Python pour Calcul Scientifique

Résumé

*Cette première vignette d'initiation au langage Python décrit l'exécution de commandes interactives ou de scripts Python avec un calepin (*notebook*) ou encore un IDE Spyder, les types et structures élémentaires de données, les premières structures de contrôle, les fonctions et modules. L'utilisation des librairies scientifiques (*`Numpy, Matplotlib, Scipy`*) et du type* `array` *est également introduite. Les tutoriels suivants abordent progressivement les outils plus spécifiques pour la "Science des Données".*

- *Python pour Calcul Scientifique*
- *Trafic de Données avec Python.*`Pandas`
- *Apprentissage Statistique avec Python.*`Scikit-learn`
- *Programmation élémentaire en Python*
- *Sciences des données avec* `Spark-MLlib`

1 Introduction

1.1 Pourquoi Python

Le langage Python est développé et diffusé par la *Python Software Foundation* selon une licence *GPL-compatible*. À partir d'applications initialement de calcul scientifique (image, signal...), son utilisation s'est généralisée dans de nombreux domaines et notamment pour l'analyse statistique de données trop volumineuses pour R ou Matlab. Il est donc "libre", efficace en calcul numérique (librairie `NumPy`), orienté objet... et bénéficie d'une communauté très active qui développe de nombreuses applications et librairies.

L'objectif de ce tutoriel est d'introduire le langage Python et quelques librairies pour surtout préparer puis commencer à analyser des données trop volumineuses pour la mémoire vive (RAM) d'un ordinateur basique. Lorsque, 2ème étape, elles sont trop volumineuses pour la taille du disque et distribuées sur les nœuds d'un *cluster* sous *Hadoop*, c'est encore le langage Python qui permet de passer à l'échelle pour exécuter des analyses en utilisant la librairie MLlib de la technologie *Spark*.

De façon plus précise, Python et la librairie `pandas` offrent des outils efficaces, comme le découpage automatique en morceaux (*chunks,*) adaptés à la taille de la mémoire vive ou encore l'accès à des données au format binaire HDF5 (librairie `Pytable`), pour lire (format `csv` ou fixe), gérer, pré-traiter, trafiquer (en jargon : *data munging* ou *wrangling*), visualiser des données volumineuses. Néanmoins, la parallélisation des traitements pour des analyses complexes (apprentissage statistique) sera sans doute plus efficace sur une architecture adaptée avec la librairie `MLlib` de Spark/Hadoop plutôt qu'avec la librairie `Scikit-learn` de Python. A part quelques méthodes relativement frustres, les principaux développements de cette librairie sont en effet adaptés à un `data frame` chargé en mémoire.

La version 3.4. de Python est celle actuellement la "plus récente". Le passage à la version 3 introduisit une rupture de compatibilité par rapport à la version 2 qui est toujours en développement (2.7). Il reste actuellement nécessaire de pouvoir utiliser les 2 versions selon les librairies utilisées (la 2.7 pour Spark) et applications recherchées. La version 2.7 inclut des ajouts permettant des éléments de "rétro"-compatibilité avec la version 3. Pour l'usage rudimentaire de ce tutoriel, il semble que les deux versions soient compatibles.

1.2 Prérequis

Cette vignette introduit le langage libre Python et décrit les premières commandes nécessaires au pré-traitement des données avant l'utilisation de méthodes statistiques avec ce langage ou avec R. Les aspects statistiques développés dans les différents scénarios de Wikistat sont supposés acquis ainsi qu'une connaissance des principes élémentaires de programmation dans un langage matriciel comme R ou Matlab.

Pour des approfondissements, il existe de très nombreuses ressources pédagogiques accessibles sur la toile dont le tutoriel officiel de Python 3.4., les sites `pythontutor.com`, courspython.com, le livre de Sheppard (2014)[2] qui présentent une introduction à Python pour l'Économétrie et la Statistique et celui de Mac Kinney (2013)[1], principal auteur de la bibliothèque `pandas`. À noter que ces ouvrages offrent une large part aux outils de gestion des séries chronologiques pour l'analyse financière.

1.3 Installation

Python et ses librairies peuvent être installés dans quasiment tout environnement matériel et système d'exploitation à partir du site original. Voici les principales librairies scientifiques définissant des structures de données et fonctions de calcul indispensables.

`ipython` : pour une utilisation interactive de Python,

`numpy` : pour utiliser vecteurs et tableaux,

`scipy` : intègre les principaux algorithmes numériques,

`matplotlib` pour les graphes,

`pandas` : structure de données et feuilles de calcul,

`patsy` : formules statistiques,

`statsmodels` : modélisation statistique,

`seaborn` : visualisation de données,

`scikit-learn` : algorithmes d'apprentissage statistique.

Néanmoins, compte tenu de la complexité de l'opération, il est plus simple pour le néophyte, surtout sous Windows, de faire appel à une procédure d'installation intégrant les principales librairies. Ces procédures sont développées par des entreprises commerciales mais libres de droits pour une utilisation académique.

Continuum Analytics propose Anaconda avec au choix la version 3.4 ou celle 2.7. `Conda` est l'utilitaire (commande en ligne) qui permet les mises à jour et installations des librairies complémentaires.

Enthought propose Canopy qui n'installe pour le moment que la version 2.7 et intègre un *package manager* avec interface graphique. Attention, certaines librairies même "collectives" ne sont disponibles que dans la version commerciale ou celle "académique" de Canopy après inscription.

D'un point de vue légal, les propositions sont identiques mais Canopy nécessite la création d'un compte académique. Seul "souci", ces versions n'incluent ques les versions dites stables des différentes librairies et donc avec un temps de retard vis-à-vis des versions encore développement.

2 Utilisation de Python

Python exécute programmes ou scripts, programmes qui peuvent être précompilés pour plus d'efficacité. Ce langage s'exécute également à l'aide d'un interprète de commande (`IDLE` ou `IPython`) de manière interactive.

2.1 IPython

A la suite du prompt : `In[i]`, entrer chaque commande qui sera immédiatement exécutée. Cet environnement offre différentes possibilités :

Tab complétion. La touche de tabulation propose une complétion automatique d'objet ou commande ;

Aide : `objet?`, `objet??` où `objet` est une variable, une fonction ou une commande affiche les caractéristiques de l'objet ou de l'aide en ligne ;

Magic Certaines commandes précédées de `%` sont spécifiques à IPython. Par exemple :

- `%run` pour exécuter tout un fichier (extension `.py`) de commandes python,
- `%timeit` pour afficher la durée d'exécution d'une commande,
- `%reset` pour effacer les objets et réinitialiser la session,
- `%cpaste` pour copier/coller en respectant les indentations,
- `%pdb` mode de débogage,
- `%magic` : liste des commandes "magiques".

2.2 *Notebook* de `IPython`

IPyhton s'exécute de façon interactive dans une fenêtre de lignes de commande ou à partir d'un navigateur pour créer un *Notebook* ou calepin. Les commandes sont regroupées dans des cellules suivies de leur résultat après exécution. Ces résultats et commentaires sont stockés dans un fichier spécifique `.ipynb` et sauvegardés. Les commandes LaTeX sont acceptées pour intégrer des formules, la mise en page est assurée par des balises HTML ou `Markdown`.

La commande de sauvegarde permet également d'extraire les seules commandes Python dans un fichier d'extension `.py`. C'est une façon simple

et efficace de conserver tout l'historique d'une analyse pour en faire une présentation ou créer un tutoriel. Le calepin peut être en effet chargé sous un autre format : page `html`, fichier `.rst` "restructuré", `.pdf` ou encore converti au format LaTeX, `.js` (diaporama HTML) par la commande `ipython nbconvert --to slides` (ou `latex`).

Important Le projet Jupyter propose le même environnement de type calepin (*IPython Notebook*) pour beaucoup de langages (Pyhton, Julia, Scala...) et environnements logiciels (R, Spark...). Jupyter devient de la sorte un standard pour élaborer et diffuser (tutoriels) des enchaînements de traitements. C'est un outil important pour assurer la *reproductibilité* des analyses.

L'ouverture d'un navigateur sur un calepin (Ipython ou Jupyter) est obtenu, selon l'installation, à partir des menus ou en exécutant :

```
ipython notebook
```

ou

```
jupyter notebook
```

dans la fenêtre de commande.

Une fois le calepin ouvert,

- Entrer des commandes Python dans une cellule .
- Cliquer sur le bouton d'exécution de la cellule.
- Ajouter une ou des cellules de commentaires et balises HTML ou `Markdown`.

Itérer l'ajout de cellules. Une fois l'exécution terminée :

- Sauver le calepin `.ipynb`
- Charger éventuellement une version `.html` pour une page web.
- Charger le fichier `.py` regroupant les commandes python pour une version opérationnelle.

Attention Un calepin de IPython ou Jupyter est un outil de travail exploratoire efficace et un compte rendu nécessairement *chronologique* d'exécution ; ce n'est pas le *rapport* d'une étude statistique dont l'organisation suit des règles spécifiques.

2.3 IDE `Spyder`

Pour la réalisation d'applications et programmes plus complexes, l'usage d'un IDE (*Integrated Development Environment*) libre comme Spyder est recommandé. Ce dernier est intégré à la distribution `Anaconda` et sa présentation proche de celles de Matlab ou RStudio. Cet environnement exécutant IPython reconnaît évidemment les commandes précédentes.

Comme pour RStudio, `Spider` ouvre plusieurs fenêtres :

- un éditeur de commande dont les boutons du menu exécutent tout le fichier ou interactivement la cellule courante, sauvent le fichier, contrôlent le débogage. Une cellule débute par la balise : `#%%`.
- Un explorateur d'objets avec aide en ligne, des variables en cours, du répertoire courant. Les boutons de l'explorateur de variables permettent de supprimer, sauver les objets créés ou encore d'importer des données.
- La console IPython avec les résultats et son historique.

2.4 Exemple

En résumé, utiliser un calepin pour des analyses exploratoires élémentaires et un IDE (*Spyder*) pour la construction de programmes et modules.

Sous windows, utiliser au choix l'installation

- *Anaconda* (python 3.4 ou 2.7) pour lancer un calepin ou *spyder]*,
- *Canopy* (python 2.7)pour ouvrir un calepin.

Sous Unix, utiliser l'IDE de son choix comme `Eclipse` (un peu compliqué !) ou lancer, à partir du répertoire de travail, la commande :

```
ipython notebook
```

qui ouvre le navigateur par défaut avec les menus contextuels ou encore

```
canopy.sh
```

si cette distribution est installée comme c'est le cas à l'INSAT.

Entrer les commandes ci-dessous dans le calepin et les exécuter cellule par cellule en cliquant sur le bouton (*widget*) d'exécution de la cellule courante.

Outre des commentaires, les premières lignes déclarent les librairies à utiliser comme par exemple :

```
# Ceci est le début d'une session Python
# importer les librairies
import matplotlib.pyplot as plt
import numpy as np
```

```
import pandas as pd
from pylab import *
import os
# Définir si nécessaire le répertoire courant
#  spécifique  de l'utilisateur.
# A modifier selon l'environnement
os.chdir(r"D:\Users\utilisateur\Documents\Exemple")
# Commande "magique" demandant d'intégrer les
# graphiques dans le calepin
%matplotlib inline
```

L'utilisation d'une commande de librairie est alors élémentaire :

```
# Créer un data frame avec pandas
data = pd.DataFrame({
    "Gender": ["f", "f", "m", "f", "m",
    "m", "f", "m", "f", "m"],
    "TV": [3.4, 3.5, 2.6, 4.7, 4.1,
    4.0, 5.1, 4.0, 3.7, 2.1]
    })
data
```

Séparer les cellules avec chacune un résultat.

```
# Graphique élémentaire
xx = randn(100,100)
y = mean(xx,0)
plot(y)
show()
```

3 Types de données python

Comme précédemment, exécuter les commandes de ce tutoriel *cellule par cellule* dans le calepin `IPython` ou `Jupyter`, ou encore dans un IDE `Spyder` ; en analyser les résultats.

3.1 Scalaires et chaînes

La déclaration des variables est implicite (integer, float, boolean, string), la syntaxe est très proche de celle de R mais il n'y a pas de type `factor`.

```
a=3  # est un entier
b=1. # est un flottant
# Attention
a/2  # a pour résultat 1.5 en Python 3.4
     # mais 1 en 2.7
```

Opérateurs de comparaison : `==`, `>`, `<`, `!=` de résultat booléen.

```
# Comparaison
a==b
#  affichage et type des variables
a
#
type(a)
# Chaîne de caractère
a="bonjour"
b="le"
c="monde"
a+b+c
```

3.2 Structures de données basiques

Listes

Les listes permettent des combinaisons de types. Attention, le premier élément d'une liste ou d'un tableau est indicé par 0, pas par 1.

Il est plus lisible dans un calepin de présenter un résultat par cellule et donc de séparer les liches de commandes provoquant un résultat.

```
# exemples de listes
liste_A = [0,3,2,"hi"]
liste_B = [0,3,2,4,5,6,1]
liste_C = [0,3,2,"hi",[1,2,3]]
# Elément d'une liste
```

```
Liste_A[1]
#  dernier élément
liste_C[-1]
liste_C[-1][0]
liste_C[-2]
#  Sous-liste
liste_B[0:2]
# début:fin:pas
liste_B[0:5:2]
liste_B[::-1]
# Fonctions de listes
List=[3,2,4,1]
List.sort()
List.append("hi")
List.count(3)
List.extend([7,8,9])
List.append([10,11,12])
```

Tuple

Un tuple est similaire à une liste mais ne peut être modifié, il est défini par des parenthèses.

```
# Tuple
MyTuple=(0,3,2,"h")
MyTuple[1]
MyTuple[1]=10 # TypeError: "tuple" object
# does not support item assignment
```

Dictionnaire

Un dictionnaire est similaire à une liste mais chaque entrée est assignée par une clé/un nom, il est défini avec des accolades.

```
# dictionnaire
months = {"Jan":31 , "Fev": 28, "Mar":31}
months["Jan"]
months.keys()
months.values()
months.items()
```

Cet objet est utilisé pour la construction de l'index des colonnes (variables) du type *DataFrame* de la librairie `pandas`.

4 Syntaxe de Python

4.1 Structures de contrôle élémentaires

Un bloc de commandes ou de codes est défini par deux points suivis d'une indentation fixe. Cela oblige à l'écriture de codes faciles à lire mais à être très attentif sur la gestion des indentations car la fin d'indentation signifie la fin d'un bloc de commandes.

Structure conditionnelle

```
# si alors sinon
a=2
if a>0:
        b=0
        print(b)
else:
        b=-1
print(b)
```

Structure itérative

```
# itération
for i in range(4):
        print(i)
for i in range(1,8,2):
        print(i)
```

4.2 Fonctions

Syntaxe

La syntaxe de la définition d'une fonction est la suivante :

```
def FunctionName(args):
    commands
    return value
```

Exemple

```
# Définition d'une fonction
def pythagorus(x,y):
        """ calcule l'hypoténuse d'un triangle """
        r = pow(x**2+y**2,0.5)
        return x,y,r
pythagorus(3,4)
# appel
pythagorus(x=3,y=4)
pythagorus(y=4,x=3)
# aide intégrée
help(pythagorus)
pythagorus.__doc__
```

Arguments avec valeur par défaut

```
# Valeurs par défaut
def pythagorus(x=1,y=1):
        """ calcule l'hypoténuse d'un triangle """
        r = pow(x**2+y**2,0.5)
        return x,y,r
pythagorus()
```

4.3 Modules et librairies

Modules

Un module contient plusieurs fonctions et commandes qui sont regroupées dans un fichier d'extension `.py`. Insérer un fichier vide de nom `_init_.py` dans chaque dossier et sous-dossier contenant un module à importer. Un module est appelé par la commande `import`. Un module est considéré comme un script s'il contient des commandes. Lors de l'import d'un script, les commandes sont exécutées tandis que les fonctions sont seulement chargées.

Commencer par définir un module dans un fichier texte contenant les commandes suivantes.

```
def DitBonjour():
        print("Bonjour")
def DivBy2(x):
        return x/2
```

Sauver le fichier avec pour nom `testM.py` dans le répertoire courant de `IPython`.

Il est possible d'importer toutes les fonctions en une seule commande `import`.

```
import testM
testM.DitBonjour()
print(testM.DivBy2(10))
#  autre possibilité
from testM import *
DitBonjour()
print(DivBy2(10))
```

ou seulement celles qui seront utilisées. Préférer cette dernière méthode pour les grosses librairies.

```
import testM as tm
tm.DitBonjour()
print(tm.DivBy2(10))
%reset
from testM import DitBonjour
DitBonjour()
print(DivBy2(10)) # erreur
```

Lors de son premier appel, un module est pré-compilé dans un fichier `.pyc` qui est utilisé pour les appels suivants. Attention, si le fichier a été modifié / corrigé, il a besoin d'être rechargé par la commande `reload(name)`.

Librairies

Une librairie (*package*) regroupe plusieurs modules dans différents sous-répertoires. Le chargement spécifique d'un des modules se fait en précisant le

chemin.

```
import sound.effects.echo
```

5 Calcul scientifique

Voici trois des principales librairies indispensables au calcul scientifique. Deux autres librairies : `pandas`, `scikit-learn`, sont exposées en détail dans des vignettes spécifiques.

5.1 Principales librairies ou *packages*

`NumPy`

Cette librairie définit le type de données `array` ainsi que les fonctions de calcul qui y sont associées. Il contient aussi quelques fonctions d'algèbre linéaire et statistiques. Il est supporté par Python 2.6 et 2.7, ainsi que 3.2 et plus récent.

Il n'est finalement utilisé que pour la définition du type `array` car les fonctions numériques sont beaucoup plus développées dans `SciPy`.

`Matplotlib`

Celle-ci propose des fonctions de visualisation / graphs avec des commandes proches de celles de Matlab. Aussi connue sous le nom de `pylab`.

```
# Importation
import numpy as np
from pylab import *
gaussian = lambda x: np.exp(-(0.5-x)**2/1.5)
x=np.arange(-2,2.5,0.01)
y=gaussian(x)
plot(x,y,label='$y=\exp(-(0.5-x)^2/1.5))$')
# On peut mettre des commentaires Latex
xlabel("x values")
ylabel("y values")
title("Gaussian function")
legend(loc='upper left')
show()
```

La gallerie de cette librairie propose tout un ensemble d'exemples de graphiques avec le code Python pour les générer.

`SciPy`

Cette librairie est un ensemble très complet de modules d'algèbre linéaire, statistiques et autres algorithmes numériques. Le site de la documentation en fournit la liste.

5.2 Type `array`

C'est de loin la structure de données la plus utilisée pour le calcul scientifique sous Python. Elle décrit des tableaux ou *matrices* multi-indices de dimension $n = 1, 2, 3, \ldots, 40$. Tous les éléments sont de même type (booléen, entier, réel, complexe).

Il est possible de contrôler précisément le type d'un `array`, par exemple pour gagner de la place en mémoire, en codant les entiers sur 8, 16, 32 ou 64 bits, de même pour les réels (*float*) ou les complexes.

Les tableaux ou tables de données (`data frame`), bases d'analyses statistiques et regroupant des objets de type différents sont décrits avec la librairie `pandas`.

Définition du type `array`

```
# Importation
import numpy as np
my_1D_array = np.array([4,3,2])
print(my_1D_array)
my_2D_array = np.array([[1,0,0],[0,2,0],[0,0,3]])
print(my_2D_array)
myList=[1,2,3]
my_array = np.array(myList)
print(my_array)
a=np.array([[0,1],[2,3],[4,5]])
a[2,1]
a[:,1]
print(a.dtype)
a[0,0]=1
```

```
print(a)
a[0,0]=1.5
print(a)
#1.5 a été converti en entier
#Il faut changer le type des éléments de a
B=a.astype(float)
B[0,0]=1.5
print(B)
```

Fonctions de type `array`

Génération de matrices.

```
# Une ligne par cellule
np.arange(5)
np.ones(3)
np.ones((3,4))
np.zeros((2,3))
np.eye(3)
np.linspace(3, 7, 3)
np.mgrid[0:3,0:2]
D=np.diag([1,2,4,3])
print(D)
print(np.diag(D))
M=np.array([[10*n+m for n in range(3)]
for m in range(2)])
print(M)
```

Le module `numpy.random` fournit toute une liste de fonctions pour la génération de matrices aléatoires.

```
from numpy import random
random.rand(4,2) #tirage uniforme [0,1)
random.randn(4,2) #tirage selon la loi N(0,1)
v=random.randn(1000)
import matplotlib.pyplot as plt
h=plt.hist(v,20) # histogramme à 20 pas
show()
A=random.randn(64,64)
plt.imshow(A,interpolation="nearest")
plt.colorbar
plt.show()
M=random.randn(10,10)
np.savetxt('data.csv',M,fmt='%2.2f',delimiter=',')
#au format propre à numpy : npy
np.save('data.npy',M)
np.load('data.npy')
```

Slicing

Extraction d'une partie d'un vecteur ou d'une matrice

```
v=np.array([1,2,3,4,5])
print(v)
v[1:4]
v[ : : ]
v[ : : 2] # par pas de 2
v[: 3] # les 3 premiers éléments
v[3 :] # à partir de l'indice 3
v[-1] # dernier élément
v[-2 :] # deux derniers éléments
M=random.rand(4,3)
print(M)
ind=[1,2]
M[ind] # lignes d'indices 1 et 2
M[:,ind] # colonnes d'indices 1 et 2
M[[0,2],[1,2]] # renvoie M[0,1] et M[2,2]
M[np.ix_([0,2],[1,2])]
(M>0.5)
M[M>0.5]
```

Autres fonctions

```
a=np.array([[0,1],[2,3],[4,5]])
# Nombre de dimensions
np.ndim(a)
# Nombre d'éléments
```

```
np.size(a)
# Tuple contenant la dimension de a
np.shape(a)
# Transposée
np.transpose(a)
a.T # autre façon de définir la transposée
# Valeur min
a.min(), np.min(a)
# Somme des valeurs
a.sum(), np.sum(a)
# Somme sur les colonnes
a.sum(axis=0)
# sur les lignes
a.sum(axis=1)
# aussi max, mean, std,...
```

Quelques manipulations :

```
# Concaténation en ligne
np.r_[1:4,10,11]
# Concaténation en colonne
np.c_[1:4,11:14]
# erreur
np.c_[1:4,11:15]
np.arange(6).reshape(3,2)
A=np.array([[1,2],[3,4]])
# Répétition de la matrice A
np.tile(A,(3,2))
A=np.array([[1,2],[3,4]])
B=np.array([[11,12],[13,14]])
#Concaténation en ligne
np.concatenate((A,B),axis=0)
#Equivalent à
np.vstack((A,B))
#Concaténation en colonne
np.concatenate((A,B),axis=1)
#Equivalent à
np.hstack((A,B))
```

Conversion de type avec les fonctions `dtype` et `astype`.

Opérations sur `array`

```
# somme
a=np.arange(6).reshape(3,2)
b=np.arange(3,9).reshape(3,2)
c=np.transpose(b)
c=b.T
a+b
# produit terme à terme
a*b
# produit matriciel
np.dot(a,c)
#
np.power(a,2)
#
np.power(2,a)
#
a/3
```

Les fonctions `genfromtxt`, `savetxt` permettent de lire, écrire des fichiers textes par exemple au format `.csv` mais ces fonctionnalités sont plus largement abordées avec la librairie `pandas`.

Fonctions d'algèbre linéaire

```
# Importation
import numpy as np
from scipy import linalg
A = np.array([[1,2],[3,4]])
linalg.inv(A)
#
linalg.det(A)
#
la,v = linalg.eig(A)
l1,l2 = la
```

```
# valeurs propres
print(l1, l2)
# 1er vecteur propre
print(v[:,0])
# 2ème vecteur propre
print(v[:,1])
U,s,V = linalg.svd(A) # SVD de A
print(s**2)
# vérifier les valeurs propres
linalg.eig(np.dot(np.transpose(A),A))
```

Tests élémentaires de Statistique

```
# Importation
import scipy.stats
rvs1 = scipy.stats.norm.rvs(loc=5, scale=10,
   size=500)
rvs2 = scipy.stats.norm.rvs(loc=5, scale=10,
   size=500)
rvs3 = scipy.stats.norm.rvs(loc=8, scale=10,
   size=500)
# t-test returns: t-statistic/two-tailed p-value
scipy.stats.ttest_ind(rvs1, rvs2)
scipy.stats.ttest_ind(rvs1, rvs3)
# Kolmogorov-Smirnov test
# returns: KS statistic / two-tailed p-value
scipy.stats.ks_2samp(rvs1, rvs2)
#
scipy.stats.ks_2samp(rvs1, rvs3)
```

Références

[1] W. Mac Kinney, *Python for Data Analysis*, O'Reilly, 2013, http://it-ebooks.info/book/1041/.

[2] K. Sheppard, *Introduction to Python for Econometrics, Statistics and Data Analysis*, 2014, https://www.kevinsheppard.com/images/0/09/Python_introduction.pdf.