L2-S3 : UE Méthodes numériques

SEANCE 1

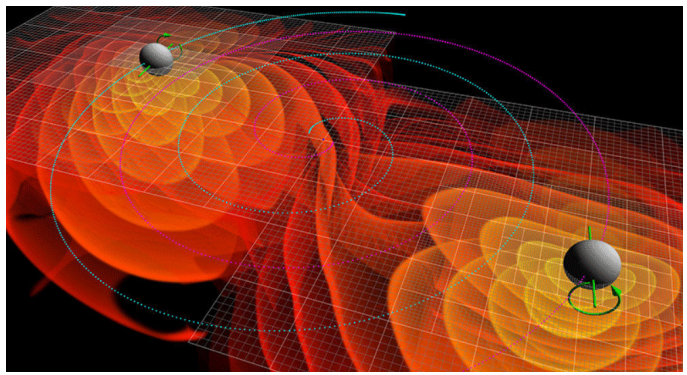
Python et Numpy

5 septembre 2024

Méthodes numériques L2

Cours avancé suivant celui de 1ère année (y revenir si besoin est)

- 3 séances de rappel (numpy, représentation graphique, Intégration, Ajustement de données) ;
- puis fonctions 2D, intégrales avancées, traitement d'image, éq. différentielles, éq. linéaires
- pour les LDD-MP : classes et objets, projet collaboratif (+2 séances)



Objectifs : vous fournir les outils numériques essentiels à l'ingénieur, au mathématicien, au physicien et au chimiste !

Modalités de contrôle des connaissances, L2

Pour tous les étudiants sauf les LDD-MP, la note finale de l'UE est constituée de :

- 20% contrôle continu (DM à rendre à la fin de la séance 5)
- 30% partiel (après la séance 6)
- 50% examen

Pour les étudiants de LDD-MP, la note finale est composée de :

- 30% contrôle continu (20% DM à rendre à la fin de la séance 5 + 10% mini projet à discuter dans la séance 12)
- 30% partiel (après la séance 6)
- 40% examen

Partiel en salle informatique de 2 heures. Examen en salle informatique de 3 heures.

Accès aux ordinateurs

- Le serveur d'authentification de l'université restera hors service encore pendant plusieurs semaines.
- L'accès aux ordinateurs de la salle informatique se fait avec un identifiant unique pour tout le monde. Les comptes ne sont donc pas privés.
- À la fin de chaque séance, vous devrez transférer votre travail (par mail ou via un service de cloud) et effacer tous les fichiers et dossiers que vous avez créés.
- N'enregistrez pas vos mots de passe dans le navigateur, car ils seront accessibles aux utilisateurs suivants. Il est également recommandé d'ouvrir votre navigateur en mode privé.

Où trouver les documents de travail

- Le travail avec le serveur JupyterHub et l'environnement Methnum que vous avez utilisé en L1 est pour l'instant impossible.
- Les contenus du cours et les notebooks pour les TDs se trouvent sur cette page ecampus.
- Les documents se trouvent également sur le serveur `https://methnum.pages.in2p3.fr/l2-profs/` qui permet également le travail à distance sans cependant la flexibilité que le serveur jupyterhub offrait l'année dernière.
- Vous n'avez pas accès au travail personnel que vous avez effectué en L1. Tous les cours de L1, avec les solutions des exercices, sont mis à votre disposition sur cette page `https://methnum.pages.in2p3.fr/l1-profs/`.
- Vous aurez la possibilité de travailler soit en utilisant un serveur Jupyter local, soit directement sur le web via une page JupyterLite.

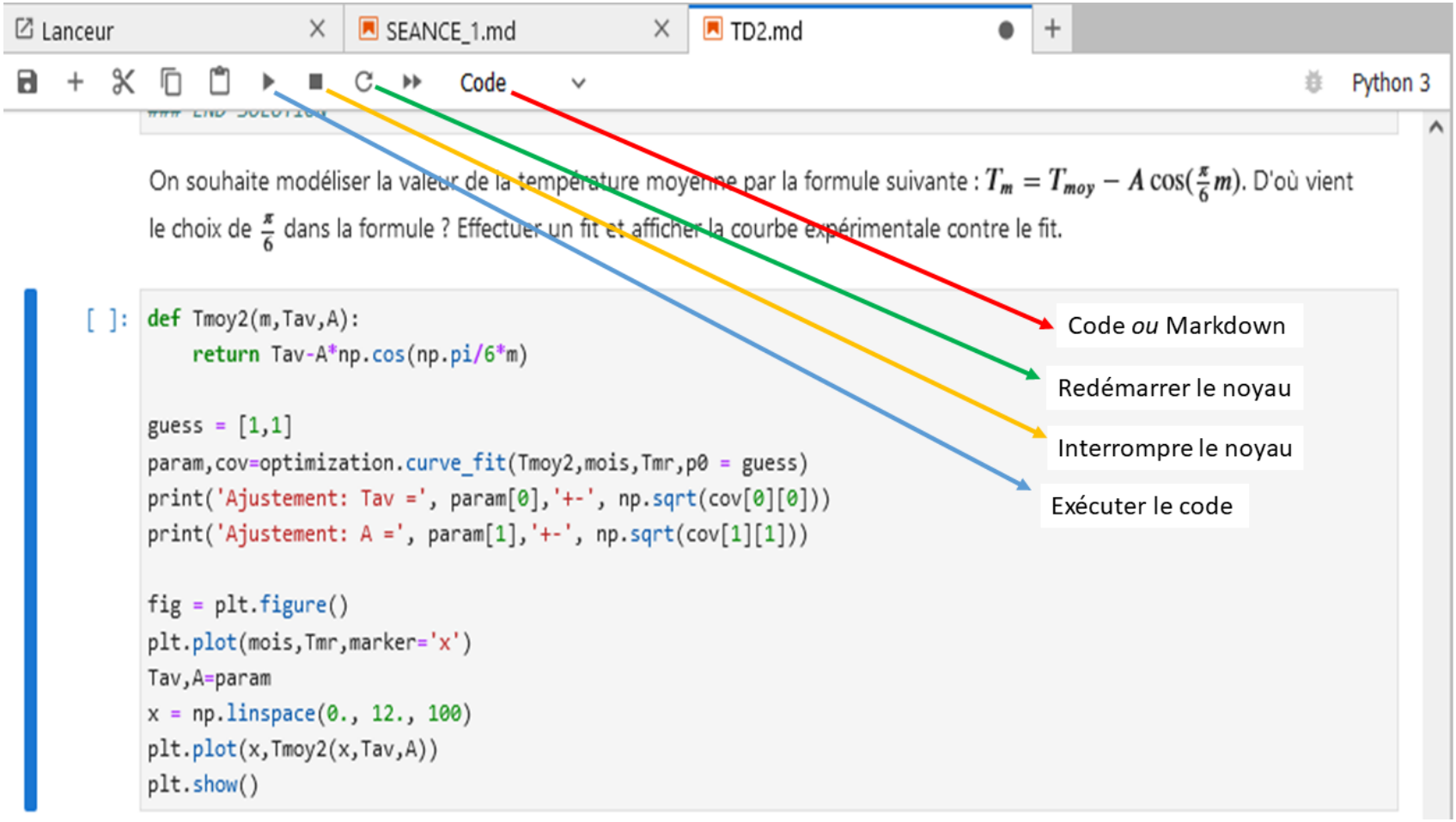
Pour survivre dans un terminal : B.A. BA du shell

- `pwd` : *print working directory*, affiche l'adresse du répertoire courant.
- `ls` : *list segments*, liste les fichiers et dossiers dans le répertoire courant. `ls -lhrt` pour plus d'information.
- `cd` : *change directory*, change de répertoire courant. `cd ..` pour remonter l'arborescence, `cd sous_repertoire` pour descendre.
- `mkdir repertoire_cree` : *make directory*, crée un répertoire dans le répertoire courant.
- `cp fichier1 fichier2` : *copy*, copie le contenu de `fichier1` dans un fichier nommé `fichier2`. `cp fichier1 dossier2/` : copie/colle `fichier1` dans le répertoire `dossier2`. `cp *.txt dossier2/` : copie/colle tous les fichiers d'extension `.txt` vers `dossier2`.
- `mv fichier1 dossier2/` : *move*, coupe/colle `fichier1` dans le répertoire `dossier2`.
- `rm fichier1` : *remove*, supprime définitivement un fichier. **!!** `rm *` efface définivement tous les fichiers dans le répertoire courant **!!**
- `man nom_de_commande` : accède au manuel d'aide linux pour la commande. Pour sortir, taper q (*quit*).

Comment exécuter un code source ?

- **Mode interpréteur :** dans un terminal, taper `python` (classique) ou `ipython` (interpréteur plus évolué) pour accéder à un interpréteur Python. Taper les instructions Python à exécuter.
- **Mode programmation** :
  - écrire le code à exécuter dans un fichier d'extension `.py`. L'exécuter dans un terminal avec la commande `python nomdufichier.py` ou `ipython nomdufichier.py`.
  - stocker le code (et son résultat) dans un **Jupyter Notebook** (fichiers d'extension `.md` et `.ipynb`). Interfaçage via un browser (e.g. Firefox), environnement convivial, facilitant les interactions $\rightarrow$ **mode préféré dans ce cours**.

Jupyter : système de “notebooks”



Jupyter : système de “notebooks”

Pour lancer un serveur jupyter :

- sur un ordinateur de l'université, tapez dans un terminal

```
methnum jupyter notebook
```

- Créez un notebook : en haut à droite menu “New” -> “Python 3”
- Divers types de cellules peuvent être créées (menu déroulant en haut à droite) :
  1. Markdown : cellules de texte pour décrire ce que l'on fait par exemple
  2. Code : cellules où on rentre du code python, exécutez avec **Shift+Entrée**

Astuces : pour interrompre une exécution Menu **Kernel/Interrupt**, pour réinitialiser la mémoire **Kernel/Restart**

Travailler avec des Jupyter Notebooks en locale

- Téléchargez de ecampus le fichier `.ipynb` et les possibles fichiers associées correspondant au TD du jour (voir les instructions sur la page de la séance dans ecampus).
- Exécutez la commande `jupyter notebook` dans un terminal pour démarrer un serveur Jupyter. Ne fermez pas le terminal utilisé, sinon vous arrêterez le serveur.
- À la fin de chaque séance, pensez à sauvegarder et transférer votre travail.
- N'oubliez pas de supprimer les fichiers et dossiers locaux que vous avez générés.

Travailler avec JupyterLite

- Sur la page `https://methnum.pages.in2p3.fr/l2-profs/` vous trouvez un lien vers une page JupyterLite. Il s'agit d'une distribution Python qui est téléchargée en local et exécutée directement dans le navigateur. Le premier téléchargement peut prendre un peu de temps, soyez patients si rien ne s'affiche immédiatement.
- Vous retrouverez un environnement Jupyter dans lequel apparaissent le fichier de cours ainsi que les fichiers .ipynb des TD sur lesquels vous pouvez directement travailler.

Travailler avec JupyterLite

- Toute modification est conservée dans le cache du navigateur, mais les fichiers modifiés ne sont effectivement sauvegardés nulle part.
- Le système de cache pourrait générer des incohérences entre les versions des fichiers modifiés précédemment. Pour vous assurer de travailler avec la version originale des fichiers, **il est fortement recommandé d'ouvrir votre navigateur en mode privé**.
- Pendant la séance, vous devrez régulièrement penser à sauvegarder les fichiers sur votre ordinateur local. À la fin de la séance, vous devrez effectuer une dernière sauvegarde et vous transmettre les fichiers.

Types des objets en Python

- Le travail avec le serveur JupyterHub et l'environnement Methnum que vous avez utilisé en L1 est pour l'instant impossible.
- Les contenus du cours et les notebooks pour les TDs se trouvent sur une page publique de ecampus : `https://ecampus.paris-saclay.fr/course/view.php?id=155363`
- Téléchargez le fichier .ipynb et les possibles fichiers associées correspondant au TD du jour (voir les instructions sur la page de la séance dans ecampus).
- Exécutez la commande "jupyter notebook" dans un terminal pour démarrer un serveur Jupyter. Ne fermez pas le terminal utilisé, sinon vous arrêterez le serveur.
- À la fin de chaque séance, pensez à sauvegarder et transférer votre travail.

Types des objets en Python

En python on trouve différents types de base pour les objets. Les types numériques qu'on utilisera dans ce cours seront :

- **int** : nombres entiers
- **float** : nombres réels
- **complex** : nombres complexes (de la forme *a+bj ou a+bJ)*
- **bool** *: pour un booléen qui peut prendre une valeur 'True' ou 'False'*

On utilisera également les types suivants :

- **str** : chaîne de caractère
- **list** : pour une liste

et d'autres...

La commande **type()** permet de connaître le type d'une variable.

Print

La commande **print()** permet d'afficher le contenu d'une ou plusieurs variables :

In [1]:
```
a=2
b=3
c=a+b
d='+'
e='='
print(a,d,b,e,a+b)
```

Out[1]:
```
2 + 3 = 5
```

In [2]:
```
c = 2.7 + 3.8j
print("le type de ", c," est ", type(c))
```

Out[2]:
```
le type de  (2.7+3.8j)  est  <class 'complex'>
```

In [3]:
```
f = a+b==5
g = a+b<1
print("f:", f, "  g:", g)
```

Out[3]:
```
f: True   g: False
```

Types numériques et opérateurs

- un nombre entier s'écrit sans point, sinon c'est un **float**.
- le type d'une variable numérique peut être changé : **int()**, **float()**, **complex()** (ou même `bool()` ou `str()`)
- opérateurs +, −, *, /, % (modulo)
- opérateur de comparaison : ==, !=, >, <, >=, <= (seul les deux premiers sont définis pour les complexes)

In [4]:
```
1 a=10
2 b=10.
3 c=10.+0.j
4 a == b
```

Out[4]:
```
True
```

In [5]:
```
1 type(a) == type(b)
```

Out[5]:
```
False
```

In [6]:
```
1 a < c
```

Out[6]:
```
TypeError: no ordering relation is defined for
```

Booléens et opérateurs logiques

Il y a trois opérateurs logiques en Python : **and**, **or** et **not** (pour exprimer la négation d'un booléen ou simplement d'une expression ou proposition). Par exemple

In [7]:
```
1 5 > 4 and 8 == 2 * 4
```

Out[7]:
```
True
```

In [8]:
```
1 True and False
```

Out[8]:
```
False
```

In [9]:
```
1 False or True
```

Out[9]:
```
True
```

In [10]:
```
1 not False
```

Out[10]:
```
True
```

Booléens et opérateurs logiques

Note 1 : toute valeur Python a un équivalent de "vérité" booléen : une valeur **nulle** (numérique) ou **vide** (chaîne de caractères, liste) est **fausse**. Les autres valeurs sont vraies, ainsi :

```
In [11]:  1 '' and 'pommes'

Out[11]:    ''

In [12]:  1 ('a', 'b', 'c') or [5, 6]

Out[12]:    ('a', 'b', 'c')
```

Note 2 : Python court-circuite l'évaluation quand il le peut : `True or ...` est forcément vrai (pas besoin de connaître ...)

Listes : définition et création

Une liste est un ensemble **ordonné** d'éléments placés les uns à la suite des autres, qui peuvent être de **types différents.** Elle est délimitée par des crochets $[\cdots, \cdots, \cdots]$ et ses éléments sont séparés par des virgules :

Exemple

In [13]:
```
print([1,10,24,29])
print([2.3,10.5,26.8,50.0])
print(["lundi","mardi","mercredi","jeudi"])
print([2,3.4,"mardi",3+2j,"3"])
```

Out[13]:
```
[1, 10, 24, 29]
[2.3, 10.5, 26.8, 50.0]
['lundi', 'mardi', 'mercredi', 'jeudi']
[2, 3.4, 'mardi', (3+2j), '3']
```

Comme vous le constatez dans le dernier exemple, les éléments contenus dans cette liste sont de différents type. Nous avons des entiers, des floats, des nombres complexes et des chaines de caractères.

Listes : définition et création

On peut affecter une liste à une variable, qui sera donc de type **list** :

In [14]:
```
1 a=[5,4,'ok',1+9j]
2 print(a)
```

Out[14]:
```
[5, 4, 'ok', (1+9j)]
```

In [15]:
```
1 print(type(a))
```

Out[15]:
```
<class 'list'>
```

La notation a=[] correspond à créer une liste vide qui pourra être remplie par la suite.

Listes : définition et création

On extrait un groupe de valeurs grâce à la notation
`[indice de début:indice de fin:pas de l'indice]` :

In [16]:
```
1 a=[1, 10, 24, 29.100, 12, 35]
2 print(a[0:5:2])
```

Out[16]:
```
[1, 24, 12]
```

In [17]:
```
1 print(a[0:3])
```

Out[17]:
```
[1, 10, 24]
```

In [18]:
```
1 print(a[0:-2])
```

Out[18]:
```
[1, 10, 24, 29.1]
```

In [19]:
```
1 print(a[0:-2:3])
```

Out[19]:
```
[1, 29.1]
```

L'élément avec l'indice du début est inclus, celui avec l'indice de la

Opérations : +, *

L'opération + agit sur des listes en les **concaténant** :

In [20]:
```
1 a=[3, 4, 6]
2 b=[2,3]
3 print(a+b)
```

Out[20]:
```
[3, 4, 6, 2, 3]
```

L'opération *n (avec n entier) permet d'allonger une liste en répétant n fois ses éléments.

In [21]:
```
1 a=[1,2,3,4]
2 a*3
```

Out[21]:
```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Remarque : *** ne multiplie pas chaque nombre de la liste**, une **liste** n'est pas un **vecteur**.

Chaînes de caractères

Objet de type **str** indiqué entre les symboles " " ou ' ', permettant de définit une chaîne de caractères.

Caractères spéciaux : `\n` pour le retour à la ligne, `\t` pour la tabulation

Opérateurs supportés : == et != ; + concaténation ; * auto-concaténation :

```
In [22]:  1 print('hello'+' '+'world')   # concatenation

Out[22]:    'hello world'
```

```
In [23]:  1 print('hello ' * 2)   # auto-concatenation

Out[23]:    'hello hello'
```

Chaînes de caractères

Les méthodes de slicing sont aussi utilisables pour toute chaîne de caractères :

In [24]:
```
1 a="Lorem ipsum dolor sit amet"
2 a[3]
```

Out[24]:
```
'e'
```

In [25]:
```
1 a[0:3:2]
```

Out[25]:
```
'Lr'
```

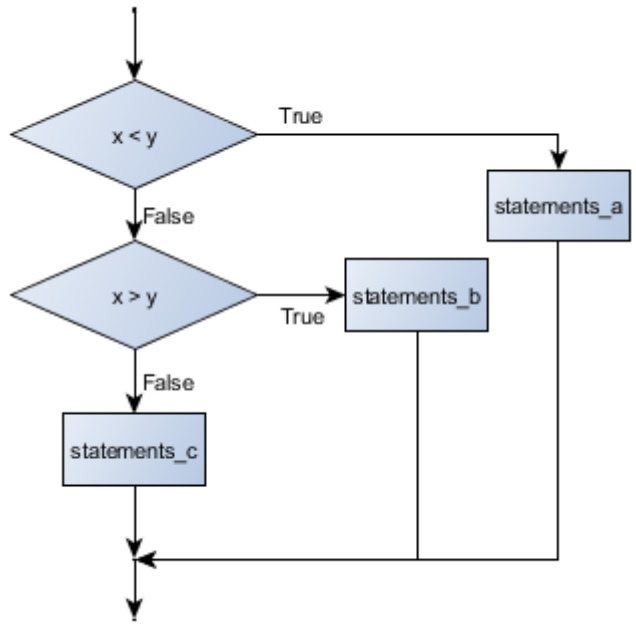
In [26]:
```
1 a[0:-2:3]
```

Out[26]:
```
'Leiudos '
```

Algorithmique - if / elif / else

Les instructions `if`, `elif` (contraction de else if), `else` conditionne une action à des tests (valeurs booléennes) :

```
1 choice = 'c'
2 if choice == 'a':
3     print("Vous avez choisi 'a'."
      )
4 elif choice == 'b':
5     print("Vous avez choisi 'b'."
      )
6 elif choice == 'c':
7     print("Vous avez choisi 'c'."
      )
8 else: # permet d etre certain d
      avoir couvert toutes les
      possibilites
9     print("Mauvais choix.")
```

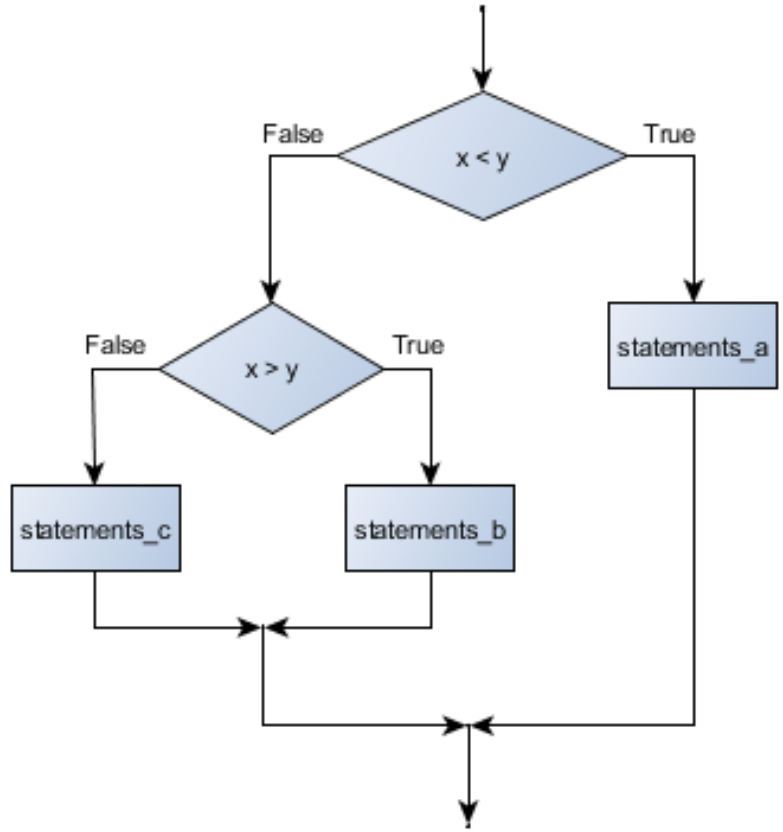


- : permet de distinguer la condition des actions à réaliser
- l'indentation détermine ce qui dépend de la condition.
- les conditions sont évaluées dans l'ordre, attention à la logique !

Algorithmique - if / elif / else

On peut aussi imbriquer des conditions :

```
x = 7
y = 2
if x < y:
    print('STATEMENTS_A')
else:
    if x > y:
        print('STATEMENTS_B'
    )
    else:
        print('STATEMENTS_C'
    )
```



Évitez les imbrications quand c'est possible !

```
x = 5
if x > 0:
    if x < 10:
        print("0 < x < 10")
```

```
x = 5
if 0 < x < 10:
    print("0 < x < 10")
```

Algorithmique - Boucle `while`

while condition : exécute les instructions contenues dans la boucle tant que **condition** est **False**, c'est à dire un nombre de fois **non déterminé a priori**

```
# On jette trois des jusqu'a avoir obtenu un triple 6
import numpy as np

triple_six = False
compteur = 0
while triple_six is False:
  de1, de2, de3 = np.random.randint(1, 7, size=3)
  compteur += 1
  if de1==6 and de2==6 and de3==6:
    triple_six = True
print("Triple six obtenu en", compteur, "coups.")
```

Attention aux **conditions d'arrêt** ! Risque de boucle infinie si **condition** n'est jamais **True** ! ! ! On pourra utiliser un incrément pour s'assurer de la terminaison de la boucle :

```
while triple_six is False and compteur < 10:
  ...
```

Algorithmique - Boucle `for`

for exécute un **nombre de fois prédeterminé** les instructions contenues dans la boucle :

```
for i in range(15) :
  print(i)
```

```
notes = [12, 19, 9, 14, 15, 13, 15]
for note in notes:
  print(note)
```

On ne modifiera **jamais** l'incrément `i` d'une boucle `for` dans le corps de la boucle !

On peut aussi énumérer les éléments d'un tableau et ainsi avoir accès au compteur :

```
notes = [12, 19, 9, 14, 15, 13, 15]
for i, note in enumerate(notes):
  print(i, note)
```

On évitera `break` et `continue` (sortir immédiatement de la boucle ou sauter l'itération en cours).

Algorithmique - Boucle `for`

for peut aussi être utile pour créer des listes, à l'aide d'une syntaxe particulière à Python :

In [27]:
```
numbers = list(range(5))
print (numbers)
print([x**2 for x in numbers])
```

Out[27]:
```
[0, 1, 2, 3, 4]
[0, 1, 4, 9, 16]
```

In [28]:
```
print([x**2 for x in numbers if x**2 > 8])
```

Out[28]:
```
[9, 16]
```

On évitera les boucles **à chaque fois qu'on peut** avec **numpy** :

In [29]:
```
import numpy as np
numbers = np.arange(5)
a = numbers**2
print(a[a>8])
```

Out[29]:
```
array([9, 16])
```

Numpy : vecteurs

numpy array : type non-natif de python pour les listes de valeurs numériques (complexes, float, entiers, booléens). Vaste bibliothèque de fonctions mathématiques, fonctions basiques d'algèbre linéaire, transformées de Fourier.

On commence par importer le module `numpy` en début de programme. Un numpy array peut être créé à partir d'une liste.

In [30]:
```
1 import numpy as np
2 a = np.array([0,3,2])
```

Opérations aisément réalisées sur l'ensemble du **numpy array** :

- `np.exp(a)`, `np.sin(a)`, `np.cos(a)` : numpy arrays contenant l'exponentiel, le sinus et le cosinus des termes contenus dans a
- `np.sign(a)` : numpy array contenant le signe (-1/+1) des termes de a
- `np.pi`, `np.e`, `np.inf`, `np.nan`
- ...

Numpy : vecteurs

Comparons trois calculs de $y = 3x - 1$ pour un million de valeurs de x :

```
1  import numpy as np
2  import time
3
4  x = np.linspace(0, 1, 1000000)
5
6  #Methode 1: LENT
7  tic = time.time()
8  y = []
9  for i in range(x.size):
10     y.append(3*x[i] - 1)
11 print("Methode 1: temps de calcul", time.time()-tic)
12 # renvoie 0.6396753787994385
13 #Methode 2 : UN PEU MIEUX
14 tic = time.time()
15 y = [3*a - 1 for a in x]
16 print("Methode 2: temps de calcul", time.time()-tic)
17 # renvoie 0.5166025161743164
18 #Methode 3 : IDEAL
19 tic = time.time()
20 y = 3*x -1
21 print("Methode 3: temps de calcul", time.time()-tic)
22 # renvoie 0.015623909759521
```

Numpy : vecteurs

Array à plusieurs indices, manipulations Peut être omis

In [31]:

```
X2D = np.array([[1,2,3, 4],[5,6, 7, 8],[9, 10, 11, 12]])
print ("X2D: tableau 2D (3x4)\n",X2D)
print("X2D[0][3]:", X2D[0][3]," X2D[2][0]:", X2D[2][0]) # premier indice: lignes; second : colonnes
Y2D = np.array ([[11,12,13,14],[15,16,17,18],[19,20,21,22]])

print ("Y2D: tableau 2D (3x4)\n",Y2D)
XY3D = np.array([X2D, Y2D])
print("XY3D: tableau 3D (2x3x4)\n", XY3D)
print("XY3D[0][1][3]:",XY3D[0][1][3],"  XY3D[1][1][3]:",XY3D[1][1][3])
print("XY3D[0][2][1]:",XY3D[0][2][1],"  XY3D[1][2][1]:",XY3D[1][2][1])
XY2D = np.vstack((X2D, Y2D))  # attention: doubles parentheses
print("XY2D: tableau 2D (6x4)\n", XY2D) # concatene les deux listes de listes: 2x3 lignes et 4 colonnes
```

Numpy : vecteurs

Out[31]:

```
X2D: tableau 2D (3x4)
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
X2D[0][3]: 4  X2D[2][0]: 9
Y2D: tableau 2D (3x4)
 [[11 12 13 14]
 [15 16 17 18]
 [19 20 21 22]]
XY3D: tableau 3D (2x3x4)
 [[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]
 [[11 12 13 14]
  [15 16 17 18]
  [19 20 21 22]]]
XY3D[0][1][3]: 8   XY3D[1][1][3]: 18
XY3D[0][2][1]: 10   XY3D[1][2][1]: 20
XY2D: tableau 2D (6x4)
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [11 12 13 14]
 [15 16 17 18]
```

Numpy : vecteurs

In [32]:

```
1 XY1D = XY3D.ravel() # transforme un array a N
      dimensions en un array a 1 dimension
2 print("XY1D\n", XY1D)
3 print("XY1D[12*0 + 4*1 + 3]:",XY1D[12*0 + 4*1 +
       3], "XY1D[12*1 + 4*1 + 3]:",XY1D[12*1 + 4*
      1 + 3])
4 print("XY1D[12*0 + 4*2 + 1]:",XY1D[12*0 + 4*2 +
       1], "XY1D[12*1 + 4*2 + 1]:",XY1D[12*1 + 4*
      2 + 1])
```

Out[32]:

```
XY1D
 [ 1  2  3  4  5  6  7  8  9 10 11 12 11 12 13
    14 15 16 17 18 19 20 21 22]
XY1D[12*0 + 4*1 + 3]: 8 XY1D[12*1 + 4*1 + 3]:
    18
XY1D[12*0 + 4*2 + 1]: 10 XY1D[12*1 + 4*2 + 1]:
    20
```

Listes et numpy arrays : similarités

La copie d'une liste ou d'un tableau numpy est subtile : l'opérateur = est à éviter !
Utiliser **a.copy()** pour les numpy arrays, **list()** pour les listes.

In [33]:
```
a = list(range(1, 5, 1))
b = a     # on affuble la liste d'un second nom
c = list(a)    # on cree une liste c, qui est
    identique a la liste a
b[1] = 9  # en modifiant la liste b, on modifie
     en realite la liste a (c est la meme)
print(a, b, c)    # en revanche, c n est pas
    modifiee
```

Out[33]:
```
[ 1., 9., 3., 4.]
[ 1., 9., 3., 4.]
[ 1., 2., 3., 4.]
```

In [34]:
```
a = np.arange(1, 5, 1)
b = a
c = a.copy()
b[1] = 9
print(a, b, c)
```

Listes et numpy arrays : différences

+ concatène les listes :

In [35]:
```
1 liste_a = [3, 4, 6]
2 liste_b = [2, 3, 5]
3 print(liste_a + liste_b)
```

Out[35]:
```
[3, 4, 6, 2, 3, 5]
```

+ ajoute les éléments (tableaux de même taille ou scalaire) :

In [36]:
```
1 vecteur_a = np.array([3, 4, 6])
2 vecteur_b = np.array([2, 3, 5])
3 print(vecteur_a + vecteur_b)
```

Out[36]:
```
array([5, 7, 11])
```

In [37]:
```
1 print(vecteur_a + 1)
```

Out[37]:
```
array([4, 5, 7])
```

Listes et numpy arrays : différences

***n auto-concatène les listes** (n entier) :

In [38]:
```
1 print(liste_a*2)
```

Out[38]:
```
[3, 4, 6, 3, 4, 6]
```

La multiplication mathématique entre listes impossibles.

*** multiplie les éléments** (tableaux de même taille ou scalaire) :

In [39]:
```
1 print(vecteur_a * 2)
```

Out[39]:
```
array([6, 8, 12])
```

In [40]:
```
1 print(vecteur_a * vecteur_b)
```

Out[40]:
```
array([6,12,30])
```

Quelques fonctions spécifiques à numpy

Initialisation avec des `float` :

- `np.zeros(3)` crée `[0., 0., 0.]`
- `np.ones(5)` crée `[1., 1., 1., 1., 1]`,
- `np.arange(xmin,xmax,dx)` crée un tableau des `xmin + i*dx`, `xmax` **exclu**
- `np.linspace(xmin,xmax,n)` crée un tableau de n termes $\texttt{xmin} + \texttt{i} * \frac{\texttt{xmax} - \texttt{xmin}}{\texttt{n} - 1}$, `xmax` **inclus**

Informations sur le numpy array :

- **a.dtype** : retourne le type des éléments du tableau
- **a.shape** : retourne les dimensions du tableau (utile à N dimensions)
- **a.ndim** : retourne la dimension du tableau (1 pour un vecteur, 2 pour une matrice)

Quelques fonctions spécifiques à numpy

Récupération d'une partie d'un tableau sous condition :

In [41]:
```
1 a = np.array([5, 16, -2, 3, -4, 6])
2 print(a[a < 0])   # condition exprimee entre [ ]
```

Out[41]:
```
array([-2, -4])
```

Opérations sur un tableau :

- `np.sum(a)` : renvoie la somme des éléments de a
- `np.prod(a)` : renvoie le produit des éléments de a
- `np.argmax(a)` : renvoie un tableau d'indices des valeurs maximales
- `np.conj(a)` : renvoie le conjugué de a
- `np.transpose(a)` ou `a.T` : renvoie le transposé de a
- `a.astype(int)` : renvoie un tableau de a converti en entiers

Listes et numpy arrays : similarités

Accès à un élément via son indice : `a[i]`, où `i` va **de 0 à n-1**

- Liste de -10 (inclus) à 15 (exclu) par pas de 0.5 : `lis = list(range(-10.,15.,0.5))`
- Taille de la liste, min et max : `len(lis)`, `min(lis)`, `max(lis)` (si ordonnable)
- Tri par valeur croissante (si ordonnable) : `lis.sort()`
- Ajout d'un élément, suppression d'un élément : `lis.append(valeur)`, `lis.remove(valeur)` (première supprimée), `lis.pop(indice)`

- Liste de -10 (inclus) à 15 (exclu) par pas de 0.5 : `a = np.arange(-10.,15.,0.5)`
- Taille de la liste, min et max : `np.size(a)`, `np.min(a)`, `np.max(a)`
- Tri par valeur croissante : `a = np.sort(a)`

Entrées et Sorties avec numpy

Numpy peut lire – `numpy.loadtxt()` – ou sauvegarder – `numpy.savetxt()` – des tableaux dans un simple fichier ASCII :

In [42]:
```
x = range(1000000)    # entiers de 0 a 999 999
# Sauvegarde dans le fichier
np.savetxt('donnees.txt', x)
# Relecture a partir du fichier
y = np.loadtxt('donnees.txt')
# Test d'egalite stricte
print(np.all(x == y))
```

Out[42]:
```
True
```

Entrées et Sorties avec numpy

Le format texte n'est pas optimal pour de gros tableaux, le format binaire .npy est beaucoup plus compact.

In [43]:
```
x = range(1000000)  # entiers de 0 a 999 999
# Sauvegarde dans le fichier 'donnees.npy'
np.save('donnees.npy', x)
# Relecture a partir du fichier 'donnees.npy'
y = np.load('donnees.npy')
print(np.all(x == y))
```

Out[43]:
```
True
```

Le fichier `donnees.npy` est ici 6,5 fois moins volumineux que le fichier `donnees.txt`, et encore on n'a stocké que des entiers ! La représentation en texte des floats est encore plus coûteuse.