Introduction à Python pour les mathématiques

Laurent Tournier

Ces quelques pages donnent brièvement quelques bases du langage Python. Je vous invite à les compléter avec les innombrables ressources (cours, tutoriels) disponibles sur internet pour mieux comprendre cette introduction, si vous débutez en programmation, ou pour approfondir le sujet (vérifiez toutefois que ces ressources portent bien sur Python 3, et non sur Python 2).

1 Introduction et installation

Python est un langage de programmation, c'est-à-dire une notation pour décrire des algorithmes. Il est décrit dans un document de référence (cf. `https://docs.python.org/`). Il existe divers interpréteurs de Python : ce sont des logiciels informatiques qui, à partir d'un texte écrit en Python (un *programme*), font exécuter à l'ordinateur l'algorithme qu'il décrit.
Le langage Python a évolué depuis sa création en 1991 ; on fait le choix d'utiliser ici la version la plus récente, Python 3. En revanche, le choix de l'interpréteur n'a pas d'importance. Par exemple, je propose de télécharger et d'installer la distribution Python *Anaconda* sur `www.anaconda.com/distribution/` (choisir la version supérieure à 3) ; disponible sous Windows, MacOS et Linux, c'est un ensemble contenant un interpréteur et de nombreux modules de calcul scientifique (qui pourraient se télécharger séparément), ainsi qu'un éditeur nommé *Spyder* qui facilite l'écriture et l'exécution des programmes en Python.
Alternativement, on pourrait simplement télécharger un interpréteur Python sur `https://www.python.org/downloads/`, que l'on utilise depuis un terminal, écrire les programmes avec un quelconque éditeur de texte (comme Notepad) et télécharger les modules nécessaire au besoin. Cette approche demanderait cependant davantage de connaissances en informatique que l'utilisation de la distribution Anaconda.
Dans la suite, on utilisera le mot "Python" pour désigner aussi bien le langage que l'interpréteur, c'est-à-dire le logiciel permettant d'exécuter un programme écrit en Python.
Le choix de Python dans ce cours tient à plusieurs raisons : il s'agit d'un langage de haut niveau (les considérations techniques, notamment la gestion de la mémoire, sont gérées automatiquement, ce qui permet un développement plus naturel, rapide et moins sujet à erreurs), relativement simple et néanmoins puissant et rapide, ayant un fonctionnement interactif facilitant l'apprentissage (et rappelant matlab), et surtout disposant grâce à sa popularité de nombreux modules très élaborés pour des usages plus spécialisés, notamment, en ce qui nous concerne, en calcul scientifique (outils et méthodes numériques, représentations graphiques), analyse de données, analyse d'images, intelligence artificielle, etc.

2 Dialoguer avec la console Python

Les programmes Python sont *interprétés*, c'est-à-dire lus et exécutés ligne-à-ligne (à la différence des langages compilés, où le programme entier est analysé, avant d'être exécuté). Cela donne lieu à deux modes d'utilisation :
- ou bien on entre des commandes une à une pour suivre progressivement le résultat de leur exécution (à la façon d'une calculatrice),
- ou bien on écrit un programme de plusieurs lignes, que l'on sauvegarde dans un fichier, que l'on envoie ensuite à l'interpréteur Python qui va l'exécuter ligne-à-ligne.

On utilisera en général la seconde méthode, mais la première est utile pour faire des tests, et notamment pour apprendre à utiliser Python : on va donc commencer par là.
On trouve dans Spyder une sous-fenêtre "Console IPython" qui permet d'envoyer des commandes à l'interpréteur Python (on obtiendrait le même résultat en exécutant `python3` dans un terminal). Taper par exemple

```
1+2
```

puis appuyer sur la touche Entrée ; l'interpréteur affiche `3`, ce qui est la valeur de l'instruction précédente. Taper ensuite

```
a=1
```

appuyer sur Entrée (rien ne s'affiche car cette instruction d'affectation n'a pas de valeur), puis

```
a+2
```

pour constater que la *variable* `a` a été remplacée par sa valeur 1 dans ce calcul.

3 Objets et variables

Les données en Python sont représentées sous forme d'**objets**. Chaque objet est constitué de son **identifiant**, de son **type** et de sa **valeur** :

- l'**identifiant** est un nombre entier associé à chaque objet, de telle sorte qu'à tout instant chaque objet actif a un identifiant différent : c'est une façon pour l'interpréteur de désigner la position dans la mémoire où est stocké l'objet (il n'est en général pas utile de connaître l'identifiant d'un objet) ;
- le **type** est la nature de l'objet (par exemple : nombre entier, liste, chaîne de caractère, fonction,...) ;
- la **valeur** est le contenu de l'objet (par exemple : -36, [1,2,3.5,"abc"], "Nom Prénom", $x \mapsto x^2$,...).

Dès que l'on utilise un nombre, une chaîne de caractères,..., un objet lui est associé. Ainsi, quand on entre la commande

```
36
```

un objet de type entier est créé, contenant la valeur 36 (et possédant un nouvel identifiant). Les commandes `type(36)` et `id(36)` renvoient le type et l'identifiant de l'objet créé lors de l'exécution de cette ligne. De même, si on entre la commande

```
"Bonjour !"
```

alors un objet de type chaîne de caractères est créé contenant le texte "Bonjour !".
Pour pouvoir réutiliser un objet au cours d'un programme, il faut lui associer une **variable** : c'est un nom qui fait référence à un objet. On associe un objet à une variable par une **affectation** : par exemple, la commande

```
a=36
```

crée un objet de type entier contenant la valeur 36, et crée une variable nommée `a` et faisant référence à cet objet. Les commandes `type(a)` et `id(a)` renvoient maintenant le type et l'identifiant de l'objet référencé par `a`. On peut vérifier la valeur de `a` en entrant la commande

```
a
```

Ensuite, la commande

```
a+5
```

crée un nouvel objet de type entier contenant la valeur $36 + 5 = 41$. On peut changer l'objet référencé par une variable : la commande

```
a=a+5
```

associe à `a` un objet dont la valeur est 41.
Selon son type, diverses fonctions ou opérations peuvent être appliquées à un objet. Il peut s'agir de fonctions définies par l'utilisateur (on le verra), ou prédéfinies : la commande

```
print(a)
```

a pour effet d'afficher le contenu de l'objet `a` (s'il se prête à un affichage). On a aussi déjà vu `id(a)` et `type(a)`. Certaines fonctions sont associées à un objet : par exemple, la commande suivante

```
"Texte quelconque".upper()
```

renvoie la chaîne de caractères `'TEXTE QUELCONQUE'` ; on a fait appel à la fonction `upper` de l'objet de type chaîne de caractères qui a été créé pour contenir le texte "Texte quelconque". Cette fonction ne prend pas d'argument, d'où les parenthèses vides. Voyons un exemple avec un argument :

```
"Texte quelconque".find("q")
```

renvoie `6`, qui est l'indice de la première occurrence de "q" dans la chaîne (en commençant par l'indice 0). Le code suivant est équivalent au précédent :

```
a="Texte quelconque"
a.find("q")
```

Voyons les principaux types d'objets, et quelques opérations sur ceux-ci.

3.1 Entiers (`int`)

En Python, les entiers relatifs n'ont pas de valeur maximale ni minimale (dans la limite de la mémoire disponible). On les écrit naturellement en base 10. On peut aussi utiliser les bases 2, 8 ou 16 : `2019`, `0b11111100011`, `0o3743` et `0x7E3` représentent 2019 en base 10, 2, 8 et 16 respectivement (en base 16, les lettres de A à F représentent les "chiffres" de 10 à 15).
Les principales opérations sur les entiers sont les suivantes :

Opérateur	Opération	Exemple	Résultat de l'exemple
`+`	addition	`1+2`	`3`
`-`	soustraction	`1-2`	`-1`
`-`	opposé	`-1`	`-1`
`*`	multiplication	`2*3`	`6`
`**`	exponentiation (puissance)	`2**3`	`8`
`/`	division réelle	`10/3`	`3.3333333333`
`//`	division entière	`10//3`	`3`
`%`	reste de la division entière (modulo)	`10%3`	`1`

À tester dans la console :

```
a=2**(2**5)+1
a
a%641
2019**100
```

3.2 "Réels", nombres à virgule flottante en double précision (`float`)

Les nombres réels en revanche ne sont pas stockés de façon exacte en général, mais arrondie de façon à occuper en mémoire un espace prédéfini (8 octets, soit 64 bits).
*Plus précisément : tout réel non nul x peut se représenter de façon unique sous la forme $x = a \cdot 2^b$, où $a \in [1,2[$ est un réel et $b \in \mathbb{Z}$; a est la **mantisse** et b l'**exposant** de x. En Python, la mantisse est écrite avec 52 bits et l'exposant avec 11 bits. Avec le signe (sur 1 bit), un nombre flottant occupe donc 64 bits en mémoire.*
Pour entrer un nombre flottant en base 10, on écrit par exemple, `2.345e-7` pour représenter le réel $2.345 \cdot 10^{-7}$.
Ou encore `2.` (ou `2.0`) pour représenter le réel 2 (tandis que `2` représente l'entier 2).
On dispose des opérations classiques `+`, `-`, `*`, `/`, `**`. De nombreuses fonctions classiques (sinus, exponentielle, etc.) sont disponibles dans le module `maths` (on y reviendra).

À tester dans la console :

```
a=0.2
type(a)
a+0.1
a+0.1-0.3
2019.**100
sin(pi/3)
from math import *
sin(pi/3) # Utiliser la touche "flèche haut" pour parcourir l'historique
```

Le texte qui suit le caractère `#` n'est pas interprété comme un code en Python : c'est un **commentaire** à destination de quiconque lira le programme.

Pour convertir un objet d'un type en un objet d'un autre type, lorsque cela a été défini, on utilise le nom du nouveau type comme une fonction :

```
int(4.0)
int(2.5)
int(-2.5)
float(3)
```

3.3 "Complexes"

Les nombres complexes sont représentés par un couple de nombres "réels" comme ci-dessus : deux nombres à virgule flottante en double précision (partie réelle et partie imaginaire).
Le nombre complexe $2i$ s'écrit `2j` en Python : on écrit `1.5+2j` pour représenter le nombre complexe $1.5 + 2i$. (Attention, pour représenter i, on écrira `1j` : par exemple `2-1j`)

À tester dans la console :

```
a=1j
type(a)
a**2
b=1/(2-1j)
b.real
b.imag
```

3.4 Listes (`list`)

Une liste est une suite finie d'objets, chacun de type quelconque (y compris le type liste). On définit une liste avec des crochets : `[1, 2e-3, 1-j]`. On peut définir une liste vide : `[]`. Pour ajouter un élément `x` à la fin d'une liste `a`, on utilise `a.append(x)`.
La concaténation (mise "bout à bout") s'obtient avec `+` : `[1,2]+[3]` est un objet liste `[1,2,3]`. *Remarque : cette opération crée un nouvel objet liste, tandis que* ***`a.append(3)`*** *ne change que la valeur de l'objet* ***`a`****.*
Le type liste est assez sophistiqué pour rendre rapides à la fois les opérations d'ajout d'élément en fin de liste et les accès au n-ième élément (ce qui fait qu'une liste est souvent un bon équivalent d'un "tableau" dans d'autres langages) ; en revanche, ajouter un élément en milieu de liste est relativement coûteux en temps d'exécution [1].
L'extraction d'éléments s'obtient comme suit :

```
a=[1,2,3,4]
a[0]
a[1]
```

On peut aussi modifier les éléments d'une liste :

```
a[0]=4
```

Plus généralement, on peut extraire une sous-liste composée des indices r tels que $i \leq r < j$ par `a[i:j]`. Et `a[i:j:k]` contient les indices $i + rk$ (où $r \in \mathbb{N}$) tels que $i \leq i + rk < j$ (si $k < 0$, on obtient une liste à indices décroissants).

À tester dans la console :

```
a=[0,1,2,3,4,5,6]
a
a[0]
a[1]
a[7]
a[0:3]
a[4:7]=[2,1,0]
a
a[4:7]=[4,5,6,7,8]
a
a[0:7:2]
a[-1]
a[4:]
a[:4]
a[:]
a[::2]
a[6:3:-1]
a[::-1]
```

Attention aux copies de listes : si `a` désigne une liste, alors la commande `b=a` assigne à `b` le **même** objet (même type et même valeur, mais aussi même identifiant), c'est-à-dire que modifier `b` va aussi modifier `a`. Pour assigner à `b` une nouvelle liste, dont le contenu est identique à celle désignée par `a`, on peut utiliser `b=a.copy()`

```
a=[0,1,2,3]
b=a
a[0]=5
b[0]
```

1. On pourra consulter `https://wiki.python.org/moin/TimeComplexity` pour connaître la complexité des opérations sur les différents types en Python.

```
b=a.copy()
a[0]=0
b[0]
```

Il est souvent utile de parcourir une liste et d'effectuer une opération sur chaque élément. On utilise pour cela la structure `for` *variable* `in` *liste*:

```
a=[1,5,2,-1,9,4,3]
s=0
for i in a:
    print(i*i)
    s+=i        # équivalent à s=s+i
print(s)        # ceci affichera la somme des éléments de a
print(sum(a))   # pour comparaison
```

NB : il est essentiel d'indenter le bloc des instructions qui seront répétées (en le précédant de 4 espaces). Plutôt que de l'écrire dans la console, il sera plus pratique d'écrire le programme ci-dessus dans un script pour l'exécuter.
La structure `for` fournit aussi une manière pratique de créer des listes à partir d'une autre :

```
a=[0,1,2,3,4]
[i**2 for i in a]
```

Pour les suites arithmétiques, Python dispose d'un format de "liste abstraite" nommé `range`. Ces "fausses" listes ne sont mémorisées que via leur premier terme, leur borne supérieure et leur incrément (au lieu de mémoriser chaque terme), ce qui économise de la place en mémoire. On peut en déduire une liste "classique" par `list(a)`

```
a=range(10)
a[0]
a[1]
a[10]
list(a)
list(range(0,20,3))
```

Cela est très utilisé dans les structures `for` :

```
s=1
for i in range(1,101):
    s*=i
print(s)
```

```
[i*i for i in range(10)]
sum(1/i**2 for i in range(1,1000))
```

Si `a` est une liste d'entiers ou de réels, `sorted(a)` renvoie une (nouvelle) liste contenant les éléments de `a` triés par ordre croissant. En revanche, `a.sort()` trie la liste `a` sans créer de nouvelle liste.
`len(a)` renvoie la longueur de la liste `a`, c'est-à-dire son nombre d'éléments.

3.5 Chaînes de caractères (`str`)

Une chaîne de caractère contient un texte, c'est-à-dire une suite de caractères ; on peut les définir avec des apostrophes `'Exemple'`, des guillemets anglais `"Exemple"` ou trois apostrophes `'''Exemple'''` (les premiers autorisent les textes contenant des guillemets anglais, les deuxième autorisent les apostrophes, et les derniers autorisent les deux et même les sauts de ligne).
Comme pour les listes, on peut en extraire des caractères par extraction, ou les parcourir dans une boucle `for`. En revanche, on ne peut pas les modifier caractère par caractère.

```
a="Exemple d'expression"
a[0]
for c in a:
    print(c)
```

Au passage, on remarque que `print(c)` affiche `c` puis un retour à la ligne. Cela peut être modifié : `print(c,end=',')` écrit une virgule après `c`, et `print(c,end='')` n'affiche que `c` (sans retour à la ligne).
Diverses fonctions sont spécifiques aux chaînes de caractères : par exemple, `a.lower()` renvoie une chaîne convertissant `a` en minuscules (cela crée une nouvelle chaîne car une chaîne ne peut être modifiée).

4 Branchement conditionnel (`if`)

Dans la suite, on écrira les programmes proposés dans des scripts.
Pour que certaines instructions ne s'exécutent que si une condition est réalisée, on utilise une structure `if` :

```
a=input("Entrer un entier : ")  # a contient une chaîne entrée par
    l'utilisateur
i=int(a)  # On convertit a en entier
if(i<10):
    print("L'entier est strictement inférieur à 10")
else:
    print("L'entier est supérieur à 10.")
```

On peut aussi ne pas spécifier de bloc `else`.
La condition est un objet de type **booléen** (ce type n'a que deux valeurs possibles, `True` et `False`), en général obtenu par une opération de comparaison, ou une opération entre booléens :

Opérateur	Opération	Exemple	Résultat de l'exemple
`<`	strictement inférieur à	`1<2`	`True`
`<=`	inférieur ou égal à	`1<=1`	`True`
`==`	égal à	`1==2`	`False`
`!=`	différent de	`1!=2`	`True`
`or`	ou	`(1<2)or(2<1)`	`True`
`and`	et	`(1<2)and(2<1)`	`False`
`not`	non	`not(1<2)`	`False`

Signalons aussi l'opération `in` : si `b` est une liste, alors `a in b` renvoie `True` si l'un des éléments de `b` (au moins) est égal à `a`.

```
a=[i*i for i in range(10)]
20 in a
25 in a
```

5 Boucle conditionnelle (`while`)

On a déjà vu la "boucle `for`" qui parcourt les éléments d'une liste. La "boucle `while`" répète un bloc d'instructions tant qu'une condition reste satisfaite :

```
S=n=0
while(S<10):
    n+=1
    S+=1/n
print(n)
```

6 Fonctions (`def`)

Comme en mathématiques, une fonction en Python prend en paramètre des arguments, et renvoie une valeur. Une fois définie, on peut y faire appel comme aux fonctions prédéfinies (`print`, `sorted`, etc.) :

```
def factorielle(n):
    produit=1
    i=1
    while(i<=n):
        produit*=i
        i+=1
    return(produit)

factorielle(10)
```

S'il n'y a pas d'instruction `return`, la fonction ne renvoie pas de valeur. C'est le cas de la fonction `print`, qui réalise une action (elle affiche dans la console la valeur d'un objet) mais ne renvoie pas de valeur.
Une fonction peut faire appel à elle-même ; elle est alors dite **récursive**. Cette structure permet parfois une écriture très concise :

```
def factorielle(n):
    if(n==0):
        return 1
    else:
        return n*factorielle(n-1)

print(factorielle(10))
```

Une fonction peut prendre plusieurs paramètres :

```
def maximum(a,b):
    if(a>b):
        return a
    else:
        return b

print(maximum(1,2))
```

On peut rendre certains paramètres optionnels en leur donnant des valeurs par défaut :

```
def logarithme(x,base=2):
    from math import log   # on importe la fonction log (cf. section
       suivante)
    return log(x)/log(base)

print(logarithme(256))
print(logarithme(256,2))
print(logarithme(256,base=10))
print(logarithme(base=10,x=256)) # l'ordre n'importe pas tant que l'on nomme
```

Pour renvoyer plusieurs valeurs, on peut renvoyer un objet liste contenant chaque valeur ou, plus simplement, un objet tuple (`a,b` est un objet de type tuple, c'est une liste non modifiable) :

```
def division(p,q):
    return p//q, p\%q

a,b=division(100,7)
print(a,b)
```

Une fonction en Python est un objet : on peut notamment l'utiliser comme paramètre d'une autre fonction.

```
def somme(f,a,b):
    return sum(f(i) for i in range(a,b+1))

def carre(x):
    return x*x

print(somme(carre,1,10))
```

Il est possible de définir des fonctions simples par une syntaxe plus légère : `lambda x:f(x)` définit une fonction (sans lui donner de nom) qui à `x` associe `f`(x). Dans l'exemple précédent, on aurait ainsi pu écrire

```
print(somme(lambda x:x*x,1,10))
```

ou

```
carre=lambda x:x*x
print(somme(carre,1,10))
```

Importer un module

Des définitions de fonctions (et de types) peuvent être stockées dans des fichiers, appelés **modules**, qui peuvent être mis à disposition de la communauté d'utilisateurs de Python pour être utilisés quand on en a besoin. De nombreux modules sont présents dans la distribution Anaconda, et il est possible d'en télécharger d'autres. Pour charger les définitions présentes dans le module `math` par exemple, on utilise la syntaxe

```
from math import *
```

On peut ensuite utiliser les fonctions `log`, `sin`, etc. Pour importer quelques fonctions seulement, on pourrait écrire

```
from math import log,sin
```

Si une fonction nommée `log` était déjà définie, elle serait "écrasée" par cette importation. Par exemple, on utilisera souvent le module `numpy` qui a aussi une fonction `log`, d'où un conflit, que l'on résout en utilisant

```
from math import *
import numpy
```

Alors `log` fera référence à la fonction de `math`, et `numpy.log` à celle de `numpy`. Pour alléger, on écrira plutôt

```
from math import *
import numpy as np
```

et `np.log` pour faire appel au `log` de `numpy`.

7 Calculs en précision arbitraire avec Decimal

On se rappelle que les réels (type `float`) sont mémorisés avec une précision fixée (environ 15 chiffres significatifs). Le module `decimal` fournit un type permettant de stocker et de faire des opérations avec une précision plus grande.

```
from decimal import *
getcontext().prec=100   # choix de la précision (nombre de chiffres
   significatifs en base 10)
a=Decimal('1.5')    # On entre un nombre comme une chaîne de caractères
b=Decimal(7)    # ou comme un entier
1/b # Dans les opérations, les entiers sont convertis en Decimal
sum(1/Decimal(factorial(n)) for n in range(0,72))   # (Non optimisé)
```

Exemple. On souhaite calculer, avec une grande précision, l'arctangente d'un nombre $x \in]0,1]$ via le développement en série entière

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1}.$$

Par le théorème des séries alternées, le reste est inférieur à son premier terme :

$$\left| \sum_{n=N+1}^{\infty} \frac{(-1)^k}{2k+1} x^{2k+1} \right| < \frac{x^{2N+3}}{2N+3}.$$

On a donc un critère pour garantir une erreur inférieure à 10^{-d} entre $\arctan x$ et la somme partielle jusqu'à $n = N$. Sachant que $4 \arctan \frac{1}{5} - \arctan \frac{1}{239} = \frac{\pi}{4}$ (formule de Machin), on peut d'ailleurs en déduire un calcul des décimales de π :

```
from math import *
from decimal import *

def arctan_approx(x,erreur):
    x2=x*x
    puissance=terme=somme=x
    denom=signe=1
    while(terme>erreur):
        puissance*=x2
        signe*=-1
        denom+=2
        terme=puissance/denom
        somme+=terme*signe
    return(somme)

e=Decimal('1e-1000') # Erreur voulue
getcontext().prec=1050 # Précision du calcul (marge pour erreurs d'arrondis)
pi_approx=16*arctan_approx(Decimal('0.2'),e)-4*arctan_approx(1/Decimal('239'),e)
s=str(pi_approx) # Conversion en texte
print(s[:1002]) # Affichage de 1000 décimales
```

8 Calculs matriciels avec Numpy

Le module Numpy fournit le type d'objet `array` pour les tableaux de nombres, qui est plus adapté aux calculs matriciels que les listes.
La documentation de numpy (en anglais) se trouve sur internet : `https://docs.scipy.org/doc/`

```
import numpy as np
a=np.array([1, 2, 3, -1.5])
a[0]
5*a+1
1/a
a**2
np.exp(a)
a[:]=1
a
a.sum()
```

On peut créer un tableau de valeurs équiréparties avec `arange(a,b,pas)` (valeurs `a`, `a+pas`,..., sans dépasser `b`), ou avec `linspace(a,b,n)` (`n` valeurs équiréparties de `a` à `b`).
On dispose également des fonctions pratiques `np.zeros(m)` et `np.ones(m)` pour créer un tableau de taille m initialisé à 0 ou à 1, et `np.random.random(m)` pour initialiser avec des nombres aléatoires de loi uniforme sur [0,1] indépendants (on y revient plus loin).
On peut aussi manipuler des tableaux bidimensionnels (matrices) :

```
import numpy as np
a=np.array([[1,2,3],[4,5,6]])
a[0,1]
a*a
np.sin(a)
np.shape(a)
np.transpose(a)
a>=2
(a>=2).sum()
a[a>=2]=0
a
np.hstack([a,a])
np.vstack([a,a])
np.tile(a,[2,4])
```

On remarque que, pour calculer des fonctions usuelles (`exp`, `sin`, `sqrt`,...) de chaque élément, on utilise la fonction de Numpy correspondante (`np.exp`, `np.sin`, `np.sqrt`,...) sans quoi on obtient une erreur. Alternativement, si on dispose d'une fonction `f` qui s'applique à un réel et renvoie un réel, on peut la **vectoriser**, c'est-à-dire en déduire une fonction qui s'applique à un tableau, composante par composante, par la commande `np.vectorize(f)` :

```
def f(x):
    if(x<=0):
        return 0.   # On met un point pour indiquer que f renvoie un réel
    else:
        return sqrt(x)

vf=np.vectorize(f)
print(vf(np.arange(-5,5,1)))
```

Certaines commandes fournissent des tableaux bidimensionnels particuliers :
- `np.zeros((m,n))` crée un tableau de `m` lignes et `n` colonnes, initialisé à 0
- `np.ones((m,n))` de même avec la valeur 1
- `np.eye(m)` crée une matrice identité
- `np.random.random((m,n))` crée un tableau initialisé avec des valeurs tirées aléatoirement, indépendamment entre elles, selon la loi uniforme sur [0,1].

9 Graphes avec Matplotlib

Le module `matplotlib.pyplot` contient plusieurs fonctions pour représenter graphiquement des fonctions, notamment `plot` et `hist` (qui ressemblent aux fonctions Matlab du même nom).

Si `X=[x1,x2,...]` et `Y=[y1,y2,...]` sont des `array` ou des `list` de même longueur, `plot(X,Y)` représente graphiquement les points (`x1`,`y1`), (`x2`,`y2`),... Dans un script, le graphe est affiché par la fonction `show()`.

```
from math import *
import numpy as np
import matplotlib.pyplot as plt

X=np.linspace(0,2*pi,1000)
Y=[cos(x) for x in X]          # ou Y=np.cos(X)
plt.plot(X,Y)

plt.plot([0, 2*pi],[0,0],'--', color='r')
plt.show()
```

La syntaxe complète est `plot(X,Y,paramètres)` où `paramètres` est une suite de la forme

`'--',color='r',linewidth=5,...`

où

- `'--'` correspond à un trait en pointillés, et peut être remplacé par `'-'` (trait plein, par défaut), `'.'` (points), `'+'` (croix), `'o'`, `'-o'`,...
- `'r'` correspond à la couleur rouge, et peut être remplacé par `'k'`, `'b'`, `'g'`, `'y'`,...

Le détail des paramètres peut se trouver ici : `https://matplotlib.org/tutorials/index.html`
La fonction `hist(X,bornes,paramètres)` affiche un histogramme des données dans `X`, avec des corbeilles dont les bornes sont données par `bornes`. Alternativement, `hist(X,bins=n,paramètres)` fixe à `n` le nombre de corbeilles et ajuste les bornes en conséquence.
En plus des paramètres précédents, signalons `density=True` qui fournit un histogramme normalisé (c'est-à-dire normalisé en ordonnée pour que l'aire sous la courbe vale 1), `rwidth=0.5` qui fixe la largeur relative des barres (ici, la moitié de la largeur de la corbeille), et `edgecolor='k'` qui fixe la couleur de la bordure des barres (ici, noir).

```
import numpy as np
from numpy.random import randint
import matplotlib.pyplot as plt

X=randint(1,7,size=1000)
bornes=np.arange(1,8)-0.5
plt.hist(X,bornes,density=True,rwidth=0.5,edgecolor='k')
plt.title('Histogramme normalisé des fréquences parmi 1000 lancers de dé')
plt.show()
```

10 Tirages aléatoires avec numpy.random

Pour générer des nombres aléatoires, plusieurs modules sont disponibles, notamment `random` et `numpy.random` (qui fait partie de Numpy). Ce dernier a l'avantage de pouvoir générer efficacement des tableaux (`array`) de nombres aléatoires de taille donnée.

```
from numpy.random import random, randint

random()
random(10)
random((2,5))      # ou random(size=(2,5))
randint(1, 7, size=50)
```

Références

`http://fsincere.free.fr/isn/python/cours_python.php`
`https://www.brianheinold.net/python/A_Practical_Introduction_to_Python_Programming_Heinold.pdf`
`http://inforef.be/swi/download/apprendre_python3_5.pdf`

Pour obtenir une description de l'usage d'une fonction, on peut utiliser la commande `help` en Python (par exemple, `help(print)`). Pour toute aide sur Python, on peut se référer à la documentation de référence `https://docs.python.org/fr/3/`, ou faire une recherche sur internet car de nombreuses ressources pédagogiques sont disponibles.