Les bases de la programmation sous Python

PAR ERIC.HERBERT@U-PARIS.FR

Université Paris Cité

Résumé

Ce cours a pour but de vous permettre de produire et exploiter des données *via* le language Python.

Table des matières

1 Découverte de Python

Python est un language massivement utilisé pour toutes sortent d'applications, pour lequel une aide en ligne est facilement trouvable. Il est par ailleurs plus simple à apprendre que d'autres language et constitue pour ces raisons une bonne base d'apprentissage.

1.1 Pourquoi Python ?

1. C'est un language de script, il n'a pas besoin d'être compilé, il est interprété.

2. C'est un language de très haut niveau, c'est à dire que son expression est au plus proche de la langue écrite (américaine, évidemment)

3. C'est un language libre, c'est à dire que vous pouvez l'installer sur votre ordinateur sans licence, ou l'utiliser sur des projets commerciaux sans licence payante.

4. C'est un language très utilisé, très actif, qui permet littéralement de faire le café,[1] avec une communauté très développée qui permet d'obtenir de l'aide facilement.[2] Autrement dit, votre question a déjà été posée.

5. C'est un language implémenté en natif dans les IDE[3] classique, comme *anaconda* ou *spyder*. Ce qui permet de disposer d'environnements à jour et fonctionnel en quelques instants sur tous type d'OS.

1.2 Accéder aux données

Pour pouvoir travailler, vous allez utiliser les ressources mises à disposition dans le dépot de l'enseignement sur git-hub.

1. Aller sur `https://github.com/ericherbert/E2S_algo`

2. Lire le README. Identifier les programmes et les données

3. Télecharger le dépot *via* le bouton vert, puis choisissez *clone* ou *download*

1.3 Mon tout premier programme

Dans les exemples qui suivent, il ne s'agit pas de savoit écrire les programmes proposés mais d'avoir une première vision de la manière dont la formulation doit être faite

1. voir par exemple `https://fr1.ipp-online.org/10-major-uses-of-python-5136`, ou ce sondage sur l'usage des langages `https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted`

2. Essayez votre moteur de recherche favori. Jetez un oeil sur `https://stackoverflow.com/`

3. Vous devez comprendre les mots et acronymes utilisés.

Ouvrir *premier_programme.py* dans spyder ou dans un éditeur de texte.

```
# premier programme
a = 10
print("\t%%%%\t%%%%%\t")
print("programme␣de␣calcul␣des␣carrés")
print( a**2 )
print("\n")
```

1. Comprendre ligne à ligne ce que fait ce programme
2. L'executer, c'est à dire…
3. Que s'est il passé ?
4. le modifier et le rééxecuter, jusqu'à ce qu'il ne s'execute plus et comprendre pourquoi.

Faire de même avec *second_programme.py*.

```
# programme avec interaction
n = 3
print("Je␣vais␣vous␣demander", n, "nombres")
for i in range(n):
    x = int(input("Donnez␣un␣nombre␣:␣"))
    if x > 0:
        print(x, "est␣positif")
    else:
        print(x, "est␣négatif␣ou␣nul")
print("Fin")
```

Enfin avec le troisième programme:

```
# programme avec importation de bibliothèque,
# de données dans un fichier externe,
# et représentation graphique
import numpy as np
import matplotlib.pyplot as plt

filename = 'test.txt'
AHA = np.loadtxt(filename)

plt.figure( filename )
plt.plot( AHA[], AHA[] , '.k')
plt.savefig( filename + '.png' )
```

1.4 Quelques bases rapides

Un langage informatique est **très** précis. Il n'y a aucune tolérance ou reconstruction: le lecteur est une machine qui n'a ni imagination, ni culture commune avec vous.

Quelques connaissances indispensables :

Rêgles d'écriture Tous les caractères sont autorisés. Certains mots sont reservés, comme `for`, `print`, `def`, `import`… Il est possible de commenter des morceaux du code, ou d'ajouter des précisions pour le lecteur humain, qui ne doivent pas être pris en compte par l'interpreteur. Dans ce cas on précède le contenu de `#`:

```
# commentaires non interprétés sur la ligne
```

On donne une valeur à une variable (affectation) avec le signe =. Les rêgles +-/* s'appliquent.

```
a = 2
a = a / 2 # on peut réalouer une valeur au même paramètre
a = (a - 2*a ) / a
print(a) # rien n'est␣affiché,␣tant,␣que␣cela␣n'est pas explicitement demandé
```

String (chaine de caractères)

Les strings sont des objets très utile pour les manipulations de fichier ou d'entete de fichier. Python dispose d'outils utile pour leur manipulation:

```
# création d'un␣string␣
c = "n'importe␣" + "quels␣" + "␣caractères\n" +  "\t␣espaces␣compris" +
"\n\t␣tabulation␣et␣RC␣interprétés"
print(c)

# Donner les caractères 11 et 15 à 20:
print(c[14:21]) # attention, les indices commencent à 0 !
```

Interactions avec les listes Il existe dans python différents types de tableau. Les listes permettent d'associer n'importe quel type d'entrée et de jouer avec:

```
# création d'une␣liste
# a**2 signifie a puissance 2
abc = [ a , a , a**2 ]

# liste composée de caract-res et de valeurs numériques
liste = [ 1 , 'z' , 56 , 1e5]

# doubler la liste
print( liste*2 )

# obtenir l'élément␣3␣et␣5␣de␣la␣liste␣(attention␣Python␣commence␣à␣0)
print( liste[2] )
print( liste[4] )

# obtenir la longueur de la liste
print( len(liste) )

# créer une suite d'entier␣avec␣range()
range(10)
list(range(10))
```

Librairies Accéder aux fonctions non implémentés dans le Python de base. Il en existe une très grand nombre dédiés à des applications aussi diverses que la communication avec des capteurs, l'analyse fine de données particulières ou la représentation graphique. Tant que la bibliothèque n'a pas été chargée, il n'est pas possible d'utiliser les commandes qui s'y trouve. On appelle la bibliothèque avec la commande `import` sous un nom spécifique comme `np` pour `numpy` (ce n'est pas obligatoire).

```
# faire un vecteur composé de dix "1" grace à la bibliothèque numpy
import numpy as np
a = np.ones(10)
print(a)
print(a*10)

# faire un plot avec la librairie matplotlib
import matplotlib.pyplot as plt
```

```
x = np.arange( 0 , 100 , 1 )
plt.plot( x, 5*x + 2 , '-or')
plt.show()
```

Vous pouvez accéder à une aide directement depuis le prompt en ajoutant un ? à la commande qui vous intéresse, par exemple `np.diff?` ou `range?`

Fonctions personnalisées Il est possible d'executer une partie du programme dans un sous programme. Cela permet d'alléger la lecture et de s'assurer que le code est rigoureusement le même malgré les différents appels:

```
# la fonction est définie avec def
# et porte le nom lineaire
def lineaire( a ):
    return a*10

# et celle ci loi de puissance
# on notera que le parametre beta doit être donné en entré
def puissance( a , beta ):
    return a**beta

# on l'appelle ailleurs n'importe où dans le code
a = 5
z = lineaire( puissance( a , 2 ) )
print(z)
```

Débugage Débugger consiste à comparer ce que le programme devrait faire et ce que le programme fait réellement. Cela se fait en deux étapes,

1. il faut que le programme puisse s'executer sans erreur
2. qu'il fasse ce que vous pensez qu'il doit faire.

Exemple avec un programme de calcul de perimetre d'un quadrilatère:

```
height = int(input("Height: "))
width = int(input("Width: "))
print("perimeter =", wdth + height + width + width )
```

Executer ce programme. Il ne tourne pas. Pour trouver pourquoi, il faut comprendre ce que fait chaque ligne en commençant par la ligne où se situe l'erreur. Vous avez deux indices, les sorties du programme, et le message d'erreur.

Une fois corrigée, le programme s'execute. Pour vérifier qu'il est correcte, appliquez le sur des cas où vous connaissez la réponse. Que concluez vous ? Pour le corriger il faut lire ligne à ligne le programme et comprendre d'où vient le problème. Il faut pouvoir répondre à la double question **Que fait cette ligne ? Est ce bien ce qu'elle est supposée faire ?**

2 Exporter / Importer un fichier

2.1 Avec `numpy`

La plupart du temps vous cherchez à accéder à des données pour en faire une représentation ou un traitement. Les méthodes d'accès vous seront donc fondamentales. Il existe un grand nombre de méthodes permettant d'accéder à des données. Par exemple dans la librairie `numpy` vous pouvez charger ou enregistrer des tableaux de chiffres à l'aide de `np.loadtxt` et `np.savetxt`. Mais attention ces fichiers ne peuvent contenir que des valeurs numériques.

```
# on définit le programme que l'on␣veut␣importer␣et␣son␣chemin␣d'accès
f_path = './souslepont/delariviereKwai' # ./ veut dire "à␣partir␣d'ici."
filename = 'fname.txt'

# chargement du fichier filename dans le repertoire du terminal
tab = np.loadtxt( f_path + '/' + filename )
print( tab )

# loadtxt dispose de nombreuses options utiles, ici on saut les 3 première
lignes pour lire les colonne 2 et 5
tab = np.loadtxt( filename , skiprows = 10, usecols= [1,4])
print( tab )

# pour un apperçu des options
np.loadtxt?

# inversement on peut enregistrer des données sous la même forme pour les
réutiliser
f_out = './nouveau/'
np.savetxt( f_out + 'nom_de_sortie.txt' , tab )
```

Exercices

1. Charger les fichier *LOAD_FILE_EXAMPLE-1* et *LOAD_FILE_EXAMPLE-2*, en leur donnant les noms `f1` et `f2`. Afficher leur contenu. Ré-enregistrer les avec les noms *output-1* et *output-2* dans le répertoire *nouveau* que vous aurez créé.

2. Créer un vecteur de N chiffres négatifs entiers que vous enregistrerez sous le nom *vecteur*. Le recharger et afficher la valeur absolue (fonction `abs`) de ce vecteur pour les indices entre $N - 2\,\varepsilon$ et $N - \varepsilon$. Vous préciserez quel valeurs de N et ε vous avez choisis.

3. Créer deux fichiers composés de deux matrices de nombres aléatoires à l'aide de la fonction `np.random.rand`, après avoir lu sa documentation. Recharger ces fichiers et sommer les valeurs terme à terme avec la fonction `np.add`

2.2 Importer des données produites sous Excel

Dans ce cas il existe deux options. Réenregistrer le fichier sous un format csv (Comma Separated Value) ne contenant que les données vous intéressant, ou charger le fichier excel à l'aide d'une librairie spécifique. Nous allons travailler à l'aide la librairie `panda`, mais il en existe d'autres.

2.2.1 Fichiers csv

Extrait de l'aide de la librairie `csv`:

> Le format CSV (Comma Separated Values) est le format d'importation et d'exportation le plus courant pour les feuilles de calcul et les bases de données. Le format CSV a été utilisé pendant de nombreuses années. L'absence d'une norme bien définie signifie que des différences subtiles existent souvent dans les données produites et consommées par différentes applications. Ces différences peuvent rendre fastidieux le traitement de fichiers CSV provenant de sources multiples.
>
> Le module csv implémente des classes pour lire et écrire des données tabulaires au format CSV. Il permet aux programmeurs de dire "écrivez ces données dans le format préféré d'Excel" ou "lisez les données de ce fichier qui a été généré par Excel", sans connaître les détails précis du format CSV utilisé par Excel. Les programmeurs peuvent également décrire les formats CSV compris par d'autres applications ou définir leurs propres formats CSV à usage spécifique.

Exercice Ouvrir le fichier *fichier_ excel_ simple.xls* avec un tableur. Que voyez vous ? Quelles sont les données que vous pouvez récupérer ? Nous allons ouvrir le fichier à l'aide de `np.loadtxt`

1. Enregistrer les données pertinentes des deux feuilles du fichier dans un format csv

2. En vous appuyant sur l'aide de `np.loadtxt`, pour notamment définir correctement le délimiteur de colonne, ouvrir le fichier.

L'inconvenient de cette methode est que nous perdons les informations liées aux colonnes. Il est tout à fait possible de charger également cette information, comme vous pouvez le voir dans l'aide de la fonction, mais la méthode n'est pas forcémenent la plus simple.

2.2.2 Fichiers Excel

Il existe un grand nombre de librairie qui permettent d'importer des fichiers Excel, de manière plus ou moins sophistiquée. Nous nous appuyons ici sur la fonction `pd.read_excel` de la librairie `panda`:

```
f_name =  # path to file + file name
s_name =  # sheet name or sheet number or list of sheet numbers and names

import pandas as pd
df = pd.read_excel(io=f_name, sheet_name=sheet)
print(df.head(5))  # print first 5 rows of the dataframe
```

Exercice Reprendre le fichier *fichier_ excel_ simple.xls* et ouvrir directement la première feuille depuis python, sans passer par un export en csv. Une fois la feuille de données obtenue, affichez la. Faites quelques opérations dessus, somme, produit, exposant, logarithme... Que remarquez vous sur les indices et les noms de colonne ?

Pour explorer le fichier vous pouvez utiliser la fontion `head`. À quoi sert elle ? Que pouvez vous obtenir ? Pour obtenir les entrées des colonnes vous pouvez utiliser la fonction `columns`. Comment écrire la commande pour obtenir le nom de la seconde colonne ?

Importez la seconde feuille dans Python. Quelle est la différence avec la première ?

Exercice d'application Pour commencer, rendez vous sur le site de l'entreprise du réseau de transport électrique RTE, puis eCO2mix et production d'electricité par filière. Télécharger un exemple de production de puissance sur une periode quelconque

1. Quel est le format des données que vous avez télechargé ?

2. Ouvrez le avec un tableur, que voyez vous, comment allez vous pouvoir exploiter ces données ?

3. Importez le sous python. Attention aux caractères accentuées qui peuvent poser problème. Dans ce cas les supprimer par une recherche automatique.

3 Faire une figure

3.1 Introduction

Une fois les données obtenues, et avant de les travailler, il est nécessaire de pouvoir en faire une visualisation simple et rapide. Nous allons nous appuyer sur la librairie `matplotlib` que nous appelons de la manière suivante :

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

Comme vous le voyez, ce n'est qu'une fraction de la librairie que nous chargeons. Toutes les fonctions seront précédées du préfixe `plt`. Comme d'habitude, n'hésitez pas à utiliser les ressources en ligne. Vous trouverez un grand nombre d'exemple sur les sorties graphiques disponibles sur le site de la librairie https://matplotlib.org/. Pour commencer, un exemple:

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

# production des données
X = np.arange(0,100,1)
# creation de la figure
figname = 'ma_figure'
plt.figure(figname)
# plot
plt.plot( X , X**2 , 'r--o')
# affichage
plt.show()
# sauvegarde
plt.savefig(figname)
```

Comme d'habitude, copier le fichier, l'ouvrir dans un éditeur de texte, l'excuter, et le manipuler. Que voyez vous ? À quoi servent les différentes lignes ?

1. Ajouter des axes à la figure avec les fonctions `plt.xlabel` et `plt.ylabel`

2. Représenter la fonction $\sqrt{X}$ avec une ligne noire et $X^2 + 2\,X - 1$ avec une ligne discontinue bleue

3. Modifier l'axe des abscisse, avec `plt.xlim` et des ordonnées avec `plt.ylim`

3.2 Plus sophistiqué

3.2.1 Plot multiples

Avec la fonction `plt.subplot` il est possible de représenter plusieurs graphes dans la même fenêtre. Par exemple avec `plt.subplot(2,3,4)` on va créer une fenetre comprenant 2×3 graphiques, dont le graphique suivant sera en position 4.

Exercice Faire un graphe multiples de 3×2 graphiques et y afficher 6 graphiques différents.

3.2.2 Histogrammes

Une distribution doit être représentée sous forme de d'histogramme, normalisé ou non.

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

# creation de la figure
figname = 'ma_figure'
plt.figure(figname)
# plot
plt.hist( np.random.randn(100) )
# affichage
plt.show()
```

Exercice Représenter sur des figures séparées la quantité y (log-normale), son histogramme, sa pdf (option weight) et sa cumulative:

```
y = 4 + np.random.normal(0, 1.5, 200)
```

3.2.3 Matrices

Pour représenter une image ou un tableau, on utilise une représentation matricielle, avec par exemple `plt.pcolormesh`. Exemple:

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

# données
X = np.random.rand(100,100)
# creation de la figure
figname = 'ma_figure'
plt.figure(figname)
# plot
plt.pcolormesh( X )
# affichage
plt.show()
```

Exercice

1. Représenter la sortie de la fonction suivante:

```
sinc2d = np.zeros((50, 50))
for x, x1 in enumerate(np.linspace(-10, 10, 50)):
     for y, x2 in enumerate(np.linspace(-10, 10, 50)):
          sinc2d[x,y] = np.sin(x1) * np.sin(x2) / (x1*x2)

x1 = np.linspace(-10, 10, 50)
x2 = np.linspace(-10, 10, 50)
sinc2d = np.outer(np.sin(x1), np.sin(x2)) / np.outer(x1, x2)
```

2. Pour charger une image, on peut utiliser la fonction `mpimg.imread` de la librairie `import matplotlib.image as mpimg`. Pour la représenter `plt.imshow`. Trouver une image sur internet. L'enregistrer puis l'ouvrir sous python. Enfin la représenter et l'enregistrer.

3.2.4 Affichage / Enregistrement

Représenter $v = g\,t$ en fonction de t, sur la gamme, $t = 0$ à 100 secondes avec $g = 10$ m/s^2. Quelle est la dimension de v ? Ajouter les noms des axes en incluant leur dimension.

1. Ajouter un titre à la figure précédente
2. Enregistrer
    a. avec une résolution de 300 dpi.
    b. dans le répertoire */temp/ou/?*
    c. en format *jpeg*, puis *pdf*

4 Les fonctions

Quand une tache est répétée dans un programme et que seules les valeurs numériques changent, il est alors intéressant de créer sa propre fonction, que l'on appelle comme une fonction issue d'une librairie. Exemple de création de la fonction `func()`

```
# ne pas oublier les ":"
# attention à l'indentation
def func(x):
    a = x**2
    return a

var = func(2)
print( var )
var = func(-10)
print( var )
```

1. S'il n'y a rien à retourner, la commande `return` peut être omise.

2. Les paramètres d'entrée ne sont pas obligatoires non plus. On peut donner une valeur par défaut à un paramètre, par exemple `def func(x=3)`.

3. L'ordre des paramètres n'est pas important SI les paramètres sont étiquetés. Par exemple pour la fonction à deux paramètres `func(x,y)`, les deux écritures `func(x=3,y=2)` et `func(y=2,x=3)` sont équivalentes, mais `func(2,3)` et `func(3,2)` ne le sont pas.

4. Enfin, on peut regrouper dans une classe une série de fonction `class lib_perso`, ce qui crée une librairie, que l'on pourra charger de la même manière que les autres avec la commande `import lib_perso`

4.1 Applications

1. Faire une fonction sans paramètre, qui renvoie (imprime sur le prompt) *bonjour* et 3 nombres aléatoires sur deux lignes différentes.

2. Faire une fonction de conversion d'une surface de km^2 à hectare. Cette fonction ne doit rien afficher.

3. Faire une fonction qui calcule le volume d'une sphère dans la variable `tic` sans l'afficher, avec comme paramètre le rayon de la sphère.

4. Faire une fonction qui convertit une puissance en Watt et un temps en seconde en une énergie en kWh. On explicitera les unités de la puissance et du temps choisis. Cette fonction doit afficher les paramètres entrés (puissance et temps) et l'énergie calculée avec leurs unités.

5. Faire une fonction qui charge un fichier texte. Les paramètres sont le nom du fichier (par défaut *tsoin.tsoin*) et le chemin pour y accéder (par défaut *./taga/da/*).

4.2 Plus compliqué

Une manière de structurer un script python est d'écrire une succession de fonctions, qui sont appelées dans le corps du script par une commande spécifique:

```
# les librairies sont souvent appelées en entete
import numpy as np

def func1(a):
    # à quoi sert cette fonction ?
    return a**2

def func2(a):
    # à quoi sert cette fonction ?
```

```
    a = np.arange(a)
    return a

if __name__ == "__main__":
    # corps du programme
    a = 18
    b = func1(a)
    c = func2(a)
    print(a,b,c)
```

Le corps du programme est écrit dans `if __name__ == "__main__":`. À l'interieur les fonctions sont appelées (ou pas) successivement. Le reflexe à la lecture d'un code écrit en python est de chercher le corps du programme, de le lire ligne à ligne pour ensuite trouver l'usage des fonctions.

Exercice À faire

1. Ouvrir le programme xxx. Commenter les différentes fonctions. Que fait il ?
2. Créer un programme consitué de fonctions et d'un corps de programme qui fasse
   a.
   b.

5 Les boucles et vecteurs

Les boucles servent à répéter une opération suivant un incrément, ou itération. Les boucles s'ouvrent en faisant apparaitre les commandes appropriées et sont marquées par une indentation. Il exite trois types principaux de boucle. La boucle `for` se reproduit pour un nombre d'itération fixe:

```
# boucle for
for inc range(10):
    inc*10
```

La boucle `if` se reproduit si une condition est satisfaite:

```
# boucle if
a = 6
if a<5 :
    print(a*10)
```

La boucle `while` se reproduit tant qu'une condition n'est pas atteinte:

```
# boucle while
i = 1
while i<100:
    i = i+1
```

5.1 Boucle `for`

Exécute la même série d'instructions en changeant un incrément, qui peut être un nombre ou un string. Exemple:

```
c = np.array([0,1,2,3,4,5,6,7,9])
for i in c:
    print('l incrément vaut ' + str(i))
```

La boucle se lit littéralement comme *pour i variant de 0 à 9, afficher « l'incrément vaut i »* . On remarque une structure précise qui doit être respectée pour tous les types de boucles:

1. La boucle s'ouvre avec la commande `for`, la gamme de l'incrément `i in c`, et `:`

2. Les instructions sont à une tabulation de la marge gauche. C'est l'indentation.

3. La boucle ne se ferme pas avec une commande, c'est la fin de l'indentationt qui la marque.

Application

1. Il est nécessaire de définir une gamme d'incrément. Pour cela on peut utiliser la fonction `range`. En vous appuyant sur la documentation, expliciter la manière d'obtenir une variation d'incrément de 0 à 10 par pas de 0.1. Réécrire la boucle `for` de l'exemple avec cet incrément.

2. Faire une boucle `for` qui calcule la somme cumulée de ses incréments.

3. On peut imbriquer deux boucles :

   ```
   for i in range(10):
       for j in range(5)
          print('i=' + str(i) + '␣et␣j␣=␣' str(j))
   ```

   corriger les deux boucles imbriquées précédentes pour qu'elles puissent s'executer.

4. Il est possible de donner un incrément qui ne soit pas un nombre. Faire une boucle dont la sortie est *la valeur de l'increment est a*, puis *la valeur de l'increment est b* et ainsi de suite.

5. On peut utiliser la valeur de l'incrément comme indice. Que donne

   ```
   c = ['a','b','c']
   for i in range(len(c)):
       print('l␣incrément␣vaut␣' + str(i) + '␣et␣c[i]␣vaut␣' + c[i])
   ```

   dans ce cas, attention `i` doit être un entier. On notera que `c` n'est pas une liste de nombre mais que la fonction `range(len(c))` permet d'en recréer une. Expliquer comment.

5.2 Boucle `if`

Execute une série d'instruction si une condition est respectée. Exemple:

```
a = 0
# si a est nul
# attention au "=="
if a==0:
    print('a␣=␣0')
# si a n'est␣pas␣nul
elif a!=0:
    print(a*10)
# sinon
else:
    print('␣erreur␣')
```

La boucle se lit littéralement comme *si* $a=0$, *afficher a=0, si* $a\neq 0$ *mutiplier par 10, sinon afficher « erreur »*. On remarque que la structure est la même que pour les boucles `for`. On a en plus

1. La condition d'égalité qui s'écrit `==`.

2. La condition de différence qui s'écrit `!=`

3. La condition superieur ou inferieur qui s'écrivent `>` et `<`

et les commandes

1. `elif`, qui permet de définir une autre condition

2. `else`, qui permet de remplir tous les autres cas, souvent employé pour renvoyer une erreur

Application

1. Faire une boucle `if` qui calcule une surface si les grandeurs proposées sont positives.

2. Faire une boucle `if` qui enregistre un fichier texte si la taille du vecteur proposé dépasse un seuil, sinon affiche le vecteur à l'écran et demande de recommencer.

3. Faire une boucle `for` qui ajoute un caractère en bout de string à chaque itération, et y ajouter la boucle `if` créée précédemment

5.3 Boucle `while`

Execute une série d'instruction qui sera réalisée tant qu'une condition est respectée. Exemple:

```
a = 0
# si a est nul
while a<0.8:
    print(a)
    a = np.random.rand(1)
```

La boucle se lit littéralement comme *tant que* $a < 0.8$*, afficher a, et renouveler la valeur de a avec la fonction* `random.rand`. On remarque que la structure est la même que pour les boucles `for`. En executant cette boucle on ne connait pas à priori le nombre d'itération à faire, sinon on aurait uilisé une boucle `for`. Le risque d'une telle boucle est qu'elle ne se cloture (converge) jamais, si la condition demandée n'est jamais remplie. Il y a souvent une boucle `if` ajoutée pour arréter la boucle en cas d'absence de convergence.

Application

1. Comprendre pourquoi la boucle proposée converge (s'arrête). La réécrire pour qu'elle ne converge plus et vérifier que c'est bien le cas. Ctrl+C devrait forcer l'arret de la boucle.

2. Écrire une boucle `while` qui calcule le logarithme de l'incrément et s'arrête lorsque sa valeure est positive.

5.4 Vecteurs

Les boucles sont lentes à executer, dans le sens où les instructions sont executées l'une dernière l'autre. C'est l'itération. Une méthode pour diminuer le temps de calcul est de paralléliser la boucle. Cela n'est possible que si les itérations sont indépendantes entre elles. Une autre méthode consiste, lorsque cela est possible de travailler avec des vecteurs. C'est à dire d'appliquer des opérations globalement sur des vecteurs. Exemple:

```
import time

# application de l'opération␣par␣boucle␣for
```

```
tic = time.time()
a = np.arange(1,1000,1)
for i in range(len(a)):
    a[i] = np.sqrt(a[i])
toc = time.time()
print(toc-tic)

# application directe de l'opération␣sur␣le␣vecteur␣vec
vec = np.arange(1,1000,1)
vec = np.sqrt(a)
```

Justifier que les instructions sont comparables, qu'elles mènent au même résultat, et comparer les temps d'execution des deux opérations en expliquant la methode utilisée.

Créer un vecteur composé des valeurs comprises entre -10 et 10 avec un incrément de 0.01, puis calculer successivement, le carré, le cube et la racine carrée. Faire une représentation graphique.

6 Un peu d'analyse statistique

6.1 Organiser ses données

Pour organiser ses données, il existe quelques fonctions utiles. Beaucoup proviennent de la bibliothèque `numpy`.

1. `np.sort`, Return a sorted copy of an array.
2. `abs`, Return the absolute value of the argument.
3. `np.max`, Return the maximum of an array or maximum along an axis.
4. `np.min`, Return the minimum of an array or minimum along an axis.
5. `np.argmax`, Returns the indices of the maximum values along an axis
6. `np.isnan`, Test element-wise for NaN and return result as a boolean array.

6.2 Statistique

1. `np.mean`, Compute the arithmetic mean along the specified axis.
2. `np.std`, Compute the standard deviation along the specified axis.