Quantitative methods in finance - beginner python exercise with DataFrames

Eric Vansteenberghe

October 17, 2017

Contents

1 Work with panda DataFrame 3
1.1 Import the package 3
1.2 Build a DataFrame 3
1.3 Indexing with time 3
1.4 Renaming the column and plotting 3
1.5 Plotting 4
1.6 Computing the monthly changes of the population 4
1.7 Importing the full data set from INSEE 4
1.8 Indicating where your file is located 4
1.8.1 Frequently Asked Question 5
1.9 Import the csv file 5
1.10 Cleaning the imported data set 5
1.10.1 Reverse the row order 6
1.10.2 Rename the columns 6
1.10.3 Reset the index 6
1.10.4 Delete a column 6
1.11 Try to plot the DataFrame 6
1.12 Resample our DataFrame (from monthly to yearly observations) 7
1.13 Compute monthly changes 7
1.13.1 Descriptive statistics 7
1.14 Plot some previsions 8
1.14.1 Take the seasonality into account 8
1.14.2 Show a "confidence" interval for your forecast 8

2 Finding location of an element in a DataFrame 10
2.1 Selection By Label 10
2.2 Selection by position (integer) 10
2.3 Other selection methods 11
2.4 Unstack 11
2.5 Loop through a DataFrame 11

3 Correlation, regression, VAR 13
3.1 Correlation 13
3.2 Before regressing 14

3.3 Cointegration test 15
3.4 Linear regression 15
3.4.1 R-square and t-student "interpretation" 15
3.4.2 Visuals 16
3.5 VAR model 17
3.5.1 Get the data 17
3.5.2 Preparing the VAR model 17
3.6 A VAR model that "works" 18
3.6.1 Import the data 18
3.6.2 Verify that the series are stationary 18
3.6.3 Select the lag order 19
3.6.4 Estimate the VAR coefficients 19
3.7 A rolling regression 19
3.8 Rice price and world supply 20

Link to the source codes and the data sets used in this lecture

1 Work with panda DataFrame

1.1 Import the package

First we want to import the package panda.

```
import pandas as pd
```

1.2 Build a DataFrame

We propose to build a DataFrame of the evolution of the French population as taken from INSEE website: here.
We want to create the following DataFrame:

Septembre	2016	66 790
Aout	2016	66 763
Juillet	2016	66 735
Juin	2016	66 710
Mai	2016	66 688
Avril	2016	66 672
Mars	2016	66 659
Fevrier	2016	66 644
Janvier	2016	66 628

We can create a DataFrame with the population values:

```
pop=pd.DataFrame([66790,66763,66735,66710,66688,66672,66659,66644,66628])
```

Look carefully at the way your DataFrame is indexed, it starts from 0 and not from 1. Therefore here you have 9 observations indexed from 0 to 8.
Oups, we entered the data in the wrong order, we need to reverse:

```
pop=pop.iloc[::-1]
```

1.3 Indexing with time

This is optional, but you might want to index your DataFrame with time. First you want to create an object with the dates as 'year-month' from 2016-01 to 2016-09.

```
dates = pd.date_range('2016-01', '2016-10', freq='M')
```

Note that we have to range until 2016-10 for the last item of our dates object to be 2016-09.
Now we modify the index of our DataFrame pop:

```
pop.index=dates
```

1.4 Renaming the column and plotting

We see that the name of our column is '0' we want to rename it with 'population':

```
pop.columns=['Population']
```

1.5 Plotting

We can have a look at the data by plotting it and choosing a title:

```
pop.plot(title='French␣population␣in␣thousands')
```

1.6 Computing the monthly changes of the population

We want to compute the change of the French population from month m to month $m+1$:

$$\text{change}_{m+1} = \frac{\text{pop}_{m+1} - \text{pop}_m}{\text{pop}_m}$$

In visual terms, that will be:

pop	**pop.shift(1)**	**change_pop**
pop_m		NaN
pop_{m+1}	pop_m	$\frac{\text{pop}_{m+1}}{\text{pop}_m} - 1$
pop_{m+2}	pop_{m+1}	$\frac{\text{pop}_{m+2}}{\text{pop}_{m+1}} - 1$
pop_{m+3}	pop_{m+2}	$\frac{\text{pop}_{m+3}}{\text{pop}_{m+2}} - 1$
...	...	...

This operation is only one line of code:

```
change_pop=pop/pop.shift(1)-1
```

1.7 Importing the full data set from INSEE

From the INSEE website here you can choose to import the table as a csv file. It will be called 'valeurs.zip'.
We prefer to work with comma-separated values file, for some of the reasons described here.
First unzip the file to have it a Valeurs.csv
Copy the Valeurs.csv file into your usual R_data folder.

1.8 Indicating where your file is located

Now you need to 'tell' python where you file with the values, the file Valeurs.csv is located.
Import the package Miscellaneous operating system interfaces:

```
import os
```

Then you need to tell him where you R_data folder is located, in my computer, that would be:

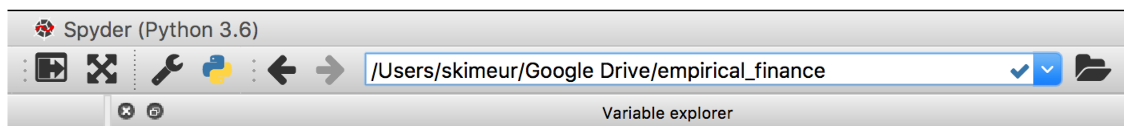
```
os.chdir('/Users/skimeur/Google␣Drive/empirical_finance/R_data')
```

1.8.1 Frequently Asked Question

If you are note sure where your file is located, here is a step-by-step guide:

1. make sure you downloaded the `R_data` zip file from the lecture website, you can access it here
2. you want to unzip the `R_data` folder and place it in the folder where you have your python code for this lecture
3. if you are not sure how to write the path to your `R_data` folder, you have a function (inspired from here) to search for your file
4. once you have your location, you can use it to change your working directory

Another way to approach it is by using the Spyder workin directory browser until you find the folder where your Valeurs.csv file is located:



1.9 Import the csv file

If you open your file with a text editor, it looks like this:

LibellĂČĖĘ;;DĂČĖĘmographie – Population au...
IdBank;;001641607
AnnĂČĖĘe;Mois
2016;9;66 790
2016;8;66 763
2016;7;66 735
2016;6;66 710

So because we are working with French data:

- they separate variables with ; instead of ,
- they use some special character as à

Hence we need to tell that:

- the values are separated by ';'
- it is encoded in latin1 style
- the frist two rows are not relevant for our study, so skip them

df=pd.read_csv('Valeurs.csv',sep=';',encoding='latin1',skiprows = [0,1])

1.10 Cleaning the imported data set

The data set that has been imported and is stored into the DataFrame named df is not clean.

1.10.1 Reverse the row order

First we want to have an timely ascending series:

```
df=df.iloc[::-1]
```

1.10.2 Rename the columns

We want to rename the columns of the DataFrame:

```
df.columns=['Month','Population']
```

1.10.3 Reset the index

We want to reset the index, knowing that the observations are monthly and start in January 1994:

```
dates2 = pd.date_range('1994-01', '2016-10', freq='M')
df.index = dates2
```

1.10.4 Delete a column

Now that we have reset the index, we can drop the column with the months that is not bringing much information now, you drop vertically, a column, that is axis = 1:

```
df=df.drop('Month',1)
```

1.11 Try to plot the DataFrame

Now try to plot your DataFrame df:

```
df.plot()
```

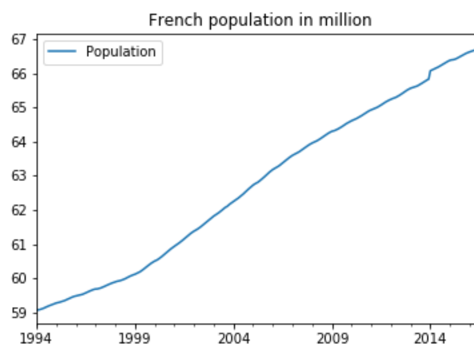
It doesn't work and tells you:

Empty 'DataFrame': no numeric data to plot

This means that when importing the data from the .csv file, python did not recognise that the column 'Population' were numeric. We need to 'tell' him. First you need to remove the space in the numbers, then converting strings to float:

```
df=df.replace({' ': ''}, regex=True)
df=df.astype(float)
```

Now when you try to plot, you should get:



1.12 Resample our DataFrame (from monthly to yearly observations)

There is the possibility to resample our DataFrame from one frequency to another using the resample() function. We need to indicate the desired frequency described in the **Offset Aliases** section of the time series documentation available here. We list some of the more commonly used:

- H hourly frequency
- D calendar day frequency
- W weekly frequency
- Q quarter end frequency
- A year end frequency

The we need to chose a "method" for resampling, for example .sum() or .mean()
In our case, if we want to move to yearly frequency and get the average over the 12 months:

dfyear=df.resample('A').mean()

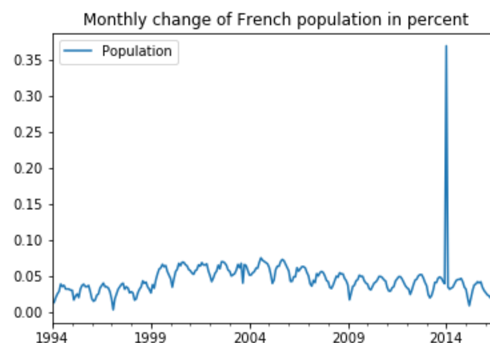
If we plot the obtained DataFrame, we see that the plot is slightly smoother:

dfyear.plot()

1.13 Compute monthly changes

Now you can plot and compute monthly changes:

df_change=df/df.shift(1)−1



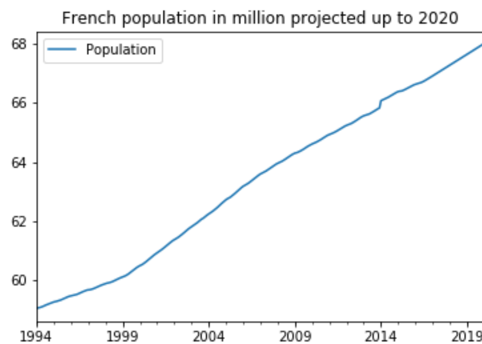
1.13.1 Descriptive statistics

You can get some descriptive statistics on the monthly population changes:

df_change.describe()

1.14 Plot some previsions

From the average monthly French population change, append to your DataFrame some projections until 2020, plot this projection.



1.14.1 Take the seasonality into account

Exercise: stick to the monthly seasonality, compute a monthly average population change and use it to project the population up to 2020.

1.14.2 Show a "confidence" interval for your forecast

If we assume that the French population evolves according to its average, for each actual observation we have some errors compare to our model:

$$\text{pop}_{m+1} = \text{pop}_m \left(1 + \text{avgchg}\right) + \epsilon_m$$

For the 273 observation we have, we can compute the error terms ϵ_m and deduct its standard deviation σ_{pop}. Then for a $1-\alpha$ % interval of confidence, it would be, with an approximate formula[1]

$$\text{pop}_{m+1} = \text{pop}_m \left(1 + \text{avgchg}\right) \pm Z_{\alpha/2}\sigma_{pop}$$

Where:

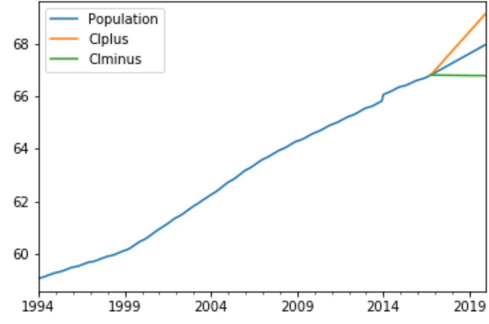
$$\sigma_{pop}^2 = \sigma_\epsilon^2 \left(1 + \frac{1}{n} + \frac{(\text{POP} - \bar{\text{POP}})^2}{\sum_m (\text{pop}_m - \bar{\text{POP}})^2}\right)$$

and we approximate the residual variance σ_ϵ^2 by:

$$s^2 = \frac{\sum \epsilon_m^2}{n-k}$$

with n the number of observations and k the number of independent variables.
For a 5% two-sided confidence interval we take a t distribution with n degrees of freedom (which we approximate by a normal distribution as $n >> 120$): $Z_{\alpha/2} = 1.96$.

[1] For more information, there is a detailed lecture on this here


French populatio projection with an idea of confidence interval
Population
CIplus
CIminus
68
66
64
62
60
1994
1999
2004
2009
2014
2019

2 Finding location of an element in a DataFrame

The panda's documentation explains how to access specific element of a DataFrame, using either the label or the integer positions. Link here.
Let's create a new DataFrame: We can create a DataFrame:

df2=pd.DataFrame([[0,1,2],[3,4,5],[7,8,9]])

We can rename the column of df2:

df2.columns=['A','B','C']

Now we have df2:

Index	A	B	C
0	0	1	2
1	3	4	5
2	7	8	9

2.1 Selection By Label

You can use the .loc attribute to access elements in a DataFrame. For df2, that would be:

	A	B	C
0	df2.loc[[0],['A']]	df2.loc[[0],['B']]	df2.loc[[0],['C']]
1	df2.loc[[1],['A']]	df2.loc[[1],['B']]	df2.loc[[1],['C']]
2	df2.loc[[2],['A']]	df2.loc[[2],['B']]	df2.loc[[2],['C']]

You can also create subset of your dataframe:

slice1=df2.loc[:,['A', 'C']]
slice2=df2.loc[1,['A', 'C']]
slice3=df2.loc[1:,:'B']

Which yields from left to right:

Index	A	C
0	0	2
1	3	5
2	7	9

Index	1
A	3
C	5

Index	A	B
1	3	4
2	7	8

If you want to work with the value of the element directly, you can use:

df2.loc[1, 'A']

2.2 Selection by position (integer)

If you want to be able to select a cell in a DataFrame, regardless of the name of the column or the name of the row, you can use integer based indexing with .iloc attribute.
Creating the above slices would become: You can also create subset of your dataframe:

slice1i=df2.iloc[:,[0, 2]]
slice2i=df2.iloc[1,[0, 2]]
slice3i=df2.iloc[1:,:2]

Note the subtlety, illustrated by our example slice3i:

- indexing starts at 0
- with python slices, the start is included but not the stop; so $':2'$ yields $[0,1]$ and not $[0,1,2]$

If you want to select a row or a column, you can use:

```
#Row 0
df2.iloc[2]
#Column C
df2.T.iloc[2]
```

If you want to work with the value of the element directly, it is very similar to the method above, you can check that this statement is True:

```
df2.loc[1, 'A']==df2.iloc[1,0]
```

2.3 Other selection methods

You can also use other selection methods, described in panda's documentation: .at, .iat, .ix

2.4 Unstack

We can use the function unstack on the DataFrame to transform it to a list with the combinations of row and column index:

```
df2_unstack=df2.unstack()
```

We observe for df2_unstack:

A,0	df2.iloc[0,0]
A,1	df2.iloc[1,0]
A,2	df2.iloc[2,0]
B,0	df2.iloc[0,1]
B,1	df2.iloc[1,1]
...	...

2.5 Loop through a DataFrame

We can multiply each element of the DataFrame df2 by 2:

```
df3=df2*2
```

Now if we want to multiply each element of the DataFrame df2 by 2 manually: First we can create a new DataFrame, df4, empty:

```
df4=pd.DataFrame(index=[0,1,2], columns=[0,1,2])
```

df4 and df2 have the same number of columns and rows. We can loop through each elements:

```
for i in range(0,len(df2)):
    for j in range(0,len(df2.columns)):
        df4.iloc[i,j]=df2.iloc[i,j]*2
```

The idea behind this is that the elements of the DataFrame df2 are found:

df2.iloc[0,0]	df2.iloc[0,1]	df2.iloc[0,2]
df2.iloc[1,0]	df2.iloc[1,1]	df2.iloc[1,2]
df2.iloc[2,0]	df2.iloc[2,1]	df2.iloc[2,2]

You should find that df4 matches exactly df3.
Now create df5 where you loop only through the column number 2 of df2.
Now create df6 where the value of an element is the column index times its row index.
If you want to transpose a DataFrame:

```
df7=df2.T
```

Recreate the transpose function with the for loops we have seen above to create a df8.

3 Correlation, regression, VAR

3.1 Correlation

Let's now download the French GDP from the INSEE website, here.
We do the same manipulations we did on the population csv file, except that now we have quarterly data:

```
gdp=pd.read_csv('GDP.csv',sep=';',encoding='latin1',skiprows = [0,1])
gdp=gdp.iloc[::-1]
gdp.columns=['Quarter','GDP']
dates3= pd.date_range('1949-01', '2016-09', freq='Q')
gdp.index=dates3
gdp=gdp.drop('Quarter',1)
gdp=gdp.replace({'␣': ''}, regex=True)
gdp=gdp.astype(float)
gdp.plot()
```

Now we want to concatenate the GDP and the population time series, on the dates, as explained here:

```
both=pd.concat([df,gdp],axis=1)
```

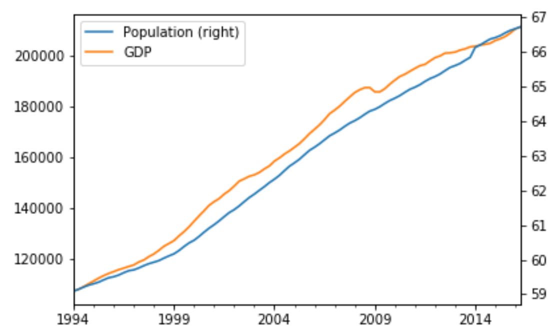
In order to only keep rows where there are both number, we get rid of Not A Numbers:

```
both=both.dropna(axis=0)
```

We can now plot, using a secondary y axis on the right for the population:

```
both.plot(secondary_y=['Population'])
```

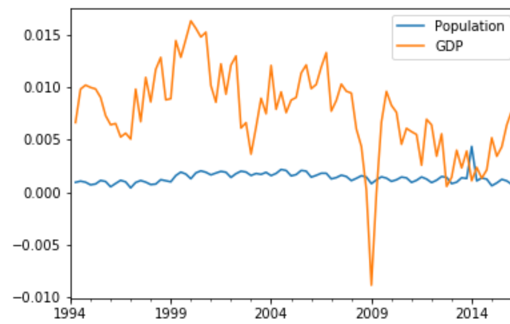
You should get:



We can compute the quarterly changes:

```
both_change=both/both.shift(1)-1
```

We expect to have a low correlation between both series:



We can check this by computing the correlation between the two:

both_change.corr()

We observe a low correlation of 19%.

3.2 Before regressing

As detailed in the lecture note here, before regressing (simple or VAR), we need to go step by step on the integration order and the cointegration of the variables.
We import the package that will enable us to use directly a function to perform an augmented Dickey-Fuller test:

import statsmodels.tsa.stattools

In order to apply the function adfuller (augmented Dickey-Fuller), we need to input a time series, hence the use of .unstac() on our DataFrame:

#Unemployment
statsmodels.tsa.stattools.adfuller(u.unstack())
#Population
statsmodels.tsa.stattools.adfuller(df.unstack())
#GDP
statsmodels.tsa.stattools.adfuller(gdp.unstack())

The outcome of the function adfuller are detailed in the "Returns:" section of its website description, here. H_0: the time series has a unit root.

The output of the test are:

- The statistics
- The p-value
- The number of lags used for the test
- The number of observations used
- The critical values

The monthly variables of unemployment, population and GDP are integrated of order 1.

3.3 Cointegration test

After we differenced the variables, we are cannot check for their long term relation in level.
With to I(1) series y_t and z_t, this would be difficult. But two I(1) series are said to be cointegrated, if there exist a linear combination of them (e.g. $\alpha + y_t + \beta_z z_t$) that is I(0). If $E(\alpha + y_t + \beta_z z_t) = 0$, then there is a long term relationship between y_t and z_t, with an equilibrium relationship: $y_t = -\alpha - \beta_z z_t$.
We can then test for cointegration between variable with a cointegration test: the null hypothesis is no cointegration. The second output give in the p-value of the test.
For our variables, we do not reject the null hypothesis and assume no cointegrations.

3.4 Linear regression

Now we want to perform a simple linear regression of population change on French GDP change:

$$\Delta Y_{\text{Pop}} = \alpha + \beta \Delta X_{\text{GDP}} \tag{1}$$

For performing an Ordinary Least Square method, we propose to use the following package:

```
import statsmodels.formula.api as smf
```

We store the results of the regression in the variable "results":

```
results = smf.ols('Population ~ GDP',data=both_change).fit()
```

If you would want to avoid the constant:

$$\Delta Y_{\text{Pop}} = \beta_{\text{no constant}} \Delta X_{\text{GDP}} \tag{2}$$

Then you just slightly modify the code:

```
results_no_constant = smf.ols('Population ~ GDP-1',data=both_change).fit()
```

We can print the summary of the OLS, with a ridiculous R^2 of 2.6% and a p-value of 7% for β:

```
print(results.summary())

print(results_no_constant .summary())
```

3.4.1 R-square and t-student "interpretation"

For a simple regression model: $y_i = \beta x_i + \epsilon_i$, our Ordinary Least Square programme is:

$$\min_\beta \sum_i (y_i - \beta x_i)^2$$

With first order conditions, we can find the estimate of β and we could prove that it is linear, unbiased, and efficient:

$$\hat{\beta} = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$$

We can demonstrate that, if var$(\epsilon) = \sigma^2$:

$$\text{var}(\beta) = \frac{\sigma^2}{\sum_i x_i^2}$$

By decomposing the variance and with enough observations, it can be demonstrated that the total variance is decomposed by the variance of the model and the error terms:

$$\text{var}(y_i) = \text{var}(\hat{y}_i) + \text{var}(\hat{\epsilon}_i)$$

we then define R^2 as the ratio between the variance of the model and the total variance:

$$R^2 = \frac{\sum_i (\hat{y}_i - \bar{y})^2}{\sum_i (y_i - \bar{y})^2}$$

$R^2 \in [0,1]$, the closest to 1, the more explanatory our model (if our econometric methods were correct).
The t statistic for a coefficient is defined as:

$$t = \frac{\hat{\beta}}{\hat{\sigma}_{\hat{\beta}}}$$

This statistics follow a Student law with $N-k$ degree of freedom (N being the number of observations and k the number of estimated parameters).
The null hypothesis H_0 is : $\beta = 0$, we can use a threshold of 5%, as it is two-sided, we search for t^* such that:

$$\text{Prob}(N-k < t^*) = 0.025$$

If $|t| > t^*$, then we can reject H_0, again, if our econometric methods were correct, x has a significant influence over y.

3.4.2 Visuals

We can store the α and β value of that regression for future use:

```
alpha=results.params[0]
beta=results.params[1]
```

We want to use the matplotlib pyplot package in order to be able to show some scatter plot:

```
import matplotlib.pyplot as plt
```

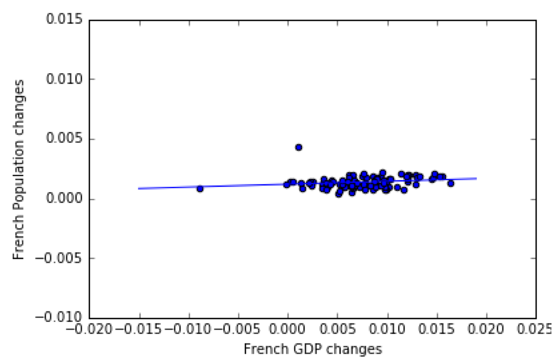
A basic scatter plot:

```
axes=plt.scatter(both_change['GDP'],both_change['Population'])
axes=plt.xlabel('French GDP changes')
axes=plt.ylabel('French Population changes')
```

Now if you want to plot the regression line with x ranging from -0.015 to 0.02 with steps of 0.001:

```
import numpy as np
x = np.arange(-0.015,0.02,0.001)
axes = plt.gca()
axes=plt.scatter(both_change['GDP'],both_change['Population'])
axes=plt.xlabel('French GDP changes')
axes=plt.ylabel('French Population changes')
axes=plt.plot(x,alpha+beta*x,'-')
figaxes = axes.get_figure()
figaxes.savefig('GDP_Pop_scatter.png')
```

You should get:



3.5 VAR model

Once again, this is only for illustration purpose. We want to compute a VAR model of the form:

$$\begin{bmatrix} \Delta p_t \\ \Delta g_t \\ \Delta u_t \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \begin{bmatrix} \Delta p_{t-1} \\ \Delta g_{t-1} \\ \Delta u_{t-1} \end{bmatrix}$$

With:

- p_t the population at quarter t
- g_t the GDP at quarter t
- u_t the unemployment rate at quarter t

3.5.1 Get the data

As we did with the GDP series, we fetch the unemployment series from INSEE website and import the data:

```
u=pd.read_csv('unemployment_france.csv',sep=';',encoding='latin1',skiprows = [0,1])
u=u.iloc[::−1]
u.columns=['Quarter','u']
dates4= pd.date_range('1996−01', '2016−09', freq='Q')
u.index=dates4
u=u.drop('Quarter',1)
u=u.replace({'␣': ''}, regex=True)
u=u.replace({',':'.'}, regex=True)
u=u.astype(float)
```

3.5.2 Preparing the VAR model

We prepare the VAR model, first we import the corresponding package:

```
from statsmodels.tsa.api import VAR
```

We resample the population series in quarterly:

```
df=df.resample('Q').mean()
```

Then we concatenate the 3 series into a DataFrame, take the differences related to the integration order of each variable and drop the rows where there are some NaN:

```
df_var=pd.concat([df,gdp,u],axis=1)
df_var_change=df_var/df_var.shift(1)−1
df_var_change=df_var_change.dropna(axis=0)
```

Finally we apply the VAR model and print the results summary:

```
varmodel = VAR(df_var_change)
result_var = varmodel.fit(1)
result_var.summary()
```

We are blocked by the fact that the unemployment figures are stable from one period to the other. We propose to remove it from the VAR:

$$\begin{bmatrix} \Delta p_t \\ \Delta g_t \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} + \begin{bmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{bmatrix} \begin{bmatrix} \Delta p_{t-1} \\ \Delta g_{t-1} \end{bmatrix}$$

We find, among other:

- $\hat{b}_1 = 0.00065$
- $\hat{c}_{1,2} = 0.39$

3.6 A VAR model that "works"

It is not easy to find $I(0)$ data and to model a VAR that makes sense.
We use the data in Bourbonnais' Econometrie, Dunod. You can download the file online here.
Demand Y_1 and prices Y_2 of some commodity are given.

3.6.1 Import the data

We import the data and clean it:

```
bour=pd.read_excel('C10EX2.XLS')
bour=bour.dropna(axis=0)
datesb = pd.date_range('2001−01', '2019−01', freq='Q')
bour.index=datesb
bour=bour.drop('Date',axis=1)
```

3.6.2 Verify that the series are stationary

We verify that both given series are stationary:

```
import statsmodels.tsa.stattools
#ADF test
statsmodels.tsa.stattools.adfuller(bour['Y1'], regression='ct')
statsmodels.tsa.stattools.adfuller(bour['Y2'], regression='ct')
```

We find indeed that both series are $I(0)$.

3.6.3 Select the lag order

We create the VAR model and select the lag, search for up to lag order 15:

```
varmodelb = VAR(bour)
#Lag selection
varmodelb.select_order(15)
```

The VAR order selected is 1:

```
                VAR Order Selection
=====================================================
            aic      bic      fpe        hqic
-----------------------------------------------------
0          14.58    14.65    2.138e+06   14.60
1          14.27*   14.48*   1.572e+06*  14.35*
2          14.33    14.69    1.683e+06   14.47
* Minimum
```

3.6.4 Estimate the VAR coefficients

Now that we selected the lag order 1, we can estimate the VAR coefficients:

```
result_varb = varmodelb.fit(1)
result_varb.summary()
```

This time we find all the coefficients significative except one:

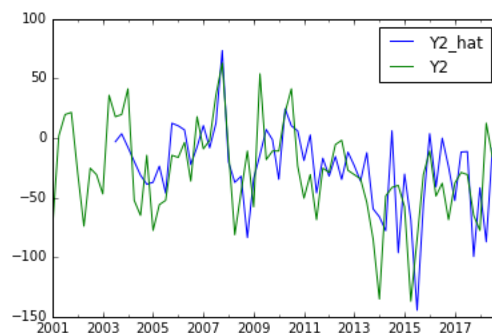
$$\begin{bmatrix} Y_{1,t} \\ Y_{2,t} \end{bmatrix} = \begin{bmatrix} 17.12 \\ -12.86 \end{bmatrix} + \begin{bmatrix} 0 & -0.61 \\ -0.17 & 0.29 \end{bmatrix} \begin{bmatrix} Y_{1,t-1} \\ Y_{2,t-1} \end{bmatrix}$$

3.7 A rolling regression

If we want to perform a rolling regression of Y_2 on Y_1, we can simply use the function:

```
model = pd.stats.ols.MovingOLS(y=bour.Y2, x=bour.Y1,window_type='rolling', window=10, intercept=False)
bour['Y2_hat'] = model.y_predict
bour[['Y2_hat','Y2']].plot()
```

We get:



We can also define the rolling regression ourselves:

```
#Let's define a moving OLS ourselves:
import statsmodels.api as sm
def movOLS(df,window):
    Betas=[]
    df.columns=['X','Y']
    for i in range(0,(len(df)-window)):
        resultat=sm.OLS(df[i:(i+window)].X,df[i:(i+window)].Y).fit()
        Betas.append(resultat.params[0])
    return Betas;
```

And apply our function. Note that we need to add some zeros at the beginning of our beta list with the size of our window:

```
window=10
betas=movOLS(bour[['Y2','Y1']],window)
debut=pd.DataFrame(np.zeros(window))
Betas=debut.append(betas)
Betas.index=bour.index
bour['Y2_hatbis']=Betas.multiply(bour['Y1'],axis=0)
```

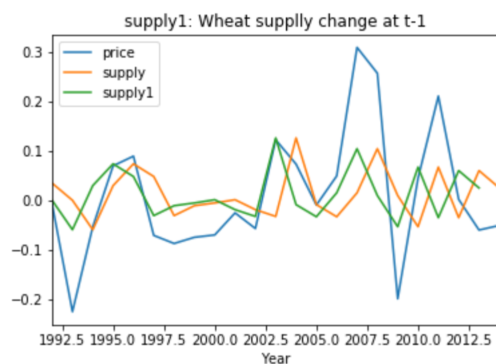
When we test our function, it provides the same result as the python defined function, up to some rounding errors (10^{-15}).

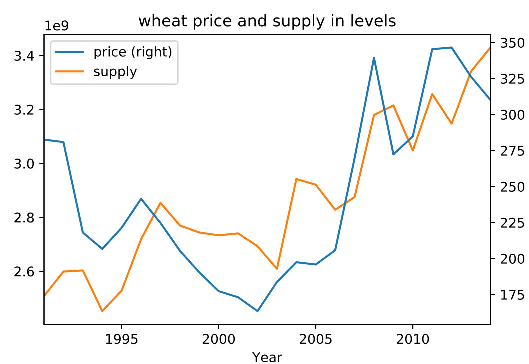
```
diff=bour['Y2_hat']-bour['Y2_hatbis']
```

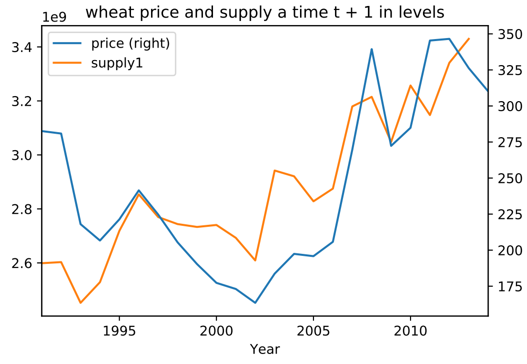
3.8 Rice price and world supply

We use data from FOOD AND AGRICULTURE ORGANIZATION OF THE UNITED NATIONS:

Exercise: try the same with rice and wheat prices and world supplies, we can apply a VAR model and rolling regression.
What is striking is that the wheat price and world supply in our data set seems to correlate with a lag:






wheat price and supply a time t + 1 in levels
1e9
price (right)
supply1
3.4
3.2
3.0
2.8
2.6
350
325
300
275
250
225
200
175
1995
2000
2005
2010
Year