Liste de commandes PYTHON usuelles pour les TP

Ce document est une référence de commandes usuelles que nous utiliserons dans Python, avec l'accent mis sur les commandes faisant parties des modules `numpy`, `matplotlib` et `scipy`, qui sont fortement utilisés pour l'usage scientifique de Python. Nous omettons quelques fonctionnalités de base de Python que nous aborderons tout de même en cours.

Nous recommandons d'utiliser les références suivantes :

Références

- `http://www.math.u-psud.fr/~lemaire/polyl3python.pdf`
  Introduction à Python 3, polycopié de Sophie Lemaire (Orsay). Présente une grande partie des notions utilisés dans ce cours.
- `http://docs.python.org/fr/3` (français) et `http://docs.python.org/3` (anglais)
  La documentation officielle de Python. Pour les débutants, nous recommandons en particulier les *chapitres 1 à 4 du tutoriel*, qui devraient suffire en ce qui concerne les fonctionnalités de base
- `http://github.com/jrjohansson/scientific-python-lectures`
  Introduction to Scientific Computing in Python par Robert Johansson. Une excellente introduction pédagogique (en anglais) à `numpy`, `matplotlib` et `scipy`, sous PDF ou comme *notebooks* IPython, visionnable directement sur le web. Les chapitres 1 à 5 du PDF ou les cours (*lectures*) 1 à 4 sont largement suffisants comme prérequis.
- `http://scipy.org/docs.html`
  La documentation officielle de `numpy`, `matplotlib` et `scipy` peu servir comme référence mais semble être trop exhaustive pour une première utilisation.

Nous utiliserons la version **3** de **Python** et nous travaillerons avec l'environnement de développement **Spyder 3**. *Attention :* vérifier à chaque fois que vous avez bien démarré Spyder 3 et non Spyder 2 (dans le titre de la fenêtre, vous devez voir « Spyder (Python 3.X) » avec X un nombre).

Nous commencerons chaque TP par importer les modules `numpy`, `matplotlib.pyplot` et `scipy.stats`.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as scs
```

Avant la première utilisation de Spyder

- *Console.* L'interface Spyder se compose de trois fenêtres principales : l'éditeur, la console et la fenêtre d'aide. Vérifier que la console est une console *IPython* et non pas une console classique Python (regarder le nom de la console). Pour ouvrir une console *IPython* aller dans le menu : `Consoles`
- *Graphiques.* Les tracés peuvent soit apparaître dans la console IPython, soit dans une nouvelle fenêtre. Il est mieux de les faire apparaître dans une nouvelle fenêtre ; cela permet d'interagir de manière interactive avec les tracés, par exemple en zoomant sur une partie du tracé. Pour cela, aller dans le menu `Outils -> Préférences -> Console IPython`, puis l'onglet `Graphiques`, puis sélectionner `Sortie : Automatique`. Puis redémarrer Spyder.
- *Dossier de travail.* Définir un dossier facilement retrouvable dans lequel vous enregistrerez vos documents (exemple : créer un dossier avec le nom du cours dans le dossier `Documents`). Avant d'entrer du code dans l'éditeur, enregistrer le fichier dans ce dossier sous le nom `TP1_NomPrénom.py`. Pour la prochaine séance, refaire ce procédé avec le nom `TP2_NomPrénom.py` et ainsi de suite.

Utilisation de Spyder

— *Editeur : exécution du code et cellules.* Le code Python rentré dans l'éditeur peut être exécuté via la touche F5 ou l'icône  . Le résultat de l'exécution sera affiché dans la console et des fenêtres graphiques seront éventuellement ouvertes. Afin de ne pas toujours exécuter l'ensemble du script de l'éditeur, mais seulement quelques lignes, on prendra l'habitude d'utiliser des *cellules*. Par exemple, on pourra créer une cellule par question d'un TP dans laquelle on insérera le code Python répondant à la question. Pour délimiter les cellules, on utilise `#%%`. Pour exécuter une cellule donnée, il suffit alors de cliquer sur une cellule et de l'éxécuter via CTRL+ENTREE ou l'icône  . Le raccourci MAJ+ENTREE ou l'icône  peuvent également être utilisés, ils font en même temps avancer le curseur à la prochaine cellule, ce qui permet d'exécuter rapidement plusieurs cellules.

— *Aide en ligne.* En plus de l'aide classique dans Python (avec la fonction `help()` ou la commande `?`), Spyder propose une aide interactive qui permet d'avoir facilement accès à la documentation d'une fonction (ou plus généralement, d'un objet). Il suffit d'utiliser le raccourci clavier CTRL+I quand le curseur de l'éditeur se trouve sur la fonction en question.

Attention : Pour les fonctions du module `matplotlib`, le raccourci CTRL+I ne fonctionne pas dans l'éditeur. Pour contourner ce problème, l'utiliser dans la *console* : taper le nom de la fonction dans la console, puis appuyer sur CTRL+I.

Listes et tableaux

Python inclut un type `list` qui permet de manipuler une liste de valeurs numériques ou d'autres objets. La construction d'une telle liste est très simple : par exemple, l'expression `[0,2,1]` construit une liste contenant les nombres 0,2,1 (dans cet ordre). Malgré la simplicité de l'utilisation et de la définition des listes, nous allons souvent préférer travailler avec des *tableaux*, c'est-à-dire des objets du type `ndarray` défini dans le module `numpy` et construits avec la fonction `np.array`. Ce type ajoute des fonctionnalités utiles au type liste, notamment la possibilité de faire des opérations sur tous les éléments d'une liste à la fois en une simple commande. Voici un exemple :

```
>>> X=np.array([0,1,2])
>>> Y=np.array([3,4,5])
>>> print (X+2)
[2 3 4]
>>> print (X+Y)
[3 5 7]
```

```
>>> print (X*Y)
[ 0  4 10]
>>> print (Y**2)
[ 9 16 25]
>>> print (X**Y)
[ 0  1 32]
```

Remarquer dans les deux dernières lignes de commande l'opérateur puissance `**`. Le produit scalaire de deux vecteurs s'effectue avec la fonction `np.dot` :

```
>>> print (np.dot(X,Y))
14
```

Le type tableau permet aussi de manipuler des matrices ou plus généralement des tableaux de n'importe quelle dimension (liste = dimension un, matrice = dimension deux). Si `A` est un tableau 2D, alors on accède aux entrées de `A` par `A[i,j]`, ce qui donne l'entrée dans la i-ième ligne et j-ième colonne. Pour extraire la i-ième ligne en entier, on utilise `A[i,:]` et pour extraire la j-ième colonne, `A[:,j]`. Finalement, beaucoup de fonctions du package `numpy` permettent de faire des opérations sur les lignes ou les colonnes d'un tableau. Par exemple, `np.sum(A)` retourne un scalaire égal à la somme de tous les éléments du tableau, mais `np.sum(A,axis=0)` retourne un vecteur égal à la somme sur toutes les lignes des éléments dans chaque colonne. Voici des exemples simples :

```
>>> A = np.array([[1,2],[3,4],[5,6]])
>>> A[2,0]
5
>>> A[:,0]
array([1, 3, 5])
>>> A[2,:]
```

```
array([5, 6])
>>> np.sum(A)
21
>>> np.sum(A,axis=0)
array([ 9, 12])
```

La multiplication d'une matrice et d'un vecteur ou de deux matrices s'effectue également avec la fonction `np.dot`. Attention, l'opérateur `*` ne fait que multiplier les entrées une par une :

```
>>> A=np.array([[0,1],[-1,0]])
>>> print (A)
[[ 0  1]
 [-1  0]]
>>> print (A*A)
[[0 1]
 [1 0]]
>>> print (np.dot(A,A))
[[-1  0]
 [ 0 -1]]
 >>> print (np.dot(A,[1,2]))
[ 2 -1]
```

Voici quelques autres fonctions utiles :

- Si `X` est une liste ou un tableau uni-dimensionnel, alors `len(X)` renvoie la longueur de `X`.
- Si `X` est une liste ou un tableau uni-dimensionnel de nombres réels alors
  - `sorted(X)` renvoie une liste avec les valeurs de `X` dans un ordre croissant. `np.sort(X)` fait la même chose, mais renvoie un tableau. Si `X` est un tableau, alors `X.sort()` a le même effet que `X=np.sort(X)`.
  - `np.sum(X)` calcule la somme de toutes les valeurs de `X`,
  - `np.cumsum(X)` renvoie le tableau des sommes cumulées des éléments de `X`.

```
>>> X=[7,1,6,2.5]
>>> np.array(X)
array([7.,1.,6.,2.5])
>>> len(X)
4
```

```
>>> print (np.sort(X))
[ 1.   2.5  6.   7. ]
>>> np.sum(X)
16.5
>>> np.cumsum(X)
array([7.,8.,14.,16.5])
```

- Si $a \leqslant b$ sont des entiers alors `range(`a`,`$b+1$`)` renvoie la liste des entiers compris entre a et b, rangés dans un ordre croissant.

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

- Plus généralement, pour obtenir des listes de nombres réels uniformément espacées, on peut utiliser `np.arange` ou `np.linspace`. Le premier permet de préciser l'espacement entre les nombres et le dernier le nombre de valeurs. Notons que le premier n'inclut pas la borne supérieure, alors que le dernier l'inclut.

```
>>> range(-3,2)
[-3,-2,-1,0,1]
>>> print (np.arange(-5,5,0.5))
[-5.  -4.5 -4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2.
  2.5  3.   3.5  4.   4.5]
>>> print (np.linspace(-5,5,21))
[-5.  -4.5 -4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2.
  2.5  3.   3.5  4.   4.5  5. ]
```

Appliquer une fonction à une liste ou un tableau

Supposons qu'on veut appliquer une fonction à tous les éléments d'une liste ou un tableau. La plupart des fonctions des modules `numpy`, `matplotlib` et `scipy` permettent de faire cela très simplement, il suffit alors de donner la liste ou le tableau en argument de la fonction, comme si c'était un scalaire. Dans les autres cas, il y a la possibilité d'appliquer au préalable la fonction `np.vectorize` à la fonction, ou encore d'utiliser la compréhension de listes de Python. Voici des exemples :

```
>>> import math
>>> X=[0,1,2]
>>> math.sin(X)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a float is required
```

```
>>> np.sin(X)
array([0.,0.84147098,0.90929743])
>>> np.vectorize(math.sin)(X)
array([0.,0.84147098, 0.90929743])
>>> [math.sin(x) for x in X]
[0.,0.84147098, 0.90929743]
```

Les boucles et l'instruction `if`

— La syntaxe pour l'instruction *if* est la suivante

```
if condition:
   instructions
else:
   instructions
```

ou

```
if condition1:
   instructions
elif condition2:
   instructions
...
else:
   instructions
```

— La syntaxe pour une boucle *while* est la suivante

```
while condition:
   instructions
```

— La syntaxe pour une boucle *for* est la suivante

```
for variable in liste:
   instructions
```

— La condition *et* s'écrit avec la commande `and`. La condition *ou* s'écrit avec la commande `or`. Les opérateurs de comparaison sont

`<`	strictement inférieur à	`<=`	inférieur ou égal à
`>`	strictement supérieur à	`>=`	supérieur ou égal à
`==`	égal à	`!=`	différent de

Définir une fonction Python

— Pour créer une fonction qui prend en entrée des paramètres $x_1, \ldots, x_k$ on utilise la syntaxe suivante :

```
def NomDeLaFonction(x1,...,xk):
   instructions
```

— La commande `return x` quitte la fonction et la fait retourner la valeur de l'expression `x`. Par exemple, définissons la fonction `DensGauss` qui prend en entrée (x, m, σ^2) et renvoie la densité en x de la loi Gausienne de paramètres m et σ^2 si $\sigma^2 > 0$ ou emet un message d'erreur sinon.

```
def DensGauss(x,m,sigma2):
   if sigma2<=0:
      print (u'erreur : la variance doit être strictement positive')
   else:
      y=np.exp(-(x-m)**2/sigma2)/np.sqrt(2*np.pi*sigma2)
```

```
    return y

>>> DensGauss(0,-2,-1)
erreur : la variance doit être strictement positive
>>> DensGauss(0,-2,1)
0.0073068827452807761
```

Importer des données à partir d'un fichier

— Pour importer un fichier de données prélablement fourni et nommé *donnees.txt*, on utilise la commande `np.loadtxt('donnees.txt')`.
Attention : le fichier doit se trouver dans le « répertoire de travail » de la console Python. Le plus simple : enregistrer le fichier dans le même dossier que le fichier de code, puis dans Spyder, cliquer avec la touche droite sur l'onglet du fichier code et choisir (selon la version de Spyder) « Répertoire de travail de la console » ou « Définir le répertoire de travail de la console ».
Par exemple, si le contenu de *donnees.txt* est

1	2.3	9.8	-5.9	3.1	4	-8.7	0.6
12	-7	-3.5	0.1	6	7.4	-3.3	7

```
>>> X=np.loadtxt('donnees.txt')
>>> X
array([[  1. ,   2.3,   9.8,  -5.9,   3.1,   4. ,  -8.7,   0.6],
       [ 12. ,  -7. ,  -3.5,   0.1,   6. ,   7.4,  -3.3,   7. ]])
>>> [X,Y]=np.loadtxt('donnees.txt')
>>> print (X)
[  1. ,   2.3,   9.8,  -5.9,   3.1,   4. ,  -8.7,   0.6]
```

Représenter des données graphiquement

Nous allons utiliser plusieurs fonctions du module `matplotlib` pour représenter graphiquement ou tracer des données ou des fonctions, en voici le récapitulatif :

Fonction	*Objectif*
`plt.plot`	Tracer une courbe ou une fonction
`plt.step`	Tracer une fonction en escalier
`plt.stem`	Tracer un diagramme en bâton
`plt.hist`	Représenter des données par un histogramme
`plt.scatter`	Tracer un nuage de points

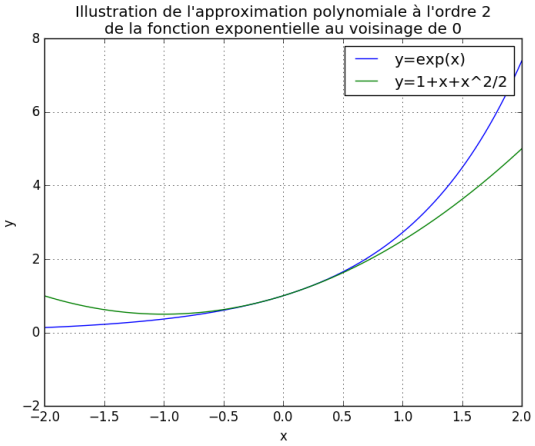
Voici une liste de plusieurs fonctions générales permettant d'ouvrir les fenêtres qui contiennent les diagrammes, d'annoter les diagrammes...

Fonction	*Objectif*
`plt.figure`	Créer ou sélectionner une fenêtre pour les diagrammes
`plt.show`	Indiquer qu'on a terminé un diagramme
`plt.close('all')`	Fermer toutes les fenêtes graphiques
`plt.subplot`	Subdiviser une fenêtre pour en insérer plusieurs diagrammes
`plt.title`	Donner un titre au diagramme
`plt.xlim` / `plt.ylim`	Définir les bornes des axes du diagramme
`plt.xlabel` / `plt.ylabel`	Annoter les axes du diagramme
`plt.legend`	Ajouter une légende (utile quand on superpose plusieurs diagrammes)
`plt.grid`	Ajouter une grille

Il est très important de prendre l'habitude d'annoter les diagrammes avec `plt.title`, `plt.xlabel`, `plt.ylabel` et `plt.legend`. Sans ces annotations, les diagrammes n'ont pas de sens sortis de leur contexte.

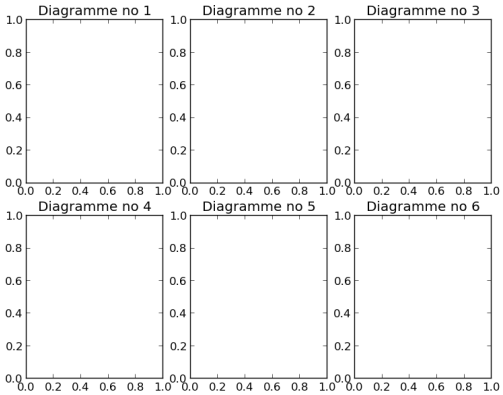
Voici un exemple pour illustrer l'utilisation de ces fonctions :

```
>>> plt.figure()
>>> X=np.linspace(-2,2,1000)
>>> plt.plot(X,np.exp(X))
>>> plt.plot(X,1+X+X**2/2)
>>> plt.grid()
>>> plt.ylim(-2,8)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend([u'y=exp(x)',u'y=1+x+x^2/2'])
>>> plt.title(u'Illustration...')
>>> plt.show()
```



La fonction `plt.subplot(`m`,`n`,`k`)` découpe une même fenêtre graphique en un tableau $m \times n$ et insère les instructions de type `plt.` qui suivent dans la $k^{\text{ième}}$ case :

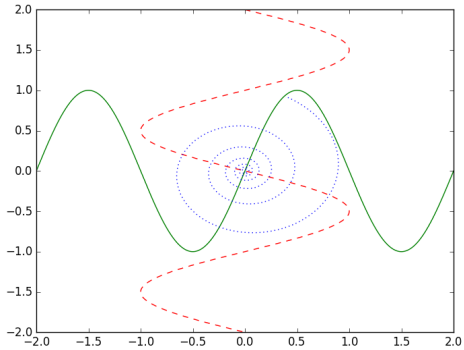
```
>>> plt.figure()
>>> for k in range(1,7):
>>>     plt.subplot(2,3,k)
>>>     plt.title(u'Diagramme no ' + str(k))
>>> plt.show()
```



Voici en détail l'utilisation des cinq fonctions du début de la section :

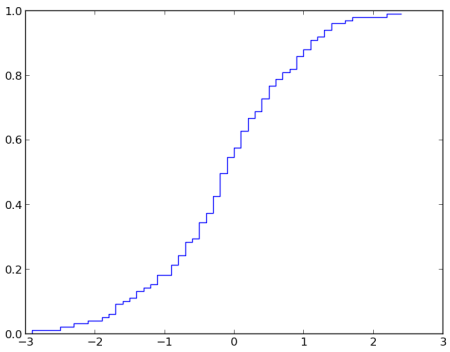
— `plt.plot` : Si nous avons en entrée deux listes `X` et `Y` de même longueur ℓ, alors `plt.plot(X,Y)` trace la courbe dans le plan reliant les points de coordonnées (`X[1]`, `Y[1]`), (`X[2]`, `Y[2]`), ..., (`X[`ℓ`]`, `Y[`ℓ`]`). Des arguments optionnels `color` et `ls` (pour "linestyle") permettent de modifier l'apparence de la courbe.

```
>>> X = np.linspace(-2,2,1000)
>>> Y = np.sin(X*np.pi)
>>> plt.plot(X,Y,color='g')
>>> plt.plot(Y,-X,color='r',ls='dashed')
>>> U=np.exp(X-2)*np.cos(10*X)
>>> V=np.exp(X-2)*np.sin(10*X)
>>> plt.plot(U,V,color='b',ls='dotted')
```



— `plt.step` : Si nous avons en entrée deux listes `X` et `Y` de même longueur ℓ, alors `plt.step(X,Y)` trace la fonction en escalier qui vaut `Y[k]` sur l'intervalle [`X[k-1]`, `X[k]`[. Ceci est très utile pour tracer la fonction de répartition empirique d'un échantillon :

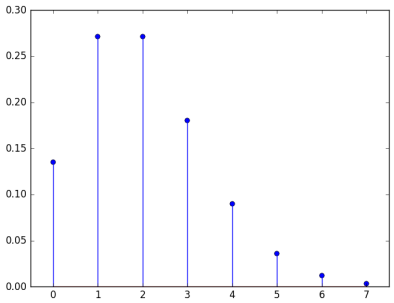
```
>>> X=[-0.2,0.3,-0.3,0.9,-0.1,-0.7,-0.2,2.4,
       -0.7,-0.4,0.1,2.2,-0.1,0.7,0.1,-0.7,1,
       0.3,-2.3,1.1,0,-0.4,0.1,-1.4,0.4,-1.5,
       -0.9,-1.7,0.2,-1.8,0.1,1.1,-0.1,0.6,
       0.6,1,-1.1,-0.8,-0.2,0.2,-1.2,-0.5,
       -0.3,-0.5,-0.5,0.7,-0.3,0.9,-0.3,-0.7,
       -1.1,-2.1,0.4,-0.2,1.4,1.1,0.9,-1.3,
       -2.9,1.7,0.9,-1.7,1.3,0.8,-2.5,-0.3,0,
       0.5,1.2,0.2,-0.5,-1.1,-0.2,-0.6,0.5,
       0.4,-1.6,-0.9,-0.9,0.4,-0.8,-0.4,1.3,
       0,0.1,-0.2,0.5,1.4,-1.9,-0.8,-0.5,
       -1.4,0.5,-0.1,1.6,-0.1,-0.2,-1.7,0.2]
>>> plt.step(np.sort(X),np.arange(0,1,1./len(X)))
```



— `plt.stem` : Si `X` et `Y` sont deux listes ou tableaux de nombres réels et de même longueur ℓ, alors `plt.stem(X,Y)` affiche un diagramme en bâton tel que les bâtons se situent aux points d'absisses `X[1]`,...,`X[`ℓ`]` et ont pour longueurs `Y[1]`,...,`Y[`ℓ`]` respectivement.

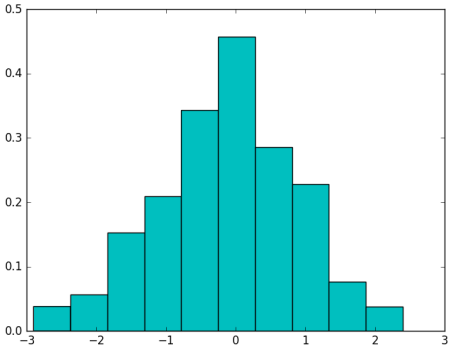
Attention : l'option `color` ne fonctionne pas avec `plt.stem`. On peut par exemple utiliser `markerfmt` qui permet de modifier la couleur et la forme du « marqueur » en haut du bâton, ex. `markerfmt='bD'` pour un diamant bleu, `'gs'` pour un carré vert ou `'ro'` pour un cercle rouge. Se référer à l'aide pour plus d'informations.

```
>>> X=range(0,8)
>>> Y=[0.135,0.271,0.271,0.18,
       0.09,0.036,0.012,0.003]
>>> plt.xlim(-0.5,7.5)
>>> plt.stem(X,Y)
```



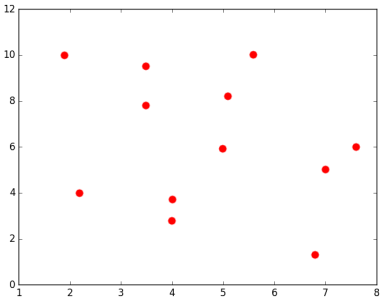
— `plt.hist` : Si `X` est une liste ou un tableau de nombres réels, alors `plt.hist(X,bins=`c`,density=True)` construit un histogramme renormalisé (c'est-à-dire l'échelle de l'ordonnée est changée de telle sorte que l'aire total des c rectangles composant l'histogramme est égale à 1) des valeurs de `X` ayant c classes (par défaut, $c = 10$). Sans l'option `density=True`, la hauteur des barres est égal au nombre de valeurs dans les classes.

```
>>> X = ... #comme l'exemple pour plt.step
>>> plt.hist(X,bins=10,density=True,color='c')
```



— `plt.scatter` : Si `X` et `Y` sont deux listes ou tableaux de nombres réels et de même longueur ℓ, alors `plt.scatter(X,Y)` trace le nuage des points de points (`X[1]`, `Y[1]`), (`X[2]`, `Y[2]`), ..., (`X[`ℓ`]`, `Y[`ℓ`]`).

```
>>> X=[5.6,5,3.5,7.6,2.2,4,1.9,8.8,7,5.1,3.5,4]
>>> Y=[10,5.9,7.8,6,4,3.7,10,1.3,5,8.2,9.5,2.8]
>>> plt.scatter(X,Y,color='r')
```



Échantillonner des variables aléatoires uniformes avec `numpy.random`

`np.random.random_sample(s)` renvoie un tableau de dimensions `s` (entier ou tuple d'entiers) contenant un échantillon de variables (pseudo-)aléatoires de loi uniforme sur $[0, 1[$.

Lois de probabilité avec `scipy.stats`

Le module `scipy.stats` possède de nombreuses fonctionnalités pour les calculs en probabilités et statistiques. Entre autres, il contient un grand nombre de lois de probabilités, représentées par des instances de deux classes : `scs.rv_discrete` et `scs.rv_continuous`, représentant respectivement des lois discrètes et continues. Voici des exemples ; une liste complète peut être obtenue par l'aide interne de Spyder :

Lois discrètes

- `scs.randint(a,b)` :loi uniforme sur $\{a, \dots, b-1\}$
- `scs.bernoulli(p)` : loi de Bernoulli de paramètre p
- `scs.binom(n,p)` : loi binomiale de paramètres n et p
- `scs.geom(p)` : loi géométrique de paramètre p (à support $\{1, 2, \dots\}$)
- `scs.poisson(λ)` : loi de Poisson de paramètre λ
- `scs.rv_discrete(values=(xk,pk))` : loi discrète qui prend les valeurs dans `xk` (une liste d'entiers) avec probabilités respectives `pk` (une liste de la même taille de nombres positifs sommant à 1).

Lois continues

Toutes les lois continues peuvent être translatées et/ou dilatées en fournissant les arguments optionnels `loc=...` et `scale=...`.

- `scs.uniform()` : loi uniforme sur $[0, 1[$.
- `scs.norm()` : loi normale centrée réduite
- `scs.expon()` : loi exponentielle de paramètre 1.
- `scs.gamma(α)` : loi gamma de paramètres α et 1 (densité $x \mapsto \Gamma(\alpha)^{-1} x^{\alpha-1} e^{-x}$, $x \geq 0$).
- `scs.cauchy()` : loi de Cauchy (densité $x \mapsto \left(\pi(1+x^2)\right)^{-1}$).
- `scs.pareto(b)` : loi de Pareto de paramètre b (fonction de répartition $x \mapsto 1 - x^{-b}$, $x \geq 1$)

Méthodes

Une instance d'une des classes `scs.rv_discrete` ou `scs.rv_continuous` possède les méthodes suivantes :

- `rvs(s)` : renvoie un tableau de dimensions `s` (entier ou tuple d'entiers) contenant un échantillon de variables aléatoires de la loi donnée.
- `cdf(`x`)` : renvoie la valeur de la fonction de répartition en x, c'est-à-dire $P(X \leq x)$
- `sf(`x`)` : renvoie la valeur de la fonction de survie en x, c'est-à-dire $P(X > x)$
- `pmf(`k`)` (`rv_discrete` seulement) : renvoie la valeur de la fonction de masse en k, c'est-à-dire $P(X = k)$
- `pdf(`x`)` (`rv_continuous` seulement) : renvoie la valeur de la fonction de densité en x
- `ppf(`q`)` : renvoie le q-quantile, c'est-à-dire le plus petit x tel que $P(X \leq x) \geq q$ (inverse généralisé de la fonction de répartition)
- `isf(`q`)` : équivalent à `ppf(`$1-q$`)`
- `interval(`α`)` : renvoie un intervalle de confiance de niveau de confiance α ; plus précisément, l'intervalle qui contient la même masse des deux côtés de la médiane

Ces méthodes peuvent aussi recevoir comme arguments les paramètres de la loi, il faut alors faire **très attention** à l'ordre des paramètres (se référer à l'aide en ligne) ! Voici la règle pour les méthodes ci-dessus :

- Pour toutes les méthodes sauf `rvs`, l'argument doit apparaître *avant* les paramètres de la loi.
- Pour la méthode `rvs`, l'argument `s` doit être fourni par le mot-clé `shape=s`,

Par exemple, les commandes suivantes renvoient le même résultat :

```
# fonction de masse en x de la loi de Poisson de paramètre l
scs.poisson(l).pmf(x)
P = scs.poisson(l) ; P.pmf(x)
scs.poisson.pmf(x,l)

# échantillon de taille k de la loi binomiale de paramètres n et p
scs.binom(n,p).rvs(k)
B = scs.binom(n,p) ; B.rvs(k)
scs.binom.rvs(n,p,size=k)
```

Analyse statistique d'échantillons

Le module `numpy` fournit quelques fonctions pour faire de l'analyse rudimentaire d'échantillons :

- `np.mean` : moyenne empirique
- `np.var` : variance empirique
- `np.std` : écart-type empirique
- `np.cov` : matrice de covariance empirique
- `np.median` : médiane
- `np.quantile`, `np.percentile` : quantile et percentile (les commandes suivantes donnent le même résultat : `np.quantile(A,0.7)` et `np.percentile(A,70)`)
- `np.average` : moyenne pondérée
- `np.min`, `np.max` : minimum et maximum

Le module `scipy.stats` en fournit d'autres, par exemple :

- `scs.skew`, `scs.kurtosis` : coefficients d'asymétrie et d'aplatissement
- `scs.moment` : moments empiriques
- `scs.gmean` : moyenne géométrique
- `scs.tmean`, `scs.tvar`, ... : moyenne et variance empiriques tronquées