How To Plot Time Series

Nadim Kawwa Nov 3, 2019 · 5 min read ★

Photo by Chris Lawton on Unsplash

Dealing with time series can be one of the most insightful parts of exploratory data analysis, if done right. In This post, we are going to use the checkin log from the Yelp Dataset to explore trends across different time periods using Pandas and Matplotlib.

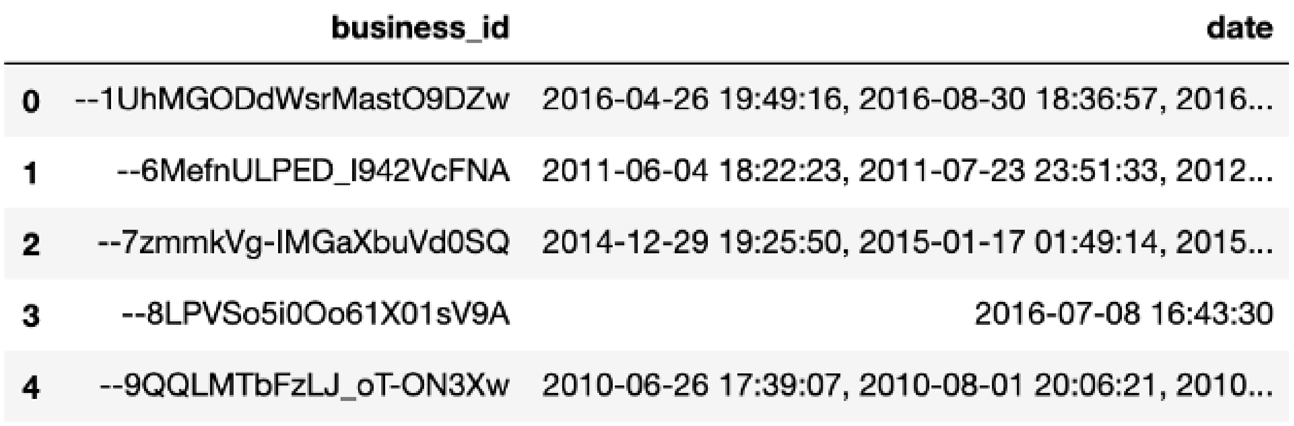
Data Acquisition

After downloading the data, we need to know what to use. The block below shows a sample entry from the `checkin.json` file based on the Yelp Documentation:

```
{
    // string, 22 character business id, maps to business in business.json
    "business_id": "tnhfDv5Il8EaGSXZGiuQGg"

    // string which is a comma-separated list of timestamps for each checkin, each with format YYYY-MM-DD HH:MM:SS
    "date": "2016-04-26 19:49:16, 2016-08-30 18:36:57, 2016-10-15 02:45:18, 2016-11-18 01:54:50, 2017-04-20 18:39:06, 2017-05-03 17:58:02"
}
```

We can read the input file with pandas read_json method with arguments `orient=columns` and `Lines=True`.

Upon reading the data, our dataframe looks something like this:

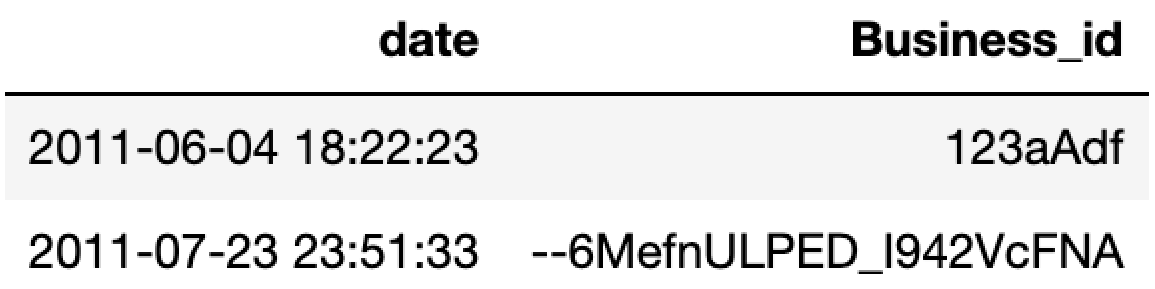


	business_id	date
0	--1UhMGODdWsrMastO9DZw	2016-04-26 19:49:16, 2016-08-30 18:36:57, 2016...
1	--6MefnULPED_I942VcFNA	2011-06-04 18:22:23, 2011-07-23 23:51:33, 2012...
2	--7zmmkVg-IMGaXbuVd0SQ	2014-12-29 19:25:50, 2015-01-17 01:49:14, 2015...
3	--8LPVSo5i0Oo61X01sV9A	2016-07-08 16:43:30
4	--9QQLMTbFzLJ_oT-ON3Xw	2010-06-26 17:39:07, 2010-08-01 20:06:21, 2010...

The `date` column entries are strings such that each date is separated by a comma. By looking at them we can tell that the format is indeed `YYYY-MM-DD HH:MM:SS`.

Data Wrangling

For readability, we want our dataframe to look more like this:



date	Business_id
2011-06-04 18:22:23	123aAdf
2011-07-23 23:51:33	--6MefnULPED_I942VcFNA

The unfolded event log makes more sense from an EDA perspective whereas the format in the JSON format makes more sense for memory storage.

We can obtain this result by creating a dictionary where the keys correspond to dates and the values correspond to business_id. Here we assume that the dates are granular enough so that no two dates are the same. Keep in mind that the python documentation does not allow duplicate keys within a single dictionary.

The implementation can be done as follows:

```python
#create dict that maps ID
bus_to_check = {}

for bus, dates in zip(df['business_id'], df['date']):
    dates_as_list = dates.split(',')
    bus_to_check[bus] = pd.to_datetime(dates_as_list)

#create the dict with dates as keys
check_to_bus = dict( (v,k) for k in bus_to_check for v in bus_to_check[k])

#create dataframe from dict
event_df = pd.DataFrame.from_dict(check_to_bus, orient='index', columns=['business_id'])
#sort by the index for neatness
event_df = event_df.sort_index(ascending=True)
```

df_to_eventlog_yelp.py hosted with ♡ by GitHub view raw

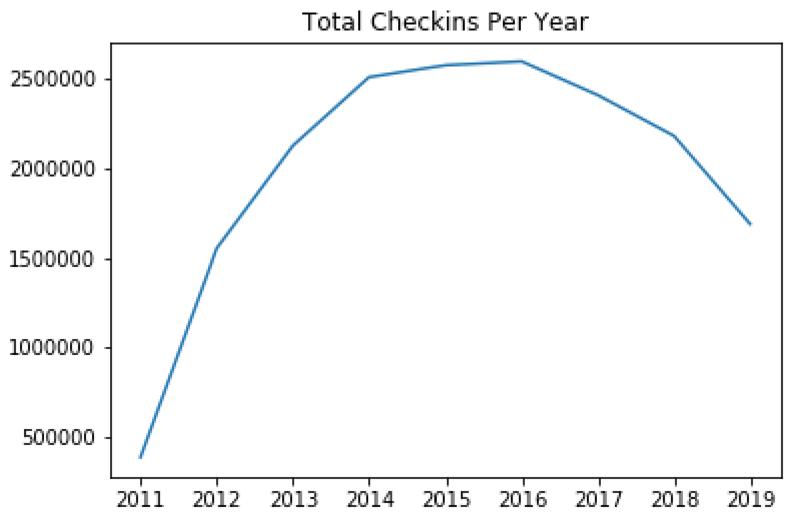
Seasonality

Say we want to know what are the total checkins for all the years available. We will use this as a gateway to introduce the pandas Grouper which can be used inside the groupby method. For more information about frequency aliases refer to the pandas docs.

```python
#sum over years
checkin_year_sigma = event_df.groupby(pd.Grouper(freq='Y')).count()
#plot year v total checkins
plt.plot(checkin_year_sigma.index, checkin_year.business_id)
```

yelp_groupby_year_simga.py hosted with ♡ by GitHub view raw

The code returns this neat plot:



it is a good start, however what if we want to dive deeper? We will zoom in on the year 2014, though any other year will do. Selecting date ranges is an easy one-liner:

```python
#get 2014-2015 date indeces
df_2014 = checkin_day[(checkin_day.index>'2014-01-01') & (checkin_day.index<'2015-01-01')]
```

yelp_select_year_2014.py hosted with ♡ by GitHub view raw

A first indicator of seasonality is to look at the weekend, since the data is from the USA & Canada the weekend is Saturday and Sunday. Bear in mind that the weekend varies by country, for example Dubai's weekend is Friday and Saturday.

We therefore want to detect the days that are a weekend and write a customizable function for it:

```python
def find_weekend_indices(datetime_array, weekend=5):
    """
    Returns all indices of Saturdays & Sundays in a datetime array
    datetime_array(pandas) = pandas datetime array
    weekend(int) = assume weekend starts at day=5=Saturday
    """
    #empty list to tore indeces
    indices = []


    for i in range(len(datetime_array)):
        #  get day of the week with Monday=0, Saturday=5, Sunday=6
        if datetime_array[i].weekday() >= weekend:
            indices.append(i)

    return indices
```

find_weekend_indices.py hosted with ♡ by GitHub view raw

The function above returns indices that are weekends. Next for plotting we define another helper function to highlight spans of the plot that would correspond to these indices with the help of pyplot axvspan:

```python
def highlight_datetimes(indices, ax, df, facecolor='green', alpha_span=0.2):
    """
    Highlights all weekends in an axes object
    indices(list) = list of Saturdays and Sundays indeces corresponding to dataframe
    ax(matplot) = pyplot object
    df(pandas) = pandas dataframe
    """
    i = 0
    #iterate over indeces
    while i < len(indices)-1:
        #highlight from i to i+1
        ax.axvspan(df.index[indices[i]],
                   df.index[indices[i] + 1],
                   facecolor=facecolor,
                   edgecolor='none',
                   alpha=alpha_span)
        i += 1
```

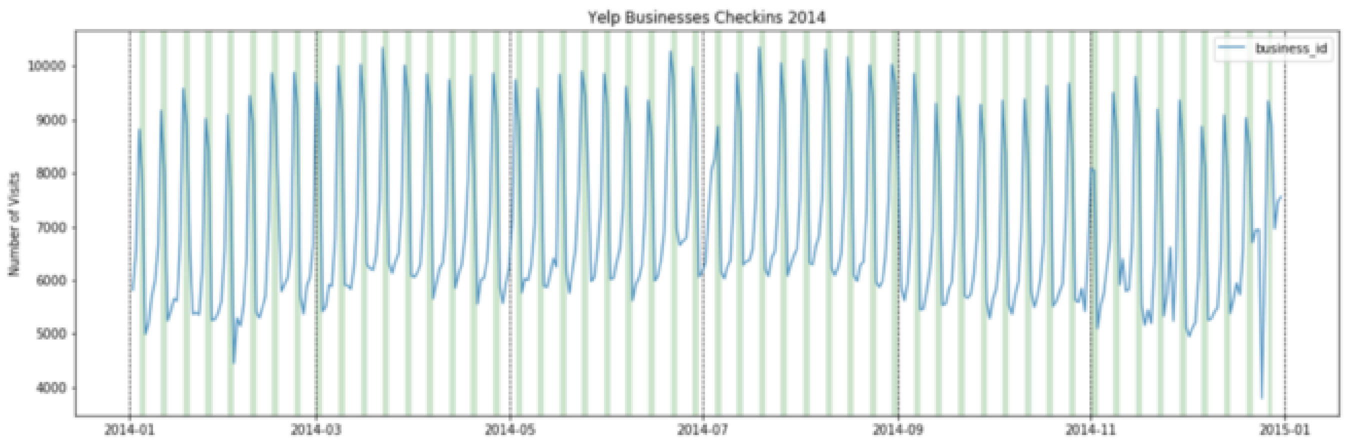
highlight_datetimes.py hosted with ♡ by GitHub view raw

Finally we can use these two functions inside a larger one that takes in a dataframe:

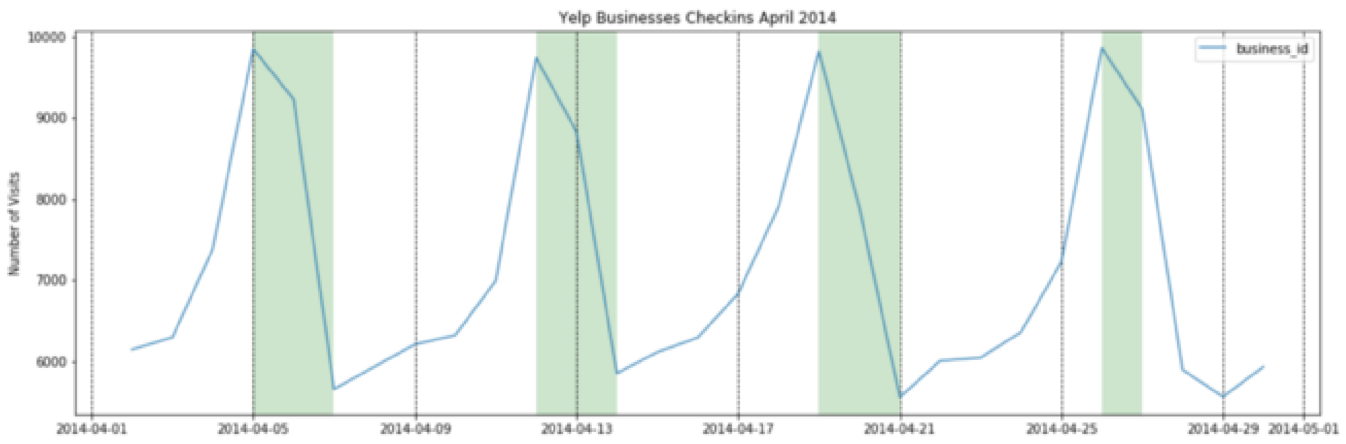
```
def plot_datetime(df, title="Yelp Businesses Checkins",
                  highlight=True,
                  weekend=5,
                  ylabel="Number of Visits",
                  saveloc=None,
                  facecolor='green',
                  alpha_span=0.2):
    """
    Draw a plot of a dataframe that has datetime object as its index
    df(pandas) = pandas dataframe with datetime as indeces
    highlight(bool) = to highlight or not
    title(string) = title of plot
    saveloc(string) = where to save file
    """

    #instantiate fig and ax object
    fig, axes = plt.subplots(nrows=1, ncols=1, sharex=True,figsize=(15,5))

    #draw all columns of dataframe
    for v in df.columns.tolist():
        axes.plot(df[v], label=v, alpha=.8)

    if highlight:
        #find weekend indeces
        weekend_indices = find_weekend_indices(df.index, weekend=5)
        #highlight weekends
        highlight_datetimes(weekend_indices, axes, df, facecolor)

    #set title and y label
    axes.set_title(title, fontsize=12)
    axes.set_ylabel(ylabel)
    axes.legend()
    plt.tight_layout()

    #add xaxis gridlines
    axes.xaxis.grid(b=True, which='major', color='black', linestyle='--', alpha=1)

    #savefig if
    if saveloc:
        fig.savefig(saveloc)
    plt.show()
```

plot_datetime.py hosted with ♡ by GitHub view raw

The result is this neat plot:



The plot above matches the intuition that people go out more on weekends than weekdays. To take a better look at that buildup we can take a closer look at April 2014 using the same function.



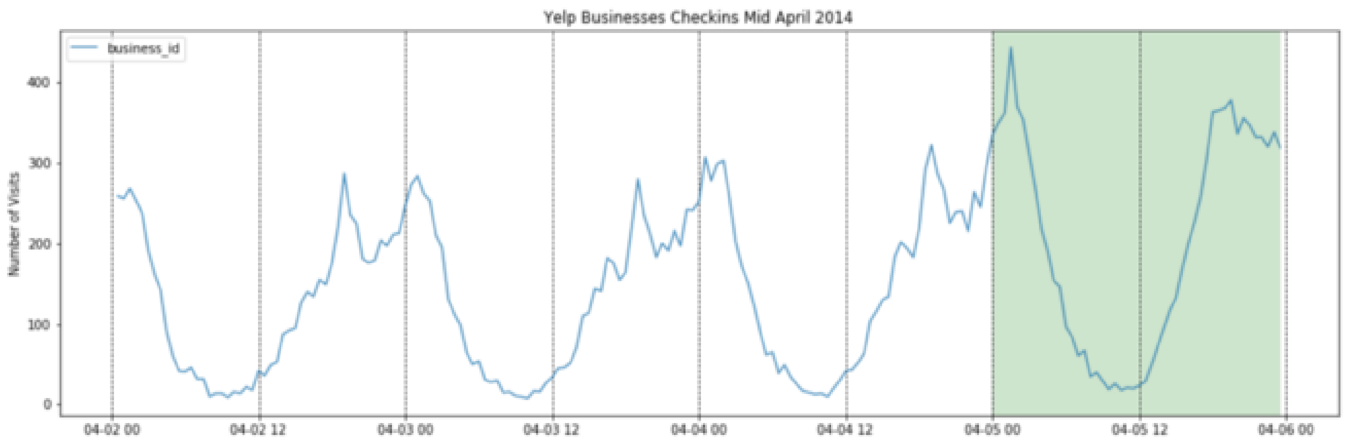
What if we want to show a more granular view of April? Since each entry in the original dataframe is a single checkin, we can groupby 30 minute frequencies and count as follows:

```
#get checkins by 30min intervals
checkin_halfhour = event_df.groupby(pd.Grouper(freq='30min')).count()
```

```
#select interal
checkin_halfhour_midapril = checkin_halfhour[(checkin_halfhour.index>'2014-04-02 00:00:00')
                                             & (checkin_halfhour.index<'2014-04-06 00:00:00')]
#plot
plot_help.plot_datetime(checkin_halfhour_midapril,
                        title='Yelp Businesses Checkins Mid April 2014',
                        saveloc='plots/checkin_2014_april_30min.png')
```

yelp_midapril_30min.py hosted with ♡ by GitHub view raw

The result is the plot below:



Plotting Confidence Intervals

We saw how we can zoom in on time periods to get a sense of seasonality. However, businesses also want to know what is the expected number of checkins for a given day and how it might vary. For example, we might be solving for a regression.

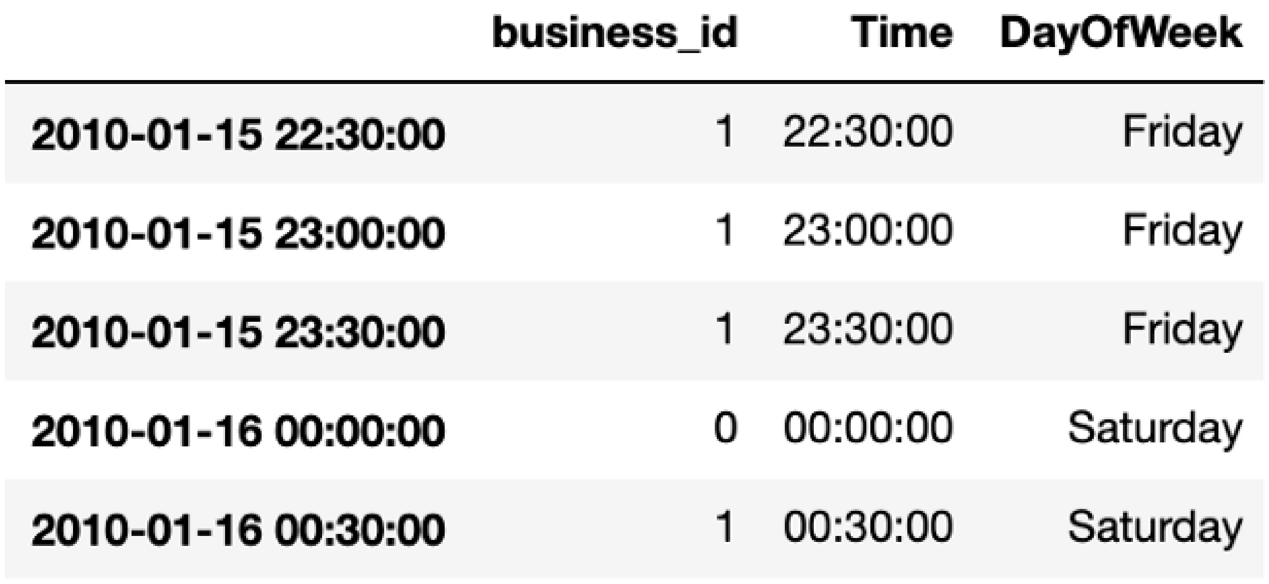
Therefore we need to build plots that show the average checkin for a given day of the week and some sort of confidence interval.

We will use the `checkin_halfhour` defined in the previous section to extract the day of the week:

```
# add a column including time only
checkin_halfhour['Time']=checkin_halfhour.index.time

#column for name of day of week
checkin_halfhour['DayOfWeek'] = checkin_halfhour.index.weekday_name
```

checkin_halfhour_dayofweek.py hosted with ♡ by GitHub view raw

Our dataframe now looks as follows:



	business_id	Time	DayOfWeek
2010-01-15 22:30:00	1	22:30:00	Friday
2010-01-15 23:00:00	1	23:00:00	Friday
2010-01-15 23:30:00	1	23:30:00	Friday
2010-01-16 00:00:00	0	00:00:00	Saturday
2010-01-16 00:30:00	1	00:30:00	Saturday

Next, we build the `day_avg` dataframe using pandas aggregate method. For our purposes we will aggregate:

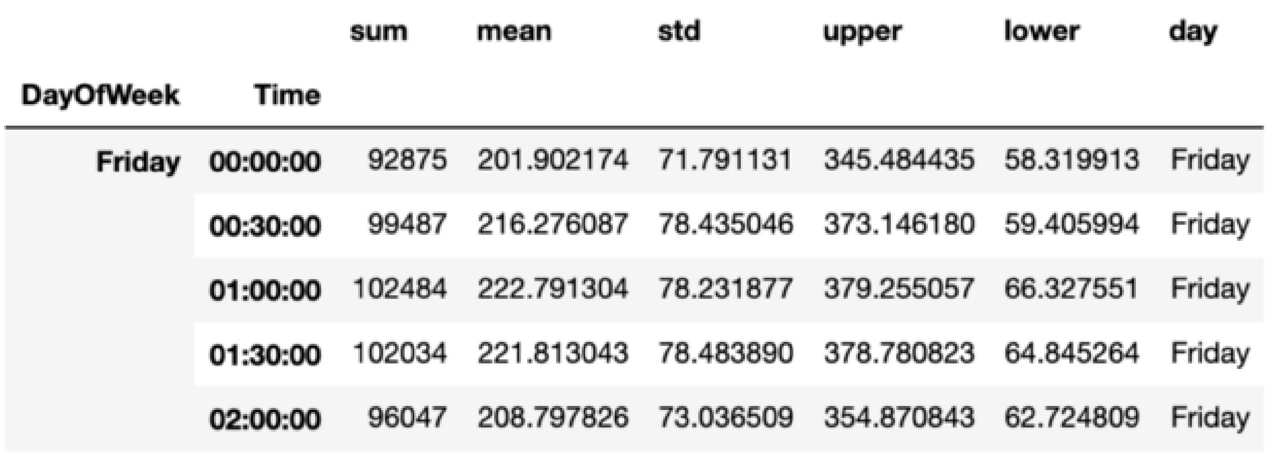
- Sum
- Mean
- Standard Deviation

The code block below does this for us:

```
#group by day of week and aggregate by sum+mean+stf
day_avg=checkin_halfhour.groupby(['DayOfWeek','Time']).agg([np.sum, np.mean, np.std])

#drop level in colums
day_avg.columns = day_avg.columns.droplevel()

#extract upper and lower bound as mean +/- 2STD
day_avg['upper'] = day_avg['mean'] + 2*day_avg['std']
day_avg['lower'] = day_avg['mean'] - 2*day_avg['std']

#extract day of week name
day_avg['day']=day_avg.reset_index()['DayOfWeek'].tolist()
```

yelp_day_avg.py hosted with ♡ by GitHub view raw

We now have a dataframe we can use to draw intervals:



		sum	mean	std	upper	lower	day
DayOfWeek	**Time**						
Friday	**00:00:00**	92875	201.902174	71.791131	345.484435	58.319913	Friday
	00:30:00	99487	216.276087	78.435046	373.146180	59.405994	Friday
	01:00:00	102484	222.791304	78.231877	379.255057	66.327551	Friday
	01:30:00	102034	221.813043	78.483890	378.780823	64.845264	Friday
	02:00:00	96047	208.797826	73.036509	354.870843	62.724809	Friday

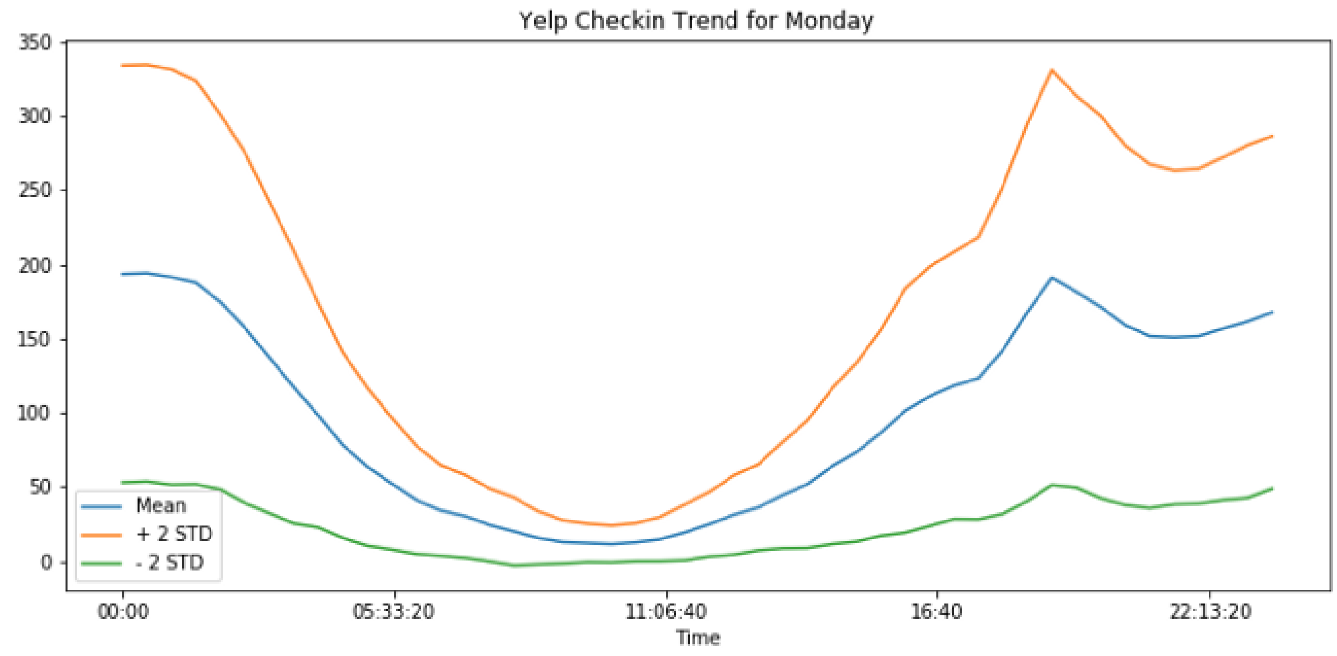
How we define the interval depends on the distribution of the data and the business value of narrow/larger intervals. In our case, we want to generate the expected checkin for each day of the week and grab all the information within two standard deviations.

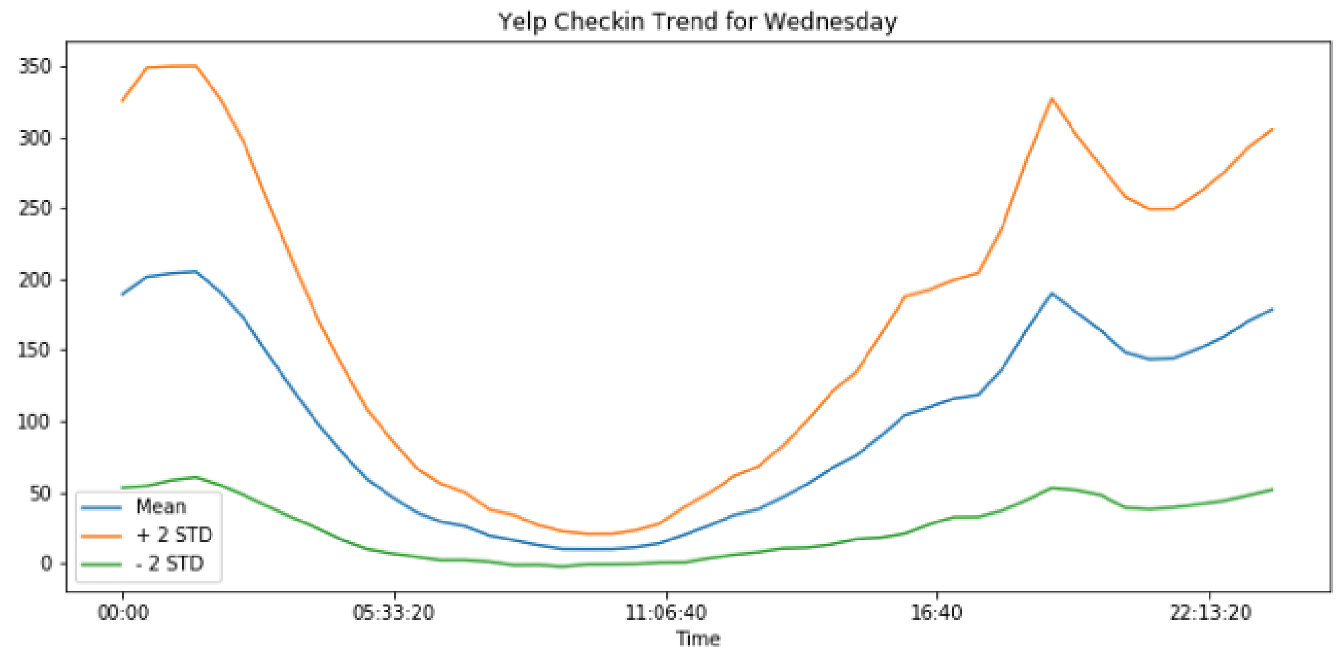
Doing so is a matter of watching out for the levels in the index of `day_avg` as shown below:

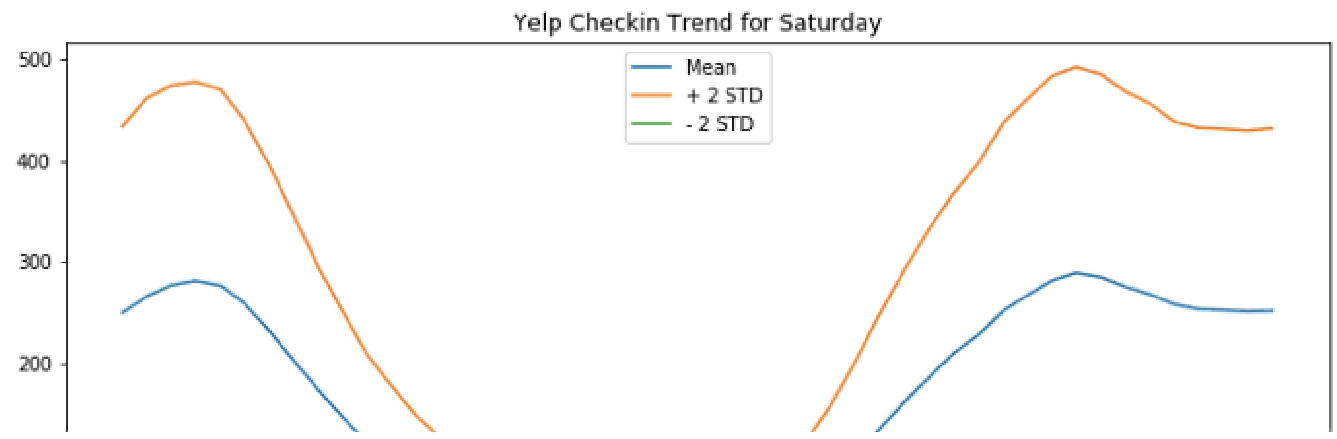
```python
#days of week
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

for day in days:
    ax=day_avg[day_avg['day']==day].unstack(level=0)['mean'].plot(figsize=(10,5))
    day_avg[day_avg['day']==day].unstack(level=0)['upper'].plot(ax=ax)
    day_avg[day_avg['day']==day].unstack(level=0)['lower'].plot(ax=ax)
    ax.set_title('Yelp Checkin Trend for {}'.format(day))
    ax.legend(['Mean','+ 2 STD','- 2 STD'])
    plt.tight_layout()
    plt.savefig('plots/checkin_trend_'+day+'.png')
```

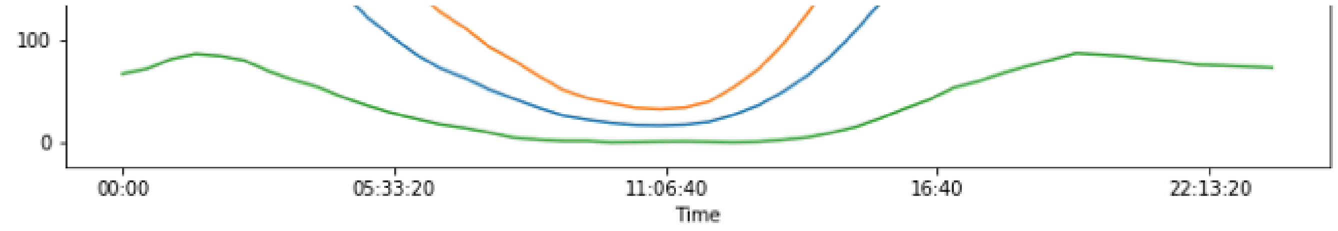
yelp_checkin_expected_dayofweek.py hosted with ♡ by GitHub view raw

The resulting plots reveal a lot in terms of checkin behavior, for illustration we show Monday, Wednesday, and Saturday.


Yelp Checkin Trend for Monday
Mean
+ 2 STD
- 2 STD
Time
00:00
05:33:20
11:06:40
16:40
22:13:20
0
50
100
150
200
250
300
350


Yelp Checkin Trend for Wednesday
Mean
+ 2 STD
- 2 STD
Time
00:00
05:33:20
11:06:40
16:40
22:13:20
0
50
100
150
200
250
300
350


Yelp Checkin Trend for Saturday
Mean
+ 2 STD
- 2 STD
200
300
400
500



Conclusion

The amount and depth of time series analysis depends on the problem we are trying to solve for. For example, we might want to evaluate restaurants, or businesses that only open from 8AM to 3PM such as cafés.

An objective of EDA is to prepare for feature extraction. With the exercise above we can start thinking about features such as the most common checkin time, or checkins by day.

Data Science Python Education Business Intelligence Yelp