Data Exploration

Hugo Dolan Follow
Jul 22, 2018 · 10 min read

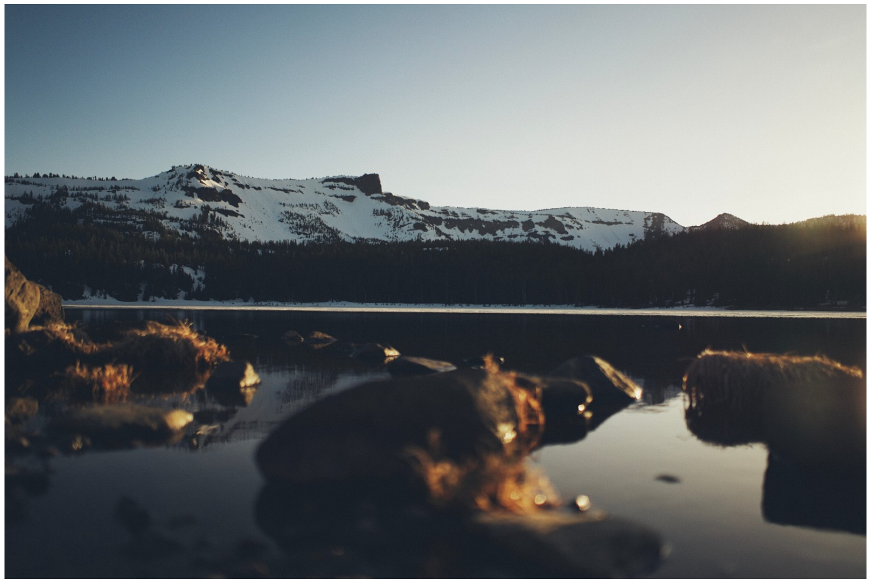


Photo by Clint McKoy on Unsplash

After recently using Pandas and Matplotlib to produce the graphs / analysis for this article on China's property bubble , and creating a random forrest regression model to find undervalued used cars (more on this soon). I decided to put together this practical guide, which should hopefully be enough to get you up and running with your own data exploration using Pandas and MPL!

> *This article is broken up into the following Sections:*

The Basic Requirements

- Reading Data From CSV
- Formatting, cleaning and filtering Data Frames
- Group-by and Merge

Visualising Your Data

- The Plot Function Basics
- Seaborn violin and lm-plots
- Pair plots and Heat maps

Figure Aesthetics

- Plotting with multiple axis
- Making your charts look less scientific

The Basic Requirements

Reading CSV / Required Imports for Matplotlib & Pandas

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
%matplotlib inline
```

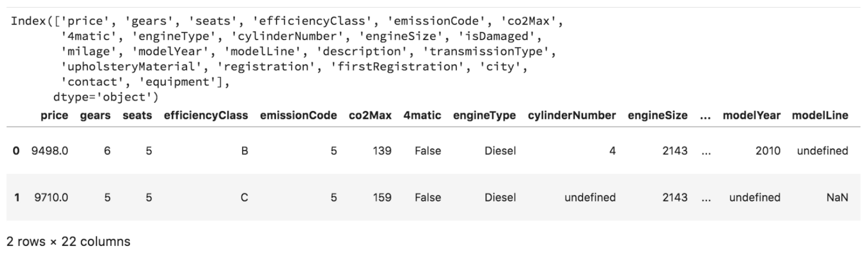
```
car_data = pd.read_csv('inbox/CarData-E-class-Tue Jul 03
2018.csv')
```

Inline indicates to present graphs as cell output, *read_csv* returns a DataFrame, the *file path* is relative to that of your notebook.

Formatting, Cleaning and Filtering DataFrames

Often when dealing with a large number of features it is nice to see the first row, or the names of all the columns, using the *columns* property and *head(nRows)* function. However if we are interested in the types of values for a categorical such as the modelLine, we can access the column using the square bracket syntax and use *.unique()* to inspect the options.

```
print(car_data.columns)
car_data.head(2)
```

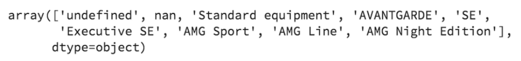


```
Index(['price', 'gears', 'seats', 'efficiencyClass', 'emissionCode', 'co2Max',
       '4matic', 'engineType', 'cylinderNumber', 'engineSize', 'isDamaged',
       'milage', 'modelYear', 'modelLine', 'description', 'transmissionType',
       'upholsteryMaterial', 'registration', 'firstRegistration', 'city',
       'contact', 'equipment'],
      dtype='object')
```

	price	gears	seats	efficiencyClass	emissionCode	co2Max	4matic	engineType	cylinderNumber	engineSize	...	modelYear	modelLine
0	9498.0	6	5	B	5	139	False	Diesel	4	2143	...	2010	undefined
1	9710.0	5	5	C	5	159	False	Diesel	undefined	2143	...	undefined	NaN

2 rows × 22 columns

```
car_data['modelLine'].unique()
```



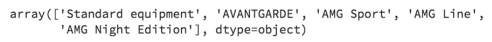
```
array(['undefined', nan, 'Standard equipment', 'AVANTGARDE', 'SE',
       'Executive SE', 'AMG Sport', 'AMG Line', 'AMG Night Edition'],
      dtype=object)
```

There are clearly multiple versions of the same model line entered under different variations of ‘Special Equipment’ so we will use a *regex* to replace anything containing SE with Special equipment. Similarly there are some columns with *Nans (Not a Number)* so we will just drop these with *dropna(subset=[‘modelLine’])*.

```
car_data = car_data.dropna(subset=['modelLine'])
car_data['modelLine'] =
car_data['modelLine'].replace(to_replace={'.*SE.*':
'Standard equipment'}, regex=True)
```

We can also filter out unwanted values such as ‘undefined’ by comparing the rows of modelLine against some boolean question, this returns a boolean array of the same dimensions as the DataFrame rows which can be used to filter with the square bracket syntax again.

```
car_data = car_data[(car_data['modelLine'] != 'undefined')]

car_data['modelLine'].unique()
```



```
array(['Standard equipment', 'AVANTGARDE', 'AMG Sport', 'AMG Line',
       'AMG Night Edition'], dtype=object)
```

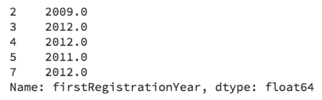
This is looking much better!

> Note above how pandas never mutates any existing data, hence we have to overwrite our old data manually when we perform any mutations / filters. Whilst this may seem redundant, its extremely effective method of reducing unwanted side effects and bugs in your code.

Moving on, we also need to change the firstRegistration field typically this should be treated as a python date format, but instead we will treat it as a numeric field for convenience in performing regressions on the data in a future article.

Considering this data is associated with car registration, the year is really the important component we need to keep. Thus treating this as a numeric field means we can apply numerical rounding, multiplication / division to create a Registration Year feature column as below.

```
car_data['firstRegistration'].head(5)


car_data[‘firstRegistrationYear’] =
round((car_data[‘firstRegistration’] / 10000),0)
car_data[‘firstRegistrationYear’] .head(5)
```



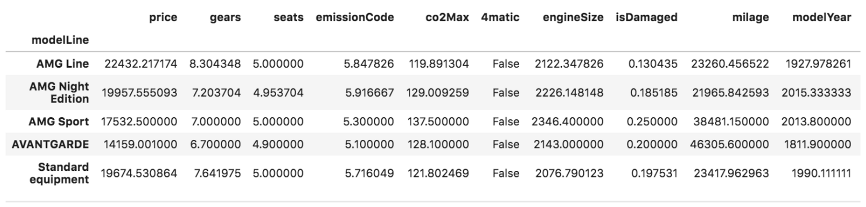
```
2    2009.0
3    2012.0
4    2012.0
5    2011.0
7    2012.0
Name: firstRegistrationYear, dtype: float64
```

Looks like the output we were looking for.

Using Group-by’s and Merges

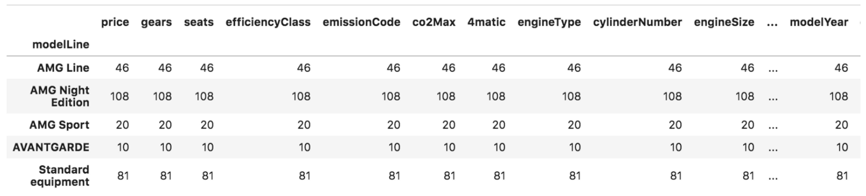
Group-by’s can be used to build groups of rows based off a specific feature in your dataset eg. the ‘modelLine’ categorical column. We can then perform an operation such as mean, min, max, std on the individual groups to help describe the sample data.

```
group_by_modelLine = car_data.groupby(by=['modelLine'])
car_data_avg = group_by_modelLine.mean()
car_data_count = group_by_modelLine.count()
```



modelLine	price	gears	seats	emissionCode	co2Max	4matic	engineSize	isDamaged	milage	modelYear
AMG Line	22432.217174	8.304348	5.000000	5.847826	119.891304	False	2122.347826	0.130435	23260.456522	1927.978261
AMG Night Edition	19957.555093	7.203704	4.953704	5.916667	129.009259	False	2226.148148	0.185185	21965.842593	2015.333333
AMG Sport	17532.500000	7.000000	5.000000	5.300000	137.500000	False	2346.400000	0.250000	38481.150000	2013.800000
AVANTGARDE	14159.001000	6.700000	4.900000	5.100000	128.100000	False	2143.000000	0.200000	46305.600000	1811.900000
Standard equipment	19674.530864	7.641975	5.000000	5.716049	121.802469	False	2076.790123	0.197531	23417.962963	1990.111111

Averages Data



modelLine	price	gears	seats	efficiencyClass	emissionCode	co2Max	4matic	engineType	cylinderNumber	engineSize	...	modelYear
AMG Line	46	46	46	46	46	46	46	46	46	46	...	46
AMG Night Edition	108	108	108	108	108	108	108	108	108	108	...	108
AMG Sport	20	20	20	20	20	20	20	20	20	20	...	20
AVANTGARDE	10	10	10	10	10	10	10	10	10	10	...	10
Standard equipment	81	81	81	81	81	81	81	81	81	81	...	81

Count Data: Note that this is simply a count of the records for each model Line

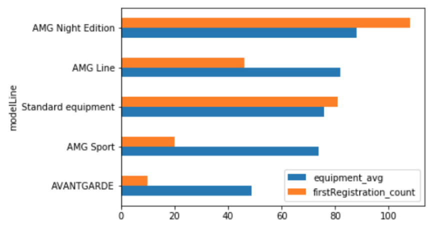
As you can see the mean value for each numeric feature has been calculated for each model Line. Group by's are highly versatile and also accept lambda functions for more complex row / group labelling.

Next we will assemble a DataFrame of only the relevant features to plot a graph of availability (or car count) and average equipment per car. This DataFrame can be created by passing in a dictionary of keys which represent the columns and values which are single columns or Series from our existing data. This works here because both Data Frames have the same number of rows. Alternatively we can merge the two Data Frames by their indexes (modelLine) and rename the suffixes of repeated columns appropriately.

We will then plot these two variables sorting by equipment then availability as a horizontal bar graph.

```
# Since all the columns in car_data_count are the same, we
will use just the first column as the rest yield the same
result. iloc allows us to take all the rows and the zeroth
column.

car_data_count_series = car_data_count.iloc[:,0]

features_of_interest = pd.DataFrame({'equipment':
car_data_avg['equipment'], 'availability':
car_data_count_series})

alternative_method = car_data_avg.merge(car_data_count,
left_index=True, right_index=True, suffixes=
['_avg','_count'])

alternative_method[['equipment_avg',
'firstRegistration_count']].sort_values(by=['equipment_avg',
'firstRegistration_count'],
ascending=True).plot(kind='barh')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x12165f4a8>
```



Visualising Your Data

The Pandas Plot Function

Pandas has a built in .plot() function as part of the DataFrame class. It has several key parameters:

kind—'bar','barh','pie','scatter','kde' etc which can be found in the docs.
color—Which accepts and array of hex codes corresponding sequential to each data series / column.
linestyle—'solid', 'dotted', 'dashed' (applies to line graphs only)
xlim, ylim—specify a tuple (lower limit, upper limit) for which the plot will be drawn
legend— a boolean value to display or hide the legend
labels—a list corresponding to the number of columns in the dataframe, a descriptive name can be provided here for the legend
title—The string title of the plot

These are fairly straightforward to use and we'll do some examples using .plot() later in the post.

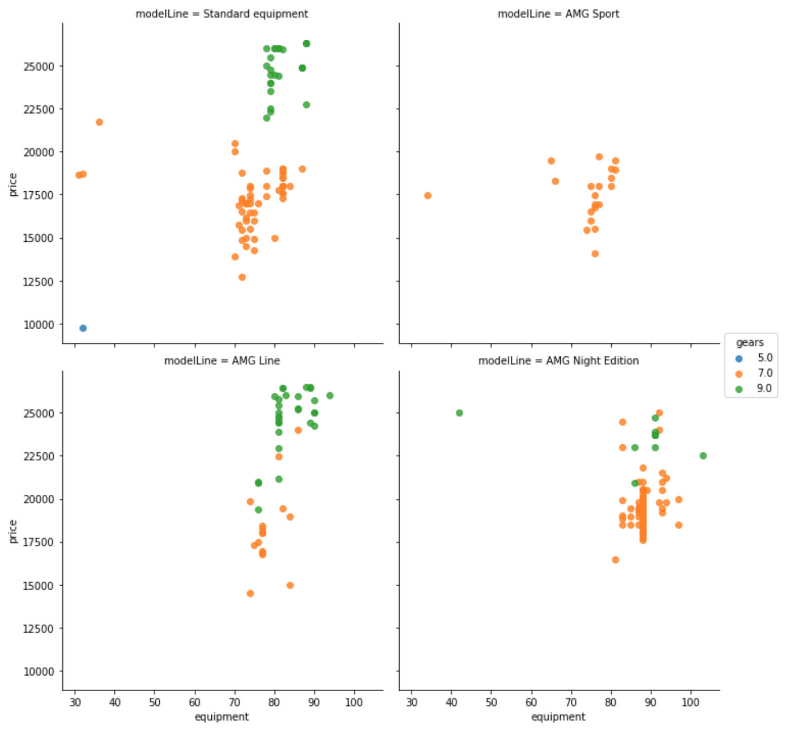
Seaborn lmplots

Seaborn builds on top of matplotlib to provide a richer out of the box environment. It includes a neat lmplot plot function for rapid exploration of multiple variables. Using our car data example, we would like to understand the association between the equipment kit-out of a car and the sale price. Obviously we would also like this data segmented by model line to compare like with like.

```
import seaborn as sns
```

Passing in our column labels for equipment and price (x and y axis) followed by the actual DataFrame source. Use the col keyword to generate a separate plot for each model line and set the col_wrap 2 to make a nice grid.

```
filtered_class = car_data[car_data['modelLine'] !=
'AVANTGARDE']

sns.lmplot("equipment", "price", data=filtered_class,
hue="gears", fit_reg=False, col='modelLine', col_wrap=2)
```

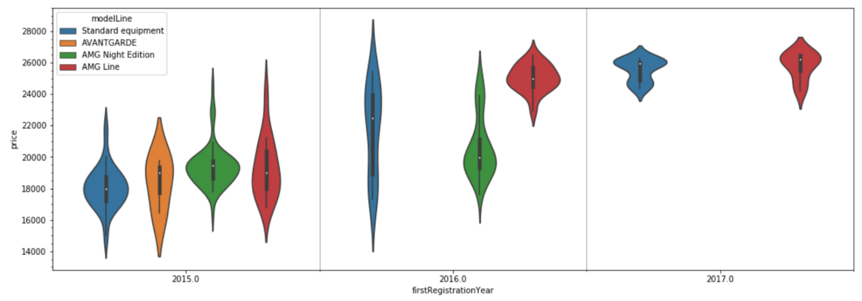


As you can see putting a hue onto the chart for the number of gears was particularly informative, as these types of car tend to be no better equipped but more expensive. As you can see we could perform significant exploration of our dataset in 3 lines of code.

Seaborn Violin Plots

These plots are excellent for dealing with large continuous datasets, and can similarly be segmented by an index. Using our car dataset we can gain a greater understanding about the price distribution of used cars. Since the age of a car dramatically affects the price we will plot the first regsitration year as our x axis variable and price as our y. We can then set our hue to sepearate out the various model variants.

```
from matplotlib.ticker import AutoMinorLocator


fig = plt.figure(figsize=(18,6))

LOOKBACK_YEARS = 3
REGISTRATION_YEAR = 2017


filtered_years = car_data[car_data['firstRegistrationYear']
> REGISTRATION_YEAR - LOOKBACK_YEARS]


ax1 = sns.violinplot('firstRegistrationYear', "price",
data=filtered_years, hue='modelLine')


ax1.minorticks_on()
ax1.xaxis.set_minor_locator(AutoMinorLocator(2))
ax1.grid(which='minor', axis='x', linewidth=1)
```

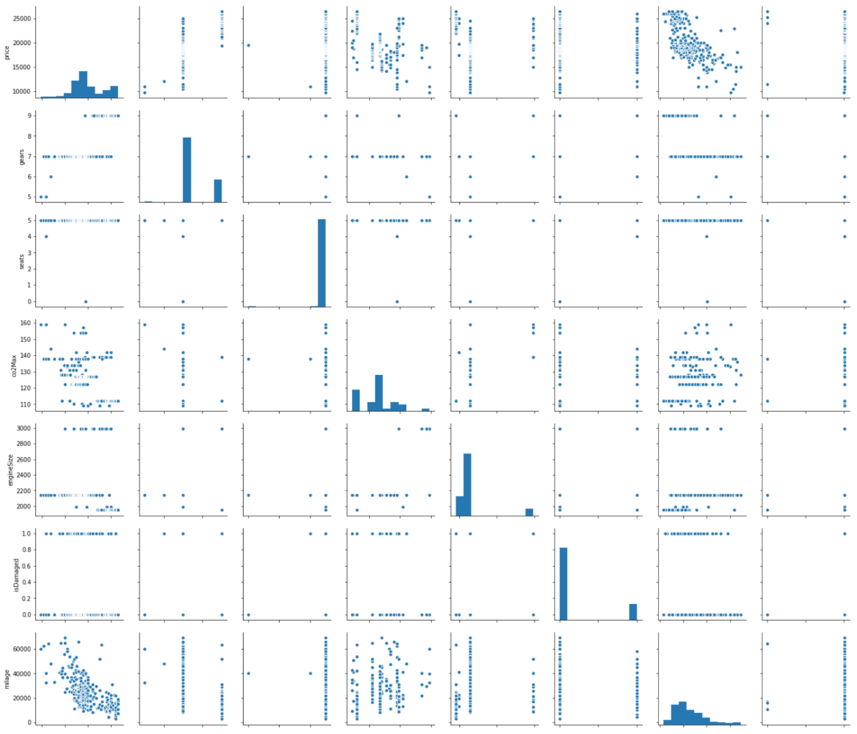


Notice that the violin plot function returns the axis on which the plot is displayed. This allows us to edit property of the axis. In this case we have set minor ticks on and used the AutoMinorLocator to place 1 minor tick between each major interval. I then made the minor grid visible with line width of 1. This was neat hack to put a box around each registration year.

Pairplots & Correlation Heatmaps

In datasets with a small number of features (10–15) Seaborn Pairplots can quickly enable a visual inspection of any relationships between variables. Graphs along the left diagonal represent the distribution of each feature, whilst graphs on off diagonals show the relationship between variables.

```
sns.pairplot(car_data.loc[:,car_data.dtypes == 'float64'])
```

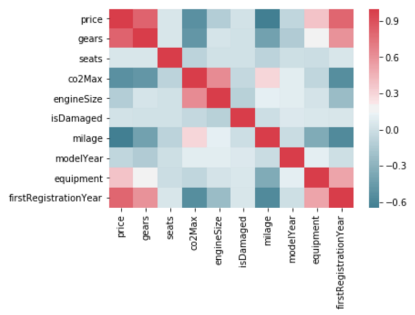


(This is only a section, I couldn't fit all the variables in, but you get the concept.)

Similarly we can utilise the pandas Corr() to find the correlation between each variable in the matrix and plot this using Seaborn's Heatmap function, specifying the labels and the Heatmap colour range.

```
corr = car_data.loc[:,car_data.dtypes == 'float64'].corr()

sns.heatmap(corr, xticklabels=corr.columns,
yticklabels=corr.columns, cmap=sns.diverging_palette(220,
10, as_cmap=True))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1a2c1b15f8>
```

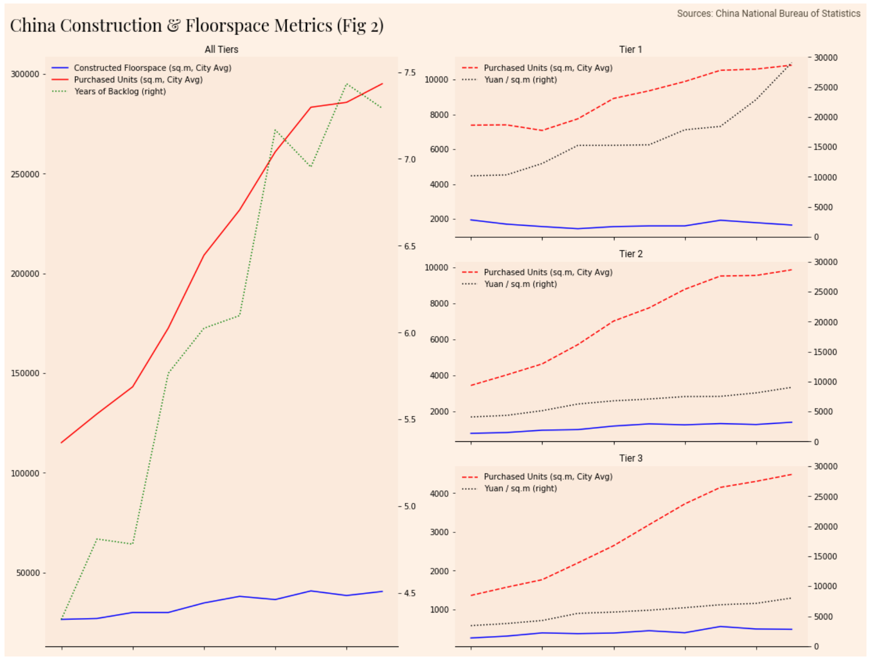


These two tools combined can be quite useful for identifying important features to a model quickly. Using the Heatmap for example we can see from the top row, that the number of gears and the first registration are positively correlated with price, where as milage is likely to be negatively correlated. Its by far a perfect tool for analysis, but useful at a basic level.

Figure Aesthetics

Plotting With Multiple Axis

Below is some data from my previous post on China's Property Bubble. I wanted to show construction data for all cities and then provide a subsequent breakdown by city tier in a single figure.



Lets breakdown how we might create such a figure:

First we define the size of the figure to provide adequate graphing space. When plotting with multiple axis we define a grid on which axis may be place on. We then use the subplot2grid function to return an axis at the desired location (specified from top left corner) with the correct span of rows / columns.

```
fig = plt.figure(figsize = (15,12))
grid_size = (3,2)
hosts_to_fmt = []

# Place A Title On The Figure
```

```
fig.text(x=0.8, y=0.95, s='Sources: China National Bureau of
Statistics',fontproperties=subtitle_font,
horizontalalignment='left',color='#524939')

# Overlay multiple plots onto the same axis, which spans 1
entire column of the figure

large_left_ax = plt.subplot2grid(grid_size, (0,0),
colspan=1, rowspan=3)
```

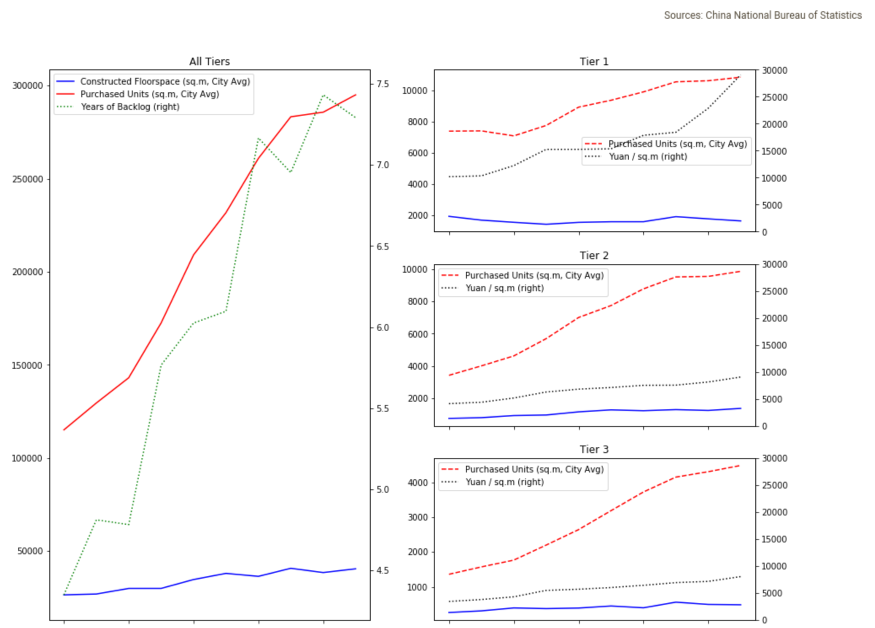
We can then subsequently plot onto this axis by specifying the ax property of the plot function. Note that the despite plotting onto a specific axis, the use of the secondary_y parameter means a new axis instance will be created. This will be important to store for formatting later.

```
# Aggregating to series into single data frame for ease of
plotting

construction_statistics = pd.DataFrame({
    'Constructed Floorspace (sq.m, City Avg)':
     china_constructed_units_total,
    'Purchased Units (sq.m, City Avg)':
     china_under_construction_units_total,
})

construction_statistics.plot(ax=large_left_ax,
    legend=True, color=['b', 'r'], title='All Tiers')

# Second graph overlayed on the secondary y axis

large_left_ax_secondary =
china_years_to_construct_existing_pipeline.plot(
    ax=large_left_ax, label='Years of Backlog',
linestyle='dotted',
    legend=True, secondary_y=True, color='g')


# Adds the axis for formatting later


hosts_to_fmt.extend([large_left_ax,
large_left_ax_secondary])
```

To produce the breakdowns by city tier, we again utilise the subplot2grid but this time change the index on every loop, such that the 3 tier charts plot one below the other.

```
# For each City Tier overlay a series of graphs on an axis
on the right hand column
# Its row position determined by its index


for index, tier in enumerate(draw_tiers[0:3]):
    tier_axis = plt.subplot2grid(grid_size, (index,1))

    china_constructed_units_tiered[tier].plot(ax=tier_axis,
     title=tier, color='b', legend=False)

    ax1 = china_under_construction_units_tiered[tier].plot(
     ax=tier_axis,linestyle='dashed', label='Purchased Units
     (sq.m,City Avg)', title=tier, legend=True, color='r')

    ax2
=china_property_price_sqmetre_cities_tiered[tier].plot(
     ax=tier_axis, linestyle='dotted', label='Yuan / sq.m',
     secondary_y=True, legend=True, color='black')

    ax2.set_ylim(0,30000)



hosts_to_fmt.extend([ax1,ax2])
```

Ok so now we have generated the correct layout and plotted data:



Make Your Charts Look Less Scientific

In the case of the above chart, I went for a styling similar to the ft.com. First up we need to import our fonts via Matplotlib font manager, and create a font properties objects for each respective category.

```
import matplotlib.font_manager as fm

# Font Imports
```

```
heading_font =
fm.FontProperties(fname='/Users/hugo/Desktop/Playfair_Displa
y/PlayfairDisplay-Regular.ttf', size=22)

subtitle_font = fm.FontProperties(
fname='/Users/hugo/Library/Fonts/Roboto-Regular.ttf',
size=12)

# Color Themes


color_bg = '#FEF1E5'
lighter_highlight = '#FAE6E1'
darker_highlight = '#FBEADC'
```

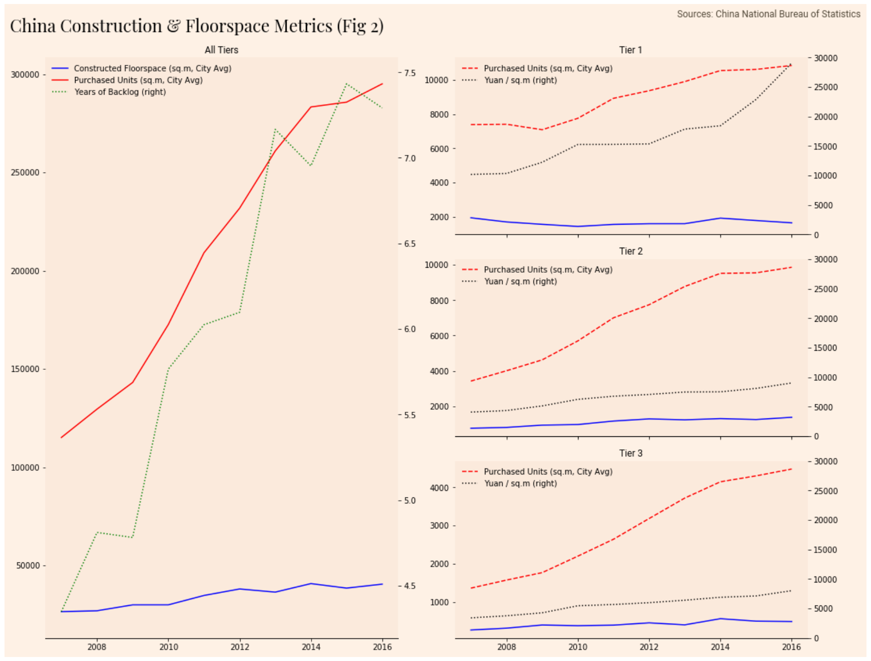
Next we will define a function which will:

- Set the figure background (using set_facecolor)
- Apply a title to the figure using the specified title font.
- Call the *tight layout* function which utilises the plot space more compactly.

Next we will iterate over each axes within the figure and call a function to:

- Disable all except the bottom spines (axes borders)
- Set the background colour of the axis to be slightly darker.
- Disable the white box around the legend if a legend exists.
- Set the title of each axis to use the subtitle font.

Finally we just need to call the formatter function we created and pass in our figure and the axes we collected earlier.



Conclusion

Thanks for reading this tutorial, hopefully this helps get you up and running with Pandas and Matplotlib.