Time Series Analisys on Geoespatial Data with Python

Author: João Otavio Nascimento Firigato

email: joaootavionf007@gmail.com

LinkedIn: https://www.linkedin.com/in/jo%C3%A3o-otavio-firigato-4876b3aa/

First instructions:

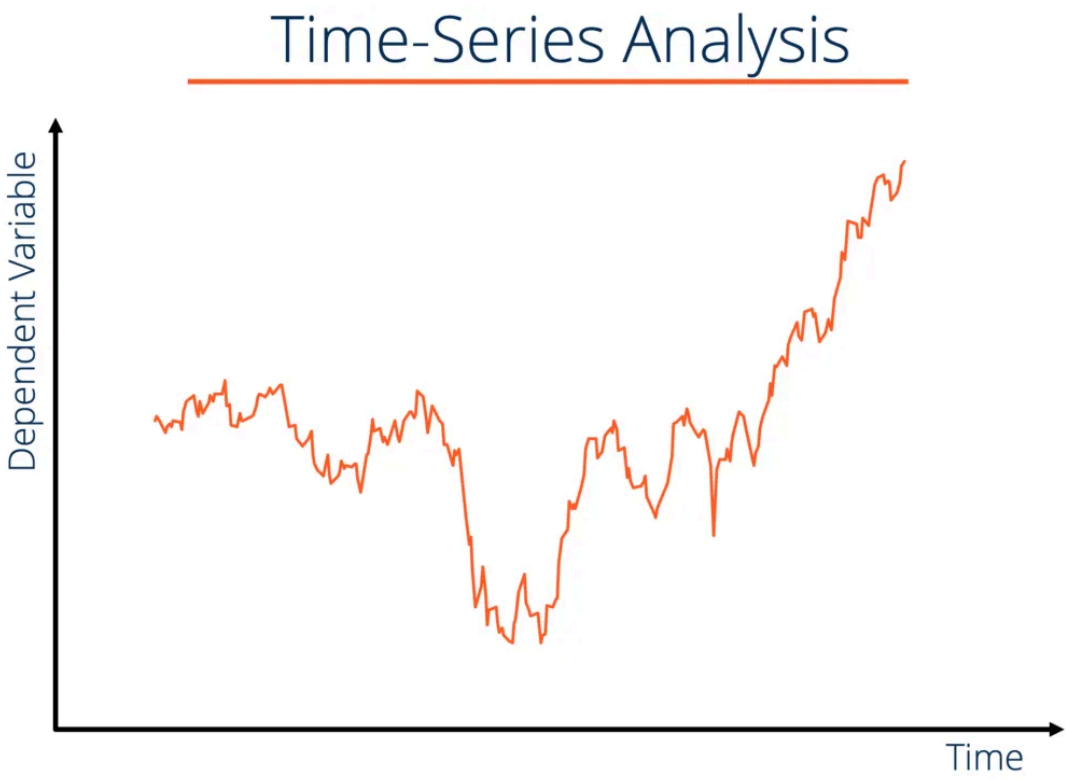
✅ Access the link to join our private WhatsApp community for students: https://chat.whatsapp.com/EPn27ZgR07lF3e1vnj8FiI

❗ It is important to access the Whatsapp Group to get the Colab Notebooks, as the PDF files are protected from text copying.

Chapter 1 - Introduction to Time Series and Temporal Data Manipulation

What is a Time Series Analysis

Time series analysis is a specific way of analyzing a sequence of data points collected over a period of time. In time series analysis, analysts record data points at consistent intervals over a given period of time, rather than just recording data points intermittently or randomly. However, this type of analysis is not just about collecting data over time. What sets time series data apart from other data is that the analysis can show how variables change over time. In other words, time is a crucial variable because it shows how the data adjusts across data points, as well as the final results. It provides an additional source of information and a defined order of dependencies between the data. Time series analysis typically requires a large number of data points to ensure consistency and reliability. A large data set ensures that you have a representative sample size and that the analysis can eliminate noisy data. It also ensures that any trends or patterns discovered are not outliers and could account for seasonal variation. Additionally, time series data can be used for forecasting – predicting future data based on historical data.



Time Series Forecasting Taxonomy

When you're faced with a new time series forecasting problem, there are many things to consider. The choices you make directly impact every step of the project, from designing a test suite to evaluate forecasting models to the fundamental difficulty of the forecasting problem you're working on. You can narrow down your options very quickly by working through a series of questions about your time series forecasting problem. By considering a few themes and questions within each theme, you can narrow down the type of problem, test suite, and even the choice of algorithms for your project.

Time series forecasting involves developing and using a predictive model on data where there is an ordered relationship between observations. Before you begin your project, you can answer a few questions and greatly improve your understanding of the structure of your forecasting problem, the structure the model requires, and how to evaluate it.

Inputs vs. Outputs

Typically, a forecasting problem involves using past observations to predict or forecast one or more possible future observations. The goal is to guess what might happen in the future. When you need to make a forecast, it is crucial to think about the data you will have available to make the forecast and what you will be guessing about the future. We can summarize this as what are the inputs and outputs of the model when making a single forecast.

- Inputs: Historical data fed to the model to make a single forecast.

- Outputs: Forecast or prediction for a future time step beyond the data provided as input.

The input data is not the data used to train the model. We haven't gotten to that point yet. It is the data used to make a forecast, for example, the last seven days of sales data to predict the next day of sales data. Defining the inputs and outputs of the model forces you to think about what exactly is or might be needed to make a forecast. You may not be able to be specific when it comes to input data. For example, you may not know whether one or several previous time steps are needed to make a prediction. But you will be able to identify the variables that can be used to make a prediction.

Endogenous vs. Exogenous

Input data can be broken down to better understand its relationship with the output variable. An input variable is endogenous if it is influenced by other variables in the system and the output variable depends on it. In a time series, the observations for an input variable are independent of each other. For example, an observation at time depends on the observation at t-1; t-1 may depend on t-2, and so on. An input variable is an exogenous variable if it is independent of other variables in the system and the output variable depends on it. Simply put, endogenous variables are influenced by other variables in the system (including themselves), while exogenous variables are not and are considered to be outside the system.

- Endogenous: Input variables that are influenced by other variables in the system and on which the output variable depends.
- Exogenous: Input variables that are not influenced by other variables in the system and on which the output variable depends.

Typically, a time series forecasting problem has endogenous variables (e.g., the output is a function of some number of previous time steps) and may or may not have exogenous variables. Exogenous variables are often overlooked due to the heavy focus on time series. Thinking explicitly about both types of variables can help identify easily overlooked exogenous data or even engineered features that can improve the model.

Regression vs. Classification

Regression predictive modeling problems are those in which a quantity is predicted. A quantity is a numeric value; for example, a price, a count, a volume, and so on. A time series prediction problem in which you want to predict one or more future numeric values is a regression predictive modeling problem. Classification predictive modeling problems are those in which a category is predicted. A category is a label from a small, well-defined set of labels; for example, hot, cold, up, down, and buy, sell are categories. A time series prediction problem in which you want to classify input time series data is a classification predictive modeling problem.

- Regression: Predicting a numeric quantity.
- Classification: Classifying as one of two or more labels.

Are you working on a regression or classification predictive modeling problem? There is some flexibility between these types. For example, a regression problem can be rephrased as classification, and a classification problem can be rephrased as regression. Some problems, such as predicting an ordinal value, can be framed as classification and regression. It is possible that reformulating your time series forecasting problem can simplify it.

Unstructured vs. Structured

It is useful to plot each variable in a time series and inspect the graph for possible patterns. A time series for a single variable may have no obvious pattern. We can think of a series without a pattern as unstructured, as if there were no discernible time-dependent structure. Alternatively, a time series may have obvious patterns, such as a trend or seasonal cycles, as structured. We can often simplify the modeling process by identifying and removing obvious structures from the data, such as an increasing trend or repeating cycle. Some classical methods even allow you to specify parameters to handle these systematic structures directly.

- Unstructured: No obvious time-dependent systematic patterns in a time series variable.
- Structured: Time-dependent systematic patterns in a time series variable (e.g., trend and/or seasonality).

Univariate vs. Multivariate

A single variable measured over time is called a univariate time series. Univariate means one variable or one variable. Multiple variables measured over time are called multivariate time series: multiple variables or multiple variables.

- Univariate: One variable measured over time.
- Multivariate: Multiple variables measured over time.

Are you working on a univariate or multivariate time series problem?

Considering this issue with respect to inputs and outputs can add an additional distinction. The number of variables may differ between inputs and outputs, for example, the data may not be symmetric. For example, you may have multiple variables as inputs to the model and only be interested in predicting one of the variables as output. In this case, there is an assumption in the model that the multiple input variables assist and are necessary in predicting the single output variable.

- Univariate and multivariate inputs: One or multiple input variables measured over time.
- Univariate and multivariate outputs: One or multiple output variables to be predicted.

Single-step vs. Multi-step

A forecasting problem that requires a forecast of the next time step is called a single-step forecasting model. While a forecasting problem that requires a forecast of more than one time step is called a multi-step forecasting model. The more time steps that are projected into the future, the more challenging the problem becomes, given the compounding nature of the uncertainty at each forecasted time step.

- Single-step: Forecast the next time step.
- Multi-step: Forecast more than one future time step.

Static vs. Dynamic

It is possible to develop a model once and use it repeatedly to make predictions. Since the model is not updated or changed between predictions, we can think of this model as being static. On the other hand, we may receive new observations before making a subsequent prediction that could be used to create a new model or update the existing model. We can think of developing a new or updated model before each prediction as a dynamic problem. For example, if the problem calls for a prediction at the beginning of the week for the following week, we may receive the true observation at the end of the week that we can use to update the model before making the prediction for the next week. This would be a dynamic model. If we do not get a true observation at the end of the week, or if we do get a true observation and choose not to re-t the model, this would be a static model. We may prefer a dynamic model, but domain constraints or limitations of a chosen algorithm may impose constraints that make it intractable.

- Static. A forecast model is t once and used to make predictions.
- Dynamic. A forecast model is based on newly available data before each prediction.

Contiguous vs. Discontiguous

A time series in which the observations are uniform over time can be described as contiguous. Many time series problems have contiguous observations, such as one observation every hour, day, month, or year. A time series in which the observations are not uniform over time can be described as discontiguous. The lack of uniformity of observations can be caused by missing or corrupted values. It can also be a characteristic of the problem in that observations are available only sporadically or at increasingly or decreasingly spaced time intervals. In the case of nonuniform observations, specific data formatting may be required when fitting some models to make the observations uniform over time.

- Contiguous. The observations are made uniform over time.
- Discontiguous. The observations are not uniform over time.

Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, or physics. Anything that is observed or measured at many points in time forms a time series. Many time series have a fixed frequency, meaning that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once a month. Time series can also be irregular without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application, and you might have one of the following:

- Timestamps, specific instants in time
- Fixed periods, such as the month of January 2007 or the entire year of 2010
- Time intervals, indicated by a start and end timestamp. Periods can be considered special cases of intervals
- Experience or elapsed time; Each timestamp is a measurement of time relative to a specific start time.

Date and Time Data Types

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The datetime, time, and calendar modules are the best places to start. The datetime.datetime type, or simply datetime, is widely used:

In [ ]:
```
from datetime import datetime
import pandas as pd
```

In [ ]:
```
now = datetime.now()
```

In [ ]:
```
now
```

Out[ ]:
```
datetime.datetime(2025, 5, 13, 23, 24, 45, 770412)
```

In [ ]:
```
print(now.year)
print(now.month)
print(now.day)
```

```
2025
5
13
```

In [ ]:
```
from datetime import timedelta
```

In [ ]:
```
timedelta(12)
```

Out[ ]:
```
datetime.timedelta(days=12)
```

In [ ]:
```
now - timedelta(12)
```

Out[ ]:
```
datetime.datetime(2025, 5, 1, 23, 24, 45, 770412)
```

Datetime and pandas timestamp objects can be formatted as strings using str or the strftime method, passing a format specification:

In [ ]:
```
str(now)
```

Out[ ]:
```
'2025-05-13 23:28:01.530451'
```

In [ ]:
```
now.strftime('%Y-%m-%d')
```

Out[ ]:
```
'2025-05-13'
```

In [ ]:
```
fecha = '2011-01-03'
```

In [ ]:
```
datetime.strptime(fecha, '%Y-%m-%d')
```

Out[ ]:
```
datetime.datetime(2011, 1, 3, 0, 0)
```

Type	Description
%Y	4-digit year
%y	2-digit year
%m	2-digit month [01, 12]
%d	2-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	2-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]

Type	Description
%U	Week number of the year [00, 53]. Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0".
%W	Week number of the year [00, 53]. Monday is considered the first day of the week, and days before the first Monday of the year are "week 0".
%z	UTC time zone offset as +HHMM or -HHMM, empty if time zone naive
%F	Shortcut for %Y-%m-%d, for example 2012-4-18
%D	Shortcut for %m/%d/%y, for example 04/18/12

We can create date ranges by specifying the start and end date

In [ ]:
```
list_of_str_data = ['2024/1/1','2024/12/31']
```

In [ ]:
```
data_pd = pd.to_datetime(list_of_str_data)
```

In [ ]:
```
data_pd
```

Out[ ]:
```
DatetimeIndex(['2024-01-01', '2024-12-31'], dtype='datetime64[ns]', freq=None)
```

Creating date ranges

We will create an index using a date range:

In [ ]:
```
index = pd.date_range('2024/1/1', '2024/12/31')
```

In [ ]:
```
index
```

Out[ ]:
```
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
               '2024-01-05', '2024-01-06', '2024-01-07', '2024-01-08',
               '2024-01-09', '2024-01-10',
               ...
               '2024-12-22', '2024-12-23', '2024-12-24', '2024-12-25',
               '2024-12-26', '2024-12-27', '2024-12-28', '2024-12-29',
               '2024-12-30', '2024-12-31'],
              dtype='datetime64[ns]', length=366, freq='D')
```

The start and end dates set strict boundaries for the date index generated. For example, if you wanted a date index containing the last business day of each month, you would pass the frequency 'BM' (business month end) and only dates that fall within or within the date range would be included:

In [ ]:
```
pd.date_range(start='2024/1/1', periods=20)
```

Out[ ]:
```
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
               '2024-01-05', '2024-01-06', '2024-01-07', '2024-01-08',
               '2024-01-09', '2024-01-10', '2024-01-11', '2024-01-12',
               '2024-01-13', '2024-01-14', '2024-01-15', '2024-01-16',
               '2024-01-17', '2024-01-18', '2024-01-19', '2024-01-20'],
              dtype='datetime64[ns]', freq='D')
```

Alias	Offset Type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1000th of 1 second)
U	Micro	Microsecond (1/1000000th of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
W-MON, W-TUE, ...	Week	Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month. For example, WOM-3FRI for the 3rd Friday of each month.
Q-JAN, Q-FEB, ...	QuarterEnd	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.
BA-JAN, BA-FEB, ...	BusinessYearEnd	Annual dates anchored on last weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

In [ ]:
```
type(index)
```

Out[ ]:
```
pandas.core.indexes.datetimes.DatetimeIndex
```

Importing data from files

Link to Dataset: https://drive.google.com/file/d/15j6VsL8OJVFQo-KFI4HK3Qw-k-tzRAF6/view?usp=sharing

We will use Google Earth Engine datasets and import this data using pandas:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
path = '/content/drive/MyDrive/Datasets_TS/data_et0.csv'
```

In [ ]:
```
data = pd.read_csv(path)
```

In [ ]:
```
data
```

Out[ ]:

	Year	Month	Date	ET0_mm	Unnamed: 4	Unnamed: 5	Unnamed: 6
0	1990	1	1990-02-01	229.1	NaN	NaN	NaN
1	1990	2	1990-03-04	155.9	NaN	NaN	NaN
2	1990	3	1990-04-04	167.9	NaN	NaN	NaN
3	1990	4	1990-05-05	81.3	NaN	NaN	NaN
4	1990	5	1990-06-05	40.9	NaN	NaN	NaN
...	...	...	...	...	...	...	...
367	2020	8	2020-09-05	51.4	NaN	NaN	NaN
368	2020	9	2020-10-06	80.8	NaN	NaN	NaN
369	2020	10	2020-11-06	137.4	NaN	NaN	NaN
370	2020	11	2020-12-07	213.4	NaN	NaN	NaN
371	2020	12	2021-01-01	98.4	NaN	NaN	NaN

372 rows × 7 columns

We have date information and repeated month and year information. Let's remove this information:

In [ ]:
```
data = data.drop(columns=['Year', 'Month', 'Unnamed: 4', 'Unnamed: 5', 'Unnamed:
```

We check if there are any duplicate records:

In [ ]:
```
data['Date'].duplicated().sum()
```

Out[ ]:
```
np.int64(0)
```

Now we convert the date into a datetime object and set it as the index:

In [ ]:
```
data['Date'] = pd.to_datetime(data['Date'])
```

In [ ]:
```
data.set_index('Date', inplace=True)
```

In [ ]:
```
data.index
```

Out[ ]:
```
DatetimeIndex(['1990-02-01', '1990-03-04', '1990-04-04', '1990-05-05',
               '1990-06-05', '1990-07-06', '1990-08-06', '1990-09-06',
               '1990-10-07', '1990-11-07',
               ...
               '2020-04-03', '2020-05-04', '2020-06-04', '2020-07-05',
               '2020-08-05', '2020-09-05', '2020-10-06', '2020-11-06',
               '2020-12-07', '2021-01-01'],
              dtype='datetime64[ns]', name='Date', length=372, freq=None)
```

We can select the values of the ET0_mm column by indexed date ranges:

In [ ]:
```
data['2009-05-01' :'2010-03-01']
```

Out[ ]:

Date	ET0_mm
2009-05-05	120.4
2009-06-05	41.6
2009-07-06	28.2
2009-08-06	27.8
2009-09-06	56.6
2009-10-07	80.6
2009-11-07	181.8
2009-12-08	191.2
2010-01-01	100.8
2010-02-01	265.6

In [ ]:
```
data['2009' :'2010']
```

Out[ ]:

Date	ET0_mm
2009-01-01	92.0
2009-02-01	286.4
2009-03-04	239.4
2009-04-04	147.0
2009-05-05	120.4
2009-06-05	41.6
2009-07-06	28.2
2009-08-06	27.8
2009-09-06	56.6
2009-10-07	80.6
2009-11-07	181.8
2009-12-08	191.2
2010-01-01	100.8
2010-02-01	265.6
2010-03-04	212.6
2010-04-04	144.8
2010-05-05	70.0
2010-06-05	34.4
2010-07-06	17.2
2010-08-06	25.2
2010-09-06	42.2
2010-10-07	84.6
2010-11-07	124.0
2010-12-08	147.2

Resampling and Grouping

We can resample our data temporally. For example, resample by day. This creates sequential dates that we did not have in the original dataset:

In [ ]:
```
res_data = data.resample('D').asfreq()
```

In [ ]:
```
res_data
```

Out[ ]:

Date	ET0_mm
1990-02-01	229.1
1990-02-02	NaN
1990-02-03	NaN
1990-02-04	NaN
1990-02-05	NaN
...	...
2020-12-28	NaN
2020-12-29	NaN
2020-12-30	NaN
2020-12-31	NaN
2021-01-01	98.4

11293 rows × 1 columns

Or by year, applying a form of aggregation, for example the sum:

In [ ]:
```
res_data = data.resample('Y').sum()
```

```
<ipython-input-39-342c1ec8ca5c>:1: FutureWarning: 'Y' is deprecated and will be r
emoved in a future version, please use 'YE' instead.
  res_data = data.resample('Y').sum()
```

In [ ]:
```
res_data
```

Out[ ]:

Date	ET0_mm
1990-12-31	1267.2
1991-12-31	1423.5
1992-12-31	1239.4
1993-12-31	1199.0
1994-12-31	1366.0
1995-12-31	1325.8
1996-12-31	1247.8
1997-12-31	1439.2
1998-12-31	1429.4
1999-12-31	1299.1
2000-12-31	1274.2
2001-12-31	1326.5
2002-12-31	1530.7
2003-12-31	1407.4
2004-12-31	1333.8
2005-12-31	1326.6
2006-12-31	1542.4
2007-12-31	1527.8
2008-12-31	1401.2
2009-12-31	1493.0
2010-12-31	1268.6
2011-12-31	1190.8
2012-12-31	1231.8
2013-12-31	956.4
2014-12-31	1425.0
2015-12-31	1347.4
2016-12-31	1274.8
2017-12-31	1240.6
2018-12-31	1385.6
2019-12-31	1411.2
2020-12-31	1264.2
2021-12-31	98.4

We can now use only the year as an index:

In [ ]:
```
res_data.index = res_data.index.year
```

In [ ]:
```
res_data
```

Out[ ]:

Date	ET0_mm
1990	1267.2
1991	1423.5
1992	1239.4
1993	1199.0
1994	1366.0
1995	1325.8
1996	1247.8
1997	1439.2
1998	1429.4
1999	1299.1
2000	1274.2
2001	1326.5
2002	1530.7
2003	1407.4
2004	1333.8
2005	1326.6
2006	1542.4
2007	1527.8
2008	1401.2
2009	1493.0
2010	1268.6
2011	1190.8
2012	1231.8
2013	956.4
2014	1425.0
2015	1347.4
2016	1274.8
2017	1240.6
2018	1385.6
2019	1411.2
2020	1264.2
2021	98.4

In [ ]:
```
data.to_period('Y')
```

Out[ ]:

Date	ET0_mm
1990	229.1
1990	155.9
1990	167.9
1990	81.3
1990	40.9
...	...
2020	51.4
2020	80.8
2020	137.4
2020	213.4
2021	98.4

372 rows × 1 columns

We can also group the original set by monthly or annual average:

In [ ]:
```
data.groupby(lambda x: x.month).mean()
```

Out[ ]:

Date	ET0_mm
1	93.754839
2	226.429032
3	203.916129
4	143.832258
5	79.177419
6	35.300000
7	25.474194
8	31.709677
9	54.470968
10	93.751613
11	150.829032
12	199.896774

In [ ]:
```
data.groupby(lambda x: x.year).mean()
```

Out[ ]:

Date	ET0_mm
1990	115.200000
1991	118.625000
1992	103.283333
1993	99.916667
1994	113.833333
1995	110.483333
1996	103.983333
1997	119.933333
1998	119.116667
1999	108.258333
2000	106.183333
2001	110.541667
2002	127.558333
2003	117.283333
2004	111.150000
2005	110.550000
2006	128.533333
2007	127.316667
2008	116.766667
2009	124.416667
2010	105.716667
2011	99.233333
2012	102.650000
2013	79.700000
2014	118.750000
2015	112.283333
2016	106.233333
2017	103.383333
2018	115.466667
2019	117.600000
2020	105.350000
2021	98.400000

Visualization

Data represented at a single point in time is known as cross-sectional data. As a data scientist or analyst, you may sometimes come across data collected over periods of time, known as time series data.

Time series data appears in the real world quite frequently. For example, weather readings, company stock prices, and sales data are all examples of data that can be tracked over time. Therefore, it is important that you are able to explore and visualize data with a time component.

Line plot

A line chart is commonly used to visualize time series data.

In [ ]:
```
data
```

Out[ ]:

Date	ET0_mm
1990-02-01	229.1
1990-03-04	155.9
1990-04-04	167.9
1990-05-05	81.3
1990-06-05	40.9
...	...
2020-09-05	51.4
2020-10-06	80.8
2020-11-06	137.4
2020-12-07	213.4
2021-01-01	98.4

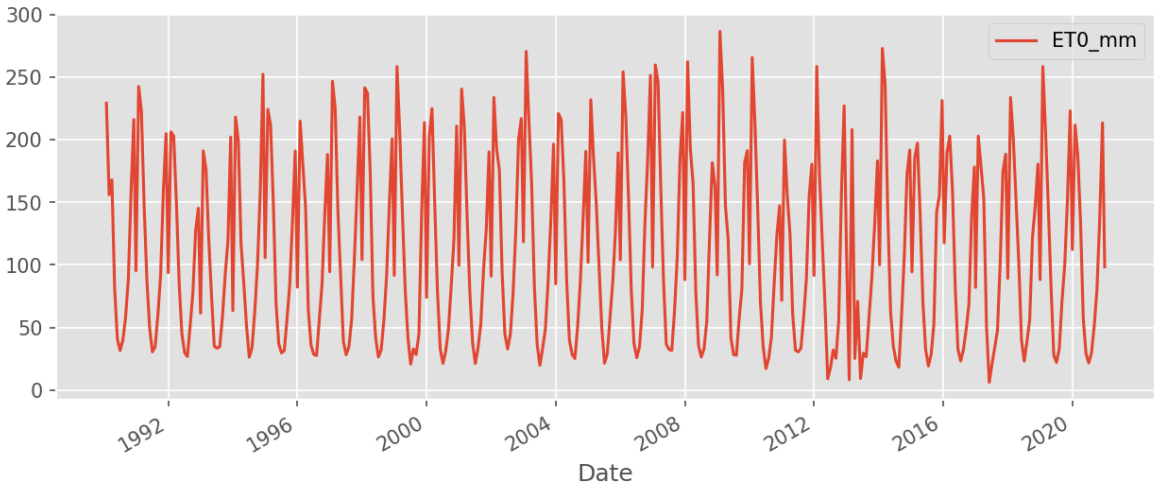
372 rows × 1 columns

In [ ]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

mpl.rcParams['figure.dpi'] = 150

pd.set_option('display.expand_frame_repr', False)
```

```
plt.style.use('ggplot')
mpl.rcParams['figure.figsize'] = (10, 4)
mpl.rcParams['axes.grid'] = True
```

In [ ]:
```
data.plot()
```

Out[ ]: <Axes: xlabel='Date'>

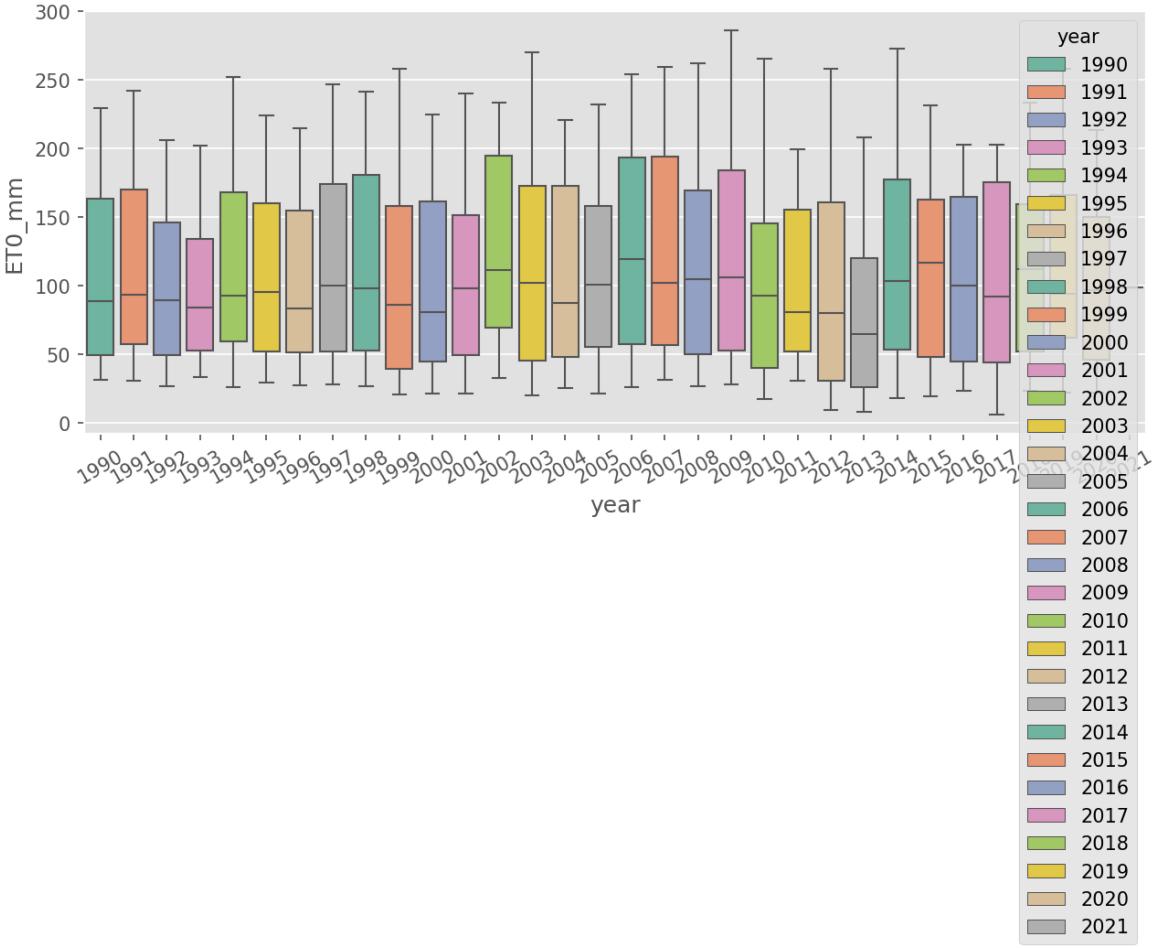


Box plot

When working with time series data, box plots can be useful for seeing the distribution of values grouped by time interval.

For example, let's create a box plot for each year and place them side by side for comparison:

In [ ]:
```
data["year"] = data.index.year

sns.boxplot(data=data, x="year", y="ET0_mm", hue="year", palette = 'Set2')
plt.xticks(rotation=30)
plt.show()
```



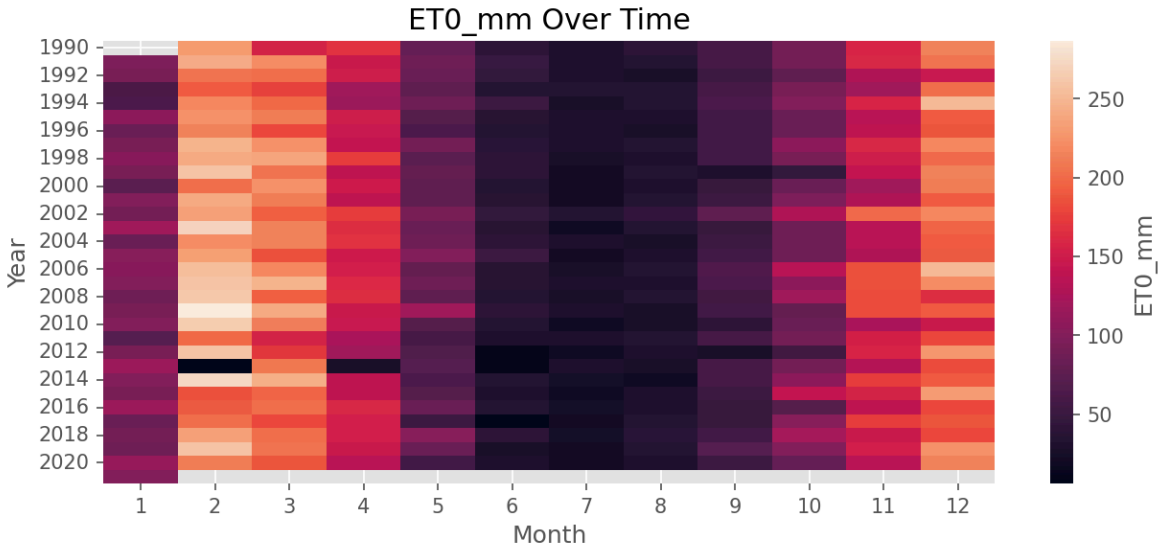
Heatmap

We can also use a heatmap to compare observations across time intervals in time series data.

For example, let's create a density heatmap with year on the y-axis and month on the x-axis. This can be done by calling the heatmap() function of the Seaborn sns object:

In [ ]:
```
data["month"] = data.index.month
```

In [ ]:
```
data_grouped = data.groupby(["year", "month"]).sum()


sales_month_year = data_grouped.reset_index().pivot(index="year", columns="month


sns.heatmap(sales_month_year, cbar_kws={"label": "ET0_mm"})
plt.title("ET0_mm Over Time")
plt.xlabel("Month")
plt.ylabel("Year")
plt.show()
```



Thank you! See you in the next Chapter!