5 Tips To Write Idiomatic Pandas Code

This tutorial covers 5 ways in which you can easily write pandorable or more idiomatic Pandas code.

Pandas is the *defacto* toolbox for Python data scientists to ease data analysis: you can use it, for example, before you start analyzing, to collect, explore, and format the data. Pandas makes these steps a breeze via its numerous I/O and handy data manipulation functions. What's more, as you will learn later in this tutorial, it also has some visualizations capabilities (through the use of `matplotlib`).

Even though the Pandas data structures are very powerful and flexible, they also come with some complexities which can make your analysis rather harder than easier, especially for beginners. In this tutorial, you'll focus on learning more about how you can save yourself a headache and write more idiomatic Pandas code. Besides learning how you can load, explore and clean the data, you'll learn more about the following five topics:

1. Indexing with the help of `loc` and `iloc`, and a short introduction to querying your DataFrame with `query()`;
2. Method Chaining, with the help of the `pipe()` function as an alternative to nested functions;
3. Memory Optimization, which you can achieve through setting data types;

4. `groupby` operation, in the naive and Pandas way; and
5. Visualization of your DataFrames with Matplotlib and Seaborn.

Of course, this tutorial is by no means exhaustive; The Pandas package is very rich and there are, without a doubt, other ways in which you might improve your Pandas code so that it becomes more idiomatic.

Let's not wait any longer and get started!

Load And Explore The Data

As always, your data analysis starts with loading and exploring your data. This section is just a warm-up in which you'll use the `read_csv()` function to load in world university rankings, after which you can already quickly explore your data with the functions `head()` and `describe()`. If you're interested in a full introduction to Pandas, consider taking DataCamp's three-part Pandas course, starting with the Pandas Foundations.

If you already know all of this, you can just skip this section and see how you can perform indexing with Pandas in a more idiomatic way.

Importing Packages

To start, let's import the usual Python packages for data science. As you might have guessed, Pandas is among these.

```
# If you're working with a notebook, don't forget to use
Matplotlib magic!
%matplotlib inline
```

```
# Import `pandas` under the alias `pd`
import pandas as pd

# Import `seaborn` under the alias `sns`
import seaborn as sns

# Set the Seaborn theme if desired
sns.set_style('darkgrid')
```

Moreover, you can also use the `watermark` package: it is a nice tool that makes your notebook more reproducible. Reproducible work is very important when collaborating with colleagues. Your future self will thank you for that as well! You can read more about reproducible data science here.

The Data

In order to start this tutorial, you will also need to fetch some "real" world data. For this tutorial, you will be working with a dataset from Kaggle. If you are unfamiliar with Kaggle, it is one of the biggest data science and machine learning communities. Moreover, it is a good place to learn more about data science once you know the basics.

The dataset that you'll be using is the World University Rankings.

As mentioned above, Pandas has many convenient method to read data from different data sources (you can learn more about it here). Since the data is in a `csv` format, we will be using the `.read_csv` method. But before starting, you need to download the data from Kaggle (or get it from the code repository under the `data` folder if you don't want to create an account).

Once you have fetched the data, you should have a folder `data` that contains the different csv files.

Next, you will be working with the Times Higher Education World University Rankings ranking (Times for short) and Academic Ranking of World Universities (Shanghai for short) ranking data. There is also a third ranking system in the `data` folder CWUR but it is much less known so you can safely ignore it for now.

```
# Import Times Higher Education World University Rankings data

times_df = pd.read_csv('data/timesData.csv', thousands=",")


# Import Academic Ranking of World Universities data
shanghai_df = pd.read_csv('data/shanghaiData.csv')
```

Quickly Inspecting The Data

As you have learned in the DataCamp's Exploratory Data Analysis tutorial, Pandas offers some methods to quickly inspect DataFrames, namely `.head()` to inspect the first `n` rows (`n` being 5 by default) and `.describe()` to get a quick statistical summary.

Refresh these functions by executing the following lines of code. The data has been loaded in already for you under the variables `times_df` and `shanghai_df`:

- script.py
- IPython Shell
-

```
# Return the first rows of `times_df`
```

```
times_df.____()

# Describe `times_df`

times_df.describe()

# Return the first rows of `shanghai_df`

shanghai_df.head()

# Describe `shanghai_df`

shanghai_df.________()
```

SolutionRun

Now that you have loaded in your data, you can start thinking about how you can manipulate the data with Pandas in an idiomatic way!

1. Indexing

First off, writing more idiomatic Pandas code means leveraging the power of indexing. Indexing means that you select subsets from your DataFrame.

When you're just starting out with Pandas, you might need some time to get used to how indexes work. In fact, if you have worked before with (Python) lists, you might be already familiar with the use of square brackets `[]` in combination with the colon `:` to select elements. This method works on DataFrames.

In addition, there are two other, more idiomatic ways to select a subset DataFrame. These two methods are namely `iloc` and `loc`:

- `loc` is label-based. This means that if you write `loc[2]`, you are looking for the values of your DataFrame that have an index labeled `2`.
- `iloc` is position-based. This means that if you write `iloc[2]`, you are looking for the values of your DataFrame that are at index `2`.

Try this out in the DataCamp Light chunk below by typing `times_df.loc[2]` and `times_df.iloc[2]` in the IPython console to see the difference. Next, try solving the exercise below with the help of `loc`, `iloc` and the traditional square brackets!

- script.py
- IPython Shell
-

```
# Retrieve the total score of the first row

print(times_df.loc[0, '___________'])

# Retrieve rows 0 and 1

print(times_df[_:_])

# Retrieve the values at columns and rows 1-3

print(times_df.iloc[1:4,1:4])

# Retrieve the column `total_score`

print(times_df['__________'])
```

SolutionRun

Of course, your exploration can go a lot further than just simply subsetting or selecting from rows and columns of your data. Things can get very interesting when you combine the `loc` and iloc with, for example, boolean arrays.

You see that this goes somewhat beyond selecting data on the basis of columns or indexing; It's somewhat like querying your data!

- script.py
- IPython Shell
-

```
# Are the last entries after 2006?

print(shanghai_df.loc[:-10, 'year'] > 2006)

# Was the alumni count higher than 90 for the first

 ten universities?

print(shanghai_df.loc[0:11, 'alumni'] > 90)
```

SolutionRun

`loc` and `iloc` are great to index your data, but be careful when you start using two sets of square brackets after each other! Pandas could return a copy of a view and in such cases, you wouldn't know with what you're still working! So think twice if you're using two sets of square brackets in combination with `loc` or `iloc`.

Note also that, to query your data, Pandas also provides you with a `query()` function which you can use to quickly inspect your data (available since version 0.13). For example, you can compose a query to see which universities have a `total_score` that is slightly below 50:

- script.py
- IPython Shell
-

```
# Query `shanghai_df` for universities with total
score between 40 and 50
averge_schools = shanghai_df.query('total_score >
__ and total_score < __')
# Print the result
print(average_schools)
```

SolutionRun

Of course, there are many more possibilities when it comes to querying your data! Why not try out to come up with some queries and test them in the IPython console of the above DataCamp Light chunk?

If you're short on inspiration, try queries that return the universities with a first national rank and universities with a first world rank. Additionally, you can query the alumni numbers of the universities.

Are you lost on how to compose the queries? Check out some of the examples below:

```
shanghai_df.query("national_rank == 1 and world_rank == 1")
shanghai_df.query("alumni < 20")
```

2. Method Chaining

Method chaining is something typical of Pandas, but when you're just starting out with the Python data manipulation package, it might not be straightforward how this exactly works. It's basically calling methods on an object one after another.

In this section, you will go deeper into method chaining by creating data pipelines with `pipe()`.

- script.py
- IPython Shell
-

```
# Extract info
def extract_info(input_df, name):
    df = input_df.copy()
    info_df = pd.DataFrame({'nb_rows': df.shape[0],
'nb_cols': df.shape[1], 'name': name}, index=range
(1))
    return info_df
# Gather all info
all_info = pd.concat([times_df.pipe(extract_info,
'times'), shanghai_df.pipe(extract_info, 'shanghai'
)])
```

SolutionRun

As you have already noticed, the datasets are quite different. Here are some of the differences:

- The Times dataset has 14 columns whereas the Shanghai one has 11

- The Times dataset has 2603 rows whereas the Shanghai one has 4897
- One major missing column from the Shanghai dataset is the country of the University. You will get this information from the Times dataset later if needed.

Thus, in order for us to use both DataFrames, we will need to only select the common columns. Hopefully, these common columns have similar content.

- script.py
- IPython Shell
-

```
common_columns = set(shanghai_df.columns) & set
(times_df.columns)
# Return `common_columns`
print(common_columns)
```

SolutionRun

The common columns are thus:

- `total_score`: Score used to determine rank. As mentioned in the Kaggle description page, these contain range ranks (i.e. between two values) and equal ranks (there is an `=` sign in front)
- `university_name`: University name
- `world_rank`: Rank of the university

- `year`: Year for which the ranking is done

In order to concatenate both data, we need to do some cleaning first. Then apply the complete pipeline now to clean the data:

- script.py
- IPython Shell
-

```
# Clean up the `world_rank`
def clean_world_rank(input_df):
    df = input_df.copy()
    df.world_rank = df.world_rank.str.split('-'
).str[0].str.split('=').str[0]
    return df

# Assign the common years of `shanghai_df` and
`times_df` to `common_years`
common_years = set(shanghai_df.year) & set
(times_df.year)
# Print `common_years`
print(common_years)
# Filter years
def filter_year(input_df, years):
    df = input_df.copy()
    return df.query('year in {}'.format(list
(years)))
# Clean `times_df` and `shanghai_df`
cleaned_times_df = (times_df.loc[:,
common_columns]
                             .pipe(filter_year,
common_years)
```

```
                             .pipe
(clean_world_rank)
                             .assign(name='times'
))
cleaned_shanghai_df = (shanghai_df.loc[:,
common_columns]
                                   .pipe
(filter_year, common_years)
                                   .pipe
(clean_world_rank)
                                   .assign(name
='shanghai'))
```

SolutionRun

As you have probably noticed, I use the `.pipe` method a lot in this tutorial. If you come from the R world, you should probably be (somehow) familiar with it.

It is a handy new Pandas (since 0.16.2 version) method that allows you to chain operations and thus eliminate the need for intermediate DataFrames. Your code becomes more readable.

In fact, without this operator, instead of writing `df.pipe(f).pipe(g).pipe(h)` you would write: `h(g(f(df)))`. This becomes harder to follow once the number of nested functions grows large.

To learn more about the subject, I highly recommend reading the following blog post.

Now that both DataFrames are cleaned, we can concatenate them into a single DataFrame. Also, don't forget to clean the missing data:

- script.py
- IPython Shell
-

```
# Compose `ranking_df` with `cleaned_times_df` and
`cleaned_shanghai_df`
ranking_df = pd.concat([cleaned_times_df,
cleaned_shanghai_df])
# Calculate the percentage of missing data
missing_data = 100 * pd.isnull(ranking_df
.total_score).sum() / len(ranking_df)
# Drop the `total_score` column of `ranking_df`
ranking_df = ranking_df.drop('total_score', axis=1)
```

SolutionRun

Since you have around 38% of data missing from the `total_score` column, it's better to drop this column with the `.drop` method.

3. Memory Optimization

A third way of making your Pandas code more "idiomatic" -which is in this case more equal to "performant"- is by optimizing the memory usage. Especially when you start making data pipelines by using method chaining, you'll see that memory can start to play a big part in how fast your pipelines can run. You'll see more on all of this here.

This is probably overkill for this small dataset but it is often a good practice to cast some columns to specific types to optimize the memory usage. To start, let's find how much memory is allocated for `ranking_df` using the `.info` method.

- script.py
- IPython Shell
-

```
# Print the memory usage of `ranking_df`

ranking_df.____()

# Print the deep memory usage of `ranking_df`

ranking_df.info(memory_usage="deep")
```

SolutionRun

The result of the first line of code tells us that the data takes 144 KB. However, upon inspecting the arguments of the `.info` method, one finds out that there is an optional one namely `memory_usage` that is set to None by default. What happens when you set the argument `memory_usage` to `deep`?

The new memory footprint is 6 -5.6 to be more precise- times larger than your initial answer.

What happened here?

The difference in the memory estimation stems from the fact that without the “deep” flag turned on, Pandas won’t estimate memory consumption for the `object` dtype. This (Python) type of data takes more space than other `numpy` optimized `dtypes`. This is why it is recommended to cast `object` types to more appropriate ones (let’s say `category` when dealing with categorical data).

Let’s do it!

```
def memory_change(input_df, column, dtype):
    df = input_df.copy()
    old = round(df[column].memory_usage(deep=True) / 1024, 2) # In KB
    new = round(df[column].astype(dtype).memory_usage(deep=True) / 1024,
2)# In KB
    change = round(100 * (old - new) / (old), 2)
    report = ("The inital memory footprint for {column} is:
{old}KB.\n"
              "The casted {column} now takes: {new}KB.\n"
              "A change of {change} %.").format(**locals())
    return report


print(memory_change(ranking_df,'world_rank', 'int16'))
print(memory_change(ranking_df,'university_name', 'category'))
print(memory_change(ranking_df,'name', 'category'))
```

Now that you know that, by changing the `world_rank` to `int16`, for example, your memory will be used more efficiently, it’s time to actually apply these changes. You can use the `astype()` function to do this. End by double-checking what your memory usage looks like now:

- script.py
- IPython Shell
-

```
# Cast `world_rank` as type `int16`
ranking_df.world_rank = ranking_df.world_rank
._______('int16')
# Cast `unversity_name` as type `category`
ranking_df.university_name = ranking_df
.university_name.astype('category')
# Cast `name` as type `category`
ranking_df.name = ranking_df.____.astype
('category')
# Double check the memory usage after type
casting
ranking_df.info(memory_usage='deep')
```

SolutionRun

That’s much better. You can optimize even further by casting the `year` column down to `int32`.

4. GroupBy

Now that we have a well-formed data set (i.e. it is in a tidy state) with an optimized memory footprint, data analysis can start.

Now there are some questions that you might be interested in answering:

- What are the top 5 universities given by each ranking system over the years?
- How different are these rankings for each year?

Let's find out by tackling the first question by making use of the more idiomatic Pandas pointers that you have already seen in the previous sections!

Before you start, notice that 'Massachusetts Institute of Technology (MIT)' and 'Massachusetts Institute of Technology' are two different records of the same university. Thus, you change the first name to the latter.

Note that in cases like these, the `loc` function, which you saw earlier in this tutorial, comes in particularly handy!

- script.py
- IPython Shell
-

```
# Query for the rows with university name
'Massachusetts Institute of Technology (MIT)'
print(ranking_df._____("university_name ==
'Massachusetts Institute of Technology (MIT)'"))
# Localize the rows with the MIT university name
and replace value
ranking_df.loc[lambda df: df.university_name ==
'Massachusetts Institute of Technology (MIT)',
'university_name'] = 'Massachusetts Institute of
Technology'
# Print `ranking_df`
```

```
print(__________)
```

SolutionRun

To find the 5 (more generally `n`) top universities over the years, for each ranking system, here is how to do it in pseudo-code:

- For each year (in the `year` column) and for each ranking system (in the `name` column):
- Select the subset of the data for this given year and the given ranking system
- Select the 5 top universities and store them in a list
- Store the result in a dictionary with (year, name) as key and the list of the universities (in descending order) as the value

Let's apply this.

- script.py
- IPython Shell
-

```
# Load in `itertools`
import itertools
# Initialize `ranking`
ranking = {}
for year, name in itertools.product(common_years,
["times", "shanghai"]):
    s = (ranking_df.loc[lambda df: ((df.year ==
year) & (df.name == name)
                                    & (df
.world_rank.isin(range(1,6)))), :]
                   .sort_values('world_rank',
ascending=False)
                   .university_name)
    ranking[(year, name)] = list(s)

# Print `ranking`
print(ranking)
```

Run

Now that we have this ranking dictionary, let's find out how much (in percentage) both ranking methods differ over the years: the two are 100% set-similar if the selected 5-top universities are the same even though they aren't ranked the same.

- script.py
- IPython Shell
-

```
# Import `defaultdict`
from collections import defaultdict
# Initialize `compare`
compare = defaultdict(list)
# Initialize `exact_similarity` and
`set_similarity`
exact_similarity = {}
set_similarity = {}
for (year, method), universities in ranking.items
```

```
():
    compare[year].append(universities)


for year, ranks in compare.items():
    set_similarity[year] = 100 * len(set(ranks[0])
& set(ranks[1])) / 5.0
# Print `set_similarity`
print(set_similarity)
```

Run

As you have noticed, this was tedious. Is there a better, more idiomatic Pandas way?

Of course, there is! Firstly, filter the ranking DataFrame to only keep the 5 top universities for each year and ranking method. Also check out the result with the help of the `head()` function:

- script.py
- IPython Shell
-

```
# Construct a DataFrame with the top 5 universities
top_5_df = ranking_df.loc[lambda df: df.world_rank
.isin(range(1, 6)), :]
# Print the first rows of `top_5_df`
top_5_df.____()
```

SolutionRun

Now that you have the correct DataFrame to work with, let’s use the `.groupby` method to compare both ranking methods using the set similarity defined above.

- script.py
- IPython Shell

```
# Compute the similarity
def compute_set_similarity(s):
    pivoted = s.pivot(values='world_rank', columns
='name', index='university_name').dropna()
    set_simlarity = 100 * len((set
(pivoted['shanghai'].index) & set(pivoted['times']
.index))) / 5
    return set_simlarity
# Group `top_5_df` by `year`
grouped_df = top_5_df._______('year')
# Use `compute_set_similarity` to construct a
DataFrame
setsimilarity_df = pd.DataFrame({'set_similarity':
grouped_df.apply(compute_set_similarity)}
).reset_index()
# Print the first rows of `setsimilarity_df`
setsimilarity_df.____()
```

SolutionRun

You get the same results as above with less hassle.

5. Visualization

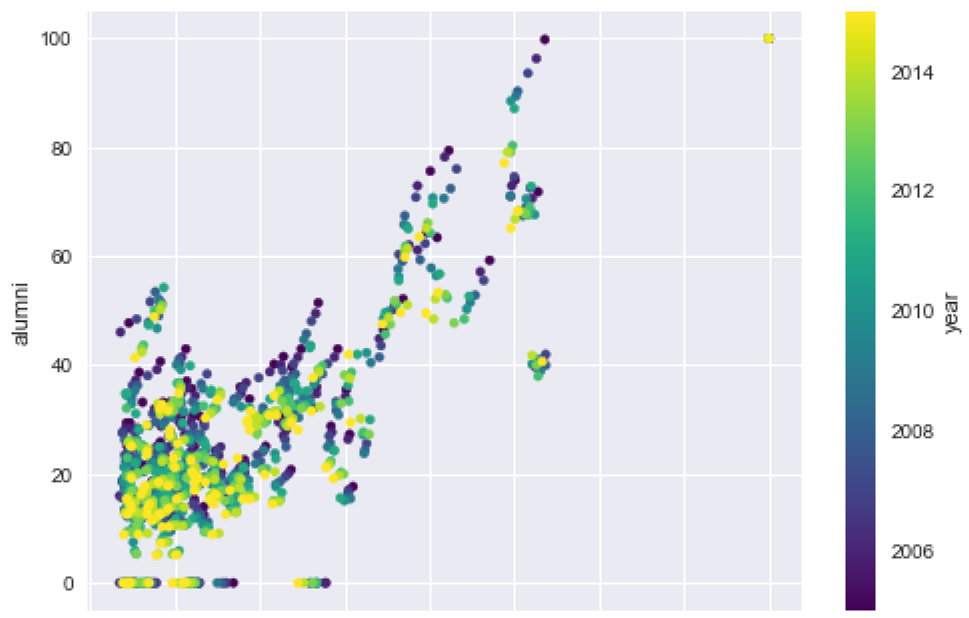
Now that you have gone through some initial analysis of your data sets, it’s also time to explore your data visually. More idiomatic Pandas code also means that you make use of Pandas’ plotting integration with the Matplotlib package. Because of this, you’ll make great plots in no time!

For this section, you’ll go back to the `times_df` and `shanghai_df` datasets to make some basic visualizations with Matplotlib and Seaborn.

- script.py
- IPython Shell
-

```
# Plot a scatterplot with `total_score` and `alumni`
shanghai_df.plot.scatter('total_score', 'alumni', c='year', colormap='viridis')
plt.show()
```

Run



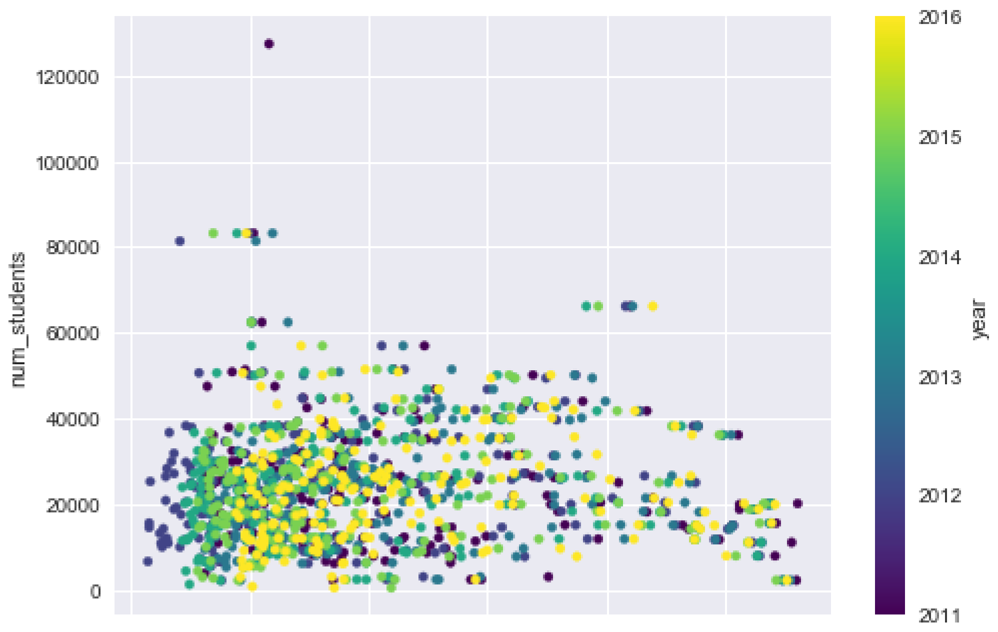
The above plot immediately makes clear that there are some 0 values for the `alumni` column. This is something to definitley take into account when you're exploring your data: with the help of data profiling, you'll be able to see these 0 values and be able to handle them. If you'd like to know more about this, check out DataCamp's Data Profiling Tutorial.

Or, in case you have values that don't have a place in your data set, you can simply replace them by `NaN` and then remove all the `NaN` values of the columns which would make plotting the data more difficult. Note that this is just a quick way of adjusting your data and that you probably need a more elaborate data profiling step to make sure that your data quality improves because you still see 0 values for the `num_students` column in the plot below:

- script.py
- IPython Shell
-

```
# Replace `-` entries with NaN values
times_df['total_score']= times_df['total_score']
.replace("-", "NaN").astype('float')
# Drop all rows with NaN values for `num_students`
times_df = times_df.dropna(subset=['num_students']
, how='all')
# Cast the remaining rows with `num_students` as
int
times_df['num_students'] =
times_df['num_students'].astype('int')
# Plot a scatterplot with `total_score` and
`num_students`
times_df.plot.scatter('total_score',
'num_students', c='year', colormap='viridis')
plt.show()
```

Run



Of course, you don't need to limit yourself to Matplotlib to visualize your data. There are also other libraries out there that allow you to quickly visualize your data, such as Seaborn.

The Seaborn plotting tool is mainly used to create statistical plots that are visually appealing (as stated in the official website). In fact, prior to version 2.0 of Matplotlib, it was a hassle (though possible) to create beautiful plots.

The combination of Pandas with Seaborn also allows you to quickly iterate over your data by means of visualizations.

Let's explore some examples!

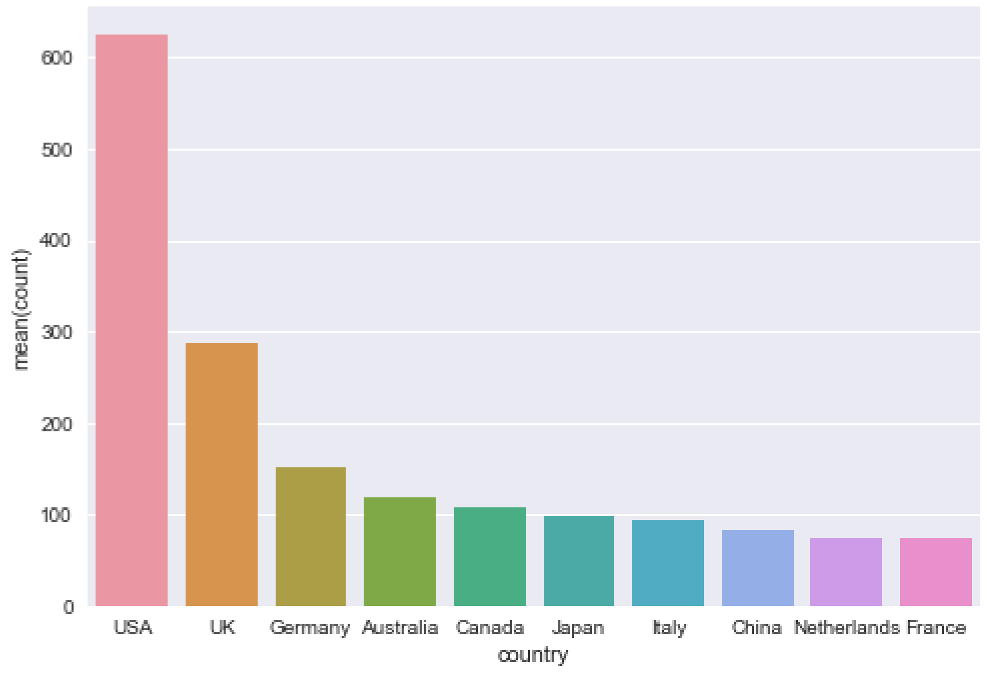
Take a look at the following plot, where you count the number of times a specific country appears in your data, you take the first ten observations and you make sure that you sort the counts from highest to lowest value when you plot the barplot:

- script.py
- IPython Shell
-

```
# Abbreviate country names of United States and
```

```
United Kingdom

times_df['country'] = times_df['country'].replace

("United States of America", "USA").replace

("United Kingdom", "UK")

# Count the frequency of countries

count = times_df['country'].value_counts()[:10]

# Convert the top 10 countries to a DataFrame

df = count.to_frame()

# Reset the index

df.reset_index(level=0, inplace=True)

# Rename the columns

df.columns = ['country', 'count']

# Plot a barplot with `country` and `count`

sns.barplot(x='country', y='count', data=df)

sns.despine()

plt.show()
```

Run



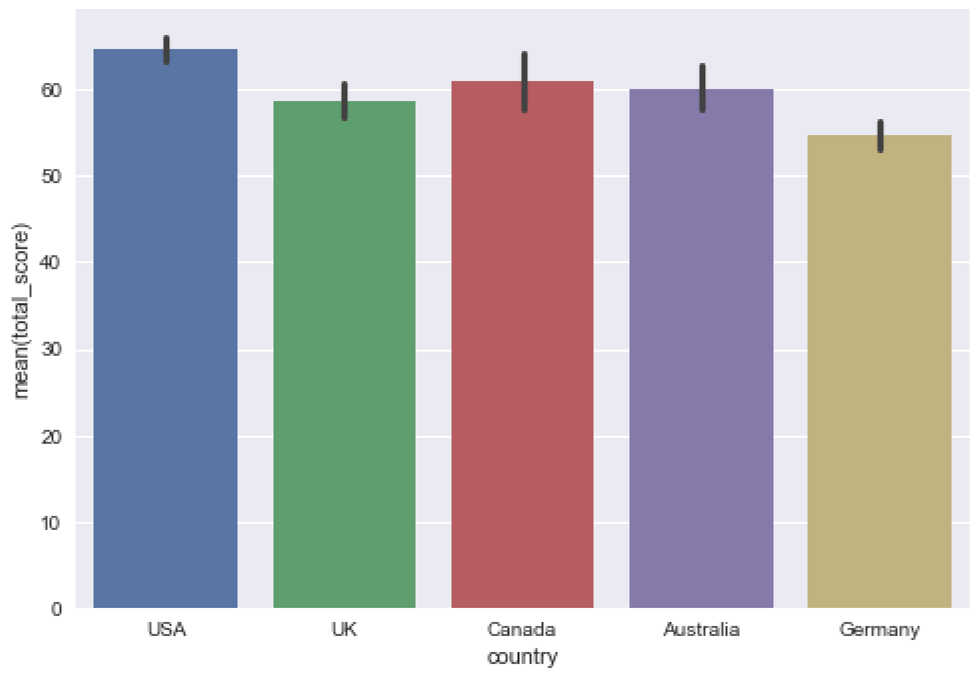
Then, you can go on and plot these countries and their mean scores:

- script.py
- IPython Shell
-

```
# Barplot with `country` and `total_score`
sns.barplot(x='country', y='total_score', data
=times_df_filtered)
```

```
sns.despine()

plt.show()
```

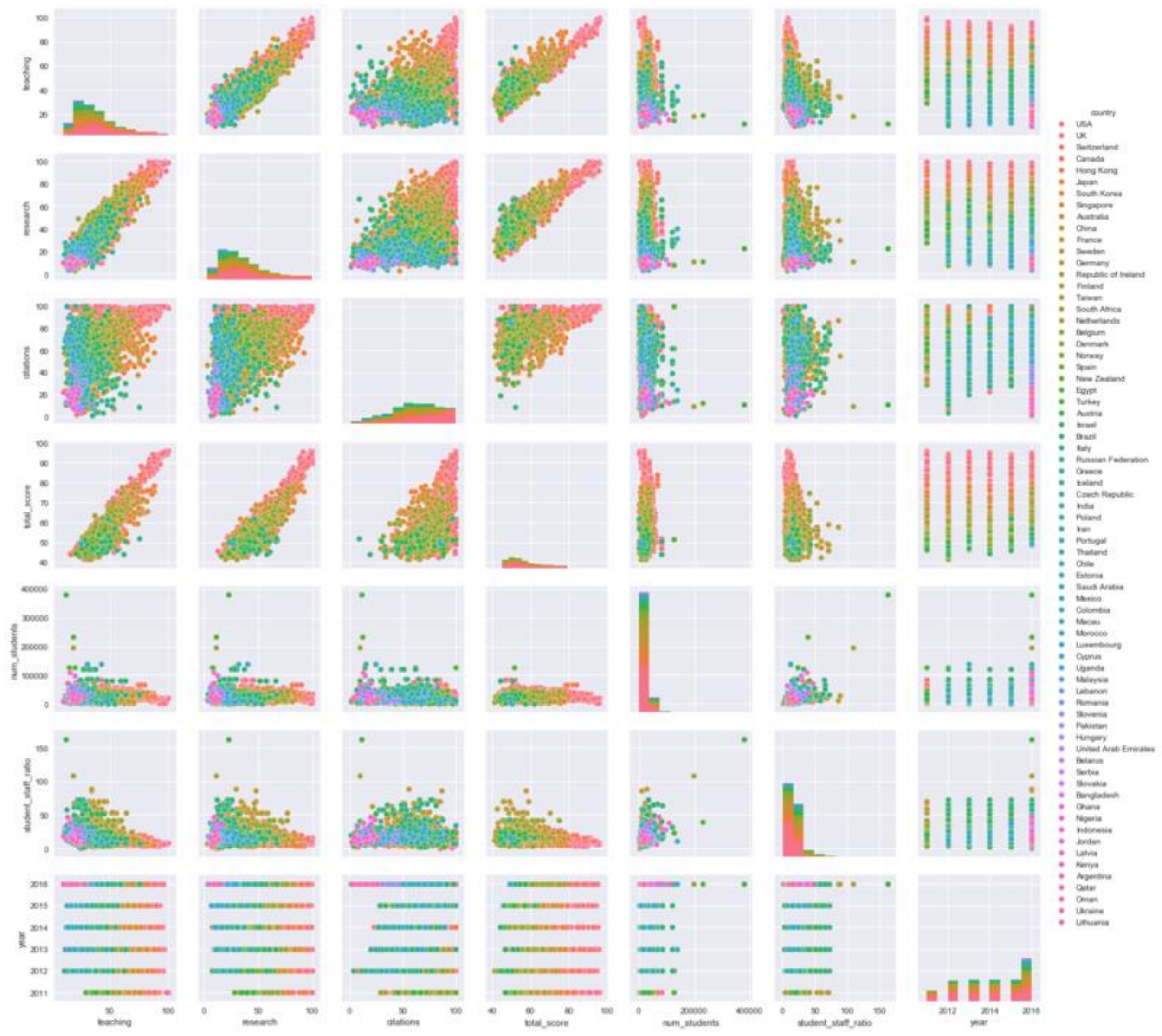
Run



Your plots can also get pretty complex. Look at the following pairplot, which is designed to show you the pairwise relationships in a dataset more clearly:

```
import numpy as np


np.seterr(invalid='ignore')


sns.pairplot(times_df, hue='country')
```

```
plt.show()
```

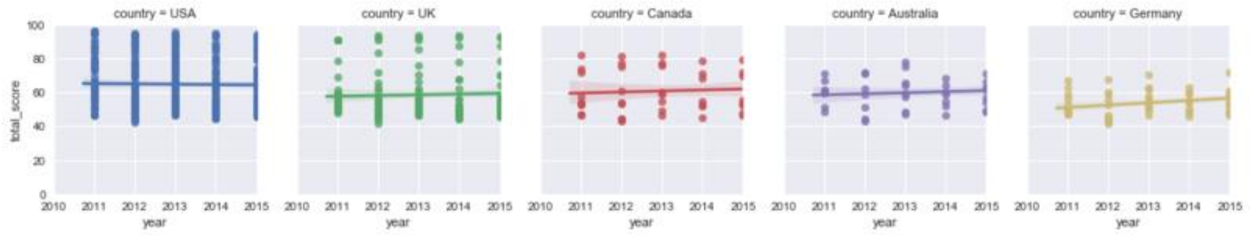


Likewise, you can check out this combination of `FacetGrid` and `regplot`, which gives you some insights on the evolution of the `total_score` over the years for the top 5 countries:

- script.py
- IPython Shell
-

```
# Filter out rows with NaN entry for `total_score`
times_df_filtered = times_df_filtered.dropna(subset
=['total_score'], how='all')
g = sns.FacetGrid(times_df_filtered, col='country',
hue='country')
g.map(sns.regplot, 'year', 'total_score').set(xlim
=(2010, 2015), ylim=(0,100))
g.fig.subplots_adjust(wspace=.2)
```

Run

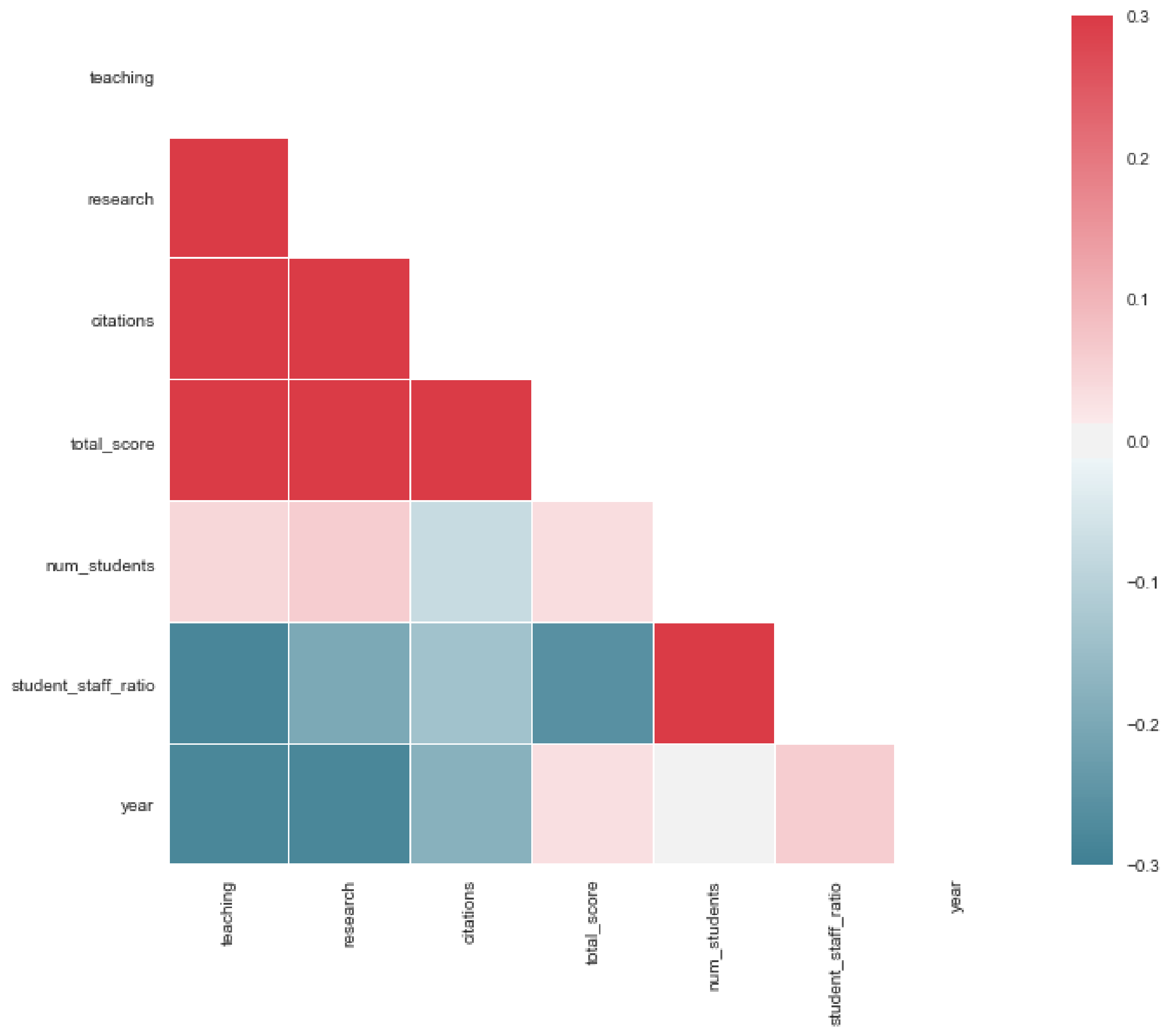


Lastly, also a correlation plot might come in handy when you're exploring your data. You can use the `heatmap()` function from Seaborn to get this done:

- script.py
- IPython Shell
-

```
import matplotlib.pyplot as plt

sns.set(style="white")

# Compute the correlation matrix

corr = times_df.corr()

# Generate a mask for the upper triangle

mask = np.zeros_like(corr, dtype=np.bool)

mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure

f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap

cmap = sns.diverging_palette(220, 10, as_cmap=True
)

# Draw the heatmap with the mask and correct
aspect ratio

sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3,

            square=True, linewidths=.5, ax=ax)


plt.show()
```

Run



You can keep going on and on with these visualizations, so maybe it's best to stop for now :).

To go beyond

That's it for this time, I hope you have enjoyed learning some intermediate techniques working with Pandas and more specifically how to write more idiomatic Pandas code.

If you have enjoyed this blog post and want to dig deeper about performance in Pandas, I highly recommend the following additional material:

- Tom Augspurger's Modern Pandas tutorial,
- This Pandas From The Inside talk by Stephen Simmons, and
- If you want to learn more about how to implement the groupby operation using other Python tools, I highly recommend reading the following blog post.