Quelques points de programmation avec python

NB : On travaille avec python 3.

1 Généralités

1.1 Écrire un programme

Un programme est une suite d'instructions.

On appelle en général langage de programmation un logiciel (donc aussi un programme) qui va lire ces instructions et agir en conséquence (afficher des choses à l'écran, faire des bruits, écrire des données dans des fichiers, etc.) L'ordinateur traite la liste des instruction en gros dans l'ordre où elles sont écrites.

Nous allons parler succinctement ici de trois types d'instructions : Affectation d'une valeur à une variable. Test. Boucle.

1.2 Un point sur les variables

Il est commode (même si c'est en réalité inexact) de voir une variable comme une case mémoire.

Par exemple, affecter 3,1 à la variable a revient à placer 3,1 dans la case mémoire a.

Chaque variable a un certain type : nombre, chaîne de caractères, liste, etc. Pour nous, au début, c'est surtout le type nombre qui est intéressant. Mais en fait, il y a plusieurs types de nombres. Les langages font en général la différence entre un « réel » ou « flottant » (type « float ») et un entier (type « integer »).

Le terme « réel » ou « flottant » désigne un nombre quelconque, ou une approximation décimale, écrite en notation scientifique si le nombre est très grand ou très petit. Un « entier » est un nombre entier relatif.

En python, il y a une instruction qui permet de connaître le type d'une variable. Elle s'appelle type.

```
>>> a=3.0
>>> type(a)
<class 'float'>
```

Remarque 1 : Les caractères >>> en début de ligne signalent qu'on est en mode interactif (console).

Déclaration : Dans beaucoup de langages, il faut déclarer chaque variable utilisée, avec son type. En python, ce n'est pas nécessaire : le langage déclare tout seul la variable et son type au moment où on l'utilise (« typage dynamique ».)

Remarque 2 : Le séparateur décimal est un point, pas une virgule. Pour que python comprenne qu'on parle d'un nombre « flottant », et non pas d'un « entier », il suffit donc de l'écrire avec un point.

Remarque 3 : En python, une liste se signale par des crochets, par ex : `[0, 2.4, 8.6]`.

1.3 Affectation d'une valeur à une variable

En python, c'est le signe égal qui désigne l'affectation.

```
>>> a=1
>>> print(a)
1
```

Remarque 4 :

Regardons à présent ce qui suit :

```
>>> a=a+1
>>> print(a)
2
>>> b=b+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Python ajoute la valeur de `a` (1 pour l'instant) et 1. Ça donne 2, c'est la nouvelle valeur de `a`.

Par contre, l'instruction « `b=b+1` » lui demande d'ajouter 1 à la variable `b`, mais la variable `b` n'est pas définie. Il faut lui donner auparavant une valeur numérique.

Au fait, comment supprimer une variable après qu'on l'a définie (par exemple, si elle prend trop de place en mémoire) ? Facile :

```
>>> del a
```

1.4 Les tests

```
>>> a=3
>>> a==0
False
>>> a!=0
True
>>> a>=0
True
```

Attention : le test d'égalité nécessite deux signes « `=` » (pour le distinguer d'une affectation). « Différent » s'écrit « `!=` ». Les autres symboles sont faciles : « `<` », « `<=` », etc.

Exemple d'utilisation : taper les instructions suivantes dans un fichier `nom_de_fichier.py`.

```
a=float(input("Que vaut a ? "))
if a==0:
    print("a est positif et négatif")
elif a<0:
    print("a est négatif")
else:
    print("a est positif")
```

Puis exécuter les instructions (par exemple en tapant `python3 nom_de_fichier.py` dans un terminal.)

Remarque 5 : En python, les blocs d'instructions correspondant à un `if`, un `elif`, un `else`, un `while`, etc. sont déterminés par l'indentation : il faut insérer un certain nombre d'espaces en début de ligne, le même nombre pour toutes les instructions.

Par exemple, toutes les instructions à exécuter quand la condition du `if` est satisfaite seront précédées de 4 espaces.

Remarque 6 : Bien noter les deux points à la fin de chaque condition.

On voit facilement ce que fait ce programme. Précisons quand même : La première instruction, `input`, attend une saisie au clavier, le texte entre parenthèses s'affiche à l'écran. Le résultat est une chaîne de caractères qu'il faut convertir en un nombre : c'est ce que réalise l'instruction `float`.

Pour mieux comprendre :

```
>>> a=3
>>> print(a, type(a))
3 <class 'int'>
>>> b=str(a)
>>> print(b,type(b))
3 <class 'str'>
>>> print(a+1)
4
>>> print(b+1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

La variable `a` contient le *nombre* 3, on peut y ajouter 1. La variable `b` contient le *caractère* 3, c'est seulement un symbole.

```
>>> b=str(3)
>>> print("Résultat : "+b+" bananes")
Résultat : 3 bananes
```

Entre chaînes de caractères, le symbole « `+` » signifie qu'on les assemble l'une à la suite de l'autre.

1.5 Les boucles « tant que »

La condition commence par le mot-clé : `while`.

Exemple : le programme suivant calcule et affiche les termes d'une suite géométrique de premier terme 1 et de raison 1,1 et s'arrête lorsque le calcul donne un nombre supérieur à 3.

Les termes successifs sont stockés dans la liste `termes`. L'instruction `termes.append(a)` sert à placer le nouveau terme calculé à la fin de la liste.

```
a=1.0
termes=[a]
while a <= 3:
    a = 1.1*a
    termes.append(a)
print("Fin du calcul")
```

Rappel : En python, les deux instructions `a = 1.1*a` et `termes.append(a)` doivent être précédés du même nombre d'espaces, sinon le programme plante avec un message du type :

```
    File "...", line ...
      print(a)
            ^
  IndentationError: unindent does not match any outer indentation level
```

Remarque 7 : L'usage des boucles « tant que » demande quelques précautions. D'abord, il faut avoir initialisé correctement la variable sur laquelle porte la condition. Ensuite, il faut s'assurer que le programme sortira effectivement de la boucle à un moment ou à un autre, ce qui n'est pas forcément évident.

1.6 Les boucles « pour »

La condition commence par le mot-clé : `for`.

Exemple : le programme suivant calcule et affiche 10 ! (factorielle de 10), c'est à dire $10 \times 9 \times 8 \times \cdots \times 2 \times 1$.

```
a=1
for compteur in range(1,11):
    a=a*(compteur)
print(a)
```

Remarque 8 : En python, l'instruction `for` a une forme un peu particulière. Au lieu d'avoir quelque chose du genre `for compteur from 1 to 10`, on a `for compteur in range(1,11)`.

Pour le comprendre, regardons ce que c'est que ce `range(1,11)` :

```
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ainsi, `range(1,11)` crée une liste de 10 nombres entiers consécutifs commençant à 1. La variable compteur prend successivement toutes les valeurs de cette liste.

Une autre solution est d'écrire :

```
a=1
for compteur in range(10):
    a=a*(compteur+1)
print(a)
```

En effet :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On peut aussi spécifier un pas (entier) :

```
>>> list(range(5,21,3))
[5, 8, 11, 14, 17, 20]
```

Cette instruction a l'air peu intuitive a priori ; en fait, elle est très pratique. Exemple :

```
>>> liste=["papa", "maman", "Titi", "Toto"]
>>> for i in liste:
...     print("Bonjour "+i)
...
Bonjour papa
Bonjour maman
Bonjour Titi
Bonjour Toto
```

Ici la variable `i` parcourt une liste de mots et non pas une liste de nombres. Le programme est plus lisible et compréhensible qu'avec un compteur numérique.

2 Graphiques

Premier exemple, très simple :

Il utilise la bibliothèque `mpmath`.

```
from mpmath import plot
from mpmath import sin
def f(x): return sin(x**2) # Définition d'une fonction ; voir ci-après
plot([f],points=500)
```

Deuxième exemple :

Il utilise la bibliothèque `matplotlib`. On obtient la représentation graphique de sin entre $-\pi$ et π.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-np.pi, np.pi, 0.1)
y = np.sin(x)
plt.plot(x, y)
plt.grid()
plt.show()
```

La bibliothèque `numpy` permet surtout de faire directement des calculs mathématiques sur de gros tableaux de nombres. On préfixe les instructions avec le nom de ces bibliothèques ; « np » et « plt » sont des alias un peu plus courts.

L'instruction `x = np.arange(-np.pi, np.pi, 0.1)` produit un tableau de valeurs entre $-\pi$ et π avec un pas de 0,1. L'instruction `y = np.sin(x)` retourne le tableau des images de `x` par la fonction sinus.

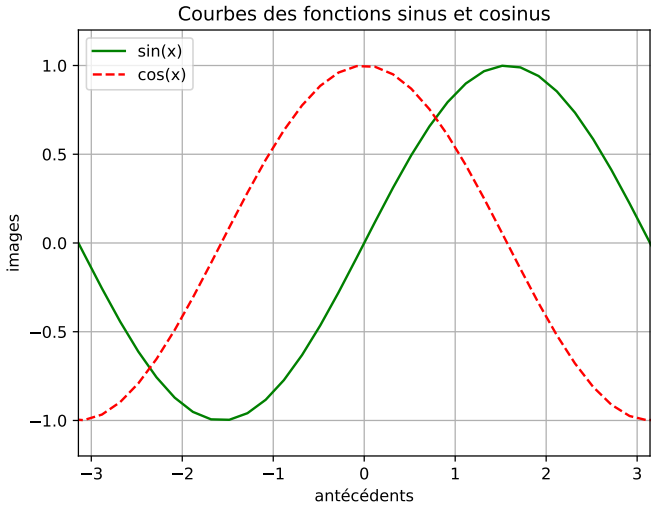
L'instruction `plt.plot(x,y)` affiche les points de coordonnées (x ; y). L'instruction `plt.grid()` affiche une grille. L'instruction `plt.show()` génère la fenêtre comportant le graphique.

On peut remplacer `plt.plot(x, y)` par `plt.plot(x, y,'.')` pour avoir des points au lieu d'une courbe continue.

Troisième exemple :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.arange(-2*np.pi, 2*np.pi, 0.2)
5  y1 = np.sin(x)
6  y2 = np.cos(x)
7  plt.plot(x, y1,'g-',label="sin(x)") # 'g-' Signifie que le tracé est en vert (green)
8  # label="sin(x)" : Donne un nom à la courbe
9  plt.plot(x, y2,'r--',label="cos(x)") # 'r--' : Le tracé est en pointillé rouge (red)
10 plt.xlabel("antécédent") # Légende sur l'axe des abscisses
11 plt.ylabel("images") # et sur l'axe des ordonnées
12 plt.title("Courbes des fonctions sinus et cosinus") # Titre du graphique
13 plt.axis([-np.pi,np.pi,-1.2,1.2]) # Définit la fenêtre : [xmin, xmax, ymin, ymax]
14 plt.grid() # Affiche la grille
15 plt.legend(loc=2) # Affiche le nom des courbes
16 plt.show()
```

Ligne 15 : Le paramètre entre parenthèses est optionnel et peut prendre les valeurs 0 à 10, correspondant à diverses positions de la légende sur le graphique.



Les boutons en bas de la fenêtre graphique permettent de modifier l'affichage de façon interactive. En cliquant sur la croix (4e bouton), on peut ensuite déplacer la courbe (clic-gauche), la compresser ou la dilater (clic-droit). On peut aussi zoomer à l'aide de la 5e touche (loupe), en traçant un cadre à la souris.

La première touche (maison) permet de retrouver la fenêtre dans son état initial et les deux flèches permettent de retrouver les différentes valeurs de zoom utilisées. La 7e touche permet de sauver le graphique.

Quatrième exemple :

Représenter graphiquement les premiers termes de deux suite (a_n) et (b_n) donnant des approximations successives de $\sqrt{2}$ par la méthode de dichotomie. On pose donc : $a_0 = 1$; $b_0 = 2$; à chaque étape, on calcule $c = \dfrac{a_n + b_n}{2}$.

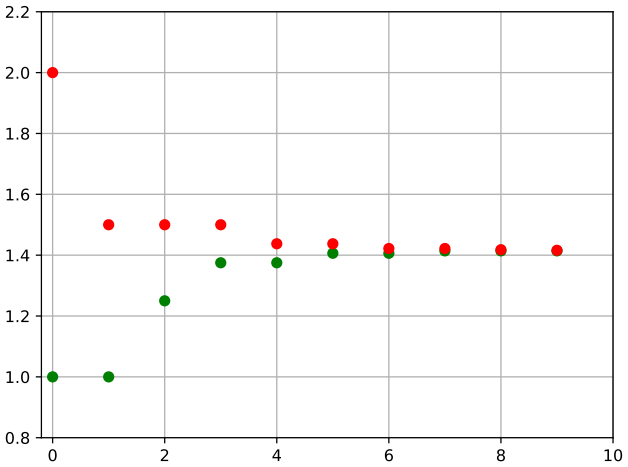
Si $c^2 \leq 2$ alors $\sqrt{2}$ est entre c et b_n donc on pose $a_{n+1} = c$; sinon $\sqrt{2}$ est entre a_n et c donc on pose $b_{n+1} = c$.

```
import numpy as np
import matplotlib.pyplot as plt
a,b=1.0,2.0
liste_a=[] # cette liste contiendra a_0, a_1, ... a_9
liste_b=[] # cette liste contiendra b_0, b_1, ... b_9
for i in range(10):
    liste_a.append(a) # ajoute une nouvelle valeur à liste_a
    liste_b.append(b) # ajoute une nouvelle valeur à liste_b
    c=(a+b)/2
    if c**2<=2:
        a=c
    else:
        b=c
    print(a,b,b-a)

plt.plot(liste_a,'go') # Points dont les ordonnées sont les valeurs de liste_a
plt.plot(liste_b,'ro') # Points dont les ordonnées sont les valeurs de liste_b
plt.grid(True)
plt.axis([-0.2,10,0.8,2.2])
plt.show()
```

On n'a pas écrit les abscisses : lorsqu'elles sont absentes, le programme considère que ce sont les nombres entiers 0, 1, etc.(c'est ce qui se passe aussi avec un tableur.) Si on avait voulu les préciser, on aurait écrit :

```
liste_n=range(10)
plt.plot(liste_n, liste_a,'go')
plt.plot(liste_n, liste_b,'ro')
```



3 Définition d'une fonction

On a vu (page 5) comment représenter graphiquement la fonction sinus. Si on veut représenter autre chose (un polynôme, par exemple), il est souhaitable de changer cette instruction, sans modifier le reste.

Le moyen le plus simple est d'utiliser ce qu'on appelle une fonction (au sens algorithmique). C'est un morceau de code qu'on place à part, pour une raison ou une autre.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def ma_fonction(t):
5     return t**2+3*t-4
6
7 x = np.arange(-4, 4, 0.1)
8 y = ma_fonction(x)
9 plt.plot(x, y)
10 plt.grid()
11 plt.show()
```

Ligne 4 : On définit la fonction par l'instruction `def`. Le nom `ma_fonction` peut être remplacé par n'importe quoi, à part les mots réservés du langage. Entre parenthèse, on place ce qu'on appelle le (ou les) paramètre(s) de la fonction. Ici, `t` est un tableau complet de valeurs.

Remarque 9 : Bien noter les deux points et l'indentation du corps de la fonction. C'est la même technique que pour une boucle ou un test. En revenant à l'indentation normale (en cessant de commencer la ligne par 4 espaces), on signale à python que la définition de la fonction est terminée.

Ligne 5 : La valeur que retourne la fonction est définie par ce qui suit l'instruction `return`. Noter que `t` à la puissance 2 se note `t**2` (et non pas `t^2`). Cette notation se retrouve dans d'autres langages de programmation.

4 Sauvegarde des données dans un fichier

On souhaite pouvoir écrire puis lire les données issues d'un calcul dans un fichier. Reprenons l'exemple de deux suite (a_n) et (b_n) donnant des approximations successives de $\sqrt{2}$ par la méthode de dichotomie (voir page 6). On souhaite écrire trois colonnes séparées par des espaces, avec dans l'ordre : n, a_n, b_n.

4.1 Première méthode : instructions de lecture/écriture standard de python.

Python peut écrire et lire des fichiers textes ou des fichiers binaires. Dans un fichier texte, les données sont stockées sous forme de caractères ; on peut donc lire ce fichier avec n'importe quel éditeur de texte. Dans un fichier binaire, les données sont directement stockées sous forme de nombres, sans codage permettant de rendre ces données lisibles. On ne peut donc pas les lire directement.

On va ici sauver nos données dans un fichier texte. Pour cela, les nombres utilisés seront d'abord codés sous forme de chaîne (string) de caractères grâce à la fonction `str()`.

Illustration :

```
>>> a=2.3
>>> print(a, type(a))
2.3 <class 'float'>
>>> b=str(a)
>>> print(b,type(b))
2.3 <class 'str'>
```

Vérifions que `b` n'est pas un nombre :

```
>>> print(a+2)
4.3
>>> print(b+2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Comme prévu, python peut calculer `a+2` mais pas `b+2` : il ne peut pas ajouter une chaîne de caractère et un nombre. À présent, voyons le programme.

Écriture :

```
1 liste_a=[1.0, 1.0, 1.25, 1.375, 1.375, 1.4062, 1.4063, 1.4141]
2 liste_b=[2.0, 1.5, 1.5, 1.5, 1.4375, 1.4375, 1.4219, 1.4219]
3
4 with open('donnees.txt', 'w') as mon_fichier:
5     for n in range(8):
6         mon_fichier.write(str(n)+' ')
7         mon_fichier.write(str(liste_a[n])+' ')
8         mon_fichier.write(str(liste_b[n])+'\n')
```

Le répertoire de travail contient maintenant un fichier du nom de `donnees.txt`.

Ligne 4 : on ouvre un objet-interface particulier que l'on appelle `mon_fichier` (on peut lui donner n'importe quel nom autorisé, comme pour une variable). Le nom `donnees.txt` est le nom du fichier reconnu par le système d'exploitation, tel qu'il apparaît dans un explorateur de fichiers. On ajoute 'w' pour signifier que ce fichier est ouvert en écriture (write).

Ligne 6 : on écrit la valeur de n convertie en chaîne de caractères par l'instruction `str`. On ajoute une espace à la fin : `' '`.

Ligne 7 : on écrit la valeur de a_n convertie de la même façon en chaîne de caractères. Remarquons que

l'instruction `liste_a[n]` renvoie l'élément n° n de la liste `liste_a` (attention : on numérote à partir de 0.)

Ligne 8 : même chose pour b_n ; on ajoute un retour à la ligne à la fin : `'\n'`.

NB : Lorsqu'on ouvre un fichier en écriture avec l'instruction 'w', le fichier est réécrit depuis le début. Cela veut dire que si des données étaient déjà stockées dans le fichier, elles sont écrasées.

Si l'on veut conserver les données existantes et ajouter de nouvelles données à la fin du fichier, il faut remplacer le `'w'` par `'a'` (append : ajouter). Notons aussi que `'r+'` ouvre le fichier en lecture et en écriture.

Il suffirait donc de remplacer la ligne 13 par celle-ci : `with open('donnees.txt', 'a') as mon_fichier:`

Lecture :

```
with open('donnees.txt', 'r') as mon_fichier: # on ouvre le fichier en lecture (read)
    texte = mon_fichier.read() #on lit le fichier.
print(texte)
```

4.2 Deuxième méthode : utilisation des instructions de lecture/écriture de numpy.

Les fonctions de lecture/écriture précédentes permettent d'écrire n'importe quel type de données. La bibliothèque `numpy` propose des fonctions plus adaptées lorsque les données sont de type numérique.

Écriture :

```
1 import numpy as np
2 liste_n=[0, 1, 2, 3, 4, 5, 6, 7]
3 liste_a=[1.0, 1.0, 1.25, 1.375, 1.375, 1.4062, 1.4063, 1.4141]
4 liste_b=[2.0, 1.5, 1.5, 1.5, 1.4375, 1.4375, 1.4219, 1.4219]
5
6 donnees=np.array([liste_n,liste_a,liste_b])
7 donnees=donnees.transpose() # ou donnees=donnees.T, ou donnees=np.transpose(donnees)
8 np.savetxt('fichier_de_donnees.txt',donnees, fmt='%8.4g')
```

Ligne 6 : On fabrique un tableau (array) `numpy` en « empilant » les trois listes : `liste_n,liste_a,liste_b`.

En fait, on a créé une liste de listes grâce aux crochets `[]`.

On voudrait que les données soient rangées en colonne. Pour le moment, on a une ligne pour `liste_n`, puis une ligne pour `liste_a`, etc.

Ligne 7 : Pour avoir la présentation désirée, il suffit de permuter les lignes et les colonnes du tableau. Ceci se réalise grâce à la méthode (fonction) `transpose()`.

Ligne 8 : on écrit les données dans le fichier grâce à la fonction `savetxt()`. Le premier argument est le nom du fichier créé sur le disque, le deuxième argument est le tableau de données à sauvegarder, le troisième, ici, indique le format désiré.

L'expression `fmt='%8.4g'` se décompose ainsi :

- `8` signale qu'on utilise au moins 8 espaces pour écrire un nombre,
- `.4` qu'on utilise 4 décimales,
- et le `g` signifie que le programme va utiliser la notation décimale ou la notation scientifique suivant les cas , de façon à prendre le moins d'espace possible.

On peut remplacer `g` par `f` (notation décimale), `e` (notation scientifique), `i` (entier), `x` (hexadécimal), etc.

Il y a d'autres arguments, par exemple, les nombres sont séparés par défaut par une espace ; on peut la remplacer par une virgule ou tout autre caractère. Voir la documentation `numpy`.

Lecture :

```
1 import numpy as np
2
3 donnees_lues=np.loadtxt('fichier_de_donnees.txt')
4 x=donnees_lues[:,0]
5 y=donnees_lues[:,1]
6 z=donnees_lues[:,2]
```

Ligne 3 : L'instruction `donnees_lues=loadtxt('fichier_de_donnees.txt')` permet de lire le fichier et de placer son contenu dans le tableau `donnees_lues`. C'est tout. On peut le vérifier en exécutant un `print(donnees_lues)`.

Ligne 4 : On place le contenu de la première colonne de `donnees_lues` dans la variable `x` ; `x` est donc un tableau, et correspond à `liste_n`.

Le ou les éléments à copier sont repérés par deux coordonnées : [n° de ligne, n° de colonne]. Attention : on numérote à partir de zéro. Comme pour un tableur, les deux points (`:`) signifient une plage de cellules.

Précision : le codage `i:j` signifie de `i` à `j`, `i` inclus et `j` exclus. Attention aux erreurs.

Exemples : `M[0,2]` est le nombre de la première ligne, 3e colonne du tableau `M`.

`M[0:2,3]` : tableau constitué des nombres des 2 premières lignes et 4e colonne de M.

Les deux points utilisés seuls signifient qu'on prend toute les lignes ou toutes les colonnes, suivant le cas.

`M[:,3]` : tableau constitué de la 4e colonne.

`M[2,:]` : tableau constitué de la 3e ligne.

On peut également numéroter à partir de la fin (avec un $-$).

`M[:,-1]` : tableau constitué de la dernière colonne.

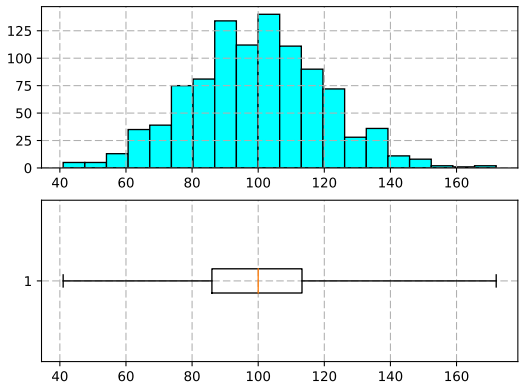
Lignes 6 et 7 : c'est le même codage, on place la 2e colonne dans `y` et la 3e dans `z`.

5 Statistiques

Numpy calcule les indicateurs basiques pour une série donnée sous forme de tableau. Les valeurs ne sont pas pondérées (l'effectif est toujours de 1.)

```
import numpy as np
import matplotlib.pyplot as plt
#On fabrique un tableau de valeurs entières aléatoires suivant une loi normale :
data=np.round(100+20*np.random.randn(1000))
#Calcul de la moyenne :
print("moyenne = ",np.mean(data))
#Calcul de l'écart-type :
print("écart-type = ",np.std(data))
#Calcul de la variance :
print("variance = ",np.var(data))
#Calcul de la médiane :
print("médiane = ",np.median(data))
#Calcul du premier quartile :
print("premier quartile = ",np.percentile(data,25,interpolation='higher'))
#Calcul du 3e quartile :
print("troisième quartile = ",np.percentile(data,75,interpolation='higher'))

#Histogramme de la série, avec 20 classes:
plt.subplot(211)
plt.grid(linestyle="--")
plt.hist(data,20,facecolor="cyan", edgecolor="black")
#Boîte à moustaches :
plt.subplot(212)
plt.grid(linestyle="--")
plt.boxplot(data, vert=False, whis=1e12)
#whis*écart interquartile est la distance à la boîte au-delà de laquelle
#les valeurs sont considérées comme isolées
plt.show()
```



On peut aussi définir manuellement chaque classe :

```
...
classes=[0,60,80,90,100,110,120,140,200] # classes prédéfinies,
plt.hist(data,bins=classes,density=True) #density=True -> fréquences
plt.show()
```

6 Probabilités

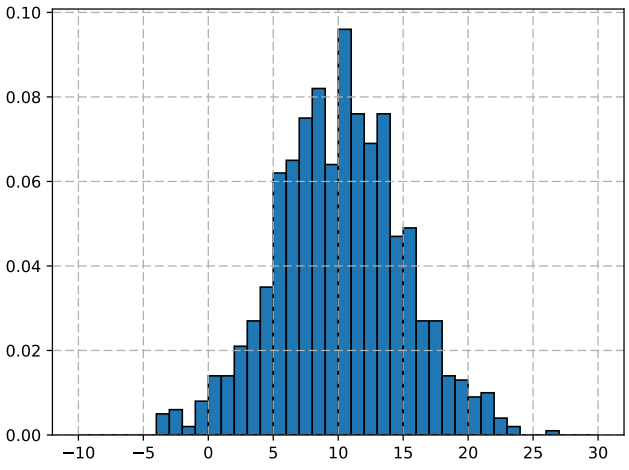
6.1 Loi normale

Utilisation du module `scipy`. Pour la loi normale et les lois continues, `cdf` = fonction de répartition, `ppf` = fractile et `pdf` = densité ; `rvs` génère un échantillon aléatoire de X.

```
>>> from scipy.stats import norm # La bibliothèque nécessaire
>>> norm(10,5).cdf(20) # P(X<20), où X suit N(10;5^2)
0.9772498680518208
>>> norm.cdf(20, loc=10, scale=5) # autre façon ; loc=mu ; scale = sigma
0.9772498680518208
>>> norm(10,5).ppf(0.84) # P(X<t) =0.84 pour t≃14.97
14.972289416048765
>>> norm.pdf(0.8) # densité
0.28969155276148273
>>> norm(10,5).rvs(20) # échantillon de taille 20
array([10.72447425,  6.1996473 , 10.83090666,  3.18010144, 12.70592693,
        9.73568646, 16.4029409 ,  8.02085713, 12.618044  ,  7.38159327,
        1.47643509,  8.10904058, 11.07446039,  7.81567731,  6.79531014,
        9.74046132,  8.09500916,  3.10928907, 20.9866777 , 11.31959919])
```

Un histogramme :

```
from scipy.stats import norm
import matplotlib.pyplot as plt
r=norm(10,5).rvs(1000)
plt.hist(r,bins=[x-10 for x in range(41)],normed=True,edgecolor="black")
plt.grid()
plt.show()
```



Il existe aussi `expon` pour la loi exponentielle, `uniform`, etc.

6.2 Loi binomiale

Pour la loi binomiale et les lois discrètes, `cdf` = fonction de répartition, `ppf` = fractile, `pmf` = probabilité ponctuelle, `rvs` génère un échantillon aléatoire.

```
>>> from scipy.stats import binom

>>> binom(20,0.4).pmf(5) # P(X=5) où X suit B(20;0.4)
0.07464701952887058
>>> binom.pmf(5,20,0.4) #Idem
0.07464701952887058
>>> binom.cdf(5,20,0.4) # P(x<=5)
0.12559897272303736
>>> a=int(binom(50,0.4).ppf(0.025)) # Bornes d'un intervalle de
>>> b=int(binom(50,0.4).ppf(0.975)) # fluctuation à 95%
>>> print(a,b)
13 27
>>> binom.rvs(100, 0.4, size=20) # échantillon de 20 valeurs suivant B(100,0.4)
array([42, 50, 42, 43, 42, 45, 46, 44, 41, 34, 37, 38, 38, 38, 50, 42, 38,
       50, 40, 38])
```

Avec `binom`, on a aussi `poisson`, `randint`, `geom`, etc.

7 Utilisation de sympy

`sympy` est une bibliothèque de calcul formel très intéressante.

Graphiques :

```
1 from sympy import *
2 x=symbols('x')
3 from matplotlib import style # optionnel :
4 style.use('seaborn-whitegrid') # pour avoir un quadrillage
5 plot(x**2-3*x+2, (x, -6, 6))
```

Ligne 1 : On importe toute la bibliothèque sympy, on pourra alors utiliser ses fonctions sans les préfixer, mais il vaut mieux dans ce cas l'utiliser seule (éviter les mélanges avec numpy et scipy.)

Ligne 2 : Il est nécessaire de déclarer la variable `x` en tant que symbole mathématique. Il faut le faire pour toutes les variables mathématiques employées.

```
>>> x,y=symbols('x,y') # ou var('x,y')
>>> type(x)
<class 'sympy.core.symbol.Symbol'>
```

`var("a:k")` : Déclare comme variables toutes les lettres de a à k

Ligne 5 : remarquer qu'il faut écrire les symboles de multiplication et que la puissance se note avec `**`.

Algèbre :

```
>>> expand((x+y)**3) # Développement
x**3 + 3*x**2*y + 3*x*y**2 + y**3
>>> pprint(_) # "pprint" : pretty print ; "_" rappelle le dernier résultat
 3      2          2    3
x  + 3*x *y + 3*x*y  + y
>>> factor(y*x**2+5*x**3) # Factorisation
x**2*(5*x + y)
>>> factor(x**2-5*x+4)
(x - 4)*(x - 1)
>>> a=1/(x+1)**2+1/(x+3); b=together(a) # Réduire au même dénominateur
>>> b, apart(b)
((x + (x + 1)**2 + 3)/((x + 1)**2*(x + 3)), 1/(x + 3) + (x + 1)**(-2))
>>> solve(x**2-2,x) # Résout l'équation : x^2-2=0
[-sqrt(2), sqrt(2)]
>>> solve(Eq(x**2,x+1),x) # Résout x^2=x+1 ; solve(x**2=x+1,x) produit une erreur
[1/2 - sqrt(5)/2, 1/2 + sqrt(5)/2]
>>> solve(x**2+2,x) # Le i complexe se note I
[-sqrt(2)*I, sqrt(2)*I]
>>> solve([x-3*y+2,5*x+y-4]) # Un système
{x: 5/8, y: 7/8}
```

Calcul numérique :

```
>>> x/2+x/3, 1/2+1/3 # On voudrait calculer directement dans Q
(5*x/6, 0.8333333333333333)
>>> a=Rational(1,2)+Rational(1,3) ; b= S(5)/2 +S("4/7") # S : raccourci pour "sympify"
>>> a,b # Voilà
(5/6, 43/14)
>>> a.evalf() , N(a) # pour obtenir une valeur approchée
(0.833333333333333, 0.833333333333333)
>>> pi.evalf(4) , N(5*E,4) # En paramètre, le nombre de chiffres significatifs
(3.142, 13.59)
>>> S("354/168") # Simplification automatique
59/28
```

Analyse :

```
>>> sin(pi/3), cos(pi/3), tan(pi/4) # pi se note pi
(sqrt(3)/2, 1/2, 1)
>>> n=symbols('n', integer=True) # n est défini comme un entier
>>> sin(n*pi), cos(n*pi)
(0, (-1)**n)
>>> log(1), log(E), exp(1) # e=exp(1) se note E
(0, 1, E)
>>> expand(log(3*E**5))
log(3) + 5
>>> simplify(E**3*E**x/E**y)
exp(x - y + 3)
>>> f=x**2*sin(x) # Définition d'une fonction
>>> f.subs(x,pi), f.subs(x,pi/2) # Calcul d'images
(0, pi**2/4)
>>> diff(f,x) # Dérivée
x**2*cos(x) + 2*x*sin(x)
>>> integrate(f,x) # Primitive
-x**2*cos(x) + 2*x*sin(x) + 2*cos(x)
>>> integrate(f,(x,0,pi)) # Intégrale
-4 + pi**2
>>> limit(sin(x)/x,x, 0) # Limite simple
1
>>> limit(1/x,x,0,dir="+") # Limite à droite ; +∞ se note oo
oo
>>> limit(1/x,x,0,dir="-") # Limite à gauche ; -∞ se note -oo
-oo
>>> limit(tan(x),x,pi/2,dir="-"),limit(tan(x),x,pi/2,dir="+")
(oo, -oo)
>>> series(tan(x),x,0,10) # Développement limité
x + x**3/3 + 2*x**5/15 + 17*x**7/315 + 62*x**9/2835 + O(x**10)
```

Trigonométrie :

```
>>> expand(sin(x+y)), expand_trig(sin(x+y)) # Développement avec expand_trig
(sin(x + y), sin(x)*cos(y) + sin(y)*cos(x))
>>> trigsimp(cos(x)+sin(x))
sqrt(2)*sin(x + pi/4)
>>> expand_trig(cos(5*x))
16*cos(x)**5 - 20*cos(x)**3 + 5*cos(x)
```

Complexes :

```
>>> E**(I*pi), E**(I*pi/2)
(-1, I)
>>> a=1+I ; b=2-3*I
>>> a+b, a*b, a**4
(3 - 2*I, (1 + I)*(2 - 3*I), (1 + I)**4)
>>> expand(a*b), expand(a**4)
(5 - I, -4)
>>> re(a), im(a), abs(a), arg(a)
(1, 1, sqrt(2), pi/4)
>>> conjugate(b), abs(b), arg(b)
(2 + 3*I, sqrt(13), -atan(3/2))
>>> expand(E**(I*pi/3))
exp(I*pi/3)
>>> expand(E**(I*pi/3),complex=True)
1/2 + sqrt(3)*I/2
```

Divers :

```
>>> t = symbols('t', positive=True)
>>> sqrt(t**2)
t
>>> gcd(12,16) # PGCD
4
>>> lcm(15,10) # PPCM
30
>>> prime(10) # prime(n) : nième nombre premier
29
>>> isprime(51) # Teste si n est premier
False
>>> factorint(18) # Décompose en produit de facteurs premiers
{2: 1, 3: 2}
>>> factorint(18, visual=True)
2**1*3**2
>>> primefactors(18) # Donne les facteurs premiers sans leur multiplicité
[2, 3]
>>> divisors(18) # Donne tous les diviseurs
[1, 2, 3, 6, 9, 18]
```

Matrices :

```
>>> u=Matrix([5,7,1]) ; v=Matrix([[4,1,3]]) ; A=Matrix([[1,7,8],[0,1,-3],[4,5,0]])
>>> A
Matrix([
[1, 7,  8],
[0, 1, -3],
[4, 5,  0]])
>>> A*u, v*A
(Matrix([
[62],
[ 4],
[55]]), Matrix([[16, 44, 29]]))
>>> A**2
Matrix([
[ 33,  54, -13],
[-12, -14,  -3],
[  4,  33,  17]])
>>> A.transpose()
Matrix([
[1,  0, 4],
[7,  1, 5],
[8, -3, 0]])
>>> A.det()
-101
>>> A.inv() # ou A**-1
Matrix([
[-15/101, -40/101, 29/101],
[ 12/101,  32/101, -3/101],
[  4/101, -23/101, -1/101]])
>>> A.LUsolve(u) # Système
Matrix([
[-326/101],
[ 281/101],
[-142/101]])
>>> B=4*eye(3)+2*ones(3) ; B # eye : identité ; ones, zeros...
Matrix([
[6, 2, 2],
[2, 6, 2],
[2, 2, 6]])
>>> C=Matrix(2, 3, lambda i,j: i+j) ; D=Matrix(2,3,[4,7,8,3,2,0])
>>> C
Matrix([
[0, 1, 2],
[1, 2, 3]])
>>> C.shape
(2, 3)
>>> E=diag(A[0:2,0:2],9) ; E
Matrix([
[1, 7, 0],
[0, 1, 0],
[0, 0, 9]])
```

Diagonalisation, valeurs propres :

```
>>> A=Matrix([[3,8,1],[12,7,-6],[10,10,0]])
>>> p=A.charpoly(x) # Polynôme caractéristique
>>> factor(p)
(x - 10)*(x - 5)*(x + 5)
>>> B=A.eigenvects() # Valeur propre, multiplicité, vecteur(s) propre(s)
>>> #for k in B:
>>> #    print("valeur propre : {}, multiplicité : {}".format(k[0],k[1]))
>>> #    print("vecteur propre : \n{}".format(pretty(k[2])))

>>> P,D=A.diagonalize() # A=P.D.P^(-1)
>>> P
Matrix([
[-1, 1, 3],
[ 1, 0, 2],
[ 0, 2, 5]])
>>> D
Matrix([
[-5, 0,  0],
[ 0, 5,  0],
[ 0, 0, 10]])
```

8 Utilisation de tkinter, une interface graphique

Le code suivant affiche une fenêtre :

```
from tkinter import * #Appel de la bibliothèque.
#-----------------------------------------------
fenetre = Tk() # Création de la fenêtre principale
#-----------------------------------------------
fenetre.mainloop() # Départ du gestionnaire d'événements
```

Maintenant, il faut mettre des choses dedans (ce sont les "widgets".) On ajoute un bouton « quitter » qui ferme la fenêtre.

```
fenetre = Tk()
#-----------------------------------------------
bouton_quit=Button(fenetre, text="quitter", command=fenetre.destroy)
bouton_quit.pack() # Le bouton est placé dans la fenêtre
#-----------------------------------------------
fenetre.mainloop()
```

Création d'une zone de texte simple :

```
fenetre = Tk()
bouton_quit=Button(fenetre, text="quitter", command=fenetre.destroy)
bouton_quit.pack()
#-----------------------------------------------
affichage=Label(fenetre, text="La raison du plus fort est toujours la meilleure...")
affichage.pack()
#-----------------------------------------------
fenetre.mainloop()
```

Création d'une zone de texte et récupération des caractères entrés :

```
def recopie(event):
    contenu=entree.get() # Les caractères tapés sont recopiés
                         # dans la variable 'contenu'.
    liste_frappe.append(contenu) # Puis stoqués dans une liste
    l=len(contenu)
    entree.delete(0,l) # On efface la zone d'entrée
#-----------------------------------------------
fenetre = Tk()
#-----------------------------------------------
entree=Entry(fenetre) # Création d'une zone d'entrée au clavier
entree.pack()
entree.bind("<Return>",recopie) # Les caractères tapés sont recopiés
entree.bind("<KP_Enter>",recopie) # à chaque appui sur une des touches "enter".
#-----------------------------------------------
liste_frappe=[]
fenetre.mainloop()
```

Même idée, en utilisant une variable `tkinter`. On affiche aussi le texte entrée dans une zone de texte.

```
def recopie(event):
    liste.append(contenu.get())
    l=len(contenu.get())
    entree.delete(0,l)

#-------------------------------------------------
fenetre = Tk()
contenu=StringVar() # Déclaration d'une variable tkinter de type chaîne
#-------------------------------------------------
affichage=Label(fenetre, textvariable=contenu)
affichage.pack()
entree=Entry(fenetre,textvariable=contenu)
entree.pack()
entree.bind("<Return>",recopie)
entree.bind("<KP_Enter>",recopie)
entree.bind("<Control-KeyPress-q>", lambda a : fenetre.destroy())
# L'appui sur Ctrl-q déclenche la fermeture de la fenêtre
#-------------------------------------------------
liste=[]
entree.focus_set() # La zone d'entrée clavier est active au début
fenetre.mainloop()
```

Les variables `tkinter` sont du type `IntVar()` (nombre entier), `DoubleVar()` (flottant), `StringVar()` (chaîne de caractère) ou `BooleanVar()`.

Le code suivant utilise une zone de type éditeur de texte. Une barre de défilement verticale ("ascenseur") permet de faire défiler les lignes.

```
from tkinter import *
from tkinter.scrolledtext import ScrolledText

le_texte=""" Mais comment exprimer cette foule de sensations fugitives...
 Levez-vous vite, orages désirés..."""
#-------------------------------------------------
fenetre=Tk()
text = ScrolledText(master=fenetre, height=30, width = 70, wrap="word", bg="#ffc")
text.pack(fill=BOTH, expand=Y)
text.insert(END, le_texte)
```

Un chronomètre rudimentaire :

```
from tkinter import *
def tempo():
    """ Chronomètre """
    global cestparti
    if cestparti:
       k.set(k.get()+1)
    fenetre.after(1000,tempo) # Instruction de temporisation, temps en ms

def marchearret():
    """ Déclenche/arrête le chrono """
    global cestparti
    cestparti = not cestparti
#-----------------------------------------------------------------
cestparti=False
fenetre = Tk()
k=IntVar()
#-----------------------------------------------------------------
bouttonMA = Button(fenetre, text='Marche/Arrêt', width =20, command=marchearret)
bouttonMA.pack()
#-----------------------------------------------------------------
boutonQuitter = Button(fenetre, text ='Quitter', width =20, command = fenetre.destroy)
boutonQuitter.pack()
#-----------------------------------------------------------------
compteur=Label(fenetre,font=('TkTextFont', '72'),textvariable=k)
compteur.pack()
#-----------------------------------------------------------------
tempo()
fenetre.mainloop()
```