Data manipulation: Time Series & Geographical visualization

AAA-Python Edition

Plan

- 1- Date and time types
- 2- Time series basics
- 3- Date ranges, frequencies and shifting
- 4- Timezone
- 5- Periods and resampling
- 6- Rolling and expanding
- 7- Geographical visualization with basemap

1- Date and time types

Datetime module

```
# importing datetime module,
import datetime as dt
```

```
# datetime type
# to have the current time
now =dt.datetime.now()
```

year month day minute second

```
datetime.datetime(2018, 10, 27, 11, 37, 0, 875665)
```

hour microsecond

```
# to access the year
now.year
```

→ 2018

```
# datetime.timedelta type
diff = dt.datetime.now() - dt.datetime(2018,10,26)
```

```
datetime.timedelta(1, 41841, 327812)
```

```
#  difference in days
diff.days
```

→ 1

```
# timedelta type
# operations with timedelta
dt.datetime.now()+dt.timedelta(2)*2
```

```
datetime.datetime(2018, 10, 31, 11, 39, 23, 519331)
```

```
# string conversions
dateS = dt.datetime(2018,11,5)
# from datetime to string using str
str(dateS)
```

```
'2018-11-05 00:00:00'
```

```
# from datetime to string using strftime
dt.datetime.now().strftime("%Y/%m/%d, %H:%M")
```

```
'2018/10/27, 11:54'
```

```
# from string to datetime
from dateutil.parser import parse
parse("2018/10/27, 13h05")
```

```
datetime.datetime(2018, 10, 27, 13, 5)
```

2- Time series basics

Indexing

```
# indexing series with datetime  indexes
ser1 = S(range(3),index=[dt.datetime(2018,6,1),dt.datetime(2018,6,3),dt.datetime(2018,6,2)] )
```

ser1

```
2018-06-01    0
2018-06-03    1
2018-06-02    2
dtype: int64
```

```
# DatetimeIndex type
ser1.index
```

```
DatetimeIndex(['2018-06-01', '2018-06-03', '2018-06-02'], dtype='datetime64[ns]', freq=None)
```

ser2

```
2018-06-03    0
2018-06-04    1
2018-06-02    2
dtype: int64
```

The values to be added must have the same index.

```
# adding series values is done by aligning dates in the index
ser1 +ser2
```

ser2 + ser2

```
2018-06-01    NaN
2018-06-02    4.0
2018-06-03    1.0
2018-06-04    NaN
dtype: float64
```

If one of the indexes values is missing from one of the two series, the corresponding addition result will be a “**Nan**” value.

[By Amina Delali]

Selecting

```
# datetimeindex values type
times= ser1.index[1]
```

```
Timestamp('2018-06-03 00:00:00')
```

ser1

```
2018-06-01    0
2018-06-03    1
2018-06-02    2
dtype: int64
```

```
# selecting a value by a timestamp

ser1[times]
```

```
1
```

```
# selecting a value by a datetime value
ser1[dt.datetime(2018,6,3)]
```

Selecting the same value

```
# selecting a value (or a slice) by a string: a date
ser1["6/3/2018"]
```

```
2018-06-03    1
dtype: int64
```

```
# selecting a value ( or a slice)by a string: a date
ser1["20180603"]
```

```
# selecting a value (or a slice )by a string: a year
ser1["2018"]
```

```
2018-06-01    0
2018-06-03    1
2018-06-02    2
dtype: int64
```

The order of month, year, day is the same: year-month-day (from left to right)

```
# selecting a slice  by 2 strings
ser1["20180602 ":"20180603 "]
```

```
2018-06-03    1
2018-06-02    2
dtype: int64
```

included

The order is different from previous one: month-day-year (from left to right)

[By Amina Delali]

3- Date ranges, frequencies and shifting

Date ranges

```
# creating dateitmeindex using a date range
dr = pd.date_range("2018-10-05","2018-10-10")
```

included

included

By default, the frequency is “one day”

```
DatetimeIndex(['2018-10-05', '2018-10-06', '2018-10-07', '2018-10-08',
               '2018-10-09', '2018-10-10'],
              dtype='datetime64[ns]', freq='D')
```

```
# creating the same datetimeindex differently
# with start and period key argument
dr = pd.date_range(start="2018-10-05",periods= 6)
```

included

```
# creating the same datetimeindex differently
# with start and period key argument
dr = pd.date_range(end="2018-10-10",periods= 6)
```

+ 1 hour

Frequencies (using date_range)

```
# creating a datetimeindex  with an hour frequency
dr2 = pd.date_range(start="2018-10-05",periods =6,freq="H")
```

By default, time starts at **00:00:00**

```
DatetimeIndex(['2018-10-05 00:00:00', '2018-10-05 01:00:00',
               '2018-10-05 02:00:00', '2018-10-05 03:00:00',
               '2018-10-05 04:00:00', '2018-10-05 05:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Frequencies (using resample)

```
# resample the series by a 12 hours frequency
ser2.resample("12h").mean()
```

New values

```
2018-06-02 00:00:00    2.0
2018-06-02 12:00:00    NaN
2018-06-03 00:00:00    0.0
2018-06-03 12:00:00    NaN
2018-06-04 00:00:00    1.0
Freq: 12H, dtype: float64
```

[By Amina Delali]

3- Date ranges, frequencies and shifting

Frequencies (using date_range)

```
# creating a datetimeindex  with 3 hours frequency
dr3 = pd.date_range(start="2018-10-05-15-00",periods =6,freq="3h")
```

```
DatetimeIndex(['2018-10-05 15:00:00+00:00', '2018-10-05 18:00:00+00:00',
               '2018-10-05 21:00:00+00:00', '2018-10-06 00:00:00+00:00',
               '2018-10-06 03:00:00+00:00', '2018-10-06 06:00:00+00:00'],
              dtype='datetime64[ns, tzlocal()]', freq='3H')
```

+ 3 hours

```
# creating a datetimeindex  with 30 minutes frequency
dr4 = pd.date_range(start="2018-10-05-15-00",periods =6,freq="30min")
```

+ 30 minutes

```
DatetimeIndex(['2018-10-05 15:00:00+00:00', '2018-10-05 15:30:00+00:00',
               '2018-10-05 16:00:00+00:00', '2018-10-05 16:30:00+00:00',
               '2018-10-05 17:00:00+00:00', '2018-10-05 17:30:00+00:00'],
              dtype='datetime64[ns, tzlocal()]', freq='30T')
```

```
# creating a datetimeindex  with month begin frequency
dr5 = pd.date_range(start="2018-10-05",periods =6,freq="MS")
```

First day of the month

Not included

```
DatetimeIndex(['2018-11-01', '2018-12-01', '2019-01-01', '2019-02-01',
               '2019-03-01', '2019-04-01'],
              dtype='datetime64[ns]', freq='MS')
```

3- Date ranges, frequencies and shifting

shifting

```
# shifting values without shifting indexes
ser1.shift(1)
```

The index didn't change

```
2018-06-01    NaN
2018-06-03    0.0
2018-06-02    1.0
dtype: float64
```

ser1

```
2018-06-01    0
2018-06-03    1
2018-06-02    2
dtype: int64
```

```
# shifting values with indexes
ser1.shift(1,freq="D")
```

The index will change

```
2018-06-02    0
2018-06-04    1
2018-06-03    2
dtype: int64
```

```
from pandas.tseries.offsets import Day, Week
from datetime import date

#shift using time offsets
now = dt.datetime.now()

print(now.strftime("Today = %Y/%m/%d"))
# timestamp created by shifting by 1 day from now
tomorrow = now +Day()
print(tomorrow.strftime("Tomorrow = %Y/%m/%d"))
```

offsets

```
Today = 2018/10/27
Tomorrow = 2018/10/28
```

```
# shift to the next week using rollforward
Week().rollforward(now).strftime("Next week = %Y/%m/%d")
```

```
'Next week = 2018/11/03'
```

```
# shift to the previous week using rollback
Week().rollback(now).strftime("Previous week = %Y/%m/%d")
```

```
'Previous week = 2018/10/20'
```


[By Amina Delali]

4- Timezone

Timezone conversions

```
# from a naive (without time zone) serieses to a localized  one
print("ser1 timezone:",ser1.index.tz)
ser3=ser1.tz_localize("Africa/Algiers")
ser3.index.tz
```

```
ser1 timezone: None
<DstTzInfo 'Africa/Algiers' LMT+0:12:00 STD>
```

ser1

```
2018-06-01    0
2018-06-03    1
2018-06-02    2
dtype: int64
```

ser3

```
2018-06-01 00:00:00+01:00    0
2018-06-03 00:00:00+01:00    1
2018-06-02 00:00:00+01:00    2
dtype: int64
```

```
# from a localized serieses to another time zone series
print("ser3 timezone:",ser3.index.tz)
ser4= ser3.tz_convert("America/New_York")
ser4.index.tz
```

```
ser3 timezone: Africa/Algiers
<DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STD>
```

ser3

```
2018-06-01 00:00:00+01:00    0
2018-06-03 00:00:00+01:00    1
2018-06-02 00:00:00+01:00    2
dtype: int64
```

ser4

```
2018-05-31 19:00:00-04:00    0
2018-06-02 19:00:00-04:00    1
2018-06-01 19:00:00-04:00    2
dtype: int64
```

```
# additionning 2 series with different time zones
# the result will be in UTC time zone
(ser3 + ser4).index.tz
```

<UTC>

```
ser3 +ser4
```

```
2018-05-31 23:00:00+00:00    0
2018-06-02 23:00:00+00:00    2
2018-06-01 23:00:00+00:00    4
dtype: int64
```

```
# generating a localized date_range
ser5 = S(np.random.rand(1),index= pd.date_range(start= "2018-10-27",periods=1,tz="Africa/Algiers"))
```

```
2018-10-27 00:00:00+01:00    0.041752
Freq: D, dtype: float64
```

[By Amina Delali]

5- Periods and resampling

Periods

```
# Periods represent durations in time.
# aPer will represent a duration equal to a month
aPer = pd.Period("January 2018",freq="M")
```

```
Period('2018-01', 'M')
```

```
# adding values to a pariod will shift the period to that value * frequency
aPer + 5
```

```
Period('2018-06', 'M')
```

```
PeriodIndex(['2018-01', '2018-02', '2018-03', '2018-04', '2018-05', '2018-06',
             '2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-12'],
            dtype='period[M]', freq='M')
```

```
# period_range will create a range of periods and also a PeriodIndex
aPerR = pd.period_range("January 2018","December 2018",freq="M")
```

```
# period indexes can be used as Series indexes
# (they can also be creatted using pd.PeriodIndex)
ser6 = S(range(12),index=aPerR)
```

```
2018-01     0
2018-02     1
2018-03     2
2018-04     3
2018-05     4
2018-06     5
2018-07     6
2018-08     7
2018-09     8
2018-10     9
2018-11    10
2018-12    11
Freq: M, dtype: int64
```


[By Amina Delali]

5- Periods and resampling

Periods conversions

```
PeriodIndex(['2018-01', '2018-02', '2018-03', '2018-04', '2018-05', '2018-06',
             '2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-12'],
            dtype='period[M]', freq='M')
```

They didn't change

Before

Used to convert frequencies

```
# we can convert periods from frequency to another
aPerR.asfreq("D",how="start")
```

After

```
PeriodIndex(['2018-01-01', '2018-02-01', '2018-03-01', '2018-04-01',
             '2018-05-01', '2018-06-01', '2018-07-01', '2018-08-01',
             '2018-09-01', '2018-10-01', '2018-11-01', '2018-12-01'],
            dtype='period[D]', freq='D')
```

A day value added (but the temporal distance between the two periods remains the same)

```
# time series indexed by timestamps (datetime indexes) can be
# converted to series indexed by periodindexes
ser1.to_period("M")
```

Duplicated values

```
2018-06    0
2018-06    1
2018-06    2
Freq: M, dtype: int64
```

[By Amina Delali]

5- Periods and resampling

Resampling

```
# resampling can be done in 2 directions
# from hight to low frequency = downsampling
# ( from shorter to longer time duration)
ser6.resample("A-DEC").sum()
```

```
2018    66
Freq: A-DEC, dtype: int64
```

From 1 January to 31 December

In this case we can do aggregations: the short periods fall into longer ones

```
# from low to high frequency = upsampling
# ( from longer to shorter time duration)
ser6.resample("D").asfreq()
```

```
2018-01-01    0.0
2018-01-02    NaN
2018-01-03    NaN
2018-01-04    NaN
2018-01-05    NaN
              ...
2018-12-31    NaN
Freq: D, Length: 365, dtype: float64
```

```
ser6.resample("D").asfreq().dropna()
```

```
2018-01-01     0.0
2018-02-01     1.0
2018-03-01     2.0
2018-04-01     3.0
2018-05-01     4.0
2018-06-01     5.0
2018-07-01     6.0
2018-08-01     7.0
2018-09-01     8.0
2018-10-01     9.0
2018-11-01    10.0
2018-12-01    11.0
Freq: D, dtype: float64
```

No need for aggregation

Only the first day of each month has defined values (corresponding to the month value of the previous series. All the other values are **Nan**

[By Amina Delali]

6- Rolling and expanding

rolling

ser7

```
2018-01-01    0
2018-01-02    1
2018-01-03    2
2018-01-04    3
2018-01-05    4
2018-01-06    5
2018-01-07    6
2018-01-08    7
2018-01-09    8
2018-01-10    9
Freq: D, dtype: int64
```

```
# creating ser7 series using a period index
ser7 = S(range(10),index=pd.period_range("20180101",periods=10))
```

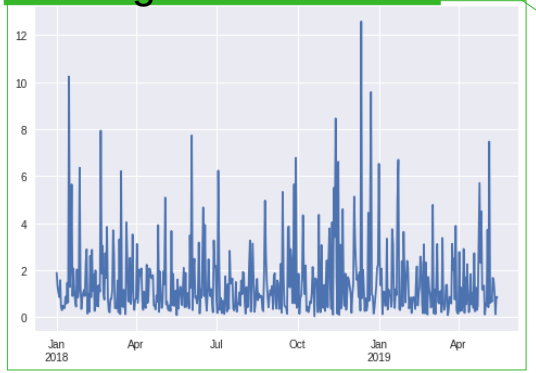
```
# rolling.mean() (in this case)will calculate the mean
# of each 2 consecutive values
ser7.rolling(2).mean()
```

```
2018-01-01    NaN
2018-01-02    0.5
2018-01-03    1.5
2018-01-04    2.5
2018-01-05    3.5
2018-01-06    4.5
2018-01-07    5.5
2018-01-08    6.5
2018-01-09    7.5
2018-01-10    8.5
Freq: D, dtype: float64
```

== (2+3)/2

== (6+7)/11

Plotting of ser8 Series



Plotting of ser8.rolling(50).mean()


```
# rolling.mean in this case will calculate
# the mean of each 50 consecutive values
ser8.rolling(50).mean().plot()
```

[By Amina Delali]

6- Rolling and expanding

expanding

```
#  expanding.mean (in this case )wil caclulate
# the mean of 1, then 2, then 3, ..., until 10 consecutive values
ser7.expanding().mean()
```

ser7

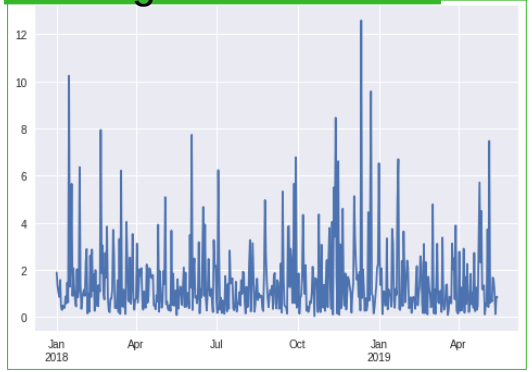
```
2018-01-01    0
2018-01-02    1
2018-01-03    2
2018-01-04    3
2018-01-05    4
2018-01-06    5
2018-01-07    6
2018-01-08    7
2018-01-09    8
2018-01-10    9
Freq: D, dtype: int64
```

==(0+1+2)/3

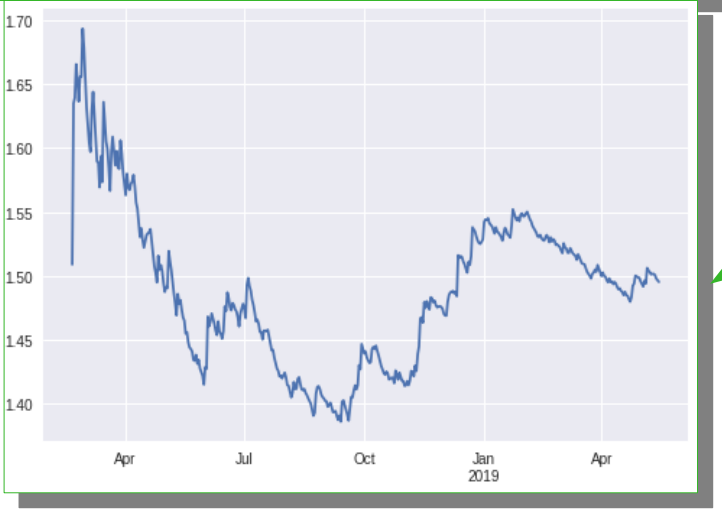
==(0+1+2+3+4+5+6+7)/8

```
2018-01-01    0.0
2018-01-02    0.5
2018-01-03    1.0
2018-01-04    1.5
2018-01-05    2.0
2018-01-06    2.5
2018-01-07    3.0
2018-01-08    3.5
2018-01-09    4.0
2018-01-10    4.5
Freq: D, dtype: float64
```

Plotting of ser8 Series



Plotting of ser8.expanding().mean()




```
# apply expanding.mean to the ser8 series
ser8.expanding(50).mean().plot()
```

[By Amina Delali]

7- Geographical Visualization with basemap

Installation

- To install **basemap** in google colab, you have to:
  - Install the following libraries:
    - libproj-dev, proj-data, proj-bin

```
!apt-get install libproj-dev proj-data proj-bin
```

    - Libgeos-dev

```
!apt-get install libgeos-dev
```

  - Finally, install **basemap** from a GitHub repository:

```
!pip install https://github.com/matplotlib/basemap/archive/v1.1.0.tar.gz
```

- The import it from **matplotlib toolkit:**

```
# necessary imports
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
```

Usage

- Before projecting using **basemap**, you have to select the **projection** to use.
- In this example, we will use an "**orthographic projection**" (it shows half the globe at a time.

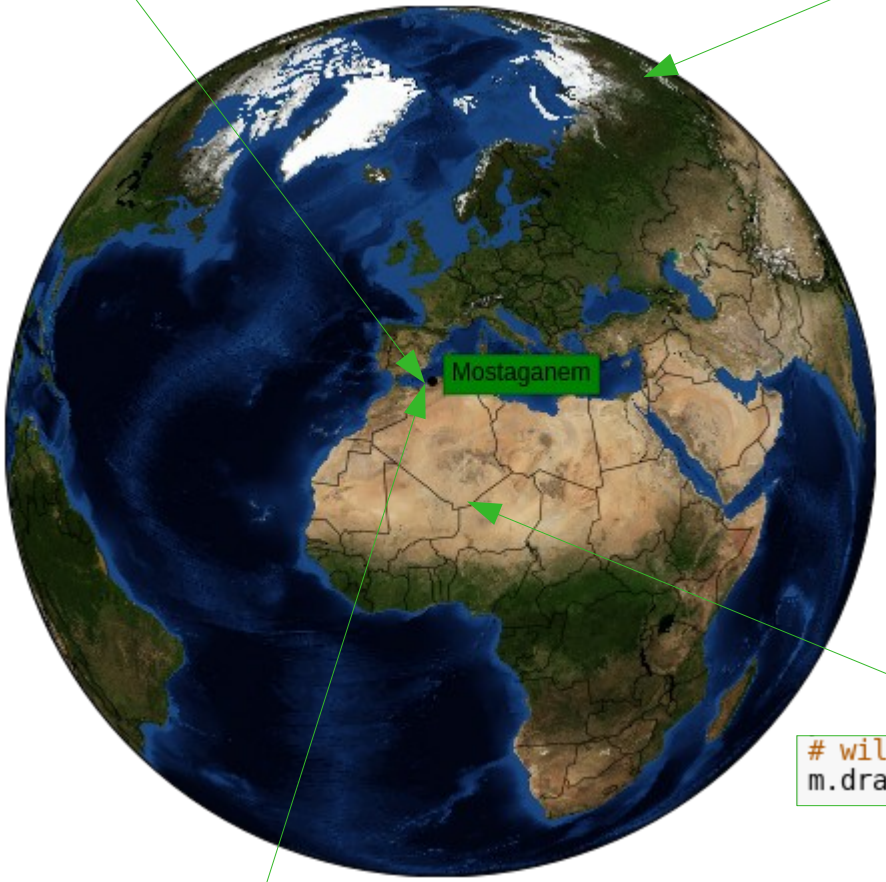
```
# you can specify your own country center lattitude and longitude coordinates (in decimal degrees)
country_lat=28.0339
country_long=1.6596
# you can specify longitude and latitude for your own city (in decimal degrees)
city_lon =0.1401
city_lat = 36.0131

# a figure with a defined size
plt.figure(figsize=(8, 8))
# selecting orthographic projection
m = Basemap(projection='ortho',lat_0=country_lat, lon_0=country_long)
# display the blumarble image as map background
m.bluemarble();
# convert the geographic coordinates to projection coordinates
x, y = m(city_lon,  city_lat)
#plotting the mark corresponding to the city coordinates
plt.plot(x, y, 'ok', markersize=5)
# plotting a text showing the name of the city
plt.text(x+300000, y,"Mostaganem", bbox=dict(facecolor="green"),fontsize=12,color="black");
# will draw political country bondaries
m.drawcountries()
```

7- Geographical Visualization with basemap

Usage

```
# you can specify longitude and latitude for your own city (in decimal degrees)
city_lon =0.1401
city_lat = 36.0131
```

```
# display the blumarble image as map background
m.bluemarble();
```




```
# will draw political country bondaries
m.drawcountries()
```

```
#plotting the mark corresponding to the city coordinates
plt.plot(x, y, 'ok', markersize=5)
```

7- Geographical Visualization with basemap

Plotting

The data used for plotting

```
cities = pd.read_csv('AAA-Ped-Week3/A3P-w3-dz.csv')
# display only the first row
cities.head(1)
```

	city	lat	lng	country	iso2	admin	capital	population	population_proper
0	Algiers	36.763056	3.050556	Algeria	DZ	Alger	primary	3354000.0	1977663.0

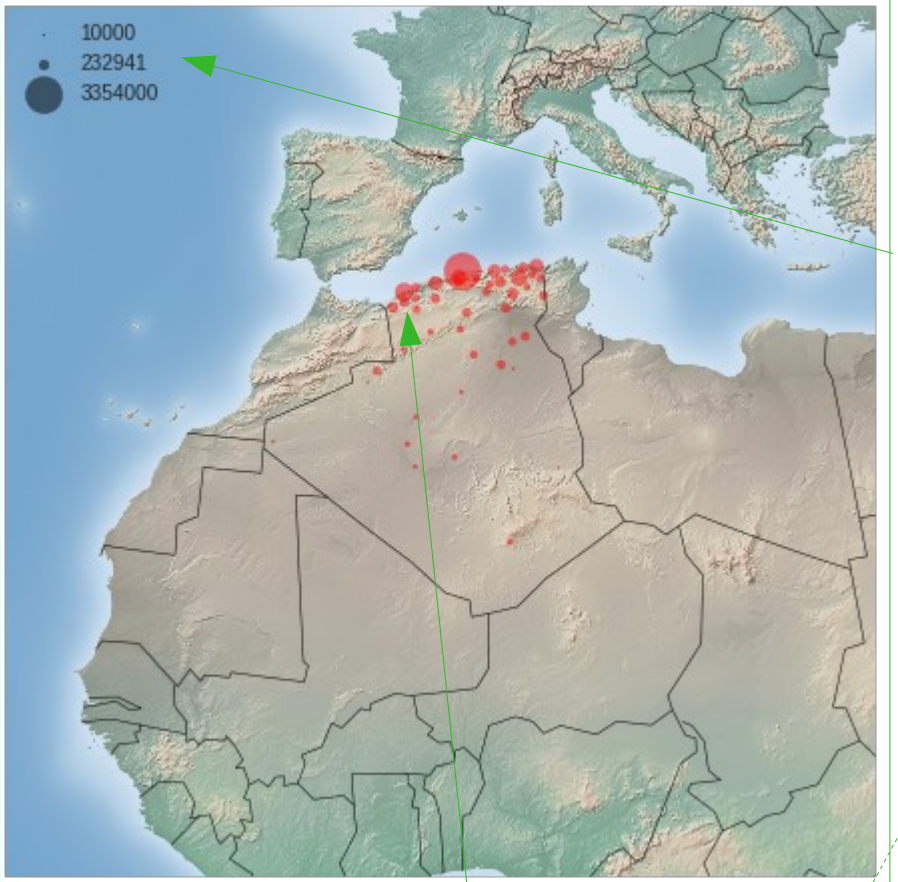
An other projection was used: The Lambert conformal conic projection

```
# extract latitues and longitues values from corresponding columns
lat = cities['lat'].values
lon = cities['lng'].values
# extract the population values
population = cities['population'].values

fig = plt.figure(figsize=(8, 8))
# use of , you have to specify the width and the height( in projections units : meters)
# or specify the four corner coordinates
m = Basemap(projection='lcc',lat_0=country_lat, lon_0=country_long,width=5E6,height=5E6)
# Project a shaded relief image onto the map
m.shadedrelief()
m.drawcountries(color='black')
# scatter city data, with size reflecting population
m.scatter(lon, lat, latlon=True,s= population/10000,c="red",alpha=0.5 )

# selecting 3 values: 10000, mean, maximum of population values as legend
legend_values= [10000,int(cities.population.mean()),int(cities.population.max())]
```


[By Amina Delali]

7- Geographical Visualization with basemap

Plotting



A shaded relief image projected on the map

```
12 m.shadedrelief()
```

```
# plotting the legend
for a in legend_values:
 plt.scatter([], [], c='k', alpha=0.5, s=a/10000,label=str(a) )

plt.legend(loc='upper left');
```

The lon, lat parameter represent the actual longitudes, latitudes coordinates

The sizes will be calculated from population size column

```
14 # scatter city data, with size reflecting population
15 m.scatter(lon, lat, latlon=True,s= population/10000,c="red",alpha=0.5 )
```

[By Amina Delali]

References

- Google Colab. Basemap-install. On-line at https://colab.research.google.com/drive/1_Xw_MEIriI0leP-v8vlmhUj6BLJLKmoV. Accessed on 31-10-2018.
- Wes McKinney. Python for data analysis: Data wrangling with Pandas,NumPy, and IPython. O’Reilly Media, Inc, 2018.
- simplemaps. Algeria cities database. On-line at https://simplemaps.com/data/dz-cities. Accessed on 31-10-2018.
- Jake VanderPlas. Python data science handbook: essential tools for working with data. O’Reilly Media, Inc, 2017.

Thank you!

FOR ALL YOUR TIME