

2018

Introduction à Python et aux statistiques descriptives

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 codes plus avancés de codeurs réputés 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.

Enfin, 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

I	Introduction à Python	5
1	Python : historique et évolution	6
1.1	Historique	6
1.2	Évolution : Python 2 vs. Python 3	8
1.3	Syntaxe	8
1.4	Conseils bibliographiques	9
2	Installation et premiers pas	11
2.1	Anaconda	11
2.2	pip install	12
2.3	Environnement de développement	13
2.4	Les librairies populaires	14
3	Prise en main de jupyter notebook	17
3.1	Lancer un jupyter notebook	17
3.2	Markdown et commentaires	18
3.3	Visualisation et matplotlib	18
4	Formats et commandes usuelles	21
4.1	Chaînes de caractères et nombres	21
4.2	Booléens	26
4.3	Structure de liste	27
4.4	Structure de dictionnaire	28
4.5	Conditions et boucles	29
4.6	Fonctions	30
II	Librairies classiques en science des données	34
5	numpy, ou comment se passer de Matlab	35

5.1 numpy arrays	36
5.2 Import de fichiers en numpy	41
5.3 Export de fichiers depuis numpy	44
5.4 Slicing et masques	44
5.5 Algèbre linéaire	47
6 matplotlib, ou des graphiques en Python	48
III Introduction aux statistiques descriptives	50
7 Avertissement	51
7.1 Vocabulaire : (petite) histoire du mot statistique	52
8 Statistiques uni-variées	55
8.1 Échantillon, population	55
8.2 Types de variables	56
8.3 Résumé de l'échantillon	58
8.4 Fonction de répartition, fonction quantile	61
8.5 Variance, écart-type, etc	64
8.6 Boîte à moustaches	65
8.7 Violons	66
8.8 Propriétés de l'espérance et de la médiane	66

Première partie

Introduction à Python

1

Python : historique et évolution

Sommaire

1.1 Historique	6
1.2 Évolution : Python 2 vs. Python 3	8
1.3 Syntaxe	8
1.4 Conseils bibliographiques	9

Mots-Clefs: Anecdote, Monthly Python

1.1 Historique

Python a été pensé pour être un langage lisible, avec une volonté d'être un langage visuellement épuré. Python utilise plutôt des mots de langue anglaise là où d'autres langages se servent d'éléments de ponctuation. Il a été créé par Guido van Rossum en 1989. Le nom Python ne renvoie pas au serpent (bien que ce genre d'allusion soit souvent utilisé dans les logos de certaines librairies) mais renvoie au nom d'une troupe d'humoristes britanniques : les [Monthly Python](#).

Parmi les premiers points à retenir :

- l'extension des fichiers Python est `.py` (on verra par la suite aussi le format `.ipynb`, pour **I**nteractive **P**ython **N**ote**B**ook)
- les commentaires sont indiqués par le caractère `#`.
- les blocs sont identifiés par l'indentation (au lieu d'accolades comme en C ou C++). Une augmentation de l'indentation marque le début d'un bloc, et une réduction de l'indentation marque la fin du bloc courant.

Une grosse difficulté en Python, par rapport à d'autres langages qui utilisent des parenthèses ou des accolades pour marquer le début d'un bloc, est que Python, lui, marque le début

d'un bloc par une indentation; l'indentation est donc cruciale dans ce langage. Il faudra donc se méfier des "tab" qui ne sont pas des espaces. En particulier sur certains éditeurs de texte on veillera à paramétrer l'indentation en utilisant quatre espaces (nombre standard et obligatoire en Python).



En guise de motivation pour l'apprentissage de Python voici quelques éléments :

- Python est devenu obligatoire au programme des classes préparatoires scientifiques en France en 2013 :

“Depuis la réforme des programmes de 2013, l'informatique est présente dans les programmes de CPGE à deux niveaux. Un tronc commun à chacune des trois filières MP, PC et PSI se donne pour objectif d'apporter aux étudiants la maîtrise d'un certain nombre de concepts de base : conception d'algorithmes, choix de représentations appropriées des données, etc. à travers l'apprentissage du langage Python.”

Source : <https://info-llg.fr/>

- Python est en passe de devenir le langage informatique le plus utilisé (selon un critère biaisé : le nombre de vues dans les questions de [Stackoverflow](#)).

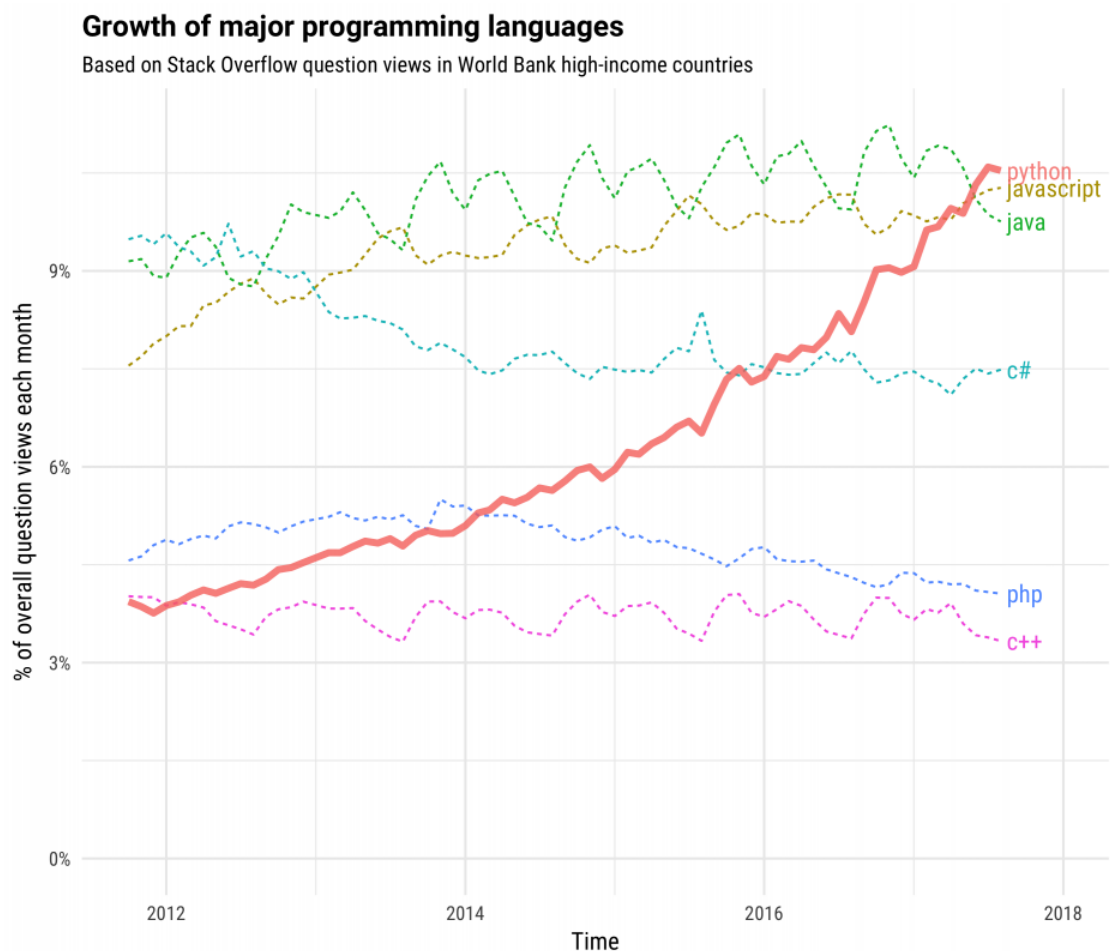


FIGURE 1.1 – Évolution de l'usage de Python

1.2 Évolution : Python 2 vs. Python 3

Surtout n'installez que Python 3 (en particulier j'utiliserai Python 3.6 dans la suite, mais Python 3.5 pourrait suffire pour beaucoup). Je déconseille l'utilisation de Python 2 sachant que la plupart des bibliothèques populaires ne sont maintenant plus maintenues en Python 2. Soyez donc vigilant à la version de Python que vous utilisez. Une manière de tester votre version (sous le logiciel que vous utilisez) est de taper :



```
from platform import python_version
print(python_version())
```

En 2008, les développeurs de Python décident de passer de la version 2 à la version 3. Ils décident d'un grand ménage dans le langage afin de clarifier le code. Le problème est que ce grand ménage rend Python 3 non rétrocompatible avec Python 2. Les millions de lignes en Python 2 doivent être réécrites afin de pouvoir être utilisées en Python 3. Après environ 10 ans de coexistence, Python 2 est en train de disparaître : de plus en plus de bibliothèques ont annoncé que les nouvelles fonctionnalités ne seront plus développées en Python 2 ce qui va forcément accélérer le processus de passage à Python 3. La **“Python software foundation”** a annoncé la fin de la maintenance de Python 2.7 pour 2020 (<https://www.python.org/dev/peps/pep-0373/>).

C'est aussi le cas pour les bibliothèques les plus populaires. Ainsi, numpy a annoncé l'arrêt de la maintenance des versions compatibles Python 2.7 à compter de 2019 :

“A major decision affecting future development concerns the schedule for dropping Python 2.7 support in the runup to 2020. The decision has been made to support 2.7 for all releases made in 2018, with the last release being designated a long term release with support for bug fixes extending through 2019. In 2019 support for 2.7 will be dropped in all new releases.”

[NumPy 1.14.0 Release Notes](#)

Un détail des différences est disponible sur le site : <http://apprendre-python.com/page-syntaxe-differente-python2-python3-python-differences> et rappelé dans les Tableaux 1.1, 1.2, 1.3 et 1.4. En particulier, parmi les points importants la gestion de la fonction `print` et des chaînes de caractères (la gestion du texte en gros) ont beaucoup changé.

Python 2	Python 3
<code>print "Bonjour"</code>	<code>print ("Bonjour")</code>
<code>print "Bonjour", variable1</code>	<code>print ("Bonjour", variable1)</code>
<code>"\n".join([x, y])</code>	<code>print(x, y, sep="\n")</code>
<code>print "une ligne ",</code>	<code>print("une ligne", end="")</code>

TABLE 1.1 – Différences Python 2 vs. Python 3 : `print`

1.3 Syntaxe

Il faut être précautionneux avec la gestion des “4” espaces qui structurent le code Python. Par exemple, il faut veiller à éviter les espaces finaux (🇬🇧 : *trailing spaces*). Ce point est facile

Python 2	Python 3
raise IOError, "file error"	raise IOError("file error")
raise "Erreur 404"	raise Exception("Erreur 404!")
raise TypeError, msg, tb	raise TypeError.with_traceback(tb)<\/pre>

TABLE 1.2 – Différences Python 2 vs. Python 3 : “Exceptions”

Python 2	Python 3
<code>_builtin__</code>	<code>builtins</code>
<code>ConfigParser</code>	<code>configparser</code>
<code>copy_reg</code>	<code>copyreg</code>
<code>cPickle</code>	<code>pickle</code>
<code>Queue</code>	<code>queue</code>
<code>repr</code>	<code>reprlib</code>
<code>SocketServer</code>	<code>socketserver</code>
<code>Tkinter</code>	<code>tkinter</code>
<code>_winreg</code>	<code>winreg</code>
<code>thread</code>	<code>_thread</code>
<code>dummy_thread</code>	<code>_dummy_thread</code>
<code>markupbase</code>	<code>_markupbase</code>

TABLE 1.3 – Différences Python 2 vs. Python 3 : “Changement nom de modules”

à traiter pour ceux qui utilisent des éditeurs de texte avancés comme Sublime Text ou VScode (à noter que pour des gros projets tous les utilisateurs devraient d’ailleurs utiliser de tels logiciels). Le site suivant explique comment paramétrer votre éditeur pour vous prémunir de tels espaces inutiles : <https://github.com/editorconfig/editorconfig/wiki/Property-research:-Trim-trailing-spaces>.

Les points les plus importants à respecter dans la syntaxe Python sont :

- bien respecter les indentations (e.g., régler la touche “tab” pour que cela crée quatre espaces dans votre éditeur si besoin)
- ne pas oublier les “deux points :” à la fin des certaines instructions (définition des fonctions, boucles, tests “**if**, **elif**, **else**”).

1.4 Conseils bibliographiques

Général : COURANT et al., *Informatique pour tous en classes préparatoires aux grandes écoles : Manuel d’algorithmique et programmation structurée avec Python*

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) et dont le site internet associé est une mine de ressources :

<https://jakevdp.github.io/PythonDataScienceHandbook/index.html>

Code et style : BOSWELL et FOUCHER, *The Art of Readable Code* (en anglais)

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

Python 2	Python 3
xrange ()	range ()
reduce ()	functools. reduce ()
intern ()	sys. intern ()
unichr ()	chr ()
basestring ()	str ()
long ()	int ()
itertools.izip ()	zip ()
itertools.imap ()	map ()
itertools.ifilter ()	filter ()
itertools.ifilterfalse ()	itertools.filterfalse ()
cookielib	http.cookiejar
Cookie	http.cookies
htmlentitydefs	html.entities
HTMLParser	html.parser
httplib	http.client
Dialog	tkinter.dialog
FileDialog	tkinter.FileDialog
ScrolledText	tkinter.scrolledtext
SimpleDialog	tkinter.simpledialog
Tix	tkinter.tix
Tkconstants	tkinter.constants
Tkdnd	tkinter.dnd
tkColorChooser	tkinter.colorchooser
tkCommonDialog	tkinter.commondialog
tkFileDialog	tkinter.filedialog
tkFont	tkinter.font
tkMessageBox	tkinter.messagebox
tkSimpleDialog	tkinter.simpledialog
robotparser	urllib.robotparser
urlparse	urllib.parse
cStringIO.StringIO ()	io.StringIO
UserString	collections.UserString
UserList	collections.UserList

TABLE 1.4 - Différences Python 2 vs. Python 3 : “Réorganisation”

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

2

Installation et premiers pas

Sommaire

2.1 Anaconda	11
2.2 pip install	12
2.3 Environnement de développement	13
2.4 Les bibliothèques populaires	14

Mots-Clefs: Anaconda, jupyter notebook, bibliothèques/packages

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. Cependant, je ne décrirai que l’utilisation sous Linux (Ubuntu), les autres systèmes d’exploitation ne seront donc pas abordés ici, mais les opérations nécessaires seront très similaires.

Anaconda (voir <https://www.anaconda.com/distribution/>) 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 bibliothèques Python de manière automatique.

Pour l’installation il faut suivre les instructions données sur la page : <https://www.anaconda.com/distribution/>. Notez que la première installation peut prendre un peu de temps selon la qualité de votre connexion internet et votre machine (comptez 10 à 30 mn grosso modo).



Une notion importante à connaître si vous voulez avoir plusieurs installations de bibliothèques et/ou de Python sur votre système est celle d’environnement. Un environnement est défini par son nom et conserve les bibliothèques (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 `intro_python` (cela permettra à toute la classe d'avoir les mêmes versions). Il est recommandé d'exécuter les six prochaines boîtes pour obtenir l'installation voulue pour ce cours. Pour cela il suffit d'utiliser la fonction `create` de Conda dans un terminal :

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

Rem: ici l'option `-n` indique l'option nom (🇬🇧 : *name*).

Ensuite pour utiliser votre installation Python dans le bon environnement il suffit d'appeler la commande `conda activate` :

```
$ conda activate intro_python
```

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

```
$ conda env list
```

On peut maintenant installer la liste des librairies dont on aura besoin. Si l'on souhaite des versions spécifiques de ces librairies (exemple : la version 1.14.3 de **numpy**), on peut forcer leur installation :

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

On pourra aussi rajouter la liste des librairies suivantes : `scikit-learn`, `seaborn`, `statsmodels`, etc.

Pour obtenir une liste de toutes 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.

C'est le cas par exemple du package `download` qui permet de gérer le téléchargement des fichiers automatiquement (sans avoir à cliquer et faire "enregistrer sous..."). Ce dernier peut être installé en lançant la ligne de commande suivante :

```
pip install download
```

dans un terminal (sous Linux et Mac OS), ou bien dans une invite de commande de type `dos` sous Windows.

Noter que sous Windows, il peut être nécessaire d'installer également le package `tqdm` (qui gère l'affichage d'une barre de progression pendant un téléchargement), ce qu'on peut faire avec la commande :



```
pip install download tqdm
```

Quand on utilise un `jupyter notebook`, on peut aussi lancer la commande “magique” suivante dans un cellule de code :


```
%pip install download
```

Liens pour aller plus loin :

<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 ( : *Integrated Development Environment, IDE*), Python offre de nombreuses possibilités. Pour utiliser Python vous pouvez ainsi choisir plusieurs méthodes :

Python : Python peut être utilisé en mode interpréteur, c’est la version la plus basique des usages. Pour démarrer l’interpréteur Python, il suffit de taper dans une console Unix (ou DOS) la commande : `python`.

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 commandes usuelles (en tapant la touche “tab” sous Linux, comme cela est possible dans un terminal classique). Cette manière de travailler requiert d’utiliser un éditeur de texte pour écrire et sauvegarder son code. Parmi les choix possibles, on conseille par exemple : **Atom**, **Visual Studio Code**, **Sublime Text**, **Vim**, **Emacs**, etc. On manipule ici du pur Python et donc des fichiers de type `.py`

jupyter notebook : c’est l’outil recommandé pour ce cours, et les comptes-rendus de TP / projets se feront sous ce format (d’extension `.ipynb`). Cela lance une interface dans votre navigateur (Firefox, Chrome, etc.). De plus ces fichiers peuvent être visualisés en ligne¹ facilement (en les mettant sur un site comme github par exemple). `jupyter notebook` est particulièrement riche en extensions, comme notamment la visualisation de la table des matières, l’automatisation de PEP8, ou **RISE** (pour créer des slides), etc. Pour installer de telles extensions, la démarche est décrite sur la page [jupyter_contrib_nbextensions](https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/). Une liste de toutes les extensions disponibles se trouve dans l’aide associée : <https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/>

jupyter lab : c’est une IDE plus récente qui ressemble à `jupyter notebook`, tout en étant plus proche de `Matlab`.

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

1. voir par exemple le **Los Angeles Times Data Desk** : <https://github.com/datadesk/notebooks>

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

Pour lancer `jupyter notebook` sur les machines de l'université, il y a deux solutions :



Lancement par ligne de commande : il est nécessaire de faire une phase d'initialisation via le raccourci graphique `anaconda3` du menu applications/Développement, comme illustrée en Fig. 2.1a. Dans le nouveau terminal ouvert, taper ensuite la commande `jupyter-notebook` comme indiqué Figure 2.1b :

Rem: sur certaines machines, il peut être nécessaire, pour le premier lancement, d'exécuter la commande : `anaconda_init3` dans ce terminal.

Lancement avec `anaconda-navigator` : Une autre solution consiste à utiliser `anaconda-navigator` après avoir lancé "Anaconda3". Dans le terminal obtenu en cliquant sur "Anaconda3" (comme dans la méthode ci-dessus), après avoir lancé la commande `anaconda-navigator` comme en Figure 2.2a, vous pourrez lancer `jupyter notebook` en cliquant sur l'icône "Launch" associée comme indiqué en Figure 2.2b.

2.4 Les bibliothèques populaires

Voici maintenant une brève description des bibliothèques que nous utiliserons :

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, l'algèbre linéaire, l'intégration, l'interpolation, la FFT, et le traitement du signal et des images,

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

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

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

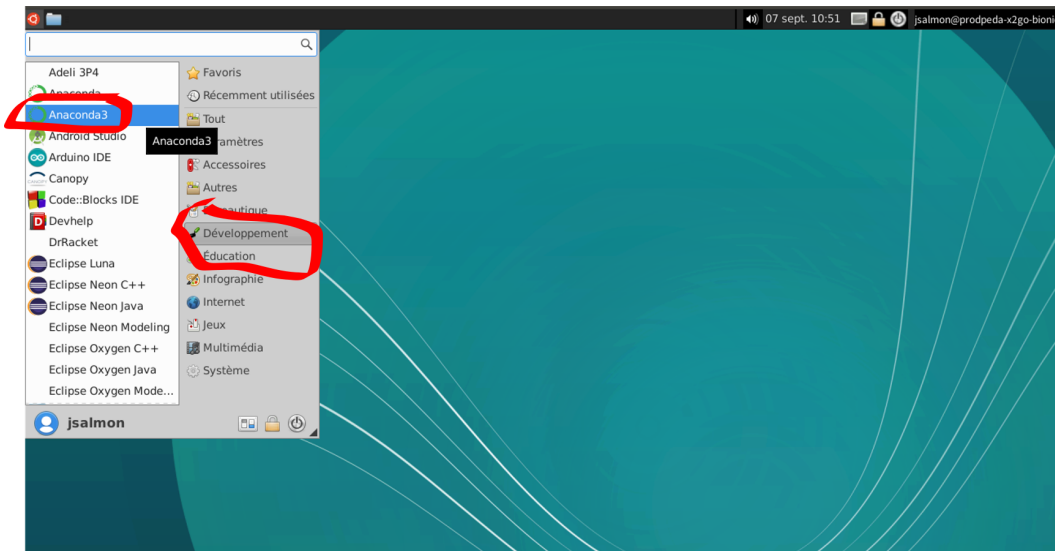
seaborn : c'est une extension de `matplotlib` qui permet d'avoir un affichage graphique plus esthétique, et de facilement afficher des visualisations statistiques classiques (boîte à moustache, estimateur de densité à noyau, intervalle de confiance, etc.).

Exemple de chargement :

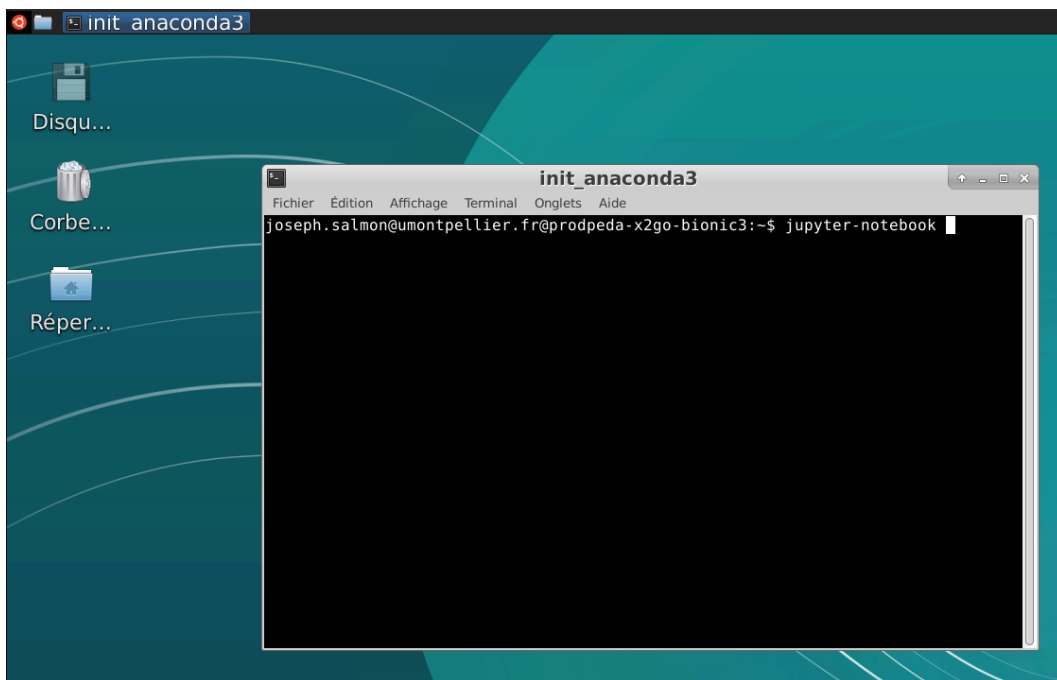
```
import numpy as np # importe Numpy sous un nom raccourci
np.__version__ # affiche la version de Numpy
```

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
```

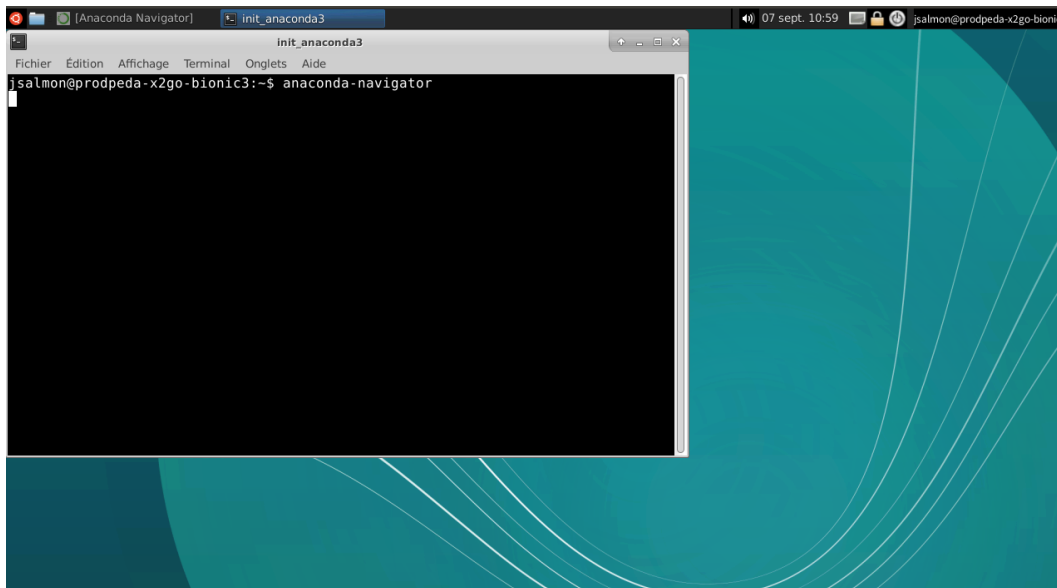


(a) Lancer Anaconda 3

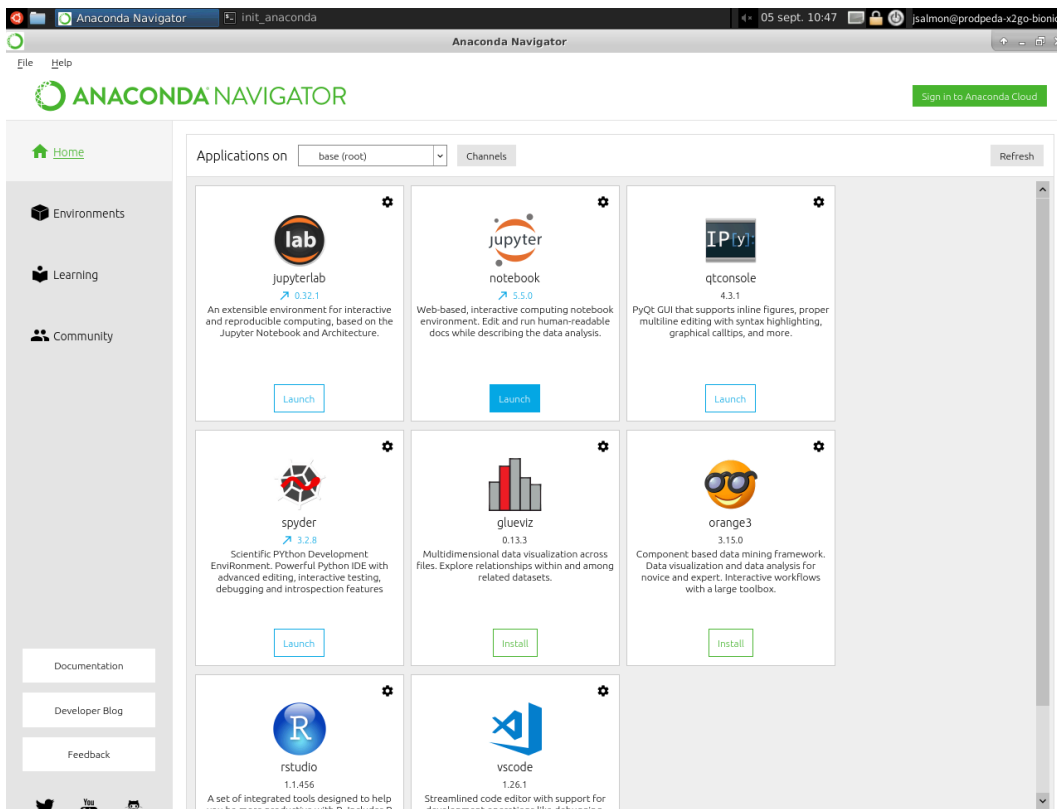


(b) Lancer jupyter-notebook

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



(a) Lancer Anaconda 3



(b) Lancer jupyter-notebook

FIGURE 2.2 - Lancement de jupyter notebook avec anaconda navigator

3

Prise en main de jupyter notebook

3.1 Lancer un jupyter notebook

Une fois la procédure d'installation terminée, 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

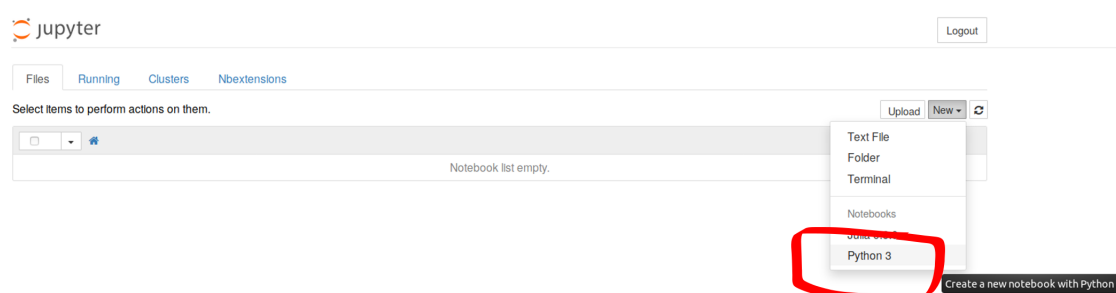


FIGURE 3.1 - Création d'un nouveau fichier .ipynb

L'interface principale est sous forme de cellule. Pour en créer de nouvelles il suffit de cliquer sur l'icône “+” en haut de l'éditeur.

Rem: Il peut être utile d'apprendre les raccourcis (🇬🇧 : *shortcuts*) pour naviguer, ajouter des cellules, changer de format, ... :

<https://www.dataquest.io/blog/jupyter-notebook-tips-tricks-shortcuts/>

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: le symbole # permet d'afficher des commentaires en lignes.

Rem: la commande %matplotlib notebook permet d'avoir plus d'interactions avec les graphiques en jupyter notebook, notamment la possibilité de zoomer sur les figures.

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)
```

```
# 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
```

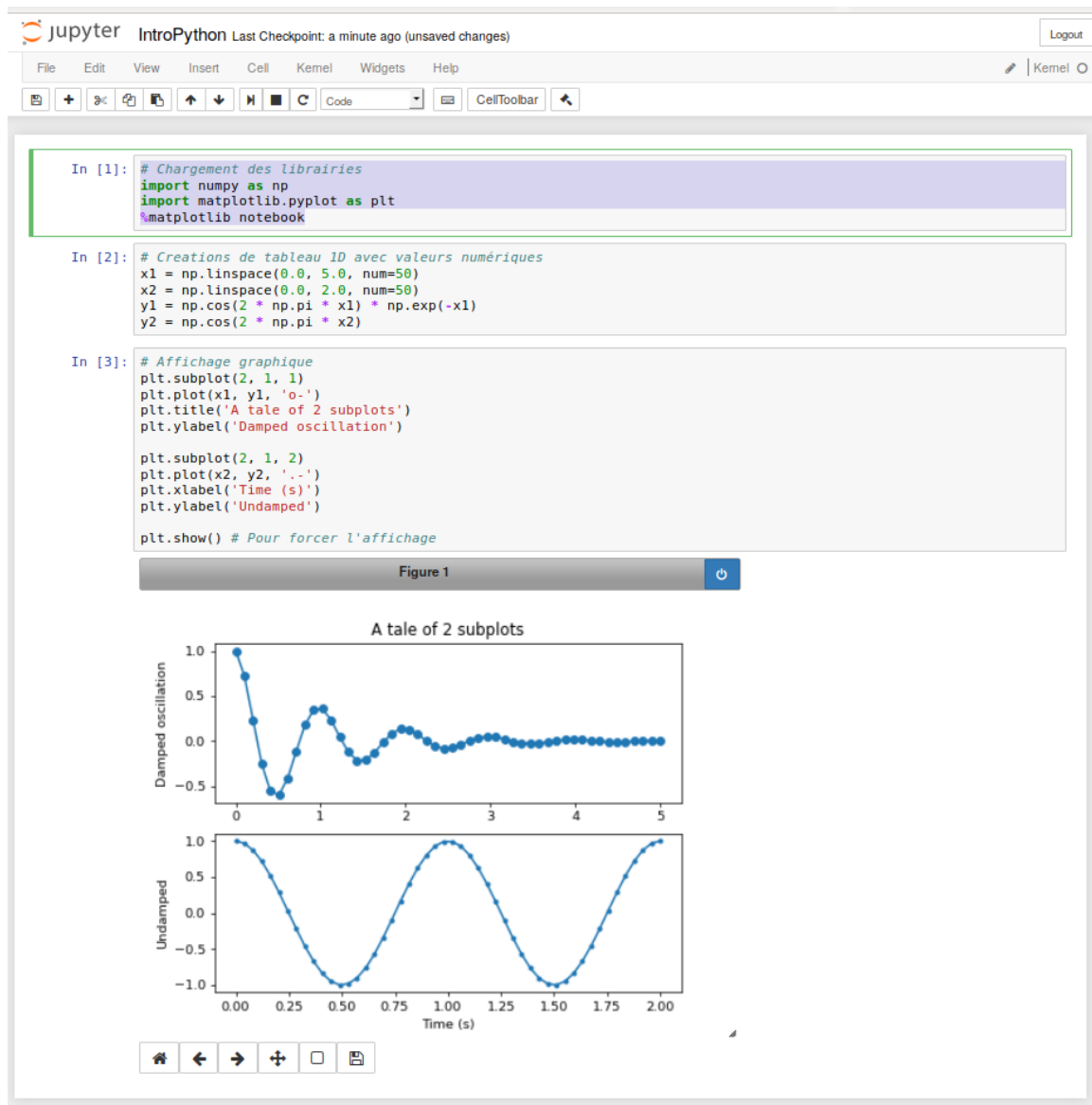


FIGURE 3.2 – Résultat du premier graphique

Le rendu sous jupyter notebook est donné en Figure 3.2. On reviendra sur ces fonctionnalités d'affichage plus en détail au Chapitre 6.

3.3.1 Extensions

Une des forces de jupyter notebook est qu'il existe un grand nombre d'extensions. Parmi les plus utiles on retiendra particulièrement

- `toc2` : visualisation de la table des matières
<https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/toc2/>
- `autopep8` : automatisation de PEP8
<https://github.com/kenko000/jupyter-autopep8>

- RISE : création de diapositives

<https://github.com/damianavila/RISE> etc.

Pour installer de telles extensions, la démarche est décrite sur la page [jupyter_contrib_nbextensions](#). Enfin, une liste de toutes les extensions disponibles se trouve dans l'aide associée : <https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/>

4

Formats et commandes usuelles

Sommaire

4.1 Chaînes de caractères et nombres	21
4.2 Booléens	26
4.3 Structure de liste	27
4.4 Structure de dictionnaire	28
4.5 Conditions et boucles	29
4.6 Fonctions	30

Mots-Clefs: print, help, int, float

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. Pour commencer voici un exemple francisé du fameux "hello world!" :

```
>>> print('Salut tout le monde')
Salut tout le monde
```

Cette manière d'afficher est donc une façon simple d'obtenir une information dans la console pour connaître l'état d'une variable.

4.1 Chaînes de caractères et nombres

4.1.1 Chaînes de caractères

Les chaînes de caractères (🇬🇧 : *strings*), qui permettent de représenter du texte (que ce soit en français, en anglais ou dans une autre langue), peuvent être exprimées de plusieurs

façons. Elles peuvent être entourées par des apostrophes ou des guillemets.

Ainsi les deux chaînes de caractères suivantes :

```
>>> salutation_apostrophe = 'Salut tout le monde'
>>> salutation_guillemet = "Salut tout le monde"
```

sont en fait identiques ¹.

Une version de **print** qui permet d'afficher des listes des caractères plus évoluées et contenant des paramètres 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 flottants et les entiers que l'on introduit dans les sections suivantes)

Rem: pour connaître la taille (en anglais :*length*) il suffit d'utiliser **len**

```
>>> len(first_name), len(last_name)
(6, 6)
```

Quelques caractères spéciaux méritent d'être connus, comme l'apostrophe, le retour à la ligne ou encore la tabulation :

```
>>> print('l\'apostrophe dans une chaîne')
>>> print('Hacque ,\n post \n postridie')
>>> print('\t \t descriptione, post \npostridie')
l'apostrophe dans une chaîne
Hacque ,
 post
 postridie
      descriptione, post
postridie
```

Parmi les opérations utiles sur les chaînes de caractères, on notera la **concaténation** :

```
>>> 'Et un,' + ' et deux,' + ' et trois zéros'
'Et un, et deux, et trois zéros'
```

la **répétition** :

```
>>> 'Et un,' * 8
```

et l'extraction :

1. on pourra tester ce fait en utilisant la commande: `salutation_apostrophe == salutation_guillemet`, ou plus naïvement, en visualisant les deux chaînes

```
>>> chef_gaulois = 'Abraracourcix'
>>> print(chef_gaulois[0], chef_gaulois[-1])
>>> print(chef_gaulois[4:], chef_gaulois[:4])
>>> print(chef_gaulois[2:5])
>>> print(chef_gaulois[::2])
>>> print(chef_gaulois[::-1])
A x
racourcix Abra
rar
Arrcucx
xicruocararbA
```

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 :

- <http://sdz.tdct.org/sdz/comprendre-les-encodages.html>
- <http://sametmax.com/lencoding-en-python-une-bonne-fois-pour-toute/>

Pour terminer la section, voici deux propriétés importantes des chaînes de caractères

Polymorphisme : il faut noter que le signe plus '+' signifie différentes choses pour différents objets : l'addition pour les entiers et la concaténation pour les chaînes de caractères. C'est une propriété générale de Python que l'on appelle polymorphisme, la nature des opérations dépend des objets sur lesquels elles agissent.

Immuabilité : Les chaînes sont immuables en Python - elles ne peuvent pas être modifiées sur place (en anglais *in place*) après leur création. Par exemple, on ne peut pas modifier une chaîne en affectant une nouvelle valeur à l'une de ses entrées, mais vous pouvez toujours en créer une nouvelle et l'assigner au même nom. L'immuabilité peut être utilisée pour assurer que votre objectif reste constant tout au long de votre programme

4.1.2 Les entiers (en anglais : *int*)

Les nombres les plus simples que l'on peut manipuler sont les entiers (ou *integer* en anglais). Ce format de donnée 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)
120
>>> type(fact5)
int
```

Attention tout de même car la division est particulière : la division entière est en fait obtenue par // et le reste par %. Ainsi,

```
>>> 7 // 3
2
>>> 7 % 3
1
```

mais en revanche :


```
>>> 7 / 3
2.3333333333333335
```

Une manière de comprendre cela est que

```
>>> type(7 / 3)
float
>>> type(7 // 3)
int
```

Cela vient de ce que Python a changé le type des données en flottants (que nous allons voir par la suite) pour faire la division requise. Vigilance donc sur ce type d'opérations.

4.1.3 Les nombres flottants

Les nombres flottants ( : *float*) sont les nombres qui permettent de représenter, à précision choisie (ou donnée par défaut), les nombres réels que l'on manipule en mathématiques. Cependant du fait de la gestion de l'arrondi² pour représenter de tels nombres, "l'arithmétique" associée à ce type de nombre est parfois troublante quand on retranche des quantités petites du même ordre, ou que l'on ajoute des nombres trop grands. Une représentation classique pour écrire des nombres avec beaucoup de chiffres (avant ou après la virgule) est d'utiliser l'écriture suivante :

```
>>> print(1e-3)
0.001
>>> print(1e10)
10000000000.0
```

qui permet d'afficher les nombres en format scientifique (donc sous format float).



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
>>> nb_small1 - nb_small2
-8.999999999999999e-41
```

De manière tout aussi perturbante pour des grands nombres :

2. Notez que les nombres réels en mathématiques sont représentés 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


```
>>> nb_big1 = 1e150
>>> nb_big2 = 1e150
>>> nb_big1 * nb_big2
9.999999999999999e+299
```

voire même pour des nombres plus grands on “touche” la valeur “infinie”, représentée par `inf` en Python. Une quantité tout aussi étrange mais parfois fort utile à connaître est le `nan` (🇬🇧 : *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 :

```
>>> too_big = nb_big1 * nb_big2 ** 2
>>> print(too_big)
inf
>>> too_big / too_big
nan
>>> too_big - too_big
nan
```

Les opérations usuelles sur les entiers (**int**) et les nombres “flottants” (**float**) sont données dans le Tableau 4.1.

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

TABLE 4.1 - Opérations mathématiques usuelles (**int/float**)

Il est aussi facile de passer d'un format de nombre à l'autre :

```
>>> int(3.0), type(int(3.0))
(3, int)
```

Passer de **int** à **float**

```
>>> float(3), type(float(3))
(3.0, float)
```

Passer de **float** à **str**

```
>>> str(3.), type(str(3.))
('3.0', str)
```

Passer de **str** à **float**

```
>>> float('3.'), type(float('3.'))
(3.0, float)
```

4.2 Booléens

Les booléens³ sont des variables qui valent zéro ou un, ou de manière équivalente en Python **True** (vrai) ou **False** (faux).

L'opération la plus commune pour créer ce genre de variable est un test :

```
>>> a = 3
>>> b = 4
>>> print(a == 4)
False
```

et par contre :

```
>>> print(a < b)
True
```

On notera que par contre

```
>>> petit_nb = 0.1 + 0.2 - 0.3
>>> mon_test = petit_nb == 0
>>> print(mon_test)
False
```

Ceci est normal. Si l'on veut tester le fait que `petit_nb` est proche de zéro il faut en fait tester que le nombre est "très petit".

Cela peut être fait avec un test de la librairie `math` :

```
>>> import math
>>> print(petit_nb)
5.551115123125783e-17
>>> print(math.isclose(0., petit_nb, abs_tol=1e-5))
True
>>> print(math.isclose(0., petit_nb, abs_tol=1e-17))
False
```

Comme en logique on a les opérations

- "ou" (en anglais : `or`), commande **or**
- "et" (en anglais : `and`), commande **and**
- "non" (en anglais : `not`), commande **not**
- "ou exclusif" (en anglais : `xor`), commande **!=**

3. le nom est associé à [George Boole](#) (1815-1864)

TABLE 4.2 - **and**

Expression	Résultat
True and True	True
True and False	False
False and True	False
False and False	False

TABLE 4.3 - **or**

Expression	Résultat
True or True	True
True or False	True
False or True	True
False or False	False

TABLE 4.4 - **xor**

Expression	Résultat
True != True	False
True != False	True
False != True	True
False != False	False

TABLE 4.5 - Table logique de **not**

Expression	Résultat
True	False
False	True

4.3 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)
```

Les listes (en anglais *list*) peuvent être créées en écrivant une suite de valeurs séparées par des virgules, le tout entre crochets.

```
>>> ma_liste = ["bras", "jambes", 10, 12]
>>> ma_liste
['bras', 'jambes', 10, 12]
```

Rem: les items d'une liste peuvent être de types différents

Comme les chaînes de caractères :

- le premier item d'une liste est à l'indice 0
- les listes supportent le *slicing*
- les listes supportent la concaténation

Ainsi,

```
>>> a[0:2] = [] # Enlève quelques items
>>> a
[10, 12]
>>> a[1:1] = ['main', 'coude'] # Insertion
[10, 'main', 'coude', 12]
>>> a[:] = [] # Vider la liste
>>> a, len(a), type(a)
([], 0, list)
```

4.4 Structure de dictionnaire

La notion de dictionnaire est une structure dont les indices (clefs) peuvent avoir un sens (valeur) et fonctionnent par paires :

```
>>> dico = {}
>>> dico['car'] = 'voiture'
>>> dico['plane'] = 'avion'
>>> dico['bus'] = 'bus'
>>> print(dico)
{'car': 'voiture', 'plane': 'avion', 'bus': 'bus'}
```

On peut obtenir les clefs (anglais *key*) et valeurs (anglais *value*) du dictionnaire assez simplement :

```
>>> print(dico.keys())
>>> print(dico.values())
```

On peut aussi créer un dictionnaire en “zipant”

```
>>> dico_2 = dict(zip(['car', 'plane'], ['voiture', 'avion']))
>>> print(dico_2)
{'car': 'voiture', 'plane': 'avion'}
```

4.4.1 Opérations sur les dictionnaires

Pour effacer un couple clef / valeur il faut procéder comme suit :

```
>>> del(dico['bus'])
>>> dico
{'car': 'voiture', 'plane': 'avion'}
```

Attention : les opérations comme la concaténation et l’extraction n’ont pas de sens pour un dictionnaire :

```
>>> print(dico[1:2])
TypeError: unhashable type: 'slice'
>>> print(dico + dico)
unsupported operand type(s) for + : 'dict' and 'dict'
```

Enfin un dictionnaire peut aussi mélanger plusieurs types de valeurs :

```
>>> dico_3 = {'nom': {'first': 'Alex', 'last': 'PetitChateau'},
...          'emploi': ['enseignant', 'chercheur'],
...          'age': 36}
```

Pour aller plus loin : https://fr.wikibooks.org/wiki/Programmation_Python/Listes

4.5 Conditions et boucles

4.5.1 `if ... : ... else: ...` et autres tests

Les instructions `if ... : ... else:` permettent de créer des tests : ainsi, en fonction de la valeur du test, on renvoie un booléen qui a comme valeur “vrai” (**True**) ou “faux” (**False**). Par exemple pour tester si un nombre est pair, on peut procéder comme suit :

```
nb_a = 14
if nb_a%2==0:
    print("nb_a est pair")
else:
    print("nb_a est impair")
```

Pour des tests impliquant plus de deux cas possibles (cas non binaires), la syntaxe nécessite d'utiliser `elif`. Par exemple c'est le cas si l'on cherche à faire des tests sur 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")
```

4.5.2 Boucle `for`

La boucle `for` permet d'itérer des opérations sur un objet dont la taille est prédéfinie. Par exemple pour afficher (avec 3 chiffres après la virgule) la racine des nombres de 0 à 9 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 créé la liste des carrés de 1 à 10 :

```
liste_carres =[] # la liste est initialement vide
for i in range(1, 11):
    liste_carres.append(i**2)
```

On peut alors itérer sur cette liste pour afficher ses éléments :

```
for i, carre in enumerate(liste_carres):
    print(carre) # affichage des carrés
```

Python inclut aussi une opération avancée connue sous le nom de compréhension de liste et qui utilise les boucles **for** :

```
>>> a = [1, 4, 2, 7, 1, 9, 0, 3, 4, 6, 6, 6, 8, 3]
>>> [x for x in a if x > 5]
[7, 9, 6, 6, 6, 8]
```

4.5.3 Boucle **while**



La boucle **while** permet d'itérer des opérations tant qu'une condition n'est pas satisfaite. Il faut être vigilant quand on crée une telle boucle et s'assurer que celle-ci se terminera. En effet, il se pourrait qu'elle soit infinie si le critère d'arrêt n'est jamais vérifié (ce qui peut parfois "bloquer" votre machine, en particulier si l'espace mémoire utilisé augmente rapidement) :

```
>>> i = 0
... print("Nb x t.q. 2^x < 10:")
... while (2**i < 10):
...     print("{0}".format(i))
...     i += 1 # incrémente le compteur
Nb x t.q. 2^x < 10:
0
1
2
3
```

Pour comprendre le déroulement des opérations précédentes, on rappelle que $16 = 2^4 > 10$ mais $8 = 2^3 < 10$.

4.6 Fonctions

Une fonction en Python est définie avec le mot clé **def**, suivi par le nom de la fonction, la signature entre parenthèses (), et un symbole ":" en fin de ligne. Il faut aussi bien indenter de quatre espaces à partir de la seconde ligne. Voici un premier exemple :

```
>>> def fonction1():
...     print("mon test")
```

Pour la lancer il suffit alors de taper la commande suivante :

```
>>> fonction1()
mon test
```

Quand on écrit des fonctions il est primordial de renseigner l'aide d'une fonction avant même de l'écrire⁴. Ce cahier des charges permet d'accélérer l'écriture par la suite, et

4. un cran plus loin il devient même utile de créer un test de la fonction avant même de terminer d'écrire la

surtout, si l'on travaille avec quelqu'un d'autre (potentiellement ce "quelqu'un d'autre" est soi-même, quelques jours ou quelques mois plus tard!). Pour faire apparaître les éléments de la documentation (en anglais *docstring*) il suffit de commencer la fonction par des commentaires entre `"""` et `"""` :

```
>>> def fonction2(s):
...     """
...     Affichage d'une chaîne et de sa longueur
...     """
...     print("{} est de longueur : {}".format(s) + str(len(s)))
```

On obtient alors pour une chaîne voulue l'affichage suivant :

```
>>> fonction2("Cette chaîne de test")
Cette chaîne de test est de longueur : 20
```

L'intérêt de l'aide est qu'on peut facilement y avoir accès avec la commande `help` ou la commande `?` :

```
>>> fonction2?
Signature: fonction2(s)
Docstring: Affichage d'une chaîne et de sa longueur
...

```

Il est important de connaître cette fonction qui marche de la même manière pour les fonctions standard de Python :

```
>>> print?
```

Une variante est d'utiliser aussi la syntaxe `help(print)`.

On peut définir une sortie d'une fonction avec le mot clef `return` :

```
>>> def square(x):
...     """
...     Retourne le carré de x.
...     """
...     return x ** 2
```

La fonction ci-dessus peut être appliquée à des nombres de type `int` ou de type `float` : on n'a pas préjugé du type d'entrée/sortie dans cette fonction :

```
>>> square(3), square(0.4)
(9, 0.16000000000000003)
```

Il est aussi possible de retourner plusieurs valeurs :

fonction que l'on souhaite

```
>>> def powers(x):
...     """
...     Retourne les premières puissances de x.
...     """
...     return x ** 2, x ** 3, x ** 4
```

et l'appel de la fonction renvoie alors un triplet :

```
>>> out = powers(3)
>>> print(out)
>>> print(type(out))
>>> x2, x3, x4 = powers(3)
>>> print(x2, x3)
(9, 27, 81)
<class 'tuple'>
9 27
```

Enfin on peut aussi mettre des arguments optionnels qui, s'ils ne sont pas donnés en appel de la fonction, prennent la valeur par défaut :

```
>>> def ma_puissance(x, exposant=3, verbose=True):
...     """
...     Fonction calculant une puissance
...
...     Paramètres
...     -----
...     x : float,
...         Valeur du nombre dont on calcule la puissance
...
...     exposant : float, default 3
...         Paramètre de l'exposant choisi
...
...     verbose : bool, default False
...         Paramètre d'affichage
...
...     Retours:
...     -----
...     Retourne x élevé à la puissance exposant (default=3)
...     """
...     if verbose is True:
...         print('version verbeuse')
...     return x ** exposant
```

On peut appeler la fonction avec ou sans certains arguments optionnels :

```
>>> ma_puissance(5), ma_puissance(5, verbose=False, exposant=2)
version verbeuse
(125, 25)
```


Il est bon de noter que l'ordre de saisie des paramètres n'a pas d'importance en Python (ce n'est pas vrai pour les autres arguments).

Un dernier point concernant les fonctions : il n'est pas obligatoire de renvoyer un argument de sortie avec le mot clef **return**. On peut en effet définir des fonctions qui modifient des objets existants. Cela peut être utile pour de l'affichage graphique où l'on met à jour un objet "figure" sans en recréer un.

Deuxième partie

Librairies classiques en science des données

5

numpy, ou comment se passer de Matlab

Sommaire

5.1 numpy arrays	36
5.2 Import de fichiers en numpy	41
5.3 Export de fichiers depuis numpy	44
5.4 Slicing et masques	44
5.5 Algèbre linéaire	47

Le module **numpy** est utilisé dans presque tous les projets de calcul numérique sous Python. Il fournit des structures de données performantes pour la manipulation de vecteurs, matrices et tenseurs ¹.

La librairie **numpy** est écrite en C et en Fortran d'où ses performances élevées lorsque les calculs sont vectorisés (ce qui signifie des calculs formulés comme des opérations sur des vecteurs/matrices).

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 employée dans la suite pour toute référence à **numpy**.

Rem: La syntaxe de **numpy** est très proche de celle de Matlab (bien qu'en plus verbeuse). Pour les utilisateurs de Matlab, voici une "pierre de Rosette" fort utile pour passer des commandes d'un langage vers l'autre <https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html>.

1. les tenseurs généralisent les matrices : un vecteur est un tenseur de dimension un, une matrice de dimension deux, mais on peut manipuler des tenseurs de dimension supérieure ou égale à 3.

5.1 **numpy arrays**

Dans la terminologie **numpy**, vecteurs, matrices et autres tenseurs sont appelés **arrays** (en français : “tableaux”).

Pour la création d'**array numpy** il y a plusieurs possibilités :

- à partir de listes ou n-uplets Python
- en utilisant des fonctions dédiées, telles que **arange**, **linspace**, etc.
- par chargement à partir de fichiers

On va maintenant détailler toutes ces manières de procéder.

5.1.1 Passage d'**array** à **list**

Commençons par créer un vecteur en partant d'une liste :

```
>>> # Vecteur: l'argument est une liste Python
>>> liste = [1, 3, 2, 4]
>>> vecteur = np.array(liste)
>>> print(vecteur)
[1 3 2 4]
```

Dans le cas d'une matrice, on procède de manière similaire :

```
>>> # Matrice: l'argument est une liste emboîtée
>>> matrice = np.array([[1, 2], [3, 4]])
>>> print(matrice)
[[1 2]
 [3 4]]
1
2
```

Les types des vecteurs et des matrices sont obtenus simplement :

```
>>> type(vecteur), type(matrice)
(numpy.ndarray, numpy.ndarray)
```

et sont tous les deux des **arrays**. Pour connaître les dimensions de ces objets il faut utiliser l'argument **shape** :

```
>>> np.shape(vecteur), np.shape(matrice)
((4,), (2, 2))
```

On peut aussi forcer les types de données dans un **array** :

```
>>> matrice_cpx = np.array([[1, 2], [3, 4]], dtype=complex)
>>> matrice_cpx
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Les autres types possibles reconnus par l'argument optionnel `dtype` sont `int`, `float`, `complex`, `bool`, etc. On peut aussi spécifier la précision en bits : `int64`, `int16`, `float128`, `complex128`. Pour en savoir plus sur ces types on peut se reporter aux pages :

- <https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html>
- <https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html>

5.1.2 Génération d'array par fonction de génération

Il y a diverses manières de créer des `arrays` en `numpy`, notamment à partir de fonctions dédiées comme on va le voir dans le paragraphe suivant.

arange, linspace, logspace : Une commande standard pour créer des tableaux simples est `arange`. Cette commande crée des valeurs de `float` ou d'`int` consécutifs dans un `array`.

Par exemple pour la création d'`array` de type `int` on peut écrire :

```
>>> x = np.arange(0, 10, 2) # arguments: start, stop, step
>>> x
array([0, 2, 4, 6, 8])
```

De manière similaire, pour créer des tableaux avec des flottants :

```
>>> y = np.arange(-1, 1, 0.5)
>>> y
array([-1. , -0.5,  0. ,  0.5])
```

D'autres manières existent pour faire des `arrays` avec des valeurs équiréparties (en échelle linéaire ou logarithmique). La première est la fonction `linspace` :

```
>>> np.linspace(0, 5, 11) # début et fin sont inclus
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

La fonction `logspace` est similaire mais travaille avec une échelle logarithmique :

```
>>> np.set_printoptions(precision=2) # for display
>>> np.logspace(0, 11, 10) # début et fin sont inclus
array([1.00e+00, 1.67e+01, 2.78e+02, 4.64e+03, 7.74e+04, 1.29e+06,
       2.15e+07, 3.59e+08, 5.99e+09, 1.00e+11])
```

Ainsi le premier argument de la fonction, 0, renvoie au nombre 10^0 ; le second argument

renvoie au nombre 10^{11} . Enfin le nombre 10 permet d'afficher 10 nombres entre ces deux bornes, qui sont en progression géométrique. La terminologie est "log" car cela revient à passer au \log_{10} (log en base 10), échantillonner de manière uniforme, puis repasser dans le domaine original en élevant à la puissance 10. Ainsi on peut retrouver de manière équivalente la syntaxe précédente en tapant :

```
>>> 10**(np.linspace(np.log10(10), np.log10(10**11), 10))
array([1.00e+00, 1.67e+01, 2.78e+02, 4.64e+03, 7.74e+04, 1.29e+06,
       2.15e+07, 3.59e+08, 5.99e+09, 1.00e+11])
```

Ce genre d'échelle logarithmique est utile pour faire de l'affichage dans certains contextes applicatifs (afficher la vitesse de convergence d'algorithmes d'optimisation, étude des prix en économie, etc.).

Il existe aussi des fonctions qui créent des matrices classiques : Par exemple, pour créer une matrice diagonale, il suffit de partir de la liste des éléments diagonaux en utilisant la fonction **diag** :

```
>>> np.diag([1, 2, 3], k=0)
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

La fonction **diag** peut aussi agir sur un **array** et extraire sa diagonale sous forme de vecteur :

```
>>> np.diag(np.diag([1, 2, 3], k=0))
array([1, 2, 3])
```

On peut aussi se servir de cette fonction pour créer des matrices dont on remplit les sur/sous-diagonales. Notez ainsi l'usage suivant :

```
>>> np.diag([1, 2, 3], k=1)
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

Pour obtenir la **transposition** d'une matrice, il suffit d'appliquer l'argument **.T** à la matrice :

```
>>> np.diag([1, 2, 3], k=1).T
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0]])
```

La fonction **transpose** permet la même opération :

```
>>> np.transpose(np.diag([1, 2, 3], k=1))
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0]])
```

Des constructions classiques permettent d'obtenir des matrices nulles de tailles prédéfinies avec **zeros** :

```
>>> np.zeros((3,), dtype=int)
array([0, 0, 0])
```

ou encore :

```
>>> print(np.zeros((3,2), dtype=float))
>>> print(np.zeros((1, 3), dtype=float))
>>> print(np.zeros((3, 1), dtype=float))
[[0. 0.]
 [0. 0.]
 [0. 0.]]
[[0. 0. 0.]]
[[0.]
 [0.]
 [0.]]
```

Une fonction similaire, **ones** permet de créer des matrices pleines de 1 :

```
>>> np.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

La fonction **full** englobe en fait les deux précédentes et permet de remplir un **array** avec n'importe quel nombre :

```
>>> np.full((3, 5), 3.14)
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

Pour créer la matrice identité il faut utiliser **eye**, fort utile en algèbre linéaire :

```
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Génération aléatoire En statistique et en probabilité il est important d’avoir un générateur de nombres aléatoires. On ne rentrera pas dans le détail ici : en effet, un ordinateur ne peut créer que des nombres dits “pseudo-aléatoires”.

- Pour plus de détails sur la manière dont Python gère la création de nombres pseudo-aléatoires, on peut consulter : <https://docs.python.org/3/library/random.html>.
- Pour plus d’informations sur le type d’algorithme utilisé (par défaut l’algorithme de Mersenne Twister), on pourra se référer à https://fr.wikipedia.org/wiki/Mersenne_Twister.

Par défaut, comme pour la plupart des langages ou logiciels, le langage Python possède un générateur de nombres (pseudo-)aléatoires “uniformes”² sur $[0, 1]$.

Pour obtenir de tels nombres on peut soit utiliser la librairie `import random` (ce qu’on ne fera pas ici), soit utiliser le module `random` de **numpy**, ce qu’on fera dans la suite :

```
>>> np.random.rand(5, 5) # aléatoire entre 0. et 1.
array([[0.2 , 0.63, 0.17, 0.11, 0.8 ],
       [0.16, 0.16, 0.52, 0.03, 0.87],
       [0.05, 0.41, 0.98, 0.36, 0.68],
       [0.66, 0.45, 0.24, 1.  , 0.66],
       [0.14, 0.91, 0.13, 0.14, 0.13]])
```

Si l’on relance la commande, la matrice créée sera alors différente, mais ses éléments seront toujours tirés selon la même loi :

```
>>> np.random.rand(5, 5) # un deuxième tirage
array([[0.88, 1.  , 0.94, 0.12, 0.47],
       [0.13, 0.67, 0.72, 0.78, 0.77],
       [0.49, 0.55, 0.07, 0.95, 0.3 ],
       [0.65, 0.28, 0.69, 0.97, 1.  ],
       [0.99, 0.52, 0.92, 0.68, 0.61]])
```

Rem: Certains nombres sont égaux à “1.” car on a restreint l’affichage aux deux premiers chiffres après la virgule, pour limiter la place prise par la matrice.

Un point important pour certaines simulations numériques et pour assurer la reproductibilité des résultats est de fixer une “graine” (🇬🇧 : *seed*) qui dicte où démarre la suite aléatoire utilisée par Python. En effet si l’on relance la même commande que précédemment, les nombres produits sont alors différents (comme on peut s’y attendre) :

```
>>> np.random.rand(5, 5) # aléatoire entre 0. et 1.
array([[0.2 , 0.63, 0.17, 0.11, 0.8 ],
       [0.16, 0.16, 0.52, 0.03, 0.87],
       [0.05, 0.41, 0.98, 0.36, 0.68],
       [0.66, 0.45, 0.24, 1.  , 0.66],
       [0.14, 0.91, 0.13, 0.14, 0.13]])
```

Par contre si pour des besoins de test ou de reproduction de bug on cherche à avoir le

2. rappel sur la loi uniforme : https://fr.wikipedia.org/wiki/Loi_uniforme_continue

même comportement, on peut fixer au préalable la graine :

```
>>> np.random.seed(2018)
>>> np.random.rand(5, ) # un premier tirage aléatoire
array([0.88234931, 0.10432774, 0.90700933, 0.3063989 , 0.44640887])
```

En relançant cette commande, on produit alors la même sortie une deuxième fois :

```
>>> np.random.seed(2018)
>>> np.random.rand(5, ) # un premier tirage aléatoire
array([0.88234931, 0.10432774, 0.90700933, 0.3063989 , 0.44640887])
```

Il existe le même genre de construction pour d'autres lois, la plus connue étant la loi gaussienne, aussi appelée loi normale, dont la fonction de densité (réelle) vaut :

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) . \quad (5.1)$$

On peut vérifier que les tirages obtenus par `random.randn numpy` coïncident avec la distribution de la loi gaussienne :

```
a = np.random.randn(10000)
x = np.linspace(-4, 4, 100)


fig = plt.figure(figsize=(8, 5))
# Affichage d'un histogramme normalisé
histogramme = plt.hist(a, bins=50, density=True)
from scipy.stats import norm

# Oublier les détails matplotlib et Latex dans un premier temps
plt.plot(x, norm.pdf(x), linewidth=3, color='black',
         label=r"$\frac{\exp\left(-\frac{x^2}{2}\right)}{\sqrt{2\pi}}$")

plt.xlabel("x")
plt.ylabel("Proportion")
plt.legend()
plt.show()
```

et qui produit l'affichage donné en Figure 5.1.

5.2 Import de fichiers en **numpy**

Un format de fichier classique pour sauvegarder des données est le format `.csv` ( : *comma separated values*), un format de données où par défaut les colonnes sont séparées par des virgules donc. Pour commencer il faut charger les packages utiles avec le package `download`. Si le package `download` n'est pas installé sur votre machine, installer le avec la commande `pip install download`, voir par exemple <https://pypi.org/project/download/> :

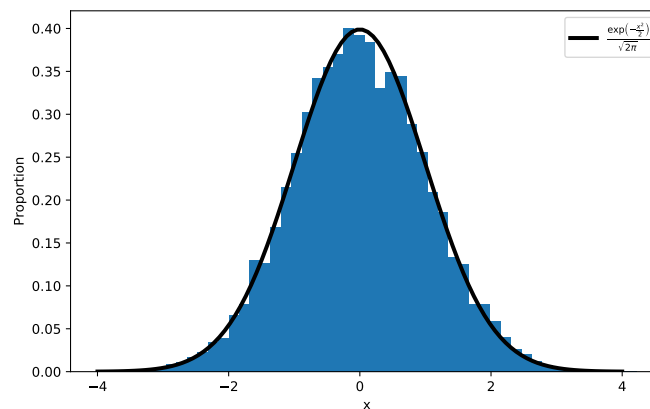


FIGURE 5.1 - Tirage de nombres gaussiens et comparaison avec la densité théorique

```
>>> import os # interface système d'exploitation
>>> from download import download
```

Ensuite on peut localiser où est le répertoire courant, histoire de savoir où l'on va télécharger et sauvegarder nos fichiers. Cela peut s'effectuer avec la commande `!pwd` sous Unix³ :

```
>>> !pwd # unix command "print working directory"
/home/jo/Documents/Mes_cours/Montpellier/HLMA310/Poly/codes
```

On définit ensuite le lien url qui contient le fichier de données que l'on va manipuler, dans notre exemple :

```
>>> url = "http://josephsalmon.eu/enseignement/datasets/data_test.csv"
```

Puis on va lancer en ligne de commande le téléchargement des fichiers dont on a besoin :

3. `pwd` est disponible sous Linux et Mac, sous Windows il faut utiliser `cd`

```
>>> path_target = "./data_set.csv"
>>> if not os.path.isfile(path_target):
>>>     print("Télécharge")
>>>     download(url, path_target, replace=False)
>>> else:
>>>     print("Données data_set.csv déjà téléchargées!")
Télécharge

file_sizes: 100% 30.0/30.0 [00:00<00:00, 37.0kB/s]

Downloading data from
http://josephsalmon.eu/enseignement/datasets/data_test.csv
(30 bytes)

Successfully downloaded file to ./data_set.csv
```

On peut maintenant visualiser le fichier qui a été téléchargé dans le répertoire courant, on peut utiliser `!cat`⁴ :

```
>>> !cat data_set.csv # commande unix cat pour visualiser
1,2,3,4,5
6,7,8,9,10
1,3,3,4,6
```

Note : d'autres fonctions Unix sont disponibles en plus de `pwd` et `cat` et peuvent être lancées avec (ou sans d'ailleurs) le point d'exclamation en début de commande :

- `cd` : pour changer de dossier (**change directory** en anglais)
- `cp` : pour copier des fichiers (**copy** en anglais)
- `ls` : pour lister les fichiers à l'endroit courant (**list** en anglais)
- `man` : pour avoir accès au manuel/aide (**manual** en anglais)
- `mkdir` : pour créer un dossier (**make directory** en anglais)
- `mv` : pour déplacer un fichier (**move directory** en anglais)
- `rm` : pour supprimer un fichier (**remove** en anglais)
- `rmdir` : pour supprimer un dossier (**remove directory** en anglais)
- etc.

Maintenant, pour lire le fichier téléchargé et le mettre au format **numpy** on peut utiliser la fonction `genfromtxt` de **numpy** :

```
>>> data_as_array = np.genfromtxt('data_set.csv', delimiter=',')
>>> data_as_array
array([[ 1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10.],
       [ 1.,  3.,  3.,  4.,  6.]])
```

4. `cat` est disponible sous Linux et Mac, sous Windows il faut utiliser **type**

5.3 Export de fichiers depuis **numpy**

On peut exporter les fichiers sous des formats classiques, comme le .csv ou le .txt comme dans l'exemple suivant :

```
np.savetxt("random_matrix.txt", data_as_array)
```

Pour vérifier qu'on a bien enregistré le fichier, on peut de nouveau utiliser la commande `cat` :

```
>>> !cat random_matrix.txt
1.00000000000000000000e+00 2.00000000000000000000e+00 ...
...
```

Vous pouvez aussi vérifier cela en ouvrant le fichier à la main (en le retrouvant dans un explorateur de fichier).

Le format pour sauvegarder un **array** (🇬🇧 : *save*) est le format `numpy` :

```
>>> np.save("random_matrix.npy", data_as_array)
```

Pour charger un tel fichier il suffit alors de faire l'opération inverse, celle de chargement (🇬🇧 : *load*) :

```
>>> data_as_array2 = np.load("random_matrix.npy")
>>> data_as_array2
array([[ 1.,  2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9., 10.],
         [ 1.,  3.,  3.,  4.,  6.]])
```

Pour effacer le fichier après usage, si l'on ne s'en sert plus on peut l'effacer avec la commande `rm`⁵ :

```
!rm data_set.csv
!rm random_matrix.txt
!rm random_matrix.npy
```

5.4 Slicing et masques

Comme pour le format *string* que l'on a déjà vu, le *slicing* est disponible pour le format **array**, avec en plus la possibilité d'y avoir accès pour chaque dimension :

```
>>> data_as_array2[:,0] # accès à la première colonne
array([1., 6., 1.]
```

ou encore :

5. `rm` est disponible sous Linux et Mac, sous Windows il faut utiliser **del**

```
>>> data_as_array2[1,:] # accès à la deuxième ligne
array([ 6.,  7.,  8.,  9., 10.]
```

Il est aussi utile de travailler avec des masques :

```
>>> print(data_as_array2 < 7)
>>> print(data_as_array2[data_as_array2 < 7])
>>> data_as_array2[data_as_array2 < 7]= 0. # met à zéro si < 7
>>> print(data_as_array2)
[[ True True True True True]
 [ True False False False False]
 [ True True True True True]]
[1. 2. 3. 4. 5. 6. 1. 3. 3. 4. 6.]
[[ 0.  0.  0.  0.  0.]
 [ 0.  7.  8.  9. 10.]
 [ 0.  0.  0.  0.  0.]
```

5.4.1 Copie / copie profonde (*deep copy*)

Pour des raisons de performance Python ne copie pas automatiquement les objets (par exemple passage par référence des paramètres de fonctions) :

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

L'exemple qui suit est à méditer :

```
>>> B = A
```

Notons que maintenant, changer B va maintenant affecter A :

```
>>> B[0,0] = 10
>>> B
array([[10,  2],
       [ 3,  4]])
```

```
et >>> print(B is A) # les deux objets sont les mêmes
>>> print(A)
True
[[10  2]
 [ 3  4]]
```

Pour éviter ce comportement, on peut demander une copie profonde (🇬🇧 : *deep copy*) de A dans B :

```
>>> B = A.copy()
```

Maintenant on peut observer le comportement différent :

```
>>> B[0,0] = 111
>>> B
array([[111,  2],
       [ 3,  4]])
```

A n'est alors plus modifié car on a créé une copie de l'objet :

```
>>> A
array([[10,  2],
       [ 3,  4]])
```

5.4.2 Concaténation d'array

On peut concaténer des arrays dans les deux directions. Commençons par le cas vertical :

```
>>> np.vstack((A,B)) # concaténation verticale
array([[ 10,  2],
       [  3,  4],
       [111,  2],
       [  3,  4]])
```

Le cas horizontal se gère de manière similaire :

```
>>> np.hstack((A,B)) # concaténation horizontale
array([[ 10,  2, 111,  2],
       [  3,  4,  3,  4]])
```

5.4.3 Fonctions sur les lignes / colonnes

Il existe des fonctions classiques qui permettent de faire les opérations simples sur des **arrays**.

Moyenne ( : *mean*) :

```
>>> np.mean(A)
4.75
```

```
>>> np.mean(B,axis=0) # moyenne en colonne
array([57.,  3.])
```

```
>>> np.mean(B,axis=1) # moyenne en ligne
array([56.5,  3.5])
```

Somme (🇬🇧: *sum*):

```
>>> np.sum(A)
19
```

```
>>> np.sum(A, axis=0) # somme en colonne
array([13,  6])
```

```
>>> np.sum(A, axis=1) # somme en ligne
array([12,  7])
```

Somme cumulée (🇬🇧: *cumsum*)

```
>>> np.cumsum(A) # noter l'ordre en ligne
array([10, 12, 15, 19])
```

```
>>> np.cumsum(A, axis=0) # somme cumulée en colonne
array([[10,  2],
       [13,  6]])
```

```
>>> np.cumsum(A, axis=1) # somme cumulée en ligne
array([[10, 12],
       [ 3,  7]])
```

Note : il existe la même chose avec `prod` et `cumprod` pour le produit au lieu de l'addition.

5.5 Algèbre linéaire

TODO: à compléter

Multiplication matricielle (deux formes : $A@B$ ou `np.dot(A, B)`)

Puissance matricielle : `np.linalg.matrix_power`.

Résolution de systèmes linéaires (on n'inverse jamais une matrice pour résoudre un système linéaire, on perdrait beaucoup de temps et d'énergie pour rien ce faisant...)

Décomposition spectrale

6

matplotlib, ou des graphiques en Python

On avait vu avec la Figure 3.2 comment afficher des éléments graphiques avec matplotlib. Nous allons y revenir en détails ici.

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

Rem: la commande `%matplotlib notebook` permet d'avoir plus d'interactions avec les graphiques en jupyter notebook et donne notamment la possibilité de zoomer sur les figures.

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

```
# Création de tableaux 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 :

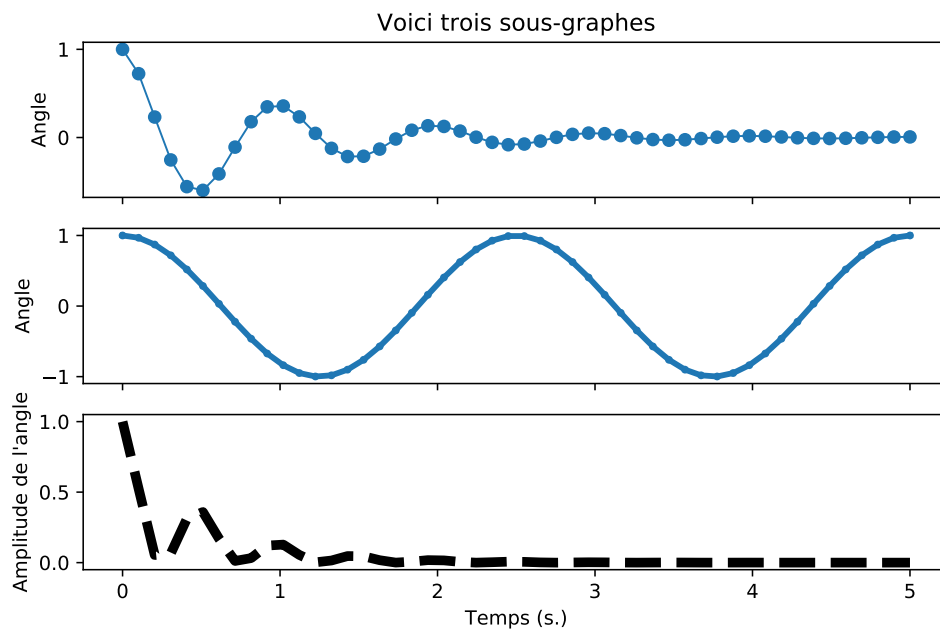


FIGURE 6.1 - Exemple de subplots

```
fig, axes = plt.subplots(3, 1, figsize=(8, 5), sharex=True)

axes[0].plot(x1, y1, 'o-', linewidth=1)
axes[0].set_title('Voici trois sous-graphes')
axes[0].set_ylabel('Angle')

axes[1].plot(x1, y2, '-.', linewidth=3)
axes[1].set_ylabel('Angle')

axes[2].plot(x1, y1**2, '--', color='black', linewidth=5)
axes[2].set_ylabel("Amplitude de l'angle")
axes[2].set_xlabel('Temps (s.)')

plt.show()
```

La commande `plt.show()` sert en général à afficher une figure en Python, c'est par exemple le cas en `ipython` (avec le mode `--pylab`) ou en `jupyter notebook`. Cette commande affiche alors toutes les figures et retourne au prompt `ipython`, en redonnant la main à l'utilisateur.

En mode non-interactif, cela affiche toutes les figures jusqu'à ce que les figures soient fermées (ce qui peut paraître étonnant les premières fois qu'on utilise Python, car on peut se demander où sont les autres images si l'on a lancé plusieurs affichages).

Troisième partie

**Introduction aux statistiques
descriptives**

7

Avertissement

L'objectif de cette partie est de décrire (et parfois de justifier) des techniques de résumé, de visualisation et d'exploration de données issues des sciences sociales ou d'ailleurs. Contrairement à l'usage du cours de statistiques inférentielles, nous ne ferons pas d'hypothèses sur la manière dont les données sont engendrées, il n'y aura pas de travail de modélisation stochastique. Nous ne pourrons pas développer une théorie de ce qui est « significatif » (autrement dit ce qui n'est pas dû au hasard), cela reviendra justement au cours de « statistiques inférentielles ». Pourtant, nous décrirons des quantités, des calculs qui trouvent leur justification grâce à la modélisation stochastique.

Les « statistiques descriptives » sont un ensemble de techniques qu'on appelle de différentes manières selon les lieux et les époques :

- Analyse de données
- Analyse de données exploratoires
- Apprentissage (*machine learning*)
- Fouille de données (*data mining*)

Des sites pour trouver des données, des idées, ...

<http://www.r-bloggers.com/> R-blogs

<http://www.worldmapper.org/> Worldmapper

<http://ncg.nuim.ie/> NCG

<http://www.rmetrics.org/blog> Rmetrics

<http://ego.psych.mcgill.ca/misc/fda/> FDA

<http://www.gapminder.org/> Gapminder

<http://www.banquemondiale.org/donnees/> WorldBank

<http://www.stat.pitt.edu/stoffer/tsa2/index.html> Time series in

R

http://cran.r-project.org/doc/contrib/Paradis-rdebuts_fr.pdf R pour débutants (E. Paradis)

<http://www.pallier.org/cours/stats.2009/> Stat Sciences Cognitives (Pailler)

<http://www.stat.columbia.edu/~cook/movabletype/mlm/> A. Gelman's blog

<http://www.infovis.info/index.php> Infovis

<http://quanti.hypotheses.org/> Quanti

<http://flowingdata.com/> Flowing data

http://alea.fr.eu.org/j/intro_R.html R sociologie (J. Barnier)

7.1 Vocabulaire : (petite) histoire du mot statistique

On décrit ici quelques techniques de résumé et de visualisation de données unidimensionnelles. Pour mieux comprendre l'objectif et le statut des statistiques dites descriptives, consultons deux dictionnaires composés à deux époques différentes et observons l'évolution du sens du mot *statistique*.

D'après le dictionnaire LITTRÉ (fin XIX^e siècle)

STATISTIQUE (sta-ti-sti-k') s. f.

1° Science qui a pour but de faire connaître l'étendue, la population, les ressources agricoles et industrielles d'un État. Achenwall, qui vivait vers la fin du milieu du XVIII^e siècle, est généralement considéré comme le premier écrivain systématique sur la statistique, et on dit que c'est lui qui lui a donné son nom actuel. Ils [les économistes] ont créé un mot pour désigner la science de cette partie de l'économie politique [les dénombrements], et l'appellent statistique, BACHAUM. Mém. secrets, t. XXIX, p. 123. La statistique, qui expose l'état des productions, des consommations, des ressources d'un État, à une époque donnée, est une science toute nouvelle, LETRONNE, Instit. Mém. inscr. et belles lett. t. VI, p. 166.

2° Plus généralement, science des dénombrements et de leurs conséquences. Statistique médicale, dénombrement de faits se rapportant aux morts, naissances, maladies, épidémies.

3° Description d'un pays relativement à son étendue, à sa population, à ses ressources agricoles et industrielles, etc. La statistique de la France. Cette branche de géographie politique que, d'après les Allemands, nous nommons statistique, MALTE-BRUN, Précis de géogr. univer. I, 8.

Selon cette définition, la statistique est un art de la description utile aux sciences sociales. Il n'est pas question de probabilités, de modélisation stochastique.

Un siècle plus tard, d'après le « Trésor de la langue française », le sens donné par Littré est réputé « vieilli ».

STATISTIQUE, subst. fém. et adj. I. Subst. fém.

1° Vieilli. Étude méthodique des faits économiques et sociaux par des classements, des inventaires chiffrés, des recensements, etc. La statistique apprendra

(...) que la population s'est accrue, que le prix des denrées a baissé ou haussé (COURNOT, *Fond. connaiss.*, 1851, p. 497). P. méton. Recueil de données numériques concernant des faits économiques et sociaux. J'ai lu avec tout l'intérêt qu'elle mérite votre intéressante statistique de l'arrondissement de Castres. Cet ouvrage renferme des renseignements précieux (...) pour votre département (LAMART., *Corresp.*, 1835, p. 105).

2° **Branche des mathématiques ayant pour objet l'analyse (généralement non exhaustive) et l'interprétation de données quantifiables.** Statistique lexicale, linguistique, prévisionnelle, sociologique, stylistique; Institut national de la statistique et des études économiques (INSEE); faire des statistiques. La statistique devait intervenir (...) en linguistique, grâce au mathématicien russe A. Markov qui, en 1913, analysa non seulement les fréquences des mots, mais aussi les fréquences des enchaînements de mots dans un poème célèbre de Pouchkine (*Hist. gén. sc.*, t. 3, vol. 2, 1964, p. 115). Une première et importante raison qui fausse les données de la statistique criminelle est l'écart qui sépare le nombre des faits criminels qu'elle enregistre, de ceux qui ont été effectivement commis (*Traité sociol.*, 1968, p. 220).

Statistique descriptive. « Ensemble des méthodes utilisables pour mettre en valeur les caractéristiques extérieures d'une série de chiffres » (COMBE 1971).

Statistique mathématique. « Ensemble des théorèmes, raisonnements et méthodes utilisables pour le traitement et l'analyse [de données chiffrées] » (COMBE 1971). PHYS. Ensemble de lois qui régissent le comportement de systèmes physiques comportant de nombreux éléments (MUSSET-LLORET 1964).

Statistique quantique. « Mode de distribution d'un ensemble de particules en fonction de leur énergie » (Nucl. 1964). Statistique de Bose-Einstein. Ensemble de lois selon lesquelles des particules peuvent être dans le même état quantique (CHARLES 1960). Statistique de Fermi-Dirac. Ensemble de lois établi par Fermi-Dirac selon (...) [lesquelles] il ne peut y avoir plusieurs particules dans le même état quantique (CHARLES 1960).

3° **Ensemble de données numériques (généralement analysées et interprétées) concernant une catégorie de faits.** Statistique annuelle, périodique; statistique criminelle, démographique, économique, électorale, financière, officielle; statistique de production; statistiques de l'état-civil; publier des statistiques. Une statistique portant sur 77 cas de rhumatismes scarlatins et 400 scarlatines (RAVAULT, VIGNON, *Rhumatol.*, 1956, p. 520). De précises statistiques anglaises et américaines mettent hors de doute que la mortalité et la morbidité croissent avec la décroissance du revenu des classes sociales (PERROUX, *Écon. XXe s.*, 1964, p. 350). P. Méton., *ÉCON.* Service, organisme qui établit de telles données. En France, la statistique comprend essentiellement l.N.S.E.E., mais aussi divers ministères (COMBE 1971).

On peut dire que l'INSEE exerce une activité qui relève de la statistique du XIX^e siècle. En fait l'INSEE utilise aussi des méthodes d'investigations modernes et ne se contente pas de recenser.

Aujourd'hui la statistique dite inférentielle ou mathématique s'appuie sur une idée fondamentale : les données à analyser sont obtenues par tirage selon une loi de probabilité inconnue mais dont on postule tout de même quelques propriétés. La statistique descriptive

se contente des données et cherche à les représenter.

8

Statistiques uni-variées

8.1 Échantillon, population

L'étude porte ici sur l'exploration et la visualisation des propriétés d'échantillons issus de populations.

Définition 8.1. [POPULATION] *En statistique, une population est un ensemble. Les éléments de cet ensemble sont appelés individus.*

Une population peut être constituée par un ensemble de communes, d'entreprises, . . . Ce n'est pas nécessairement un ensemble de personnes physiques. Parmi les populations, celles qui intéressent le plus les statisticiens sont les populations *recensées*. Dans ces populations recensées, les individus sont identifiés de manière unique. Connaître l'existence des individus qui forment la population ne signifie pas tout connaître de ces individus. Ainsi, l'administration fiscale dispose d'un recensement de tous les contribuables, mais elle ne connaît pas exactement les revenus et le patrimoine de chacun.

Le « corps électoral » d'un pays (d'une commune, etc) constitue un exemple de population recensée. L'ensemble des personnes contaminées par la grippe de Hong-Kong à un moment donné ne constitue pas une population recensée.

La suite des cotations d'une valeur auprès d'un marché boursier (NYSE, LSE, EURONEXT, etc.) est aussi une population. L'ensemble des cotations à un moment donné sur une place boursière est aussi une population.

Définition 8.2. [ÉCHANTILLON] *Un échantillon est une suite finie d'individus issus d'une population.*

L'ordre des individus au sein de l'échantillon peut être important, mais c'est rarement le cas. L'échantillon peut comporter des répétitions. Les commentaires des enquêtes et sondages soulignent souvent que l'échantillon est *représentatif* d'une population.

Nous illustrerons ces définitions à partir de la description des *municipalités belges*, une population de 577 individus, chaque individu représentant une commune belge. Ces communes peuvent être importantes (Anvers, environ 500 000 habitants) ou modestes (Hers-tappe, une centaine d'habitants).

Les individus constitutifs sont décrits par des variables (ou attributs, caractères, etc).

Définition 8.3. Une variable en statistique est en fait une fonction de la population dans un ensemble de valeurs qu'on appelle parfois modalités.

La manière de résumer/visualiser/traiter une variable dépend du type des valeurs que cette variable peut prendre.

8.2 Types de variables

Les communes belges, dans la table `belgianmunicipalities` sont décrites par un certain nombre de variables. On peut charger cette table de la manière suivante ¹ :

```
from download import download
import pandas as pd
path_target = "./belgianmunicipalities.csv"
df_belgium = pd.read_csv(path_target, index_col='Commune')
df_belgium = df_belgium.drop(['Unnamed: 0'], axis=1)
```

En particulier on peut obtenir la liste des colonnes utiles :

```
>df_belgium.columns
Index(['INS', 'Province', 'Arrondiss', 'Men04', 'Women04',
      'Tot04', 'Men03', 'Women03', 'Tot03', 'Diffmen',
      'Diffwom', 'DiffTOT', 'TaxableIncome', 'Totaltaxation',
      'averageincome', 'medianincome'], dtype='object')
```

La variable `Province` peut prendre 9 valeurs (1,...,9) qui codent les noms des différentes provinces de Belgique. Pour simplifier on va associer à chaque valeur un nom de province :

```
dictionnaire = {1 : 'Anv.', 2 : 'Brab.', 3 : 'Fl.occ.',
                4 : 'Fl.or.', 5 : 'Hainaut', 6 : 'Liège',
                7 : 'Limb.', 8 : 'Lux.', 9 : 'Namur'}
df_belgium = df_belgium.replace({'Province': dictionnaire})
```

La variable `Arrondiss` peut prendre 43 valeurs. Les variables `Men04`, `Women03`, ..., `Tot04` indiquent le nombre d'habitants (hommes, femmes, total) pour les années 2003 et 2004. Les variables `Diffmen`, `Diffxxx`, indiquent les variations du nombre d'habitants entre ces deux années (ces variables sont redondantes). Les dernières variables sont des données fiscales : `TaxableIncome` désigne le revenu imposable total en

1. Pour le téléchargement, on peut trouver les données à cette adresse : <http://josephsalmon.eu/enseignement/datasets/belgianmunicipalities.csv>

euros en 2001 (pour la commune); `Totaltaxation` désigne le revenu fiscal global en euros en 2001; `averageincome` désigne le produit moyen de l'impôt sur le revenu en euros en 2001; `medianincome` désigne le produit médian de l'impôt sur le revenu en euros en 2001.

Pour souligner que les variables ne se manipulent pas toutes de la même façon, le psychométricien Stevens a introduit une typologie, critiquée mais persistante introduite dans les prochaines sections.

8.2.1 Variables quantitatives

Intervalle Une variable ou plutôt une échelle de mesure est de type intervalle si elle est de type numérique et que la valeur zéro n'a pas de signification particulière. Par exemple, les mesures de température sont de ce type. La multiplication des températures exprimée en degrés Celsius n'a de plus aucun sens (10° Celsius n'est pas 10 fois plus chaud que 1° Celsius). Les notes obtenues aux examens peuvent parfois être considérées de ce type.

Rapport Une variable est de type ratio/rapport si elle prend ses valeurs dans un ensemble de nombres et si la multiplication est pertinente.

Dans le cas des municipalités belges, les variables `Men03`, `Totaltaxation`, ... sont des variables de type rapport. Nous les appellerons numériques.

8.2.2 Variables qualitatives

Les variables qualitatives ou catégorielles, ou facteurs (`factor` en R) prennent leurs valeurs dans un ensemble fini (le plus souvent) ou dénombrable. Elles sont dites

Ordinales si l'ensemble des valeurs est strictement ordonné, **et** si cet ordre est pertinent pour le problème étudié.

Nominale sinon.

Dans le cas des municipalités belges, les numéros des provinces prennent leur valeur dans \mathbb{N} qui est ordonné. Pourtant, nous ne souhaitons pas considérer cette variable comme ordinale : il s'agit donc d'une variable nominale.

Au passage rappelons ce qu'est une relation d'ordre.

Définition 8.4. [RELATION D'ORDRE] Une relation \mathcal{R} sur un ensemble E est dite relation d'ordre si et seulement si elle est

Transitive : $\forall x, y, z \in E, x\mathcal{R}y \text{ et } y\mathcal{R}z \Rightarrow x\mathcal{R}z$;

Anti-symétrique : $\forall x, y \in E, x\mathcal{R}y \text{ et } y\mathcal{R}x \Rightarrow x = y$;

Réflexive : $\forall x \in E, x\mathcal{R}x$;

Une relation d'ordre est dite *totale* si $\forall x, y \in E, x\mathcal{R}y$ ou $y\mathcal{R}x$. La relation \leq est totale sur \mathbb{Z}, \mathbb{R} . La relation \leq définie sur \mathbb{R}^2 par $(x, y) \leq (x', y') \Leftrightarrow (x \leq x') \text{ et } (y \leq y')$ est une relation d'ordre qui n'est pas totale, on dit qu'il s'agit d'une relation d'ordre *partiel*.

Exemple 8.5. L'ordre alphabétique est un ordre conventionnel sur les symboles qui constituent un alphabet. L'ordre lexicographique est un ordre total sur les mots construits sur cet alphabet ordonné.

La terminologie de Stevens considère comme variable ordonnée une variable qui prend ses valeurs dans un ensemble totalement ordonné. Il n'y a pas de raison de négliger des variables qui prennent leurs valeurs dans un ensemble partiellement ordonné.

8.3 Résumé de l'échantillon

Nous allons maintenant nous intéresser à ce qu'on appelle les statistiques uni-variées, à l'étude d'une variable à la fois et surtout, mais pas toujours, aux variables numériques.

Nous allons commencer par décrire des paramètres de localisation, la moyenne et l'espérance.

Définition 8.6. (MOYENNE/ESPÉRANCE) Soit $\{1, \dots, n\}$ un échantillon sur lequel on a mesuré une variable quantitative X (on note $X(i) = x_i$ pour $1 \leq i \leq n$), la moyenne de cet échantillon est définie par

$$\bar{x}_n = \frac{1}{n} \sum_{i=1}^n x_i . \quad (8.1)$$

Dans ces notes, on notera souvent cette moyenne \bar{x} (si tout est clair par ailleurs) ou \bar{x}_n si on veut se souvenir de la taille de l'échantillon.

Remarque 8.7. La moyenne est l'espérance de la variable $X: \{1, \dots, n\} \rightarrow \mathbb{R}$ qui à i fait correspondre x_i , sous la loi de probabilité uniforme P sur l'échantillon. C'est aussi le barycentre des points de la droite réelle x_1, \dots, x_n munis de la pondération $(\frac{1}{n}, \dots, \frac{1}{n})$.

Exemple 8.8. Le PIB par habitant peut être considéré comme une moyenne.

Dans le cas des municipalités belges, pour chaque municipalité, la variable `averageincome` décrit la moyenne du produit de l'impôt sur le revenu. Il s'agit d'une moyenne calculée sur la population des contribuables (les foyers fiscaux) de la municipalité. La moyenne est un indicateur de tendance centrale ou de localisation. Ce n'est pas le seul. Un autre indicateur, tout aussi important (et même plus important) est la médiane :

Définition 8.9. (MÉDIANE) Soit x_1, \dots, x_n un échantillon de réels, on appelle médiane de l'échantillon toute valeur m qui vérifie

$$|\{i : 1 \leq i \leq n, x_i < m\}| = |\{i : 1 \leq i \leq n, x_i > m\}| . \quad (8.2)$$

Exemple 8.10. Dans le cas des municipalités belges, pour chaque municipalité, la variable `medianincome` décrit la médiane du produit de l'impôt sur le revenu. Il s'agit d'une médiane calculée sur la population des contribuables de la municipalité. Cette médiane ne coïncide pas avec la moyenne, voir Figure 8.1, obtenue en lançant le code :

```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(df_belgium['medianincome'],
         df_belgium['averageincome'], '.')
plt.plot(df_belgium['medianincome'],df_belgium['medianincome'])
plt.xlabel('Médiane impôt sur le revenu')
plt.ylabel('Moyenne impôt sur le revenu')
plt.show()
```

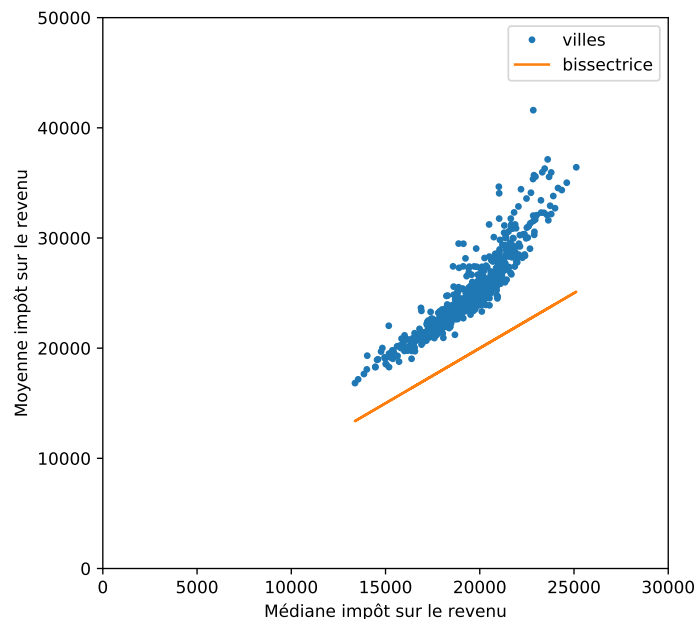


FIGURE 8.1 – Comparaison moyenne versus médiane

En 2001, dans les municipalités belges, la moyenne du retour de l'impôt sur le revenu était supérieure à la médiane (la ligne orange est la droite d'équation $y = x$). Le point le plus éloigné de la droite $y = x$ correspond à une exception de la province 2, voir Figure 8.2, le Brabant (il semblerait que la table ne distingue pas le Brabant wallon et le Brabant flamand cependant).

Ou plutôt en utilisant seaborn pour afficher les couleurs :

```
fig3 = sns.lmplot(x='medianincome', y='averageincome',
                 data=df_belgium, fit_reg=False, hue='Province')
plt.plot(df_belgium['medianincome'], df_belgium['medianincome'])
plt.xlabel('Médiane impôt sur le revenu')
plt.ylabel('Moyenne impôt sur le revenu')
```

Définition 8.11. [STATISTIQUES D'ORDRE] Si l'échantillon est formé des points x_1, \dots, x_n , en triant cet échantillon en ordre croissant, on obtient

$$x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n)} . \quad (8.3)$$

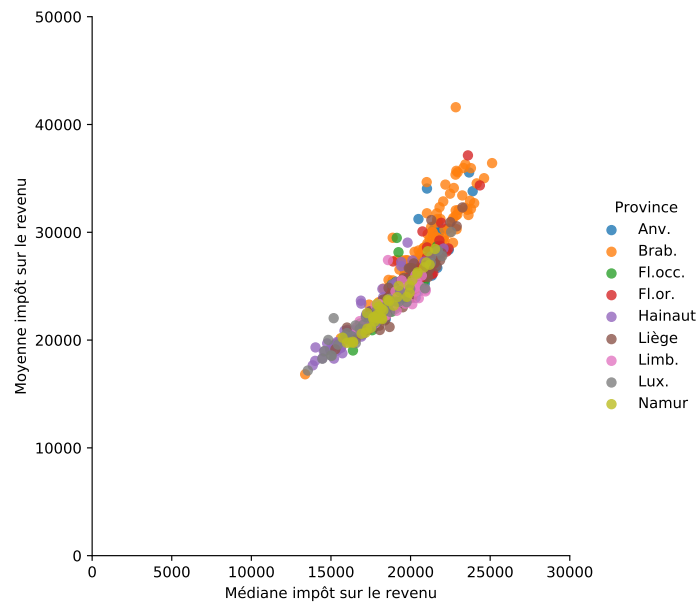


FIGURE 8.2 – Comparaison moyenne versus médiane

On appelle $x_{(i)}$ la i -ème statistique d'ordre. Si $x_i = x_{(j)}$, on dit que le rang de x_i dans l'échantillon est j .

Remarque 8.12. On suppose par la suite que les x_i sont deux à deux distincts.

Proposition 8.1. [MÉDIANE ET STATISTIQUE D'ORDRE]

Si n est impair la médiane coïncide avec $x_{((n+1)/2)}$.

Si n est pair alors toute valeur $m \in]x_{(n/2)}, x_{(n/2+1)}[$ est une médiane.

Exemple 8.13. Considérons l'échantillon formé par les 5 premières valeurs de la variable `averageincome` (les communes Aartselaar Anvers Boechout Boom Borsbeek).

```
> df_belgium['averageincome'][0:5]
Commune
Aartselaar    33809
Anvers        22072
Boechout      29453
Boom          21907
Borsbeek      26632
Name: averageincome, dtype: int64
```

On peut trier les valeurs constituant l'échantillon en ordre croissant; cela forme ce que l'on appelle les statistiques d'ordre. Ici :

```
> np.sort(df_belgium['averageincome'])[0:5])
array([21907, 22072, 26632, 29453, 33809])
```

ce qui signifie : $x_5 = 26632$ et $x_{(5)} = 33809$.

8.4 Fonction de répartition, fonction quantile

La moyenne et la médiane sont des résumés très concis de l'échantillon. A l'autre extrême, la fonction de répartition et la fonction quantile sont des résumés très riches de l'échantillon.

Définition 8.14. [FONCTION DE RÉPARTITION] *Étant donné un échantillon $\{1, \dots, n\}$ et une variable quantitative X (avec $X(i) = x_i$), la fonction de répartition associée à X est la fonction $F_n : \mathbb{R} \rightarrow [0, 1]$ définie par*

$$F_n(y) = \frac{1}{n} \sum_{i=1}^n \mathbb{1}_{(-\infty, y]}(x_i) = \sum_{i: x_i \leq y} \frac{1}{n}. \quad (8.4)$$

La valeur de la fonction de répartition en y est le poids des points de l'échantillon pour lesquels la variable X est inférieure (ou égale) à y .

Si A désigne un sous-ensemble de E , l'indicatrice de A est une fonction notée $\mathbb{1}_A$ de E dans $\{0, 1\}$ définie par

$$\mathbb{1}_A(y) = \begin{cases} 1, & \text{si } y \in A \\ 0, & \text{sinon.} \end{cases} \quad (8.5)$$

Dans la définition de la fonction de répartition $\mathbb{1}_{(-\infty, y]}$ est l'indicatrice de la demi-droite $(-\infty, y]$.

La Figure 8.3a montre la fonction de répartition de l'impôt médian sur les communes de Belgique en 2001. La fonction de répartition fournit beaucoup plus de renseignements que la moyenne et la médiane. La Figure 8.3b montre la fonction de répartition de la même variable sur les sous-échantillons formés par les communes de la province d'Anvers.

Toute fonction de répartition possède les propriétés suivantes.

Proposition 8.2. *Une fonction de répartition est croissante, continue à droite, possède une limite à gauche en tout point.*

Démonstration. [PONDÉRATION UNIFORME] La monotonie se déduit immédiatement de l'observation

$$y \leq y' \implies \{i: x_i \leq y\} \subseteq \{i: x_i \leq y'\}. \quad (8.6)$$

La continuité à droite demande à peine plus d'efforts. Notons $x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n)}$ le réarrangement croissant de x_1, x_2, \dots, x_n . Si $F_n(y) = j/n$ alors $x_{(j)} \leq y < x_{(j+1)}$. Si $(y_k)_{k \in \mathbb{N}}$ est une suite réelle décroissante qui tend vers $y \in \mathbb{R}$, alors la suite

$$(|\{i: i \leq n, x_i \leq y_k\}|)_{k \in \mathbb{N}}, \quad (8.7)$$

est une suite décroissante minorée par $|\{i: i \leq n, x_i \leq y\}|$, elle est donc convergente vers une limite $\ell \geq |\{i: i \leq n, x_i \leq y\}|$. Si $\ell > n \times F_n(y) = j = |\{i: i \leq n, x_i \leq y\}|$, alors pour tout $k \in \mathbb{N}$, $y < x_{(j+1)} \leq y_k$. Ceci contredit l'hypothèse $y_k \rightarrow y$.

L'existence d'une limite à gauche en tout point se vérifie de la même façon. \square

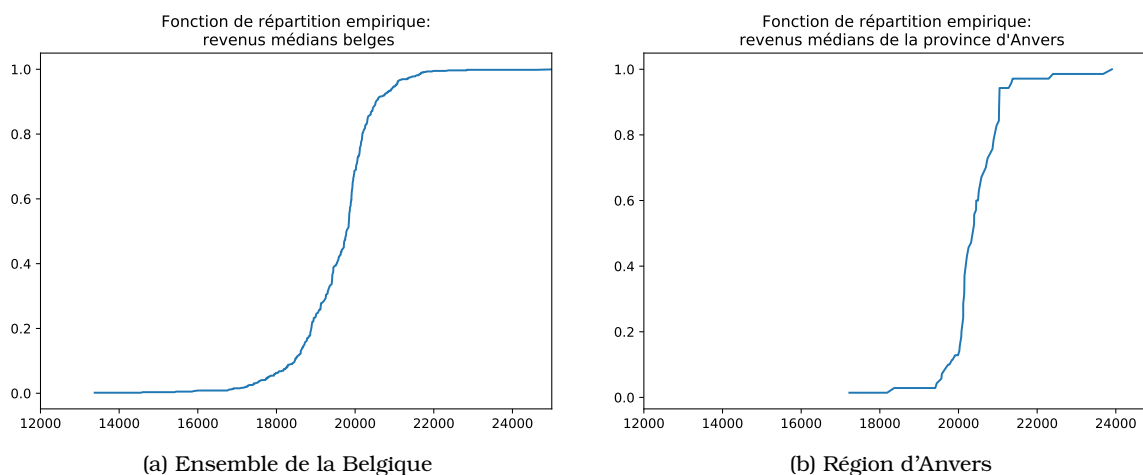


FIGURE 8.3 – Fonction de répartition de la médiane de l'impôt sur le revenu en 2001 (en Euros)

Une fonction de répartition n'est pas toujours « inversible » (dans le cas des fonctions de répartition rencontrées en statistique, qui sont constantes par morceaux, ces fonctions ne sont pas inversibles). Mais on peut définir une fonction qui est *presque* une inverse de la fonction de répartition, la fonction quantile.

Définition 8.15. [QUANTILES, QUARTILES, DÉCILES] *Étant donné un échantillon de taille n sur lequel on a mesuré la variable X ($X(i) = x_i$), de fonction de répartition associée F_n , la fonction quantile associée $F_n^{\leftarrow} :]0, 1[\rightarrow \mathbb{R}$ est définie par*

$$F_n^{\leftarrow}(p) = \inf\{x: F_n(x) \geq p\} . \quad (8.8)$$

On appelle quantile d'ordre $p \in (0, 1)$, la quantité $F_n^{\leftarrow}(p)$. La médiane est égale à $F_n^{\leftarrow}(1/2)$, les premiers et troisièmes quartiles sont égaux à $F_n^{\leftarrow}(1/4)$ et $F_n^{\leftarrow}(3/4)$. Enfin, les déciles sont les quantiles $F_n^{\leftarrow}(k/10)$ pour $k = 1, \dots, 9$.

Dans le cas d'une pondération uniforme, les quantiles sont reliés aux statistiques d'ordre.

Proposition 8.3. [FONCTION QUANTILE EMPIRIQUE ET STATISTIQUES D'ORDRE] *Si x_1, \dots, x_n est un échantillon numérique de valeurs distinctes, si $x_{(1)} < x_{(2)} < \dots < x_{(n)}$ désigne les statistiques d'ordre associées, et si F_n^{\leftarrow} désigne la fonction quantile alors*

$$\text{Si } \frac{k}{n} < p \leq \frac{k+1}{n}, \text{ alors } F_n^{\leftarrow}(p) = x_{(k+1)} . \quad (8.9)$$

Démonstration. Remarquons

$$F_n(x_{((k+1)/n)}) = \frac{k+1}{n} \quad (8.10)$$

et pour $x_{(k)} \leq x < x_{((k+1)/n)}$, $F_n(x) = k/n < p$. Donc l'infimum des x tels que $F_n(x) \geq p$ est $x_{(k+1)}$. \square

Proposition 8.4. [PROPRIÉTÉS DE LA FONCTION QUANTILE.]

1. La fonction quantile F^{\leftarrow} est définie sur $]0, 1[$ (éventuellement sur $[0, 1]$).
2. La fonction quantile est croissante (pas toujours strictement).
3. La fonction quantile est continue à gauche.
4. Pour tout $p \in]0, 1[$ $p \leq F \circ F^{\leftarrow}(p)$.
5. Pour tout $x \in \mathbb{R}$ $F^{\leftarrow} \circ F(x) \leq x$.
6. $\forall p \in (0, 1), \forall x \in \mathbb{R}, p \leq F(x) \Leftrightarrow F^{\leftarrow}(p) \leq x$.

Démonstration.

1) et 2) sont évidents.

3) Continuité à gauche. On peut observer d'abord :

$$F^{\leftarrow}(p) = \sup_n \inf \left\{ x : F(x) \geq p - \frac{1}{n} \right\}. \quad (8.11)$$

4) Soit $(x_n)_n$ une suite qui décroît vers $F^{\leftarrow}(p)$. Par continuité à droite de F (F est une fonction de répartition), $F(x_n)$ décroît vers $F \circ F^{\leftarrow}(p)$, comme les $F(x_n)$ sont par définition, plus grands que p , le résultat s'ensuit.

5) est évident.

6) Supposons $p \leq F(x)$ alors $F^{\leftarrow}(p) = \inf\{y : F(y) \geq p\} \leq x$. Réciproquement si $F^{\leftarrow}(p) \leq x$. On a $\inf\{y : F(y) \geq p\} \leq x$, il existe donc une suite décroissante et convergente $(y_n)_n$ dont la limite est inférieure ou égale à x telle que $F(y_n) \geq p$, et $\lim_n F(y_n) = p$. Or $p = \lim_n F(y_n) = F(\lim_n y_n) \leq F(x)$. \square

Dans le cas des fonctions de répartition et des fonctions quantiles empiriques, les arguments sont encore plus simples.

Démonstration. Si $k/n < p \leq (k+1)/n$ pour $0 \leq k < n$, alors $F_n^{\leftarrow}(p) = x_{(k+1)}$.

Si $k/n < p \leq (k+1)/n$, $F_n \circ F_n^{\leftarrow}(p) = F_n(x_{(k+1)}) = (k+1)/n \geq p$.

Si $x_{(k)} \leq x < x_{(k+1)}$, $F_n^{\leftarrow} \circ F_n(x) = F_n^{\leftarrow}(F_n(x_{(k)})) = F_n^{\leftarrow}(k/n) = x_{(k)} \leq x$. \square

Exemple 8.16.

```
>>> np.percentile(df_belgium['averageincome'],
...               [1 / 4, 1 / 2, 3 / 4])
17398.54, 18047.86, 18274.1
```

Proposition 8.5. [FONCTION DE RÉPARTITION, FONCTION QUANTILE]

Soit x_1, \dots, x_n un échantillon numérique et $x_{(1)} \leq x_{(2)} \leq \dots \leq x_{(n-1)} \leq x_{(n)}$ les statistiques d'ordre associées. Si $x_{(k)} < x_{(k+1)}$ pour $x_{(k)} \leq y < x_{(k+1)}$

$$F_n(y) = \frac{k}{n}. \quad (8.12)$$

Pour p : $\frac{k}{n} < p \leq \frac{k+1}{n}$

$$F_n^{\leftarrow}(p) = x_{(k+1)} . \quad (8.13)$$

Et

$$F_n^{\leftarrow}(k/n) = x_{(k)} . \quad (8.14)$$

Mais

$$F_n^{\leftarrow} \circ F_n(x_{(k)}) \geq x_{(k)} \quad (8.15)$$

avec égalité seulement si $x_{(k)} \leq y < x_{(k+1)}$ et

$$F_n \circ F_n^{\leftarrow}(k/n) \geq k/n \quad (8.16)$$

avec égalité seulement si $x_{(k)} < x_{(k+1)}$.

8.5 Variance, écart-type, etc

La moyenne et la médiane sont deux paramètres de *localisation*. Pour résumer un échantillon, on a l'habitude de compléter ces deux paramètres par un ou des paramètres de *dispersion*. Le plus courant d'entre eux est la variance, ou de manière équivalente la racine carrée de la variance : l'écart-type.

Définition 8.17. [ÉCART QUADRATIQUE/ ÉCART ABSOLU MOYEN] Étant donné un échantillon $\{1, \dots, n\}$ et une variable quantitative X (avec $X(i) = x_i$), si z est une constante réelle quelconque, l'écart quadratique moyen à $z \in \mathbb{R}$ est défini par

$$\frac{1}{n} \sum_{i=1}^n (x_i - z)^2 . \quad (8.17)$$

L'écart absolu moyen à z est défini par

$$\frac{1}{n} \sum_{i=1}^n |x_i - z| . \quad (8.18)$$

Définition 8.18. [VARIANCE ET ÉCART-TYPE] Étant donné un échantillon $\{1, \dots, n\}$ et une variable quantitative X (avec $X(i) = x_i$), de moyenne $\bar{x}_n = \frac{1}{n} \sum_{i=1}^n x_i$, on appelle variance empirique S_n^2 l'écart quadratique moyen des points de l'échantillon à la moyenne :

$$s_n^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}_n)^2 , \quad (8.19)$$

on appelle écart-type ou déviation standard $\sigma_n = \sqrt{s_n^2}$.

```
>>> np.var(df_belgium['medianincome'])
4014584.7996575553
>>> np.std(df_belgium['medianincome'])
2003.6428822665869
```

Un autre indicateur important de la dispersion :

Définition 8.19. [ÉCART INTERQUARTILE, IQR] Étant donné un échantillon $\{1, \dots, n\}$, muni d'une pondération p_1, \dots, p_n , et une variable quantitative X (avec $X(i) = x_i$), de fonction quantile associée F_n^{\leftarrow} , l'écart interquartile de cette variable est défini par

$$F_n^{\leftarrow}(3/4) - F_n^{\leftarrow}(1/4) . \quad (8.20)$$

```
>>> quantiles = np.percentile(df_belgium['averageincome'],
                               [25, 75])
>>> quantiles[1] - quantiles[0] # IQR
4249.0
```

8.6 Boîte à moustaches

En statistique descriptive, on cherche à construire de bons résumés numériques et graphiques. Les couples (moyenne, écart-type), ou (médiane, IQR) constituent des résumés numériques d'échantillons numériques. Les boîtes à moustaches, ou *boxplots*, introduites par MCGILL, TUKEY et LARSEN (« Variations of box plots »), constituent des résumés graphiques très efficaces.

La boîte à moustaches (ou *boxplot*) permet de visualiser la localisation et la dispersion d'un échantillon. La boîte est limitée par le 1^{er} et le 3^e quartile, elle est coupée en deux

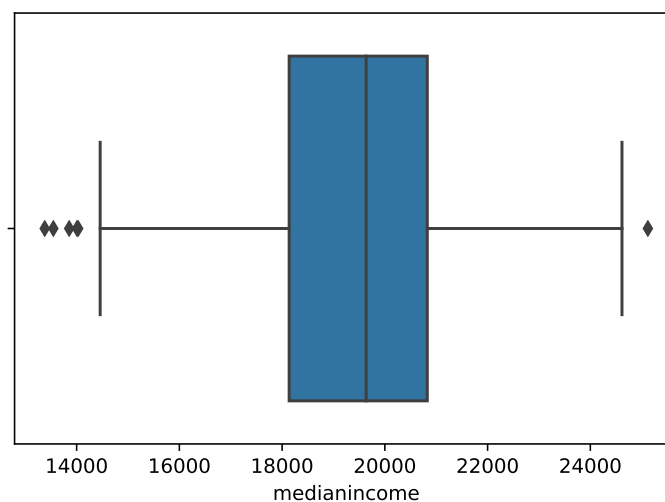


FIGURE 8.4 – Boîte à moustaches de la médiane de l'impôt sur le revenu pour les 577 municipalités belges

par la médiane. Les deux moustaches s'étendent de part et d'autre de la boîte sur une longueur qui est par défaut 3/2 fois l'écart interquartile (avec des modifications à la marge pour les cas extrêmes, cf. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html)

In other words, where IQR is the interquartile range (Q3-Q1), the upper whisker will extend to last datum less than Q3 + whisk*IQR. Similarly, the lower whisker

will extend to the first datum greater than $Q1 - \text{whisk} * \text{IQR}$. Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, whisk can be the string 'range' to force the whiskers to the min and max of the data.

Les individus (exceptionnels) qui sortent des moustaches sont représentés par des cercles.

Remarque 8.20. *Le choix par défaut de 3/2 est motivé par les propriétés de la « loi normale ». Si tous les échantillons ressemblaient à des échantillons de lois normales, on s'attendrait à couvrir 95% des données entre les moustaches. Les points situés à l'extérieur des moustaches, les extrêmes, sont signalés par un symbole spécial.*

8.7 Violons

TODO: rajouter des illustrations de violon / (🇬🇧 : violin)

8.8 Propriétés de l'espérance et de la médiane

Nous avons défini moyenne et médiane par un procédé de calcul. La moyenne et la (les) médiane(s) possèdent des propriétés remarquables qui les caractérisent. Ce sont des solutions de problèmes d'optimisation bien particuliers.

Théorème 8.1. *La moyenne minimise l'écart quadratique moyen.*

Démonstration. On note \bar{x}_n la moyenne de l'échantillon x_1, \dots, x_n . Pour tout $a \in \mathbb{R}$,

$$\begin{aligned} \sum_{i=1}^n (x_i - a)^2 &= \sum_{i=1}^n (x_i - \bar{x}_n + \bar{x}_n - a)^2 \\ &= \sum_{i=1}^n \left[(x_i - \bar{x}_n)^2 + 2(x_i - \bar{x}_n)(\bar{x}_n - a) + (\bar{x}_n - a)^2 \right] \\ &= \sum_{i=1}^n (x_i - \bar{x}_n)^2 + 2(\bar{x}_n - a) \sum_{i=1}^n (x_i - \bar{x}_n) + \sum_{i=1}^n (\bar{x}_n - a)^2 \\ &= \sum_{i=1}^n (x_i - \bar{x}_n)^2 + 2(\bar{x}_n - a) \left(\sum_{i=1}^n x_i - n\bar{x}_n \right) + (\bar{x}_n - a)^2 \\ &= \sum_{i=1}^n (x_i - \bar{x}_n)^2 + 2(\bar{x}_n - a)(\bar{x}_n - \bar{x}_n) + (\bar{x}_n - a)^2 \\ &= \sum_{i=1}^n (x_i - \bar{x}_n)^2 + (\bar{x}_n - a)^2 \geq \sum_{i=1}^n (x_i - \bar{x}_n)^2. \end{aligned}$$

□

Théorème 8.2. *Les médianes minimisent l'écart absolu moyen*

Démonstration pour un échantillon de taille impaire. Considérons un échantillon x_1, \dots, x_{2n+1} . Après un tri en ordre croissant, les statistiques d'ordre sont $x_{(1)} \leq \dots \leq x_{(2n+1)}$. La médiane m est égale à $x_{(n+1)}$. On s'accordera sur $x_{(0)} = -\infty$ et $x_{(2n+2)} = +\infty$.

Soit $a \in \mathbb{R}$, l'écart absolu moyen de a à l'échantillon est

$$\frac{1}{n} \sum_{i=1}^n |x_i - a| = \frac{1}{n} \sum_{i=1}^n |x_{(i)} - a|.$$

On passe d'une écriture à l'autre en permutant les termes de la sommation. Observons que $a \mapsto \frac{1}{n} \sum_{i=1}^n |x_i - a|$ est une fonction continue de a .

Supposons maintenant $x_{(i-1)} < a < x_{(i)}$ pour $1 \leq i \leq 2n+2$.

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n |x_i - a| &= \sum_{j=1}^{i-1} (a - x_{(j)}) + \sum_{j=i}^{2n+1} (x_{(j)} - a) \\ &= (2i - 2 - 2n - 1)a + \sum_{j=i}^{2n+1} x_{(j)} - \sum_{j=1}^{i-1} x_{(j)}. \end{aligned}$$

Si $i \leq n+1$, $2i - 2 - 2n - 1 \leq -1$, $a \mapsto \frac{1}{n} \sum_{i=1}^n |x_i - a|$ est décroissante sur $]x_{(i-1)}, x_{(i)}[$. Si $i > n+1$, $2i - 2 - 2n - 1 \geq 0$, $a \mapsto \frac{1}{n} \sum_{i=1}^n |x_i - a|$ est croissante sur $]x_{(i-1)}, x_{(i)}[$. L'écart absolu moyen est une fonction linéaire sur chaque intervalle $[x_{i-1}, x_i]$. L'écart absolu moyen atteint donc son minimum en $x_{(n+1)}$. \square

La moyenne et la médiane sont deux indicateurs de localisation. Ils peuvent être (et sont en général) différents.

Bibliographie

- BOSWELL, D. et T. FOUCHER. *The Art of Readable Code*. O'Reilly Media, 2011 (p. 9).
- COURANT, J. et al. *Informatique pour tous en classes préparatoires aux grandes écoles : Manuel d'algorithmique et programmation structurée avec Python*. Eyrolles, 2013 (p. 9).
- GUTTAG, J. V. *Introduction to Computation and Programming Using Python : With Application to Understanding Data*. MIT Press, 2016 (p. 9).
- MCGILL, R., J. W. TUKEY et W. A. LARSEN. « Variations of box plots ». *The American Statistician* 32.1 (1978), p. 12-16 (p. 65).
- SKIENA, S. S. *The algorithm design manual*. T. 1. Springer Science & Business Media, 1998 (p. 9).
- VANDERPLAS, J. *Python Data Science Handbook*. O'Reilly Media, 2016 (p. 9).