Chapter 2 – Reading Time Series Data

Ingesting Time Series Data

In this chapter we will use the pandas library to read different types of files that holds time series data. We will examine the different parameters available that we can leverage to ensure that our time series data is read in properly.

Technical Requirements

In this chapter we will be using pandas 1.2.3 (released March 02, 2021)

conda install pandas=1.2.3

Reading CSV and other Delimited files

CSV files are probably one of the most common file formats you will encounter when working with time series data, whether searching the internet for open data, or exporting time series data from another platform. We will be using pandas.read\_csv() function and examine the different options provided to us to ensure the data read is suitable for further analysis related to time series data. We will also look into how to create a datetime index from the data that we are reading all in one step.

Getting Ready

We will be reading a CSV file extracted from Box Office Mojo for Avengers: Endgame movie. Here is the URL for the online version of the dataset <https://www.boxofficemojo.com/release/rl3059975681/?ref_=bo_tt_gr_1>

![Graphical user interface, application

Description automatically generated]()

Notice the data format

The file is provided in the GitHub repo for this book which you can find here <URL> the files is named “boxofficemojo\_avengers\_endgame.csv”

In order read local files in pandas, it is important that we know the path to these files, in other words where are they stored in your local machine.

How to do it…

1. Check our current working directory

>>> from pathlib import Path

>>> Path.cwd()

1. Create our Path object

>>> filepath = Path('datasets/boxofficemojo\_avengers\_endgame.csv')

1. Read our CSV file using pandas.read\_csv()

>>> ts = pd.read\_csv(filepath\_or\_buffer=filepath, sep=',', header=0, parse\_dates=[0], index\_col=0)

>>> ts.head()

The following results are returned for the first 5 rows

DOW Rank Daily %± YD %± LW Theaters Avg To Date Day Estimated

Date

2019-04-26 Friday 1 157461641 - - 4662 33775 157461641 1 False

2019-04-27 Saturday 1 109264122 -30.6% - 4662 23437 266725763 2 False

2019-04-28 Sunday 1 90389244 -17.3% - 4662 19388 357115007 3 False

2019-04-29 Monday 1 36874439 -59.2% - 4662 7909 393989446 4 False

2019-04-30 Tuesday 1 33110349 -10.2% - 4662 7102 427099795 5 False

In Jupyter a nicer display which would appear like this

Table

Description automatically generated

1. Lastly, let’s ensure that the index is actually a datetime as we would expect

>>> ts.index

This should return an output like this

DatetimeIndex(['2019-04-26', '2019-04-27', '2019-04-28', '2019-04-29',

'2019-04-30', '2019-05-01', '2019-05-02', '2019-05-03',

'2019-05-04', '2019-05-05',

...

'2019-09-03', '2019-09-04', '2019-09-05', '2019-09-06',

'2019-09-07', '2019-09-08', '2019-09-09', '2019-09-10',

'2019-09-11', '2019-09-12'],

dtype='datetime64[ns]', name='Date', length=140, freq=None)

How it works…

We were able to read the CSV file, convert the data column from a string object to a datetime object and set it as an index to the dataframe all in one step. We leveraged the built-in parse\_dates argument and specified which column we are interested in. In this case, the first column [0]. We then set that column, after being parsed to datetime, as an index.

Let’s break down the arguments we used:

* filepath\_or\_buffer: here we specify our file path. This can also be a URL.
* sep: takes a string to specify the delimiter to use, and the default is “,” which assumes comma separated values (csv). If the file is separated by another delimiter this can be replaced e.g. sep-“|”
  + Another alias to sep is delimiter which can be used as well as an argument
* header: in this case we specified that the first row (0) contains the header information. The default value is “infer” which usually works as is.
* parse\_dates: here we specified that the first column (0) is the column we are intersted in parsing. The argument takes a list of columns e.g. [0,3] indicating first column indexed at 0, and fourth column indexed at 3.

There’s more…

There are situations were parse\_dates may not be sufficient or may not produce the desired results. In most cases, were it fails, the column will be returned unchanged. This where we the date\_parser comes in handy.

We can still use this method on the same dataset, even though it is not needed, it is a good way to demonstrated how it works.

>>> from datetime import datetime

>>> date\_parser = lambda x: datetime.strptime(x, "%b %d, %Y")

>>> ts = pd.read\_csv(filepath\_or\_bufferfilepath, sep=',', header=0, index\_col=0, parse\_dates=[0], date\_parser=date\_parser)

Let’s break it down:

1. Since the date is coming as text in the format “Apr 26, 2019”, we will need to use the date\_parser argument in pandas.read\_csv to convert the sequence of string columns to an array of datetime instances. The date\_parser argument, which is an optional argument, takes a function that will be applied on the specified date column.
2. Using datetime.strptime to create a datetime object from a string. Here we create a lambda function that we can later pass to date\_parser argument in the pandas.read\_csv. For convenience, we saved this function under a variable named date\_parser lambda x: datetime.strptime(x, "%b %d, %Y"). For strptime we provide two arguments, the string date that we want to transform, and the format code.
   1. %b for the abbreviated month name e.g. Apr, May, Jun
   2. %d for the day of the month e.g. 01, 02, ..26, 28
   3. %Y for 4-digit year e.g. 2019, 2020, 2021 ..etc
3. We used the parse\_dates argument and provided a list of columns that we want to parse. In our case, it is just the first column [0]. This is used by date\_parser as a reference to identify which columns we are applying the date\_parser on.
4. Lastly, the index\_col argument let’s panda know which column we want to move as an index. In this case, after the ‘Date’ column is parsed to datetime, it becomes an index for our dataframe and removed from the columns list.

See also

According to pandas’ documentation, infer\_datetime\_format can speed the parsing by 5-10x. Here is how we can add this to our original script

>>> ts = pd.read\_csv(filepath\_or\_buffer=filepath,

sep=',',

header=0,

parse\_dates=[0],

index\_col=0,

infer\_datetime\_format=True)

Note, that given our dataset being small, the improvement in speed may be insignificant.

* For more information please refer to pandas.read\_csv documentation <https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html>
* For more information on Python’s strptime and strftime format codes visit https://strftime.org

Reading data from an Excel file

In business setting you will encounter Excel files that contain time series data. Quite often, these files will contain multiple sheets. Sometimes the data is partitioned in a way that each sheet will contain specific data that belongs to a particular month, quarter, or a year. In This recipe we will be using pandas.read\_excel() function to read in daily sales data. The data contains 2017 and 2018 data, and thus broken into two worksheets for one for each year.

Getting Ready

In order to use pandas.read\_excel() we will need to have a particular library installed that specifies how to read/write excel files. In pandas.read\_excel() this is referenced under the argument engine in which we can specify which engine to use. Hence, the type of library to install will depend on which engine we plan on using.

The supported engines include: xlrd, openpyxl, odf, and pyxlsb. The most common ones when working with Microsoft Excel are usually xlrd and openpyxl.

In pandas, as of version 1.2.0, they made changes were xlrd now only supports .xls files. So, if you are working with an older excel format e.g. .xls then xlrd will do just fine. For newer Excel formats such as .xlsx for instance, we will need a different engine, and in this case openpyxl would be the recommendation to go with.

To install openpyxl

>>> conda install openpyxl

Or

>>> pip install openpyxl

How to do it…

1. Check our current working directory

>>> from pathlib import Path

>>> Path.cwd()

1. Create our Path object

>>> filepath = Path('datasets/sales\_trx\_data.xlsx)

1. Read our .xlxs Excel file using pandas.read\_excel() by default pandas will read for the first sheet since the default value for sheet\_name argument is 0. Since we have two sheets to read from, and our desire is to combine the data into one dataframe for time series analysis we will use two approaches that are pretty similar.
   1. Approach 1: We can specify the sheets that we are interested in by updating the sheet\_name with a list to read both first and second sheets. This will return a dictionary objects with two dataframes.

>>> ts = pd.read\_excel(io= filepath, engine='openpyxl', index\_col=1, sheet\_name=[0,1])

>> ts.keys()

This will print out the keys as follow

dict\_keys([0,1])

We can use pd.concat() to combine (stack) the two dataframes

>> ts\_combined = pd.concat(ts)

This approach is great if we want to specify exactly which sheets we are interested in.

* 1. Approach 2: If we specify sheet\_name = None it will grab all the sheets. We can then concatenate the dataframes all in one step

>> ts = pd.read\_excel(io=filepath, engine='openpyxl', index\_col=1, sheet\_name=None))

>> ts\_combined = pd.concat(ts)

We can combine the two steps above into one step like the following example

>> ts = pd.concat(pd.read\_excel(io=filepath, engine='openpyxl', index\_col=1, sheet\_name=None))

1. To ensure that both sheets have been read, all data is combined we can print a simple summary

>>> ts.info()

This will produce the following output

<class 'pandas.core.frame.DataFrame'>

MultiIndex: 74124 entries, ('sales\_2017', Timestamp('2017-01-01 00:00:00')) to ('sales\_2018', Timestamp('2018-12-31 00:00:00'))

Data columns (total 4 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Line\_Item\_ID 74124 non-null int64

1 Credit\_Card\_Number 74124 non-null int64

2 Quantity 74124 non-null int64

3 Menu\_Item 74124 non-null object

dtypes: int64(3), object(1)

memory usage: 2.5+ MB

How it works…

Reading data from InfluxDB

Time series databases are optimized for time series data and provide improved performance especially when working with large datasets such as IoT data, or sensor data. InfluxDB is a popular open source time series database with a large community base. In this recipe we will be using InfluxDB latest release v2.0.4. The latest InfluxDB introduces the Flux data scripting language, which we will use with the Python API to query the time series data.

Getting ready

In this recipe we will query the NOAA Water Sample data. In Chapter 11, Working with Time Series Databases, we will discuss in more depth how to setup an InfluxDB instance using Docker, load sample data, query the data, and finally write to the database in Python.

In this recipe it assumes a local instance of the InfluxDB is running to demonstrate how to query the databse and convert the output into a Pandas DataFrame for further analysis.

First things first, we will need to install the Python library for InfluxDB.

$ pip install influxdb-client

How to do it…

1. Import our InfluxDBClient

>>> from influxdb\_client import InfluxDBClient

>>> import pandas as pd

>>> import matplotlib.pyplot as plt

1. Will configure our variables: token, org, and bucket

token = "QOWwDHWHJZLNtIq5oqKD8XFdX1LnCtb7wTXxIfJ8TXXtSx7USomYbtLxIaHXQeVEMy82HN5ipyodqUmnxn-Veg=="

org = "my-org"

bucket = "noaa"

1. Establish our connection by providing our url, token, and org

client = InfluxDBClient(url="http://localhost:8086", token=token, org=org)

1. We will instantiate our query\_api

query\_api = client.query\_api()

1. We will then pass our query and request the results to be in a Pandas dataframe format

query = 'from(bucket: "noaa")\

|> range(start: 2019-09-01T00:00:00Z)\

|> filter(fn: (r) => r.\_measurement == "average\_temperature")'

result = client.query\_api().query\_data\_frame(org='my-org', query=query)

1. Let’s inspect our DataFrame

result.info()

which will produce the following output. Pay attention to the data types

<class 'pandas.core.frame.DataFrame'>

Int64Index: 15256 entries, 0 to 15255

Data columns (total 9 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 result 15256 non-null object

1 table 15256 non-null object

2 \_start 15256 non-null datetime64[ns, tzutc()]

3 \_stop 15256 non-null datetime64[ns, tzutc()]

4 \_time 15256 non-null datetime64[ns, tzutc()]

5 \_value 15256 non-null float64

6 \_field 15256 non-null object

7 \_measurement 15256 non-null object

8 location 15256 non-null object

dtypes: datetime64[ns, tzutc()](3), float64(1), object(5)

memory usage: 1.2+ MB

1. We can inspect the first 5 rows of our dataset

result.head()

![Table

Description automatically generated]()

1. Finally, we can properly set our index for the time series data

result.set\_index('\_time', inplace=True)

result.head()

![Table

Description automatically generated]()

1. Finally, we can plot the first 100 records

result.head(100)['\_value'].plot()

plt.show()

or

result.iloc[0:100]['\_value'].plot()

plt.show()

Chart

Description automatically generated

How it works…