACF")
plt.xlabel("Lag (months)");
```

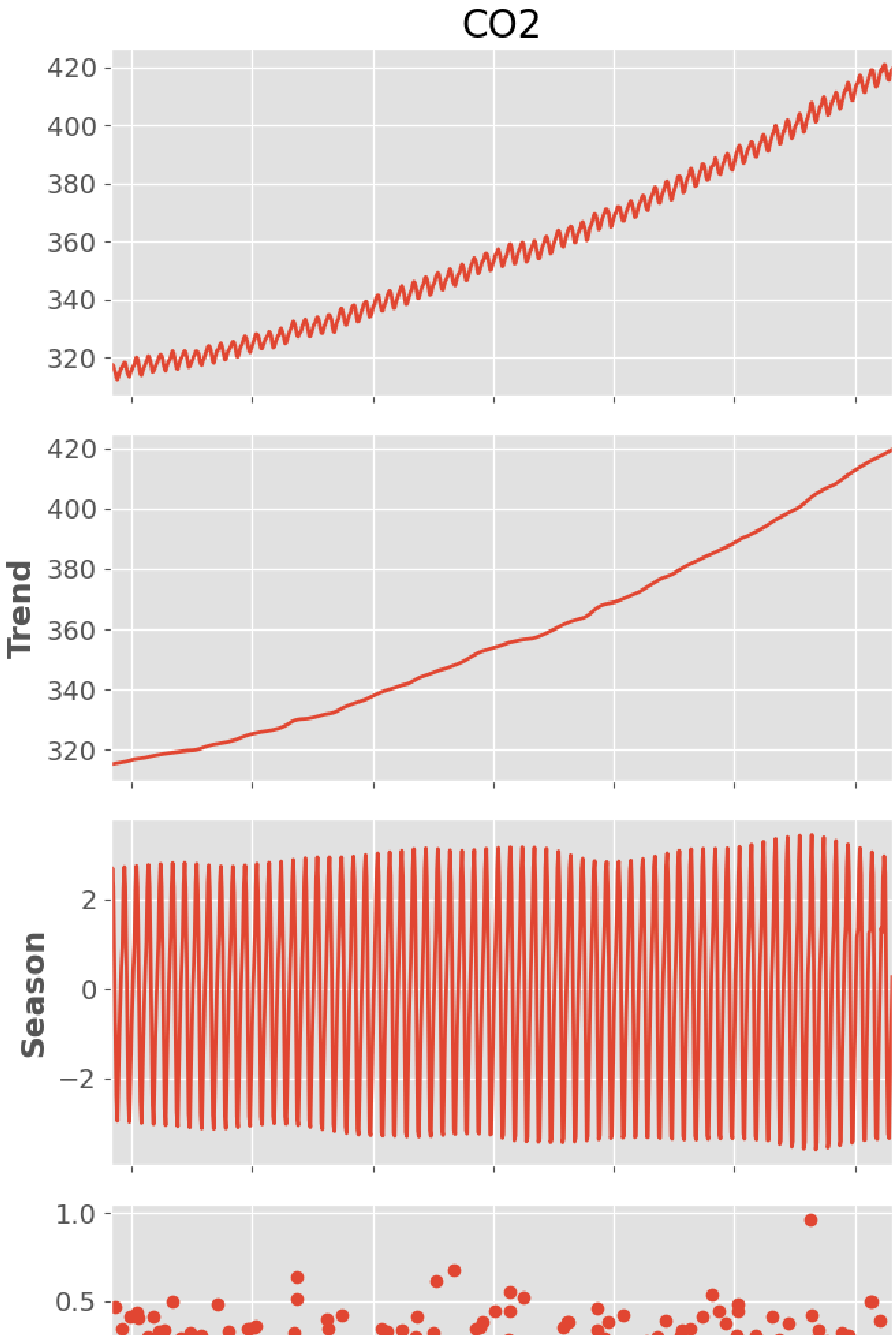


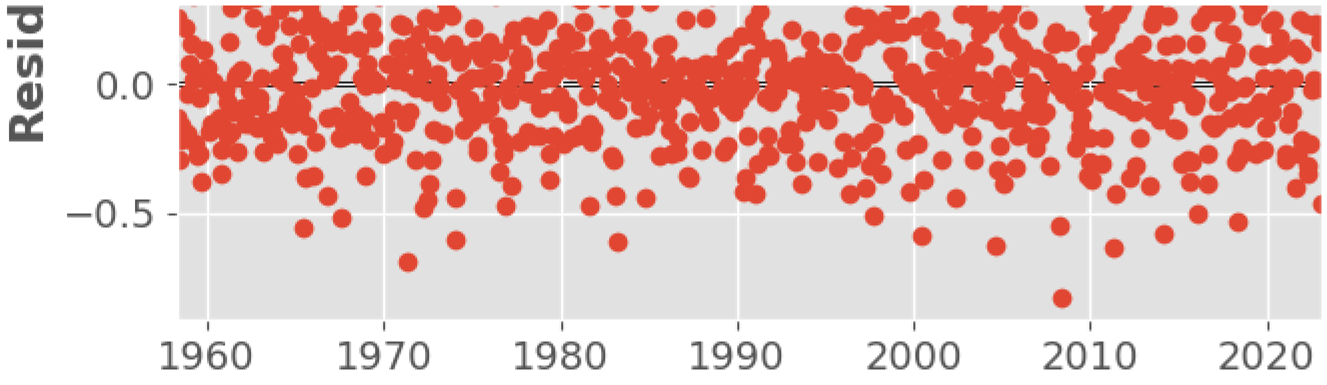
- The residual clearly contains seasonal dependence still
- We can use a more advanced method of decomposition to try and extract some of this. Probably the most common is an STL decomposition which uses LOESS (locally estimated scatterplot smoothing). Here's a good video to understand how LOESS works. You can read more about STL decomposiion here. But basically, it allows the seasonal component to change over time (amongst other things).
- Let's try an STL decomposition now using `statsmodels.tsa.seasonal.STL()`:



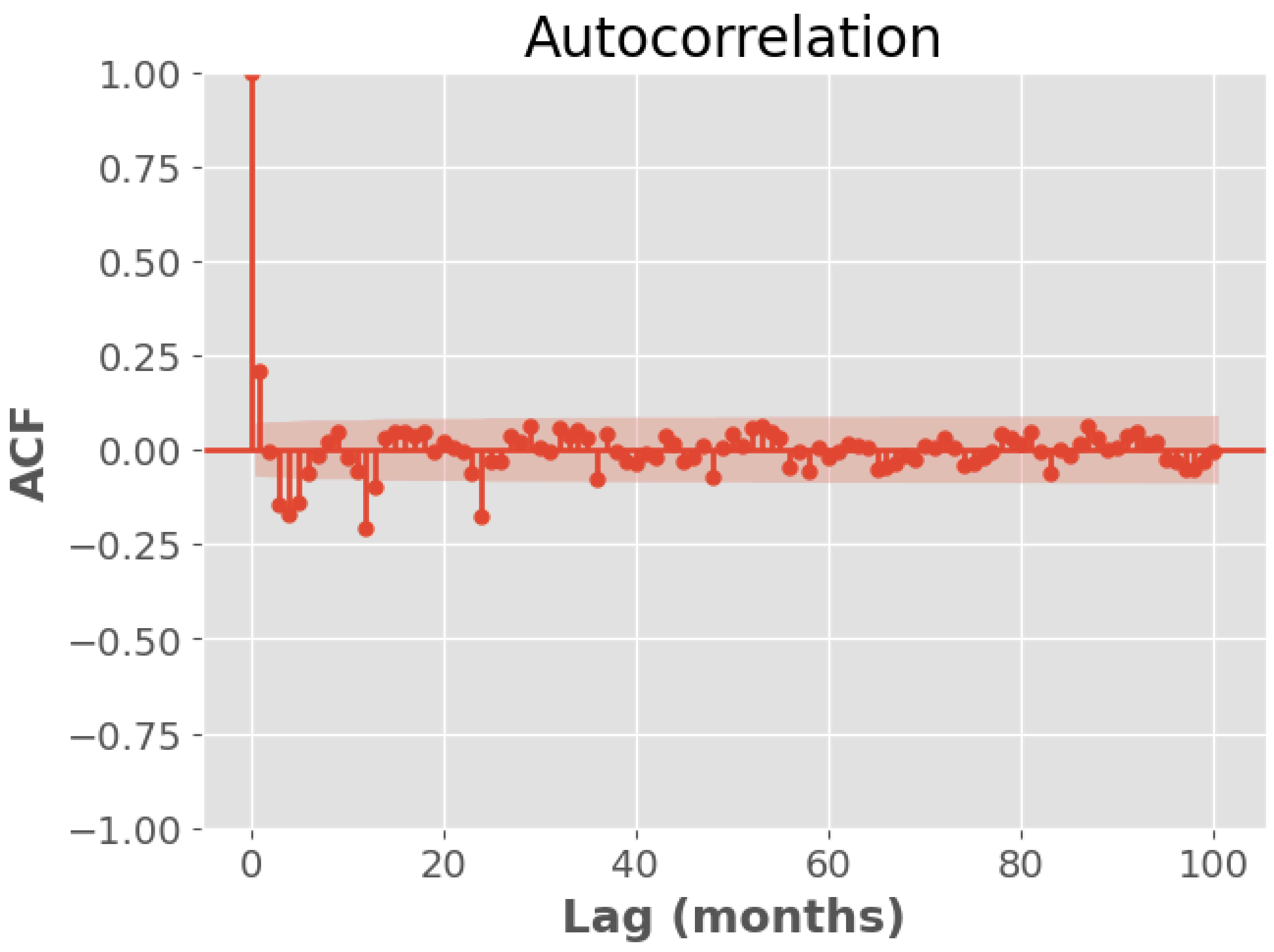
Credit: https://towardsdatascience.com/multi-seasonal-time-series-decomposition-using-mstl-in-python-136630e67530

```
model = STL(co2["CO2"], period=12, seasonal=13).fit()
with mpl.rc_context():
    mpl.rc("figure", figsize=(7, 12))
    model.plot()
```


CO2
420
400
380
360
340
320
Trend
Season
2
0
−2
1.0
0.5



```
fig = plot_acf(model.resid, lags=100)
plt.ylabel("ACF")
plt.xlabel("Lag (months)");
```

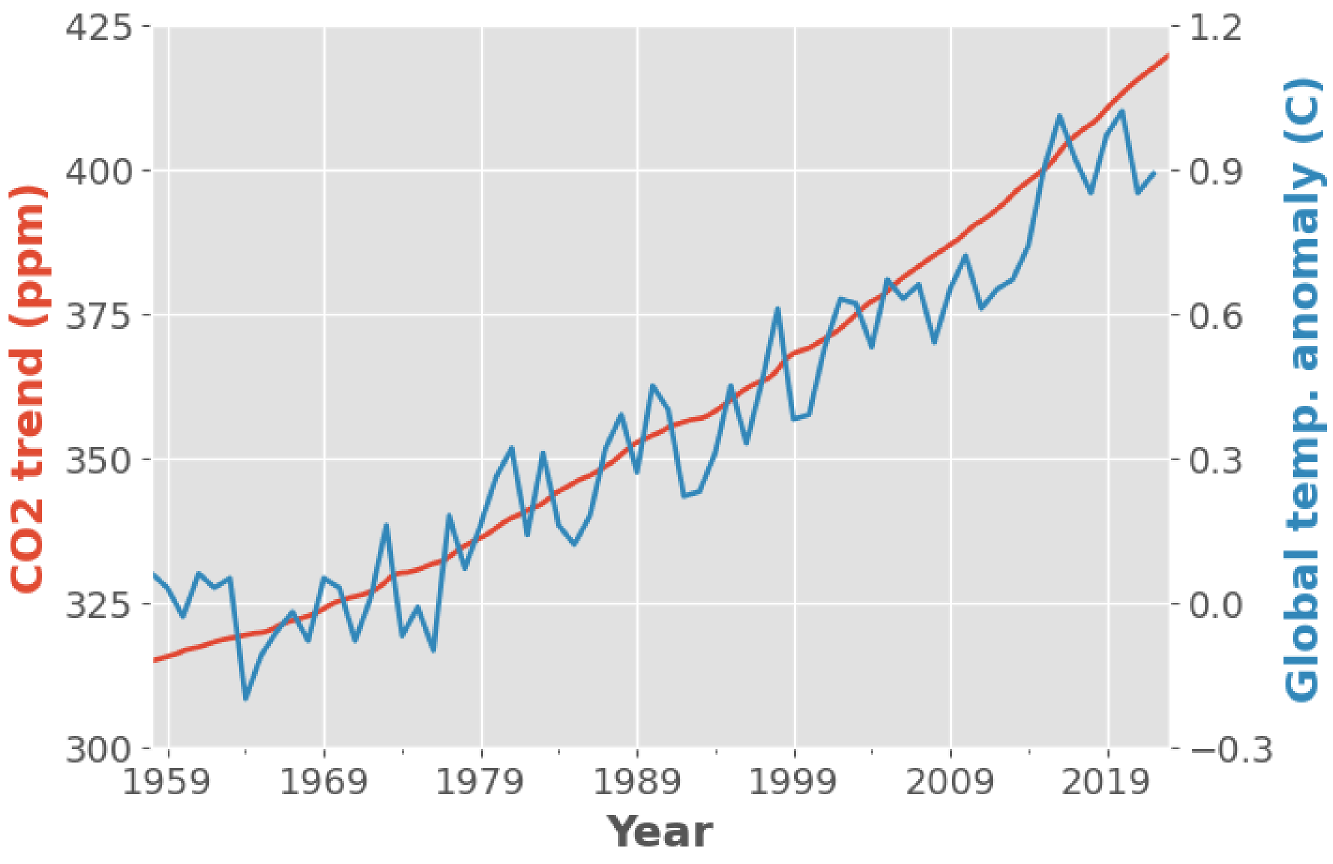


- Much better! Looks like we extracted that seasonal pattern nicely. However, there still seems to be some temporal dependence at low lags which we may wish to model with a time series model (coming up in later lectures)

3.5. Next steps

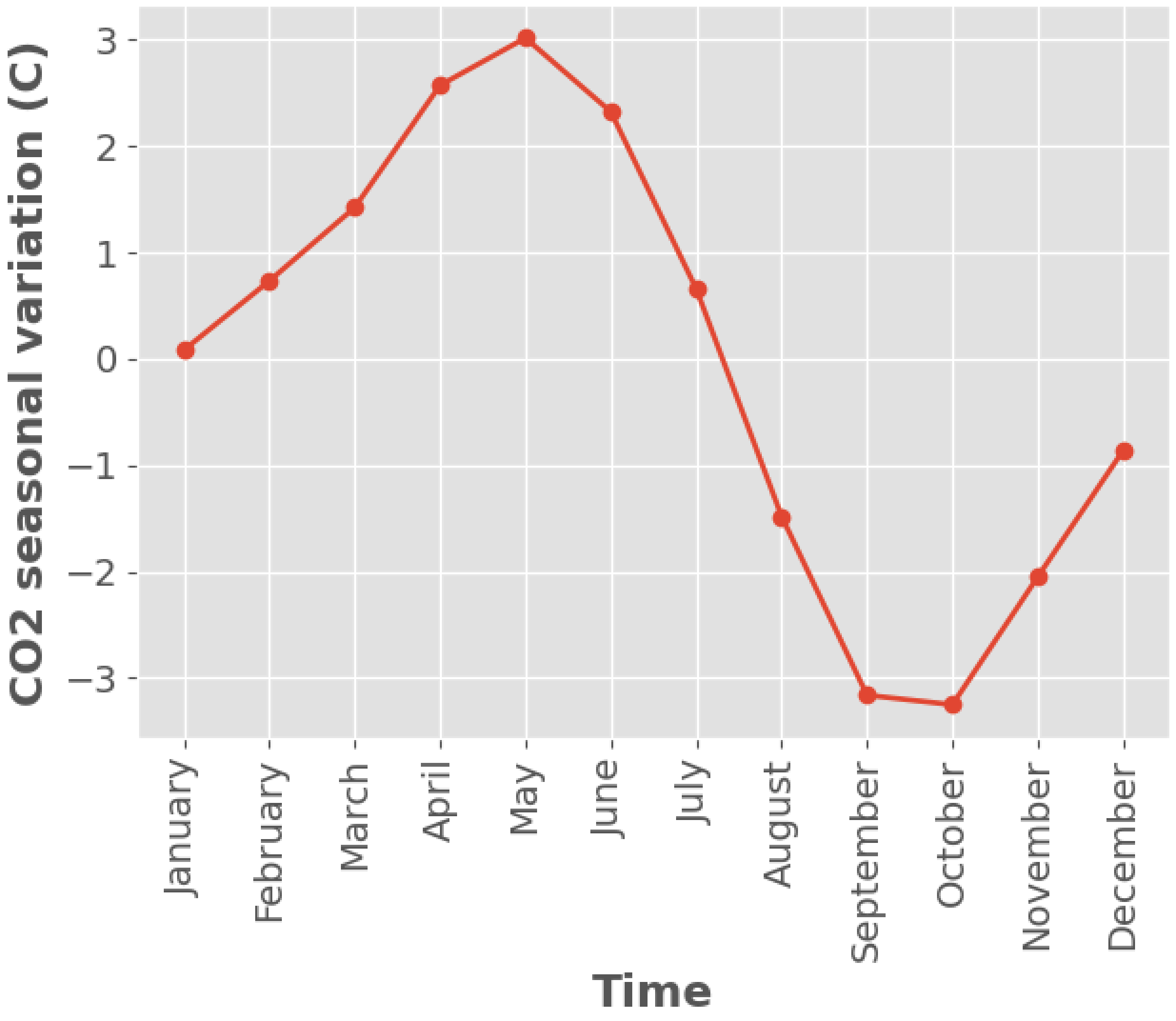
- So now we've decomposed our series somewhat, what next? Well there's two keys things we can do from here:
  1. Use the insights we've gained to better understand our data
  2. Use the insights we've gained to help us do some forecasting (next lectures)
- Let's focus on point 1:
  1. There a strong positive trend in the data
  2. There's strong seasonality which has increased slightly over time
- This helps us understand our time series and its behaviour. For example, what else do we know that has been increasing over the last century?

```
temp = (pd.read_csv("data/temperature_anomaly.csv", index_col=0, parse_dates=T
          .loc[str(co2.index[0].year):])
ax1 = model.trend.plot.line()
temp.head()
ax2 = temp["Anomaly"].plot.line(secondary_y=True, ax=ax1)
ax1.set_yticks(np.linspace(300, 425, 6)); ax1.set_ylim(300, 425); ax1.set_ylab
ax2.set_yticks(np.linspace(-0.3, 1.2, 6)); ax2.set_ylim(-0.3, 1.2); ax2.set_yl
```



- What about the seasonality? Why is it seasonal? It matches the weather seasons (summer, spring, winter, autumn) quite well don't you think?

```
seasonal = model.seasonal.groupby(model.seasonal.index.month).mean()
seasonal.plot.line(xlabel="Time", ylabel="CO2 seasonal variation (C)", marker=
plt.xticks(range(1, 13), calendar.month_name[1:], rotation='vertical');
```


CO2 seasonal variation (C)
January
February
March
April
May
June
July
August
September
October
November
December
Time