ST2195 Programming for Data Science
Graphics and Data Visualisation in Python

** Note: The code chunks below should be run in the following order **

Plotting in Python

In Python, **matplotlib** and **seaborn** are two main Python libraries for plots. **matplotlib** is a basic plotting library. **seaborn** is a Python data visualization library based on matplotlib, and it provides some "prettier" and sometimes more interesting plots. We use both **matplotlib** and **seaborn** in this notebook, although we mainly use **matplotlib** here. We import the libraries as follows:

```
import matplotlib.pyplot as plt
import seaborn as sns


# Set the colour palette to colour-blind friendly
sns.reset_orig()
my_palette = sns.color_palette("colorblind")
plt.style.use('seaborn-colorblind')
```

We will first start with the basics of **matplotlib**, which includes how to create a space for the plot, and how to make a basic plot. We will then talk about how to create different plots including:

- Line plot
- Bar plot
- Boxplot
- Scatter plot
- Heat map

using **matplotlib** and **seaborn**. Before we introduce how to use the plotting libraries, let us first load the data for visualisation.

Import and Clean the Data

We use GDP per capita data from `DBnomics` and wine chemistry composition data from `scikit-learn` in this notebook. The data are already downloaded and stored in three csv files: `gdp.csv`, `gdp_wide` and `wine.csv` and we load the data to Python with the following code:

```
import numpy as np
import pandas as pd
import json
from datetime import datetime

gdp = pd.read_csv("gdp.csv")
# convert the year column from string to datatime object
gdp['year'] = gdp['year'].apply(lambda x : datetime.strptime(x, '%Y-%M-%d'))

# get the GDP per capita data for each country
# reset_index so that each Series has indexes 0,1,2,3...
gdp_uk = gdp[gdp['country']=='UK'].reset_index()
gdp_fr = gdp[gdp['country']=='FR'].reset_index()
gdp_it = gdp[gdp['country']=='IT'].reset_index()
gdp_de = gdp[gdp['country']=='DE'].reset_index()

gdp_wide = pd.read_csv("gdp_wide.csv")
# set the row labels as countries
gdp_wide.set_index('country', inplace= True)
# set the column labels (years) with the type int
gdp_wide.columns = gdp_wide.columns.astype(int)

wine = pd.read_csv("wine.csv")
```

`gdp_uk`, `gdp_fr`, `gdp_it`, `gdp_de` store the GDP per capita time series for a corresponding country. For example

```
print(gdp_uk)
    index                year    value country
0       0 2000-01-01 00:01:00  27130.0      UK
1       1 2001-01-01 00:01:00  27770.0      UK
2       2 2002-01-01 00:01:00  28250.0      UK
3       3 2003-01-01 00:01:00  29060.0      UK
4       4 2004-01-01 00:01:00  29560.0      UK
5       5 2005-01-01 00:01:00  30210.0      UK
6       6 2006-01-01 00:01:00  30810.0      UK
```

```
7        7 2007-01-01 00:01:00  31280.0      UK
8        8 2008-01-01 00:01:00  30940.0      UK
9        9 2009-01-01 00:01:00  29460.0      UK
10      10 2010-01-01 00:01:00  29830.0      UK
11      11 2011-01-01 00:01:00  29960.0      UK
12      12 2012-01-01 00:01:00  30190.0      UK
13      13 2013-01-01 00:01:00  30660.0      UK
14      14 2014-01-01 00:01:00  31290.0      UK
15      15 2015-01-01 00:01:00  31780.0      UK
16      16 2016-01-01 00:01:00  32060.0      UK
17      17 2017-01-01 00:01:00  32430.0      UK
18      18 2018-01-01 00:01:00  32640.0      UK
19      19 2019-01-01 00:01:00  32870.0      UK
```

`gdp_wide` stores the four time series above in a table:

```
print(gdp_wide.head())
            2000     2001     2002     2003  ...     2016     2017     2018     201
9
country                                      ...
DE       28910.0  29370.0  29290.0  29100.0  ...  34610.0  35380.0  35720.0  35840.
0
FR       28930.0  29290.0  29410.0  29440.0  ...  31770.0  32380.0  32860.0  33270.
0
IT       27430.0  27950.0  27960.0  27850.0  ...  26020.0  26490.0  26780.0  26920.
0
UK       27130.0  27770.0  28250.0  29060.0  ...  32060.0  32430.0  32640.0  32870.
0

[4 rows x 20 columns]
```

`wine` stores the chemistry composition of wines in a table, with the last column provides the information of which class the wines belong to:

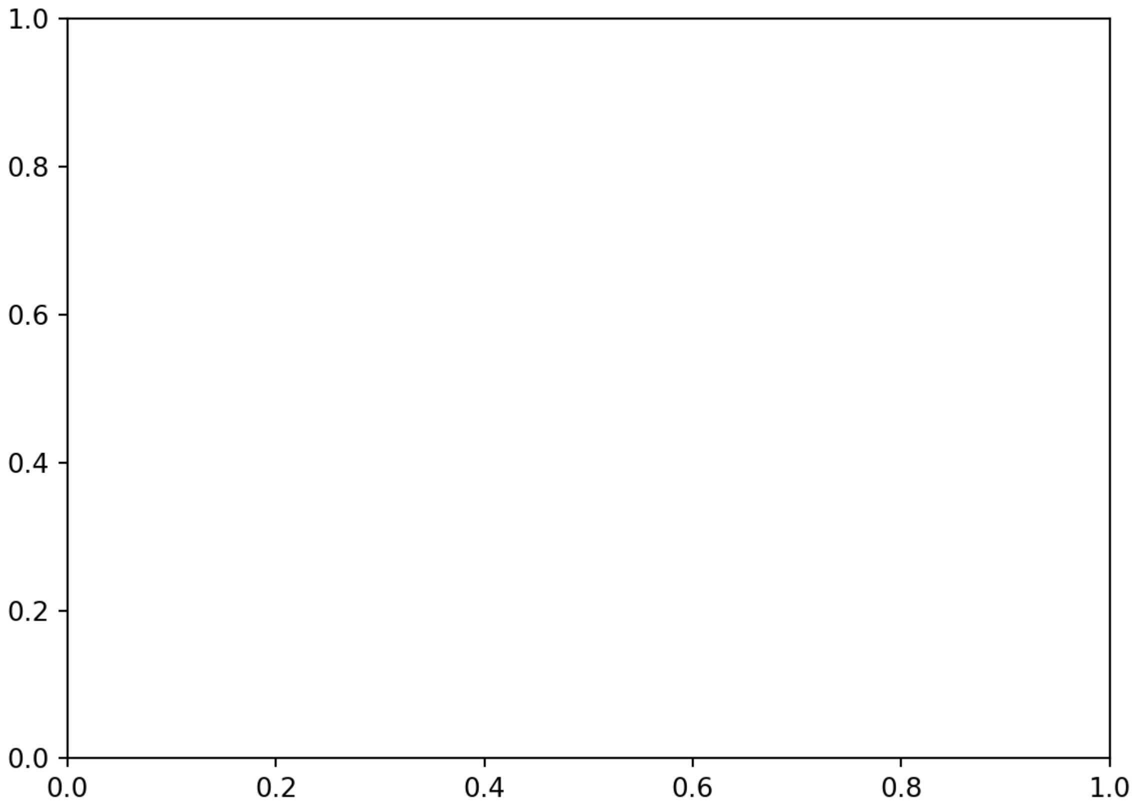
```
print(wine.head())
   alcohol  malic_acid   ash  ...  od280/od315_of_diluted_wines  proline   target
0    14.23        1.71  2.43  ...                          3.92   1065.0  class_0
1    13.20        1.78  2.14  ...                          3.40   1050.0  class_0
2    13.16        2.36  2.67  ...                          3.17   1185.0  class_0
3    14.37        1.95  2.50  ...                          3.45   1480.0  class_0
4    13.24        2.59  2.87  ...                          2.93    735.0  class_0

[5 rows x 14 columns]
```

matplotlib Basics

Creating an Empty Plot

We first learn how to create a empty figure. It can be done by:

```
fig, ax = plt.subplots()
plt.show()
```



this return a `Figure` and an `Axes` object. By manipulating these objects, we can plot and set the properties of the plot. `plt.show()` is used to make sure the plot is shown (although the plot may still be shown without calling `plt.show()`).

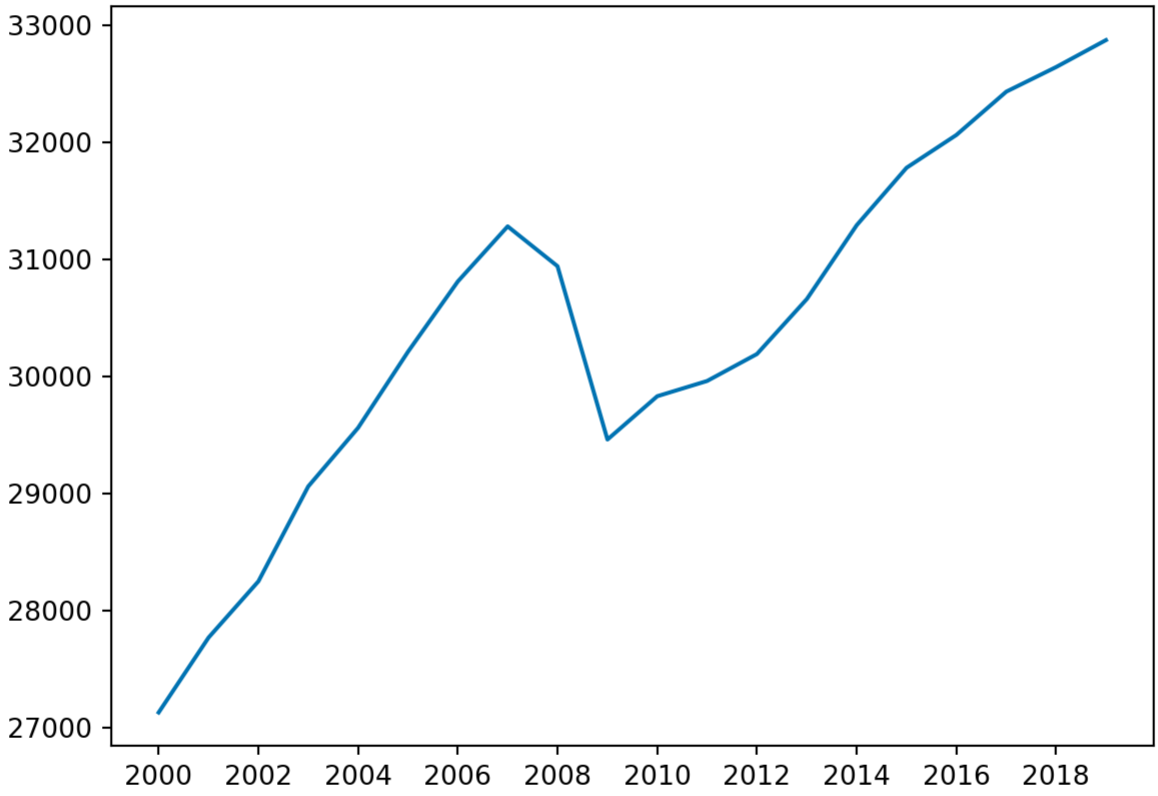
In this notebook we focus on manipulating the plots using `Axes` object `ax` or `plt` but not the `Figure` object `fig`. If you want to learn how you can use `Figure` object to manipulate the plots, please see here.

Plotting Using `ax`

After the empty plot is created, we can plot the data by using the `Axes` object `ax` and the method `plot()`. Here plot the GDP per capital time series data:

```
fig, ax = plt.subplots()
```

```
ax.plot(gdp_uk['year'], gdp_uk['value'])
plt.show()
```



Plotting Multiple Subplots

Similar to R, we can have multiple subplots in one figure. This can be done by specifying the number of rows/columns of the subplot grid in the `subplots()` function. It returns a figure and a *collection* of axes objects. We can plot the data on each of the subplots via the axes, and here we use different GDP per capita time series to illustrate it:

```
fig, ax = plt.subplots(2, 2, figsize = (15,10)) # ax is an array of an array (2x2)

ax[0][0].plot(gdp_uk['year'], gdp_uk['value']) # top left
ax[0][0].title.set_text('U.K')

ax[0][1].plot(gdp_fr['year'], gdp_fr['value']) # top right
ax[0][1].title.set_text('France')

ax[1][0].plot(gdp_de['year'], gdp_de['value'], label = 'Germany') # bottom left
```

```
ax[1][0].title.set_text('Germany')


ax[1][1].plot(gdp_it['year'], gdp_it['value'], label = 'Italy') # bottom right
ax[1][1].title.set_text('Italy')


plt.show()
```

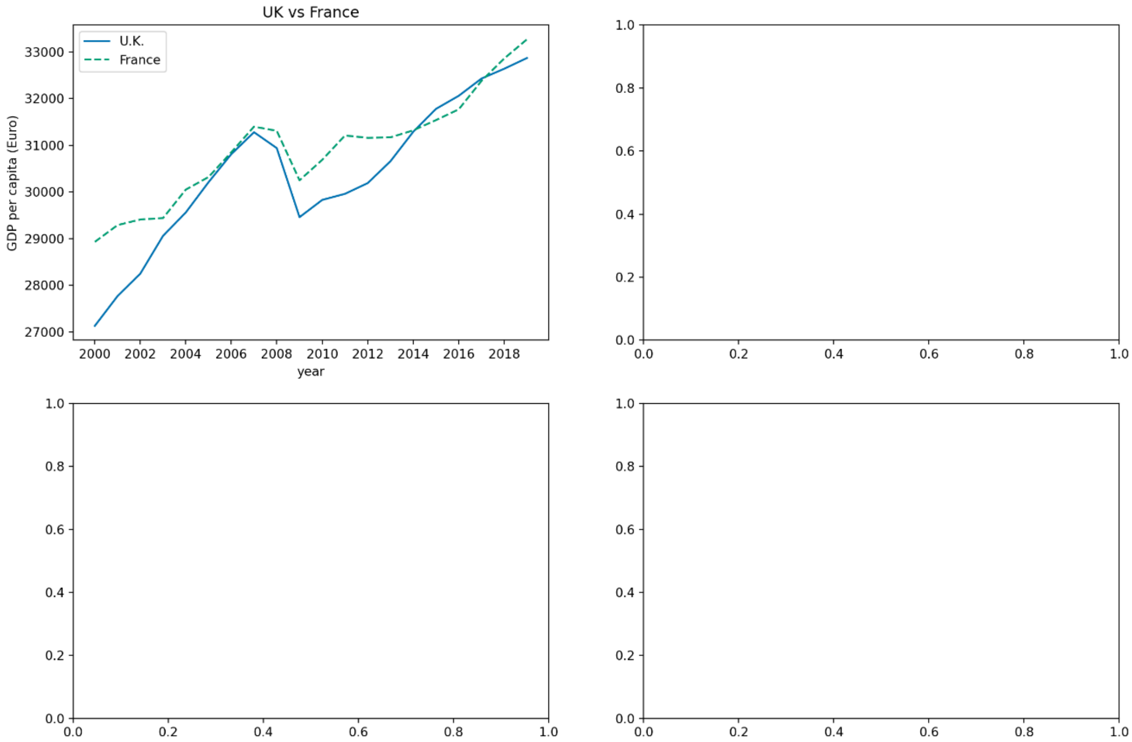
The argument `figsize` in `subplots()` determines the size of the figure.

Plotting Multiple Lines on the *Same* Plot

Similar to R, we can plot multiple lines on the same subplot. This can be done by calling `plot()` via the same axes (here `ax[0][0]`) with different data (as illustrated below). `ax[0][0].legend()` creates the legend for the lines for the top left subplot. We can add label, title, etc. to the subplot by manipulating via `ax`:

```
fig, ax = plt.subplots(2, 2, figsize = (15,10))


ax[0][0].plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.') # the label 'U.K' wi
ll be shown on the legend
ax[0][0].plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France') # '--' for d
ashed line
ax[0][0].legend()
```

```
ax[0][0].set_xlabel("year")
ax[0][0].set_ylabel("GDP per capita (Euro)")
ax[0][0].title.set_text('UK vs France')

plt.show()
```

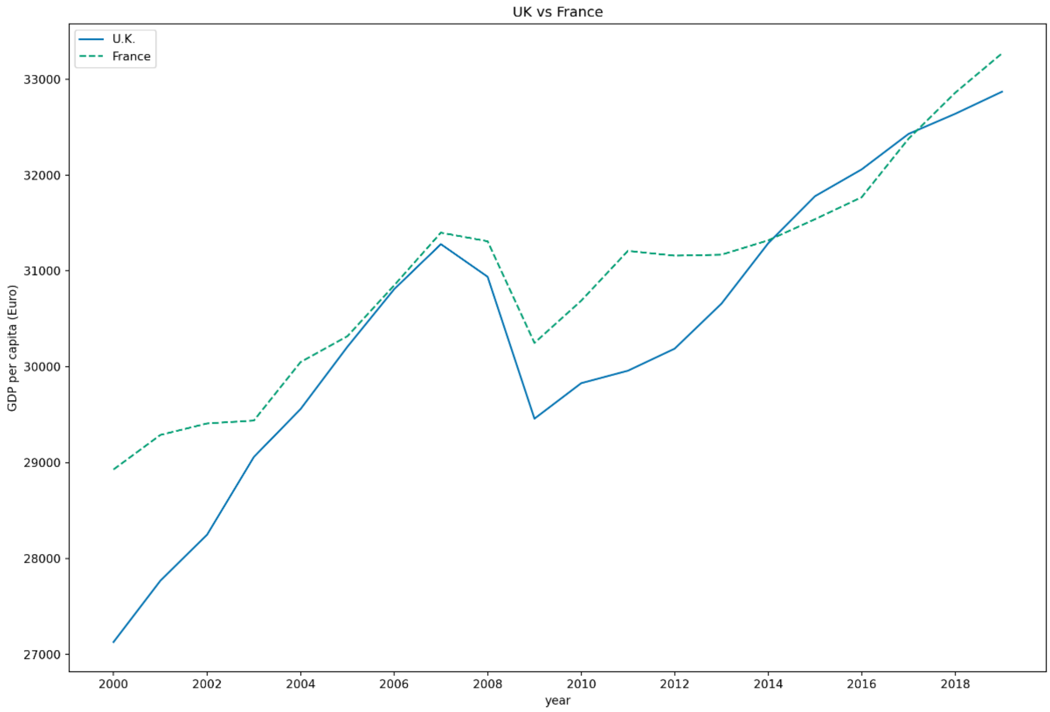


Line Plot With `plt`

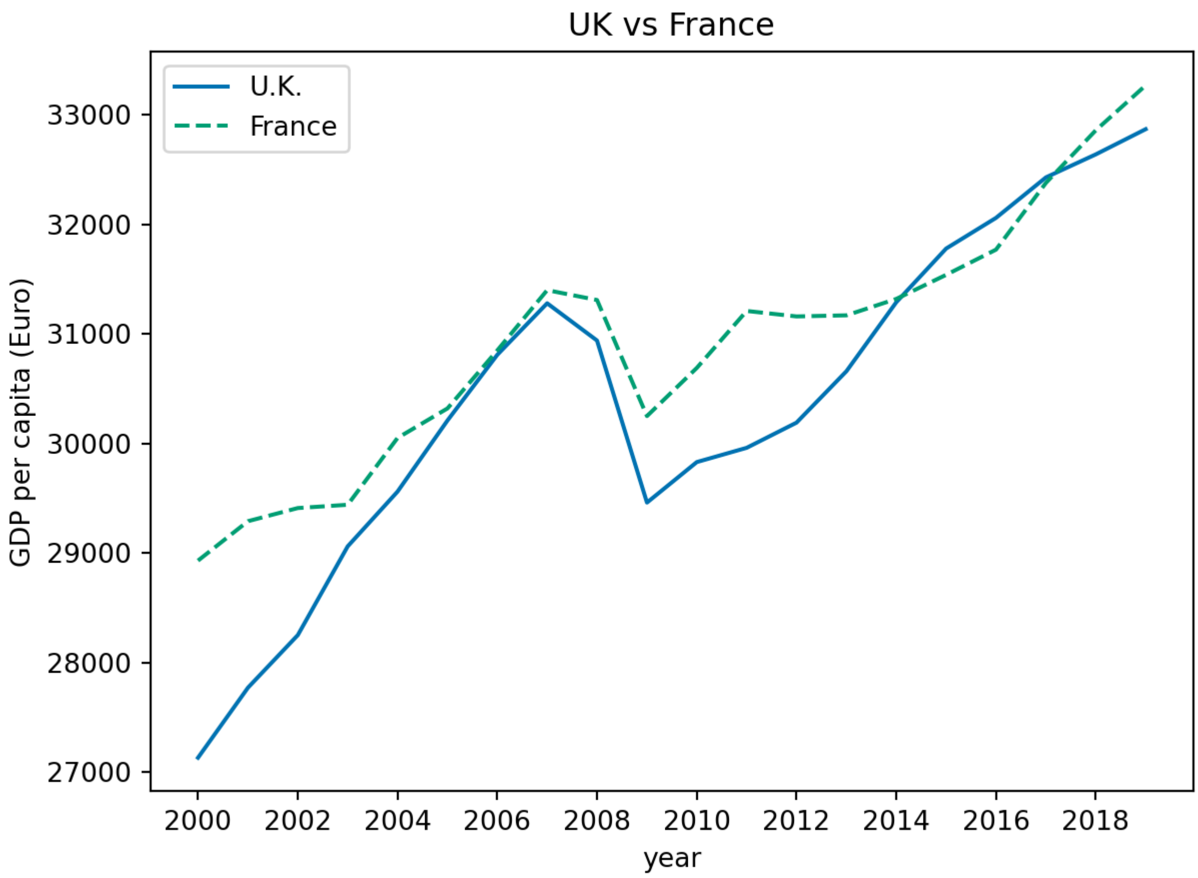
Above we have shown how to plot using `ax`, as it is handy to specify which subplot to plot by working on `ax`. If we only have one subplot, we can work on either `plt` or `ax`. They will give you the same figure, although the syntax is different. Below we show how the same plot can be made by `plt` (first plot) and `ax` (second plot):

```
plt.plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.')
plt.plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France')
plt.legend()

plt.xlabel('year')
plt.ylabel('GDP per capita (Euro)')
plt.title("UK vs France")
```

```
plt.show()
```




```
fig, ax = plt.subplots()

ax.plot(gdp_uk['year'], gdp_uk['value'], label = 'U.K.')
ax.plot(gdp_fr['year'], gdp_fr['value'], '--', label = 'France')
ax.legend()

ax.set_xlabel('year') # note it was plt.xlabel('year') above
ax.set_ylabel('GDP per capita (Euro)')

ax.title.set_text("UK vs France") # note it was plt.title("UK vs France") above
plt.show()
```

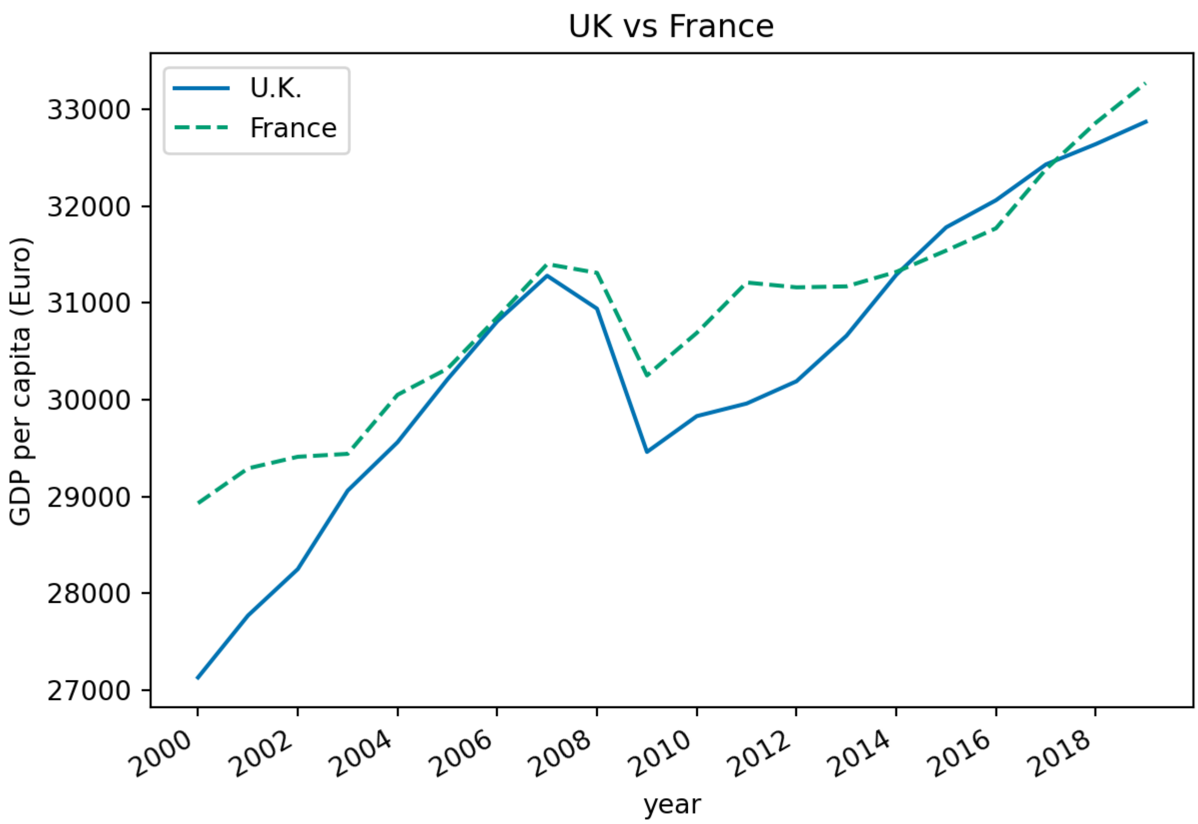


Line Plot With **pandas** Data Structure

Alternatively, you can plot by using the method `plot()` from `pandas` `Series` or `DataFrame`. Behind the scene, `pandas` calls the plot functions from **matplotlib**, so you can consider `pandas`' `plot()` method is just a convenient shortcut to plot. As you can see below, plotting via `pandas` `plot()` method can create the same plot (slight different in figure size and the x label rotation for this particular example) as above when we use **matplotlib** `plot()` function. Sometimes, though, it is more convenient to plot via `pandas`. See the side by side bar plot example later in this page.

```
fig, ax = plt.subplots()


gdp_uk.plot(x = 'year', y = 'value', ax = ax)
gdp_fr.plot(x = 'year', y = 'value', style = '--', ax = ax)
ax.legend(["U.K.", "France"])


ax.set_xlabel('year')
ax.set_ylabel('Euro')
ax.set_ylabel('GDP per capita (Euro)')
ax.title.set_text("UK vs France")
```

```
plt.show()
```

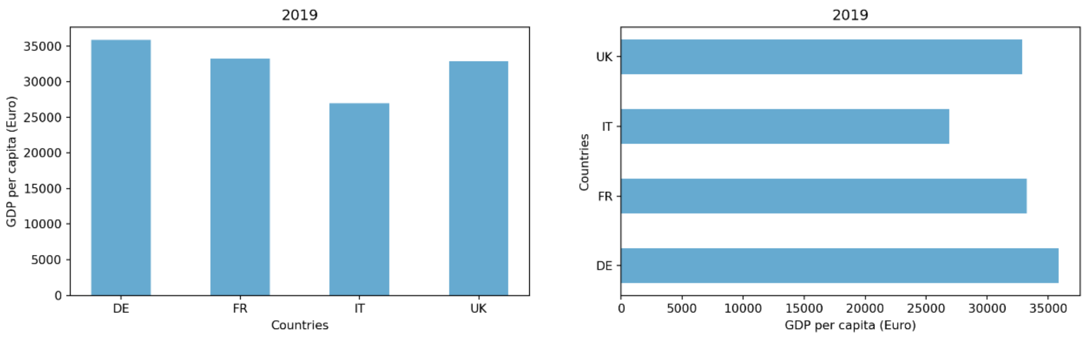


Bar Plot

Here we continue to plot the GDP per capital data, but now we are plotting against different countries rather than time. We can plot the bars vertically or horizontally:

```
fig, ax = plt.subplots(1, 2, figsize=(15, 4))


# ======== vertical bars ========

# ax = ax[0] indicates its the first subplot


ax[0].bar(gdp_wide[2019].index, gdp_wide[2019], alpha = 0.6, # the argument alpha = 0.6 makes the bars slightly transparent

          width = 0.5) # the argument width = 0.5 makes the bars thinner

ax[0].set_ylabel('GDP per capita (Euro)')

ax[0].set_xlabel('Countries')

ax[0].title.set_text("2019")


# ======== horizontal bars ========
```

```
# ax = ax[1] indicates its the second subplot
ax[1].barh(gdp_wide[2019].index, gdp_wide[2019], alpha = 0.6,
           height = 0.5) # note here we use "height" to make the bars thinner
ax[1].set_xlabel('GDP per capita (Euro)')
ax[1].set_ylabel('Countries')
ax[1].title.set_text("2019")

plt.show()
```

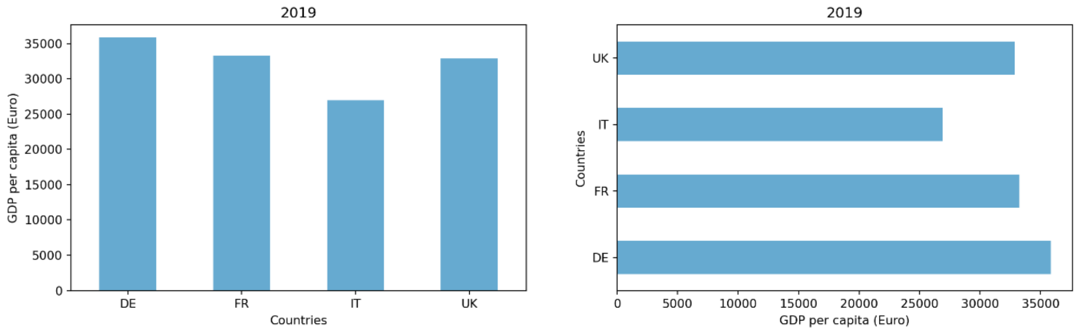


Bar Plot With **pandas**

Like the line plots, we can plot bar plot using `DataFrame` `plot()` instead. Note the `plot()` arguments are different.

```
fig, ax = plt.subplots(1, 2, figsize=(15, 4))


# ======== vertical bars ========
gdp_wide[2019].plot.bar(x = 'country', y = 'value', rot = 0, # rot = 0: what if you omit it?
                        ax = ax[0], # note we need to give ax as an argument
                        legend = False, alpha = 0.6) # legend = False: not to show legend
ax[0].set_ylabel('GDP per capita (Euro)')
ax[0].set_xlabel('Countries')
ax[0].title.set_text("2019")



# ======== horizontal bars ========
gdp_wide[2019].plot.barh(x='country', y='value',  ax = ax[1], legend = False, alpha = 0.6)
ax[1].set_xlabel('GDP per capita (Euro)')
ax[1].set_ylabel('Countries')
ax[1].title.set_text("2019")
```

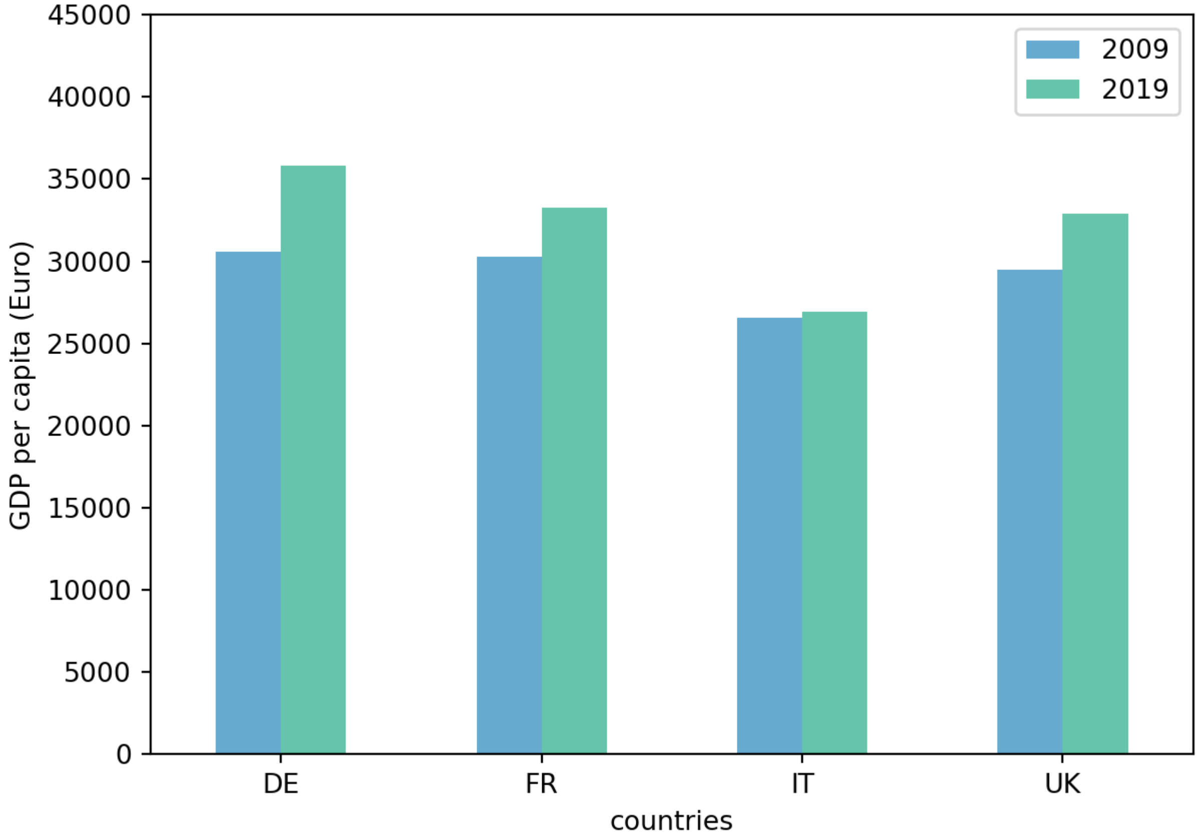
```
plt.show()
```



Side By Side barplots

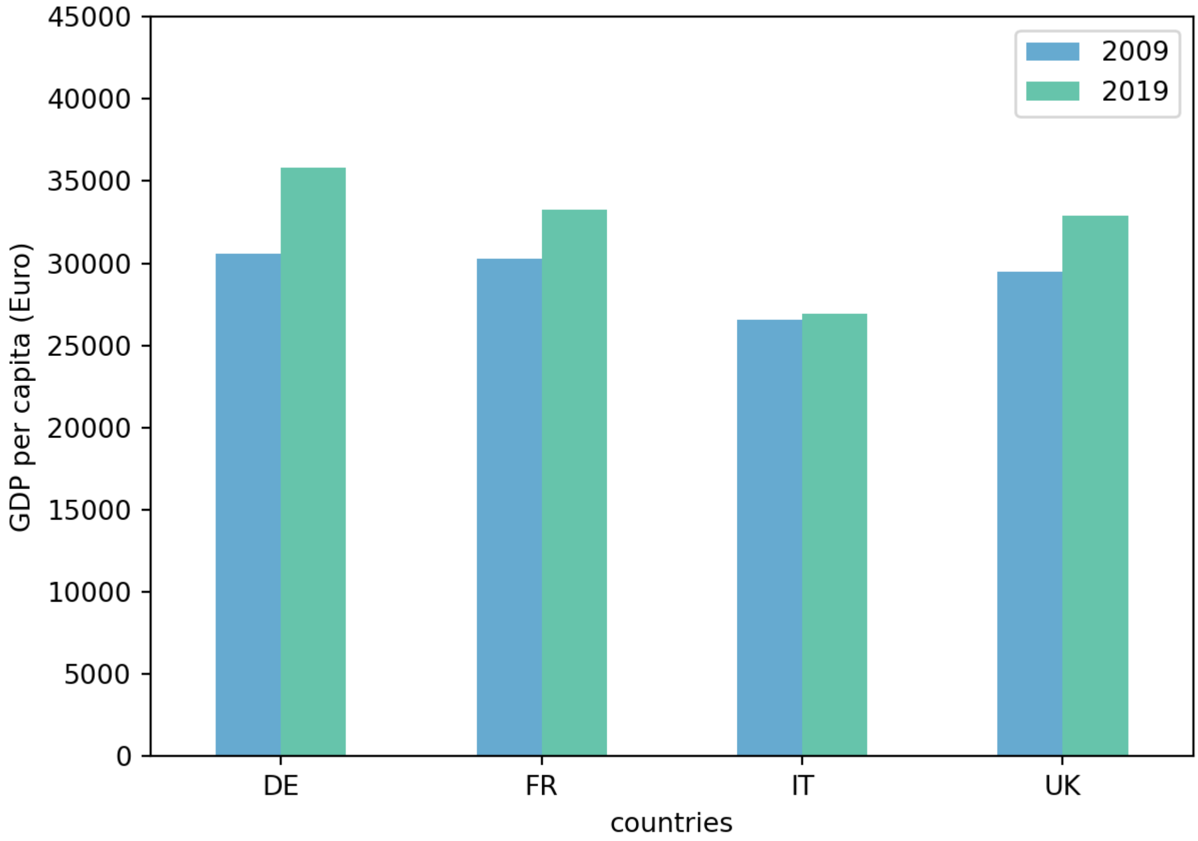
We can have side by side bars by provide using `DataFrame` `plot()` function. You could use **matplotlib** `plot()` but it is easier with `DataFrame` `plot()`. Here we plot the GDP per capital from year `2009` and `2019` side by side:

```
ax = gdp_wide[[2009,2019]].plot.bar(rot=0, alpha = 0.6)


ax.set_ylim(top = 45000) # make ylim max to be larger so that the legend and the ba
rs are not overlapping

ax.set_ylabel('GDP per capita (Euro)')

ax.set_xlabel('countries')


plt.show()
```



Note here we did not call `fig, ax = plt.subplots()` and then pass `ax` into `Dataframe plot()`. Instead we use the `ax` created from `Dataframe plot()`. We could of course use `fig, ax = plt.subplots()` and then pass `ax` into `Dataframe plot()` (as shown below), but we do not have to.

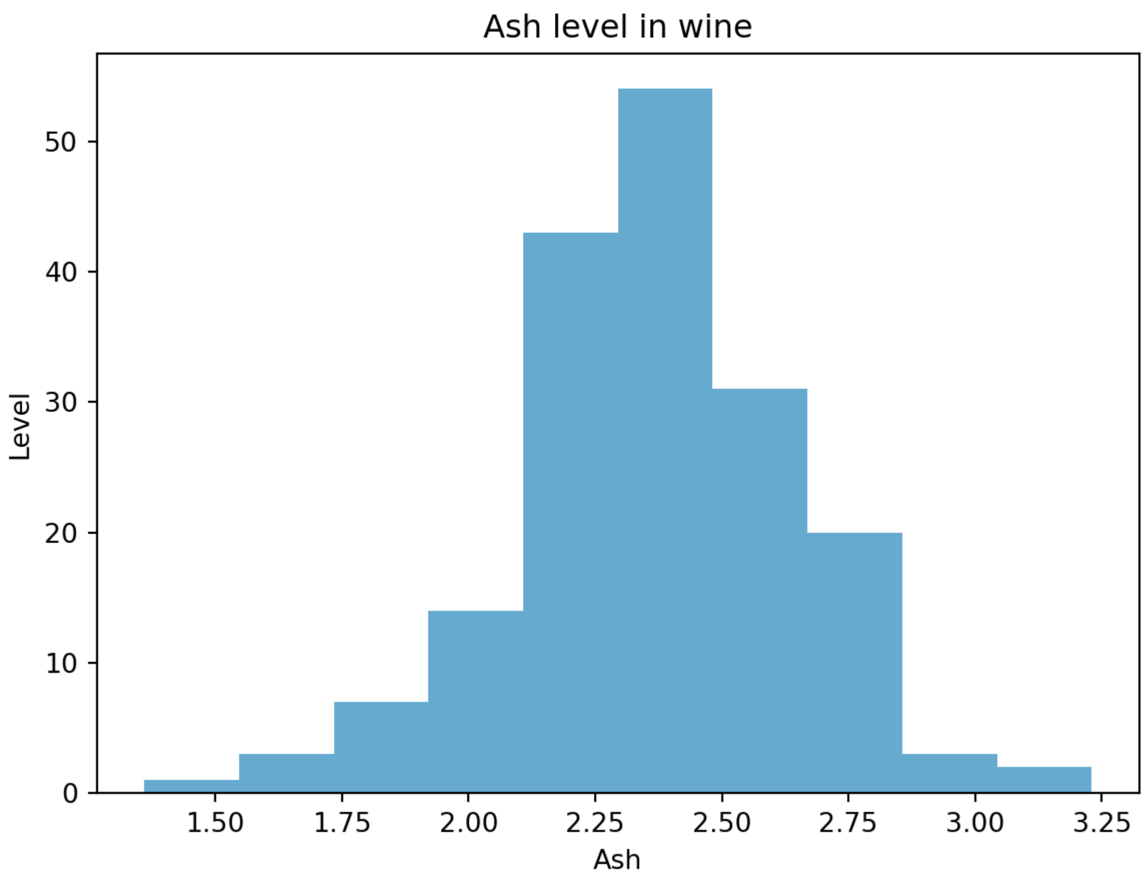
```
fig, ax = plt.subplots()


gdp_wide[[2009,2019]].plot.bar(rot=0, alpha = 0.6, ax = ax)


ax.set_ylim(top = 45000) # make ylim max to be larger so that the legend and the ba
rs are not overlapping

ax.set_ylabel('GDP per capita (Euro)')

ax.set_xlabel('countries')


plt.show()
```



Histogram: Showing the Distribution of a Variable

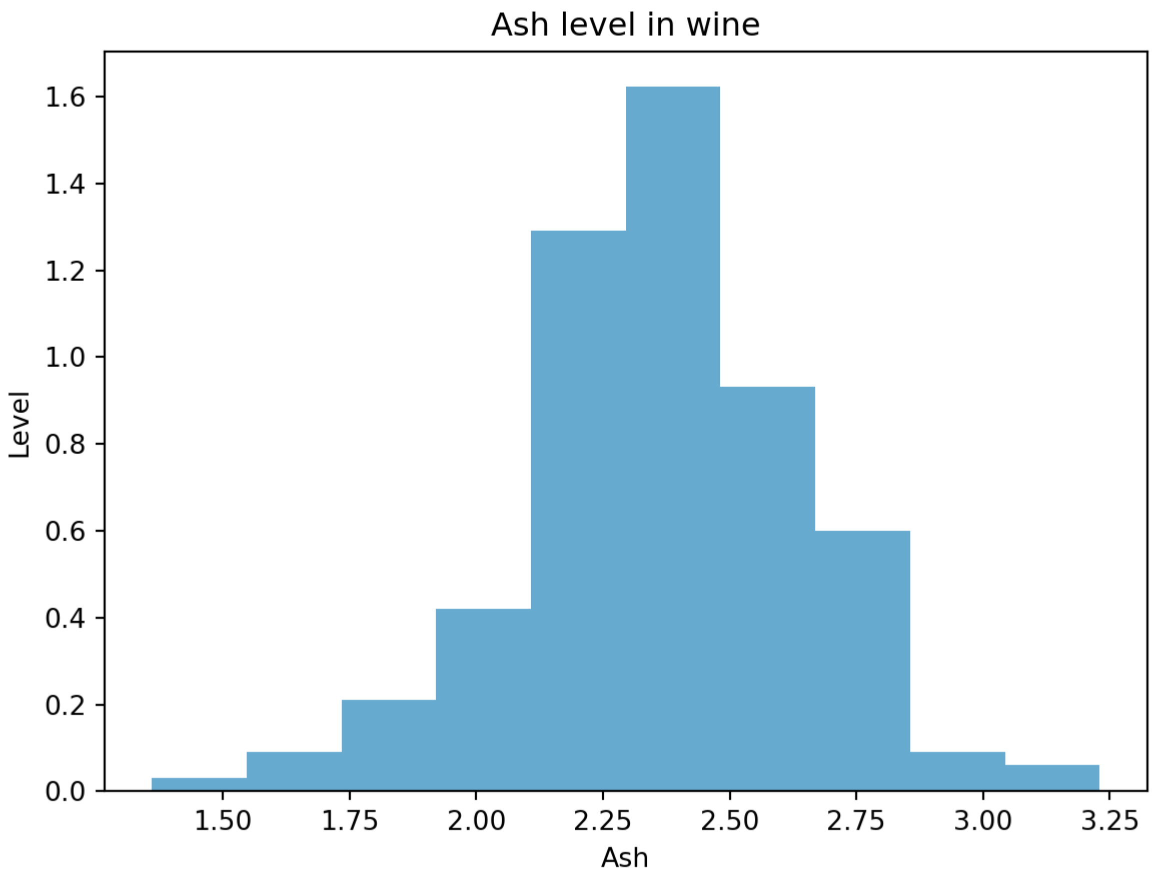
Below we create a histogram via `plt` to show the frequency of observations (here are the wines) that fall into a given `ash` level interval. You can use `ax` instead if you want to (although the syntax for setting labels and title is not the same).

```
fig, ax = plt.subplots()


plt.hist(wine['ash'], 10, alpha = 0.6)

plt.xlabel('Ash')

plt.ylabel('Level')

plt.title("Ash level in wine")


plt.show()
```



In order for the histogram to represent distribution (i.e. the area of the histogram is 1), add the argument `density = 1`:

```
fig, ax = plt.subplots()


plt.hist(wine['ash'], 10, density = 1, alpha = 0.6)

plt.xlabel('Ash')

plt.ylabel('Level')

plt.title("Ash level in wine")


plt.show()
```

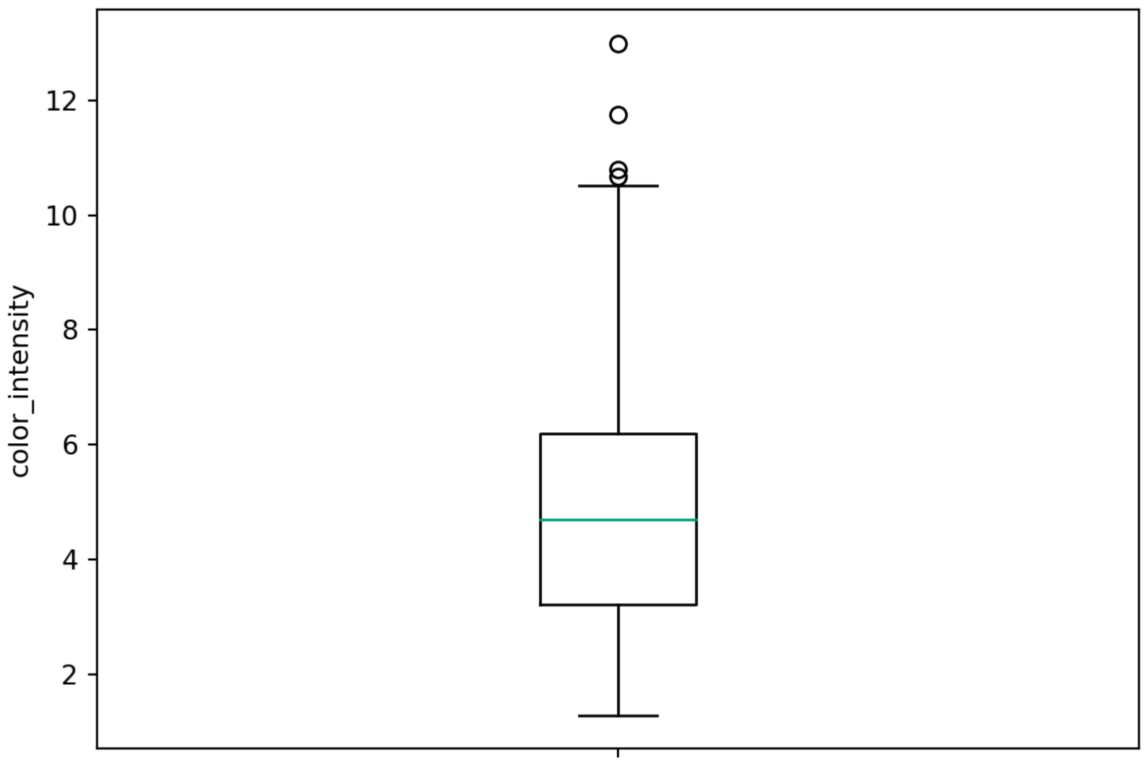


Boxplot and variation: showing the distribution of a variable

In Python boxplot (and its variations) can be created via **matplotlib** or **seaborn**. We will use the boxplots to compare the chemistry composition of wines with different class labels.

We start at looking at the colour intensity regardless the class label:

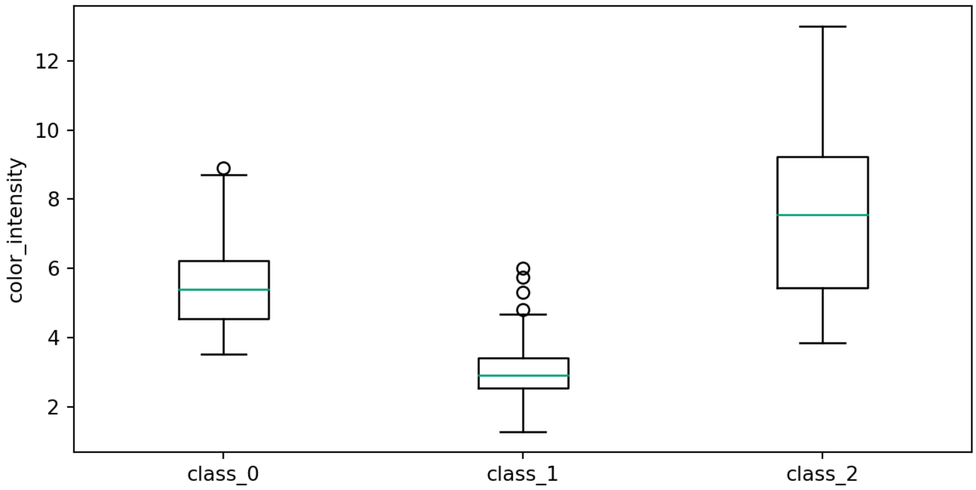
```
fig, ax = plt.subplots()


ax.boxplot(wine['color_intensity'])

ax.set_ylabel('color_intensity')

ax.set_xticklabels([''])


plt.show()
```



Side By Side (or grouped) Boxplot

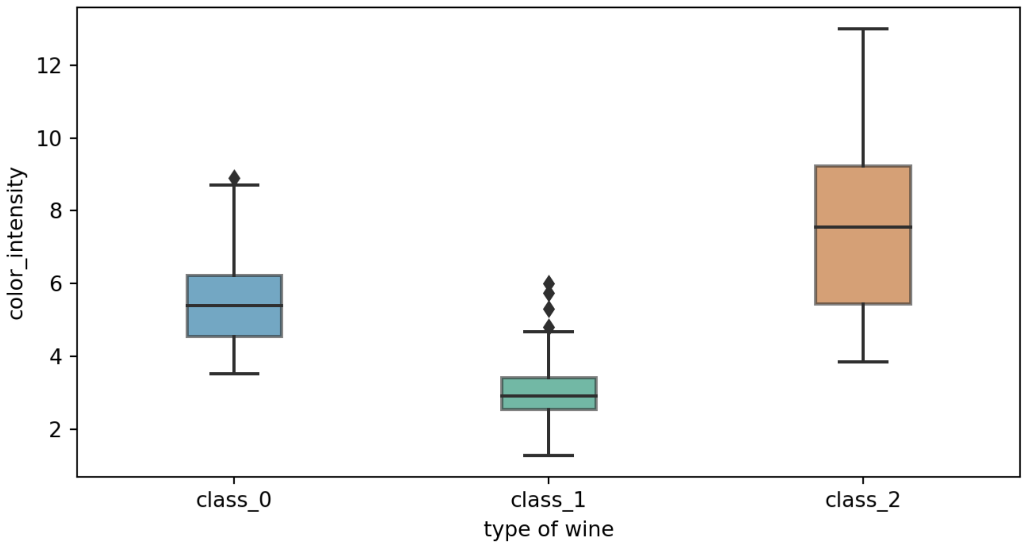
Often, it is more useful to have boxplots side by side to compare the same variables conditional on some factors. For example, here we want to see the colour intensity of wines given the type of the wine. With **matplotlib**, we need to first convert our long `wine['color_intensity']` data to wide, and then create the boxplot:

```
fig, ax = plt.subplots(figsize = (8, 4))


ax.boxplot([wine['color_intensity'][wine['target']=='class_0'],
            wine['color_intensity'][wine['target']=='class_1'],
            wine['color_intensity'][wine['target']=='class_2']])
ax.set_ylabel('color_intensity')
ax.set_xticklabels(['class_0', 'class_1', 'class_2'])
plt.show()
```



Side by side boxplot can be created more easily via **seaborn**. With **seaborn**, we can use the original `wine DataFrame` (`'target'` column in the `wine DataFrame` contains the class label information):

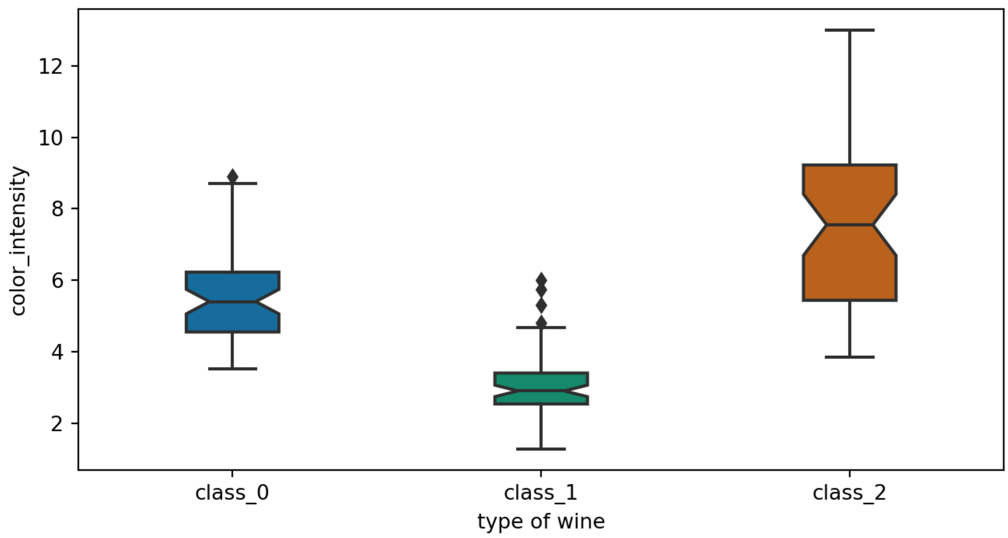
```
fig, ax = plt.subplots(figsize = (8, 4))

sns.boxplot(data = wine, x = 'target', y = 'color_intensity', width = 0.3,
            boxprops = dict(alpha=0.6)) # note in seaborn, alpha (and other paramet
ers) is set in a different way

ax.set_xlabel("type of wine")

plt.show()
```



Boxplot Variation

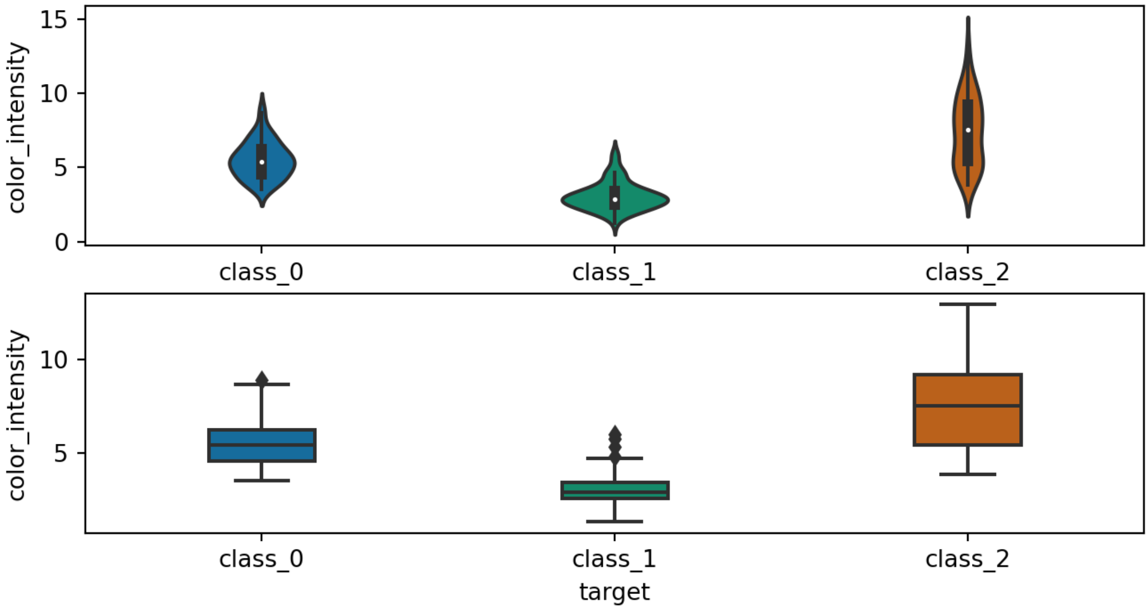
To create a notched boxplot, add the argument `notch = True`:

```
plt.subplots(figsize=(8, 4))

ax = sns.boxplot(data = wine, x = 'target', y = 'color_intensity', notch = True, wi
dth = 0.3)


ax.set_xlabel("type of wine")


plt.show()
```



To create a violin plot, we use the `violinplot()`. Below boxplot is plotted as well to show you how you can have subplots with **seaborn**:

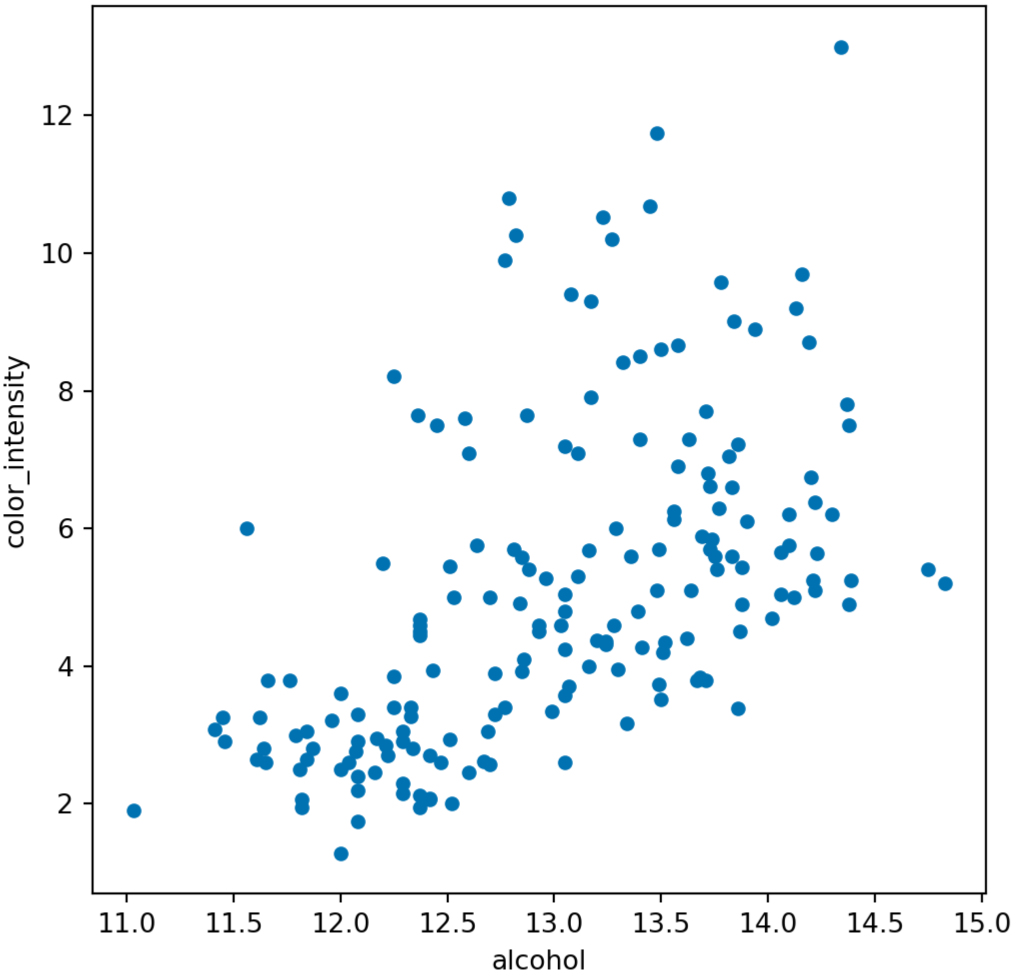
```
fig, ax = plt.subplots(2, 1, figsize=(8, 4))


sns.violinplot(data = wine, x = 'target', y = 'color_intensity', ax = ax[0], width
= 0.3)

sns.boxplot(data = wine, x = 'target', y = 'color_intensity', ax = ax[1], width = 0
.3)


ax[0].set_xlabel("type of wine")


plt.show()
```



Scatter: Showing Relations Between Two Variables

Below we create a scatter plot via `DataFrame` `plot()` to show how the two chemistry composition levels relate to each other (you can use **matplotlib** instead if you want to).

```
ax = wine.plot.scatter('alcohol', 'color_intensity', figsize=(6, 6)) # use a square
plot size


plt.show()
```

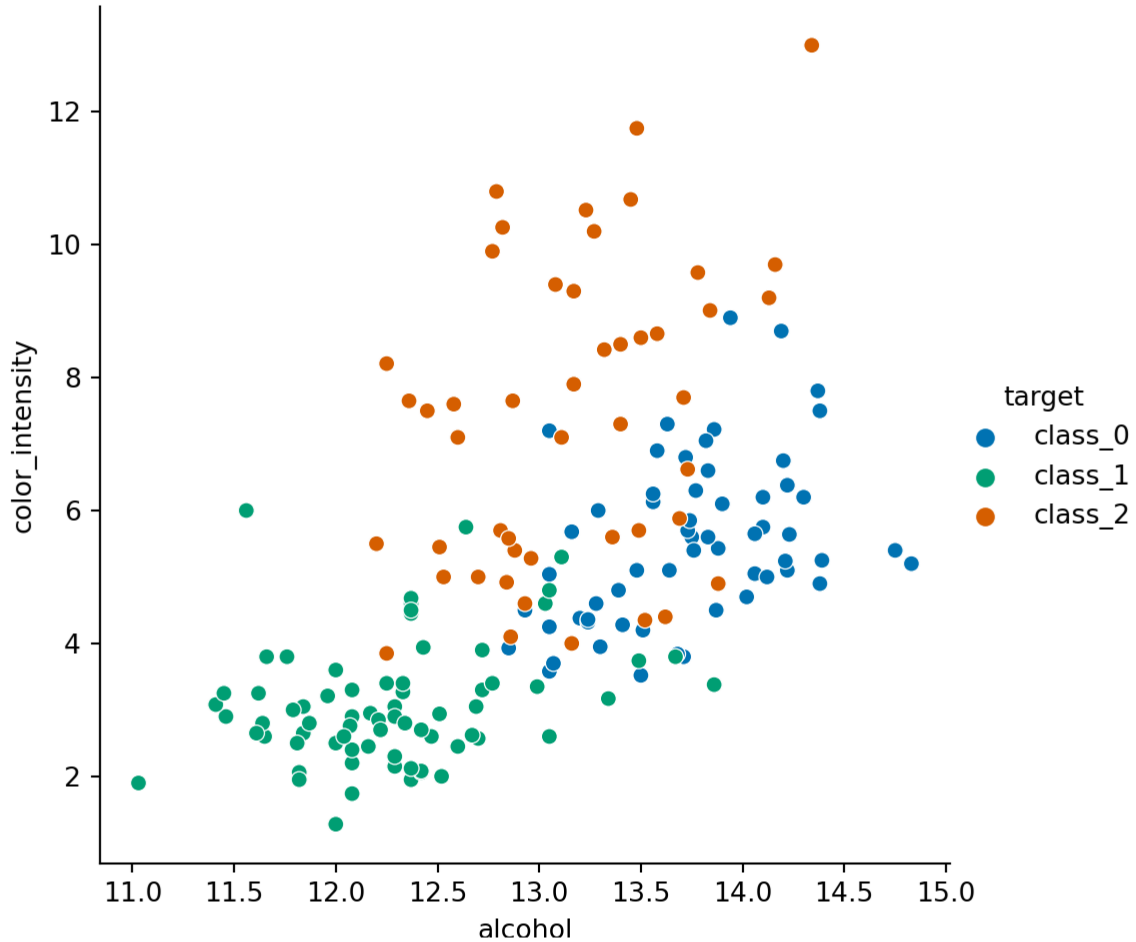


Scatter Plot With **seaborn**

With **seaborn**, we can use `relplot()` to plot the scattered plot. We can set the optional argument `hue`, `style`, `size`, etc to show an extra dimension of the data.

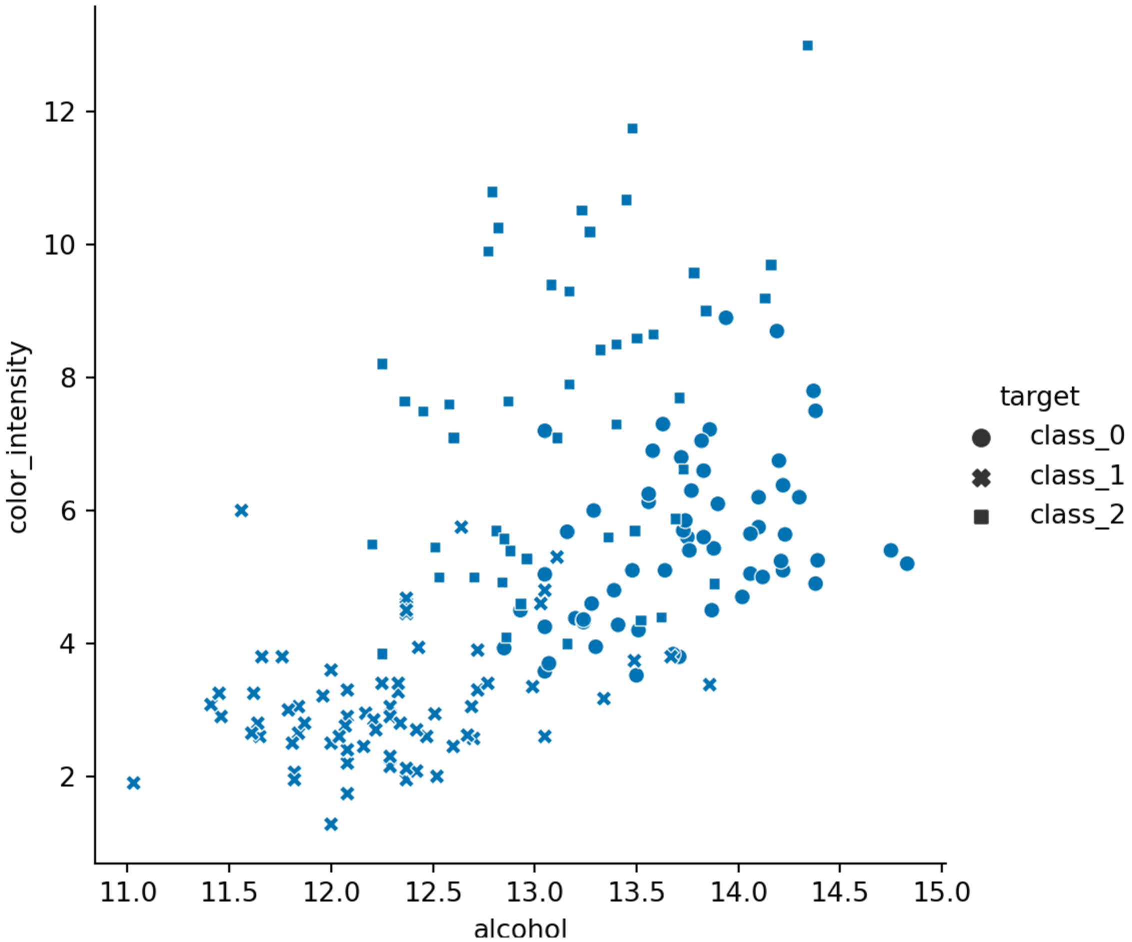
For example, here we plot the `color_intensity` against `alcohol`, with the colour of scattered points depends on which group the wines belong to by providing the optional argument `hue`:

```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "target", data = wine)
plt.show()
```



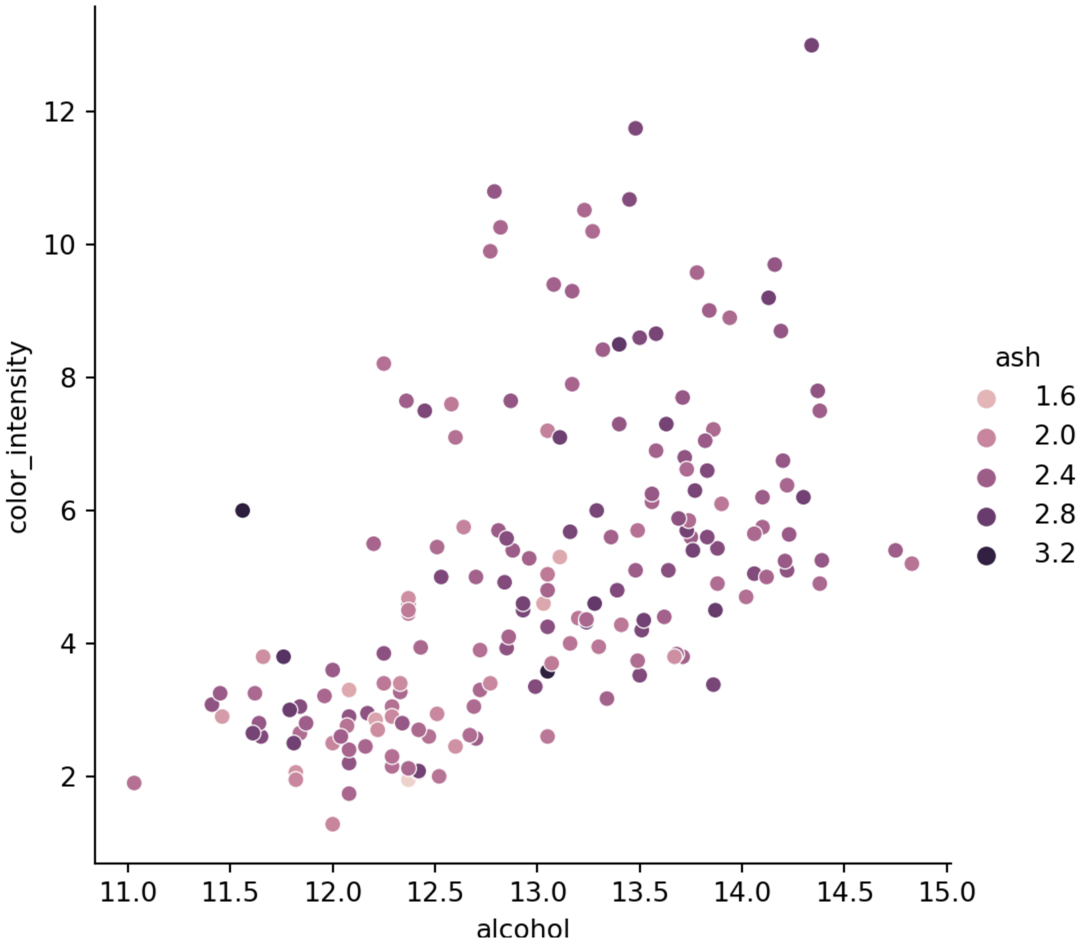
We can also use the shape instead of colour to represent the group membership by `style`:

```
sns.relplot(x = 'alcohol', y = 'color_intensity', style="target", data = wine)
plt.show()
```



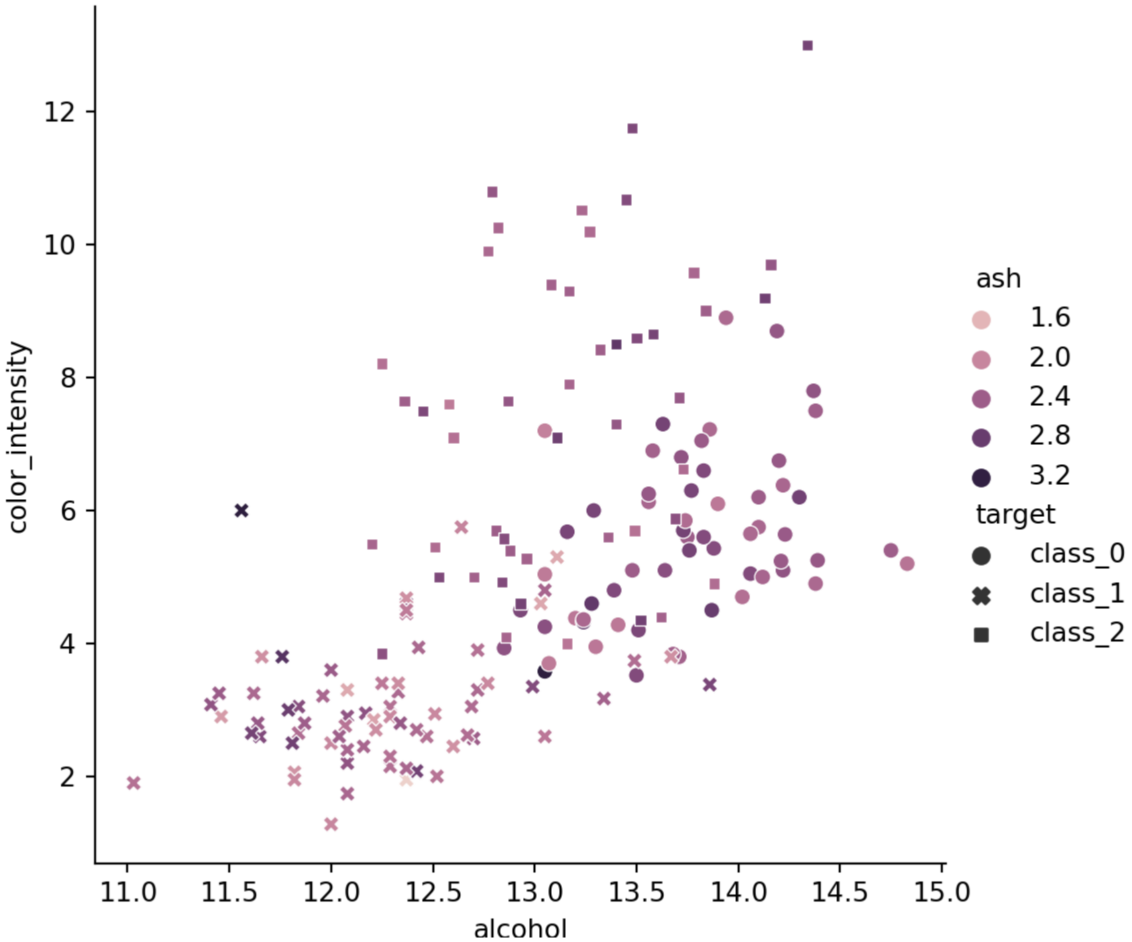
The optional argument `hue` can also be used for continuous variable. Here the colour of the points depends on the value of `ash`

```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "ash", data = wine)
plt.show()
```



We can also have a graph showing four different variables by using both `hue` and `style`:

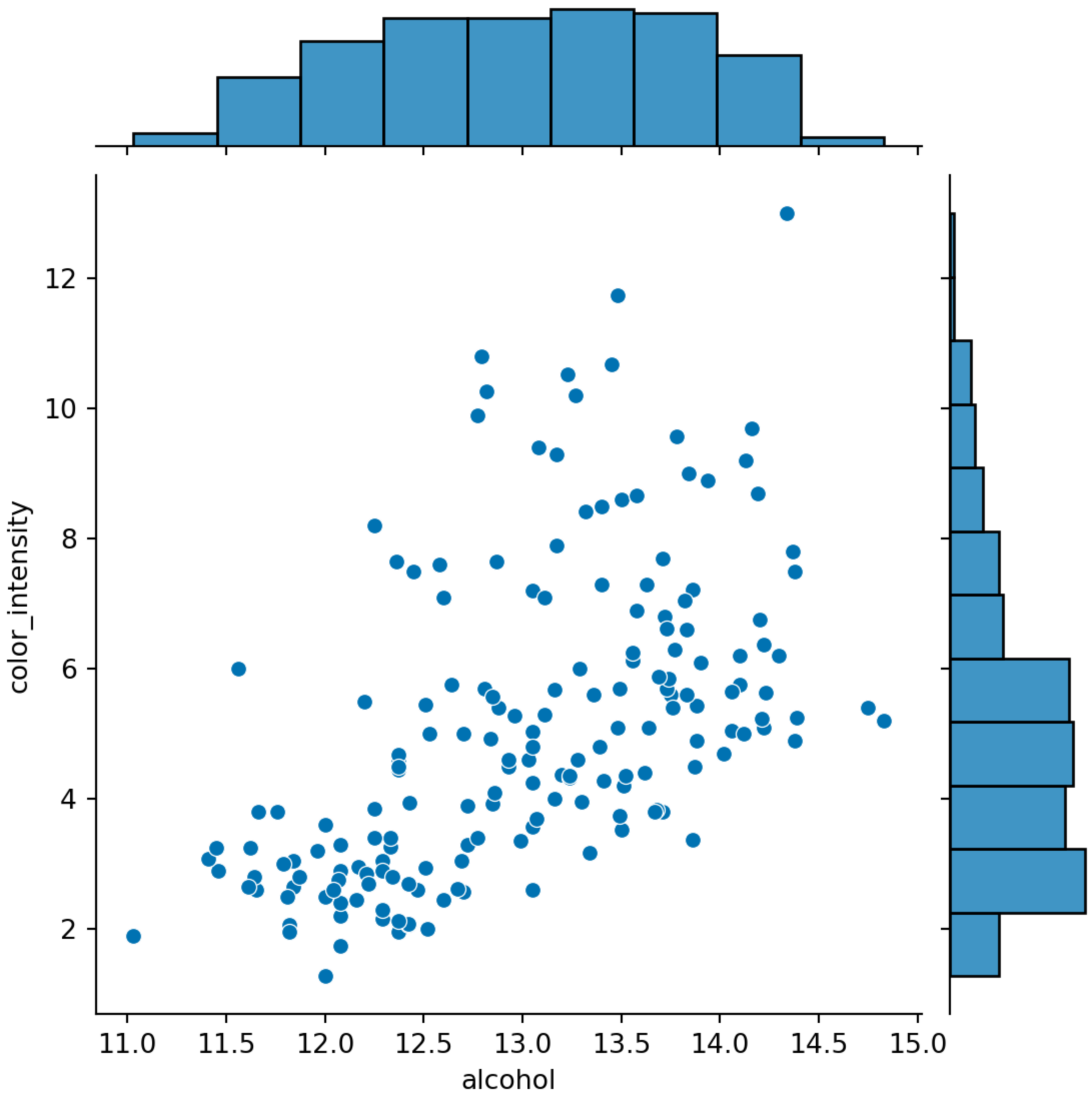
```
sns.relplot(x = 'alcohol', y = 'color_intensity', hue = "ash", style = 'target', da
ta = wine)

plt.show()
```



More on Scatterplots With **seaborn**

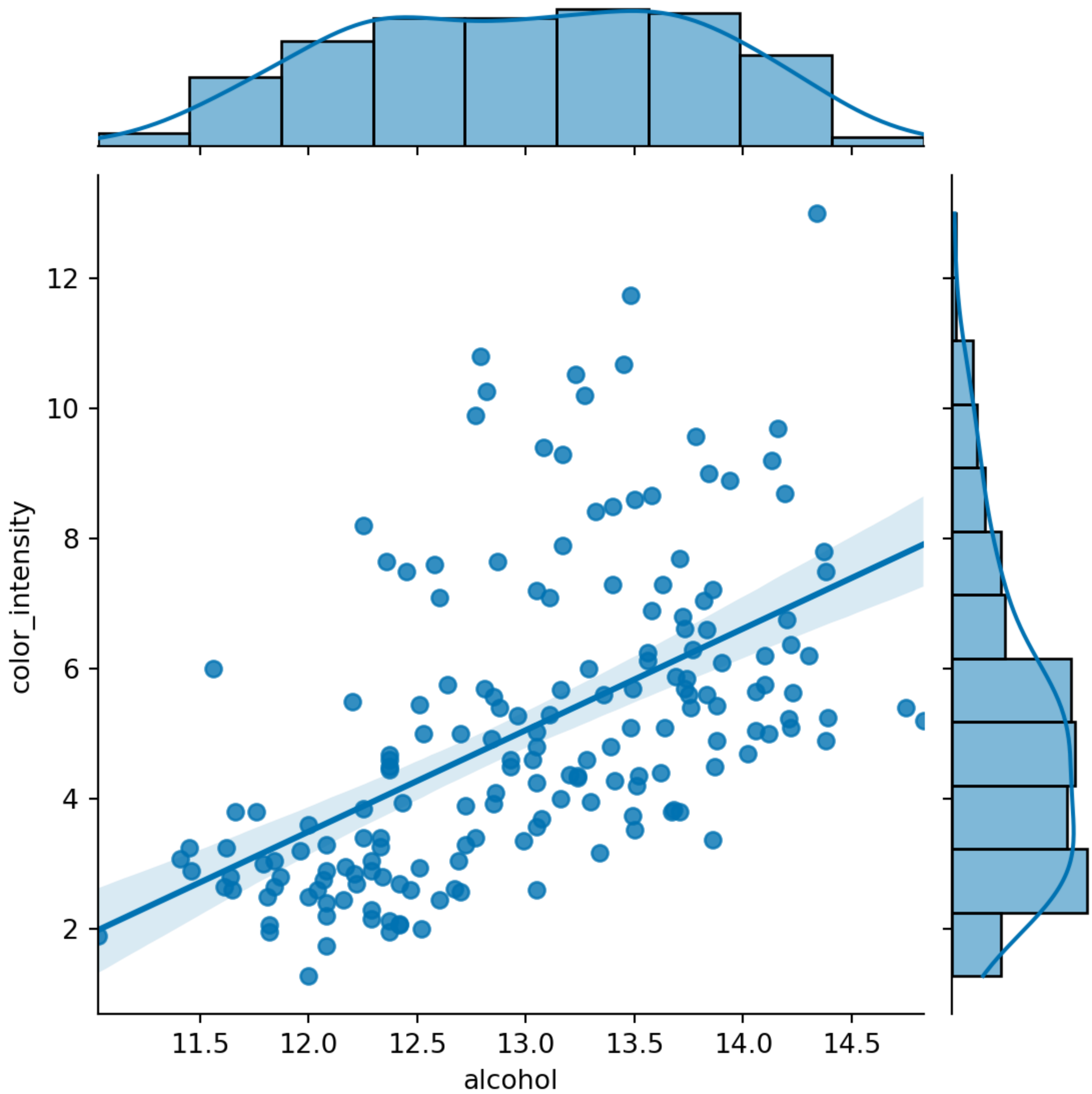
With **seaborn**, you can do some fancier scatter plots. For example, showing the marginal histogram together the scatterplot:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine)
plt.show()
```



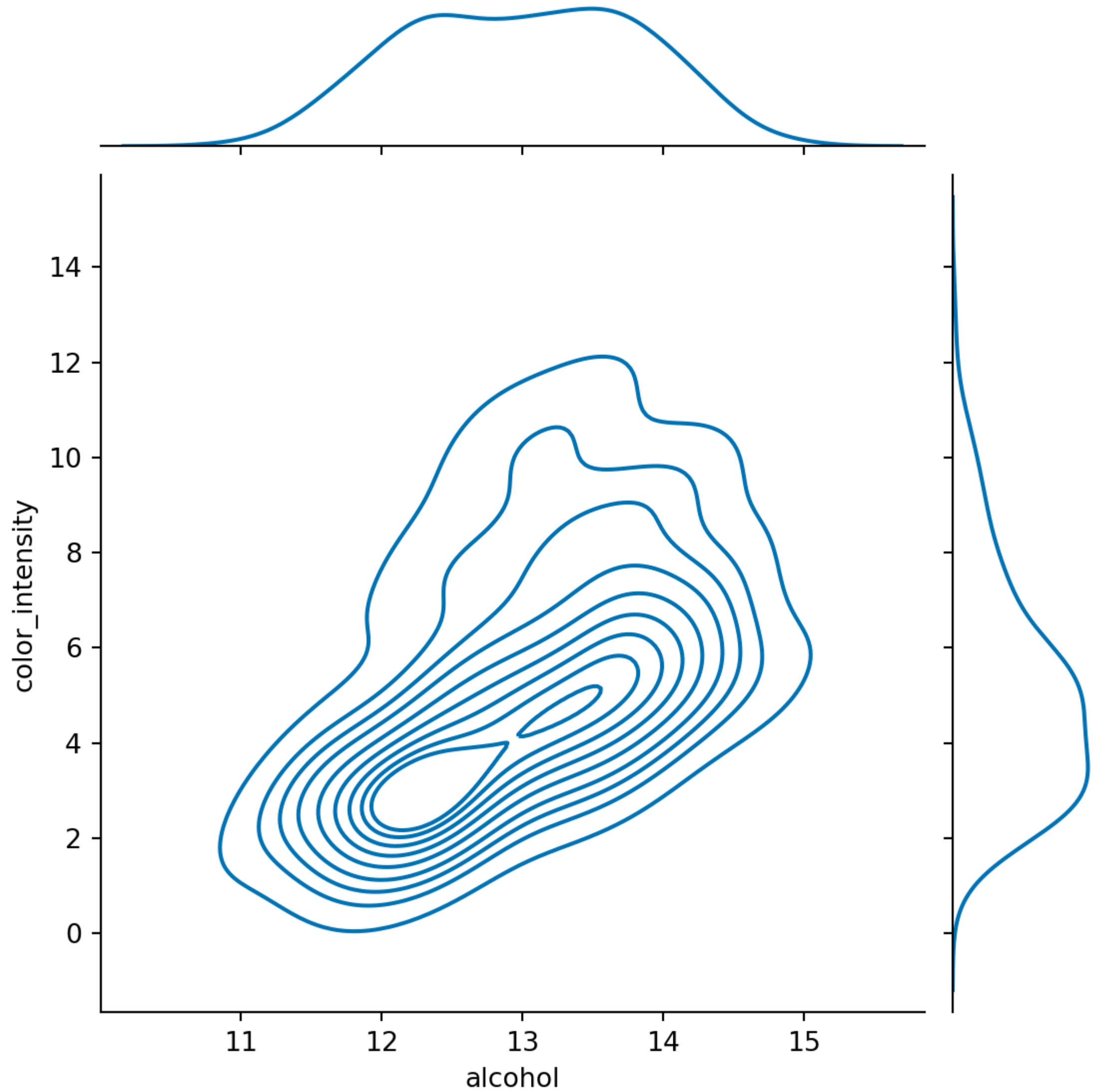
We can also add a linear regression fit and univariate KDE curves using `kind='reg'`:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind="reg")
plt.show()
```



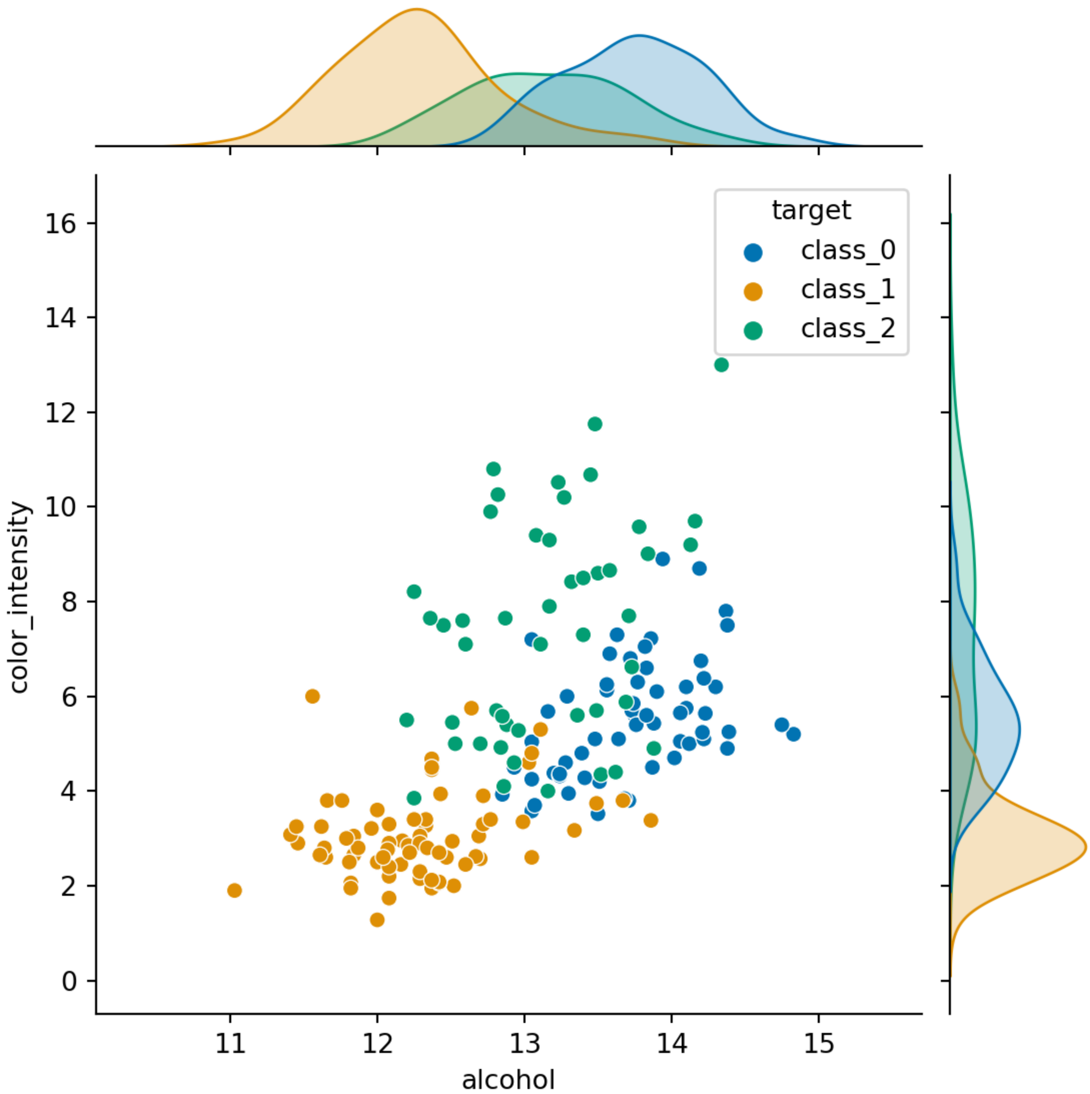
… or with the bivariate and univariate Kernel density estimation:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind = 'kde')
plt.show()
```

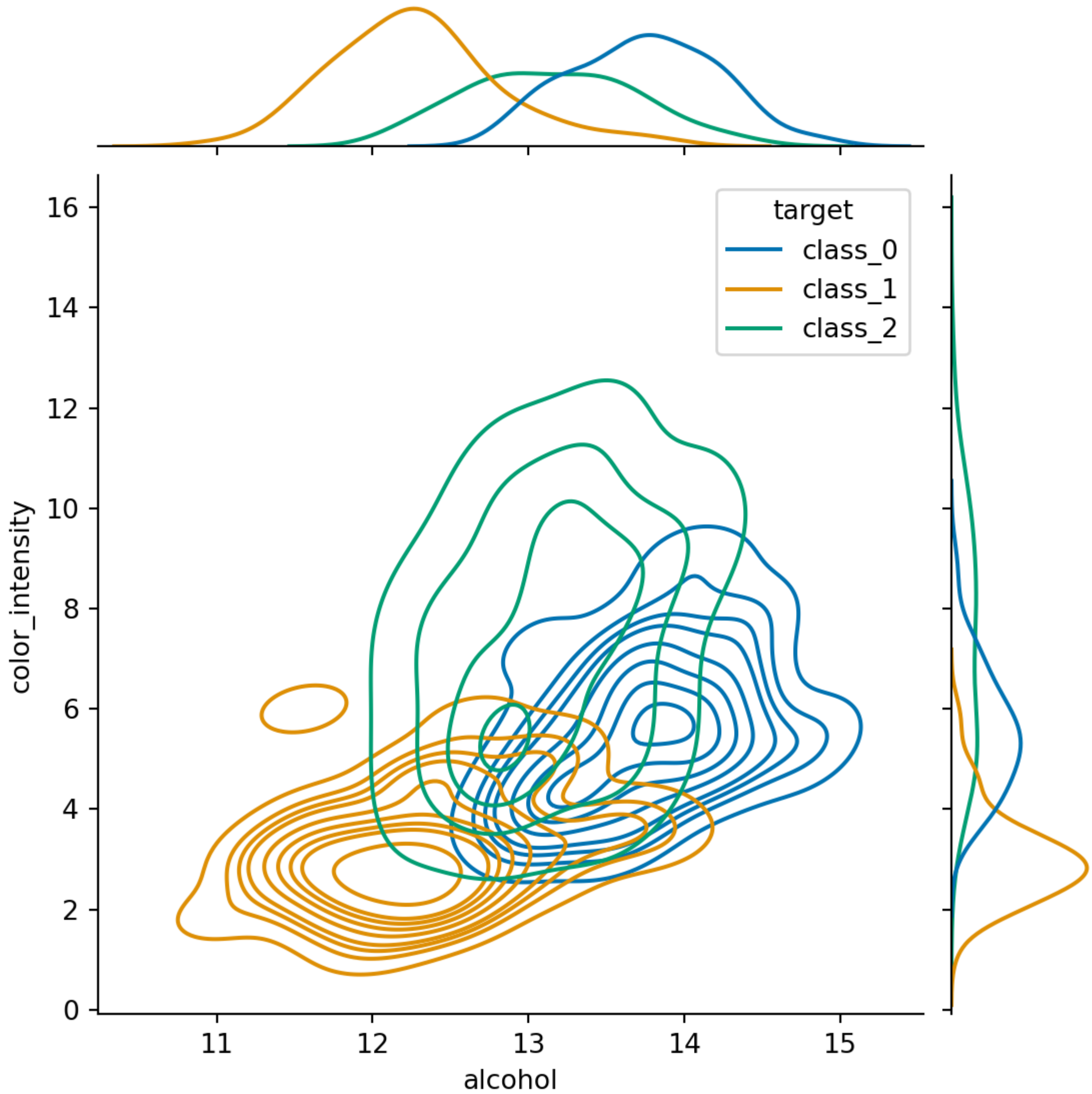


You can have show the scatter points in different colours according to the label as well:

```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, hue = "target", pa
lette = my_palette[:3])

plt.show()
```




```
sns.jointplot(x = 'alcohol', y = 'color_intensity', data = wine, kind = 'kde', hue
= "target", palette = my_palette[:3])

plt.show()
```

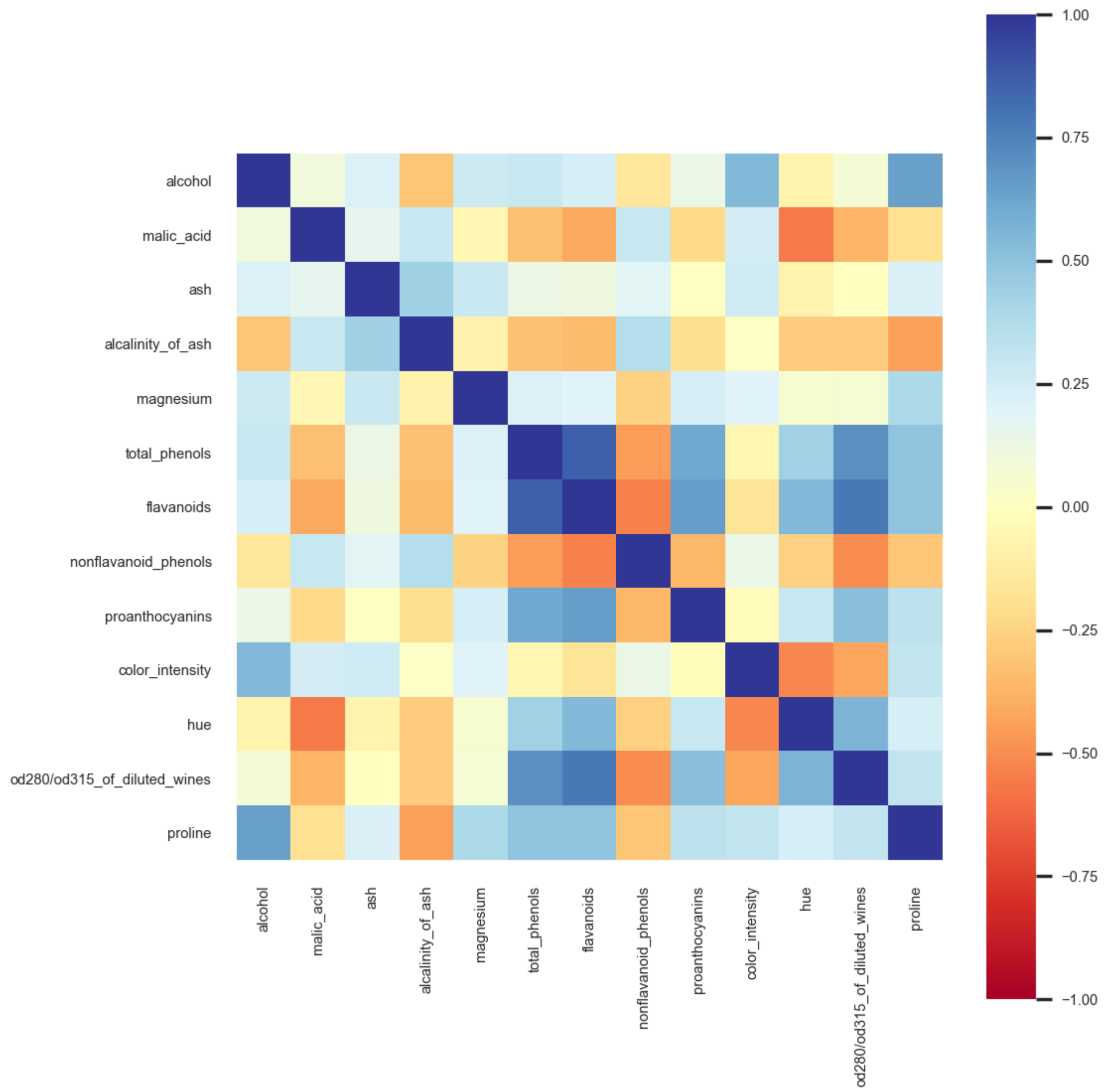


See here to see what other scatter plot you can create with **seaborn**.

Heat Map

To create a heatmap, we can use `sns.heatmap()`. Here we create a heatmap to show the correlation of the wine chemistry composition:

```
sns.set(font_scale=0.5)
ax = sns.heatmap(wine.corr(),
                  center= 0, vmin = -1, vmax = 1, cmap= "RdYlBu", # this set the co
lour of the heatmap
                  square=True) # set the heatmap to be square shape
ax.figure.tight_layout() # this makes sure all labels are shown in the plot
plt.show()
```



The argument `cmap= "RdYlBu"` sets the palette, with red corresponds to lowest value, yellow corresponds to the middle value and blue corresponds to the highest value. `center = 0` sets `0` as the middle value, and `vmax = 1` and `vmax = -1` set the maximum and minimum values to be `1` and `-1`. To set heatmap with different colours, see here.

Exporting Plots

Similar to R, you can save the figure as a separate file. In Python, you can do it by using the function `savefig()`:

```
fig, ax = plt.subplots()


gdp_uk.plot(x = 'year', y = 'value', ax = ax)
gdp_fr.plot(x = 'year', y = 'value', style = '--', ax = ax)
ax.legend(["U.K.", "France"])


ax.set_xlabel('year')
```

```
ax.set_ylabel('Euro')

ax.set_ylabel('GDP per capita (Euro)')

ax.title.set_text("UK vs France")


plt.savefig('gdp_uk_france.png') # you can find the file in the folder where you ru
n this line of code
```

Useful Links and Resources

- McKinney, W. (2013). Python for data analysis. Chapter 8.
- Matplotlib tutorial
- seaborn jointplot

Network Visualisations

** Note: The code chunks below should be run in the following order **

Networks

A network \(G\) consists of a set \(V\) of vertices and a set \(E\) of edges. An edge indicates a relationship between two vertices. Networks provide standard tools for visualisation in data science and related fields. Below are some examples of kinds of real world network data that can be represented by networks:

- Social networks: Representing people and their social interactions. Examples include looking into Facebook, LinkedIn, email exchanges, etc.
- Knowledge networks: Representing entities and their relationships. For example Google knowledge graph, Bing Satori, Freebase, Yago, Wordnet, etc.
- Collaboration networks: Representing people and their collaborations. Examples include Co-authorships from dblp, Google Scholar, Microsoft Academic search, etc.
- Product purchase networks: Representing who bought what. Example: Amazon product purchases
- Reviews: Ratings of products or services provided by users. Example: TripAdvisor
- Road networks, communication networks.

There are several reasons for visualising networks:

- For exploratory data analysis by visual inspection of network drawings
- For visual detection of network structures, e.g. community detection in social networks, tree or feed-forward structures.
- For communicating network properties

For more on networks you can read Robinson et al. (2015).

Networks in Python

The standard Python package for the creation, manipulation, and study of the structure of networks is **networkx**. A tutorial for it can be found here. To install **networkx** type the following in the terminal or Anaconda prompt:

```
pip install networkx
```

Below are some standard layout styles for networks produced by the **networkx** package:

- `circular`: Position nodes on a circle
- `random`: Position nodes uniformly at random in the unit square
- `shell`: Position nodes in concentric circles
- `spring`: Position nodes using the Fruchterman-Reingold force-directed algorithm
- `spectral`: position nodes using the eigenvectors of the network Laplacian

We will illustrate the above in a real world dataset. In next two subsections, we will explore and pre-process the data and then we will sample from the above styles.

GitHub Organisations Data

As a working example, we will study the activities of users across different GitHub organizations by a representation of data as a network. The vertices of the network will represent organisations and an edge between two vertices will indicate that there is at least one user who initiated an event to at least one repository in each of the two corresponding organizations. The edge weights are defined as the number of such users. We will use the data retrieved from GitHub archive for 2 March 2015 that are contained in the attached file `2015-03-02.csv`.

We will need to do a fair amount or pre-processing to bring the data in the desired format. This can be seen as a good data wrangling exercise. First, we initialise Python and load the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# load the data
fdate = '2015-03-02'
df = pd.read_csv(fdate + '.csv', parse_dates=['created_at'])
df.head()
```

	created_at	id	actor.login	org.login	repo.name	type
0	2015-03-02	2615962109	pdgago	simbiotica	simbiotica/charicharts	CreateEvent
1	2015-03-02	2615962111	micmania1	NaN	micmania1/silverstripe-cms	PushEvent
2	2015-03-02	2615962112	xsb	bitpay	bitpay/insight-api	WatchEvent
3	2015-03-02	2615962113	dkorolev	KnowSheet	KnowSheet/Bricks	PullRequestReviewCommentEvent
4	2015-03-02	2615962114	edwin-pers	NaN	edwin-pers/neuromancer	PushEvent

The dataset contains several variables indicating the user (`actor.login`) and the GitHub organisation (`org.login`) as well as other variables such as date created, ID, name of the repository and type of event. To check which types of events exist we use the code below:

```
# unique event types
df['type'].unique()
array(['CreateEvent', 'PushEvent', 'WatchEvent',
       'PullRequestReviewCommentEvent', 'IssueCommentEvent',
       'PullRequestEvent', 'DeleteEvent', 'CommitCommentEvent',
       'ForkEvent', 'IssuesEvent', 'GollumEvent', 'MemberEvent',
       'ReleaseEvent', 'PublicEvent'], dtype=object)
```

Next, we want to transform the data to triplets: (actor, organization, repository). Moreover, we remove events of type `WatchEvent`, drop duplicates and also remove missing values from `org.login`.

```
df = df[(df['type']!='WatchEvent')]
df = df[['actor.login', 'org.login', 'repo.name']]
```

```
df = df.drop_duplicates()
df = df[df['org.login'].notnull()]
df.head()
```

	actor.login	org.login	repo.name
0	pdgago	simbiotica	simbiotica/charicharts
3	dkorolev	KnowSheet	KnowSheet/Bricks
5	dmolsen	pattern-lab	pattern-lab/patternengine-php-twig
7	LordSputnik	BookBrainz	BookBrainz/bookbrainz.org
16	ScottNZ	OpenRA	OpenRA/OpenRA

Next, we want to focus on the most frequently occurring organisations, say the top 100. We will do so using the function `groupby()`.

```
org_actor_df = df.groupby(['org.login','actor.login']).agg('count').reset_index()
toporgs = org_actor_df.groupby(['org.login'])['org.login'].agg('count')
toporgs = toporgs.sort_values(ascending=False)[:100].keys().tolist()
toporgs[:10]
['mozilla', 'apache', 'facebook', 'google', 'angular', 'docker', 'elasticsearch', '
iojs', 'dotnet', 'owncloud']
```

The first line in the code above takes all possible pairs of `org.login` and `actor.login` and applies the operation `.agg('count')` that counts the frequency of each pair. The second line takes the frequency of each organisation and stores them into `toporgs`. Finally we sort the organisations in terms of frequency of appearance and print the top 10 of them.

The next step is almost taking us to our objective as we track the organisations where each user initiated an event and record all possible pairs between them. For example, if a user accessed three organisations, we will record the three possible pairs between them using the function `combinations()` from the Python package `itertools`. Note also that the line `for name, group in actor_df:` loops over all groups (having their name as well) contained in the `groupby` object `actor_df` that aggregates over each user.

```
from itertools import combinations


actor_df = df.groupby('actor.login')
dfs = pd.DataFrame(columns=['Actor', 'Org1', 'Org2'])
for name, group in actor_df:
    orgs = group['org.login'].unique()
    orgs = [val for val in orgs if val in toporgs]
    if len(orgs)>1:
        actor_edge = pd.DataFrame(data=list(combinations(orgs, 2)), columns=['Org1'
, 'Org2'])
        actor_edge['Actor'] = name
```

```
        dfs = pd.concat([dfs, actor_edge])
dfs.head()
```

	Actor	Org1	Org2
0	0xc0170	ARMmbed	mbedmicro
0	0xdeafcafe	projectkudu	aspnet
0	130s	ros-planning	ros-industrial-consortium
0	13abylon	arangodb	SleepyDragon
0	1Power	mongodb	rails

Each row in the `dfs` data frame corresponds to a user initiating an event to two different organisations (vertices), in other words it defines an edge to the network. We are almost ready to proceed; the final steps produce a data frame that allows for a more efficient way to construct the network by counting the number of times each edge (pair) occurs, also known as *weight* of the edge, and records it. We also print the top 10 edges in terms of their weight.

```
dfs = dfs.groupby(['Org1', 'Org2']).agg('count').reset_index().rename(columns={'Act
or': 'weight'})
dfs = dfs.sort_values(by='weight', ascending=False)
dfs.head()
```

	Org1	Org2	weight
187	odoo-dev	odoo	17
186	odoo	odoo-dev	9
133	iojs	joyent	5
105	google	GoogleCloudPlatform	5
190	openshift	GoogleCloudPlatform	4
10	GoogleCloudPlatform	openshift	4
61	chef	opscode-cookbooks	3
27	OCA	odoo	3
8	GoogleCloudPlatform	docker	3
56	babel	facebook	3

Network Visualisations

Now we proceed to the part of actually producing the network. First we load the **networkx** package.

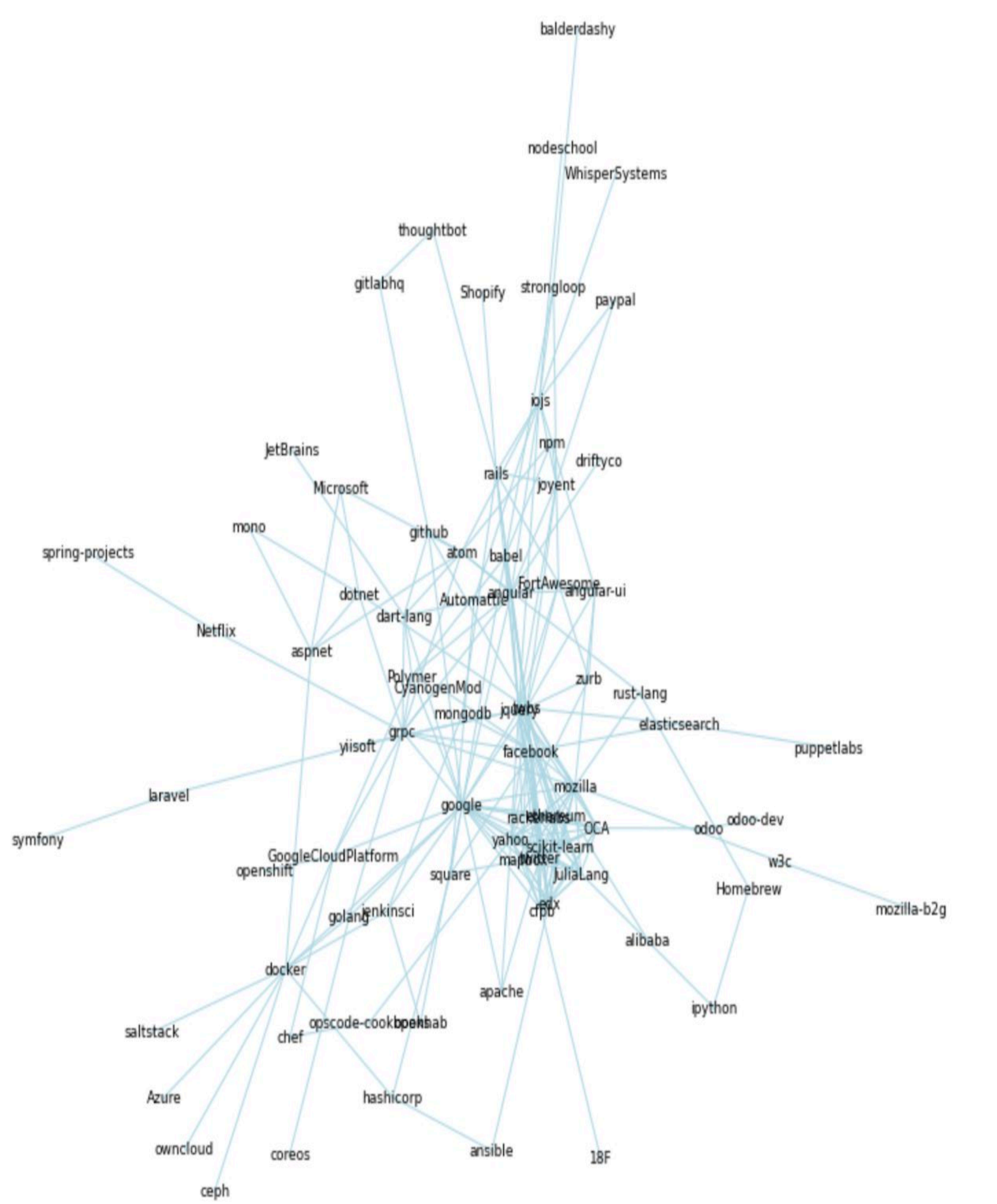
```
import networkx as nx
```

We can now create the network by adding edges one by one:

```
G = nx.Graph()


for index, row in dfs.iterrows():
    G.add_edge(row['Org1'], row['Org2'], weight=row['weight'])


# remove isolated vertices (if any)
remove = [node for node,degree in G.degree() if degree ==0]
G.remove_nodes_from(remove)
```

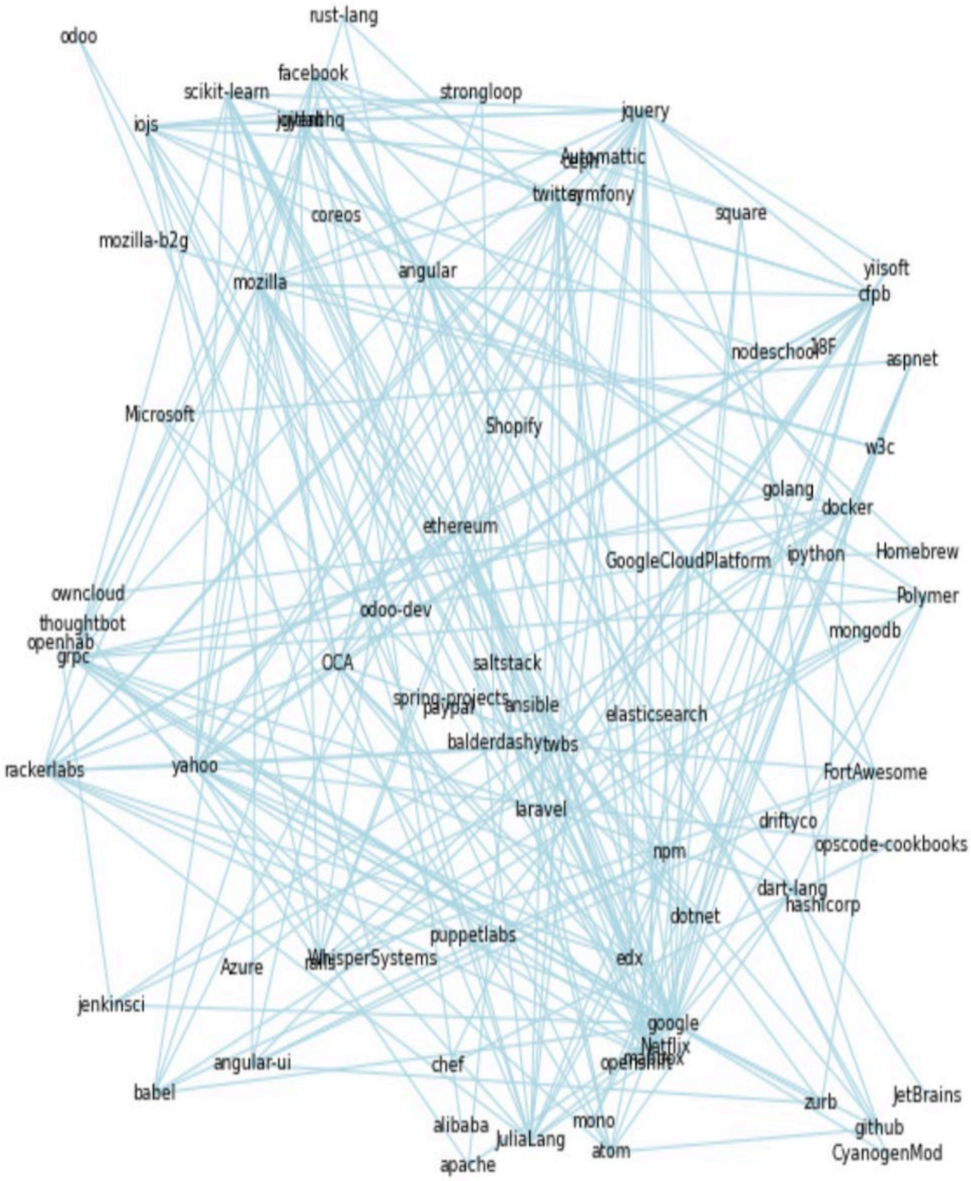
Finally, we can now produce a network with the **networkx** package. We start with the `spring` style after setting some network options regarding its size and colour:

```
#Setting size and colours
options = {
...     'node_color': 'lightblue',
...     'edge_color': 'lightblue',
...     'node_size': 1,
...     'width': 1,
...     'alpha': 1.0,
... }


# Producing the network
plt.subplots(figsize=(10,10))
pos=nx.spring_layout(G)
nx.draw(G,pos=pos,font_size=9,**options)
nx.draw_networkx_labels(G,pos=pos,font_size=9,**options)
plt.tight_layout()
plt.axis('off');
plt.show()
```

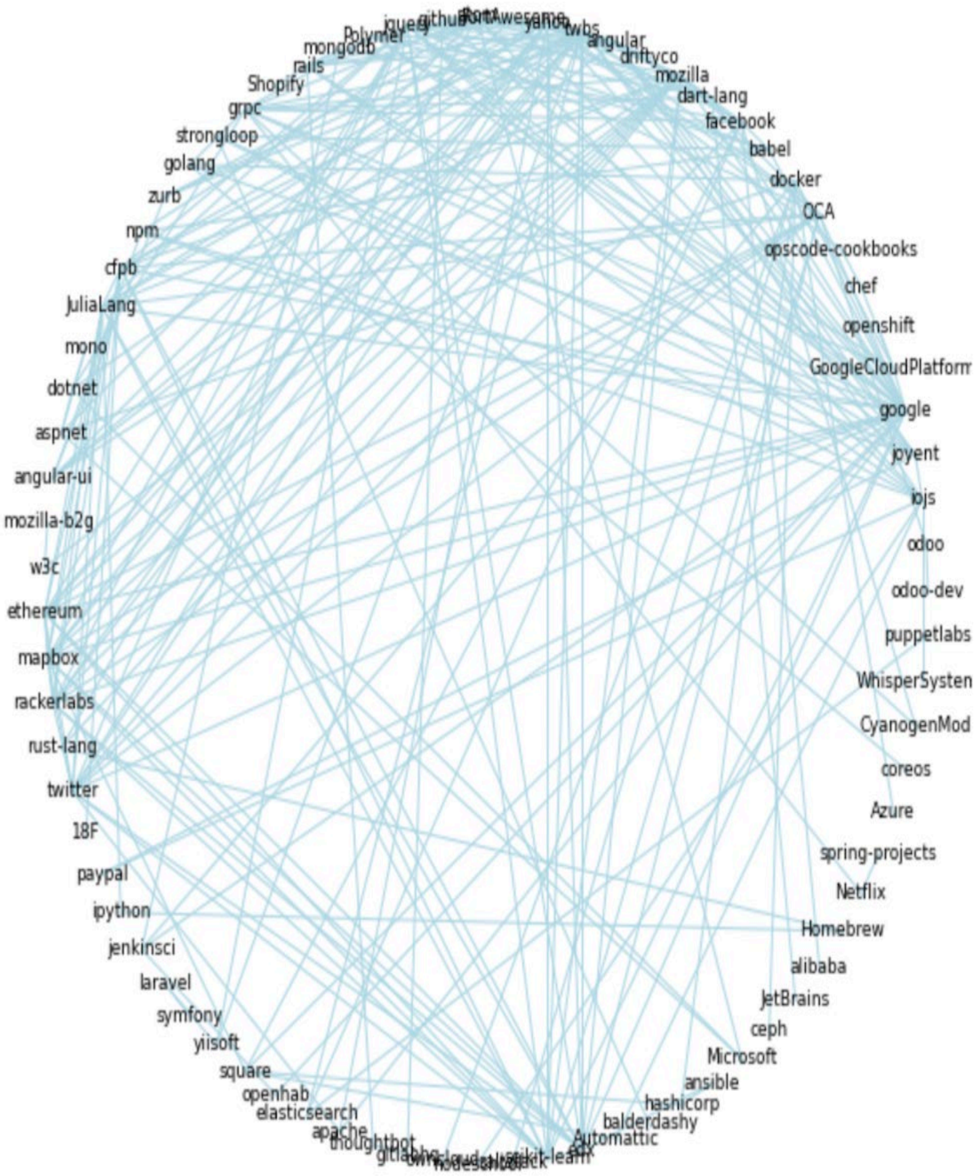



```
plt.subplots(figsize=(10,10))
pos=nx.random_layout(G)
nx.draw(G,pos=pos,font_size=9,**options)
nx.draw_networkx_labels(G,pos=pos,font_size=9,**options)
plt.tight_layout()
```

```
plt.axis('off');
plt.show()
```




```
plt.subplots(figsize=(10, 10))
pos = nx.circular_layout(G)
nx.draw(G, pos=pos, font_size=9, **options)
nx.draw_networkx_labels(G, pos=pos, font_size=9)
plt.tight_layout()
plt.axis('off')
plt.show()
```



More Network Layouts From **Graphviz**

The package **networkx** has more visualisation options (feel free to explore) and can also be combined with **Graphviz** (graph visualization software), which is a package of open-source tools initiated by AT&T Labs Research for drawing networks specified in DOT language scripts. More information can be found here. To install **Graphviz** follow the instructions here. To incorporate **Graphviz** into Python, the **Pygraphviz** interface to the **Graphviz** package can be used, see here for more information. To install **Pygraphviz**, type one of the following in the Anaconda prompt or terminal:

```
conda install -c conda-forge python-graphviz
```

or

```
pip install pygraphviz
```

Alternatively the **pydot** interface can be used (more information here).

```
pip install pydot
```

Networks in R

The most well-used package for manipulating network data in R is the **igraph** R package, which provides methods to handle and visualize network data, as well as methods for computing key network summaries (e.g. node centrality and connectivity metrics), methods for community detection and basic methods for the visualization of network data. **igraph** and its capabilities are also available in Python through the **python-igraph** module.

In addition to the visualization capabilities **igraph** provides, there is also a range of dedicated R packages for the visualization of networks. Marked examples are the **vizNetwork** R package that can be used to produce interactive network visualizations (see the **vizNetwork** pages for examples), the **ggnetwork** that provides **ggplot2** geometries for network data (see **ggnetwork**'s vignettes for examples), and the **networkD3** R package.

Katya Ognyanova's blog post provides a lot of resources about visualizing static and dynamic networks in R.

The **sna** R package also provides a range of advanced summaries and data-analytic tools for social network analysis.

Useful Links and Resources

- A tutorial on **networkx**
- The **graphviz** project
- Katya Ognyanova's blog post on visualizing static and dynamic networks in R

References

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.