DATA SCIENCE FROM SCRATCH

Part 1

Can I to I Can

First ever book completely written in jupyter notebook

JUNAID QAZI, PhD

At 50% reduced price

Unedited copy

Preface:

Dear learners,

I am pleased to mention that this book is based on lecture notes from my ***bestselling Bootcamp on Data Science and Machine Learning using Python***. It was never easy to get enough time, and I am obliged to those, who kept on sending requests to compile my notes in the form of a book.
Thank you for making it happen ☺.

My fellows, I decided to split my notes into two books:

1: **Data Science from Scratch (Part 1)**
(1 to 400 pages)
Covers all what you need for advance business intelligence. Everything from scratch to the point where you can learn to implement data preprocess pipeline and presented insights. Most of the time, you only need this part for advance analytics. The idea here is to give you the skills so that you can quickly start looking for projects on freelancing platforms, there is a lot that you can do just after finishing the part 1.

2: **Data Science from Scratch (Part 2)**
(401 to ~ 900 pages)
This part will entirely focus on business machine learning, which will be available soon. It will cover theory and complete pipeline along with hands-on exercise for widely used machine learning algorithms. I will keep you posted!

This is an unedited copy and there might be some typo errors. However, all the codes are working and using latest versions of stable libraries till my last check. Keeping it with me and waiting for the final edits could take another year, which does not make much sense. So, I decided to bring the book to you with half of its original price. Whenever, the final copy will be available, you will get with no extra price, so a chance to get final copy with half price.

I am pleased to mention here, to my knowledge, this is the ***first ever book completely written in jupyter notebook*** so that, you feel real time working environment that is preferred by data science community.

Wishing you good luck for your future plans!

Dr. Junaid S Qazi, PhD
Science Academy Inc.
https://www.linkedin.com/in/jqazi/

Dedicated to:
All the Aspiring Data Scientists!

Installation:

Dear learners,

In case, you have not installed python and required libraries on your computer, please watch my video lectures to understand the whole process. It is ok to install latest version of anaconda and other libraries, consider these video lectures to understand the process only. If you find any difficulty, the best way is to google your concern/error and you will get the answer. You are most welcome to write me as well, and I would be pleased to help.

- Setup lecture 1: Jupyter and environment setup
- Setup lecture 2: Jupyter and environment setup
- Alternative method: Another way to get your system ready!

This book uses `Python 3.7` (more specifically 3.7.5).
Following stable versions of these libraries are used in this book, and it is recommended to use the same.

- `numpy==1.18.1`
- `pandas==0.25.3`
- `matplotlib==3.1.2`
- `seaborn==0.9.0`
- `plotly==3.10.0`
- `cufflinks==0.16`

NOTE: *All the codes are tested, and they are fully functional for the above given versions. If you are using different versions, you may see some difference. These libraries/softwares get regular updates and version number should not create any confusion. Our ultimate goal is to learn skills that are making the difference, you can always explore different versions later on.* ***Please write us if you have any questions.***

Good luck!

P.S.
You don't need to specify version number if you want to get the latest one, e.g.
`pip install pandas` will install the latest available version of pandas whereas
`pip install pandas=<version number>` will install the specified version of certain library, pandas in this case!

Note: You can also use ***`conda`*** *instead of* ***`pip`****.*

Jupyter notebooks:

Here is the link where you can access all the jupyter notebooks. Solutions and datasets are also included in the relevant folder.
(https://drive.google.com/open?id=1b1PYaTxIf5w_4bDxfCrriYgDVTnKSTxD)
It's good to have these notebooks, but not good to look at them unless you really need to. Follow the book and write the code, you will feel the difference in your learning curve.

Questions?

Please write me, and I would be happy to help!

Wait, another link you may want to explore
https://github.com/junaidqazi/DataSets_Practice_ScienceAcademy
On this link, you will find all the datasets that are used in this book along with several others. I always try to keep a copy…good to have it!

One last thing*, as I have already mentioned, this is an un-edited copy, if something is not working, send me a message, and* ***you will get you name at the end of the final version of this book. If you want, I would be happy to add a link to your LinkedIn profile as well!***

Frankly speaking, I was always looking for more interactive and lecture styled book with coding and explanation during my learning so that it feels like a workplace. I never wanted to write a typical book to learn data science, hence, final version of this book will definitely remain in the form of lectures in jupyter notebook, which is the true essence of this book.

Table of Contents

range():

- Rather than creating a sequence (specially in for loops), we can use `range()`, which is a generator of numerical values.
- **With one argument**, `range` generates a list of integers from zero up to, but not including, the argument's value.
- **With two arguments**, the first is taken as the lower bound (start).
- We can give a **third argument as a step**, which is optional. If the third argument is provided, Python adds the step to each successive integer in result (default value of step is "+1").

Some working examples of range are given in the below code cels.

[7]:
```
# This (the code below) will give the range object, which is iterable.
range(5)
```

[7]:
```
range(0, 5)
```

[8]:
```
# With method "list", we can convert the range object into a list
list(range(5))
```

[8]:
```
[0, 1, 2, 3, 4]
```

[9]:
```
# Use of range(), with single argument, in a loop
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

[10]:
```
# Use of range(), with two argument, in a loop
for i in range(3,5):
    print(i)
```

```
3
4
```

[11]:
```
# Use of range(), with three argument, in a loop
for i in range(1,10,2):
    print(i)
```

```
1
3
5
7
9
```

6 List Comprehension

Do you remember the basic mathematics?

- $\{x^2 \mid x \in \mathbb{N}\}$ gives squares of natural numbers.
- $\{x^2 : x \text{ in } \{0 \ldots 9\}\}$ gives squares of numbers with in the provided set, $\{0 \ldots 9\}$.

List comprehension implements such well-known notations for sets. It is an elegant and concise way to define and create lists in Python, and off-course, it saves typing as well!

Syntax for the list comprehension is:
`"statement/expression"` followed by a `"for clause"` with in `"square brackets"`.

Let's learn with example while comparing `for loop` and `list comprehension`.

[12]:
```python
# We have a list 'x'
x = [2,3,4,5]
```

What if we want to create a new list that contains squares of all the elements in `list x`? We can do this using a for loop as given in the code cell below *(please read the comments)*.

[13]:
```python
out = [] # empty list for squares
for num in x: # loop test
    out.append(num**2) # taking squeares and appending them to the empty list "out"
print(out) # using print to get the output
```

```
[4, 9, 16, 25]
```

Ok, we have accomplished the task to compute squares using `for` loop, however, the above task can be elegantly implemented using a list comprehension in a one line of code. **Simply take `for statement` and put it after what you want in result!**

[14]:
```python
# So, this is going to be out first list comprehension to compute squares of all␣
↪the elements in a given list!
[num**2 for num in x]
```

[14]:
```
[4, 9, 16, 25]
```

[15]:
```python
# Another example using string -- notice the white space!
[letters for letters in 'Hello World']
```

[15]:
```
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

[16]:
```python
# one more example using range()
[numbers**2 for numbers in range(2,10)]
```

[16]:
```
[4, 9, 16, 25, 36, 49, 64, 81]
```

7 Functions:

You may have come across `functions` in other computer languages, where they may have been called `subroutines` or `procedures`.

Functions serve two primary development roles:

- Maximizing code reuse and minimizing redundancy.
- Procedural decomposition by splitting systems into pieces that have well-defined roles.

Functions reduce our future work radically and if the operation must be changed later, we only have to update one copy in the function, not many scattered copies throughout the code.

In Python, `"def"` creates a function object and assigns it to a name. Let's start with a simple example of our first function!

[17]:
```python
# A simplest example of a function is:
def function_name(pram1):
    """
    Body: Statements to execute
    """
    print(pram1) # this will print the pram1 only
```

[23]:
```
# function returns the result
def my_func(num1, num2):
    num = num1*num2
    return num
# we can write "return num1**num2" in a single line as well
```

[24]:
```
# Function call
my_func(2,3)
```

[24]:
```
6
```

We can get the results from our function in a variable and then print that variable, **let's try!**

[25]:
```
out = my_func(2,3)
print(out)
```

```
6
```

Functions can have documentation strings (docstring) enclosed b/w `"""doc"""`. ***I want to say, a function should have a document string, they are very useful.***
Let's create a docstring for our example function.

[26]:
```
def my_func(num1, num2):
    """
    These docstrings are very useful
    Jupyter has a great feature for these strings
    write function name, and click shift + tab
    my_func -- shift + tab
    You will see these doc string
    """
    return num1**3
```

[27]:
```
# mu_func - then press shift + tab to see the doc string
my_func
```

[27]:
```
<function __main__.my_func(num1, num2)>
```

Again, DocStrings are very useful, it is always encouraged to write a proper documentation of your custom function.

You can find the documentation of all the built-in functions very useful. If we type `range` and click `<shift+tab>` (in jupyter notebook environment), we will see the documentation of `range`, we don't need to memorize this all! **We will be using this feature lots of time in this course!**

8 Lambda expression

Now we know the basics about function, we can move on and learn `"lambda expression"`.

- Lambda expression is a way to create small **anonymous** functions, i.e. **functions without a name**.
- These **throw-away functions** are created where we need them.
- Lambda expressions are mainly used in combination with the `filter()` and `map()` built-in functions.

General syntax of a lambda expression is: `"lambda argument_list:  expression"`

The **argument_list** consists of a comma separated list of arguments and **expression** is an arithmetic expression using these arguments.

We can assign "`lambda expression`" to a variable to give it a name as well.

Lets write a function, which return square of a number and then re-write that function to a lambda expression. **This is easy!**

[28]:
```
# Function to compute square
def square(num):
    return num**2
```

[29]:
```
# Function call using its name 'square' and a required parameter
square (4)
```

[29]:
```
16
```

Let's re-arrange the above function `"square"` in a single line.

[30]:
```
def square(num):return num**2
square (4)
```

[30]:
```
16
```

Now, let's convert this one line function `"def square(num):return num**2"` into a lambda expression
Steps to re-write above function to a lambda expression

- replace `"def"` with `"lambda"`
- delete function name `"square"`
- remove `"()"` around num
- delete `"return"`

The modified code in the cell below will be your lambda expression, let's try! ***We will use lambda expressions a lot in our pandas library.***

[31]:
```
lambda num : num*2
# hold-on, we will use this expression in map and filter to understand in details
```

[31]:
```
<function __main__.<lambda>(num)>
```

9 map & filter

Let's talk about **map** and **filter** now. We have seen that we can compute squares of all the numbers in a list using `for` loop. To do this, we need to follow the following steps:

- create an empty list
- Iterate over `my_list` in a `for` loop
- append square of each number to the empty list

Lets try to do the above tasks with `for` loop first, **This is tedious!**

[32]:
```
my_list = [1,2,3,4,5]
num_sq = []
for num in my_list:
    num_sq.append(num**2)
print (num_sq)
```

```
[1, 4, 9, 16, 25]
```

The task above can be much simplified using `map()`, let's try to understand how?

9.1 `map()`

- `map()` is a function with two arguments, a `function` and a `sequence` (e.g. list).
- it maps a function to every element in a sequence

NumPy for Data Analysis

(NumPy array, array methods & attributes)

Welcome to NumPy Essentials (Part 1) lecture.

NumPy is a fundamental package for scientific computing, and provides foundations of mathematical, scientific, engineering and data science programming within the Python echo-system. Hence, it is very important to have excellent understanding of this powerful library. The main object in NymPy is it's homogeneous multidimensional array.

In this section, we will cover the key concepts from NumPy and later on, we will be frequently using them in this course. Let's start with `NumPy arrays` along with some important built-in methods & attributes that are related to these arrays.

Note: *For complete documentation and to explore more on NumPy, you can always visit it's official website, http://www.numpy.org.*

NumPy is extremely **important for Data Science** because:

- It's a linear algebra library
- It's powerful and incredibly fast
- it Integrate C/C++ and Fortran code

As a building block of most of the PyData eco-system libraries, NumPy is significantly important for Data Science. Once, we have installed NumPy, we need to import it.
Let's import "numpy" and check its version.

[1]:
```
# A standard way of importing NumPy
import numpy as np
```

[2]:
```
# Let's check numpy's version
np.__version__
```

[2]:
```
'1.18.1'
```

1 NumPy Arrays

NumPy arrays is the main concept that we will be using in this course. These arrays essentially come in two flavours:

- **Vectors:** Vectors are strictly 1-dimensional array
- **Matrices:** Matrices are 2-dimensional *(matrix can still have only one row or one column)*

We can conveniently create NumPy arrays from Python datatypes e.g. lists, tuples.
Let's start with creating a Python list, we will then create a NumPy array using that list.

1.1 NumPy arrays from Python list

[3]:
```
# Creating a Python list "my_list".
my_list = [-1,0,1]
# Well, its good idea to confirm items in "my_list" and it's type!
my_list, type(my_list)
```

[3]:
```
([-1, 0, 1], list)
```

To create a NumPy array, from a Python data structure (list in our case), we need NumPy's array function. This array function can be accessed using dot operator "." on "np", which is alias for this library (chby typing "`np.array`". We need to cast our Python data structure, my_list, as a parameter to the array function. (*"np" is an alias for NumPy, and "." is to access the required function*).

Another broadcasting example

[27]:
```
array_1 = np.arange(1,4)
array_2 = np.arange(1,4)[:, np.newaxis]
```

[28]:
```
# Official way of printing is used, format() and len() are used for revisions
print(array_1)
print("Shape of the array is: {}, this is {}-D array".format(array_1.
  ↪shape,len(array_1.shape)))
# (3,) indicates that this is a one dimensional array (vector)
```

```
[1 2 3]
Shape of the array is: (3,), this is 1-D array
```

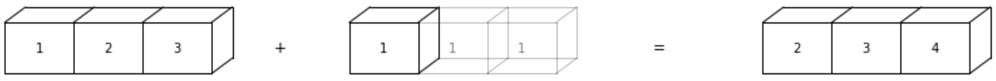
[29]:
```
# Official way of printing is used, format() and len() are used for revisions
print(array_2)
print("Shape of the array is: {}, this is {}-D array".format(array_2.
  ↪shape,len(array_2.shape)))
# (3, 1) indicates that this is a 2-D array (matrix)
```

```
[[1]
 [2]
 [3]]
Shape of the array is: (3, 1), this is 2-D array
```

[30]:
```
# Broadcasting arrays
array_1 + array_2
```

[30]:
```
array([[2, 3, 4],
       [3, 4, 5],
       [4, 5, 6]])
```

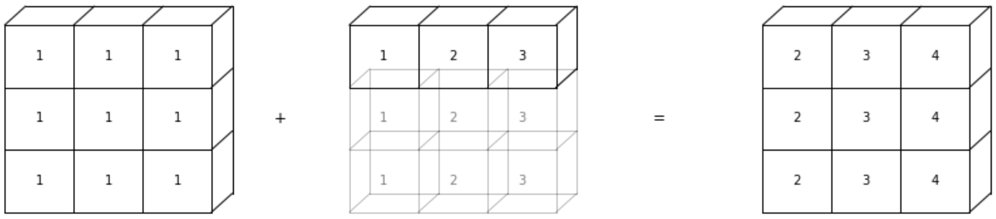
If you still have some confusion on broadcasting, let's try to understand visually and then hands-on code! Our array_1 is "1 row & 3 columns" and our array_2 is "1 row & 1 column". Let's broad case array_2 to array_1.



[31]:
```
# Here is the practical example
np.array([1,2,3]) + np.array([1])
```

[31]:
```
array([2, 3, 4])
```

Let's consider broadcasting an array of "3 rows & 1 column" to an array of "3 rows & 3 columns".



[32]:
```
# Here is a code for above visual example!
np.ones([3,3]) + np.array([1,2,3])
```

Pandas for Data Analysis

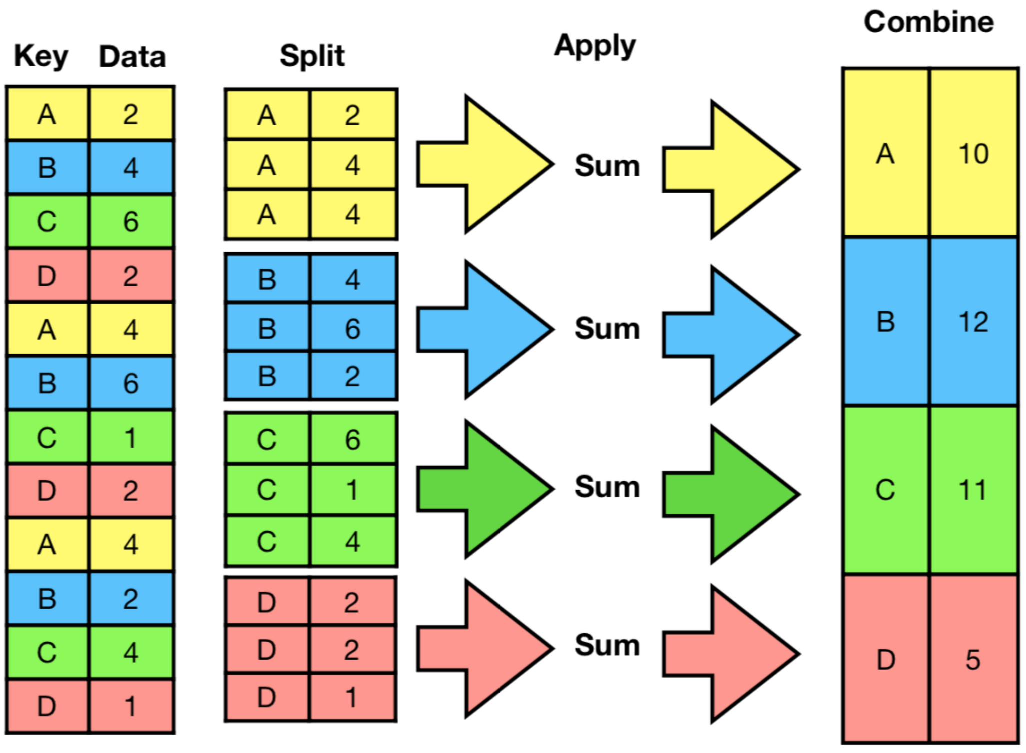
(GroupBy)

5 Groupby

Groupby is one of the most important and key functionality in pandas. It allows us to group data together, call aggregate functions and combine the results in three steps *split-apply-combine*: Before we move on to the hands-on, let's try to understand how this split-apply-combine work, using a data in different colours!

- **Split:** In this process, data contained in a pandas object (e.g. Series, DataFrame) is split into groups based on one or more keys that we provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1).
- **apply:** Once splitting is done, a function is applied to all groups independently, producing a new value.
- **combine:** Finally, the results of all those functions applications are combined into a resultant object. The form of the resulting object will usually depend on what's being done to the data.

The figure below would be helpful to conceptualize the concept of GroupBy operation!



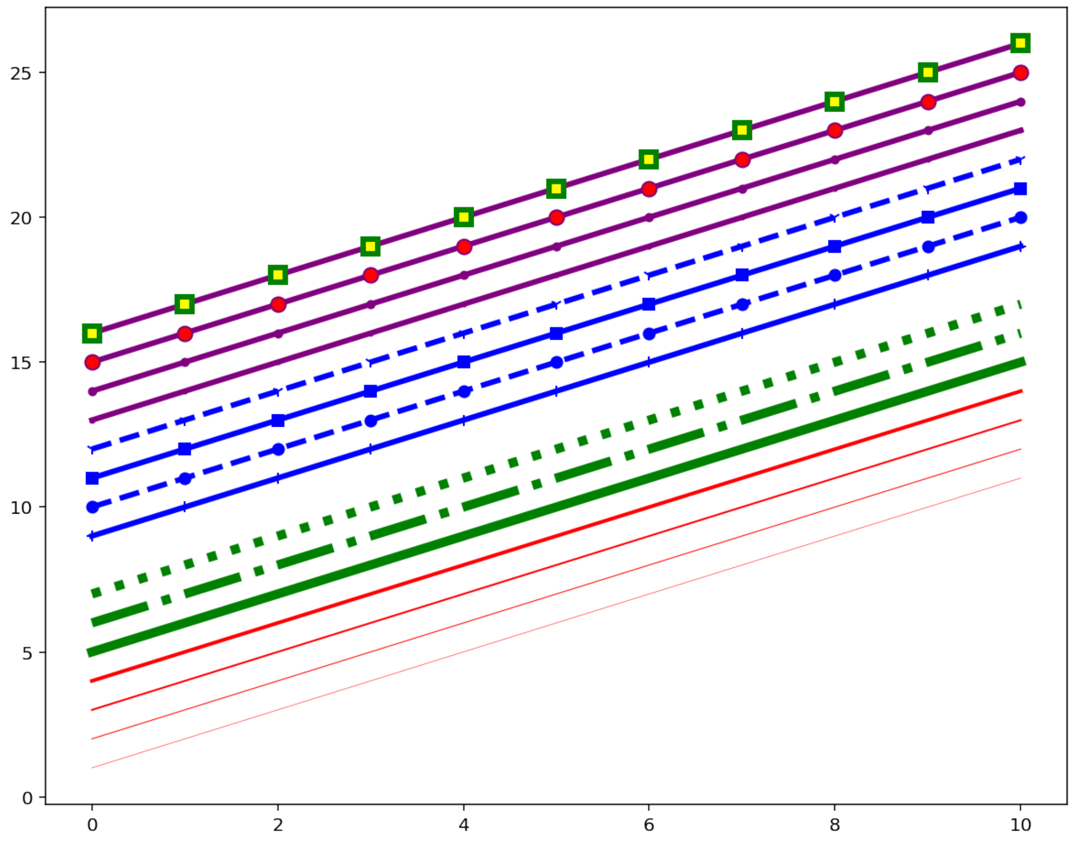
Lets explore with some examples:

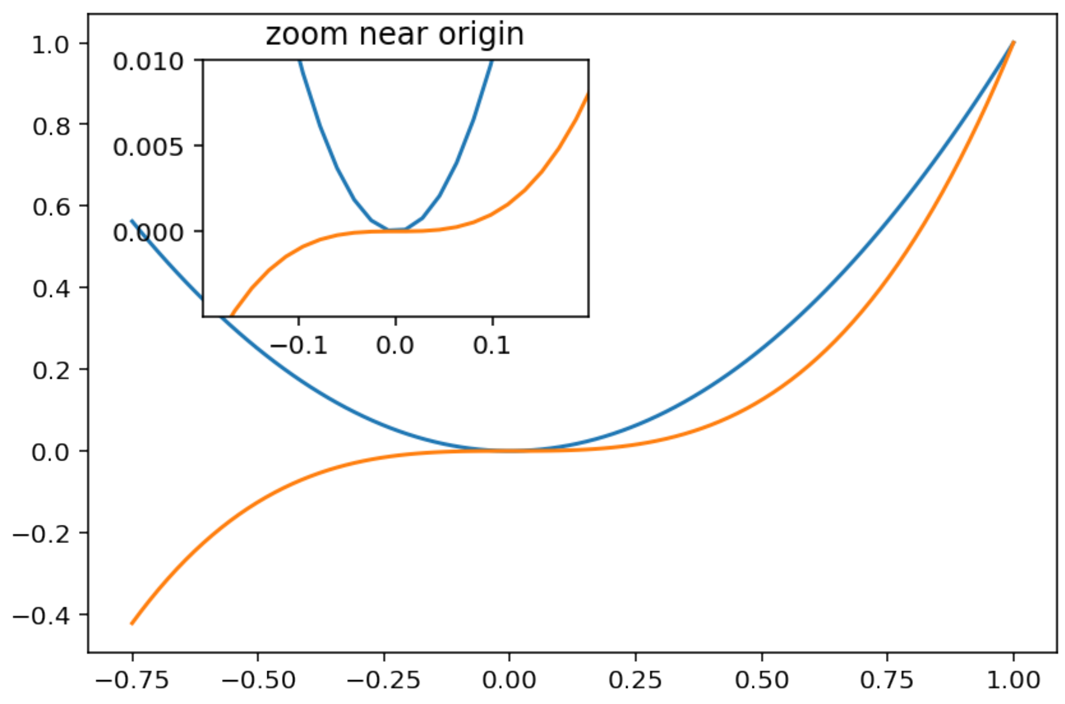
[1]:
```python
import pandas as pd # required import
```

Let's create a dictionary and convert that into pandas dataframe

[2]:
```python
# Create a dataframe
data = {'Store':['Walmart','Walmart','Costco','Costco','Target','Target'],
        'Customer':['Tim','Jermy','Mark','Denice','Ray','Sam'],
        'Sales':[150,200,550,90,430,120]}
```

```
# adding plots of different colors and line widths on axes
axes.plot(x, x+1, color="red", linewidth=0.25)
axes.plot(x, x+2, color="red", linewidth=0.50)
axes.plot(x, x+3, color="red", linewidth=1.00)
axes.plot(x, x+4, color="red", linewidth=2.00)

# possible linestype options '-', '-', '-.', ':', 'steps'
axes.plot(x, x+5, color="green", lw=5, linestyle='-')
axes.plot(x, x+6, color="green", lw=5, ls='-.')
axes.plot(x, x+7, color="green", lw=5, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3',␣
 ↪'4', ...
axes.plot(x, x+ 9, color="blue", lw=3, ls='-', marker='+')
axes.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
axes.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
axes.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# marker size and color
axes.plot(x, x+13, color="purple", lw=3, ls='-', marker='o', markersize=2)
axes.plot(x, x+14, color="purple", lw=3, ls='-', marker='o', markersize=4)
axes.plot(x, x+15, color="purple", lw=3, ls='-', marker='o', markersize=8,␣
 ↪markerfacecolor="red")
axes.plot(x, x+16, color="purple", lw=3, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```





Colormaps and contour figures are very useful for plotting functions of two variables. In most of these functions we encode one dimension of the data using a colormap. Let's learn with a simple example:

[18]:
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

# creating custom function which returns some computation
def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.cos(phi_ext -␣
 ↪2*phi_p)
```

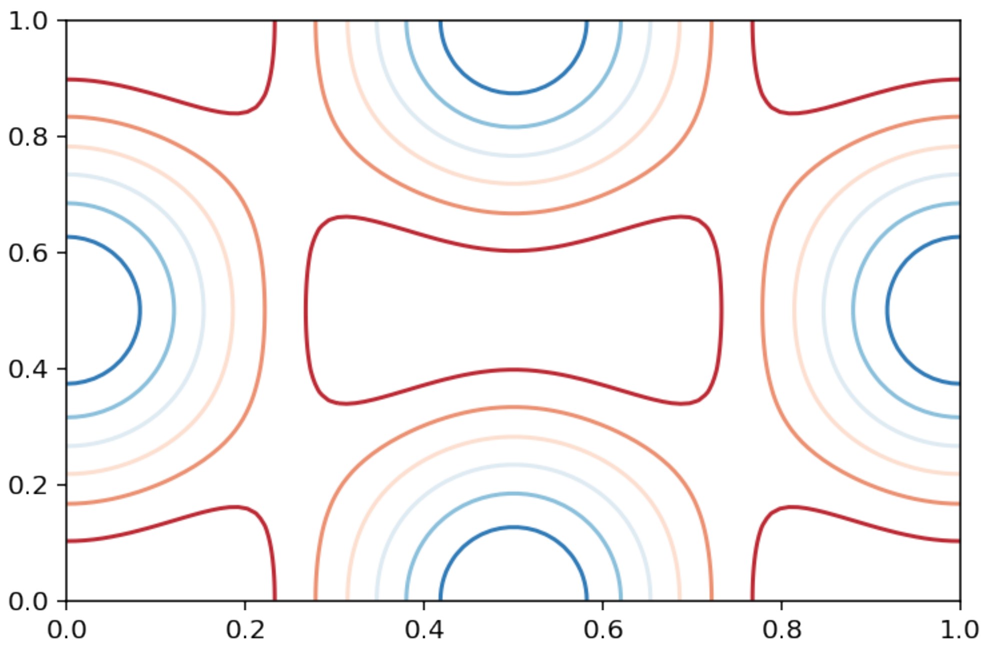
[19]:
```
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T # Function call for the function in above cell
```

pcolor(): Create a pseudocolor plot with a non-regular rectangular grid.

[20]:
```
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi),
              Z, cmap=matplotlib.cm.RdBu,
              vmin=abs(Z).min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```

contour(): As from its name, it plot contours.

[22]:
```
# contour
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(),
                 vmax=abs(Z).max(), extent=[0, 1, 0, 1])
```



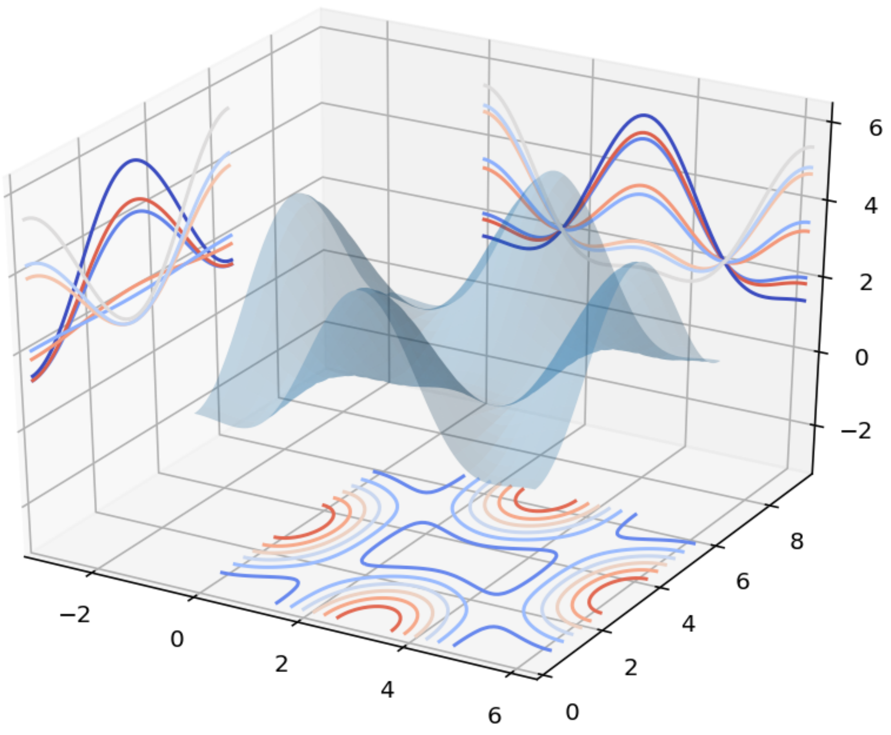
3D graphics in matplotlib is possible first, create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods. ***We need to do import for Axes3D to use its features!***

[23]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

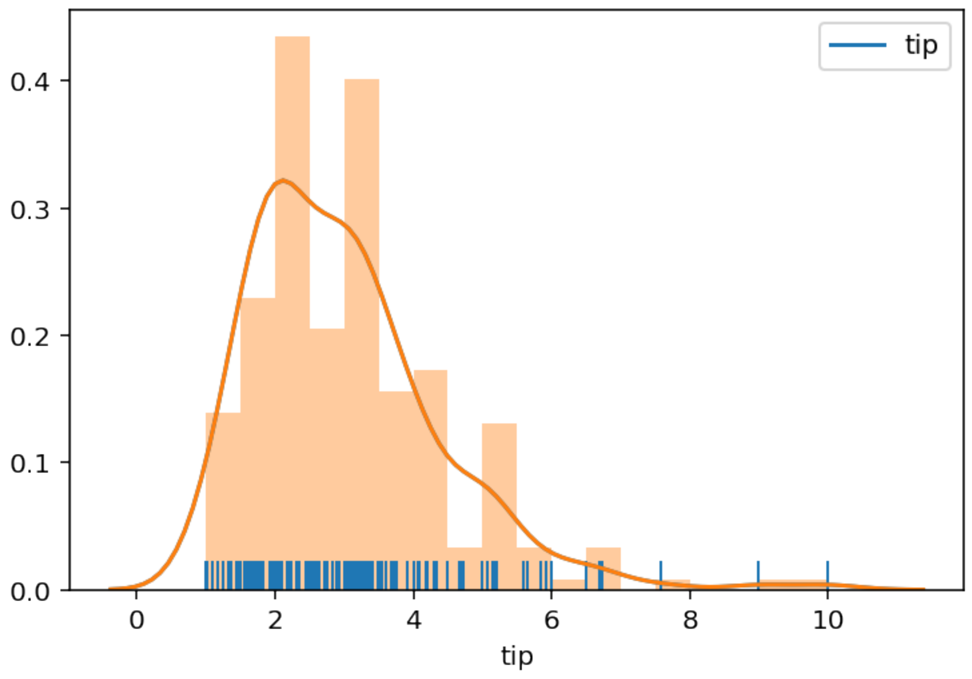
Let's create a 3D surface colored map using **plot_surface()** method!

[24]:
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword argument␣
 ↪to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                    cmap=matplotlib.cm.coolwarm, linewidth=0,
                    antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```

[26]:
```
fig = plt.figure(figsize=(8,6))

axes = fig.add_subplot(1,1,1, projection='3d')

axes.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = axes.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = axes.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = axes.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.coolwarm)

axes.set_xlim3d(-np.pi, 2*np.pi);
axes.set_ylim3d(0, 3*np.pi);
axes.set_zlim3d(-np.pi, 2*np.pi);
```



Your performance is great so far!



It’s time to understand KDE (Kernel Density Estimation plots)

Kernel Density Estimation plots - `kdeplot`

Let’s try to understand `kde` plots using `rugplot`. How do we actually build kde line based on rugplot? The figure below (source) beautifully explains the process.

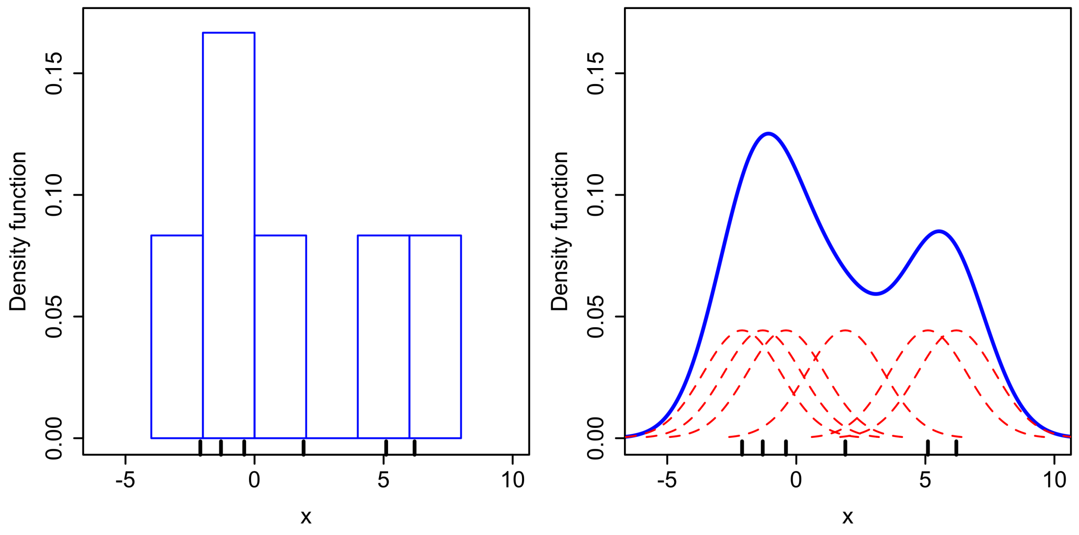


Figure shows the comparison of the histogram (left) and kernel density estimate (right) constructed using the same data. On the right, we have: Each of the 6 black dash is the `rugplot` with 6 individual kernels (normal Gaussian distribution) (the red dashed curves), on top of each black dash. The kernel density estimate is shown as blue curve – ***The kernels in red dashes are summed to make the kernel density estimate (solid blue curve)***. The data points are the rug plot on the horizontal axis.

KDE plots replace every single observation with a Gaussian (Normal) distribution centered around that value.

Let’s create a diagram to capture the concept of kde! *Don’t worry about understanding the code below, you*

We can **observe from the plot above**: * almost all the customers come for lunch on Thursday * almost all the customers come for dinner on the weekends

On Fridays, not many go out for lunch or dinner, they may want to save money and energy for the weekends!

2.2 Categorical distribution plots:

2.2.1 boxplot()

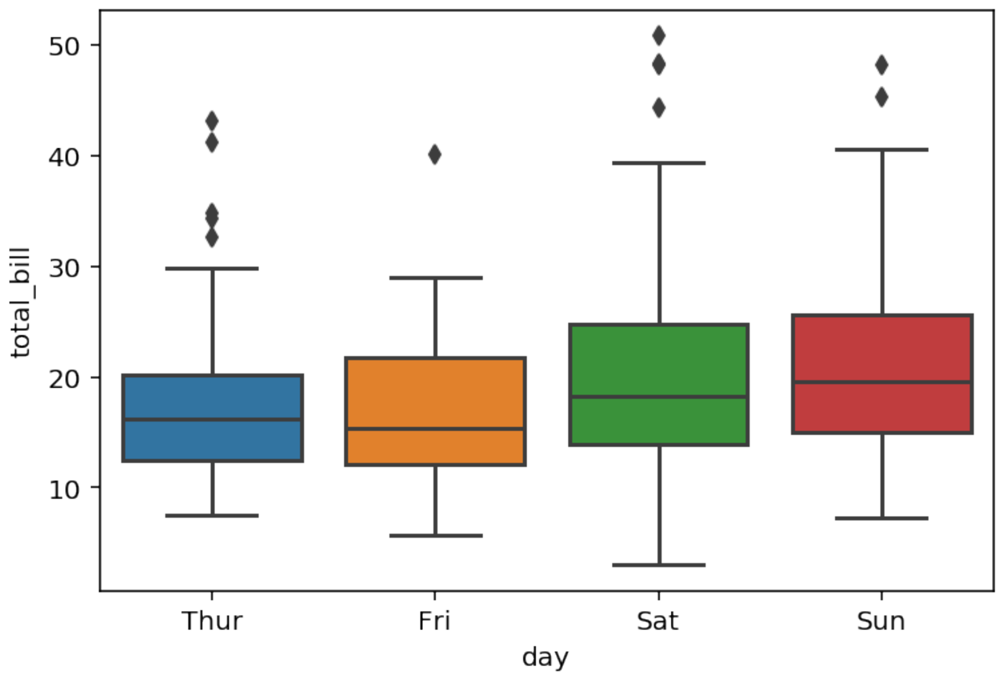
These type of plots are used to show the distribution of categorical data. A **box plot** (also known as a **box-and-whisker plot**) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable.

Little more on boxplot! *The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers" using a method that is a function of the inter-quartile range. In statistics, the quartiles of a ranked set of data values are the three points that divide the dataset into four equal groups, each group comprising a quarter of the data. A quartile is a type of quantile. The first quartile (Q1) is defined as the middle number between the smallest number and the median of the data set. The second quartile (Q2) is the median of the data. The third quartile (Q3) is the middle value between the median and the highest value of the data set.*

Let's learn with example using out tips data:

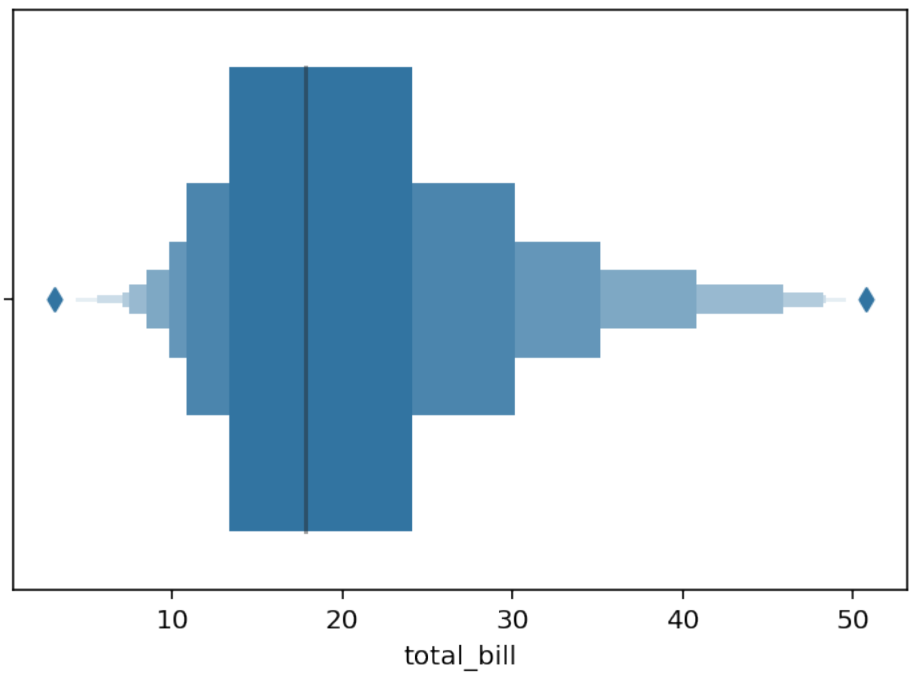
[8]:
```
sns.boxplot(x="day", y="total_bill", data=tips)#,palette='rainbow')
```

[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11a0a8d90>
```



☞ Understanding the Box Plot: We have `total_bill` along Y and category 'day' along X. If we look at any one of the box, we see the data points in each box / box-and-whisker plot are divided into four quartile groups : * **Quartile group 1** - Q1: between the bottom whisker and the bottom of the box * **Quartile group 2** - Q2: between the bottom of the box to the line in the box (median) * **Quartile group 3** - Q3: between the median and the upper end of the box * **Quartile group 4** - Q4: between upper end of the box and the upper whisker

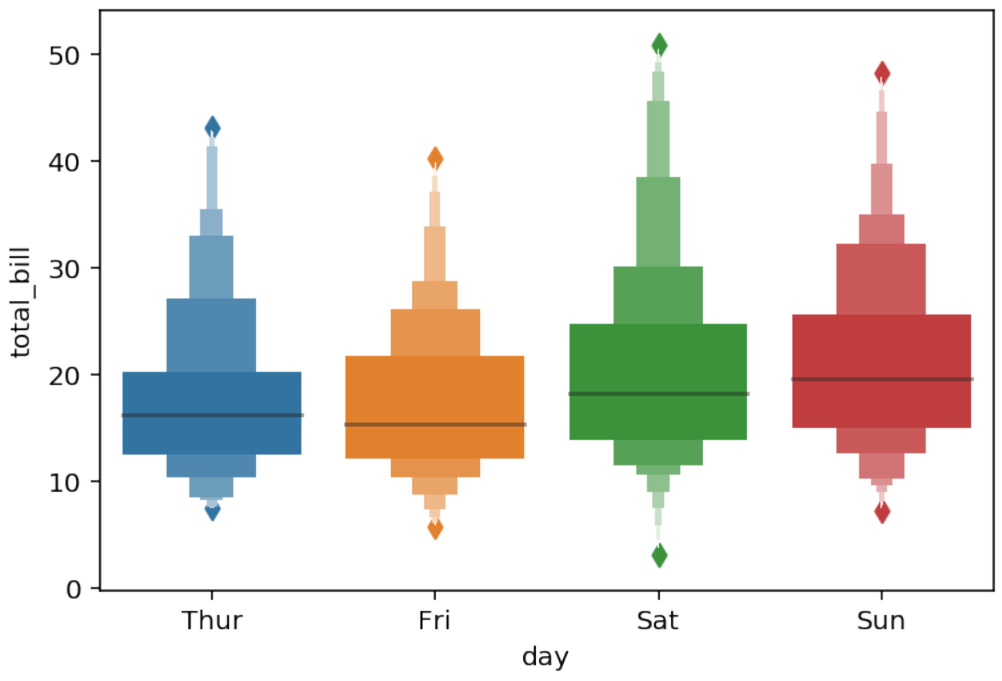
Points outside the quartiles are **outliers**.



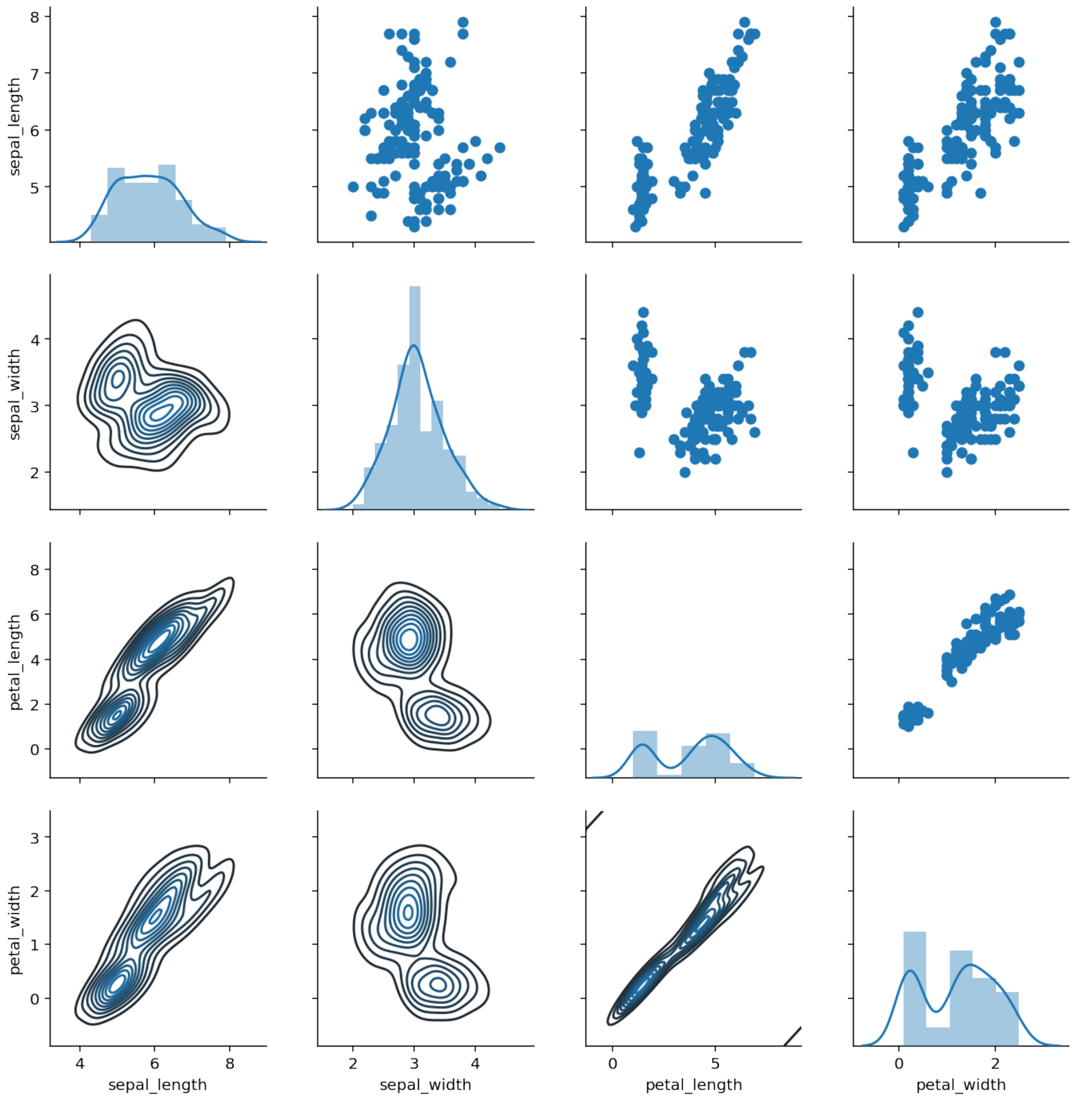
Compare the above plot with the `distplot` from the previous lecture. Bigger the bar is, bigger the box in `boxenplot` is! Let's try one more using two variables, "day" and "total_bill".

[13]:
```
sns.boxenplot(x="day", y="total_bill", data=tips)
```

[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x11c7acad0>
```



We can pass `hue` parameter along with our own choice of `palette` as well!



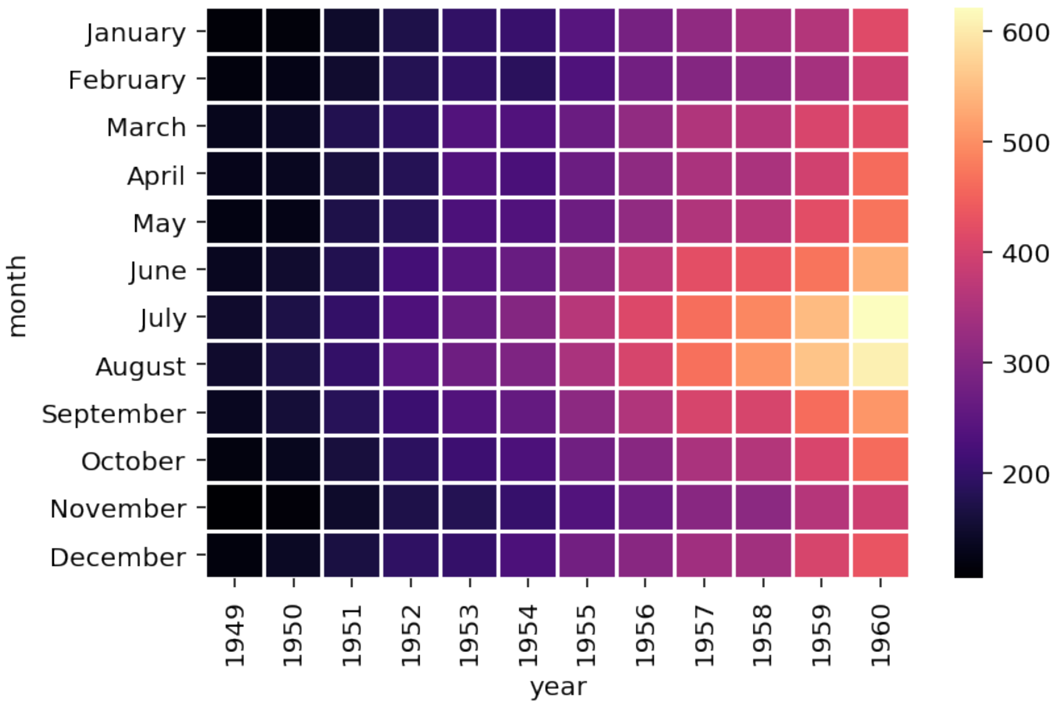
Now, we have lot more control on our plot instead of only a scatter plot of histogram, we have three different types of plots on the same grid for our data. We can chose what plot type we want to map on what part of the grid using seaborn's `PairGrid`!

3.3 pairplot()

Let's recall our understanding of "pairplot()" from the earlier lecture, distribution plots. Remember, we just pass in the data frame to pairplot() method and it will automatically create the pairplot for us. ☞ *pairplot is a simpler version of PairGrid, we will use pairplot() quite often in the course, so it is important to recall this simpler version first.*

[15]:
```
# How the pairplot look like for the entire data
sns.pairplot(iris)
```

[15]:
```
<seaborn.axisgrid.PairGrid at 0x125449d50>
```

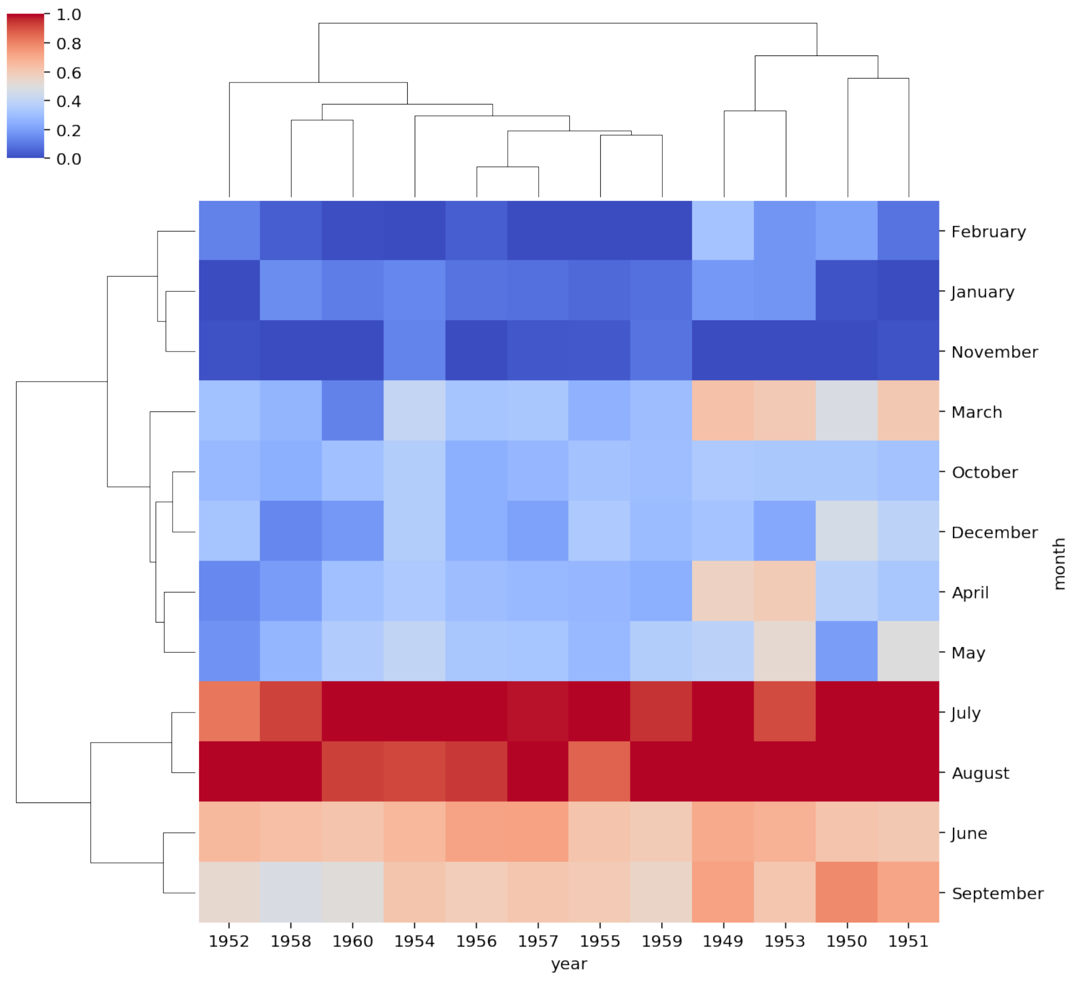


We see the map looks much better now. * We see that the overall no of passengers increases as we move from 1949 to 1960. This make sense because more people used airplanes with time and more flights were available * We can also observe that the popular months for travel are usually summer months, **Jule, July and August**!

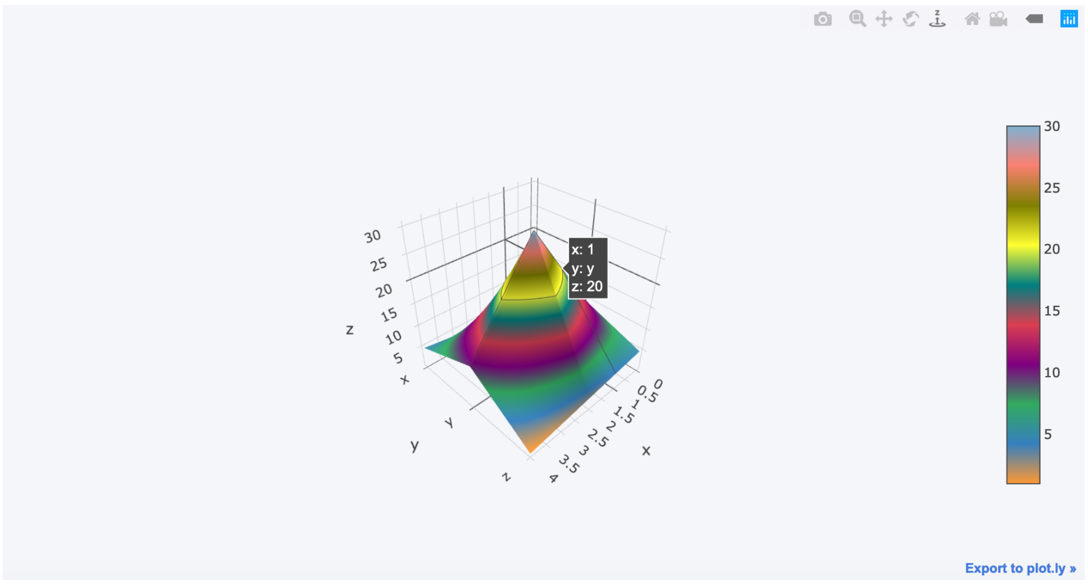
4.2 clustermap()

These are great plots for specific tasks of clustering the data! `clustermap` uses hierarchal clustering to produce a clustered version of the `heatmap` where similar groups are close to each other. For example, let's call the clustermap for the same data "pvf" (data we got after `pvt_table`):

[13]:
```
g = sns.clustermap(pvf)
```



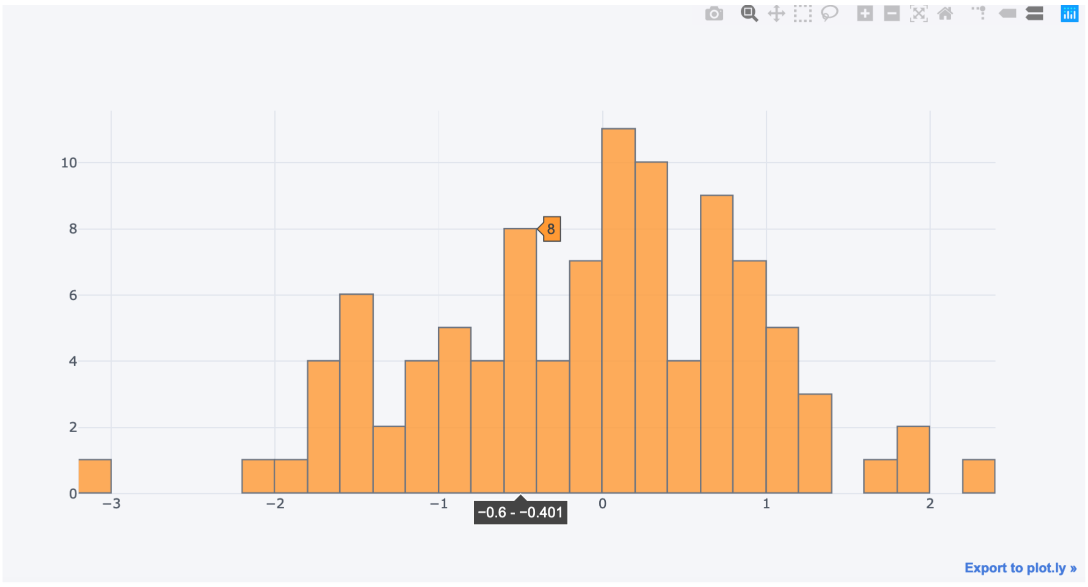
Based on the normalized scale, we get much better understanding of our data. We see that the winter months **November, January and February are always low passenger months** while the **summer months with red color are all the time high sale** or more passenger months!



1.0.5 `kind = hist`

Creating a histogram of a single column

[16]:
```
df1['A'].iplot(kind='hist',bins=50) # for a single column
#df1.iplot(kind='hist',bins=25) # overlaped hist plots, trun ON/OFF
```

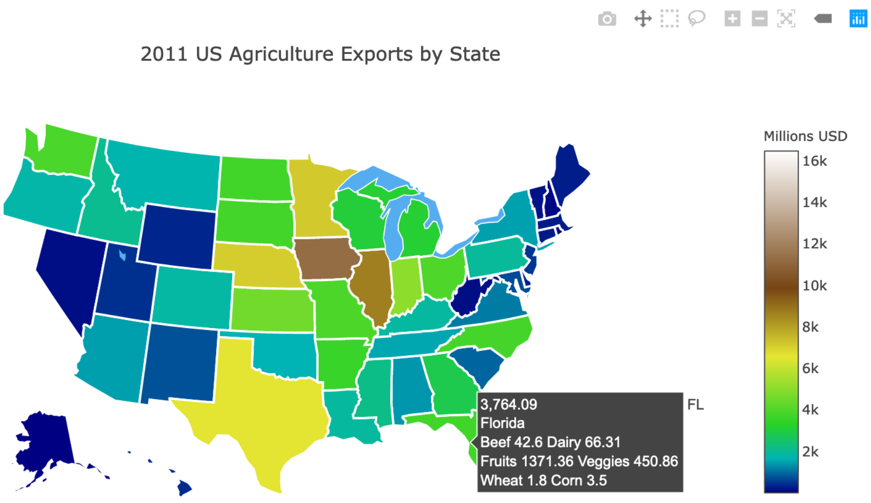


1.0.6 `kind = spread`

spread is usually used for stock data. The code below will give the plot and a spread between two selected columns at each point.

[17]:
```
df1[['A','B']].iplot(kind='spread')
```

```
                                showlakes = True, # we want actual lake on the map
                                lakecolor = 'rgb(85,173,240)') # blue for lakes
               )

# Passing data and layout to the go.Figure
choromap = go.Figure(data = [data], layout = layout)
# Passing map to iplot for plotting
iplot(choromap)
```



Some possible `colorscale` *you can try (please always look for the updated list for plotly documentation for changes):* `['Greys', 'YlGnBu', 'Greens', 'YlOrRd', 'Bluered', 'RdBu', 'Reds', 'Blues', 'Picnic', 'Rainbow', 'Portland', 'Jet', 'Hot', 'Blackbody', 'Earth', 'Electric', 'Viridis', 'Cividis']`

2.2 World Choropleth Map

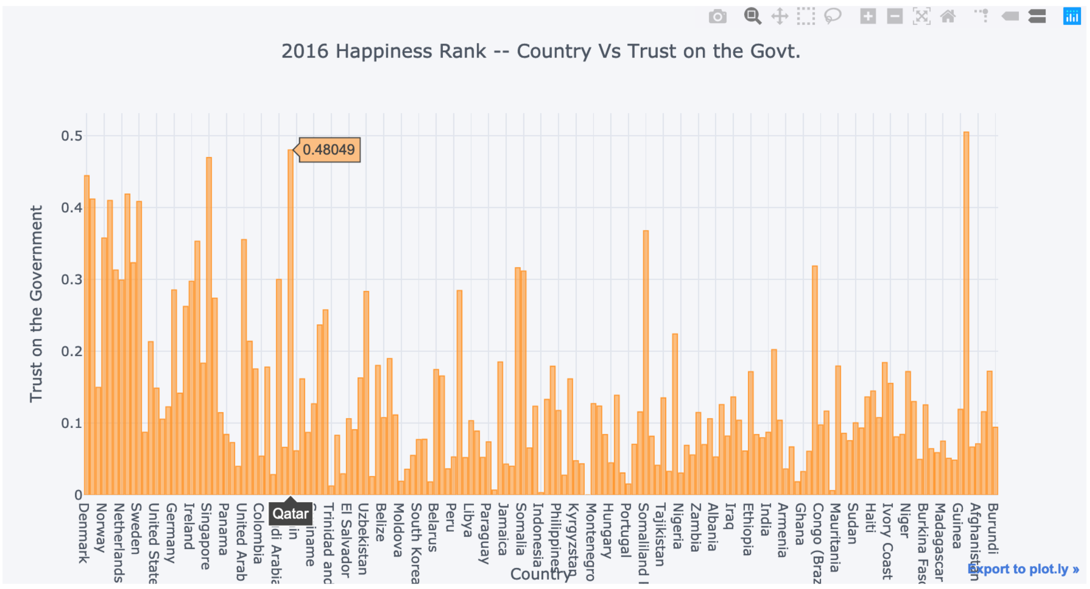
Now let's see an example with a World Map:

[32]:
```
# Let's read world GDP data from plotly datasts
gdp = pd.read_csv('https://raw.githubusercontent.com/plotly/datasets/master/
↪2014_world_gdp_with_codes.csv')
#gdp.to_csv('2014_world_gdp.csv') # used to save the file for offline use, you can␣
↪load and work with it!
```

[33]:
```
gdp.head()
```

[33]:
```
          COUNTRY  GDP (BILLIONS) CODE
0     Afghanistan           21.71  AFG
1         Albania           13.40  ALB
2         Algeria          227.80  DZA
3  American Samoa            0.75  ASM
4         Andorra            4.80  AND
```

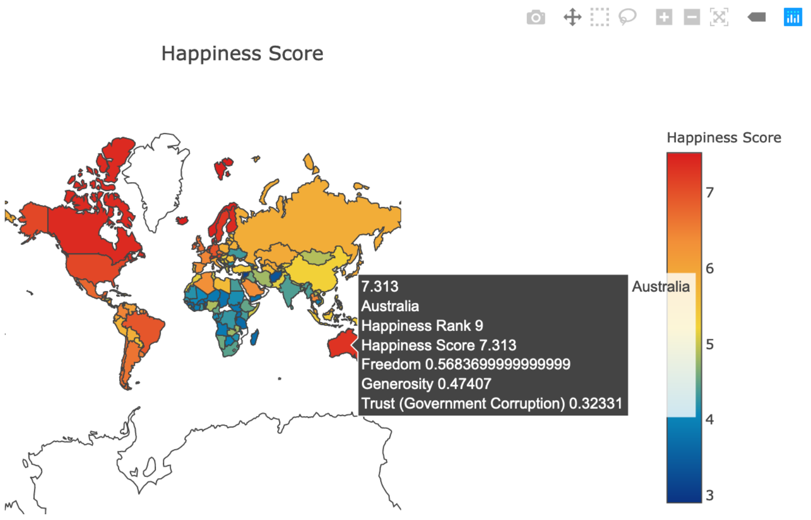
[34]:
```
# Data dictionary
data = dict(
        type = 'choropleth',
        locations = gdp['CODE'], # This is country code now
        z = gdp['GDP (BILLIONS)'],
```



Create a "choropleth" for happiness score that displays the data from the newly generated 'text' column when hover.

[17]:
```
#code here please
#Data dict


#Layout - a nested dictionary object



#plotting
#choropleth_map = go.Figure(data = [data],layout = layout)
#iplot(choropleth_map, validate=False)
```

[18]:
```
```



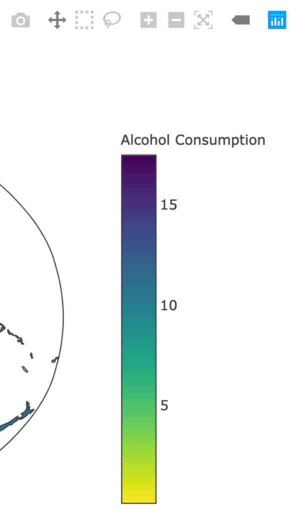
```
7.313
Australia
Happiness Rank 9
Happiness Score 7.313
Freedom 0.5683699999999999
Generosity 0.47407
Trust (Government Corruption) 0.32331
```

What conclusions you can draw from the above analysis?

```
#plotting
#choropleth_map = go.Figure(data = [data],layout = layout)
#iplot(choropleth_map,validate=False)
```

[33]:

```
```

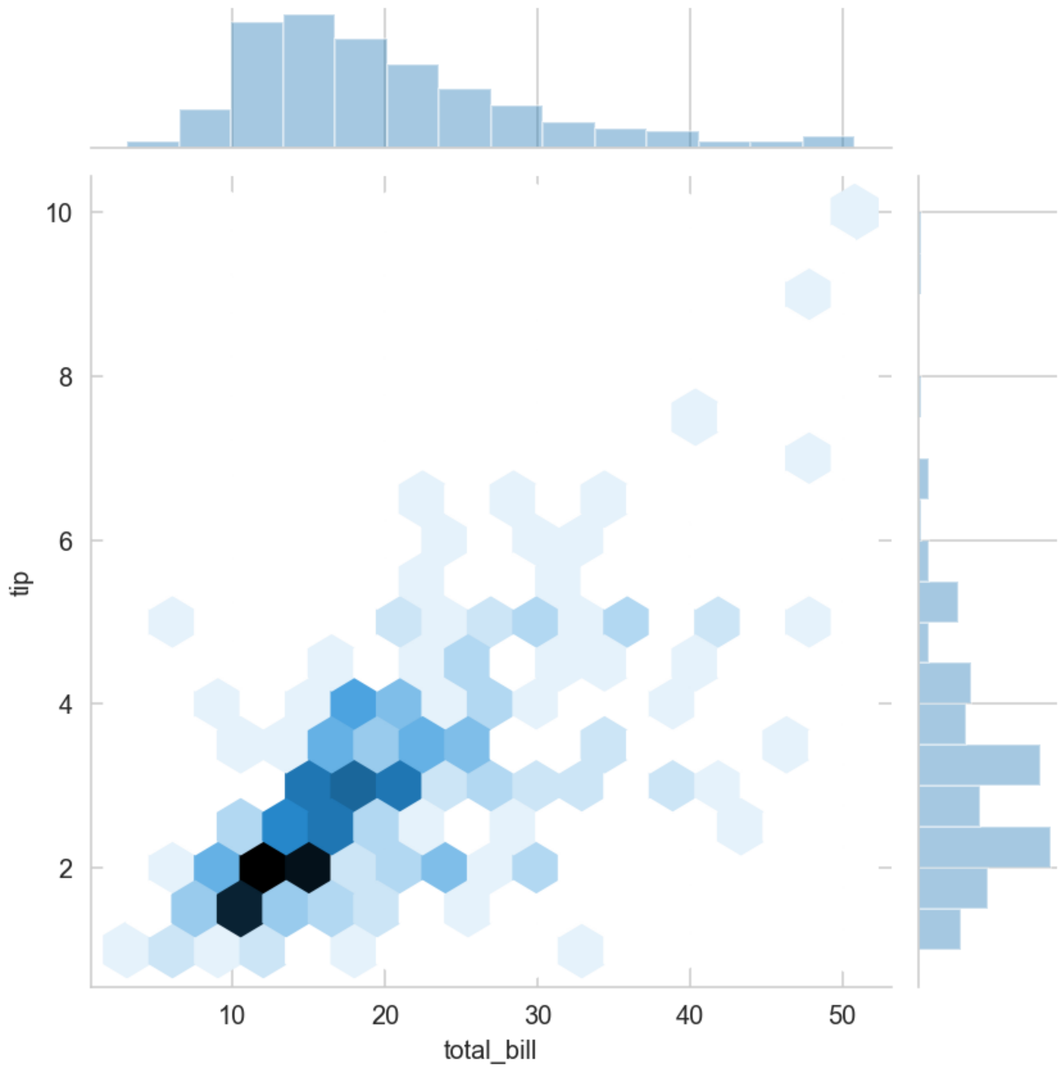


I am impressed with your progress. Keep your spirits high, we are getting ready for the Machine Learning journey in the second half of this course! learning journey is next!

Solutions

[15]:
```
# Leave this cell empty please.
sns.jointplot(x='total_bill',y='tip', data=tips, kind='hex')
```

[15]:
```
<seaborn.axisgrid.JointGrid at 0x1a1be700d0>
```

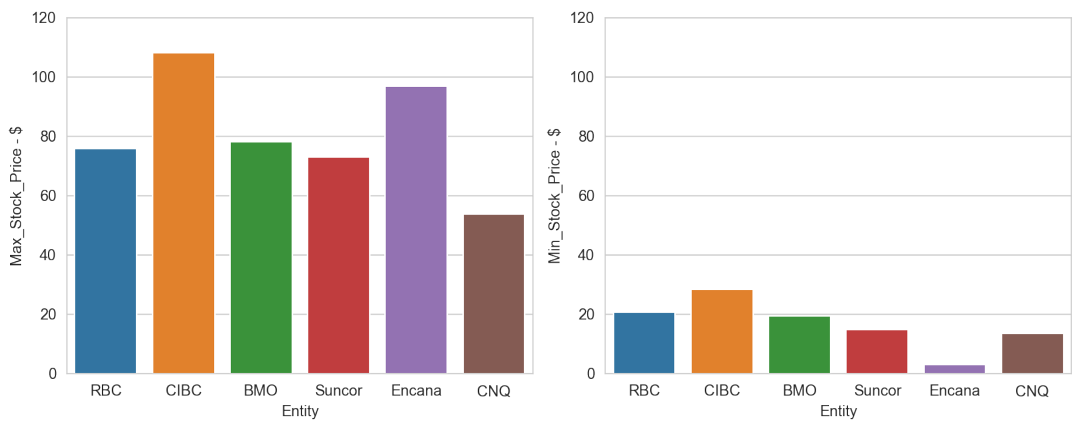


Recreate the distplot given below. Use titanic dataset. Hint: pay attention to x and y labels.

[16]:
```
# Code here and leave the next cell empty, so that you don't lose the plot!
```

[17]:
```
# Leave this cell empty please.
sns.distplot(titanic['fare'],bins=20,kde=False,color='green')
```

[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1bf58710>
```

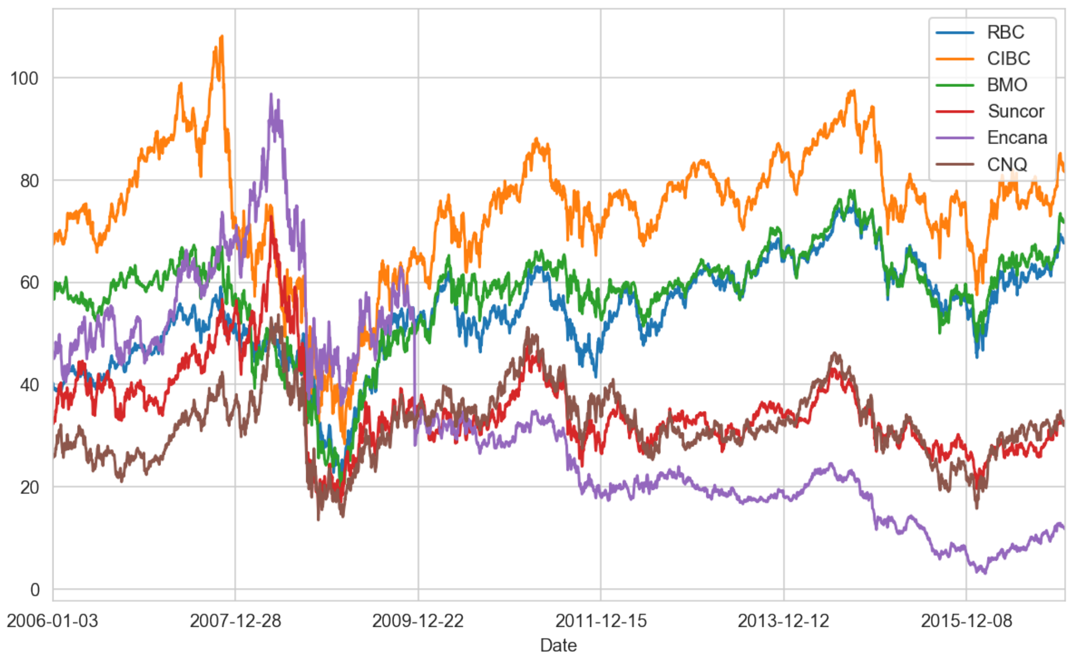


Line plot can give the idea on how the stock value changes with time, plot 'Close' value of each stock in your data against time. Recreate the plot below using `for` loop. Hint: for the look, you have `tickers = ['RBC', 'CIBC', 'BMO', 'Suncor', 'Encana','CNQ']`. Grab the column 'Close'

[21]:
```
# code here please
```

[22]:
```
# for loop
for tick in tickers:
    bo[tick]['Close'].plot(figsize=(10,6),label=tick)
plt.legend()
```

[22]:
```
<matplotlib.legend.Legend at 0x1a1d7cb910>
```

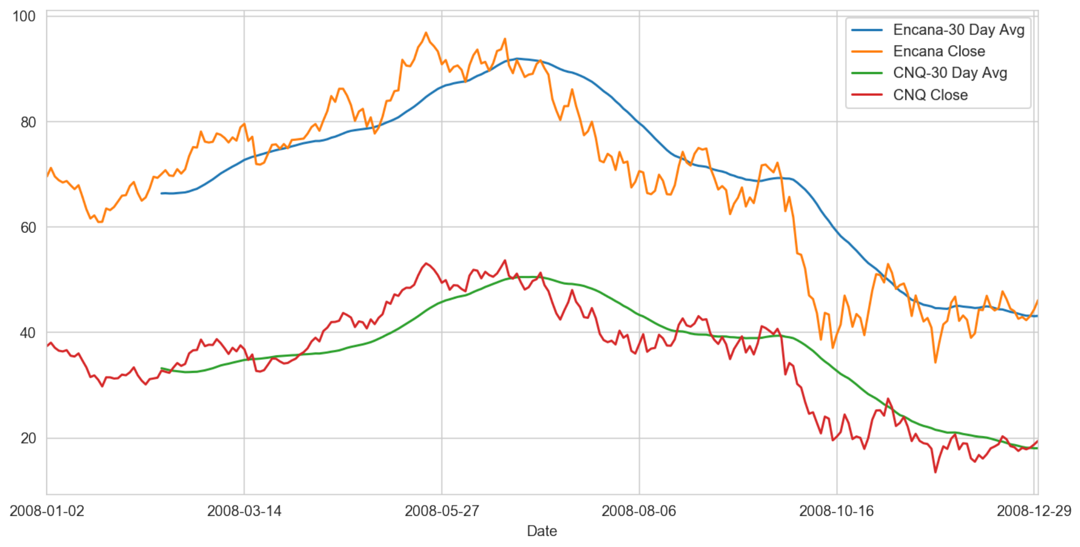


The above plot is good, however, the visualization can be improved with interactive data plotting. You can select the entity you want to display on the plot. *Use cross section `.xs()` method to plot interactive plot for your data.*

```
<class 'pandas.core.frame.DataFrame'>
Index: 2769 entries, 2006-01-03 to 2016-12-30
Data columns (total 5 columns):
Open      2769 non-null float64
High      2769 non-null float64
Low       2769 non-null float64
Close     2769 non-null float64
Volume    2769 non-null int64
dtypes: float64(4), int64(1)
memory usage: 129.8+ KB
```

[49]:
```
plt.figure(figsize=(12,6))
encana['Close'].ix['2008-01-01':'2008-12-30'].rolling(window=30).mean().
  ↪plot(label='Encana-30 Day Avg')
encana['Close'].ix['2008-01-01':'2008-12-30'].plot(label='Encana Close')
cnq['Close'].ix['2008-01-01':'2008-12-30'].rolling(window=30).mean().
  ↪plot(label='CNQ-30 Day Avg')
cnq['Close'].ix['2008-01-01':'2008-12-30'].plot(label='CNQ Close')
plt.legend()
```

[49]:
```
<matplotlib.legend.Legend at 0x1a1e774bd0>
```



A way to present correlation is to create a heatmap. Please create a heatmap of the correlation between the stocks close price.

[50]:
```
# code here please
```

[51]:
```
sns.heatmap(bo.xs(key='Close',axis=1,level='Stock').corr(),annot=True)
```

[51]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a1e81a9d0>
```

So, we have DataTime objects in the 'timeStamp' columns, great! We can grab specific attributes from a Datetime object, as we did above. For example:

```
time = df['timeStamp'].iloc[0]
time.hour, time.month etc
```

This is a good idea to do some feature extraction here. Let's create four new columns `'year'`, `'hour'`, `'month'`, `'data'` and `'day_of_week'` to your dataframe. Display first two rows to confirm if you have the new columns in the dataset? Recall `.apply()` with `lambda` expression!

[29]:
```
# Code here please
```

[30]:
```
df['year'] = df['timeStamp'].apply(lambda time: time.year)
df['hour'] = df['timeStamp'].apply(lambda time: time.hour)
df['month'] = df['timeStamp'].apply(lambda time: time.month)
df['date'] = df['timeStamp'].apply(lambda time: time.date())
df['week_day'] = df['timeStamp'].apply(lambda time: time.dayofweek)
df.head(2)
```

[30]:
```
         lat        lng                                               desc  \
0  40.297876 -75.581294  REINDEER CT & DEAD END;  NEW HANOVER; Station ...
1  40.258061 -75.264680  BRIAR PATH & WHITEMARSH LN;  HATFIELD TOWNSHIP...

       zip                    title           timeStamp                twp  \
0  19525.0   EMS: BACK PAINS/INJURY 2015-12-10 17:40:00        NEW HANOVER
1  19446.0  EMS: DIABETIC EMERGENCY 2015-12-10 17:40:00  HATFIELD TOWNSHIP

                         addr  e Reason                 Code  year  hour  \
0      REINDEER CT & DEAD END  1    EMS    BACK PAINS/INJURY  2015    17
1  BRIAR PATH & WHITEMARSH LN  1    EMS   DIABETIC EMERGENCY  2015    17

   month        date  week_day
0     12  2015-12-10         3
1     12  2015-12-10         3
```

Excellent, looks like we have the new columns in the date! The columns `'week_day'` and `'month'` are an integers from 0 to 6 and 1 to 12. We can pass a dictionary (with integers as key and day/month name as value) to `map()` and create columns 'day_name' and 'month_name' with their actual names.

Use the dictionaries below and create new columns 'day_name' and 'month_name' for the actual day and month using `map()` method.

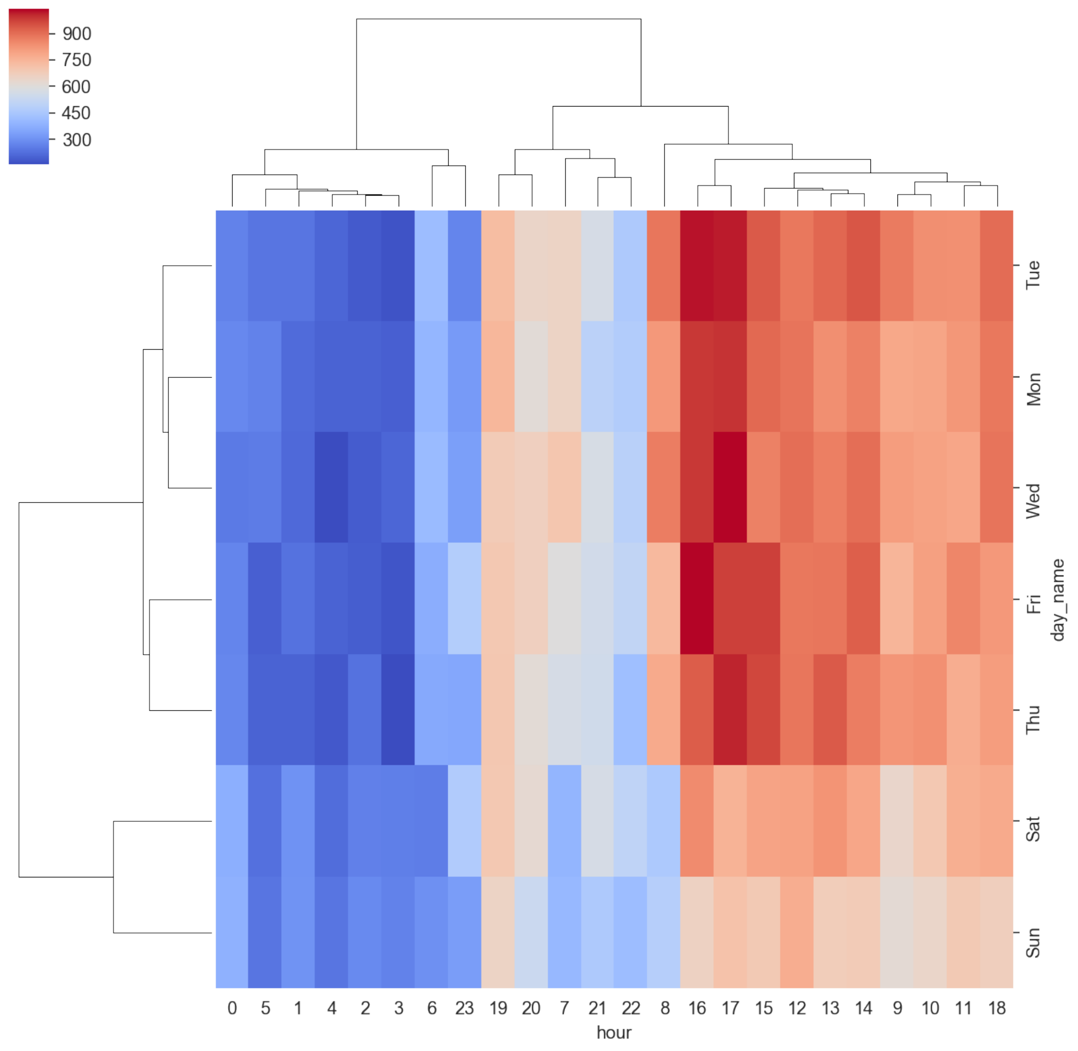
```
day_map = {0:'Mon',1:'Tue',2:'Wed',3:'Thu',4:'Fri',5:'Sat',6:'Sun'}
month_map = {1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May', 6:'Jun',
         7:'Jul', 8:'Aug', 9:'Sep', 10:'Oct', 11:'Nov', 12:'Dec'}
```

[31]:
```
day_map = {0:'Mon',1:'Tue',2:'Wed',3:'Thu',4:'Fri',5:'Sat',6:'Sun'}
month_map = {1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May', 6:'Jun',
             7:'Jul', 8:'Aug', 9:'Sep', 10:'Oct', 11:'Nov', 12:'Dec'}
```

[32]:
```
# Code here please
```

[33]:
```
df['day_name'] = df['week_day'].map(day_map)
df['month_name'] = df['month'].map(month_map)
df.head(2)
```

[33]:
```
         lat        lng                                               desc  \
0  40.297876 -75.581294  REINDEER CT & DEAD END;  NEW HANOVER; Station ...
1  40.258061 -75.264680  BRIAR PATH & WHITEMARSH LN;  HATFIELD TOWNSHIP...
```



Create an other dataframe for months as columns and recreate heatmap and cluster map.

[61]:
```
# Code here please
```

[62]:
```
by_dayMonth = df.groupby(by=['day_name','month']).count()['Reason'].unstack()
by_dayMonth.head()
```

[62]:
```
month        1     2     3     4     5     6     7     8    12
day_name
Fri       1970  1581  1525  1958  1730  1649  2045  1310  1065
Mon       1727  1964  1535  1598  1779  1617  1692  1511  1257
Sat       2291  1441  1266  1734  1444  1388  1695  1099   978
Sun       1960  1229  1102  1488  1424  1333  1672  1021   907
Thu       1584  1596  1900  1601  1590  2065  1646  1230  1266
```

[63]:
```
# Code here please
```

[64]:
```
plt.figure(figsize=(12,6))
sns.heatmap(by_dayMonth)#,cmap='coolwarm')
```

[64]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a18bdd410>
```