2018

Introduction à Python

Joseph Salmon
Université de Montpellier

Préface

Ce polycopié se veut être une introduction à la programmation scientifique et à la visualisation de données via le langage `Python`. Peu de notions probabilistes/statistiques seront abordées dans ce cours ; de sorte que le lecteur pourra se concentrer principalement sur l'apprentissage de `Python`.

Notons toutefois que le but de ce document est d'apprendre à programmer avec `Python` et non pas de faire des analyses statistiques ni de faire du « presse bouton ». Un conseil simple pour apprendre à coder : codez, codez et codez toujours !!! Vous pouvez aussi vous inspirer de code plus avancé de codeurs réputé pour améliorer vos pratiques.

J'ai essayé de faire en sorte que ce cours vous apprenne le plus possible en un temps raisonnable mais la manière d'apprendre à coder la plus efficace c'est de faire des erreurs, des « bugs », de les résoudre et de les comprendre afin de s'en souvenir.

Enfin si vous trouvez des coquilles (ce qui est plus que fort probable !) dans ce support de cours, j'apprécierais que vous me les fassiez connaître. Pour terminer je tiens à remercier Stéphane Boucheron de m'avoir laissé adapter son poly de statistiques descriptives sous R, au langage `Python`.

Bonne programmation.

Table des matières

Première partie

Introduction à Python

1

Python : histoire et évolution

Sommaire

Mots-Clefs: Anecdote, Python 2 vs. Python 3,

1.1 Historiques et évolutions

Surtout n'installez que `Python 3` (en particulier nous utiliserons `Python 3.6` dans la suite). Je déconseille l'utilisation de `Python` sachant que la plupart des librairies populaires ne sont maintenant plus maintenu en `Python 2`. Dans la

Joe: todo and add plot on langage evolutions...

1.2 Syntaxe

Une grosse difficulté en `Python` est que par rapport à d'autre langages qui utilisent des parenthèses ou des accolades, ici c'est l'indentation avec des espaces (4 en général) qui fait ce travail. Il faut donc être précautionneux avec la gestion des espaces, et par exemple éviter les espaces finaux (en : *trailing spaces*).

Le site suivant explique comment paramétré votre éditeur pour vous prémunir de tels espaces inutiles :https://github.com/editorconfig/editorconfig/wiki/Property-research:-Trim-trailing-spaces

— Indentations (*e.g.*, régler la touche “tab” pour que cela crée quatre espaces dans votre éditeur)
— : à la fin des lignes.

1.3 Conseils bibliographiques

Général : Skiena, *The algorithm design manual* (en anglais)

Général / Science des données : Guttag, *Introduction to Computation and Programming Using Python : With Application to Understanding Data* (en anglais)

Science des données : VanderPlas, *Python Data Science Handbook* (en anglais)

Code et style : Boswell et Foucher, *The Art of Readable Code* (en anglais)

Python : http://www.scipy-lectures.org/

Visualisation (sous R) : https://serialmentor.com/dataviz/

2

Installation et premiers pas

Sommaire

Mots-Clefs: Anecdote, Python 2 vs. Python 3,

2.1 Anaconda

Nous allons aborder la manière d'avoir une installation simple permettant d'avoir un environnement fonctionnel quelque soit le système d'exploitation. Je ne décrirais cependant que l'utilisation sous Linux (Ubuntu), les autres systèmes d'exploitation ne seront donc pas décrit, mais restent très similaires.

Anaconda (voir https://conda.io/docs/glossary.html#anaconda-glossary) est pratiquement incontournable de nos jours. C'est une distribution gratuite, libre, et optimisée pour Python qui repose sur Conda. En particulier Anaconda permet de gérer les dépendances entre les librairies `Python` de manière automatique.

Pour l'installation il faut suivre les instructions données sur la page : https://conda.io/docs/user-guide/install/index.html. Notez que la première installation peut prendre un peu de temps selon la qualité de votre connexion internet et votre machine (comptez 10 à 30mn).

Une notion importante à connaître si vous voulez avoir plusieurs installations de librairies et/ou de `Python` sur votre système est celle d'environnement. Un environnement est défini par son nom et conserve les librairies (potentiellement avec les versions précises que

l'on souhaite trouver) Un exposé détaillé est disponible ici : https://conda.io/docs/user-guide/tasks/manage-environments.html.

Pour ce cours on va par exemple utiliser un environnement spécifique apprentissage_statistique (cela permettra à toute la classe d'avoir les même versions). Il est recommandé d'exécuter les six prochaines boîtes pour mettre obtenir l'installation voulue pour ce cours. Pour cela il suffit d'utiliser la fonction create de Conda dans un terminal :

```
conda create -n apprentissage_statistique python=3.6
```

Rem: ici l'option -n indiquer l'option nom (en : *name*).

Ensuite pour utiliser votre installation Python dans le bon environnement il suffit d'appeler la commande source activate (sous Windows : activate) :

```
$ source activate apprentissage_statistique
```

On peut sortir de cet environnement en tapant source deactivate (testez le tout de suite), et il est aussi possible de lister l'ensemble de vos environnement avec la commande :

```
$ conda env list
```

On peut maintenant installer la liste des librairies dont on aura besoin :

```
$ conda install numpy=1.14.3 scipy=1.1.0 matplotlib=2.2.2
```

On pourra aussi rajouter la liste des librairies suivants :

```
conda install scikit-learn=0.19.1 pandas=0.23.4 seaborn=0.9.0
```

Pour obtenir un listing de tous vos librairies vous pouvez faire :

```
$ conda list
```

2.2 pip install

Pour les librairies plus rares, il peut être bon d'utiliser aussi la commande pip dans les cas où aucune version n'est disponible sous Anaconda.

Liens :

http://apprendre-python.com/page-pip-installer-librairies-automatiquement

https://pip.pypa.io/en/stable/installing/

2.3 Environnement de développement

En terme d'environnement de développement (ou Integrated Development Environment,IDE), Python offre de nombreuses possibilités.

TO DO: detailler IDE etc.

Pour utiliser Python vous pouvez ainsi choisir plusieurs méthodes :

jupyter notebook : c'est la manière recommandé pour ce cours, et les compte rendus de TP / projets se feront sous ce format (`.ipynb`). Cela lance une interface dans votre navigateur (Firefox, Chrome, etc.).

ipython : pour un projet plus conséquent il est recommandé d'utiliser la commande `ipython --pylab` qui lance `ipython` en mode interactif (l'option `--pylab` permet de gérer l'ouverture de fenêtres graphiques sous Linux). Cette surcouche de `Python` est utile pour permettre l'auto-complétion des noms (en tapant la touch "tab" sous Linux). Cette manière de travailler requiert d'utiliser un éditeur de texte pour écrire et sauvegarder son code. Parmis les choix possibles, on conseille par exemple : **Atom**, **Visual Studio Code**, **Sublime Text**, **Vim**, **Emacs**, etc. On manipule ici du pure python et donc des fichiers de types `.py`

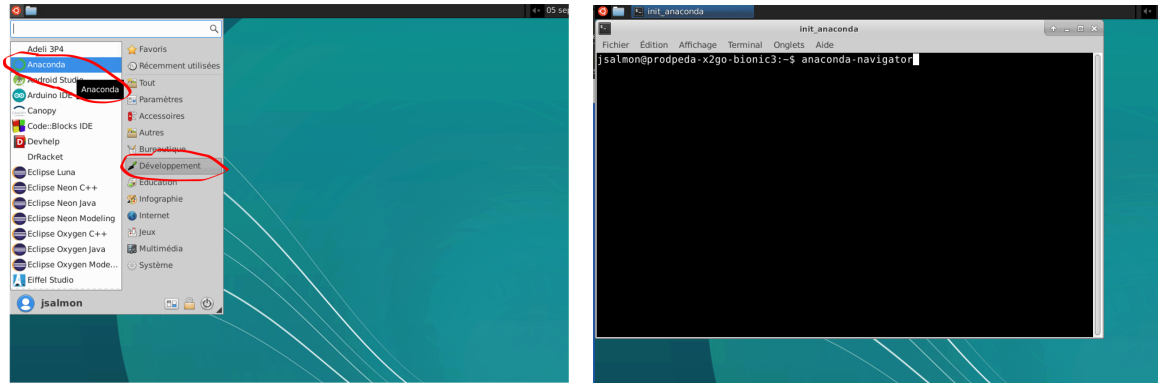
jupyter lab : c'est une IDE plus récent qui ressemble à jupyter notebook (non-testé par l'auteur).

Pycharm : IDE populaire pour coder, et lancer le code à la façon de matlab.

python : c'est la version la plus basique

2.3.1 Jupyter notebook (sur les machines de l'Université de Montpellier)

Pour lancer anaconda, il est nécessaire de l'initialiser via le raccourci graphique "Anaconda" du menu applications/Développement, comme illustrée en Fig. 2.1a. Dans le nouveau terminal ouvert, taper ensuite la commande `anaconda-navigator` comme indiquer Figure 2.1b :



(a) Lancer Anaconda (b) Lancer Anaconda Navigator

FIGURE 2.1 – Démarrage de jupyter notebook sous Linux (pour les machines en salles TP de l'Université de Montpellier uniquement)

Enfin, après quelques instants, vous pourrez lancer jupyter notebook and cliquant sur l'icône "Launch" correspondante comme indiqué en Figure 2.2.

2.4 Les librairies populaires

Voici maintenant une brève description des librairies que nous utiliserons :

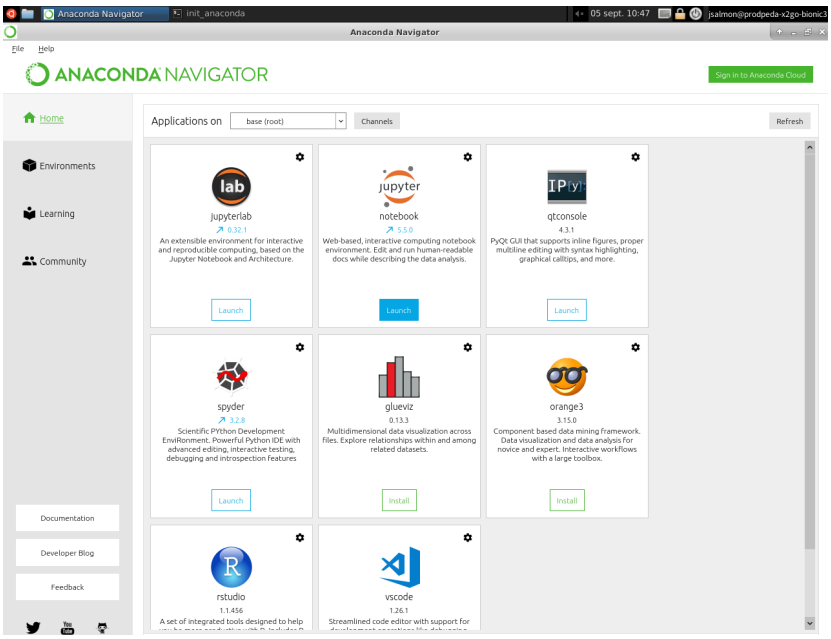


FIGURE 2.2 – Lancement de jupyter notebook

numpy : manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux. Pour les utilisateurs de Matlab ou Julia, c'est l'équivalent sous `Python`.

scipy : calcul scientifique ; contient en particulier des éléments pour l'optimisation, algèbre linéaire, l'intégration, l'interpolation, la FFT, le traitement du signal et des images,

matplotlib : affichage graphique standard (courbe, surface, histogrammes, etc.)

scikit-learn : c'est la librairie pour l'apprentissage automatique

pandas : c'est la librairie permet de manipuler des tableaux de données hétérogènes. Pour les utilisateurs de R, c'est ce qui permet de créer des `DataFrame`

seaborn : c'est une extension de `matplotlib` qui permet d'avoir des affiches graphiques plus esthétiques, et de facilement afficher des "classiques" statistiques (boîtes à moustache, estimateur de densité à noyaux, intervalles de confiances, etc.).

Exemple de chargement :

```
import numpy as np  # importe Numpy sous un nom raccourci
np.__version__  # affiche la version de Numpy
```

Quand on a besoin que d'une fonction on peut simplement charger la fonction voulue :

```
import numpy as np  # importe Numpy sous un nom raccourci
```

3

Prise en main de jupyter notebook

3.1 Lancer un jupyter notebook

Une fois la procédure d'installation terminé, on peut lancer l'application jupyter notebook, par exemple en tapant dans un terminal sous Linux :

```
jupyter notebook
```

Vous pouvez alors créer un nouveau fichier en cliquant sur "New" (ou "Nouveau") et en sélectionnant `Python 3` comme illustré en Figure 3.1

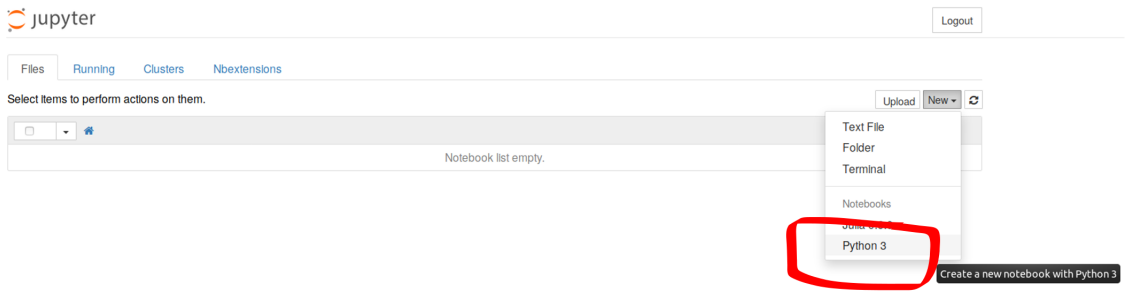


FIGURE 3.1 – Création d'un nouveau fichier `.ipynb`

3.2 Markdown et commentaires

Il est utile de découvrir la syntaxe de Markdown pour pouvoir mettre des commentaires entre les cellules de code.

On pourra se référer au site suivant pour avoir les commandes les plus courantes (titres, souligner, écrire en gras, tableaux, etc.) : https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

3.3 Visualisation et matplotlib

On peut maintenant commencer par un premier exemple pour voir que nos librairies fonctionnent correctement. Le suivant est tiré de l'aide en ligne de matplotlib : https://matplotlib.org/gallery/subplots_axes_and_figures/subplot.html

On commence par charger les librairies (toujours en début de fichier pour s'y retrouver) :

```
# Chargement des librairies
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
```

Rem: la dernière commande permet d'avoir plus d'interactions avec le graphique, notamment la possibilité de zoomer.

Ensuite on définit des quantités numériques que l'on souhaite visualiser dans une nouvelle boîte :

```
# Creations de tableau 1D avec valeurs numériques
x1 = np.linspace(0.0, 5.0, num=50)
x2 = np.linspace(0.0, 2.0, num=50)
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)
```

Enfin on passe à l'affichage graphique :

```
# Affichage graphique
plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('Time (s)')
plt.ylabel('Undamped')

plt.show()  # Pour forcer l'affichage
```

Le rendu sous jupyter notebook est donné en Figure 3.2.

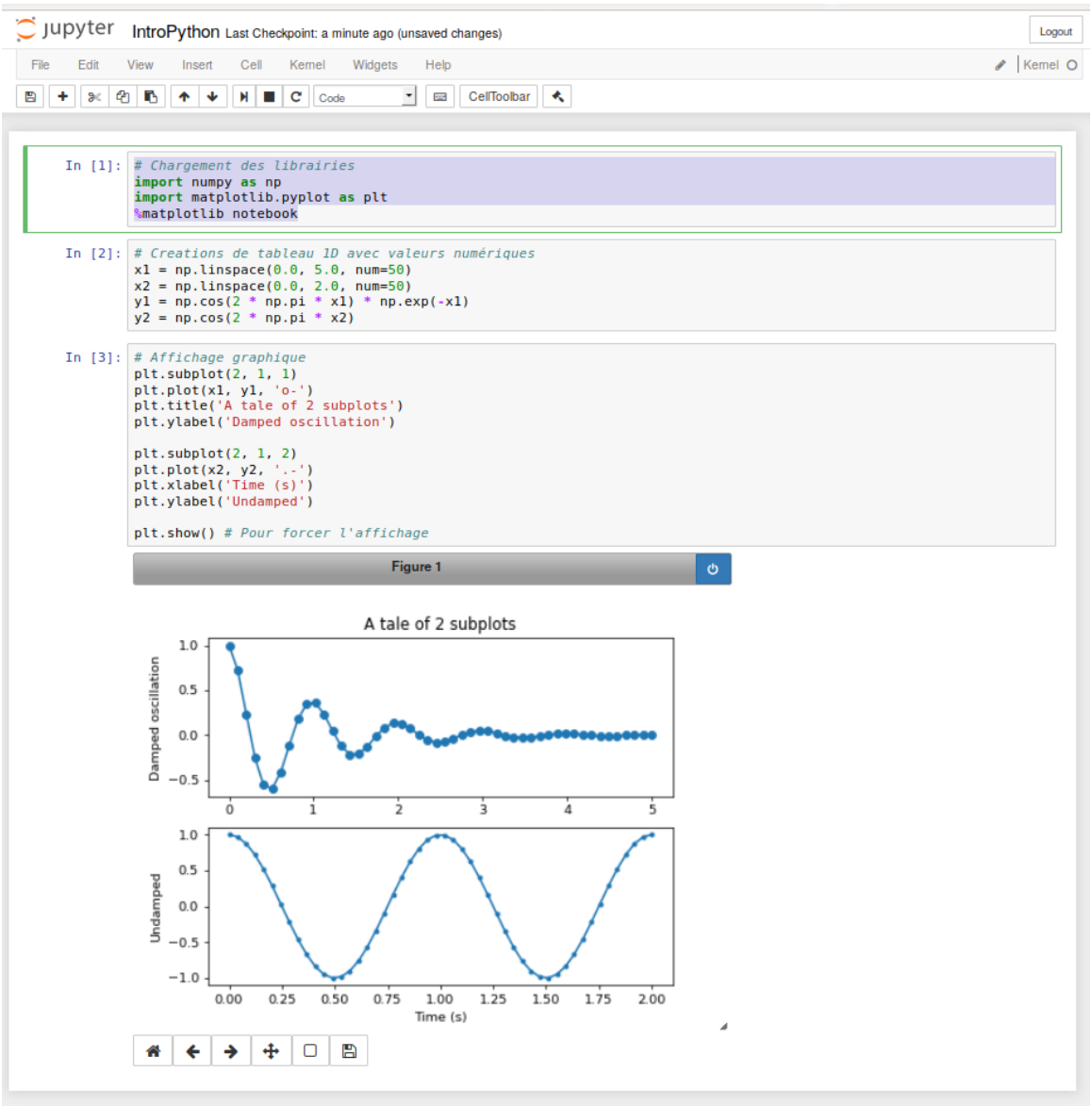


FIGURE 3.2 – Résultat du premier graphique

4

Commandes usuelles

On va commencer par introduire les commandes les plus courantes et les plus utiles en python.

La première est sans doute la commande **print** qui permet d'afficher une liste de caractères. Par exemple voici un exemple francisé du fameux "hello world !" simple :

```
print('Salut toute le monde')
```

Une version qui permet d'afficher des listes des caractères plus évoluées est la suivante

```
first_name ='Joseph'
last_name ='Salmon'
print('Prénom : {} ; nom : {}?'.format(first_name,last_name))
```

On verra comment jouer avec les types de données par la suite (par exemple avec les flottant et les entiers que l'on introduit dans les sections suivantes)

Rem: on n'abordera pas ici l'épineuse question des accents et des encodages, mais les lecteurs curieux pourront tenter de comprendre les joies de l'utf8 ici :

TO DO: utf8

4.1 Float et int

Les nombres les plus simples que l'on peut manipuler sont les entier (ou *integer* en anglais). Ce format de données est simple et dispose des opérations usuelles que vous connaissez. Ainsi on peut calculer facilement 5! de la façon suivante :

```
fact5 = 1 * 2 * 3 * 4 * 5
print(fact5)
```

Les nombres flottants (ou *float* en anglais) sont les nombres qui permettent de représenter, à précision choisie ou donnée par défaut, les nombres réels que l'on manipule en mathématique. Cependant du fait de la gestion de l'arrondi [1] pour représenter de tels nombres "l'arithmétique" associée à ce type des nombres est parfois troublantes quand on retranche des quantités petites du même ordre, ou que l'on ajoute des nombres trop grand. Une représentation classique pour écrire des nombres avec beaucoup de chiffres (avant ou après la virgule) est d'utiliser l'écriture suivante

```
print(1e-5)
print(1e10)
```

qui permet d'afficher les nombres $10^{-5} = 0.00001$ et $10^{10} = 10000000000$.

Pour comprendre l'aspect délicat des flottants, on pourra tenter de comprendre ce qui se passe quand on tape :

```
nb_small1 = 1e-41
nb_small2 = 1e-40
print(nb_small1 - nb_small2)
```

De manière toute aussi perturbante pour des grand nombres :

```
nb_big1 = 1e150
nb_big2 = 1e150
print(nb_big1 * nb_big2)
```

voir même pour des nombres plus grand on "tape" la valeur "infinie", représentée par `inf` en `Python`. Une quantité tout aussi étrange mais parfois for utile à connaître est le `nan` (en anglais *not a number*), qui représente un nombre qui n'en est pas un ! Par exemple des manières classiques d'obtenir des `nan` sont les suivantes **TO DO: remove from sublime dictionary the word suivnates TO DO: default float value**

```
inf / inf
inf - inf
```

TO DO: more on NANs in Python

Les opérations usuelles sur les entiers (**`int`**) et les nombres "flottant" (**`float`**), données dans le Tableau 4.1

Il est aussi facile de passer d'un format de nombre à l'autre : par exemple **`int`**`(3.0)` retourne l'entier 3, et **`float`**`(3)` renvoi le nombre flottant

4.2 Booléens

Les booléens des variables qui valent zéro ou un, ou de manière équivalente en `Python` **`True`** ou **`False`**

1. Notez que les nombres réels en mathématiques sont représentez par des suites infinies de chiffres, comme par exemple le fameux nombre π. Malheureusement, les ordinateurs n'ont accès qu'à une quantité finie de mémoire pour écrire ces chiffres : ils doivent donc faire des approximations pour les représenter

Commande `Python`	Interprétation
`x + y`	Somme de `x` et `y`
`x - y`	Différence de `x` et `y`
`-x`	Changement de signe de `x`
`x * y`	Produit de `x` et `y`
`x / y`	Division de `x` par `y`
`x // y`	Division entière de `x` par `y`
`x % y`	Reste de la division de `x` par `y`
`x ** y`	`x` à la puissance `y`

TABLE 4.1 – Opérations mathématiques usuelles (**`int`**/**`float`**)

4.3 Conditions et boucles

4.3.1 if then else et autres tests

Les instructions if then else, permettent de créer des tests pour savoir si un booléen (une instruction) est vrai ou fausse.

Par exemple pour tester si un nombre est paire on peut procéder comme suit :

```
nb_a = 14
if nb_a%2==0:
    print("nb_a est paire")
else:
    print("nb_a est impaire")
```

Pour des test impliquant plus de deux cas possibles, la syntaxe nécessite d'utiliser **`elif`**. Par exemple c'est le cas si l'on cherche à teste le reste d'un nombre modulo 3 :

```
if nb_a%3==0:
    print("nb_a est congru à 0 modulo 3")
elif a%3==1:
    print("nb_a est congru à 1 modulo 3")
else:
    print("nb_a est congru à 2 modulo 3")
```

4.3.2 Boucle for

La boucle for permet d'itérer des opérations sur un objet dont la taille est prédéfini. Par exemple pour afficher (avec 3 chiffre après la virgule) la racine des nombres de 0 à 10 il suffit d'écrire

```
for i in range(10):
    print("La racine du nombre {:d} est {:.3f}".format(i,i**0.5))
```

Une commande fort utile en `Python` est **`enumerate`** qui permet d'obtenir les éléments d'une liste et leur indice pour itérer sur eux.

Par exemple supposons que l'on ait créer la liste des carrées de 1 à 10 :

```
liste_carres =[] #la liste est initialement vide
for i in range(10):
    liste_carres.append(i**2)
```

Maintenant pour itérer sur cette liste et par exemple trouver la racine des nombres de cette liste on écrit

```
for i, carre in enumerate(liste_carres):
    print(int(carre**0.5)) # affichage en passant de float à int
```

4.3.3 Boucle while

La boucle **`while`** permet elle d'itérer des opérations tant qu'une condition n'est pas satisfaite. Il faut être vigilant quand on créer une telle boucle et s'assurer que celle-ci terminera. En effet il se pourrait qu'elle soit simplement infini si le critère d'arrêt n'est jamais vérifié.

```
i = 0
list_nb_premier = []
print("Nombres qui élevés à la puissance 2 sont < 1000:")
while (2**i < 1000):
    print("{0}".format(i))
    i += 1  # incrémente le compteur
```

En effet $1024 = 2^10 > 1000$.

4.4 Structure de liste

Une liste est une structure de données classique et linéaire (les éléments sont contigus en mémoire), dans laquelle il est facile d'ajouter un élément en fin en utilisant la commande

```
ma_liste =[]
for i in range(10):
    ma_liste.append(i**2)
```

4.5 Fonctions

```
def mcp_prox(x, thr, gamma=1.2):
    """MCP-proximal operator function, with gamma >1."""
    denom = (1 - 1 / gamma)
    y = np.maximum(np.abs(x) - thr, 0) / denom * np.sign(x)
    if np.abs(x) > gamma * thr:
        y = x
    return y
```

Rem: `gamma` est un paramètre. Si l'on appelle la fonction sans le renseigner, par exemple en tapant

```
mcp_prox(0.3, 2.1)
```

`Python` retourne la même valeur qu'avec l'instruction :

```
mcp_prox(0.3, 2.1, gamma=1.2)
```

à savoir 0.0.

4.6 Aide

Il est important de pouvoir obtenir de l'aide sur les fonctions standards ou sur les librairies usuelles. Bien sûr il est facile d'utiliser un moteur de recherche sur internet pour en général tomber sur des informations pertinentes. Le site Stackoverflow est lui aussi une mine d'or inestimable. Cependant pour les fonctions/commandes que l'on connaît, il est parfois beaucoup plus rapide de taper la commande `?` pour avoir de l'aide. Par exemple pour la commande **`print`**

```
print?
```

permet de connaître sa syntaxe et comment l'utiliser en `Python`.

Quand on écrit des fonctions il est particulièrement utile de renseigner l'aide d'une fonction avant de l'écrire. Ce cahier des charges permet d'accélérer l'écriture par la suite, et surtout, si l'on travaille avec quelqu'un d'autre (potentiellement quelqu'un d'autre est soi-même quelques temps plus tard !). Pour faire apparaître les éléments dans l'aide il suffit de commencer la fonction par des commentaires entre `'''` et `'''`.

Ainsi on peut voir que taper la commande :

```
mcp_prox?
```

permet d'afficher le message :

TO DO: Utiliser un autre environnement pour les docstrings

Une meilleure façon d'écrire la fonction aurait été plutôt la suivante, qui permet de connaître la signification des entrées / sorties.

```
def mcp_prox(x, thr, gamma=1.2):
    """Compute the MCP proximal operator.

       Note that gamma needs to be >1.

    Parameters
    ----------
    x : ndarray, shape (n_samples, )
        Values wher the proximal operator are evaluated

    thr : float
        The threshold parameter used, with '0 < threshold'.

    gamma : float, optional
        The slope parameter. Restrictions apply: gamma > 1!

    """
    denom = (1 - 1 / gamma)
    y = np.maximum(np.abs(x) - thr, 0) / denom * np.sign(x)
    if np.abs(x) > gamma * thr:
        y = x
    return y
```

Deuxième partie

Librairies classiques en science des données

5

Numpy, ou comment se passer de Matlab

Pour utiliser Numpy il faut charger cette librairie, ce qui est généralement fait de la manière suivante.

```
import numpy as np  # importe scikit-learn sous un nom raccourci
np.__version__  # vérifie la version de Numpy utilisée
```

Rem: L'extension `.np` est courante et sera employé dans la suite pour toute référence à Numpy.

Rem: Pour les utilisateurs de Matlab, voici une pierre de Rosette fort utile pour gérer les commandes usuelles. **TO DO: to complete**

6

Pandas, ou R sous Python

```
import pandas as pd  # importe scikit-learn sous un nom raccourci
pd.__version__   # vérifie la version de Numpy utilisée
```

7

Scikit-Learn, l'apprentissage automatique sous Python

```
import sklearn as skl  # importe scikit-learn sous un nom raccourci
skl.__version__  # vérifie la version de Numpy utilisée
```

Quelques references en apprentissage statisiques :

James et al., *An introduction to statistical learning*

Troisième partie

Éléments avancés en Python

8 Style

Grammaire /politesse :

pep8

Par exemple utiliser autopep8 notebook

exmple

Il faut écrire pour une bonne lisibilité

```
a = 8
```

et non

```
a=8
```

Aussi il est bon de couper les lignes après 80 caractères par exemple ne pas écrire

9
Classes

Un bon exemple de classes sont celle utilisés dans `sklearn` *cf.* la classe `Lasso`

10
Exceptions

11

Tests unitaires

Quatrième partie

Code et ressources extérieures

12

Versionnement : git et github

Bibliographie

Boswell, D. et T. Foucher. *The Art of Readable Code*. O'Reilly Media, Inc., 2011 (p. 7).

Guttag, J. V. *Introduction to Computation and Programming Using Python : With Application to Understanding Data*. MIT Press, 2016 (p. 7).

James, G. et al. *An introduction to statistical learning*. T. 103. Springer Texts in Statistics. With applications in R. Springer, New York, 2013, p. xiv+426 (p. 26).

Skiena, S. S. *The algorithm design manual*. T. 1. Springer Science & Business Media, 1998 (p. 7).

VanderPlas, J. *Python Data Science Handbook*. O'Reilly Media, 2016 (p. 7).