QBUS6840: Tutorial 1 – Working with Time-Series

Objectives

- Get familiar with `Dates` and `Times` in Python;
- Get familiar with `Pandas` Time Series indexing;
- Learn concepts of plotting and visualization;

While Python provides a lot of general functionality, it does not provide any more specific functionalities for data manipulation and visualisation.

Therefore, we must use external libraries to get this functionality.

Fortunately, Anaconda pre-installs the relevant libraries for us. They are:

- **Numpy** (mathematics and scientific functions) 数学科学方程式
- **Pandas** (data manipulation)
- **Matplotlib** (visualization) 数据操作

`Pandas` was developed in the context of financial modelling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data.

In this section, we will introduce how to work with each of these types of date/time data in `Pandas`. This short section is by no means a complete guide to the time series tools available in Python or `Pandas`, but instead is intended as a broad overview of how you as a user should approach working with time series. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by `Pandas`. After listing some resources that go into more depth, we will review some short examples of working with time series data in Pandas.

1. Importing libraries

Create a new Python script for this tutorial called "`tutorial_01.py`". Start with the following code

```
import pandas as pd
```

The `import` keyword makes the `pandas` library available, which we can reference as the variable `pd`.

Add the following lines to import `matplotlib` and `numpy`

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

2. Dates and times in Native Python

Python's basic objects for working with dates and times reside in the built-in `datetime` module. Along with the third-party `dateutil` module, you can use it to quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
from datetime import datetime
date = datetime(year=2015, month=7, day=4)
```

Now can you check what type the variable `date` is by typing the following code:

```
print( type(date) )
```

Write what they have seen!

Or you can check the type information from the variable explorer. Ask your tutors how this can be done.

This tells us that each variable in Python must have a type. Knowing the types of variables is very important in programming

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
```

As you can see, this method is much smarter. You don't need tell Python which year, which month and which date. The parser gets understand the sequence of statement "`4th of July, 2015`".

Once you have a `datetime` object, you can do things like printing the day of the week:

```
date.strftime('%A')
```

To understand the meaning of the above code, do a quick search at https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior. Then answer the following questions:

- what is the meaning of strftime for a `datetime` variable?
- what is the meaning of '`%A`'?

3. Typed arrays of times: NumPy's

The weaknesses of Python's `datetime` format inspired the NumPy team to add a set of native time series data type to `NumPy`. The `datetime64` dtype encodes

dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly. The `datetime64` requires a very specific input format:

```
date = np.array('2015-07-04', dtype=np.datetime64)
```

- What is the type of this variable `date`?

Once we have this `date` formatted, however, we can quickly do vectorized operations on it:

```
print(date + np.arange(12))
```

As you can see, you can create a number of dates from a single date.

4. Dates and times in Pandas

`Pandas` builds upon all the tools just discussed to provide `Timestamp` object, which combines the ease-of-use of `datetime` and `dateutil` with the efficient storage and vectorized interface of `numpy.datetime64`. From a group of these `Timestamp` objects, Pandas can construct a `DatetimeIndex` that can be used to index data in a `Series` or `DataFrame`; we'll see many examples of this below.

For example, we can use `Pandas` tools to repeat the demonstration from above. We can parse a flexibly formatted string date, and use format codes to output the day of the week:

```
date = pd.to_datetime("4th of July, 2015")
date.strftime('%A')
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

In here, "`D`" represents 'Days' which denotes the unit of the arg.
You can also try alternative settings such as "`Y`", "`M`" , "`W`" and then compare the corresponding outputs.

You may also find more information on parameter settings for this function in here:
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_timedelta.html

5. Pandas Time Series: Indexing by Time

Where the `Pandas` time series tools really become useful is when you begin to index data by `timestamps`. For example, first we create a `Series` object and use the variable explore to open this series

`Series` is a one-dimensional ndarray data type in `Pandas`. To create a `Series` with 4 elements, you can use the following statement:

```
series_noindex = pd.Series([0, 1, 2, 3])
```

Write down what the index is in this series as you see from the pop-up window when you click on the variable `series_noindex` in the variable explorer.

Now we construct another `Series` object that has time indexed data:

```
index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                          '2015-07-04', '2015-08-04'])
date_timeindex = pd.Series([0, 1, 2, 3], index=index)
```

Check the index of `date_timeindex`. Have you seen the difference from the index of `series_noindex`?

Now that we have this data in a `Series`, we can make use of any of the `Series` indexing patterns we discussed in previous sections, passing values that can be coerced into dates:

```
date_timeindex['2014-07-04':'2015-07-04']
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
date_timeindex['2015']
```

6. Play with real data: `AirPassengers` dataset

Since you have learnt the basics of processing the date and time in Python, now let's use Python to visualize some real time series data.

Time Series data is a collection of data points collected at constant time intervals. These are analyzed to determine the long term trend so as to forecast the future or perform some other form of analysis. But what makes a Time Series different from say a regular regression problem? There are 2 things:

1. It is time dependent. So the basic assumption of a linear regression model that the observations are independent doesn't hold in this case.
2. Along with an increasing or decreasing trend, most Time Series have some form of seasonality trends, i.e. variations specific to a particular time frame. For example, if you see the sales of a woolen jacket over time, you will invariably find higher sales in winter seasons.

Because of the inherent properties of a Time Series, there are various steps involved in analyzing it. These are discussed in detail below. Let's start by loading a Time Series object in Python. We'll be using the popular `AirPassengers` data set

which can be downloaded from Canvas.

6.1 Loading the dataset

Before you start loading the dataset, please make sure you have successfully load the necessary libraries `Numpy`, `Pandas`, and `matplotlib`. If you haven't do so, please go back to **step 1.**

Then you can load the data set and look at some initial rows and data types of the columns:

```
data = pd.read_csv('AirPassengers.csv')
print(data.head())
print( '\n Data Types:')
print(data.dtypes)
```

The data contains a particular month and number of passengers travelling in that month. But this is still not read as a Time Series object as the data types are '`object`' and '`int`'. (Recall how to check type information?)

Now use variable explorer to explore `data` variable and answer the following questions:

(1) What is the type of the variable `data`?
(2) How many columns in the `DataFrame` variable `data`?
(3) What are the index values? Is this in a good time series format?

In order to read the data as a time series, we have to pass special arguments to the `read_csv` command: Carefully read the code below and ask your questions until you fully understand the meaning of each statement.

```
dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m')

data_time = pd.read_csv('AirPassengers.csv',
                parse_dates=['Month'],
                index_col='Month',date_parser=dateparse)

print(data_time.head())
```

Let's understand the arguments one by one:

1. `parse_dates`: This specifies the column which contains the date-time information. As we say above, the column name is '`Month`'.

2. `index_col`: A key idea behind using Pandas for TS data is that the index has to be the variable depicting date-time information. So this argument tells pandas to use the '`Month`' column as index.

3. `date_parser`: This specifies a function which converts an input string into

datetime variable. Be default Pandas reads data in format 'YYYY-MM-DD HH:MM:SS'. If the data is not in this format, the format has to be manually defined. Something similar to the dataparse function defined here can be used for this purpose.

Can you compare this new variable `data_time` with the previously loaded variable `date`? Their difference? Answer the following questions

(1) How many columns in `data_time`? What are they?
(2) What is the index of the variable `data_time`? What are their values.

Now we can see that the data has time object as index and #Passengers as the column. We can cross-check the datatype of the index with the following command:

```
print(data.index)
```

Within the terminal output, you should see the `DatetimeIndex` information.
Notice ***the dtype='datetime[ns]'*** which confirms that it is a `datetime` object. As a personal preference, I would convert the column into a Series object to prevent referring to columns names every time when I use the TS. Please feel free to use as a `dataframe` is that works better for you.

Note: When we load a time series from a csv file with different date form, we shall provide an appropriate parser according to the actual time format in a csv file. For example, if the month in the `AirPassengers.csv` is in the form of such as 04-1950, how do you change your parser?

6.2 Indexing the dataset

Let's start by selecting a particular value in the `Series` object.
If you still remember step 5, we could use `Pandas` to index the dates and time. Or we can use Python native method `datetime` library to do the indexing as well:

```
ts = data['Passengers']
ts['1949-01-01']

from datetime import datetime
ts[datetime(1949,1,1)]
```

6.3 Visualize your data

Finally, we will visualize our data. In here, we will use the `matplotlib`:

```
ts = data['Passengers']
plt.plot(ts)
```

You can also plot a sub section of your entire dataset by defining corresponding indexing rule. For example:

```
ts1 = ts['1949']
plt.figure()
plt.plot(ts1)

ts2 = ts['1949-01-01':'1949-05-01']
plt.figure()
plt.plot(ts2)
```

Congratulations, you have learnt the Time Series data preprocessing in Python! In the next tutorial, we will learn the data manipulations in Time Series data.

The full answer is in `LabW01.py`

If you are interested in topic and wish to learn more, please go to https://towardsdatascience.com/basic-time-series-manipulation-with-pandas-4432afee64ea