Getting started with Matplotlib

To help find all the possible properties of each of your plotting objects, simply make a call to the properties method, which displays all of them as a dictionary. Let's see a curated list of the properties of an axis object:

```
>>> ax.xaxis.properties()
{'alpha': None,
'gridlines': <a list of 4 Line2D gridline objects>,
'label': Text(0.5,22.2,'X Axis'),
'label_position': 'bottom',
'label_text': 'X Axis',
'tick_padding': 3.5,
'tick_space': 26,
'ticklabels': <a list of 4 Text major ticklabel objects>,
'ticklocs': array([ 0.2 , 0.4 , 0.55, 0.93]),
'ticks_position': 'bottom',
'visible': True}
```

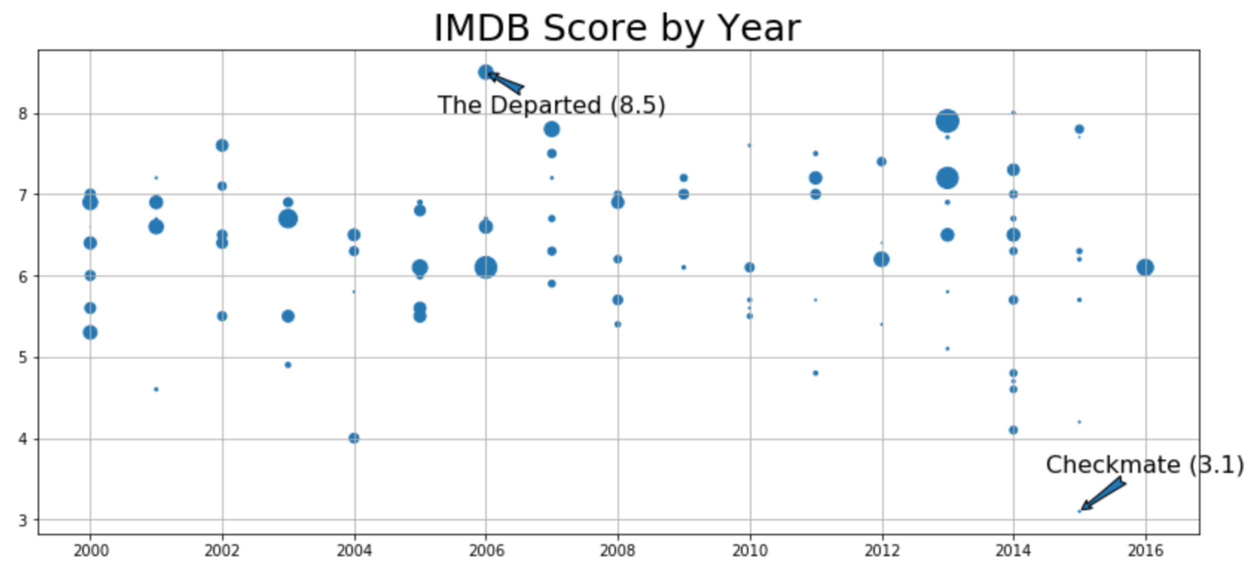
See also:

- Matplotlib official documentation of its usage guide (http://bit.ly/2xrKjeE)
- Categorized list of all the methods of an Axes object (http://bit.ly/2kEhi9w)
- *Anatomy of Matplotlib* tutorial by key contributor, Ben Root (http://bit.ly/2y86c1M)
- Matplotlib official documentation of the stateful pyplot module and the object-oriented approach (http://bit.ly/2xqYnVR)
- Matplotlib official documentation of the *Artist tutorial* (http://bit.ly/2kwS2SI)

Visualizing data with matplotlib

Matplotlib began accepting pandas DataFrames for all of its plotting functions after the release of version 1.5. The DataFrame gets passed to the plotting method through the data parameter. Doing so allows you to reference the columns with string names. The following script creates a scatter plot of the IMDB score against the year for a random selection of 100 movies made from 2000 onwards. The sizes of each point are proportional to the budget:

```
>>> cols = ['budget', 'title_year', 'imdb_score', 'movie_title']
>>> m = movie[cols].dropna()
>>> m['budget2'] = m['budget'] / 1e6
>>> np.random.seed(0)
>>> movie_samp = m.query('title_year >= 2000').sample(100)
>>> fig, ax = plt.subplots(figsize=(14,6))
>>> ax.scatter(x='title_year', y='imdb_score',
s='budget2', data=movie_samp)
>>> idx_min = movie_samp['imdb_score'].idxmin()
>>> idx_max = movie_samp['imdb_score'].idxmax()
>>> for idx, offset in zip([idx_min, idx_max], [.5, -.5]):
year = movie_samp.loc[idx, 'title_year']
score = movie_samp.loc[idx, 'imdb_score']
title = movie_samp.loc[idx, 'movie_title']
ax.annotate(xy=(year, score),
xytext=(year + 1, score + offset),
s=title + ' ({})'.format(score),
ha='center',
size=16,
arrowprops=dict(arrowstyle="fancy"))
>>> ax.set_title('IMDB Score by Year', size=25)
>>> ax.grid(True)
```



After creating the scatter plot, the highest and lowest scoring movies are labeled with the annotate method. The xy parameter is a tuple of the point that we would like to annotate. The xytext parameter is another tuple coordinate of the text location. The text is centered there due to ha being set to center.

See also:

- Matplotlib official *Legend guide* (http://bit.ly/2yGvKUu)
- Matplotlib official documentation of the scatter method (http://bit.ly/2i3N2nI)
- Matplotlib official *Annotation* guide (http://bit.ly/2yhYHoP)

Understanding the difference between Python and pandas date tools

The date formatting directive can actually make quite a large difference when converting a large sequence of strings to Timestamps. Whenever pandas uses to_datetime to convert a sequence of strings to Timestamps, it searches a large number of different string combinations that represent dates. This is true even if all the strings have the same format. With the format parameter, we can specify the exact date format, so that pandas doesn't have to search for the correct one each time. Let's create a list of dates as strings and time their conversion to Timestamps both with and without a formatting directive:

```
>>> date_string_list = ['Sep 30 1984'] * 10000
>>> %timeit pd.to_datetime(date_string_list, format='%b %d %Y')
35.6 ms ± 1.47 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> %timeit pd.to_datetime(date_string_list)
1.31 s ± 63.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Providing the formatting directive resulted in a 40 times improvement in performance.

See also:

- Python official documentation of the datetime module (http://bit.ly/2xIjd2b)
- Pandas official documentation for *Time Series* (http://bit.ly/2xQcani)
- Pandas official for *Time Deltas* (http://bit.ly/2yQTVMQ)

Slicing time series intelligently

Our original crimes DataFrame was not sorted and slicing still worked as expected. Sorting the index will lead to large gains in performance. Let's see the difference with slicing done from step 8:

```
>>> %timeit crime.loc['2015-3-4':'2016-1-1']
39.6 ms ± 2.77 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> crime_sort = crime.sort_index()
>>> %timeit crime_sort.loc['2015-3-4':'2016-1-1']
758 µs ± 42.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The sorted DataFrame provides an impressive 50 times performance improvement over the original.

Using methods that only work with a DatetimeIndex

We begin using the simple first method, but with a complicated parameter offset. It must be a DateOffset object or an offset alias as a string. To help understand DateOffset objects, it's best to see what they do to a single Timestamp. For example, let's take the first element of the index and add six months to it in two different ways:

```
>>> first_date = crime_sort.index[0]
>>> first_date
Timestamp('2012-01-02 00:06:00')
>>> first_date + pd.offsets.MonthBegin(6)
Timestamp('2012-07-01 00:06:00')
>>> first_date + pd.offsets.MonthEnd(6)
Timestamp('2012-06-30 00:06:00')
```

Both the MonthBegin and MonthEnd offsets don't add or subtract an exact amount of time but effectively round up to the next beginning or end of the month regardless of what day it is. Internally, the first method uses the very first index element of the DataFrame and adds the

DateOffset passed to it. It then slices up until this new date. For instance, step 4 is equivalent to the following:

```
>>> step4 = crime_sort.first(pd.offsets.MonthEnd(6))
>>> end_dt = crime_sort.index[0] + pd.offsets.MonthEnd(6)
>>> step4_internal = crime_sort[:end_dt]
>>> step4.equals(step4_internal)
True
```

Steps 5 through 7 follow from this preceding equivalence directly. In step 8, offset aliases make for a much more compact method of referencing DateOffsets. The counterpart to the first method is the last method, which selects the last *n* time segments from a DataFrame given a DateOffset.

It is possible to build a custom DateOffset when those available don't exactly suit your needs:

```
>>> dt = pd.Timestamp('2012-1-16 13:40')
>>> dt + pd.DateOffset(months=1)
Timestamp('2012-02-16 13:40:00')
```

Notice that this custom DateOffset increased the Timestamp by exactly one month. Let's look at one more example using many more date and time components:

```
>>> do = pd.DateOffset(years=2, months=5, days=3,
hours=8, seconds=10)
>>> pd.Timestamp('2012-1-22 03:22') + do
Timestamp('2014-06-25 11:22:10')
```

See also:

- Pandas official documentation of *DateOffsets objects* (http://bit.ly/2fOintG)

Counting the number of weekly crimes

It is possible to use resample even when the index does not contain a Timestamp. You can use the on parameter to select the column with Timestamps that will be used to form groups:

```
>>> crime = pd.read_hdf('data/crime.h5', 'crime')
>>> weekly_crimes2 = crime.resample('W', on='REPORTED_DATE').size()
>>> weekly_crimes2.equals(weekly_crimes)
True
```

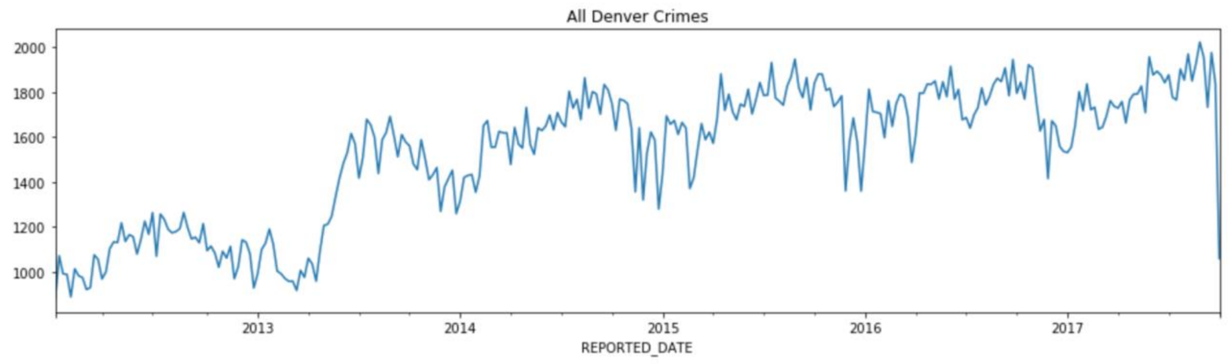
Similarly, this is possible using groupby with pd.Grouper by selecting the Timestamp column with the key parameter:

```
>>> weekly_crimes_gby2 = crime.groupby(pd.Grouper(key='REPORTED_DATE',
freq='W')).size()
```

```
>>> weekly_crimes_gby2.equals(weekly_crimes_gby)
True
```

We can also easily produce a line plot of all the crimes in Denver (including traffic accidents) by calling the plot method on our Series of weekly crimes:

```
>>> weekly_crimes.plot(figsize=(16, 4), title='All Denver Crimes')
```



See also:

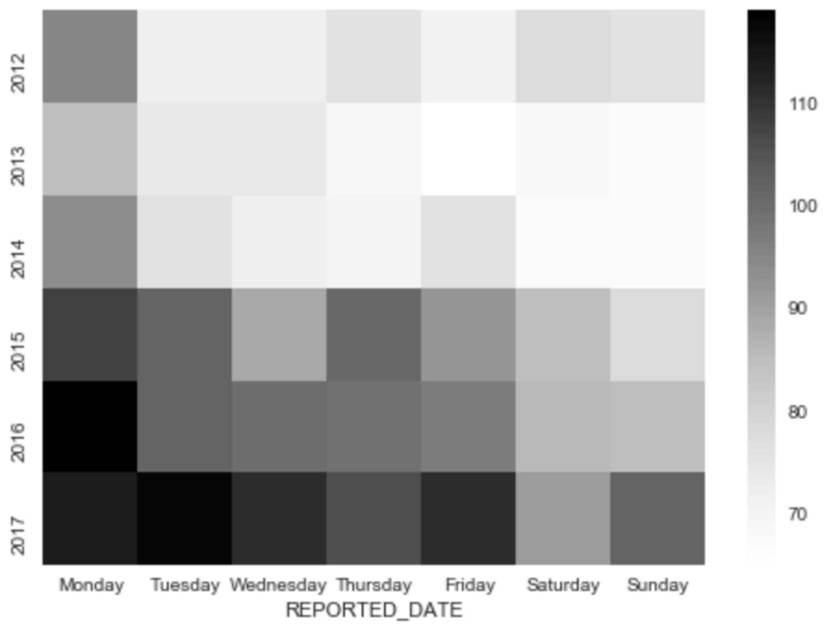
- Pandas official documentation on *Resampling* (http://bit.ly/2yHXrbz)
- Table of all *Anchored Offsets* (http://bit.ly/2xg20h2)

Measuring crime by weekday and year

Let's finalize this analysis by writing a function to complete all the steps of this recipe at once and add the ability to choose a specific type of crime:

```
>>> ADJ_2017 = .748
>>> def count_crime(df, offense_cat):
df = df[df['OFFENSE_CATEGORY_ID'] == offense_cat]
weekday = df['REPORTED_DATE'].dt.weekday_name
year = df['REPORTED_DATE'].dt.year
ct = df.groupby([year, weekday]).size().unstack()
ct.loc[2017] = ct.loc[2017].div(ADJ_2017).astype('int')
pop = pd.read_csv('data/denver_pop.csv', index_col='Year')
pop = pop.squeeze().div(100000)
ct = ct.div(pop, axis=0).astype('int')
ct = ct.reindex(columns=days)
sns.heatmap(ct, cmap='Greys')
return ct
>>> count_crime(crime, 'auto-theft')
```

REPORTED_DATE	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
2012	95	72	72	76	71	78	76
2013	85	74	74	69	65	68	67
2014	94	76	72	70	76	67	67
2015	108	102	89	101	92	85	78
2016	119	102	100	99	97	86	85
2017	114	118	111	106	111	91	102



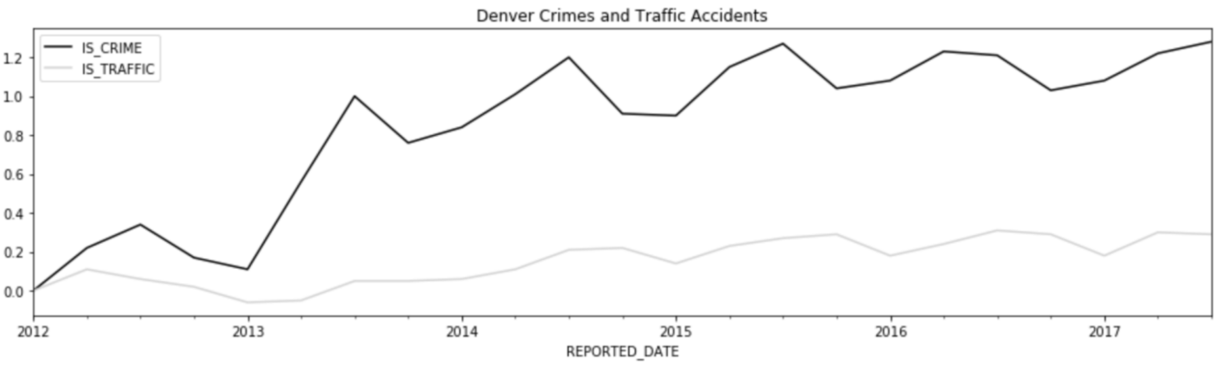
See also:

- Pandas official documentation of the reindex method (http://bit.ly/2y40eyE)
- The seaborn official documentation of the heatmap function (http://bit.ly/2ytbMNe)

Aggregating weekly crime and traffic accidents separately

To get a different visual perspective, we can plot the percentage increase in crime and traffic, instead of the raw count. Let's divide all the data by the first row and plot again:

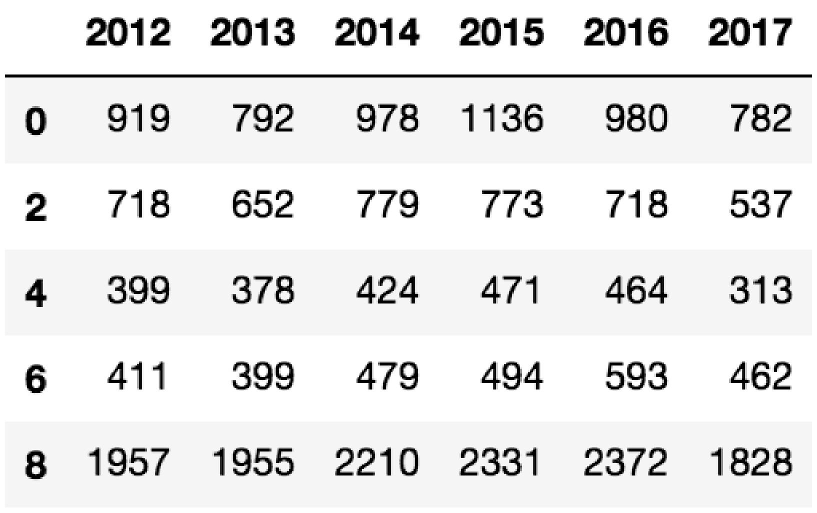
```
>>> crime_begin = crime_quarterly.iloc[0]
>>> crime_begin
IS_CRIME 7882
IS_TRAFFIC 4726
Name: 2012-03-31 00:00:00, dtype: int64
>>> crime_quarterly.div(crime_begin) \
.sub(1) \
.round(2) \
.plot(**plot_kwargs)
```



Grouping with anonymous functions with a DatetimeIndex

The final result of this recipe is a DataFrame with MultiIndex columns. Using this DataFrame, it is possible to select just the crime or traffic accidents separately. The xs method allows you to select a single value from any index level. Let's see an example where we select only the section of data dealing with traffic:

```
>>> cr_final.xs('IS_TRAFFIC', axis='columns', level=0).head()
```



	2012	2013	2014	2015	2016	2017
0	919	792	978	1136	980	782
2	718	652	779	773	718	537
4	399	378	424	471	464	313
6	411	399	479	494	593	462
8	1957	1955	2210	2331	2372	1828

This is referred to as taking a cross section in pandas. We must use the axis and level parameters to specifically denote where our value is located. Let's use xs again to select only data from 2016, which is in a different level:

```
>>> cr_final.xs(2016, axis='columns', level=1).head()
```

	IS_CRIME	IS_TRAFFIC
0	5377	980
2	4091	718
4	3044	464
6	2108	593
8	4488	2372

See also:

- Pandas official documentation of the cross section method xs (http://bit.ly/2xkLzLv)

Grouping by a Timestamp and another column

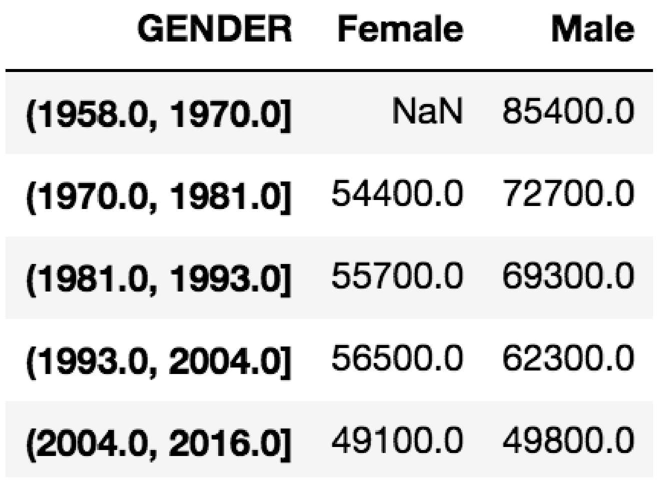
From an outsider's perspective, it would not be obvious that the rows from the output in step 8 represented 10-year intervals. One way to improve the index labels would be to show the beginning and end of each time interval. We can achieve this by concatenating the current index year with 9 added to itself:

```
>>> years = sal_final.index.year
>>> years_right = years + 9
>>> sal_final.index = years.astype(str) + '-' + years_right.astype(str)
>>> sal_final
```

GENDER	Female	Male
1958-1967	NaN	81200.0
1968-1977	NaN	106500.0
1978-1987	57100.0	72300.0
1988-1997	57100.0	64600.0
1998-2007	54700.0	59700.0
2008-2017	47300.0	47200.0

There is actually a completely different way to do this recipe. We can use the cut function to create equal-width intervals based on the year that each employee was hired and form groups from it:

```
>>> cuts = pd.cut(employee.index.year, bins=5, precision=0)
>>> cuts.categories.values
array([Interval(1958.0, 1970.0, closed='right'),
Interval(1970.0, 1981.0, closed='right'),
Interval(1981.0, 1993.0, closed='right'),
Interval(1993.0, 2004.0, closed='right'),
Interval(2004.0, 2016.0, closed='right')], dtype=object)
>>> employee.groupby([cuts, 'GENDER'])['BASE_SALARY'] \
.mean().unstack('GENDER').round(-2)
```



GENDER	Female	Male
(1958.0, 1970.0]	NaN	85400.0
(1970.0, 1981.0]	54400.0	72700.0
(1981.0, 1993.0]	55700.0	69300.0
(1993.0, 2004.0]	56500.0	62300.0
(2004.0, 2016.0]	49100.0	49800.0

Finding the Last Time Crime was 20% Lower with merge_asof

In addition to the Timestamp and Timedelta data types, pandas offers the Period type to represent an exact time period. For example, *2012-05* would represent the entire month of May, 2012. You can manually construct a Period in the following manner:

```
>>> pd.Period(year=2012, month=5, day=17, hour=14, minute=20, freq='T')
Period('2012-05-17 14:20', 'T')
```

This object represents the entire minute of May 17, 2012 at 2:20 p.m. It is possible to use these Periods in step 4 instead of grouping by date with pd.Grouper. DataFrames with a DatetimeIndex have the to_period method to convert Timestamps to Periods. It accepts an offset alias to determine the exact length of the time period.

```
>>> ad_period = crime_sort.groupby([lambda x: x.to_period('M'),
'OFFENSE_CATEGORY_ID']).size()
>>> ad_period = ad_period.sort_values() \
.reset_index(name='Total') \
.rename(columns={'level_0':'REPORTED_DATE'})
>>> ad_period.head()
```

	REPORTED_DATE	OFFENSE_CATEGORY_ID	Total
0	2014-12	murder	1
1	2013-01	arson	1
2	2016-05	murder	1
3	2012-12	murder	1
4	2016-12	murder	1

Let's verify that the last two columns from this DataFrame are equivalent to all_data from step 5:

```
>>> cols = ['OFFENSE_CATEGORY_ID', 'Total']
>>> all_data[cols].equals(ad_period[cols])
True
```

Steps 6 and 7 can now be replicated in almost the exact same manner with the following code:

```
>>> aug_2018 = pd.Period('2017-8', freq='M')
>>> goal_period = ad_period[ad_period['REPORTED_DATE'] == aug_2018] \
.reset_index(drop=True)
>>> goal_period['Total_Goal'] = goal_period['Total'].mul(.8).astype(int)
>>> pd.merge_asof(goal_period, ad_period, left_on='Total_Goal',
right_on='Total', by='OFFENSE_CATEGORY_ID',
suffixes=('_Current', '_Last')).head()
```

	REPORTED_DATE_Current	OFFENSE_CATEGORY_ID	Total_Current	Total_Goal	REPORTED_DATE_Last	Total_Last
0	2017-08	murder	7	5	2017-01	5
1	2017-08	arson	7	5	2012-01	5
2	2017-08	sexual-assault	57	45	2013-01	45
3	2017-08	robbery	108	86	2015-03	86
4	2017-08	white-collar-crime	138	110	2016-10	110