Data Journalism

Jupyter and Python Basics

Angelica Lo Duca
angelica.loduca@iit.cnr.it

Jupyter Basics

extracted from: https://www.dataquest.io/blog/jupyter-notebook-tutorial/

Jupyter Notebook

Notebook documents (or “notebooks”, all lower case) are documents produced by the Jupyter Notebook App, which contain both **computer code** (e.g. python) and rich **text elements** (paragraph, equations, figures, links, etc…).

From Jupyter Notebook Beginner Guide

Build the first Notebook

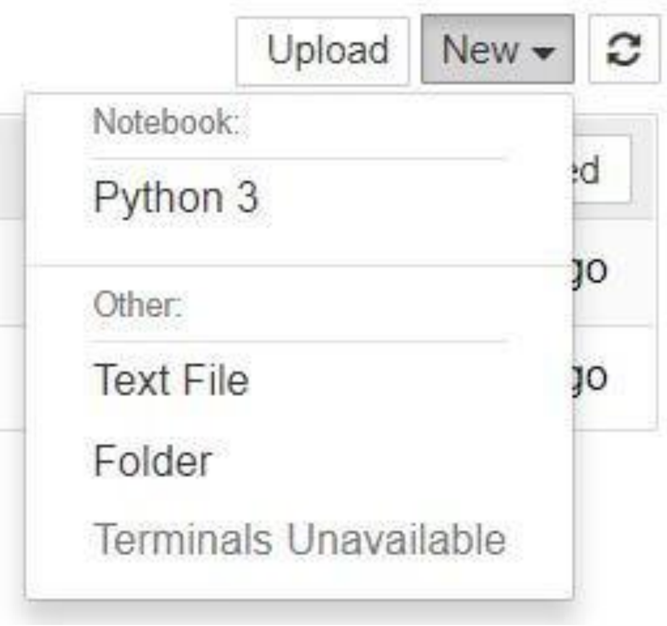
- Run Jupyter by typing on the console: **`jupyter-notebook`** or **`jupyter notebook`**
- Jupyter Notebook opens in the browser, with the URL like **`http://localhost:8888/tree`**
- An app opens

Build the first Notebook

Create the first notebook, click the “**New**” button in the top-right and select “Python 3”

The notebook opens

Its name is `Untitled.ipynb`



The Notebook Interface

- A **kernel** is a “computational engine” that executes the code contained in a notebook document.
- A **cell** is a container for text to be displayed in the notebook or code to be executed by the notebook’s kernel.
  - A code cell contains code to be executed in the kernel. When the code is run, the notebook displays the output below the code cell that generated it.
  - A Markdown cell contains text formatted using Markdown and displays its output in-place when the Markdown cell is run.

To run a cell: Ctrl + Enter / Shift + Enter

Commands

- Toggle between edit and command mode with **Esc** and **Enter**, respectively.
- Once in command mode:
  - Scroll up and down your cells with your Up and Down keys.
  - Press A or B to insert a new cell above or below the active cell.
  - M will transform the active cell to a Markdown cell.
  - Y will set the active cell to a code cell.
  - D + D (D twice) will delete the active cell.
  - Z will undo cell deletion.
  - Hold Shift and press Up or Down to select multiple cells at once. With multiple cells selected, Shift + M will merge your selection.

Data Cleaning

Python Pandas

```
pip install pandas
```

```
pip3 install pandas
```

DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object.

(Definition from https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html)

DataFrame - basic operations

```
import pandas as pd

df = pd.DataFrame() # empty dataframe

# load a csv file into a dataframe

df = pd.read_csv(‘input_file.csv’)

# show the first 10 lines of the dataframe

df.head(10)
```

Data Cleaning Definition (from Wikipedia)

Data cleansing or data cleaning is the process of **detecting and correcting (or removing) corrupt or inaccurate records** from a record set, table, or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and **then replacing, modifying, or deleting the dirty or coarse data**.

Data Cleansing involves the following aspects:

- missing values
- data formatting
- data normalization
- data standardization
- data binning
- remove duplicates

Missing Values

No data value is stored for the variable in an observation

from [Wikipedia]

Example of Missing Values

Name	Surname	Email	Count
John	Wild		5
Marc	Wales	m.wales@gmail.com	
Maria	Zack	m.zack@live.it	7
Kate	Zack	k.zack@live.it	

Identify Missing Values

In order to check whether our dataset contains missing values, we can use the function `isna()` which returns if an cell of the dataset if NaN or not.

Then we can count how many missing values there are for each column.

```
df.isna().sum()
```

```
Name        0
Surname     0
Email       1
Count       2
```

Missing Values Management

- check the source, for example by contacting the data source to correct the missing values
- drop missing values
- replace the missing value with a value
- leave the missing value as it is

Drop Missing Values

Dropping missing values can be one of the following alternatives:

- **remove rows** having missing values
- **remove the whole column** containing missing values

We can use the `dropna()` by specifying the axis to be considered.

If we set `axis = 0` we drop the entire row,

if we set `axis = 1` we drop the whole column

Examples

Original table

Name	Surname	Email	Count
John	Wild		5
Marc	Wales	m.wales@gmail.com	
Maria	Zack	m.zack@live.it	7
Kate	Zack	k.zack@live.it	

`df.dropna(axis=1)`

Name	Surname
John	Wild
Marc	Wales
Maria	Zack

`df.dropna(axis=0)`

Name	Surname	Email	Count
Maria	Zack	m.zack@live.it	7

Examples (cont.)

As an alternative, we can specify only the column on which the dropping operation must be applied.

```
df.dropna(subset=['Email'],axis=0,inplace=True)
```

Name	Surname	Email	Count
Marc	Wales	m.wales@gmail.com	
Maria	Zack	m.zack@live.it	7
Kate	Zack	k.zack@live.it	

inplace=True

We can use the argument **`inplace=True`** to store changes in the original dataframe df.

Dropping by percentage

Another alternative involves the dropping of columns where a certain percentage of not-null values is available. This can be achieved through the thresh parameter.

In the following example we keep only columns where there are at least the 75% of not null values.

```
df.dropna(thresh=0.75*len(df),axis=1,inplace=True)
```

Name	Surname	Email
John	Wild	
Marc	Wales	m.wales@gmail.com
Maria	Zack	m.zack@live.it
Kate	Zack	k.zack@live.it

Replace Missing Values

A good strategy when dealing with missing values involves their replacement with another value. Usually, the following strategies are adopted:

- for numerical values replace the missing value with the **average value** of the column
- for categorical values replace the missing value with the **most frequent** value of the column
- use other functions, such as **linear interpolation**

fillna() - numerical values

`fillna()` function replaces all the NaN values with the value passed as argument. For example, for **numerical values**, all the NaN values in the numeric columns could be replaced with the average value.

```
df[‘Count’].fillna(df[‘Count’].mean())
```

Name	Surname	Email	Count
John	Wild		5
Marc	Wales	m.wales@gmail.com	6
Maria	Zack	m.zack@live.it	7
Kate	Zack	k.zack@live.it	6

fillna() - categorical values

For categorical values, the missing values can be replaced with the most frequent value.

```
df['Car'].fillna(df['Car'].mode())
```

Car
Ferrari
Lamborghini
Ferrari

Car
Ferrari
Lamborghini
Ferrari
Ferrari

interpolate() - linear interpolation

We could replace a missing value over a column, with the interpolation between the previous and the next values.

We set Limit direction = forward so that the linear interpolation is applied starting from the first row until the last one.

Example

```
df[‘Count’] = df[‘Count’].interpolate(method ='linear',
limit_direction ='forward')
```

Count
0
4
12

Count
0
4
8
12

Data Formatting

Transforming data into a common format, which helps users to perform comparisons.

Not formatted table

City	Value
New York	3
Chicago	5
N.Y.	6
New York (USA)	7
Chicago (U.S.A.)	3

Data Formatting

- transform data in the correct format
- make data homogeneous
- use a single value to represent the same concept

Correct Format

Make sure that every column is assigned to the correct data type.

This can be checked through the property `dtypes`.

Example

`df.dtypes`

Name	Value
John	3.2999
Mary	2.3

```
Name    string
Value   float64
```

Correct Data Types

```
Name    string
Value   int64
```

Wrong Data Type

astype()

We can convert the column Value to int64 by using the function **`astype()`**

```
df['Value'] = df['Value'].astype('float64')
```

The `astype()` function supports all datatypes described at this link.

Make data homogeneous - categorical data

Categorical data should have all the same formatting style:

- lower case
  - **`df['Name'] = df['Name'].str.lower()`**
- remove white space everywhere:
  - **`df['Name']=df['Name'].str.replace(‘ ‘, ‘’)`**
- remove white space at the beginning of string:
  - **`df['Name']=df['Name'].str.lstrip()`**
- remove white space at the end of string:
  - **`df['Name']=df['Name'].str.rstrip()`**
- remove white space at both ends:
  - **`df['Name']=df['Name'].str.strip()`**

Make data homogeneous - numeric data

Numeric data should have for example the same number of digits after the point.

- Round to specific decimal places
  - `df['Value'] = df['Value'].round(2) # 2 decimal points`
- Round up – Single DataFrame column
  - `df['Value'] = df['Value'].apply(np.ceil)`
- Round down – Single DataFrame column
  - `df['Value'] = df['Value'].apply(np.floor)`

Single Value for the same concept

We can use the **`unique()`** function to list all the values of a column.

City	Value
New York	3
Chicago	5
N.Y.	6
New York (USA)	7
Chicago	3

```
df['City'].unique()
```

```
[ 'New York',
'Chicago', 'N.Y.',
'New York (USA)' ]
```

set_pattern()

We must manage each *issue* separately.

We define a function, called **`set_pattern()`** which receives as input a cell and manipulates it according to our needs.

Put here all the values which must be represented by the same value

```
import re
def set_pattern(x):
    pattern = "(?=New York \(USA\)|N.Y.)\\w+"
    res = re.match(pattern, x)
    if res:
        x = x.replace(x, 'New York')
    return x
```

set_pattern() - cont.

Now we can apply the function the specific column:

```
df['City'] = df['City'].apply(lambda x: set_pattern(x))
```

City	Value
New York	3
Chicago	5
New York	6
New York	7
Chicago	3

Data Normalisation

Adjusting values measured in different scales to a common scale. Normalization applies only to columns containing numeric values.

Techniques for Normalisation

- single feature scaling
- min max
- z-score
- log scaling
- clipping

Single Feature Scaling

Single Feature Scaling converts every value of a column into a number between 0 and 1.

The new value is calculated as the current value divided by the max value of the column.

Example

```
df['Value'] = df['Value']/df['Value'].max()
```

Value
1
3
4

MAX

Value
0.25
0.75
1

Min Max

Min Max converts every value of a column into a number between 0 and 1.

The new value is calculated as the difference between the current value and the min value, divided by the range of the column values.

Example

```
df['Value'] = (df['Value']-df['Value'].min()) /
    (df['Value'].max()-df['Value'].min())
```

	Value
MIN	1
	3
MAX	4

Value
0
0.67
1

z-score

Z-Score converts every value of a column into a number around 0.

Typical values obtained by a z-score transformation range from -3 and 3.

The new value is calculated as the difference between the current value and the average value, divided by the standard deviation.

Example

```
df['Value'] = (df['Value']-df['Value'].mean()) /
df['Value'].std()
```

Value
1
3
4

Value
-1.34
0.26
1.07

MEAN: 2.66 STD: 1.25

Log scaling

Log Scaling involves the conversion of a column to the logarithmic scale.

If we want to use the natural logarithm, we can use the `log()` function of the `numpy` library.

We must deal with log(0) because it does not exist

Example

```
df['Value'] = df['Value'].apply(lambda x: np.log(x) if x != 0 else 0)
```

Value
1
3
4

Value
0
1.09
1.39

Clipping

Clipping involves the capping of all values below or above a certain value. Clipping is useful when a column contains some outliers.

We can set a maximum vmax and a minimum value vmin and set all outliers greater than the maximum value to vmax and all the outliers lower than the minimum value to vmin.

Example

```
vmax = 35

vmin = 2

df['Value'] = df['Value'].apply(lambda x: vmax if x > vmax
else vmin if x < vmin else x)
```

Value
1
30
40

Value
2
30
35

Data Standardization

Standardization transforms data to have a mean of zero and a standard deviation of 1.

Techniques for standardization

- z-score
- z-map

z-score

The new value is calculated as the difference between the current value and the average value, divided by the standard deviation.

We can use the `zscore()` function of the `scipy.stats` library.

Example

```
from scipy.stats import zscore
df['Value'] = zscore(df['Value'])
```

Value
1
3
4

→

Value
-1.34
0.26
1.07

MEAN: 2.66 STD: 1.25

z-map

The new value is calculated as the difference between the current value and the average value of a comparison array, divided by the standard deviation of a comparison array.

We can use the `zmap()` function of the `scipy.stats` library.

Example

```
from scipy.stats import zmap
df['Value'] = zmap(df['Value'], df['Count'])
```

Value	Count
1	3
3	4
4	5

→

Value	Count
-3.67	3
-1.22	4
0	5

Data Binning

Data binning (or bucketing) groups data in bins (or buckets), in the sense that it replaces values contained into a small interval with a single representative value for that interval.

Binning

Binning can be applied to convert numeric values to categorical or to sample (quantize) numeric values.

Binning is a technique for data smoothing. Data smoothing is employed to remove noise from data.

Techniques for binning

- convert numeric to categorical
  - binning by distance
  - binning by frequency

Binning by distance - cut()

- Define the bin edges
- Convert numeric into categorical variables
- Define the number of bins and the associated labels

Size
1000
5
500
100
250
400

bins = 4

Label	Ranges
small	0-50
medium	51-100
large	101-500
very large	> 500

Size
very large
small
large
medium
large
large

Example

```
import numpy as np

bins = [ 0, 50, 100, 500, 1000 ]

labels = ['small', 'medium', 'large','very large']

df['Size'] = pd.cut(df['Size'] , bins=bins, labels=labels,
include_lowest=True)
```

Example 2 - Linear Space among ranges

```
min_value = df[‘Size’].min()

max_value = df[‘Size’].max()

n_bins = 4

bins = np.linspace(min_value,max_value,n_bins+1)

array([   5.  ,  336.66666667,  668.33333333, 1000.  ])

labels = ['small', 'medium', 'large','very large']

df['Size'] = pd.cut(df['Size'] , bins=bins, labels=labels,
include_lowest=True)
```

Example 2 (cont.)

Size
1000
5
500
100
250
400

bins = 4

Label	Ranges
small	0 - 5
medium	5 - 336.67
large	336.67-668.33
very large	668.33 - 1000

Size
very large
small
medium
small
small
medium

Binning by frequency - qcut()

- Quantile-based discretization function
- Calculate the size of each bin so that each bin contains (almost) the same number of observations, but the bin range will vary.

Example

Size
1000
5
500
100
250
400
10
30

bins = 4
2 observations for each bin

Label
small
medium
large
very large

Size
very large
small
very large
medium
large
large
small
medium

Example (cont.)

```
labels = ['small', 'medium', 'large','very large']

n_bins = 4

df['Size'] = pd.qcut(df['Size'], q=n_bins,precision=1,
labels=labels)
```

We can set the `precision` parameter to define the number of decimal points.

Remove Duplicates

Remove all rows that appear at least twice.

The concept of duplicate

	Name	Surname	Value
1	Mark	Grenn	3
2	Mark	Grenn	3
3	Mark	Grenn	4

Rows 1 and 2 are duplicates

Rows 1, 2 and 3 are duplicates in column Name and Surname

Drop duplicates on the basis of all columns

keep just one row for each duplicate

Name	Surname	Value
Mark	Grenn	3
Mark	Grenn	4

Do not maintain any row for the duplicate

Name	Surname	Value
Mark	Grenn	4

Drop duplicates on the basis of the Name and Surname Columns

Keep just one value for column

Name	Surname	Value
Mark	Grenn	3

Do not maintain any row for the duplicate

Name	Surname	Value

drop_duplicates()

```
df1 = df.drop_duplicates()

df2 = df.drop_duplicates(keep=False)

df3 = df.drop_duplicates(subset=["Name", "Surname"])

df4 = df.drop_duplicates(subset=["Name", "Surname"],
keep=False)
```