

course1review

February 1, 2022

This notebook will be a short review of key concepts in python. The goal of this notebook is to jog your memory and refresh concepts.

Table of contents

- Jupyter notebook
- Libraries
- Plotting
- Pandas DataFrame manipulation
- Unit testing
- Randomness and reproducibility
- Bonus: list comprehension

0.1 Jupyter notebook

Straight from the [Jupyter website](#): “The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.”

To run code in a cell, you can press the run tab, or press control + enter. The ‘Kernel’ tab is quite useful. If you find your code is stuck running (maybe you wrote an infinite loop), you can go to ‘Kernel’ -> ‘Interrupt’ to force quit.

A few useful keyboard shortcuts: * Run cell, select below: shift + enter * Run cell: ctrl + enter * Run cell, insert below: option + enter

By pressing the ‘esc’ key, you enter command mode (the colored border around the currently selected cell should change from green to blue). Once in command mode, you can use these shortcuts: * Insert cell above: a * Insert cell below: b * Copy cell: c * Paste cell: v * Delete selected cell(s): d d * Change selected cell to markdown: m * Change selected cell to code: y

To exit command mode, click anywhere in a cell or press enter.

And of course, don’t forget the ever useful: * Save file: command + s

0.2 Libraries

There are a few libraries that you will use almost always: * Numpy * Pandas * Matplotlib or * Seaborn

The key points to remember are how to import these libraries and their standard import names, as well as their main uses.

Numpy (from the [Numpy website](#)):
NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Pandas (from the [Pandas website](#)):
high-performance, easy-to-use data structures and data analysis tools for the Python programming language

Both the Matplotlib and Seaborn libraries are for creating graphs.

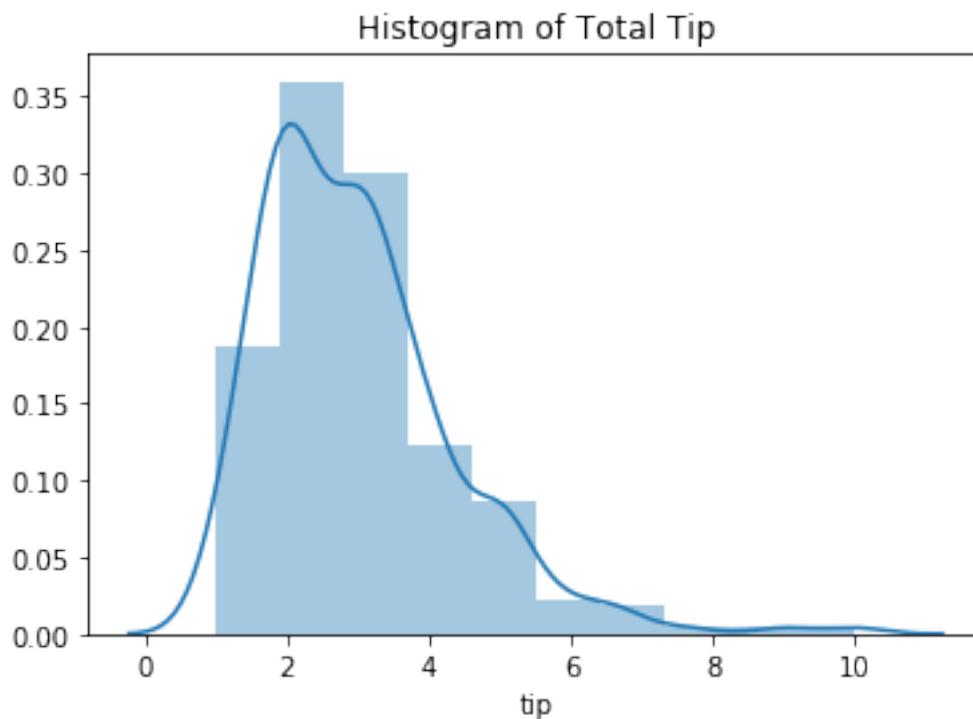
```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

0.3 Plotting

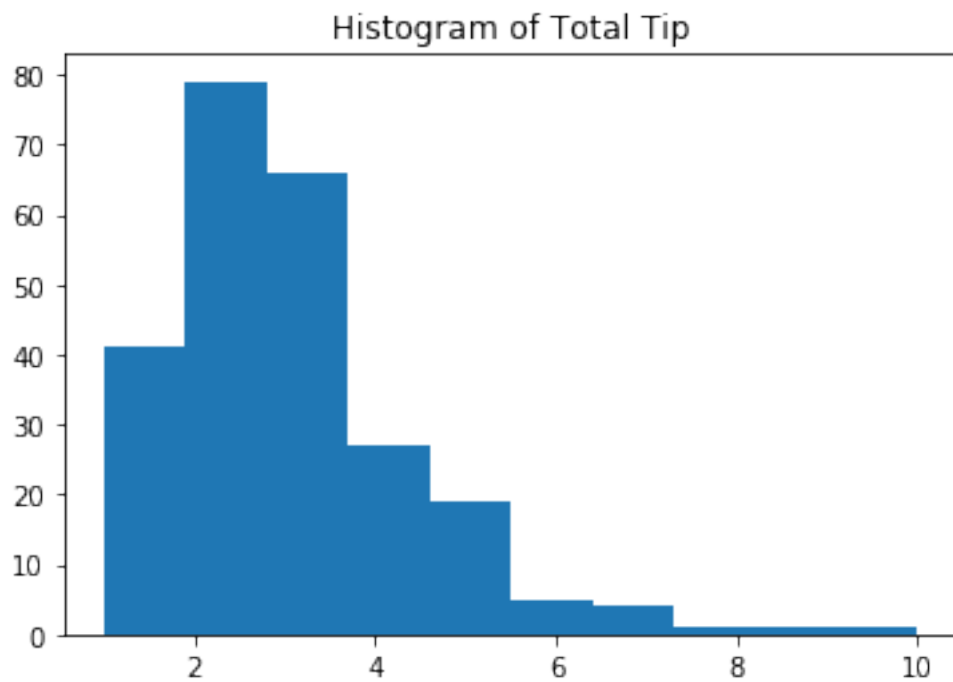
```
In [2]: # Load in the data set
tips_data = sns.load_dataset("tips")
```

0.3.1 Plot a histogram of the tips

```
In [4]: # with seaborn
sns.distplot(tips_data["tip"], kde = True, bins=10).set_title("Histogram of Total Tip")
plt.show()
```

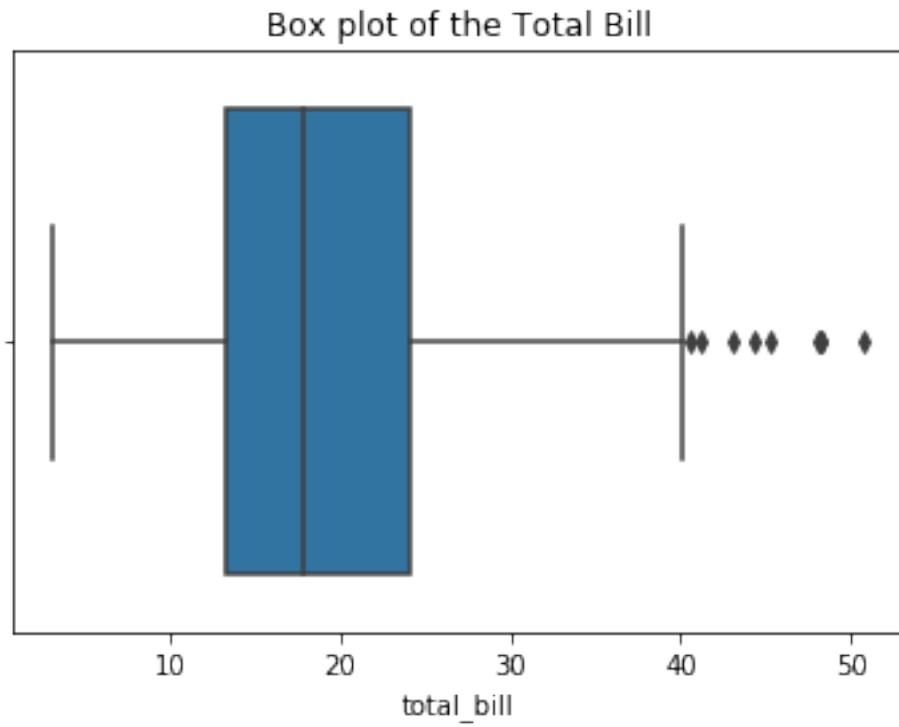


```
In [5]: # with matplotlib
plt.hist(tips_data['tip'], bins=10)
plt.title("Histogram of Total Tip")
plt.show()
```

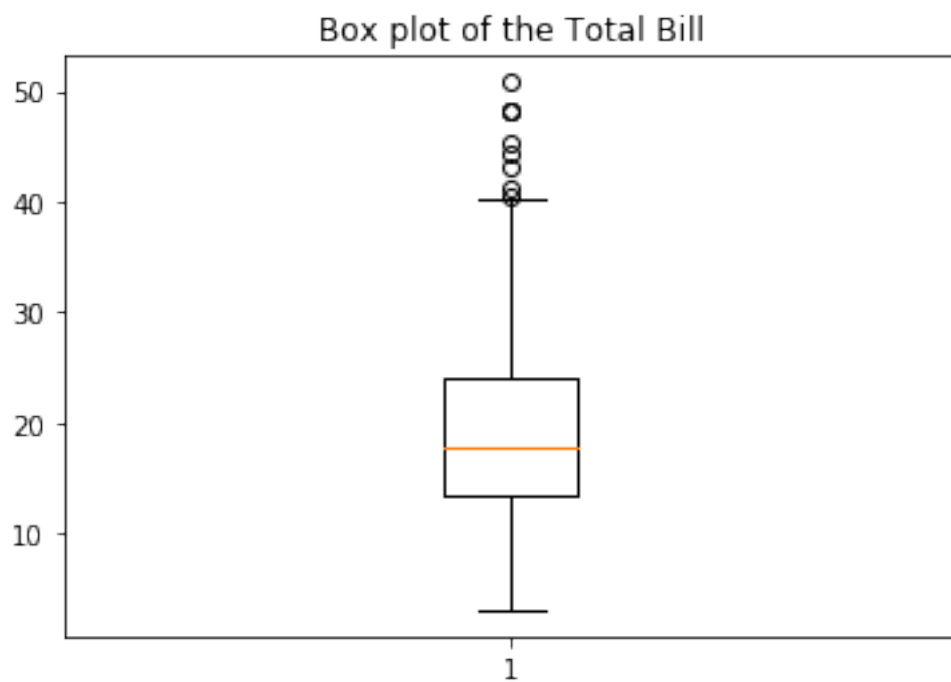


0.3.2 Create a boxplot of the total bill amounts

```
In [6]: # with seaborn
sns.boxplot(tips_data["total_bill"]).set_title("Box plot of the Total Bill")
plt.show()
```



```
In [7]: # with matplotlib
plt.boxplot(tips_data["total_bill"])
plt.title("Box plot of the Total Bill")
plt.show()
```



0.4 Pandas DataFrame manipulation

```
In [8]: # Import NHANES 2015-2016 data
df = pd.read_csv("nhanes_2015_2016.csv")
```

```
In [9]: # look at top 3 rows
df.head(3)
```

```
Out[9]:
```

	SEQN	ALQ101	ALQ110	ALQ130	SMQ020	RIAGENDR	RIDAGEYR	RIDRETH1	\
0	83732	1.0	NaN	1.0	1	1	62	3	
1	83733	1.0	NaN	6.0	1	1	53	3	
2	83734	1.0	NaN	NaN	1	1	78	3	

	DMDCITZN	DMDEDUC2	...	BPXSY2	BPXDI2	BMXWT	BMXHT	BMXBMI	BMXLEG	\
0	1.0	5.0	...	124.0	64.0	94.8	184.5	27.8	43.3	
1	2.0	3.0	...	140.0	88.0	90.4	171.4	30.8	38.0	
2	1.0	3.0	...	132.0	44.0	83.4	170.1	28.8	35.6	

	BMXARML	BMXARMC	BMXWAIST	HIQ210
0	43.6	35.9	101.1	2.0
1	40.0	33.2	107.9	NaN
2	37.0	31.0	116.5	2.0

[3 rows x 28 columns]

Pick columns by name

```
In [10]: df['SEQN'].head()
```

```
Out[10]:
```

0	83732
1	83733
2	83734
3	83735
4	83736

Name: SEQN, dtype: int64

Pick columns and rows by index name

```
In [14]: # indices 0 e 1, columnas SEQN e RIAGENDR
df.loc[[0, 1], ['SEQN', 'RIAGENDR']]
```

```
Out[14]:
```

	SEQN	RIAGENDR
0	83732	1
1	83733	1

Pick columns and rows by index location

```
In [16]: # indices 0 e 1, colunas 0 (SEQN) e 5 (RIAGENDR)
         df.iloc[[1,2], [0,5]]
```

```
Out[16]:      SEQN  RIAGENDR
         1  83733          1
         2  83734          1
```

0.5 Unit testing

This is the idea that you should run complicated code on a simple test case that you know that outcome of. If your code outputs something you did not expect, then you know there is an error somewhere that must be fixed. When working with large datasets, it is easy to get reasonable output that is actually measuring something different than you wanted.

0.5.1 Example

Perhaps you want the mean of the first row.

```
In [33]: df = pd.DataFrame({'col1':[1, 2, 3], 'col2':[3, 4, 5]})
         df
```

```
Out[33]:   col1  col2
         0     1     3
         1     2     4
         2     3     5
```

```
In [34]: # mean column 0 = (1+2+3)/3 = 2
         df.mean()
```

```
Out[34]: col1    2.0
         col2    4.0
         dtype: float64
```

```
In [35]: # axis=0 (default) indica que é por coluna e axis=1 por linha
         df.mean(axis=0)
```

```
Out[35]: col1    2.0
         col2    4.0
         dtype: float64
```

```
In [36]: df.describe()
```

```
Out[36]:      col1  col2
count    3.0    3.0
mean     2.0    4.0
std      1.0    1.0
min      1.0    3.0
25%     1.5    3.5
50%     2.0    4.0
75%     2.5    4.5
max      3.0    5.0
```

This looks correct, but lets on a DataFrame that doesn't have the same mean for the first row and the first column.

```
In [37]: df = pd.DataFrame({'col1':[1, 2, 3], 'col2':[6, 7, 8]})
df
```

```
Out[37]:
```

	col1	col2
0	1	6
1	2	7
2	3	8

```
In [38]: df.mean()[0]
```

```
Out[38]: 2.0
```

Looks like this is actually returning the mean of the first column. Doing a simple test, we found an error that would have been much harder to spot had our DataFrame been 100,000 rows and 300 columns.

```
In [39]: # Use the argument 'axis=1' to return the means of each row in the dataframe
# Use 'axis=0' (which is the default) to return the means of each column in the dataframe
# The Pandas .mean() method returns a series which can be accessed by a bracketed index
print(df.mean(axis=1))
print("\n")
print(type(df.mean(axis=1)))
print("\n")
print(df.mean(axis=0))
print("\n")
print(df.mean(axis=1)[2])
print("\n")
print(df.mean(axis=0)[1])
```

```
0    3.5
1    4.5
2    5.5
dtype: float64
```

```
<class 'pandas.core.series.Series'>
```

```
col1    2.0
col2    7.0
dtype: float64
```

```
5.5
```

```
7.0
```

0.6 Randomness and reproducibility

In Python, we refer to randomness as the ability to generate data, strings, or, more generally, numbers at random.

However, when conducting analysis it is important to consider reproducibility. If we are creating random data, how can we enable reproducible analysis?

We do this by utilizing pseudo-random number generators (PRNGs). PRNGs start with a random number, known as the seed, and then use an algorithm to generate a psuedo-random sequence based on it.

This means that we can replicate the output of a random number generator in python simply by knowing which seed was used.

We can showcase this by using the functions in the python library random.

```
In [40]: import random
```

```
In [41]: random.seed(1234)
         random.random()
```

```
Out[41]: 0.9664535356921388
```

```
In [42]: random.seed(1234)
         random.random()
```

```
Out[42]: 0.9664535356921388
```

The random library includes standard distributions that may come in handy

```
In [43]: # Uniform distribution: Samples are uniformly distributed over the half-open interval
         #[low, high) (includes low, but excludes high).
         #In other words, any value within the given interval is equally likely to be drawn by
         random.uniform(25,50)
```

```
Out[43]: 36.01831497938382
```

```
In [44]: #it is used to return a random floating point number with normal distribution.
         mu = 0

         sigma = 1

         random.normalvariate(mu, sigma)
```

```
Out[44]: 1.8038006216944658
```

0.7 List comprehension

List comprehensions allow you to easy create lists. They follow the format:

```
my_list = [expression(i) for i in input list]
```

For example, if you wanted to plot the sin curve from $-\pi$ to π :


```
In [45]: x = np.linspace(-np.pi, np.pi, 100) # create a list of 100 equally spaced points between -pi and pi
```

```
Out [45]: array([-3.14159265, -3.07812614, -3.01465962, -2.9511931 , -2.88772658,
                -2.82426006, -2.76079354, -2.69732703, -2.63386051, -2.57039399,
                -2.50692747, -2.44346095, -2.37999443, -2.31652792, -2.2530614 ,
                -2.18959488, -2.12612836, -2.06266184, -1.99919533, -1.93572881,
                -1.87226229, -1.80879577, -1.74532925, -1.68186273, -1.61839622,
                -1.5549297 , -1.49146318, -1.42799666, -1.36453014, -1.30106362,
                -1.23759711, -1.17413059, -1.11066407, -1.04719755, -0.98373103,
                -0.92026451, -0.856798 , -0.79333148, -0.72986496, -0.66639844,
                -0.60293192, -0.53946541, -0.47599889, -0.41253237, -0.34906585,
                -0.28559933, -0.22213281, -0.1586663 , -0.09519978, -0.03173326,
                0.03173326, 0.09519978, 0.1586663 , 0.22213281, 0.28559933,
                0.34906585, 0.41253237, 0.47599889, 0.53946541, 0.60293192,
                0.66639844, 0.72986496, 0.79333148, 0.856798 , 0.92026451,
                0.98373103, 1.04719755, 1.11066407, 1.17413059, 1.23759711,
                1.30106362, 1.36453014, 1.42799666, 1.49146318, 1.5549297 ,
                1.61839622, 1.68186273, 1.74532925, 1.80879577, 1.87226229,
                1.93572881, 1.99919533, 2.06266184, 2.12612836, 2.18959488,
                2.2530614 , 2.31652792, 2.37999443, 2.44346095, 2.50692747,
                2.57039399, 2.63386051, 2.69732703, 2.76079354, 2.82426006,
                2.88772658, 2.9511931 , 3.01465962, 3.07812614, 3.14159265])
```

```
In [46]: # Here's our list comprehension. For each point in x, we want y=sin(x)
y = [np.sin(value) for value in x]
# let looks at just the first 5 elements of y
y[:5]
```

```
Out [46]: [-1.2246467991473532e-16,
            -0.06342391965656484,
            -0.12659245357374938,
            -0.1892512443604105,
            -0.2511479871810793]
```

```
In [47]: # It doesn't really matter what word you use to represent a value in the input list,
# the following will built the same y
y = [np.sin(i) for i in x]
y[:5]
```

```
Out [47]: [-1.2246467991473532e-16,
            -0.06342391965656484,
            -0.12659245357374938,
            -0.1892512443604105,
            -0.2511479871810793]
```

```
In [48]: plt.plot(x,y)
plt.show()
```

