Abderrahmane Fadil

Mathématiques et statistiques appliquées avec Python

Cours et exercices corrigés

ellipses

Chapitre 1
Notions de base du développement Python

1 Introduction

Avant d'aborder les outils et méthodes de visualisation de données (graphiques, courbes, …), de mise en application de fondements mathématiques et statistiques (algèbre linéaire, statistiques descriptives, …), d'importation/exportation de structures de données complexes (tableaux, matrices, *tensors*, …) ou encore l'introduction au *Machine Learning* (analyse de données, …) , il est utile de faire quelques rappels des notions de bases du développement Python.

2 Script Python

Un script informatique est une séquence d'instructions, écrites dans un langage dit de scripts et qui fait appel à un outil dit interpréteur afin d'exécuter les instructions. L'interpréteur est souvent associé à l'éditeur de code, qui permet d'exécuter la séquence d'instructions en interprétant les instructions l'une après l'autre de façon séquentielle. L'interpréteur ne crée pas un exécutable, contrairement à un compilateur et c'est là la différence majeure avec un langage compilé comme le C++ par exemple, et donc pour toute nouvelle exécution du script l'interpréteur est appelé et refait le même travail, à savoir lecture et interprétation des instructions l'une après l'autre de façon séquentielle.

Un programme écrit sous Python est un script et a besoin donc d'un programme auxiliaire, dit interpréteur, pour lire et exécuter au fur et à mesure les instructions du script.

Un script Python peut utiliser des fonctions et procédures prédéfinies, ce qui facilite la tâche du développeur. Ces fonctions sont classées par catégories et regroupées sous des librairies importables. Pour les éléments de langage, ces librairies sont dites aussi des modules, des bibliothèques ou encore des packages.

Pour pouvoir importer ces librairies et utiliser leurs fonctions prédéfinies il faut les avoir auparavant installées sous l'éditeur Python utilisé pour le développement.

Un script doit tenir compte également des contraintes d'environnement telles que la source de données lors d'importation de données ou encore le modèle de données lors d'une étape d'apprentissage par exemple en *Deep Learning*.

2.1 Représentation systémique

La *figure 1* ci-après est une représentation systémique d'un script. En effet, une séquence d'instructions est définie afin de réaliser un traitement de données.

Ce traitement agit sur des données en entrée e_1, e_2 , …, e_n où n est le nombre de ces données en entrée (c'est-à-dire les données qui alimentent le script).

Le traitement utilise ces données pour en produire d'autres. À la fin du traitement, ces données produites sont les données en sortie du script s_1, s_2, …, s_m où m est le nombre de ces données en sortie qui correspondent aux résultats escomptés du traitement (cf. *figure 1* ci-dessous).

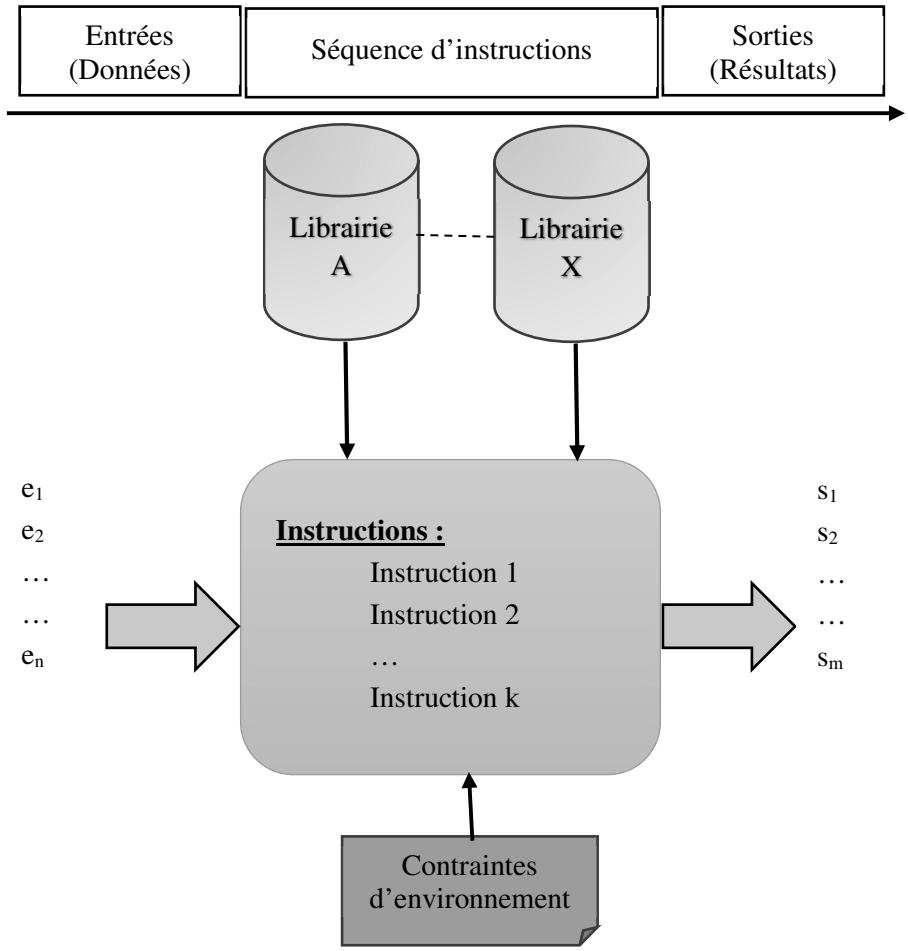


Figure 1. *Script modèle*

Voici, ci-dessous un exemple où on retrouve tous les éléments du script modèle, à savoir :

- importation de librairies ;
- contraintes d'environnement comme l'importation de données à partir d'un fichier texte ;
- les entrées / sorties ;
- les instructions de traitement.

2.2 Script Python

```
import numpy as np
import os

# Création de la matrice A
A=np.zeros(shape=(3,3))

# Chargement des données à partir d'une source externe
os.chdir("C:\\Exemples\Exemples Maths Data")
A=np.loadtxt("Matrice_ load.txt",delimiter="\t",dtype=float)
print("A = \n",A)

# taille et transpose d'une matrice
tA=np.zeros(shape=(A.shape[1],A.shape[0]))
tA=np.transpose(A)
print("tA = \n",tA)

# Déterminant d'une matrice carrée
d=np.linalg.det(A)
print("d = ",d)

# Valeurs propres et vecteurs propres
M=np.random.random(size=(2,2))*10
print("M = \n",M)
# vvp : vecteur des valeurs propres
# mvp : matrice des vecteurs propres (vecteurs colonnes)
vvp, mvp=np.linalg.eig(M)
```

```
print("vvp = ", vvp)
print("mvp = \n", mvp)

# M.v = l*v où v est un vecteur propre et l la valeur propre
correspondante
for i in range(M.shape[1]):
    vi=mvp[:,i]
    li=vvp[i]
    print("vérification n°", i+1, " : ",
          np.all(np.isclose(np.dot(M,vi),li*vi)))
```

2.3 Résultat

```
A =
 [[ 45.   62.  168. ]
  [-34.   15.5 187. ]
  [  1.   -1.2   0. ]]

tA =
 [[ 45.  -34.    1. ]
  [ 62.   15.5  -1.2]
  [168.  187.    0. ]]

d =  25942.4

M =
 [[2.45 1.87]
  [9.04 7.68]]

vvp =  [0.19 9.94]
mvp =
 [[-0.63 -0.24 ]
  [ 0.76 -0.97]]

vérification n° 1  :  True
vérification n° 2  :  True
```

3 Les instructions fondamentales

3.1 L'affectation

L'opération d'affectation permet d'affecter à une variable une valeur constante, ou le résultat d'une expression, ou le résultat d'une fonction, en utilisant l'opérateur = où le receveur est à gauche de l'opérateur et la valeur à affecter est à droite.

Exemples

- y = 20 ;
- y = (x**2 +1) ;
- y = numpy.sin((np.pi /3) * x) ;
- etc.

3.2 Les entrées / sorties

Les fonctions d'entrées / sorties qui permettent la lecture et l'écriture de données sont les suivantes :

Fonction	Syntaxe	Remarques et exemples
input()	v = input([*message*])	C'est l'instruction de saisie. v : est une variable qui reçoit la valeur saisie. [] : signifie facultatif. *message* : est un message à destination de l'utilisateur. Exemple : x = input("Saisir x : ")
print()	print(*arguments*)	C'est l'instruction d'affichage. *arguments* : la liste des arguments à afficher. Le séparateur des arguments est **,** (la virgule). Exemple : print("v = ", round(v, 2))

Exemple d'affectation et d'entrées / sorties

```
from math import sqrt

x=float(input("Saisir la valeur réelle de x : "))

# Instructions d'affectation
y=x**3
v=sqrt(y)

print("v = ",round(v,2))
```

Résultat

```
v =  111.18
```

3.3 Les structures de contrôle conditionnelles

Une structure de contrôle conditionnelle permet, selon le résultat de test d'une ou plusieurs conditions, à un script :

- de réaliser un bloc d'instructions lorsque une condition est vraie puis l'interpréteur passe à la suite du script ;
- de réaliser un bloc d'instructions lorsque une condition est vraie sinon un autre bloc d'instructions est exécuté puis l'interpréteur passe à la suite du script ;
    - de réaliser un bloc d'instructions parmi plusieurs blocs possibles (bifurcation).

Cela correspond à trois syntaxes du contrôle conditionnel.

3.3.1 Syntaxes

Syntaxe 1	**Remarque**
if Condition : *Bloc instructions*	Si *Condition* est vraie alors un bloc d'instructions est exécuté, sinon l'interpréteur passe à la suite du script.

Syntaxe 2	**Remarque**
if Condition : *Bloc instructions B1* *else :* *Bloc instructions B2*	Si *Condition* est vraie alors le premier bloc d'instructions est exécuté, sinon le deuxième bloc d'instructions est exécuté puis l'interpréteur passe à la suite du script.

Syntaxe 3	**Remarque**
if Condition$_1$: *Bloc instructions B_1* *elif Condition$_2$:* *Bloc instructions B_2* *...* *elif Condition$_n$:* *Bloc instructions B_n* *else :* *Bloc instructions B_{n+1}*	Si *Condition$_1$* est vraie alors le bloc d'instructions B_1 est exécuté, sinon si *Condition$_2$* est vraie alors le bloc d'instructions B_2 est exécuté, ainsi de suite, sinon le bloc d'instructions B_{n+1} est exécuté puis l'interpréteur passe à la suite du script.

3.3.2 Exemple

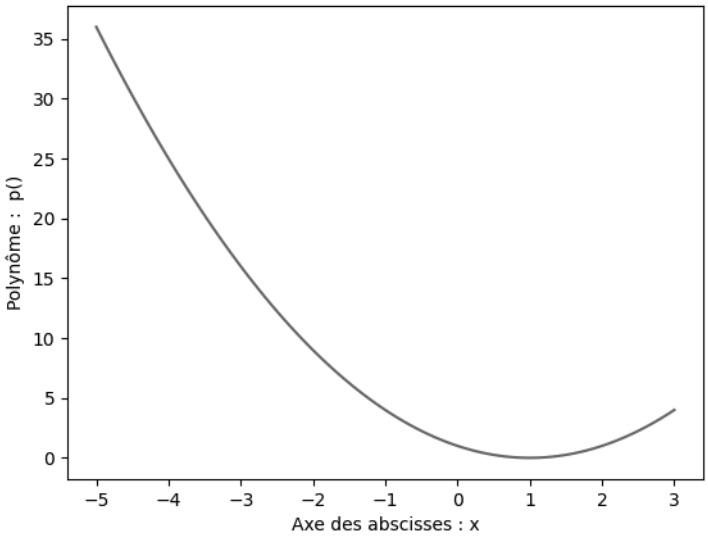
```
# Importation des librairies
import matplotlib.pyplot as plt
import numpy as np
# Définition des fonctions
def Est_Reel(valeur):
    try:
        float(valeur)
        return True
    except ValueError:
        return False

def Saisie_R():
            R=input("Saisir un réel : ")
```

```
        while(not Est_Reel(R)):
            R=input("Élément saisi n'est pas un réel. Recommencer : ")
        R=float(R)
        return(R)

# Saisie des coefficients
print("Coefficient a")
a=Saisie_R()
print("Coefficient b")
b=Saisie_R()
print("Coefficient c")
c=Saisie_R()

# Résolution
if a!=0:
        delta=b**2-4*a*c
        if delta<0:
                print("\nPas de racine réelle !")
        elif delta==0:
                x0=round(-b/(2*a),2)
                print("\nRacine unique x0 = ",x0)
        else:
                x1=round((-b-np.sqrt(delta))/(2*a),2)
                x2=round((-b+np.sqrt(delta))/(2*a),2)
                print("\nDeux racines : x1 = ",x1," ;  x2 = ",x2)
else:
        print("Ce n'est pas un polynôme de second degré")
# Graphique
p=np.poly1d([a,b,c])
print("Courbe du polynôme p(x) : ",p)
x=np.linspace(-5,3,100)
plt.plot(x,p(x))
plt.ylabel("Polynôme :  p()")
plt.xlabel("Axe des abscisses : x")
plt.show()
```

Résultat obtenu pour a = 1, b = −2 et c = 1

Coefficient a
Saisir un réel : 1
Coefficient b
Saisir un réel : -2
Coefficient c
Saisir un réel : 1

Racine unique x0 = 1.0
Courbe du polynôme p(x) : 1 x^2 - 2 x + 1



3.4 Les structures de contrôle répétitives

Une structure de contrôle répétitive permet à un script de répéter, c'est-à-dire itérer ou encore boucler, un bloc d'instructions un nombre n de fois :

- lorsque n est connu à l'avance, il s'agit alors de la boucle *for* ;
- lorsque n n'est pas connu à l'avance, il s'agit d'une boucle *while*.

3.4.1 La boucle for

La boucle *for* sous Python utilise un itérateur nommé *range()*. L'itérateur énumère des entiers dans un intervalle donné [start ;end[. Cette énumération se fait pas à pas et le pas est nommé *step*.

Syntaxe générale de la boucler *for* sous Python :

```
for i in range([start,] end [, step]) :
    bloc instructions
```

Remarque

Les crochets [] dans la syntaxe générale indiquent l'aspect facultatif et non obligatoire selon les situations considérées.

- *range(start, end, step)* # retourne les entiers de *start* à *end*−1, avec le pas *step* ;
- *range(start, end)* # retourne les entiers de *start* à *end*−1, avec le pas *step* = 1 ;
- *range(end)* # de même, avec *start* = 0 et avec le pas *step* = 1.

3.4.2 Exemple de la suite de Catalan

Ici, on utilise la boucle *for* afin de calculer les n premiers nombres de Catalan telle que la suite de Catalan est définie comme suit :

$$\begin{cases} C_0 = 1 \\ C_n = C_{n-1}\dfrac{2(2n-1)}{n+1} \quad pour\ n \geq 1 \end{cases}$$

Script

```
# Définition de la fonction de saisie
def Saisie_E(e):
    v=input("Saisir un entier : ")
    while (not v.isnumeric()) or (int(v)<e):
        v=input("Re-saisir un entier : ")
    v=int(v)
    return(v)

n = Saisie_E(1)
C=1
print("n =",0,"  , C = ",int(C))

# Boucle pour
```

```
for i in range(1,n+1):
    C=C*(2 * (2 * i - 1))/(i + 1)
    print("n =",i," , C = ",int(C))
```

Test du script

```
Saisir un entier : 10
n = 0   , C =  1
n = 1   , C =  1
n = 2   , C =  2
n = 3   , C =  5
n = 4   , C =  14
n = 5   , C =  42
n = 6   , C =  132
n = 7   , C =  429
n = 8   , C =  1430
n = 9   , C =  4862
n = 10   , C =  16796
```

3.4.3 La boucle while

La boucle *while* sous Python permet de répéter un bloc d'instructions. Le nombre d'itérations n'est pas connu à l'avance, cependant la répétition est conditionnée par la véracité d'une ou plusieurs conditions.

Syntaxe générale de la boucler *while* sous Python :

```
while (Condition) :
    bloc instructions
```

3.4.4 Exemple de script de test statistique avec while

```
# Importation des librairies
import random
import pandas as pd
import matplotlib.pyplot as plt

rPearson=0
matricule = pd.Series(range(101,111))
```

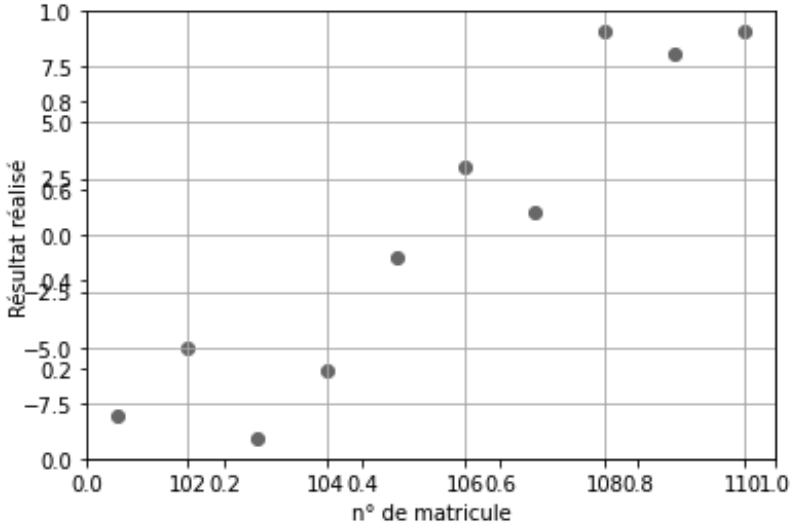
```
#Tant que le coefficient de corrélation est <0.75, regénérer une nouvelle
série
while(abs(rPearson)<0.90):
    result=pd.Series([random.randint(-10, 10) for i in range(10)])
    rPearson=matricule.corr(result)
    print("rPearson = ",round(rPearson,2),"\n","rPearson >= 0.90
?",rPearson >= 0.90)

# Construction d'un Data Frame
xy = pd.DataFrame({'Matricule': matricule, 'Résultat': result})
print(xy)
print("rPearson = ", round(rPearson,2))

# Graphique
print("Graphique")
plt.ylabel("Résultat réalisé")
plt.xlabel("n° de matricule")
axes = plt.axes()
axes.grid()
plt.scatter(matricule,result)
plt.show()
```

3.4.5 Test statistique

```
rPearson =  -0.08
 rPearson >= 0.90 ? False
rPearson =  0.21
 rPearson >= 0.90 ? False
rPearson =  0.19
 rPearson >= 0.90 ? False
rPearson =  0.94
 rPearson >= 0.90 ? True

   Matricule   Résultat
0        101        -8
```

1 102 -5
2 103 -9
3 104 -6
4 105 -1
5 106 3
6 107 1
7 108 9
8 109 8
9 110 9
rPearson = 0.94

Graphique



4 Les structures de données

Il est possible de regrouper plusieurs données élémentaires dans des structures sous Python. Ces structures sont dites des structures de données. Il en existe plusieurs :

- listes ;
- tableaux ;
- ensembles ;
- tuples ;
- dictionnaires ;
- mots (chaînes de caractères alphanumériques) ;

- piles ;
- files.

D'autres structures plus avancées sont disponibles sous Python, à condition d'importer les librairies adéquates, comme les *tensors* ou encore les *Data Frames* utilisés spécialement en *Machine Learning* et *Deep Learning*. On en utilisera certaines dans des exemples.

Le choix et l'utilisation d'une structure ou d'une autre dépend de plusieurs facteurs, comme :

- le contexte d'utilisation (applications scientifiques, mathématiques, statistiques, analyse textuelle, …) ;
- l'importance de l'ordre de stockage des données dans une structure (repérer un $i^{ème}$ élément de la structure, ordre d'arrivée et de sortie, …)
- le type de données dans la structure (numérique, alphanumérique, homogène, hétérogène, …).

4.1 Les listes

4.1.1 Définition et caractéristiques

Une liste est une collection de données telle que :

- une liste est de type *list* ;
- les données d'une liste peuvent être de types différents (données hétérogènes) ;
- les données d'une liste sont indexées (on peut repérer un ième élément dans la liste) ;
- le premier élément, c'est-à-dire la première donnée d'une liste se trouve à l'emplacement dont l'index est 0 ;
- les éléments sont accessibles et modifiables (lecture, écriture) ;
- la taille ou le nombre d'éléments de la liste est dynamique (on peut ajouter ou supprimer des éléments) ;
- l'ajout d'éléments se fait toujours à la fin de la liste ;
- on peut manipuler une liste de données élémentaires, ou bien une liste de listes ou bien une liste de listes et de données élémentaires.

4.1.2 Création et initialisation de listes

Voici, sous Shell, quelques créations/initialisations de listes :

```
# Création d'une liste vide
l=[]
```

```
# Initialisation d'une liste
l=[-1]*10
l
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]

# Création et initialisation d'une liste
L=[5,'a','b']
L
[5, 'a', 'b']
La=list(range(3,39,3))
La
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36]
S=['a','x50',20,True,15.5, L]
S
['a', 'x50', 20, True, 15.5, [5, 'a', 'b']]

# Type de données
type(S)
<class 'list'>
for i in range(len(S)):
    print("type de S[",i,"] : ",type(S[i]))
type de S[ 0 ] :  <class 'str'>
type de S[ 1 ] :  <class 'str'>
type de S[ 2 ] :  <class 'int'>
type de S[ 3 ] :  <class 'bool'>
type de S[ 4 ] :  <class 'float'>
type de S[ 5 ] :  <class 'list'>
```

4.1.3 Opérations sur liste

Voici quelques opérations sur les listes :

Opération	**Remarque**
liste.append(x)	Ajoute l'élément x à la fin de la liste
liste.sort()	Trie les éléments de la liste dans l'ordre croissant