

page début réservée PUR 1

page début réservée PUR 2

page début réservée PUR 3

page début réservée PUR 4

page début réservée PUR 5

page début réservée PUR 6

# Remerciements

Ce livre existe tout d'abord grâce à tous·t·es les contributeur·rices de la communauté Python qui développent le langage, les outils et accueillent chaleureusement les débutant·e·s.

Le point de départ de cet ouvrage a été deux formations organisées au sein des laboratoires SESSTIM et GEMASS. Sa rédaction a ensuite bénéficié de nombreuses relectures et conseils de collègues.

Nous remercions tout particulièrement François Husson d'avoir accueilli et accompagné ce livre et les deux relecteurs Renaud Debailly et Florian Cafiero pour leurs suggestions.

Nous remercions aussi Marie-Alix Molinié, Léo Mignot, Fátima Gauna, Paul Guille-Escuret, Kamil Ghouati, Sébastien Plutniak, Vincent Maioli, Marione et Éric Schultz pour avoir échangé sur l'utilisation de la programmation en SHS, joué le rôle de « cobaye » et corrigé des erreurs de fond et de forme.

L'illustration de la couverture est née de l'inspiration de KaM ([kamsblog](#) sur Instagram). Python et chat s'accordent généralement bien.

Les erreurs restantes sont évidemment celles des auteurs.

*À mon grand-père, qui a accompagné la genèse de ce livre. À ma famille.*

- Matthias -

*À celles et ceux qui transmettent.*

- Émilien -



# Prologue

Ce manuel propose aux praticien·ne·s des Sciences Humaines et Sociales (SHS), étudiant·es<sup>1</sup>, chercheur·euses ou chargé·es d'études en sociologie, science politique, géographie, histoire ou anthropologie (sans exhaustivité), de s'initier à la programmation informatique en langage Python avec comme objectif le traitement de données. Les données sont ici entendues au sens large.

Notre constat de départ est le suivant : alors que la programmation Python prend une place grandissante dans le traitement de données<sup>2</sup>, il n'existe actuellement aucune littérature en français permettant de présenter les usages possibles en SHS. Cela s'explique par le fait que l'utilisation massive du langage Python dans de nombreux secteurs s'est surtout développée durant la décennie précédente. Or, nous partageons une conviction : non seulement ces outils sont accessibles et utiles pour tous·tes mais plus encore acquérir les bases de la programmation permet d'améliorer la qualité des pratiques par une plus grande transparence et reproductibilité dans le traitement de données.

Un préjugé récurrent voudrait que les SHS se montrent méfiantes à l'égard des nouveautés technologiques et, à ce titre, se tiennent à distance des évolutions informatiques. Non seulement ce préjugé est faux, en témoignent les traditions poussées d'analyse statistique en sciences sociales ou l'usage des historiens·nes et des archéologues des bases de données, mais a pour conséquence de limiter l'appropriation de ces outils. Il nous semblait nécessaire de remédier à cela en proposant une introduction adaptée<sup>3</sup>. Cette certitude vient de nos parcours personnels. Issus tous les deux du domaine de la physique, nous avions bien un verni initial de programmation de notre formation. Nous avons cependant développé notre pratique

---

1. Le prologue et la conclusion du manuel tiendront compte de l'écriture inclusive, d'autant plus importante comme marqueur du caractère genré des pratiques dans les domaines touchant à l'informatique. Cependant, plusieurs lecteur·rices nous ont souligné que maintenir l'écriture inclusive dans l'ensemble du manuel alourdissait la lecture en plus du code ce qui nous a amené à la suspendre.

2. La littérature en anglais n'est pas non plus très développée pour les SHS. Si un livre récent de Phillip Brooker destiné à un public de chercheur·euses en *Science Studies* propose une perspective forte sur la programmation (Brooker, 2019), rares sont les *social scientists* engagé·e·s dans une diffusion des outils de programmation.

pour répondre à des problèmes de nos quotidiens respectifs. Matthias avait besoin d’automatiser du traitement de données dans ses expériences de biophysique. Émilien voulait pouvoir nettoyer les données d’une pétition trouvée sur Internet. Par la suite, Matthias, se passionnant pour ces outils, a décidé d’activement participer à leur développement pour la communauté Python, tandis qu’Émilien a de plus en plus utilisé ces outils dans ses recherches de SHS. Il utilise Python en sociologie pour récupérer des données sur Internet, pour tracer des cartes ou encore produire des visualisations faciles à communiquer..

Ce livre entend démontrer que la programmation en Python est non seulement utile pour les SHS mais aussi accessible. Il présente les ressources nécessaires pour s’y familiariser progressivement. Nous ne voulons pas vous transformer en développeur·se mais vous permettre de recourir à la programmation en Python dans les cas où cette approche peut vous être utile. Par ailleurs, comme ces outils peuvent aussi participer à faire émerger de nouvelles questions, leur utilisation ouvre l’accès à de nouvelles données et stratégies d’enquête.

Pour suivre ce manuel, une expérience préalable de programmation n’est pas nécessaire. Nous vous proposons d’apprendre, à travers un langage adapté, une philosophie minimale de la programmation pour le traitement des données.

## Un livre de plus ?

Comme pour les livres de cuisine, il existe beaucoup de manuels de programmation. Et Internet regorge de tutoriels, de vidéos et de forums permettant de trouver des réponses à ses questions voire à celles qu’on ne se pose pas<sup>3</sup>. Pourquoi alors écrire un livre en plus dans ce foisonnement de ressources ?

L’ambition de ce livre est de vous amener à comprendre les bases de la programmation et de pouvoir l’utiliser dans des applications déjà connues. Ce livre doit être lu comme une porte d’entrée pour comprendre l’intérêt de la programmation, identifier les usages possibles et les utiliser. Ce livre est aussi une boussole pour se repérer dans la jungle de la littérature existante. Pour cette raison, nous prenons le temps de développer la démarche, en ne nous limitant pas à présenter uniquement du code.

Ces pages s’adressent donc à un·e lecteur·rice qui voudrait apprendre à utiliser Python. Il s’inspire d’initiatives existantes pour d’autres langages<sup>4</sup> qui visent à adapter la programmation aux besoins spécifiques de praticiens·nes. Trop souvent, les livres se concentrent sur les premières étapes générales et formelles de la programmation puis laissent les applications et les adaptations à la charge de l’utilisateur·rice. Nous voudrions pouvoir vous accompagner tout au long de l’apprentissage de ce langage avec comme ligne de mire votre autonomisation.

---

3. Vous trouverez en fin d’ouvrage des sites et des livres que nous avons appréciés.

4. Par exemple, le manuel écrit par Julien Barnier pour R. Barnier (2010).

Par ailleurs, si des ressources didactiques existent pour les SHS, elles se trouvent être pour la très grande majorité en anglais. Or, il est souvent plus facile de débuter dans sa langue d'usage avant de se frotter à des documentations techniques en anglais. Ce livre propose de faciliter cette acculturation pour ensuite bénéficier au mieux des nombreuses ressources existantes.

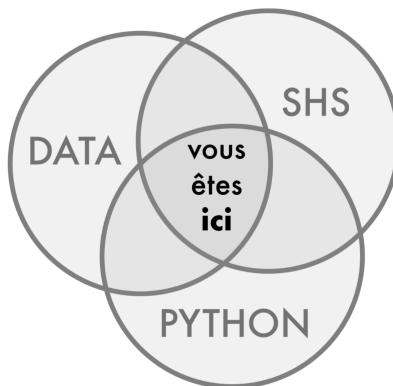


Fig. 1 – Positionnement de ce manuel

## Pourquoi lire ce manuel ?

Voici quelques bonnes raisons de parcourir ce manuel :

- ♣ se former à un outil de plus en plus utilisé pour le traitement de données ;
- ♣ se faire une idée de son potentiel ;
- ♣ identifier de nouvelles solutions voire de nouvelles données ;
- ♣ enseigner Python.

Dans la rédaction de ce manuel, nous avons fait quatre choix importants. Le premier est de restreindre l'usage de Python au domaine des SHS pour répondre à leurs besoins spécifiques : ce n'est donc pas un ouvrage générique. Le deuxième est d'insister sur la pratique plutôt que sur l'érudition ou la théorie : certains aspects ne seront pas couverts et nous ne revenons pas systématiquement sur les fondements théoriques. Le troisième est de fournir suffisamment d'informations et de vocabulaires pour permettre d'aller plus loin en programmation si cela vous était nécessaire.

Un quatrième choix important a été fait : celui de principalement s'appuyer sur les outils existants. Cela signifie deux choses : d'une part, que certains outils peuvent évoluer ; d'autre part, que de nouveaux outils plus adaptés peuvent apparaître.

Pour cette raison, nous soulignons quand certaines manières de résoudre un problème sont susceptibles de changer rapidement et nous insistons surtout sur la démarche de programmation plus que sur la forme.

En effet, ce manuel est aussi un plaidoyer pour familiariser le lecteur·rice à la programmation. Pourquoi ? La place de l'informatique et des algorithmes ne cessant de prendre de l'importance dans nos usages, nous considérons comme important d'avoir une culture minimale de l'activité de programmation, car programmer :

- ➊ développe la pensée abstraite et la décomposition des problèmes ;
- ➋ encourage la créativité pour résoudre des problèmes ;
- ➌ est une compétence professionnelle utile tout au long de la vie ;
- ➍ renforce la confiance en soi ;
- ➎ favorise un usage actif de la technologie ainsi que la pensée critique ;
- ➏ est amusant.

## Comment utiliser ce manuel

Le déroulement du manuel est progressif, un·e débutant·e peut suivre les chapitres un à un pour progressivement enrichir sa compréhension du langage et ses compétences pratiques<sup>5</sup>. À la fin de la lecture, vous aurez une vision assez diversifiée de ce qu'il est possible de faire. De la même façon qu'une langue s'apprend en identifiant des tournures de « phrases types », nous présentons des manières d'écrire certaines opérations que vous pouvez reprendre. Nous introduisons aussi de nouveaux concepts suivant les besoins pratiques. Au fil des pages, nous montrons plusieurs manières de résoudre un même problème.

Le manuel débute par un premier chapitre consacré à une présentation générale du langage et de ses spécificités en soulignant sa pertinence pour les SHS. Les chapitres suivants familiarisent progressivement à la logique de programmation en Python, débutant par les notions de base du langage puis en abordant des aspects plus spécifiques de manipulation des données.

Nous présentons ensuite comment réaliser les traitements statistiques couramment rencontrés en SHS, des statistiques descriptives à la visualisation. Nous insistons sur le lien entre la démarche et les outils, ce qui nous conduit à repréciser certaines notions sans cependant revenir sur les aspects plus théoriques ou mathématiques.

Les deux chapitres suivant abordent les usages avancés, d'abord les traitements statistiques avancés rencontrés usuellement en SHS puis d'autres traitements moins habituels comme l'analyse textuelle et l'analyse de réseaux et la collecte de données sur internet.

---

5. Nous vous conseillons de suivre les différents chapitres pour ensuite revenir sur ceux qui vous semblent les plus importants pour vous. En effet, certaines notions utilisées sont présentées dans les chapitres précédents.

Enfin, le dernier chapitre fait un rapide retour sur l'organisation du code et un complément en annexe revient sur les étapes pratiques de la programmation. Un index des notions et des fonctions vous permet de revenir sur les notions vues dans le manuel.

De nombreux exemples permettent de rendre les notions présentées plus concrètes. Nous vous recommandons fortement de les reproduire sur votre ordinateur et de faire varier leur structure pour vous approprier *expérimentalement*, par essais et erreurs, l'écriture du code. Nous pensons qu'il est important que vous écriviez vous-même du code. Ces exemples sont de deux types : des petits exemples d'illustration pour comprendre les étapes et l'utilisation de données réelles pour montrer la mise en œuvre concrète. Pour des raisons d'accès, les données utilisées sont des données publiques de *l'Insee*<sup>6</sup> et en particulier celles portant de la population française.

## Information

### Les données utilisées dans le manuel

Vous trouverez sur le site du manuel à l'adresse <http://pyshs.fr> l'ensemble des fichiers utilisés si vous souhaitez reproduire les manipulations, ce qui est important pour l'apprentissage. Ces données sont aussi disponibles sur le site institutionnel <http://www.insee.fr>. Pour la première partie de l'ouvrage, nous avons regroupé les données à l'échelle du département.

Enfin, chaque chapitre contient des exercices. Ils servent à appliquer directement les approches présentées et nous ne saurions que trop vous conseiller de les réaliser. Certains présentent aussi la solution tandis que d'autres sont laissés ouverts.

Une remarque pour finir : comme le livre est en noir et blanc, les figures doivent l'être aussi. Le rendu de votre côté sera plus joli. Cela doit être une motivation supplémentaire à reproduire les traitements.

## Ce que ce manuel n'est pas

Les ressources sur la programmation sont à la fois multiples et périssables, au gré des évolutions, des modes et des innovations. Des documentations très riches existent sur les aspects techniques de Python, des livres de *recettes* permettent de trouver la belle manière d'écrire certains codes et de très nombreux livres permettent de débuter la programmation en général.

Pour ne pas se tromper, il est important de savoir que ce manuel n'est pas :

---

6. Institut national de la statistique et des études économiques, organisme producteur de données sur de nombreux aspects de la population française.

- ⊕ un ouvrage de référence sur le langage Python : de très bon manuels existent par ailleurs ;
- ⊕ une présentation de la « belle » manière d'écrire du code : vous trouverez plus esthétique ailleurs ;
- ⊕ une entreprise exhaustive proposant de dire ce qu'est Python pour les SHS : chacun adaptera les outils à ses usages ;
- ⊕ un manuel de statistiques classiques avec Python : les statistiques sont présentées comme une partie du traitement de données ;
- ⊕ une nouvelle manière de mener des enquêtes en SHS : la programmation a pour vocation de trouver sa place dans des pratiques existantes.

Aucun livre n'est parfait et celui-ci ne déroge pas à la règle. Pour les besoins de la rédaction, nous tendons à considérer les SHS comme un ensemble cohérent, ce qu'elles ne sont pas. Par ailleurs, le propos est largement construit sur la base d'applications en recherche. Nous faisons cependant le pari d'avoir rassemblé suffisamment d'exemples pour intéresser un·e géographe devant nettoyer des données dans le cadre d'un bureau d'étude, un·e historien·ne travaillant sur un grand corpus numérisé ou un·e chargé·e d'enquêtes dans un institut de sondage.

## Quelques mots sur les auteurs

**Émilien Schultz** : chercheur en sociologie avec une formation initiale en physique appliquée, j'utilise Python pour réaliser le traitement de données dans mes travaux menés en collaboration interdisciplinaire autour des politiques scientifiques et des pratiques de recherche. Mon objectif est de créer des ponts entre des domaines différents pour favoriser les échanges méthodologiques.

**Matthias Bussonnier** : docteur en biophysique et agrégé de physique, j'ai reçu le prix *ACM Software System* pour ma participation depuis 2011 au développement de *IPython* et du projet *Jupyter* avec l'université de Berkeley. Je développe actuellement des solutions avec Python tout en participant à l'amélioration de grands nombres de projets. Mon objectif est de favoriser le développement de la communauté Python et d'améliorer l'utilisation des outils de programmation pour les sciences.

# Sommaire

<b>Remerciements</b>	<b>i</b>
<b>Prologue</b>	<b>iii</b>
<b>1 Python en sciences humaines et sociales</b>	<b>1</b>
1.1 Pourquoi programmer et pourquoi Python ? . . . . .	1
1.2 Les SHS et la programmation . . . . .	2
1.3 Quelle place pour la programmation en SHS ? . . . . .	4
1.4 Les spécificités de Python par rapport à d'autres langages . . . . .	6
1.5 Installation de Python . . . . .	10
1.6 Résumé du chapitre . . . . .	12
<b>2 Se familiariser à Python</b>	<b>15</b>
2.1 Penser comme un ordinateur . . . . .	15
2.2 Quelques conseils pour débuter . . . . .	17
2.3 Où faire du Python ? . . . . .	18
2.3.1 Différentes possibilités pour parler à l'ordinateur . . . . .	18
2.3.2 Programmer dans une console . . . . .	19
2.3.3 Dans votre navigateur internet . . . . .	21
2.3.4 Les environnements de développement intégrés . . . . .	21
2.4 Écrire notre première ligne de code . . . . .	22
2.4.1 Utiliser le Notebook Jupyter . . . . .	22
2.4.2 Cellules de codes dans ce livre . . . . .	26

2.4.3	Écrire un script dans un fichier . . . . .	26
2.4.4	Écrire un script un peu plus long . . . . .	28
2.5	Faire des erreurs . . . . .	29
2.5.1	Ne paniquez pas . . . . .	29
2.5.2	Trouver de l'aide . . . . .	31
2.6	Créer et manipuler des variables . . . . .	32
2.6.1	Qu'est-ce qu'une variable ? . . . . .	32
2.6.2	Le type des variables . . . . .	35
2.6.3	Les textes ou chaînes de caractères . . . . .	36
2.6.4	Les ensembles : listes et dictionnaires . . . . .	38
2.6.5	Conditions et comparaisons . . . . .	43
2.6.6	Encore plus de types . . . . .	44
2.6.7	#ViveLesCommentaires . . . . .	45
2.7	Les blocs structurants : conditions, boucles, fonctions . . . . .	46
2.7.1	Si [condition] alors [faire] . . . . .	47
2.7.2	Les boucles pour répéter les opérations . . . . .	49
2.7.3	Construire ses fonctions . . . . .	51
2.7.4	Une multitude de fonctions . . . . .	55
2.7.5	Un mot sur l'espace des noms . . . . .	57
2.8	Aller plus loin pour gérer les erreurs . . . . .	59
2.9	Synthèse du chapitre . . . . .	60
<b>3</b>	<b>Les bibliothèques</b> . . . . .	<b>61</b>
3.1	Qu'est-ce qu'une bibliothèque ? . . . . .	61
3.2	Un écosystème . . . . .	62
3.3	Utiliser des bibliothèques . . . . .	65
3.3.1	Charger un module déjà installé . . . . .	65
3.3.2	Installer une nouvelle bibliothèque . . . . .	67
3.3.3	Quelques problèmes que vous pouvez rencontrer . . . . .	68
3.4	Mise en pratique . . . . .	69
3.4.1	Définir ses besoins, chercher l'information . . . . .	69

3.4.2 Utiliser une nouvelle bibliothèque . . . . .	71
3.5 Les évolutions des bibliothèques . . . . .	74
3.6 Synthèse du chapitre . . . . .	76
<b>4 Manipuler les données</b>	<b>77</b>
4.1 Pourquoi manipuler des données ? . . . . .	77
4.2 Charger et sauvegarder des fichiers . . . . .	78
4.2.1 Quelques mots sur les fichiers et leurs formats . . . . .	79
4.2.2 Ouvrir un fichier . . . . .	80
4.2.3 Lire des informations dans un fichier . . . . .	81
4.2.4 Écrire dans un fichier texte . . . . .	82
4.2.5 Ouvrir d'autres fichiers . . . . .	84
4.3 Manipuler les ensembles d'éléments . . . . .	85
4.3.1 Rappel sur les listes . . . . .	85
4.3.2 Les listes emboîtées . . . . .	87
4.4 Traiter du texte . . . . .	89
4.4.1 Il faut qu'on parle... d'encodage . . . . .	89
4.4.2 Mettre en forme le texte . . . . .	90
4.4.3 Intégrer des variables . . . . .	91
4.4.4 Traiter des données réelles . . . . .	92
4.4.5 Introduction aux expressions régulières . . . . .	95
4.5 Complément : attention aux attributions . . . . .	97
4.6 Synthèse du chapitre . . . . .	99
<b>5 Traitement de données avec <i>Pandas</i></b>	<b>101</b>
5.1 La bibliothèque <i>Pandas</i> . . . . .	101
5.2 Charger des données à partir d'un fichier . . . . .	103
5.3 Le format des données <i>Pandas</i> . . . . .	104
5.4 Modifier un tableau <i>Pandas</i> . . . . .	107
5.4.1 Sélectionner des colonnes . . . . .	107
5.4.2 Manipuler l'index . . . . .	108
5.4.3 Renommer les colonnes . . . . .	109

5.4.4	Créer une nouvelle colonne . . . . .	110
5.4.5	Modifier une case du tableau . . . . .	110
5.4.6	Supprimer des lignes ou des colonnes . . . . .	111
5.4.7	Répéter une opération sur un tableau . . . . .	111
5.4.8	Gérer les valeurs manquantes . . . . .	116
5.5	Filtrer des informations . . . . .	116
5.6	Relier des informations de tableaux différents . . . . .	118
5.6.1	Passer par un dictionnaire . . . . .	119
5.6.2	Joindre des tableaux . . . . .	121
5.7	Créer un tableau <i>Pandas</i> . . . . .	122
5.8	Sauvegarder ses données . . . . .	123
5.9	Synthèse du chapitre . . . . .	125
<b>6</b>	<b>Statistiques descriptives et inférentielles</b> . . . . .	<b>127</b>
6.1	Passer de la programmation aux statistiques . . . . .	127
6.2	Construire et nettoyer ses tableaux . . . . .	129
6.2.1	Le tableau de données . . . . .	129
6.2.2	Vérifier la qualité des données . . . . .	130
6.2.3	Sélectionner les données utiles . . . . .	131
6.3	Recoder les variables . . . . .	132
6.3.1	Construire des indicateurs . . . . .	132
6.3.2	Recodage des variables numériques en intervalles . . . . .	136
6.4	Analyse univariée des données . . . . .	137
6.4.1	Le tri à plat des variables . . . . .	137
6.4.2	Les indicateurs de tendance centrale et de dispersion . . . . .	140
6.5	Relation dans les données . . . . .	146
6.5.1	Construire des tableaux croisés . . . . .	146
6.5.2	Les corrélations . . . . .	149
6.5.3	Grouper des catégories . . . . .	151
6.6	Réaliser des tests statistiques . . . . .	152
6.6.1	Quelques rappels . . . . .	152

6.6.2	Intervalle de confiance . . . . .	154
6.6.3	Significativité d'un tableau croisé . . . . .	157
6.6.4	Le V de Cramer . . . . .	158
6.6.5	Déférence entre deux groupes . . . . .	159
6.7	Synthèse du chapitre . . . . .	161
<b>7</b>	<b>Visualiser</b>	<b>163</b>
7.1	Pourquoi programmer des visualisations ? . . . . .	163
7.2	Petit tour des lieux . . . . .	165
7.2.1	Représenter les données avec <i>Pandas</i> . . . . .	166
7.2.2	<i>Matplotlib</i> , une bibliothèque de bas niveau . . . . .	167
7.2.3	<i>Seaborn</i> , une bibliothèque de haut niveau . . . . .	168
7.3	La grammaire des visualisations de <i>Matplotlib</i> . . . . .	169
7.3.1	Quelques éléments généraux . . . . .	169
7.3.2	Définir une figure . . . . .	170
7.3.3	Les différents types de figures . . . . .	173
7.3.4	Tracer plusieurs figures . . . . .	180
7.3.5	Couleurs et légendes . . . . .	186
7.3.6	Sauvegarder une figure . . . . .	187
7.3.7	Affiner une visualisation . . . . .	188
7.4	Visualiser avec <i>Pandas</i> . . . . .	190
7.4.1	Les différents types de visualisation . . . . .	190
7.4.2	Combiner <i>Pandas</i> et <i>Matplotlib</i> . . . . .	190
7.4.3	Combiner tableaux et visualisations . . . . .	191
7.5	Utiliser la bibliothèque de visualisation avancée <i>Seaborn</i> . . . . .	193
7.5.1	Quelques visualisations utiles . . . . .	193
7.5.2	Stylisation des figures avec les thèmes . . . . .	197
7.6	Aller plus loin dans la maîtrise du visuel . . . . .	197
7.6.1	Séparer exploration et finalisation . . . . .	197
7.6.2	Exemple d'une figure complexe : le <i>bubble chart</i> . . . . .	198
7.6.3	Autres visualisations . . . . .	200

7.7	Synthèse du chapitre . . . . .	201
<b>8</b>	<b>Statistiques avancées</b>	<b>203</b>
8.1	Les traitements avancés en Python . . . . .	203
8.2	Les analyses factorielles . . . . .	204
8.2.1	Une grande famille . . . . .	204
8.2.2	Analyse en composantes principales . . . . .	206
8.2.3	Analyse des correspondances multiples . . . . .	213
8.2.4	Analyse factorielle des correspondances . . . . .	217
8.3	Les modèles de régression . . . . .	219
8.3.1	Quelques éléments sur la modélisation . . . . .	219
8.3.2	Régressions linéaires . . . . .	220
8.3.3	Régressions logistiques . . . . .	222
8.4	Les classifications . . . . .	228
8.4.1	Quelques éléments généraux . . . . .	228
8.4.2	Partitionnement en k-moyennes . . . . .	228
8.4.3	Classification hiérarchique ascendante . . . . .	231
8.5	Les modèles d'apprentissage automatique . . . . .	234
8.5.1	Quelques éléments généraux . . . . .	234
8.5.2	Régression logistique pour prédire . . . . .	235
8.6	Synthèse du chapitre . . . . .	238
<b>9</b>	<b>Usages avancés</b>	<b>239</b>
9.1	Récupérer des données complexes . . . . .	239
9.1.1	Transformer du matériau qualitatif en données . . . . .	239
9.1.2	Récupérer des données sur internet . . . . .	246
9.1.3	Utiliser des API : Twitter, Google et les autres . . . . .	252
9.2	Analyse de corpus textuels . . . . .	260
9.2.1	Traiter des textes pour en dégager de l'information . . . . .	260
9.2.2	L'information relationnelle dans les réseaux . . . . .	275
9.3	Visualiser des données géographiques . . . . .	283
9.3.1	La construction de cartes . . . . .	283

9.3.2	Introduction à <i>GeoPandas</i> . . . . .	284
9.3.3	La carte des populations françaises . . . . .	288
9.4	Synthèse du chapitre . . . . .	292
<b>10</b>	<b>Organiser son code et le partager</b>	<b>293</b>
10.1	De la ligne au logiciel . . . . .	293
10.2	Organiser son script dans le Notebook . . . . .	295
10.3	Retravailler une visualisation . . . . .	296
10.3.1	Identifier les éléments génériques dans un code . . . . .	296
10.3.2	Aller vers une fonction . . . . .	299
10.3.3	Un fichier à soi : créer un module dédié . . . . .	302
10.4	Créer une bibliothèque . . . . .	304
10.5	Collaborer autour du code . . . . .	306
10.6	Synthèse du chapitre . . . . .	306
<b>Conclusion</b>		<b>307</b>
<b>A</b>	<b>Annexe : les coulisses du code</b>	<b>311</b>
A.1	Règles à avoir en tête . . . . .	311
A.2	Écrire un code pas-à-pas . . . . .	312
A.2.1	Identifier un objectif . . . . .	313
A.2.2	Procéder par étapes . . . . .	313
A.2.3	Généraliser vers une fonction . . . . .	321
A.2.4	Documentation . . . . .	322
A.3	Synthèse . . . . .	323
<b>B</b>	<b>Ressources utiles pour l'apprentissage</b>	<b>325</b>
<b>Bibliographie</b>		<b>327</b>
<b>Index des fonctions</b>		<b>331</b>
<b>Index</b>		<b>333</b>



# Chapitre 1

## Python en sciences humaines et sociales

Pourquoi s'initier à un langage de programmation quand on fait des sciences humaines et sociales (SHS)<sup>1</sup>? Pourquoi, moi, sociologue, géographe, historien ou anthropologue, je prendrais un temps toujours trop rare pour apprendre à programmer? Non seulement programmer permet de réaliser les traitements courants, comme les statistiques ou le recodage de données existantes, mais ouvre la possibilité de nouvelles stratégies comme la collecte d'informations sur internet ou l'automatisation de certaines tâches.

### 1.1 Pourquoi programmer et pourquoi Python ?

Plusieurs réponses peuvent être données à la question de « pourquoi apprendre à programmer en Python ? »

Un conseiller d'orientation pourrait vous répondre que les métiers d'avenir dépendent de l'informatique, que les employeurs sont friands de cette compétence et que Python est passé premier dans les langages demandés par les recruteurs. Cette réponse n'est qu'en partie satisfaisante. En effet, vous savez qu'il existe des logiciels dédiés pour les analyses que vous faites habituellement. Et puis, ce n'est pas votre métier à proprement parler.

Votre amie informaticienne pourrait vous répondre que le langage Python est esthétique, puissant, intuitif. Que grâce à ce langage, vous allez comprendre la logique derrière les logiciels que vous utilisez. Il n'est pas certain que cela suffise

---

1. Ce chapitre ne contient aucune ligne de code.

à vous convaincre. Pourquoi auriez-vous besoin de programmer, que ce soit en Python ou dans un autre langage, si les outils existent déjà ? Vous avez très bien vécu sans jusqu'à présent.

### ❶ Information

#### Programmer

Programmer désigne le fait de donner une série d'instructions à un ordinateur dans le but de réaliser une tâche. Un langage de programmation est alors la manière d'écrire ces instructions qui sont rassemblées dans un ensemble qualifié suivant les situations d'*algorithme*, de *programme* ou de *script*.

En fait, nous devons répondre à deux questions distinctes :

1. Que gagneraient les SHS à se familiariser avec la programmation ?
2. Pourquoi apprendre Python plutôt qu'un autre langage<sup>2</sup> ?

## 1.2 Les SHS et la programmation

Le développement des sciences des données<sup>3</sup>, souvent mentionnées sous le terme anglophone de *data science*, rend visible une tendance existante depuis plusieurs décennies : la place centrale du traitement informatisé des données, transversale à presque tous les domaines.



Fig. 1.1 – L'usage de la notion de science des données

D'activité secondaire, le traitement de données est devenu un enjeu à part entière. De la finance aux mouvements des données citoyennes<sup>4</sup> en passant par la bureautique quotidienne, la croissance des données numériques et le traitement assisté

- 
2. Les habitués penseront évidemment à R, sur lequel nous reviendrons.
  3. Que nous mettons volontairement au pluriel compte-tenu de l'hétérogénéité des pratiques regroupées sous ce terme.
  4. Le développement des *hackathons* autour de mobilisations sociales est ainsi significative.

par ordinateur de l'information sont devenus indispensables. Même dans les secteurs les plus éloignés de l'informatique, le traitement numérique des données se développe. En littérature par exemple se développent les humanités numériques<sup>5</sup>. La *digitalisation*<sup>6</sup> du monde rend les compétences informatiques centrales dans de très nombreux métiers et le métier de *data scientist* (spécialiste des données) est apparu dans la liste des postes recherchés par les entreprises<sup>7</sup>.

## ❶ Information

### Script, code et programme

Le *code* désigne des commandes exprimées dans un langage. La notion de *script* est utilisée dans le domaine de la programmation pour désigner l'ensemble des commandes qui va organiser une action de l'ordinateur, de manière similaire à un script de théâtre où chaque action est décrite. On dit qu'on *exécute* un script pour dire qu'on demande à ce que les commandes qui le composent soient réalisées. Les *programmes* sont des ensembles plus structurés orientés vers la réalisation de certaines tâches.

Deux types de développement ont accompagné la croissance de ce monde numérique. Le premier repose sur l'utilisation de compétences informatiques générales comme la connaissance du fonctionnement des ordinateurs et la programmation. Les praticiens de nombreux secteurs se sont formés à ces approches et de nouvelles spécialités sont apparues. Le second repose sur l'utilisation de logiciels dédiés, développés par quelques personnes ou entreprises au fil des années. Il n'est alors pas nécessaire de maîtriser la programmation pour pouvoir traiter des données.

Au-delà de la spécificité des objets abordés par des méthodes d'enquêtes dédiées, une partie importante de l'activité des SHS consiste à récolter, formaliser, nettoyer, traiter et représenter des données. Elles sont donc directement concernées par la place croissante des traitements informatiques qui est d'abord visible dans les tendances dites *quantitatives*, mais concerne en fait aussi les approches plus *qualitatives*, avec toute la prudence qu'il faut avoir avec cette séparation souvent trompeuse.

Pour le moment, les SHS ont surtout connu le deuxième type de développement, avec la diffusion d'outils spécifiques. Pour ne prendre que des exemples récents<sup>8</sup>, étant entendu que l'histoire des relations entre SHS et automatisation remonte au milieu du vingtième siècle, cela concerne par exemple les traitements statistiques

5. Pour le développement des méthodes numériques en SHS : Bastin & Tubaro (2018).

6. Digitalisation : la multiplication des doigts sur les claviers. À ne pas confondre avec numérisation en bon français, qui signifie passer par des données numérisées.

7. La notion de *data scientist* regroupe des métiers très différents qui ont en commun d'avoir une familiarité avec la programmation et le traitement de données.

8. Vous n'êtes évidemment pas sensé connaître tous ces logiciels, mais il se peut que certains vous soient familiers.

(avec un logiciel comme *SPSS*) ou les traitements de données géographiques (avec un logiciel comme *Cartes & Données*). Notons par exemple les logiciels de traitement des entretiens et de codage (comme *Nvivo*), ou ceux d'analyse lexicométrique (comme *Iramuteq*). D'autres logiciels permettent de faire de l'analyse de réseaux (comme *Gephi*) et des plateformes en ligne permettent d'analyser et de visualiser des grands jeux de données (comme *Cortext*). Les pratiques se sont donc construites autour d'outils plutôt que sur les compétences plus fondamentales de programmation. Seuls certains domaines ont assez tôt mobilisé les outils de la programmation pour construire des simulations<sup>9</sup>.

Cette situation peut s'expliquer par un éloignement en apparence plus grand des approches formalisées et mathématisées, mais aussi tout simplement par une histoire institutionnelle qui a privilégié d'autres approches. Cela ne traduit en rien une incompatibilité de principe. Dans ce contexte, le recours à la programmation existe surtout dans les marges de ces logiciels spécialisés. Avec R, un langage statistique libre et supporté par de nombreux utilisateurs, une partie des SHS s'est initiée à la programmation en codant des scripts pour réaliser les analyses statistiques. De même, des logiciels de cartographie comme *QGIS* permettent aussi d'utiliser le langage Python pour interagir avec les cartes<sup>10</sup>.

Pour autant, la programmation est peu répandue de nos jours<sup>11</sup> et il est rare qu'un étudiant la rencontre dans sa formation. Ce sont souvent des initiatives individuelles qui conduisent certains à s'intéresser à ces approches. Cette situation nous semble améliorable dans la mesure où le numérique devient de plus en plus central non seulement dans le traitement de données mais aussi en tant que terrain d'enquête.

## 1.3 Quelle place pour la programmation en SHS ?

La nature des données que les SHS ont à traiter a largement évolué avec le développement des ordinateurs et d'internet. Que ce soit la veille informationnelle sur internet, l'audit sur les réseaux sociaux, les ethnographies numériques<sup>12</sup>, vous êtes potentiellement amené à traiter des informations issues d'un ordinateur dans des formats variés. Les données collectées sont souvent mal organisées et compliquées : par exemple des formats différents, souvent avec des erreurs. Toutes les opérations de nettoyage peuvent être automatisées par la programmation, surtout si vous devez les répéter régulièrement.

---

9. Cela passe par exemple par des simulations du social Edelmann *et al.* (2020).

10. Notons en passant que le logiciel de statistique *SPSS* a aussi un module pour utiliser le langage Python, qui devient progressivement une langue commune.

11. Cela n'a pas toujours été le cas et ces approches ont pu être très développées dans certains secteurs des SHS, comme la sociométrie ou encore l'archéologie Plutniak (2018).

12. Dans le cas où vous travaillez sur les usages numériques des individus, vous allez être confronté à un terrain d'étude entièrement numérique Marres (2017). Maitriser la programmation vous aidera à vous orienter.

Par ailleurs, les SHS regroupent une pluralité d'approches. Cela signifie la co-existence d'une multitude de données et de traitements nécessitant en regard une diversité d'outils adaptables. Cette souplesse peut être obtenue par l'écriture des scripts dédiés. En adaptant ses propres « recettes de cuisine », il est alors possible de résoudre ses problèmes spécifiques.

En plus de ces deux raisons, une troisième est davantage *pédagogique*. Programmer nécessite la construction d'abstractions et amène à se poser des questions qui ne sont pas directement présentes dans les sujets initiaux, mais qui ont trait à la structure des données et de la démarche. Par exemple, cela amène à réfléchir aux étapes nécessaires pour réaliser une analyse statistique ou faire une recherche d'information dans un document. En cela, la programmation a des affinités avec la logique, les mathématiques, ou la philosophie analytique. Apprendre un langage de programmation est alors une manière douce d'aborder les niveaux d'abstraction.

Enfin, utiliser un langage de programmation générique et libre permet de diffuser facilement ses productions et permettre à d'autres de les utiliser. Si vous analysez des données publiques, vous pouvez ensuite diffuser rapidement à la fois vos résultats et les scripts permettant de les produire, participant ainsi à une conception ouverte de la connaissance (*open science*).

Voici une série de raisons très concrètes de programmer en SHS :

- ⊕ *les solutions existantes ne s'adaptent pas à vos besoins* : les opérations que vous devez réaliser ne sont pas adaptées aux logiciels existants ;
- ⊕ *les données disponibles dont vous disposez ne sont pas dans le bon format* : vous avez besoin d'outils spécifiques pour mettre en forme de manière systématique des données existantes ;
- ⊕ *les données n'existent pas et il faut réussir à les construire à partir d'informations variées* : vous avez besoin de collecter et nettoyer des informations ;
- ⊕ *les logiciels existants sont chers* : ils limitent alors la réutilisation aux utilisateurs ayant une licence du logiciel ;
- ⊕ *il vous faut utiliser plusieurs logiciels pour un même jeu de données et passer d'un logiciel à l'autre peut être pénible* : il vous faut une solution pour lier ces traitements ;
- ⊕ *vous avez envie d'un traitement plus automatique* : par exemple pour arrêter de faire toutes les étapes avec la souris et formaliser les étapes ;
- ⊕ *vous travaillez spécifiquement sur des domaines liés à la programmation, ou avec des collaborateurs qui utilisent de la programmation* : il vous faut acquérir des notions pour échanger avec eux.

Toutes ces raisons peuvent vous amener à apprendre les bases de la programmation. Non seulement il existe des langages adaptés, mais en plus, des communautés d'utilisateurs ont développé les briques de base pour construire des solutions à vos problèmes.

## 1.4 Les spécificités de Python par rapport à d'autres langages

Python est un langage particulièrement adapté pour un usage en SHS, non seulement pour ses traitements classiques mais aussi avec les nouvelles données numériques. Il s'agit d'un langage qui est utilisé et apprécié à la fois par des débutants et des experts en programmation.

Voici quelques opérations facilitées par l'utilisation de Python :

- ⊕ chercher de l'information présente dans plusieurs fichiers (et sur internet) pour les réunir dans un tableau d'analyse ;
- ⊕ faire des traitements *statistiques* et les automatiser ;
- ⊕ extraire de l'information de fichiers de formats différents ;
- ⊕ faire des *visualisations* ou des cartes ;
- ⊕ mettre en forme un fichier pour qu'il puisse être lu par un autre logiciel ;
- ⊕ faire de la *veille* sur des sites ou des réseaux sociaux ;
- ⊕ produire des documents mis en forme comme un sujet d'interrogation avec des questions aléatoires si vous êtes enseignant ;
- ⊕ modifier automatiquement des fichiers en fonction de nouvelles données ;
- ⊕ faire une *interface* avec une base de données spécifiques ;
- ⊕ résumer des informations présentes dans un grand ensemble de textes pour faciliter leur classement ;
- ⊕ automatiser des opérations de collecte ou de vérification de données ;
- ⊕ faire de l'apprentissage automatique sur des grands corpus ;
- ⊕ et pourquoi pas développer des logiciels adaptés à vos besoins, comme une interface d'enquête.

Initialement, Python est un langage de programmation qui se distingue par son caractère pédagogique. Il a été développé pour enseigner la programmation et, pour cette raison, il est intuitif et interactif, sans sacrifier pour autant sa généralité et sa puissance. Guido van Rossum débute la création de Python en décembre 1989. Son nom fait référence à la série comique britannique *Monty Python's Flying Circus*<sup>13</sup> et son écriture le rapproche de l'anglais courant. Sa version 0.9.0 a été publiée en février 1991, sa version 1 en janvier 1994 et sa dernière version 3.8 en octobre 2019<sup>14</sup>.

---

13. Bien que cette référence commence à vieillir, jetez un coup d'œil sur Wikipédia si vous ne connaissez pas.

14. D'ici la publication de ce manuel, Python 3.9 devrait être disponible. Une nouvelle version de Python est publiée en moyenne entre tous les 12 et 18 mois, et est toujours compatible avec la version précédente. Les bibliothèques vont généralement fonctionner avec les versions de Python publiées sur les quatre dernières années. Par exemple, au milieu de 2021, il sera recommandé d'utiliser au minimum Python 3.7.

## Information

### Versions

Comme pour les logiciels, les langages de programmation ont des versions qui marquent leur évolution, avec des modifications de syntaxe ou de fonctions. À l'heure actuelle, c'est la version 3 de Python qui est utilisée, la version 2 étant obsolète, donc *à ne plus utiliser*.

Le langage connaît depuis une adoption massive en lien avec l'essor des sciences des données dans de nombreux secteurs, de l'astrophysique à la finance. Le langage a évolué depuis sa création et s'est enrichi progressivement. Son développement est assuré par une communauté d'utilisateurs de tous horizons qui le font évoluer et construisent des outils spécifiques. En plus d'être pédagogique, Python est un *vrai* langage de programmation. En cela, il se distingue d'autres solutions qui sont des logiciels, dotés d'un langage interne spécifique. Cela permet non seulement un plus grand degré de généralité, mais aussi le développement de logiciels complets et autonomes<sup>15</sup>.

Python a certaines caractéristiques qui le situent parmi tous les langages existants et définissent ses qualités (et défauts). Il est :

- ◆ *procédural*, c'est-à-dire que programmer signifie donner une suite d'instructions à l'ordinateur ;
- ◆ *interprété*, c'est-à-dire que les instructions sont exécutées par l'ordinateur dès qu'elles sont reçues permettant un usage *interactif* ;
- ◆ *orienté objet*, ce qui signifie que l'activité de programmation consiste largement à manipuler des entités (par exemple, un texte) ;
- ◆ *multiplateforme*, c'est-à-dire que vous pouvez l'utiliser sous Windows, Linux ou Mac et que vos scripts seront compatibles ;
- ◆ *libre et gratuit* !

Actuellement, les principaux usages de Python à travers le monde sont le développement de logiciels, le développement d'infrastructures internet, l'analyse et la visualisation de données, l'analyse scientifique et le calcul numérique, l'écriture de script d'automatisation et aussi la pédagogie de la programmation, car le langage est didactique par conception. Ces usages exploitent différents aspects du langage. Si vous n'avez jamais fait de programmation, Python est le bon langage pour débuter. Cela va vous permettre de développer votre capacité à penser de manière abstraite aux opérations que vous réalisez avec un ordinateur et au traitement de vos données<sup>16</sup>. En effet, Python intègre les principaux concepts modernes de la programmation et s'adapte facilement à un usage dans les SHS.

15. Initialement, le logiciel *Dropbox* était développé en Python.

16. Python est largement adopté à travers le monde pour familiariser les élèves et les étudiants à la programmation et à la littératie informatique. Il est placé dans les principaux (si ce n'est le premier) des langages de programmation.

Un aspect important à retenir sur Python, comme R d'ailleurs, est le fait que le langage est interprété. Il est interactif et nous pouvons l'utiliser comme un langage dans une discussion, une phrase après une autre. Cela permet d'écrire des programmes morceau par morceau. Si nous écrivons une phrase incompréhensible, l'interprète nous dit « erreur » au moment où nous nous trompons, ce qui permet de bien identifier l'étape que l'ordinateur ne comprend pas. Python encourage donc l'expérimentation, ce qui facilite aussi l'apprentissage.

En fait, ce manuel s'inscrit résolument dans une perspective générale de « programmation interactive » (*interactive programming*) qui est d'une certaine manière d'échanger avec l'ordinateur sur les données en modifiant progressivement les scripts plutôt que d'écrire un programme figé qu'on exécute ensuite pour réaliser une tâche. En cela, la philosophie pratique que nous développons ici s'éloigne d'une conception statique de la programmation.

## Information

### R, Python, que choisir ? Pourquoi choisir ?

R a débuté comme un langage spécifique au traitement statistique développé par John Chambers connaît un très grand succès. Ses utilisateurs ont développé de très nombreux outils y compris en SHS, dépassant très largement les applications statistiques. Ce faisant, la question se pose de choisir entre R et Python (une question similaire peut aussi se poser, dans une moindre mesure, pour d'autres langages).

Plusieurs éléments doivent être pris en compte :

- ✚ Python est un langage générique, R est un langage plus spécifique ;
- ✚ R bénéficie d'une communauté plus ancienne qui a développé de nombreux outils dans une multitude de domaines, celle de Python, plus récente, est encore en train de se développer ;
- ✚ les deux langages ont un coût d'apprentissage similaire ;
- ✚ leur synthaxe est différente.

Au final, si vous connaissez déjà bien R, peut-être qu'apprendre Python n'est pas une priorité. Un élément central qui doit peser dans votre choix est la possibilité de trouver de l'aide auprès de collègues. Python et R ont une communauté d'utilisateurs très large, mais il est toujours plus facile d'avoir de l'aide d'un collègue proche. Il est aussi possible d'utiliser à la fois R *et* Python (ou d'autres langages dans le même code) mais ceci nécessite des connaissances techniques assez poussées dans les langages de programmation.

Pour aller plus loin avec R : *R pour la statistique et la science des données* Husson *et al.* (2018).

Dans la mesure où il s'agit d'un langage interprété, il est potentiellement plus lent que des langages dits compilés. Ne vous inquiétez cependant pas : des modules spécifiques ont été développés pour contourner ce problème potentiel si vos besoins sont la rapidité : par exemple, des morceaux de code en langage C<sup>17</sup> permettent d'optimiser certains traitements pour aller plus vite. Et Python prend en charge le calcul parallèle<sup>18</sup> et l'utilisation de processeurs graphiques<sup>19</sup> pour les applications sur des *Big Data*. Ce n'est pas pour rien qu'il est largement utilisé actuellement dans le domaine de l'Intelligence Artificielle. Il n'y a donc aucune raison de ne pas se lancer. Surtout que vous ne serez pas seul !

Au final, à quoi va *vraiment* vous servir d'apprendre Python ? Vous avez raison, il est temps de donner un exemple concret.

## Information

### Exemple de traitement

Pour un travail sur le Parlement français, Émilien a besoin de collecter des informations sur des parlementaires. Dans ce cas, nous pouvons exploiter des données existantes sur des sites citoyens qui réunissent ces informations (<https://nosdeputes.fr> par exemple). Ces sites ont une interface qui permet de facilement récupérer des informations par parlementaire, voire même déjà des outils en Python pour les utiliser.

Pour faire le traitement demandé, Émilien crée un fichier avec le nom des parlementaires qui l'intéresse et trouve la manière de noter leur nom pour pouvoir interroger le site avec un lien. Puis avec quelques lignes de code, il récupère les informations disponibles sur ce site et l'enregistre dans des fichiers séparés pour chaque parlementaire.

Ensuite, après avoir regardé le contenu des informations pour chaque parlementaire « à la main », il écrit un script qui ouvre chacun de ces fichiers et extrait les informations qui l'intéressent (les commissions dans lesquelles ils siègent, leur date d'élection initiale, leur activité annuelle, etc.) pour construire un tableau. Après cette étape, il sauvegarde ce tableau dans un fichier Excel.

---

17. C'est un langage ancien et très connu de programmation, considéré comme de « bas niveau », c'est-à-dire proche des opérations réalisées par les ordinateurs et moins compréhensible. D'autres langages sont associés (comme le C++) ou en concurrence (comme le Java).

18. Quand le traitement doit réaliser plusieurs calculs, au lieu d'attendre que le précédent soit fini pour faire le suivant, le calcul parallèle consiste à mener les calculs en même temps.

19. Les processeurs graphiques ou cartes graphiques sont souvent bien plus puissants que les processeurs classiques, mais plus compliqués à programmer et bien plus limités dans les tâches qu'ils peuvent effectuer. Dans certains cas bien choisis, un processeur graphique est mille fois plus rapide qu'un processeur classique.

Les données sont maintenant enregistrées. Ensuite, il ne reste plus qu'à explorer ces informations : regarder l'affiliation politique des parlementaires, leur âge, si ces éléments évoluent dans le temps. Une opération particulière nous intéresse : le lien entre commissions et parlementaires. Il est facile de construire un petit réseau parlementaire/commission pour identifier les parlementaires les plus centraux. Il est alors facile de répondre à la place d'une commission particulière dans le paysage parlementaire. D'autres questions pourraient aussi amener à faire une carte de la France indiquant l'activité de chaque parlementaire en fonction de sa circonscription. À chaque fois, Python peut être utilisé pour obtenir le résultat souhaité.

Nous avons maintenant des morceaux de codes qui permettent de faire ces opérations. S'il fallait refaire le même traitement avec plus de parlementaires, il suffirait d'utiliser le script déjà écrit sur les nouveaux fichiers sauvegardés. Le code est donc à la fois le résultat de l'analyse et un ensemble d'outils qu'il est possible de réutiliser. On peut dire qu'il intègre la démarche même de recherche. Bien sûr, le même résultat aurait pu être obtenu manuellement. Mais non seulement cela aurait été plus long, mais la manière d'envisager le problème aurait potentiellement pu être différente.

Un dernier point avant de présenter comment installer le logiciel nécessaire à écrire votre première ligne de code. Comme pour une langue, un apprentissage nécessite un peu de temps pour vous familiariser avec les bases de Python puis réussir à traduire vos problèmes pratiques dans ses termes. Le temps d'apprentissage de Python est similaire aux autres langages de programmation tout en dépendant largement des individus. Prévoyez une période d'un mois pour vous familiariser tranquillement au langage à travers ce manuel et commencer à l'utiliser<sup>20</sup>. La meilleure manière de progresser est de pratiquer, de faire des erreurs et de trouver des solutions.

## 1.5 Installation de Python

Finissons ce chapitre d'introduction en présentant comment installer Python sur votre ordinateur. Vous pouvez commencer à lire ce manuel sans installer Python et revenir sur cette section plus tard, dans la mesure où le prochain chapitre présente comment débuter à faire du Python sans installation.

---

20. Beaucoup d'usages peuvent être réalisés sans nécessairement maîtriser entièrement le langage, mais en adaptant les exemples. N'hésitez pas à revenir en arrière ou poser ce livre pour quelques semaines si vous vous sentez perdu. Programmer est une compétence qui prend du temps avant d'être maîtrisée.

Pour installer Python, le plus simple est d'utiliser un environnement qui installe tous les éléments nécessaires. Une solution facile d'utilisation est la distribution Anaconda<sup>21</sup> développée par l'entreprise éponyme. Cette solution a été développée pour facilement paramétrier les modules nécessaires à la programmation scientifique. Vous pouvez aussi installer uniquement Python, en allant sur le site [www.python.org](http://www.python.org). Sous Linux, l'interpréteur Python est généralement déjà installé.

En installant Anaconda vous installez non seulement le langage mais aussi des bibliothèques soit directement utiles, soit qui vont accélérer l'utilisation ultérieure. L'avantage d'Anaconda est que l'installation est similaire sur les différents systèmes d'exploitation. De plus, cela permet d'avoir une interface graphique pour lancer les différentes interfaces si vous n'êtes pas familier de la ligne de commande.

Pour l'installation, téléchargez la dernière version d'Anaconda sur <https://www.anaconda.com/products/individual> (la version 3.8 ou ultérieure), et suivez les instructions d'installation. Si vous rencontrez un problème, jetez un coup d'œil à l'aide disponible sur le site d'Anaconda. Et si Anaconda est trop volumineux pour votre ordinateur, une alternative est *Miniconda*, basé sur le même principe et plus léger.

Sous *Windows* et *Mac*, vous pouvez lancer le *navigateur Anaconda* qui permettra de naviguer dans la distribution. Vous avez une interface similaire à celle de la figure 1.2.

Vous pouvez par exemple cliquer sur *Launch* pour lancer le Notebook Jupyter, que nous allons utiliser par la suite. Si tout se passe bien, vous allez voir une fenêtre qui s'ouvre dans votre navigateur. Vous pouvez aussi lancer dans le menu démarrer le terminal « Anaconda Prompt », et lancer Python en tapant `python` puis valider avec entrée.

Sous *Linux*, l'utilisation d'Anaconda se fait en ligne de commande dans un terminal. Pour activer l'environnement de base, lancez la commande `source activate base`. Vous pouvez ensuite lancer une console Python avec `python`. Un environnement plus enrichi, IPython, est accessible avec la commande `ipython`. Le Notebook Jupyter que nous allons voir plus en détail est accessible avec la commande `jupyter notebook`.

### ❶ Information

#### Créer de nouveaux environnements

21. Anciennement *Continuum Analytics* qui a largement contribué au développement *Open Source* de la communauté Python.

Anaconda permet de gérer des environnements différents. Cela est utile quand vous avez envie de séparer différentes versions de Python. Dans l'interface graphique, vous avez un onglet « Environnements » qui permet de voir la boîte dans laquelle vous êtes et les modules qui sont installés. Sous Linux, la gestion se fait par ligne de commande. Par exemple, créer un nouvel environnement portant le nom *environnement2* avec la version 3.7 de Python se fait avec la ligne de commande `conda create -n environnement2 python=3.7`.

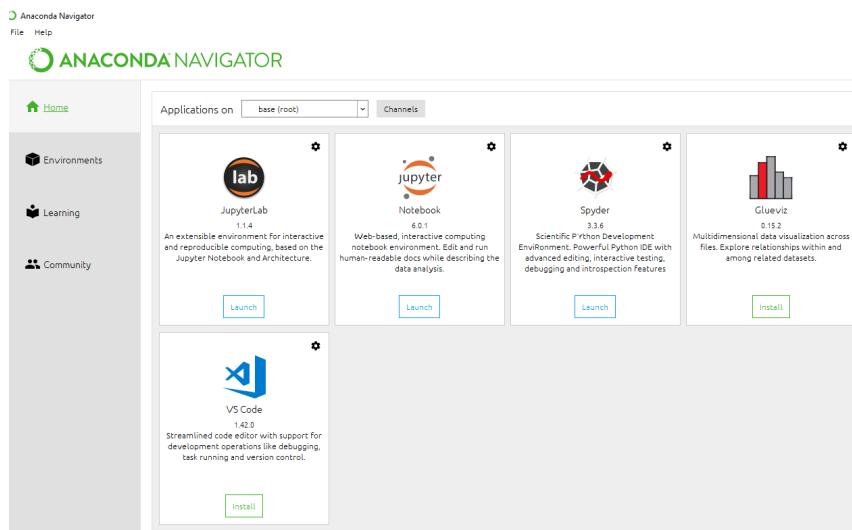


Fig. 1.2 – Interface graphique d’Anaconda

## 1.6 Résumé du chapitre

Apprendre à programmer est une nouvelle corde à votre arc pour faciliter le traitement de données. Cela permet de prendre le contrôle de l'ordinateur de manière plus complète qu'à travers un logiciel et mieux en comprendre les arcanes.

Le langage Python est particulièrement adapté pour s'initier à la logique de programmation en raison de sa clarté, de son caractère générique et des outils déjà existants libres et intercompatibles.

Il ne s'agit cependant ni d'une formule magique qui résout tous les problèmes, ni d'une nouvelle manière de faire votre travail. Néanmoins, prendre le temps de maîtriser les bases de Python vous permettra d'automatiser de nombreuses tâches dans le traitement de données, de faciliter certaines opérations et d'en découvrir de nouvelles.

## Citation

### Entretien avec Guido (créateur du langage) : Python, un langage pédagogique pour tous

*Journaliste* : [Le langage] C++ est une chose effrayante à présenter à quelqu'un qui ne sait pas comment programmer !

*Guido* : Les enseignants que j'ai pu croiser jusqu'à maintenant trouvent effectivement que c'est un désastre à enseigner [...] Alors un langage comme Python, qui a ses racines dans les langages pédagogiques, réussira mieux.

Python est très fortement inspiré par un langage appelé ABC ; j'ai travaillé sur l'implémentation d'ABC créé par des collègues au début des années 80. C'était un langage merveilleux pour enseigner. L'histoire de Python vient de cette frustration que j'ai rencontrée avec ce langage dans des utilisations en dehors de l'enseignement, pour programmer au quotidien. Mais ça c'est une autre histoire.

Python hérite de beaucoup de cette histoire qui le rend très simple, facile à comprendre, facile à retenir et facile à utiliser. C'est un très bon langage pour commencer à enseigner [...] L'autre élément, qui est encore plus excitant, concerne les outils. Si vous regardez un environnement de programmation typique, que ce soit Java ou C++ ou Visual Basic, ils sont tous très compliqués parce qu'ils sont conçus pour des développeurs professionnels qui ont besoin de beaucoup d'éléments [...] Nous avons donc conçu des outils pour aider, et nous nous sommes concentrés sur les programmeurs inexpérimentés. Les outils vont leur donner beaucoup de prises, beaucoup d'aperçus dans le code existant et vont aider à avoir une idée sur les effets des scripts écrits.

Entretien complet dans Linux Journal, traduction par les auteurs.



# Chapitre 2

## Se familiariser à Python

Apprendre un langage de programmation ressemble à l'apprentissage d'une nouvelle langue : cela nécessite de se familiariser avec un nouveau rythme, une nouvelle syntaxe mais aussi de mémoriser le vocabulaire de base. Et comme pour toute langue, la meilleure manière d'apprendre est de se plonger dans sa culture.

### 2.1 Penser comme un ordinateur

Programmer signifie donner des instructions à un ordinateur. En tant qu'utilisateur, nous voulons réaliser certaines opérations. Il nous faut les formuler de telle manière que l'ordinateur comprenne les instructions et puisse les réaliser. Ces instructions doivent respecter un langage, généralement écrit, qui s'appuie sur des opérations élémentaires. Celles-ci peuvent être combinées pour réaliser des actions plus complexes.

La différence principale entre langage de programmation et langue de communication tient au fait que le temps de la programmation n'est pas celui d'un dialogue. Vous pouvez prendre votre temps, consulter un manuel et faire autant d'essais que vous le souhaitez pour trouver les bons mots. Et puis, non seulement Python a été pensé pour vous faciliter la vie, mais en plus le vocabulaire qui le compose est limité et assez facile à apprendre<sup>1</sup>. Toutefois, avant de nous intéresser à un langage particulier, il nous faut clarifier ce que signifie parler à un ordinateur.

---

1. Disons-le dès maintenant : ne cherchez pas à apprendre tout par cœur du premier coup. Nous allons régulièrement rencontrer les mêmes notions et en les utilisant vous allez progressivement vous les approprier. Même après de nombreuses années, il nous arrive à devoir « chercher nos mots ».

Pour prendre un exemple simple, imaginons que vous voulez calculer la moyenne entre trois notes d'un questionnaire de satisfaction. Ces nombres ont déjà été collectés et inscrits dans un fichier sur votre ordinateur comprenant trois colonnes, avec une ligne par répondant. Puisque l'ordinateur ne comprend qu'un certain nombre d'opérations déterminées à l'avance, comme `ouvrir un fichier`, `lire un nombre`, `additionner un nombre`, vous ne pouvez pas simplement lui dire `calcule-moi la moyenne pour chaque ligne`. Vous allez devoir programmer votre opération à partir de ces briques élémentaires.

Par exemple, vous allez lui dire :

- ➊ ouvre le fichier ;
- ➋ prends une ligne ;
- ➌ additionne les trois éléments ;
- ➍ divise cette somme par trois ;
- ➎ garde en mémoire l'information de la ligne et la moyenne ;
- ➏ recommence avec la ligne suivante autant de fois qu'il y a de lignes ;
- ➐ sauvegarde cette information dans un fichier de l'ordinateur.

Forcément, la manière d'écrire ces opérations dépend du langage de programmation. Certains langages permettent de dire directement `calcule une moyenne` tandis que d'autres ne peuvent faire que des additions et des soustractions et il faut décomposer l'opération en sous-opérations. Résoudre un problème est alors dépendant du langage et vous amène à penser vos actions dans sa logique. Pour le même résultat, vous n'allez pas écrire les étapes de la même manière. Cela nous amène à réfléchir de manière formelle aux étapes qui forment notre code.

Écrire un algorithme rend visible des étapes qui sont sinon implicites quand l'opération est faite par un humain. Au début, cela peut donner l'impression de se compliquer inutilement la vie. Au sujet de l'exemple précédent, des problèmes peuvent apparaître comme des notes mal écrites ou absentes, que nous corrigions sans nous en rendre compte quand nous calculons à la main, mais qui doivent être explicitement pris en compte dans la programmation.

Vous pouvez à ce moment vous interroger sur l'intérêt de programmer votre opération plutôt que de faire directement le calcul avec votre calculette. Mais imaginez que vous ayez à refaire plusieurs fois la même opération, des centaines de fois, ou généraliser de trois notes à une centaine. Le faire à la main sera compliqué, tandis qu'il suffira de modifier une ligne sur votre script pour obtenir votre résultat. Un langage de programmation vous donne donc la possibilité de traduire un objectif dans des instructions compréhensibles par un ordinateur et ainsi de l'automatiser. Apprendre le langage conduit ainsi à formaliser votre pensée et à l'exprimer dans de nouveaux termes.

## 2.2 Quelques conseils pour débuter

Nous tenons à vous dire quelques mots pour faciliter cet apprentissage. Déjà, nous devons vous prévenir : maîtriser Python demande un peu de temps. Et comme pour toute langue, plus vous pratiquez, plus vous vous approprierez sa logique et ses possibilités. Ne soyez pas découragé au début de chercher vos « mots » : c'est normal. Et le manuel est fait pour pouvoir revenir sur les exemples.

Ensuite, une différence importante avec une langue est que vous parlez à un ordinateur, donc tous les détails vont jouer : si vous oubliez de fermer une parenthèse, ou si vous oubliez une lettre dans une commande, vous allez rencontrer un message d'erreur. À la différence d'une langue, vous ne pouvez pas oublier une préposition. Une virgule peut tout changer. Rappelez-vous que tout est important pour un ordinateur. En pratiquant, vous allez progressivement développer une habitude de rigueur qui permettra d'éviter certaines inattentions. Mais au final, cela arrive toujours et ce n'est pas grave. Nous verrons comment corriger ces erreurs.

Soyez *indulgent* envers vous-même et prévoyez un peu de temps pour intégrer l'information et les tours de main. Vous allez nécessairement rencontrer des petites difficultés et des erreurs longues à résoudre. Acceptez de prendre le temps pour apporter une solution à ces problèmes car ils vous permettront de vous approprier la logique de Python. Nous sommes tous passés par là.

Enfin, quelques derniers conseils avant de vous lancer :

- ➊ prenez le temps de faire les petits exercices d'application ;
- ➋ ne visez pas une compréhension complète du langage avant de commencer à l'utiliser : commencez par faire, en copiant et en modifiant des exemples ;
- ➌ acceptez de faire des erreurs : n'hésitez pas à y aller pas à pas, de vérifier chaque étape. Seul compte le résultat final et les aller-retours sont normaux<sup>2</sup> ;
- ➍ prenez le temps d'écrire les exemples pour les lancer sur votre ordinateur.

Les petits exercices présents dans les chapitres sont importants non seulement parce qu'ils permettent de voir des manières d'écrire du code mais aussi de vous tromper et d'apprendre de vos erreurs. Aussi, au cours de votre lecture et de votre progression, vous allez vous apercevoir que ce que vous avez fait précédemment aurait pu être fait différemment et probablement mieux. Mais rappelez-vous : ce que vous écrivez n'est pas gravé dans le marbre et les lignes de code sont faites pour être modifiées, mises à jour, décortiquées, recomposées. Comme pour un écrivain, il faut bien un premier jet pour avancer dans le roman... Et puis ne vous laissez pas impressionner par l'impression que les codes du livre (ou de vos collègues) sont bien écrits et fonctionnent : vous n'avez pas vu les étapes permettant leur écriture.

---

<sup>2</sup>. Il est rare qu'Émilien écrive un code qui fonctionne du premier coup. Par contre, à la longue, il a appris à corriger les erreurs et à la fin, il fonctionne. Ces étapes intermédiaires sont peu visibles, mais sont nécessaires pour trouver des solutions à des problèmes toujours différents. Vous trouverez un exemple de ces tâtonnements dans l'annexe du manuel.

Désormais, nous sommes prêts à commencer. La première étape est de savoir où donner les instructions à l'ordinateur.

## ❶ Information

### Tous les codes en ligne

Vous pourrez trouver les codes de ce manuel et les explications pour les exécuter en ligne sur le site du manuel <http://pyshs.fr>.

## 2.3 Où faire du Python ?

### 2.3.1 Différentes possibilités pour parler à l'ordinateur

Traditionnellement, les scripts sont rédigés sous la forme d'une série d'instructions dans un fichier texte. Avant cela, ils étaient inscrits sur des cartes perforées (des cartons avec des trous qui servaient à actionner des éléments de l'ordinateur). Ces fichiers sont ensuite lus (on dit qu'ils sont exécutés) par l'ordinateur afin de réaliser les opérations. Cette exécution est possible grâce à un logiciel qui est déjà installé sur l'ordinateur et fait le lien entre les commandes données par l'humain (vous) et l'ordinateur. Ce logiciel traduit en direct ces instructions et les donne à l'ordinateur<sup>3</sup>. Cela peut être un peu perturbant : parler de Python renvoie à la fois au langage (une manière d'écrire) et l'interpréteur (le logiciel installé sur l'ordinateur). Ainsi, installer Python signifie installer le logiciel permettant de faire le lien entre le langage et l'ordinateur<sup>4</sup>.

Pour programmer avec Python, vous pouvez donner ces instructions de plusieurs manières : dans un fichier texte, qui sera lu dans un second temps, de manière interactive, dans une console, ou à travers des interfaces dédiées dont certaines sont accessibles en ligne. Suivant vos usages, vous pouvez choisir la solution la plus adaptée.

Comme nous allons surtout traiter des données, nous allons privilégier les environnements interactifs qui nous permettent de tâtonner, faire des erreurs, adapter petit à petit notre réflexion<sup>5</sup>. Cette interaction va nous permettre de profiter de

---

3. On parle de compilation quand le langage est transformé dans un fichier directement lisible par l'ordinateur et d'interprétation quand un logiciel lit et exécute le langage en temps réel.

4. Vous trouverez à la fin du premier chapitre une manière d'installer le nécessaire pour programmer. Nous vous conseillons si vous êtes sur votre ordinateur d'installer le logiciel *Anaconda* qui est un environnement complet.

5. Cet environnement peut aussi être défini par l'ordinateur sur lequel vous allez travailler. Dans certains cas, peut-être serez-vous obligé de faire un fichier de script, par exemple si vous voulez exécuter à distance votre programme. Certaines données sensibles et les lois européennes peuvent limiter les environnements de programmation à votre disposition.

la souplesse de Python. Par la suite, nous allons utiliser surtout l'interface *Jupyter Notebook* qui est adaptée à un usage interactif. Nous présentons les différentes options pour écrire du code afin que vous ayez une idée générale.

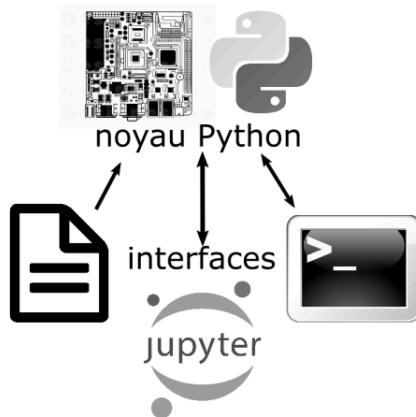


Fig. 2.1 – Relation entre l'ordinateur, le langage et les interfaces

Pour vos premières lignes de code, nous vous conseillons d'utiliser les outils disponibles sur le *cloud* déjà en place avant d'installer Python sur votre ordinateur<sup>6</sup>. Cependant, si vous programmez régulièrement, le plus simple pour faire du Python est de le faire directement sur votre ordinateur. Vos données y sont déjà présentes et vous n'avez pas besoin de connexion internet une fois l'installation effectuée<sup>7</sup>.

### 2.3.2 Programmer dans une console

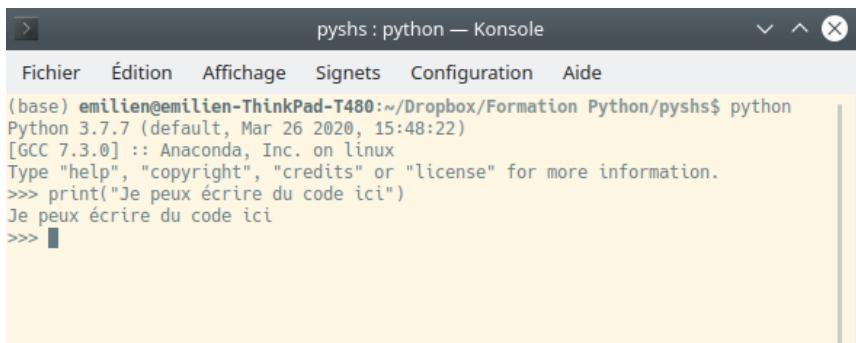
Le *terminal*, ou la *ligne de commande* est souvent l'interface privilégiée des utilisateurs plus avancés. Il est aussi appelé *shell* (coquille), car c'est une fine couche au dessus des capacités de l'ordinateur qui permet de dialoguer avec lui par des instructions. Bien qu'intimidant avec son écran noir avec du texte, la simplicité du terminal et sa puissance ne sont pas à sous-estimer. En effet, vous pouvez dia-

6. Pour vos premiers essais, vous pouvez utiliser des services en ligne permettant d'exécuter du code. Apprendre et écrire du Python sur internet est une des meilleures façons de commencer. Un grand nombre de services vous offrent la possibilité d'exécuter du code en ligne comme <https://mybinder.org>. Il permet de faire tourner gratuitement des codes en ligne avec des ressources et une durée limitée dans la mesure où ce le code et les données sont rendus publics. Après quelques heures, toutes les modifications sont effacées pour laisser la place à l'utilisateur suivant. Plus d'information sur <http://pyshs.fr>.

7. À moins, bien sûr, que votre analyse ait besoin d'accéder à internet.

loguer avec l'ordinateur très rapidement, c'est-à-dire voir le résultat en direct de vos demandes et donc de corriger ou d'adapter vos opérations. Cela est très utile quand vous êtes en train d'essayer différentes options.

L'interpréteur de Python se lance dans un terminal et fonctionne avec des lignes de code<sup>8</sup>.



```
pyshs : python — Konsole
Fichier Édition Affichage Signets Configuration Aide
(base) emilien@emilien-ThinkPad-T480:~/Dropbox/Formations Python/pyshs$ python
Python 3.7.7 (default, Mar 26 2020, 15:48:22)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Je peux écrire du code ici")
Je peux écrire du code ici
>>> █
```

Fig. 2.2 – Le terminal pour écrire du Python

L'interpréteur Python interactif présente cependant quelques limitations dans sa version basique : vous ne pouvez rentrer vos instructions que ligne par ligne, le format des données est uniquement du texte, il ne permet pas d'interagir avec tout l'écran. De plus, l'ensemble est assez austère et ne dispose daucun outil d'accompagnement.

## ❶ Information

### Différents dialectes

En fonction du système d'exploitation (Windows, Mac ou Linux), le terminal n'est pas identique et le dialecte à utiliser pour discuter avec votre ordinateur change (nous attirons votre attention sur le fait que ce n'est pas vraiment un langage de programmation, mais une manière de donner des instructions au système d'exploitation installé sur votre ordinateur).

Sur Linux, c'est souvent le dialecte **bash**, sur mac OS **bash** ou **zsh** et sous Windows **powershell**. Ces dialectes sont similaires mais varient suffisamment pour qu'une commande dans l'un ne fonctionne pas dans l'autre.

---

8. Notez la différence entre le terminal, qui est un dispositif de votre système d'exploitation permettant d'échanger avec votre ordinateur et l'interpréteur Python qui se lance dans un terminal.

Il existe une version améliorée de cet interpréteur : IPython. Ancêtre du Notebook Jupyter présenté par la suite, IPython est un « enrichissement » de Python. N'hésitez pas à installer et utiliser `ipython` plutôt que `python` : il y a des couleurs. Vous pouvez aussi le lancer directement à partir de l'environnement *Anaconda*.

### 2.3.3 Dans votre navigateur internet

Depuis quelques années, vous pouvez programmer dans votre navigateur internet. Cette solution peut paraître surprenante. Mais actuellement il est possible d'avoir le meilleur des deux mondes entre le terminal et le fichier texte avec un petit tour de passe-passe.

Le Notebook Jupyter est né des besoins pratiques des utilisateurs. L'idée derrière le Notebook Jupyter (aussi appelé Notebook ou Jupyter tout court par la suite) est d'utiliser l'environnement d'une page internet, avec son interactivité, pour faire une interface simple, conviviale et persistante avec la couche qui s'occupe d'exécuter le code. En fait, si Python est une couche entre le langage et l'ordinateur, le Notebook est une couche entre l'utilisateur et Python<sup>9</sup>. L'avantage d'utiliser notre navigateur est que celui-ci sait faire un grand nombre de choses : lire des vidéos, afficher des images, jouer de la musique... Et toutes ces possibilités peuvent être utilisées pour interagir avec l'ordinateur. Nous allons largement utiliser Jupyter par la suite. Il est donc important de comprendre comment celui-ci fonctionne<sup>10</sup>.

### 2.3.4 Les environnements de développement intégrés

Enfin, une grande partie du travail de programmation prend généralement place dans un Environnement de Développement Intégré ou *IDE* (pour *Integrated Development Environment*) en anglais. Il s'agit d'un logiciel conçu pour l'écriture de code en regroupant différents outils facilitant le travail. Il comporte souvent un éditeur de texte spécialisé qui « reconnaît » la structure du code en mettant des couleurs et apporte des aides pour la conception, la modification et l'analyse à grande échelle de code.

---

9. Il est possible d'utiliser le Notebook Jupyter avec d'autres langages que Python. Jupyter joue bien un rôle d'interface entre l'utilisateur et le noyau du langage.

10. Les Notebooks Jupyter changent la manière de programmer car vous pouvez mélanger le code. Pour cette raison, cette interface n'est pas la plus adaptée pour faire de la programmation avancée et peut amener à certaines confusions et mauvaises habitudes. Cependant, il permet de répondre à un besoin bien réel pour le traitement de données : expérimenter et partager du code qui n'est pas complètement abouti.

La plupart des IDE sont conçus pour des grands projets de construction de logiciels et peuvent être mal adaptés à notre objectif orienté vers l'écriture de scripts. Cependant, vous pourrez avoir envie à un moment donné de traiter de manière plus systématique vos fichiers de code et d'avoir un environnement plus formalisé. Dans ce cas, nous vous recommandons d'utiliser le logiciel Spyder<sup>11</sup>.

## 2.4 Écrire notre première ligne de code

Concrètement, nous allons utiliser deux principales manières de faire du Python : dans le Notebook Jupyter et dans un fichier. La ligne de code que nous allons utiliser pour la suite est la suivante `print("Au secours, je programme en Python")`, qui utilise le mot clé `print` signifiant « afficher » et qui donne entre les parenthèses un texte entre guillemets à afficher. On appelle *fonctions* ces mots clés qui réalisent une action<sup>12</sup>.

### 2.4.1 Utiliser le Notebook Jupyter

La première étape est de lancer le Notebook Jupyter. Si vous êtes sur votre ordinateur, vous avez deux manières de le lancer : soit vous passez par l'interface *Anaconda* en cliquant sur l'icône Jupyter Notebook, soit vous utilisez votre terminal avec la commande `jupyter notebook`<sup>13</sup>. Une interface apparaît dans votre navigateur internet<sup>14</sup> qui permet de circuler dans vos fichiers et de créer de nouveaux Notebooks. C'est notre point de départ.

Un Notebook est un fichier avec une extension `.ipynb` qui contient à la fois le code que vous avez écrit et le résultat de son exécution, mais aussi du texte que vous pouvez ajouter. Ils peuvent donc être sauvegardés, partagés, voire affichés directement en ligne pour être consultés ou réutilisés. Les données que vous allez charger sont par contre contenues dans des fichiers à part, stockés sur votre ordinateur ou en ligne. Ce premier Notebook contient du code et du texte.

---

11. Spyder est un environnement scientifique écrit en Python, pour Python et conçu par et pour des scientifiques, ingénieurs et analystes de données. Il permet une unique combinaison d'édition avancée du code, d'analyse, de débogage en faisant un outil de développement complet avec les capacités d'exploration de données, d'exécution interactive, d'inspection approfondie et de visualisation d'un progiciel scientifique. Plus d'information sur : <https://www.spyder-ide.org/>.

12. Nous reviendrons sur les fonctions à la fin de ce chapitre. Le texte se met entre des guillemets. Exécuter une fonction signifie l'utiliser en lui donnant les informations entre parenthèses dont elle a besoin pour fonctionner.

13. Attention, pour cela il faut avoir activé votre environnement python. Avec Anaconda, cela signifie avoir entré la ligne `source activate NOM-ENVIRONNEMENT`. Par défaut, vous avez l'environnement `base` déjà mis en place.

14. Dans certains cas, la page ne s'ouvre pas. Vous avez un lien qui s'affiche sur le terminal où vous avez écrit la commande, vous pouvez copier ce lien (voir le *token* de sécurité si besoin) pour ouvrir l'interface.

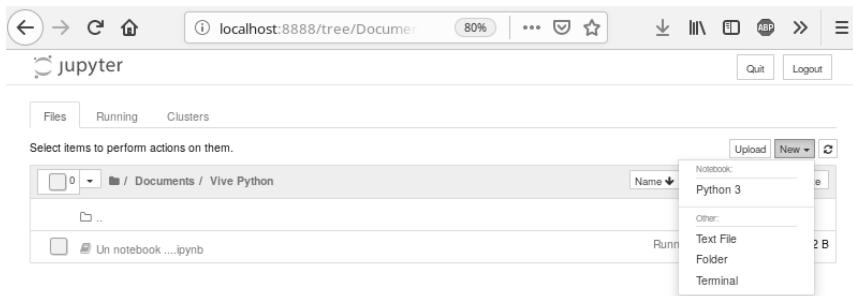


Fig. 2.3 – Page de présentation de Jupyter Notebook

## Exercice

### Créez un nouveau Notebook Jupyter.

Une fois l'interface lancée, cliquez sur le bouton **New** en haut à droite et sélectionner **Python 3** pour créer un nouveau Notebook.

Chaque Notebook est composé de cellules indépendantes qui peuvent contenir des éléments, code ou texte. Une cellule peut être *sélectionnée* (en cliquant sur le côté) ou *éditée* pour ajouter du contenu en double-cliquant dessus. Elle peut soit contenir du *code*, soit être transformée en *zone de texte* si vous voulez l'utiliser pour mettre un titre ou ajouter du contenu. Le code dans notre cas est du Python. Le texte lui est en *markdown* qui permet de mettre en forme du texte brut.

Chaque cellule exécutée donne son résultat en dessous. Le numéro qui s'affiche à gauche correspond à l'ordre d'exécution, il est absent si la cellule n'a pas été exécutée. À chaque fois que vous exéutez la même cellule, ce numéro change.

Chaque cellule peut être utilisée indépendamment et être déplacée (avec les flèches de la barre d'outils). Vous pouvez créer une cellule pour tester un morceau de code ou une idée jusqu'à ce que vous soyez satisfait. Cela vous permet aussi de décomposer votre démarche en plusieurs étapes et de déplacer les éléments en fonction de vos besoins. Par exemple, nous ouvrons souvent une nouvelle cellule pour afficher l'information contenue dans un fichier ou pour faire une visualisation simple, avant de commencer une analyse plus poussée.

La barre de navigation, en haut, permet de réaliser les opérations avec les cellules : lancer le code, la déplacer, etc. Cependant, dans un usage quotidien, les raccourcis sont bien utiles. Jetez-y un coup d'œil, cela rendra la pratique beaucoup plus fluide que d'avoir à cliquer.

## ❶ Information

### La notation markdown

Le *markdown* est une manière de décrire la mise en forme du texte. Par exemple, cela permet d'indiquer si un morceau de texte est en gras, ou en italique. C'est une solution pour mettre en forme du texte. Cela signifie que vous pouvez créer des titres en mettant un # devant (ou plusieurs, pour diminuer la taille, ## ou ###), dire qu'un passage est en italique en mettant des étoiles \* autour, en gras avec deux \*\* ou de créer des listes avec -. Cela vous permet de mettre en forme le contenu de votre Notebook.

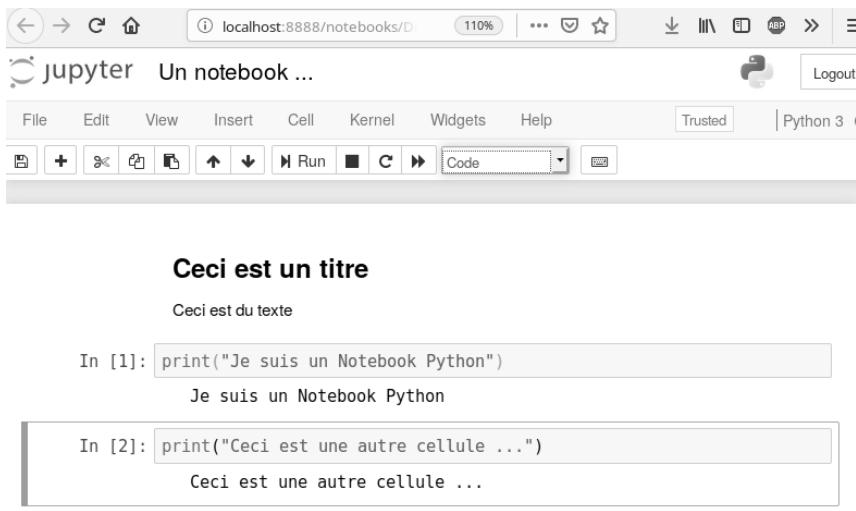


Fig. 2.4 – Un Notebook pour programmer avec ses cellules de texte et de code

Pour exécuter une cellule, on la sélectionne en cliquant dessus ou en naviguant avec les flèches du clavier puis en cliquant sur le bouton **Run** en haut, ou en pressant **Majuscule-Entrer**. Pour l'édition, il faut double-cliquer sur la zone de texte, ou appuyer sur **Entrer**.

L'utilisation de Jupyter est simplifiée par des raccourcis (ils sont présentés dans l'onglet **Help > Keyboard Shortcuts**). Les plus utiles selon nous sont les suivants :

- ➊ Majuscule + enter permet d'exécuter la cellule sélectionnée ;
- ➋ Ctrl + enter ou Cmd + Enter (sous mac) permet d'exécuter la cellule sélectionnée et la garder sélectionnée ;
- ➌ les flèches haut et bas du clavier permettent de se déplacer entre les cellules ;
- ➍ Esc pour quitter le mode édition et revenir au mode sélection ;

- ➊ En mode *sélection*, **b** crée une cellule sous la cellule sélectionnée ;
- ➋ En mode *sélection* deux fois **d** supprime la cellule sélectionnée.

De nombreuses options sont disponibles (manipulation sur les cellules, exporter le document dans d'autres formats), n'hésitez pas à jeter un coup d'œil sur la documentation plus complète de Jupyter Notebook<sup>15</sup>. Prenez le temps de vous familiariser avec cet environnement, ce sera utile pour la suite.

## ❶ Information

### Comment Émilien utilise le Notebook

Dès que j'ai besoin d'utiliser Python, je crée un nouveau Notebook. Par exemple, j'ai besoin de calculer la moyenne de différentes valeurs d'une enquête par questionnaire. Je lance Jupyter et je crée un nouveau Notebook dans le dossier où j'ai mes fichiers de données. Je lui donne un nom explicite et je commence par transformer la première cellule en format texte pour mettre un titre. Je rajoute généralement quelques informations supplémentaires sur mes objectifs pour ne pas oublier ce que j'ai fait d'ici la prochaine fois. Puis je crée une nouvelle cellule qui me permet d'écrire le code pour charger les données qui sont dans un fichier Excel ou CSV. Je crée ensuite une cellule par opération que je veux réaliser : vérifier que toutes les valeurs sont là, calculer les moyennes, faire une visualisation...

À chaque fois, je vais traiter le problème de manière indépendante dans une cellule dédiée. Je garde les résultats affichés et, à la fin, quand j'ai fait toutes les opérations, je réorganise les cellules pour que le document soit plus propre et j'enlève celles qui ne servent à rien.

Dans une cellule du Notebook Jupyter, vous pouvez rentrer votre première ligne de code puis cliquer sur exécuter (*Run*) :

```
print("Je suis une cellule code d'un Notebook Jupyter")
```

**Je suis une cellule code d'un Notebook Jupyter**

Et voilà, vous avez exécuté votre première ligne de code ! Le numéro à gauche qui apparaît sur votre Notebook (et que nous avons enlevé ici) signifie que c'est la première cellule exécutée ici.

---

<sup>15</sup>. La documentation officielle est disponible sur le site du Jupyter Notebook. Vous trouverez beaucoup de tutoriaux et de vidéos sur internet pour aller plus loin.

## 2.4.2 Cellules de codes dans ce livre

Dans la suite de ce livre, nous utiliserons une apparence similaire à celle du Notebook Jupyter<sup>16</sup>. Nous nous sommes aussi efforcés d'écrire du code relativement court. Nous allons souvent à la ligne et souvent nous utilisons « \ » en fin de ligne pour indiquer à Python que la ligne suivante doit être jointe à la ligne courante.

### ?

#### Exercice

**Créez une nouvelle cellule et affichez une phrase uniquement avec les raccourcis.**

Créez une nouvelle cellule avec la touche **b**, puis descendez sur la case avec les flèches du clavier, appuyez sur entrée pour éditer la cellule, écrivez du code, puis utilisez la combinaison **shift+enter** pour exécuter le code.

## 2.4.3 Écrire un script dans un fichier

Dans la situation où vous avez envie d'exécuter régulièrement les mêmes instructions, le Notebook Jupyter n'est pas vraiment adapté. De même, si vous devez travailler à distance sur un ordinateur, vous n'avez pas forcément la possibilité d'avoir un navigateur internet pour afficher le Notebook. La forme historique des instructions informatiques est le programme contenu dans un fichier codant une série d'instructions. Créer un nouveau fichier est facile : il suffit de créer un fichier texte dans lequel vous écrivez ligne par ligne les instructions<sup>17</sup>. Le format utilisé pour décrire les fichiers de scripts Python est **.py**.

### ?

#### Exercice

**Créer un fichier .py qui utilise la fonction print vue précédemment.**

Créez un fichier **programme.py** avec un éditeur de texte, puis écrivez dedans.

```
print("Au secours, je suis enfermé.e dans un fichier")
```

Sauvegardez le fichier.

---

16. En fait, nous avons utilisé des Notebook pour *générer* ce livre. Lorsque vous voyez une cellule Jupyter dans ce livre, celle-ci a été exécutée et son résultat directement intégré dans la page. Le code du livre devrait donc fonctionner car il est constitutif du contenu. Nous avons cependant modifié quelques options de configuration pour que l'apparence du code et des résultats soient adaptés au manuel. Cela peut générer certaines différences par rapport à votre ordinateur. En particulier nous avons retiré le numéro de ligne pour conserver la marge et mis les couleurs de la coloration syntaxique de Python en noir et blanc.

17. Nous vous conseillons d'utiliser *Sublime Text* pour éditer les fichiers textes, puissant et simple, *Atom* ou *Visual Studio Code*. Word, Libre Office et autres logiciels de bureautique ne sont pas adaptés pour éditer un programme.

Le fichier est écrit. Maintenant, il reste à dire à l'ordinateur de l'utiliser pour exécuter les instructions. Vous allez avoir besoin du chemin qui amène au fichier. Par exemple chez Émilien, qui est sous Linux, ce sera `/home/emilien/Documents/programme.py`. Si vous êtes sous Windows, ce sera peut-être chez vous `C:\Documents\programme.py`.

### Information

#### Se repérer dans les chemins de dossiers de votre ordinateur

Que vous utilisiez Windows, Linux ou Mac, chaque fichier ou dossier est situé quelque part dans votre ordinateur. La racine est le point de départ (pour Windows, c'est souvent « C :\ », et pour Unix c'est « / »). Si vous créez un dossier à la racine, son chemin sera « C :\dossier » (Windows) ou « /dossier » (Linux ou Mac). Si vous créez un fichier dans ce dossier, son chemin sera « C :\dossier\fichier » ou « /dossier/fichier ». Ces chemins qui partent de la racine pour aller vers le fichier sont des chemins dits *absolus*.

Souvent vous êtes déjà dans un dossier : votre ordinateur considère que tout se fait à partir de cet endroit (le dossier courant). Avec Jupyter, c'est là où vous avez créé le fichier. Si vous demandez d'accéder à un autre fichier, il regardera le dossier dans lequel vous êtes pour voir si ce fichier existe. À ce moment-là, vous pouvez avancer et reculer à partir de votre position pour aller chercher des fichiers ailleurs. C'est un déplacement *relatif*.

La position actuelle où se trouve votre ordinateur est symbolisée par le point « . ». Si vous voulez dire à l'ordinateur que la position actuelle doit changer pour aller dans un dossier appelé « dossier » contenu dans celui où vous êtes, il faut l'informer que le chemin à suivre est « ./dossier ». Si vous voulez dire à l'ordinateur de revenir dans le dossier parent, la commande est « .. ».

Par exemple, si je suis dans mon dossier « Téléchargements » et que je veux dire à l'ordinateur d'aller dans mon dossier « Documents », cela nécessite de lui dire de revenir en arrière avant d'aller dans le nouveau dossier. Le chemin relatif à indiquer sera « ../Documents », qui signifie : « reviens en arrière, puis va dans le dossier Documents ».

Deux commandes sont importantes à connaître :

- ➊ `cd` pour *change directory*, changer de dossier, qui permet de changer de dossier courant, par exemple « `cd ../Documents` » indique de revenir en arrière et d'aller dans le dossier « Documents » ;
- ➋ `ls` pour *list* (ou `dir` sous Windows), qui permet de lister les éléments présents dans le dossier, par exemple pour le dossier en cours « `ls .` ».

Dernier problème : le caractère « \ » est spécial et pour l'utiliser dans des noms de fichiers, on devra le « doubler » et donc d'écrire le chemin de la manière suivante : « C:\\dossier1\\dossier2\\ » (cela dépend des situations).

Prenez le temps de vous déplacer un peu dans votre ordinateur à partir du terminal. Bien comprendre l'architecture des fichiers dans un ordinateur est nécessaire dès qu'on va devoir charger des données et les sauvegarder. Si vous rencontrez une erreur dans vos chemins de fichiers, essayez de varier l'écriture pour voir laquelle fonctionne chez vous.

Lancez un terminal, puis entrez la commande qui exécutera le script en appelant l'interpréteur Python et le chemin vers le fichier `python /home/emilien/documents/programme.py`. Python va prendre le fichier et exécuter une à une les instructions.

#### 2.4.4 Écrire un script un peu plus long

Dans un Notebook Jupyter, écrivez le code suivant qui permet d'entrer un texte et de compter le nombre de lettres et de mots.

```
entree = input()
nombre_caracteres = len(entree)
nombre_mots = len(entree.split())
print("Le texte a :\n {} caractères\n {} mots"\ \
      .format(nombre_caracteres,nombre_mots))
```

Le texte a :  
52 caractères  
10 mots

Ce script utilise uniquement les propriétés de base de Python pour récupérer une information de votre clavier, puis afficher le résultat du calcul sur les données. Ici, comme le livre n'est pas interactif, nous avons rentré la phrase « Les insectes sont nos amis, il faut les aimer aussi ».

Par la suite, nous allons comprendre les différents éléments qui composent un tel script. Mais d'abord, une étape importante est de pouvoir résoudre les problèmes rencontrés. Car les erreurs sont nombreuses quand on programme. Enlevez une parenthèse au hasard dans le script précédent et regardez l'effet pour vous en convaincre.

## 2.5 Faire des erreurs

### 2.5.1 Ne paniquez pas

Commençons par rappeler une évidence : il est normal et fréquent d'avoir des messages d'erreur quand on programme. Une compétence importante est alors de résoudre ces inévitables erreurs.

Concrètement cela signifie :

- ⊕ écrire du code ;
- ⊕ constater que celui-ci ne marche pas (ça arrive presque toujours) ;
- ⊕ ne pas paniquer ;
- ⊕ lire le message d'erreur pour identifier où le problème a eu lieu ;
  - \* chercher de l'information pour le comprendre ;
  - \* simplifier le code pour isoler le passage problématique ;
- ⊕ tester de nouveau avec des modifications ;
- ⊕ recommencer jusqu'au succès ou à la pause-café.

Le comportement à éviter devant un message d'erreur est d'avoir peur de le lire. Ces messages sont faits pour vous donner des informations afin de vous permettre de corriger votre code. Si vous êtes dans un Notebook Jupyter, le message d'erreur est décomposé en étapes pour identifier à quel moment dans le script l'ordinateur rencontre un cas auquel il n'était pas préparé. Très souvent, l'origine de l'erreur est simplement que vous n'avez pas bien écrit le nom d'une variable, que vous avez oublié de fermer une parenthèse ou un guillemet, que le texte était en fait un nombre, que vous divisez par 0. Ce sont des petites étourderies qui peuvent, par contre, prendre du temps à corriger.

#### ❶ Information

##### C'est quoi la structure d'une erreur en fait ?

La majorité des erreurs en Python se manifeste par ce que l'on appelle une *exception* et sa *trace* correspondante. Ce que vous voyez en tant qu'utilisateur est un message d'erreur que vous donne Python en réaction à quelque chose qui ne lui convient pas. Les erreurs et exceptions sont importantes car elles vous indiquent où se trouve un problème dans votre démarche ou vos données.

Une grande partie de ces erreurs sont des fautes de *syntaxe* : Python vous signale que vous ne parlez pas bien sa langue, comme une parenthèse ouverte mais pas fermée. En effet, avant d'exécuter le code, un analyseur lit ce que vous avez écrit et vérifie si tout est correct. Si ce n'est pas le cas, il vous le signale. D'autres erreurs peuvent arriver au cours de l'exécution quand ce qui est demandé est interdit. Par exemple, vous voulez diviser un

chiffre par une lettre, parce que vous vous êtes trompé de variable. L'interpréteur vous renvoie une **exception** qui peut être spécifique (par exemple, `ZeroDivisionError` si vous divisez par 0). Une erreur n'est pas forcément grave, il est même possible de continuer après une erreur si celle-ci est potentiellement attendue. Nous verrons ça à la fin du chapitre.

Ainsi, si vous exécutez ce code dans un Notebook ou dans le terminal IPython ou dans un Notebook Jupyter :

```
print("Essayons de ne pas faire d'erreur")
```

```
File "<ipython-input-5-23963163c405>", line 1
    print("Essayons de ne pas faire d'erreur")  
^
```

```
SyntaxError: unexpected EOF while parsing
```

Vous rencontrez une erreur de syntaxe `SyntaxError`. Celle-ci correspond à un `EOF` (*End Of File*, ou fin de fichier). Vous regardez votre ligne et vous voyez qu'il manque une parenthèse.

En pratique, les erreurs seront probablement un peu plus difficiles à diagnostiquer. L'ordinateur ne comprenant pas ce que nous voulons dire va continuer et avoir l'impression que l'erreur est plus loin dans le code :

```
print("Additionnons 2 et 2"
2+2
```

```
File "<ipython-input-6-e8c2182df66b>", line 2
    2+2  
^
```

```
SyntaxError: invalid syntax
```

Ici la parenthèse manquante a donné à l'ordinateur l'impression qu'il fallait continuer et, ce faisant, il pense que le problème est au niveau de l'addition, alors que celle-ci est bien écrite. C'est à nous de regarder *autour* de l'endroit de l'erreur pour identifier à quel moment le code demande à l'ordinateur quelque chose qui ne lui convient pas.

Une manière de résoudre rapidement les erreurs est de décomposer le code en exécutant les lignes une à une. Dans le cas précédent, cela revient à exécuter à part la première ligne, puis la seconde. En faisant cela, nous nous rendons compte que l'erreur est sur la première ligne.

## 2.5.2 Trouver de l'aide

Trois grandes raisons peuvent vous amener à chercher de l'aide : soit vous rencontrez une erreur incompréhensible ; soit vous avez besoin d'un exemple de comment réaliser une opération ; soit vous avez du code mais vous ne comprenez pas comment il fonctionne.

Dans tous les cas, nous bénéficions de l'immense quantité d'information déjà disponible sur internet. Ainsi, n'hésitez pas à copier votre message d'erreur directement dans un moteur de recherche, en n'oubliant pas les guillemets (") autour du message pour que le moteur cherche la même formulation. Pensez aussi à rechercher aussi une partie du message d'erreur si besoin, certains messages étant en partie spécifiques à votre code.

Une autre solution est de consulter le site <https://stackoverflow.com/> qui comprend une très grande communauté de développeurs. Si vous ne trouvez pas directement la réponse (ce qui est rare, prenez le temps de chercher un peu), n'hésitez pas à poser la question (en anglais) en décrivant le problème.

De la même manière, si vous cherchez à réaliser un traitement spécifique (par exemple lire un format de fichier particulier, disons un document Word), n'hésitez pas à chercher `lire document word python` pour trouver des exemples de comment faire cette opération.

Enfin, Python est un langage pour lequel il existe une *documentation* qui présente concrètement la manière de l'utiliser. Par exemple, prenons la fonction d'affichage à l'écran, `print`. Pour avoir la documentation de la fonction, il faut exécuter la commande suivante :

```
print?
```

```
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object ; defaults to the current sys.stdout.
sep: string inserted between values, default a space.
end: string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
Type: builtin_function_or_method
```

La documentation précise que `print` affiche la valeur passée entre parenthèses et possède d'autres arguments (ce qui est mis entre parenthèses) optionnels. En faisant une recherche en ligne, vous tombez sur des documentations plus complètes avec des exemples<sup>18</sup>. Il vous faudra adapter en fonction de vos besoins.

---

<sup>18</sup>. Par exemple, [https://www.w3schools.com/python/ref\\_func\\_print.asp](https://www.w3schools.com/python/ref_func_print.asp).

## 2.6 Créer et manipuler des variables

Nous allons maintenant voir les bases du langage Python. Ces bases sont génériques et servent aux différents usages de la programmation. Nous insistons cependant surtout ici sur les utilisations liées au traitement des données.

### 2.6.1 Qu'est-ce qu'une variable ?

Vous voulez programmer pour réaliser un traitement. Généralement, cela signifie avoir des informations au départ (des données d'enquête, un fichier, des mesures) pour ensuite générer un résultat. Bref : vous allez devoir manipuler des « choses » qui vont avoir du contenu, comme des nombres ou du texte, pouvant varier au cours des opérations. Pour manipuler ces informations changeantes, la programmation utilise des entités appelées des *objets*, contenant les informations, associés à des noms pour pouvoir les identifier, que nous allons appeler variables. Une variable se trouve alors associée à un objet à un moment donné.

#### Information

##### Qu'est-ce qu'un objet ?

La meilleure manière d'envisager le processus de programmation est d'imaginer au début de l'exécution du programme un monde (presque) vide dans lequel on fait apparaître au fur et à mesure de nouveaux éléments : déclaration de nouvelles variables, chargement de bibliothèques et modifications. Par exemple, d'abord il n'y a rien, puis nous créons progressivement une planète appelée Terre, un jardin... enfin, vous voyez l'idée.

Ces éléments sont des objets, c'est-à-dire qu'ils correspondent à quelque chose (un état) et peuvent interagir de certaines manières (un comportement). Un objet particulier appartient à un groupe plus général, sa *classe*, qui lui donne son type. Il est plus précisément une *instance de classe*. Et il a un nom permettant de le retrouver dans la mémoire de l'ordinateur (le nom de la variable).

Un serpent nommé Robert est un objet de type serpent, dans la mesure où il est une instance de la classe serpent. Ce faisant, il peut faire tout ce que peut faire un serpent : vendre des pommes, siffler et bâiller très fort. Par contre, il ne peut pas se mettre du vernis sur les ongles de pieds. C'est un serpent. Son comportement est défini par sa classe. Ce comportement peut être très simple, comme un nombre, ou très complexe, comme certains objets que nous verrons plus loin dans le manuel.

Nous allons par la suite parler d'objet pour désigner les entités sur lesquelles nous agissons et que nous manipulons. Par contre, si tout est objet, les objets diffèrent suivant leur nature. Chacun a sa propre « structure ». Un objet peut contenir d'autres objets. En particulier, il peut contenir des *attributs* (des valeurs internes) et des fonctions (directement liées). Ce sont les *méthodes*. Par la suite, nous allons être amenés à utiliser ces termes.

Par exemple, imaginons que pour un calcul vous avez besoin de la population française. En Python, une ligne suffit pour créer une nouvelle variable et lui donner une valeur particulière :

```
pop_france = 67190000
```

Cette ligne de code fait deux choses : elle crée une nouvelle variable à laquelle elle attribue (le signe =) l'objet nombre qui a la valeur 67190000. L'opération = met la valeur de ce qui est à droite dans l'objet à gauche. Une remarque : le nombre d'espaces autour du signe = ne change rien. Plus généralement, il est possible de mettre des espaces entre les éléments du code sans que cela en change le sens : ce sont les sauts de ligne qui sont les véritables ruptures du code.

La notion d'attribution (le signe =) est importante, prenez le temps de la retenir : attribuer une valeur à une variable signifie que cette variable peut être utilisée pour se référer à la valeur.

Vous êtes libre de *nommer* vos variables comme vous voulez, en respectant quelques règles de base : pas d'espaces, pas de caractères spéciaux... et en écrivant toujours la variable de la même manière (même variable, même nom).

Nous nous permettons de vous donner quelques conseils. Le premier est de donner des noms explicites : n'appelez pas toutes vos variables r, variable ou v1, vous allez vous perdre. Ensuite, utilisez des minuscules et des majuscules, ou des underscore « \_ » pour rendre les noms clairs comme populationVille ou population\_ville. Nommer clairement les choses est un bon début pour éviter de se tromper. Attention, les majuscules comptent<sup>19</sup>.

## ?

### Exercice

**Créez une variable avec la population française en million d'habitants.**

```
pop_france_M = 67.19
```

Maintenant vous pouvez utiliser cette variable qui contient l'information. Pour afficher la population, nous avons déjà vu la fonction `print` :

19. Quand vous vous trompez, ou que la variable n'est pas définie, vous obtenez une erreur de type `NameError` disant que cette variable n'est pas définie.

```
print(pop_france)
```

67190000

Ce code utilise la fonction `print` qui prend comme élément à afficher la variable `pop_france`.

### ?

### Exercice

**Quelle est la différence entre avec et sans guillemets autour d'un texte ?**

`pop_france` est une variable, "`pop_france`" est un texte contenant les lettres « `pop_france` ».

Nous pouvons faire des opérations sur les variables et les transformer. Par exemple, calculons maintenant la population en France si elle augmente de 10 % avec les opérations mathématiques multiplier `*` et addition `+`.

```
pop_france_augmentation = pop_france + 0.1*pop_france  
print(pop_france_augmentation)
```

73909000.0

Ce code crée une nouvelle variable `pop_france_augmentation` et indique à l'ordinateur que cette nouvelle variable doit avoir la valeur de `pop_france` plus 10 % de cette variable. Si nous changeons `pop_france` après avoir défini `pop_france_augmentation`, cette dernière ne changera pas.

### ?

### Exercice

**Affichez directement la population augmentée sans créer une nouvelle variable.**

```
print(1.1*pop_france)
```

Notons que `1.1*pop_france` ne modifie pas `pop_france`. La majorité des opérations dans Python va retourner une nouvelle valeur et ne va pas modifier les valeurs existantes.

La variable est par définition *variable*, ce qui veut dire que vous pouvez stocker autre chose dans le même nom. Ainsi, si la population française augmente avec un facteur de 0.39 % par an, la population dans un an peut se recalculer :

```
print(pop_france)
pop_france = pop_france + 0.0039*pop_france
print(pop_france)
```

67190000  
67452041.0

Ce code :

- ➊ affiche la valeur contenue dans la variable `pop_france`;
- ➋ calcule la croissance de la population pendant un an et la stocke dans la même variable;
- ➌ affiche la valeur contenue (à ce moment) dans la variable `pop_france`.

Dans un cas le nombre est entier, dans l'autre il a des décimales. Pendant cette opération, la variable a en effet changé de type.

## 2.6.2 Le type des variables

Les variables sont associées à des objets de différentes natures. Une variable a donc un *type* qui correspond à la nature de son objet<sup>20</sup>. Ce contenu appartient en effet à une famille d'*objets* : nous avons déjà vu les nombres entiers : leur type est *entier*, en anglais *integer*. Si on regarde le type de `pop_france` en utilisant la fonction `type`, on a :

```
pop_france = 67190000
type(pop_france)
```

`int`

Ce code redéfinit la variable initiale et affiche<sup>21</sup> le type de la variable `pop_france` avec la fonction `type`.

Les entiers sont un type de base de Python. Les nombres à virgule sont des `float`. Si on met la population en millions avec des décimales, on va changer le type de la variable : ce n'est plus la même nature de nombre, on passe des entiers à des nombres à virgules.

```
pop_france_M = 67.19
type(pop_france_M)
```

20. En Python, le type d'une variable est défini implicitement au moment de son attribution avec le signe `=`. Dans d'autres langages, il est nécessaire de définir explicitement ce type. Ce qu'on gagne en flexibilité se perd en sécurité, avec potentiellement des erreurs liées à des problèmes de types.

21. Dans ce cas, nous n'utilisons pas la fonction `print` car généralement Python affiche le résultat de la dernière opération. Vous pouvez aussi utiliser `print(type(pop_france))`.

**float**

Ce code fait la même opération que précédemment, mais avec un nombre à virgule.

Il est possible de faire des opérations avec les nombres. Vous les connaissez déjà. Ces principales opérations de base sont présentées dans le tableau suivant :

Opérations	Symboles	Exemples
addition	+	2 + 5 donne 7
soustraction	-	8 - 2 donne 6
multiplication	*	6 * 7 donne 42
puissance	**	5 ** 3 donne 125
division	/	7 / 2 donne 3.5
reste division	%	7 % 3 donne 1
quotient division	//	7 // 3 donne 2

En Python, la valeur nulle (absente) est `None`. Vous pouvez définir une variable qui a une valeur nulle :

```
variable = None  
print(variable)
```

`None`

Ce code définit la variable `variable` et lui donne une valeur nulle, puis affiche cette variable (donc rien). `None` n'est pas supérieure, inférieure ou comparable à quoi que ce soit.

### 2.6.3 Les textes ou chaînes de caractères

Vous allez souvent être confronté à du texte. Ces textes sont un type spécifique qui facilite la manipulation appelé *string* (*chaîne de caractères*) et abrégé en `str`. Définir un texte se fait entre guillemets de la manière suivante :

```
pays = "France"  
type(pays)
```

`str`

Ce code définit une nouvelle variable `pays` et lui donne comme valeur un texte. Il affiche ensuite le type de cette variable.

Vous pouvez utiliser les guillemets simples ou doubles. Attention, si votre texte contient déjà des guillemets, il risque d'y avoir des erreurs. Vous avez deux choix :

- ⊕ utiliser les autres guillemets : si votre texte contient des guillemets ", utilisez les guillemets ' pour le définir ;
- ⊕ dire à Python de ne pas considérer les guillemets à l'intérieur en mettant devant chacun un backslash : \" ;
- ⊕ utiliser des triples guillemets """ ou '''.

```
print("C'est un petit pas pour l'\homme..." - N. Armstrong)
print("\C'est un petit pas pour l'\homme..." - N. Armstrong")
print(''''C'est un petit pas pour l'\homme..." - N. Armstrong''')
print(""""C'est un petit pas pour l'\homme..." - N. Armstrong""")
```

```
"C'est un petit pas pour l'\homme..." - N. Armstrong
"C'est un petit pas pour l'\homme..." - N. Armstrong
"C'est un petit pas pour l'\homme..." - N. Armstrong
"C'est un petit pas pour l'\homme..." - N. Armstrong
```

## i Information

### Les caractères spéciaux

Programmer c'est écrire du texte. Il peut donc y avoir des confusions entre les textes manipulés dans la programmation et le code lui-même qui est aussi du texte. Dans beaucoup de situations, certaines lettres vont avoir un sens spécial pour le programme, qui va donc essayer de faire des opérations spécifiques. Pour éviter cette incompréhension, il est nécessaire de prévenir Python en « échappant » les lettres spéciales qui pourraient avoir un autre sens. On fait cela en mettant un *backslash* \. Par exemple, % est souvent un symbole spécial. Pour mettre un pourcentage dans un texte comme texte, il faut écrire \% , sinon vous risquez d'avoir une erreur. Les caractères \n, \t, \r ont des significations définies : saut de ligne, tabulation et retour à la ligne.

Plusieurs stratégies existent pour mettre en forme des textes. Nous reviendrons dans les prochains chapitres sur plusieurs manières de faire. Pour le moment, voici quelques éléments importants à maîtriser : vous pouvez relier deux morceaux de texte avec le symbole + et vous pouvez transformer un nombre ou un autre objet en chaîne de caractères par la fonction str(). Cela permet de faire de la mise en forme comme dans le cas suivant :

```
texte = "Population " + str(pop_france) + " habitants"
print(texte)
```

Population 67190000 habitants

Ce code crée une nouvelle variable **texte** en additionnant des morceaux de textes et la variable **pop\_france** transformée en texte avec la fonction **str**.

### ?

### Exercice

Créez un texte qui contient à la fois le nom du pays et la population à partir des variables.

```
population = 67190000
pays = "France"
print(pays + " a " + str(population) + " habitants")
```

## 2.6.4 Les ensembles : listes et dictionnaires

Après les nombres et les textes, deux autres types sont importants : les *listes* et les *dictionnaires*, qui permettent de définir des ensembles. Par exemple, l'ensemble des réponses à une question d'un questionnaire.

Les *listes* sont des ensembles d'éléments ordonnés. Vous pouvez les repérer par l'usage des crochets [ ] qui permettent de sélectionner un ou plusieurs éléments. Prenons l'exemple d'une liste de la population de différents pays limitrophes à la France : l'Allemagne, l'Italie, l'Espagne, la Belgique, la Suisse et le Luxembourg. Nous pouvons définir une liste de leur population en millions d'habitants.

```
liste_populations = [81,60,46,11,8,0.5]
print(liste_populations)
```

[81, 60, 46, 11, 8, 0.5]

Ce code crée une nouvelle variable `liste_populations` et lui donne comme valeur une liste. Ensuite il l'affiche avec `print`.

### ?

### Exercice

Affichez le type de la variable `liste_populations`.

```
type(liste_populations)
```

Une liste est un objet qui permet de stocker d'autres objets et de les manipuler. Chaque élément de la liste est associé à une place. Celle-ci commence à 0 pour le premier élément, 1 pour le deuxième, etc.

Si vous voulez sélectionner le troisième élément, la forme est la suivante : `liste_populations[2]`.

Nous nous permettons d'insister car cela peut être contre-intuitif quand on débute : une liste commence à 0. Dans la série des chiffres de 0 à 9, 0 est bien le premier chiffre.

Une fois créée, une liste est un objet qui permet de faire des opérations. Vous pouvez *rajouter* un élément avec `append` et en *supprimer* avec `remove`. Dans les deux cas, ces fonctions s'appellent à partir de la liste avec le nom de la variable, puis le point « `.` », puis le nom de la fonction.

## Exercice

Si je ne veux que les pays ayant plus d'un million d'habitants, je peux enlever le Luxembourg.

```
liste_populations.remove(0.5)
print(liste_populations)
```

[81, 60, 46, 11, 8]

Ce code supprime un élément de la liste avec l'outil `remove` de la liste, puis affiche la liste.

**Affichez le quatrième élément de la liste et le dixième.**

```
print(liste_population[3])
print(liste_population[9])
```

Vous avez un message d'erreur pour la deuxième ligne : en effet, vous avez pris un rang qui n'existe pas dans la liste.

Si je voulais maintenant rajouter la population de la France, je peux utiliser la fonction `append` de la manière suivante :

```
liste_populations.append(67)
print(liste_populations)
```

[81, 60, 46, 11, 8, 67]

Si je me rends compte que les chiffres pour l'Allemagne ne sont pas bons car j'avais pris les chiffres erronés, je peux modifier une valeur particulière :

```
liste_populations[0] = 82
print(liste_populations)
```

[82, 60, 46, 11, 8, 67]

Ce code attribue une nouvelle valeur à l'élément 0 de la liste de la variable `liste_populations`, puis l'affiche.

Les listes permettent beaucoup d'opérations différentes. La sélection des éléments est particulièrement importante pour manipuler ces objets, et peut être un peu contre-intuitive. Voici quelques manipulations :

- ➊ Le dernier élément de la liste `liste` peut être obtenu avec `liste[-1]` ;
- ➋ L'avant dernier élément avec `liste[-2]` ;
- ➌ Un intervalle d'éléments peut être défini en précisant le premier à être inclus et le dernier qui ne sera pas inclus. Ainsi pour prendre le deuxième, troisième et quatrième élément de la liste, la syntaxe sera `liste[1:4]` (et sélectionnera `liste[1]`, `liste[2]` et `liste[3]`) ;

Les listes peuvent contenir des éléments de natures différentes, par exemple des nombres et des textes.

Un autre type de Python est le *dictionnaire*, ou `dict`. Le dictionnaire se caractérise par l'usage des accolades<sup>22</sup> { }. Comme pour les dictionnaires sous forme papier, chaque entrée ou *clé* (*key* en anglais) est associée à une information. Les clés doivent être uniques. Il peut être très utile pour stocker des informations variées. L'ordre par contre n'a pas d'importance car c'est la clé d'entrée qui permet de récupérer l'information.

### ?

### Exercice

Pour la liste [12,32,43,23,"dix",16] faites un code qui remplace avec l'ordinateur l'avant dernier élément par un nombre.

```
liste = [12, 32, 43, 23, "dix", 16]
liste[-2] = 10
```

Pour reprendre l'exemple précédent, chaque population est associée à un nom de pays. Nous pouvons avoir envie de lier le pays à la population. On va donc créer un dictionnaire des populations.

```
dictionnaire_populations = {
    "Allemagne":81,
    "Italie":60,
    "Espagne":46,
    "Belgique":11,
    "Suisse":8,
    "Luxembourg":0.5
}
print(dictionnaire_populations["Allemagne"])
```

81

Ce code crée une nouvelle variable `dictionnaire_populations` avec comme valeur un dictionnaire et affiche la valeur associée à l'Allemagne.

---

22. Il y a trop de claviers différents pour que l'on puisse détailler comment faire des accolades. Dans le cas où votre clavier n'a pas une touche explicite, faites une recherche « faire des accolades avec mon clavier » sur un moteur de recherche car ce symbole est nécessaire.

Chaque élément d'un dictionnaire est mis sous la forme « clé : valeur associée ». Chaque entrée est séparée par une virgule. Dans cet exemple, nous avons sauté une ligne après chaque élément. Ce n'est pas obligatoire, mais cela permet de rendre le code plus lisible<sup>23</sup>.

Une fois le dictionnaire créé et stocké dans une variable, il est alors possible d'accéder à chacune des populations avec sa clé.

## Exercice

Affichez la population de l'Allemagne.

```
print(dictionnaire_populations["Allemagne"])
```

Rajouter un élément dans un dictionnaire se fait simplement. Si par exemple, nous voulons rajouter l'entrée pour le Royaume-Uni, il suffit d'écrire :

```
dictionnaire_populations["Royaume-Uni"] = 65
dictionnaire_populations
```

```
{'Allemagne': 81,
'Italie': 60,
'Espagne': 46,
'Belgique': 11,
'Suisse': 8,
'Luxembourg': 0.5,
'Royaume-Uni': 65}
```

Ce code définit une nouvelle entrée pour le dictionnaire de la variable `dictionnaire_populations` avec l'entrée « Royaume-Uni » et la valeur 65 puis l'affiche.

Le même code va modifier la valeur si la clé existe déjà :

```
dictionnaire_populations["Allemagne"] = 82
dictionnaire_populations
```

```
{'Allemagne': 82,
'Italie': 60,
'Espagne': 46,
'Belgique': 11,
```

---

<sup>23</sup>. Attention, un saut de ligne n'est possible qu'après une virgule, sinon vous allez avoir une erreur. Si vous voulez du code compact, ne sautez pas de ligne. Si vous voulez sauter une ligne dans votre code, utilisez le symbole \ avant de sauter la ligne signalant à Python de ne pas compter le saut de ligne.

```
'Suisse': 8,  
'Luxembourg': 0.5,  
'Royaume-Uni': 65}
```

Ce code modifie l'entrée « Allemagne » du dictionnaire en lui donnant la valeur 82. Puis il l'affiche.

Les dictionnaires permettent de rassembler des informations différentes et facilitent la structuration de vos données. Par exemple, si vous avez à travailler sur des données d'individus (avec un nom, un prénom, un âge), vous pouvez créer un dictionnaire par individu avec ces informations, puis les rassembler dans une liste. Au cours de votre analyse, vous pourrez rajouter des éléments et à la fin sauvegarder ces données.

### ?

### Exercice

**Créez un dictionnaire avec votre nom, prénom, âge et la ville de résidence, puis une liste avec plusieurs individus.**

Par exemple, pour mettre la fiche d'un des auteurs :

```
emilien = {"nom" = "SCHULTZ", "prenom" = "Émilien",  
           "age" = 33, "ville" = "Marseille, France"}  
matthias = {"nom" = "BUSSONNIER", "prenom" = "Matthias",  
            "age" = 34, "ville" = "Merced, Californie"}  
liste_auteurs = [emilien, matthias]
```

Vous savez maintenant créer des dictionnaires. Une opération est importante : connaître le nombre d'éléments que vous avez dans votre liste ou dictionnaire : la fonction `len` qui renvoie le nombre d'éléments.

```
nombre = len(dictionnaire_populations)  
print(nombre)
```

7

Ce code crée une nouvelle variable `nombre` en lui donnant le nombre d'éléments du dictionnaire `dictionnaire_populations`, puis l'affiche avec la fonction `print`.

### ?

### Exercice

**Vérifiez qu'une chaîne de caractères se comporte comme une liste et que vous pouvez accéder à ses éléments de la même manière.**

Une chaîne de caractères est en fait un ensemble de lettres, la première étant à l'index 0.

```
chaine = "du texte pour essayer"
deuxième_lettre = chaine[1]
longueur_chaine = len(chaine)
print(longueur_chaine)
print("La deuxième lettre est : " + deuxième_lettre)
```

Attention, il n'est par contre pas possible de modifier une chaîne de caractères comme pour les listes.

Enfin, les méthodes `keys`, `values` et `items` d'un dictionnaire peuvent être utiles pour extraire toutes les clés, valeurs ou paires sous forme de listes afin de les manipuler.

## 2.6.5 Conditions et comparaisons

Vérifier si une condition est remplie (vraie) ou pas (fausse) joue un rôle central dans le déroulement du code parce que cela permet de s'adapter aux situations.

Par exemple, vous avez besoin de vérifier qu'un pays a une plus grande population qu'un autre, ou qu'il a un nombre d'habitants supérieur à un seuil, pour pouvoir le classer dans une catégorie. L'opération suivante va dépendre du résultat de la comparaison : « plus grand », « plus petit », « égal » ou « différent ». Cette comparaison va être vraie ou fausse en fonction des valeurs.

Pour comparer, il faut respecter certaines règles. La première est de comparer ce qui est comparable, sinon vous allez avoir des messages d'erreurs. Comparer un nombre avec une lettre n'a pas vraiment de sens. Ensuite, la comparaison va renvoyer un résultat de type « Vrai » ou « Faux », qui correspond à un type particulier d'objet. Le type `bool` pour `booléen` correspond à des objets qui peuvent uniquement prendre la valeur `True` (vraie) ou `False` (faux). *Attention, la majuscule est importante.* Quand on compare deux variables ou valeurs, le résultat est de type `bool`.

L'égalité entre deux objets s'écrit avec `==`. Ainsi, si vous faites `1 == 1.1` vous allez avoir comme réponse `False`. En effet, 1 est différent de 1.1. De même, `type(1) == type(1.1)` donne `False` car les entiers ne sont pas des nombres à virgules. Par contre, `type(1) == int` va être `True` car 1 est bien un entier.

Les autres comparateurs habituellement rencontrés sont :

- ♣ `>` ou `>=` pour supérieur, ou supérieur et égal ;
- ♣ `<` ou `<=` pour inférieur, ou inférieur et égal ;
- ♣ `!=` pour différent de ;
- ♣ `in` pour la présence à l'intérieur de ;

Attention = est une attribution, == une comparaison. Pour vous en convaincre, regardez le code suivant :

```
comparaison = ("deux" == 2)
print("Le type de la comparaison est ", type(comparaison))
print("La valeur de la comparaison est ", comparaison)
```

```
Le type de la comparaison est <class 'bool'>
La valeur de la comparaison est False
```

Ce code commence par créer une nouvelle variable `comparaison` et lui donne comme valeur le résultat de la comparaison entre « deux » et 2 (Python commence par traiter les étapes à partir de la droite et va vers la gauche, en commençant par résoudre ce qui se trouve dans les parenthèses). Ensuite, il affiche le type de la variable et le résultat de la comparaison. La fonction `print` peut afficher plusieurs éléments séparés par des virgules.

## 2.6.6 Encore plus de types

Nous n'avons pas épuisé tous les types possibles. Par exemple, il existe un type pour les dates. Cependant, vous allez rencontrer deux types que nous n'utilisons que peu dans ce manuel mais qui sont par ailleurs fréquents : les ensembles (*set*) et les tuples (*tuple*).

Les ensembles, ou *set*, sont des ensembles non-ordonnés d'éléments uniques. Par exemple pour avoir les éléments uniques d'une liste, vous pouvez transformer la liste en ensemble (*set*) et ensuite calculer ensuite le nombre d'éléments.

```
exemple = [1,2,3,4,5,5,5,5,5]
exemple_set = set(exemple)
print(len(exemple_set))
```

5

Ce code :

- ➊ définit une variable `exemple` qui prend comme valeur une liste ;
- ➋ transforme la liste en set avec la fonction `set`<sup>24</sup> et stocke l'information dans la variable `exemple_set` ;
- ➌ affiche le nombre d'éléments dans le set, correspondant au nombre d'éléments uniques de la liste.

---

<sup>24</sup>. À chaque type est associée une fonction qui permet de définir un objet de ce type, ou de transformer un objet en ce type si c'est possible. Par exemple, `set([1,2,3])` transforme la liste en set.

Les tuples sont similaires aux listes mais avec un ordre définitif et ne peuvent plus être modifiés une fois déclarés. Si vous pouvez ajouter ou enlever des éléments à une liste, ce n'est pas le cas pour les tuples.

```
exemple = (1,2)
print(exemple[1])
exemple[1] = 3
```

2

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-29-e06b63678274> in <module>
      1 exemple = (1,2)
      2 print(exemple[1])
----> 3 exemple[1] = 3

TypeError: 'tuple' object does not support item assignment
```

Ce code :

- ➊ définit une variable `exemple` qui prend comme valeur un tuple avec deux éléments ;
- ➋ affiche le deuxième élément du tuple ;
- ➌ essaye de changer un élément du tuple et provoque ce faisant un message d'erreur.

Vous êtes prévenu, vous pouvez croiser de drôles de types dans le monde de la programmation.

## 2.6.7 #ViveLesCommentaires

Les commentaires se font avec le symbole « `#` » qui évite que la ligne soit exécutée. L'exemple suivant n'affiche rien parce que la ligne d'affichage a été commentée<sup>25</sup> :

```
couleurs = ["bleu", "blanc", "rouge"]
# print(couleurs[2])
```

Ce code commence par définir une variable qui prend comme valeur une liste. Ensuite, il y a une ligne d'affichage mais qui est précédée par le symbole `#` qui fait que la ligne ne va pas être exécutée par le programme.

<sup>25</sup>. On arrive tous à un moment où le code qu'on écrit devient tellement incompréhensible pour gagner du temps que le lendemain, nous sommes obligés de repartir de 0.

N'hésitez jamais à mettre un *commentaire* pour expliquer l'étape que vous faites ou la raison de la création d'une variable, afin que vous (ou quelqu'un d'autre) puissiez en retrouver la logique. Faire des étapes simples avec des commentaires permet de ne pas se perdre. Nous allons ainsi utiliser les commentaires dans le manuel pour préciser les codes plus longs.

Vous pouvez aussi utiliser les commentaires pour inactiver temporairement des morceaux de votre code, par exemple si vous rencontrez un problème afin de voir où il se produit.

## 2.7 Les blocs structurants : conditions, boucles, fonctions

Les lignes que vous écrivez sont exécutées les unes après les autres. Vous savez définir des variables, changer leur valeur et faire des opérations pour arriver à un résultat. Mais pour faire un programme plus efficace, vous allez être amené à organiser votre code en étapes. Cela peut être de vérifier si une variable a une valeur particulière avant de l'afficher ou encore réaliser une opération plusieurs fois sur des données différentes.

Trois types de blocs sont généralement nécessaires pour structurer le script :

- ➊ *tester une condition* : par exemple, vérifier qu'un pays a une population supérieure à un autre ;
- ➋ *répéter une opération* : par exemple, afficher pour chaque pays son nom et sa population à partir d'une liste ;
- ➌ *construire une opération générique* : par exemple, calculer l'augmentation de population de chaque pays pour mettre à jour vos données.

Pour cela, vous avez besoin de trois notions : les *conditions*, les *boucles* et les *fonctions*<sup>26</sup>.

La particularité de Python est que ces blocs sont identifiés par un décalage de quatre espaces<sup>27</sup>. Ces alinéas, ou indentations, font partie de la forme même du langage. Ne les oubliez pas, sinon vous allez avoir un message d'erreur. En Python un alinéa est quatre symboles « espace ». Cela donne une forme un peu particulière au code, qui ressemble au schéma suivant :

---

26. Ces blocs sont les mêmes que ceux disponibles dans d'autres langages de programmation. Il est donc probable que vous les ayez déjà rencontrés. Seule la manière de les écrire peut varier.

27. Nous recommandons vivement d'utiliser quatre espaces pour éviter non seulement les problèmes de copier/coller, mais les foudres des experts Python à qui vous pourriez demander de l'aide et qui n'aiment pas les tabulations. Les éditeurs de texte qui comprennent le Python remplaceront un appui sur la touche « tabulation » par 4 espaces.

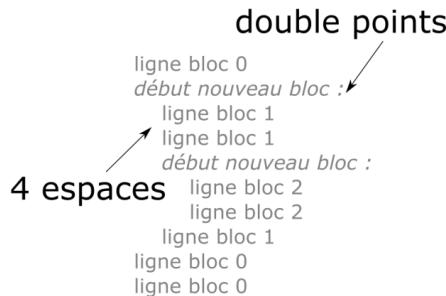


Fig. 2.5 – Organisation en bloc du code Python

Il est rare que vous ayez à faire plus de deux ou trois niveaux d'indentation. Si c'était le cas, d'une part le code devient difficile à lire, de l'autre il existe sûrement une manière d'écrire votre code de manière plus claire en décomposant le traitement avec des fonctions.

### 2.7.1 Si [condition] alors [faire]

Un premier bloc de base consiste à vérifier si une condition est réalisée avant de continuer. Par exemple, cela sert à vérifier si un nombre n'est pas nul avant de l'utiliser pour faire une division, afficher un texte si celui-ci contient un mot ou tracer un point d'une couleur spécifique suivant les valeurs. On parle de « tester une condition » pour désigner le fait qu'on regarde si une condition est vraie ou fausse pour décider si le programme doit exécuter l'opération qui suit. Le mot-clé `if` (*si*) permet de faire le test d'une condition. Son complémentaire est `else` (*sinon*).

Prenons un exemple souvent rencontré : vous voulez décider si une valeur est plus grande ou plus petite qu'un seuil pour classer des informations. Par exemple, pour recoder les pays en deux catégories « Plus de dix millions » et « Moins de dix millions », nous pouvons comparer leur population à un seuil et renvoyer l'information correspondante :

- ➊ la condition évalue si la population est plus grande que dix millions ;
- ➋ si c'est le cas :
  - ★ afficher « Plus de dix millions » ;
- ➌ sinon :
  - ★ afficher « Moins de dix millions » .

Le code est alors le suivant :

```
pop_france = 67
if pop_france >= 10:
```

```
    print('Plus de dix millions')
else:
    print('Moins de dix millions')
```

**Plus de dix millions**

Ce code :

- ➊ définit une variable `pop_france` avec une valeur 67 ;
- ➋ si cette variable est supérieure ou égale à 10 :
  - \* exécute la ligne suivante qui affiche avec `print` un texte ;
- ➌ sinon :
  - \* passe à l'autre ligne qui affiche l'autre texte.

Peut-on mettre plusieurs conditions ? Il est possible de combiner les conditions avec les opérateurs `and` (et) ainsi que `or` (ou). Si vous avez une autre variable qui indique le continent, vous pouvez écrire un autre code qui distingue les quatre situations mutuellement exclusives<sup>28</sup>.

```
population = 18
continent = "Europe"

if (population < 10) and (continent=="Europe"):
    print('Petit pays européen')
if (population < 10) and (continent!="Europe"):
    print('Petit pays non européen')
if (population >= 10) and (continent!="Europe"):
    print('Grand pays non européen')
if (population >= 10) and (continent=="Europe"):
    print('Grand pays européen')
```

**Grand pays européen**

Ce code :

- ➊ définit les deux variables `population` et `continent` ;
- ➋ fait un premier test qui renvoie `False`, donc ne fait rien et continue ;
- ➌ fait un deuxième test qui renvoie `False`, donc ne fait rien et continue ;
- ➍ fait un troisième test qui renvoie `False`, donc ne fait rien et continue ;
- ➎ fait un quatrième test qui renvoie `True` et donc exécute le contenu du bloc qui affiche « Grand pays européen ».

---

<sup>28</sup>. Il existe plusieurs manières d'écrire un même code. Dans ce cas, vous pouvez utiliser aussi une condition similaire `elif` ou encore imbriquer les conditions `if` l'une dans l'autre pour ne pas avoir besoin d'utiliser `and`.

## 2.7.2 Les boucles pour répéter les opérations

Vous avez envie de répéter une opération plusieurs fois. Par exemple, vous voulez prendre une à une chaque information d'une liste pour les analyser. Une **boucle** est un bloc qui permet de définir une répétition d'opérations.

Un cas fréquemment rencontré est que vous avez un ensemble de données (une liste ou un dictionnaire) et que vous voulez prendre un à un les éléments. Si cette variable s'appelle **elements**, l'écriture est la suivante : **for un\_element in elements** (puis suivi d'un double point pour ouvrir le bloc). La variable **un\_element** va être créée dans le bloc et va prendre les valeurs présentes dans votre liste **elements** une à une.

Vous pouvez donner le nom que vous voulez à cette variable. Une coutume souvent empruntée est d'utiliser un nom d'itérateur à une lettre, souvent **i**, **p** ou **l**. Vous allez donc rencontrer souvent une écriture comme **for i in elements:**, où **i** va donc représenter l'élément de la liste. Vous pouvez cependant lui donner un autre nom.

Pour reprendre notre exemple, vous avez une liste des populations de pays et vous voulez les afficher une à une :

```
liste_populations = [81,67,60,46,11,8,0.5]
for p in liste_populations:
    print("Population",p)
```

```
Population 81
Population 67
Population 60
Population 46
Population 11
Population 8
Population 0.5
```

Ce code :

- ➊ définit une liste ;
- ➋ débute une boucle **for** qui définit la variable **p** qui va prendre un à un les éléments de la liste **liste\_populations** :
  - ★ affiche la valeur de **p**.

### ➊ Exercice

Faites le même affichage en affichant à la fois la valeur et la position dans le tableau.

Vous avez (au moins) trois façons de le faire. La première est de définir une nouvelle variable qui va compter les étapes. La deuxième est d'utiliser la fonction `enumerate` qui va renvoyer pour chaque élément un *tuple* (`position, valeur`). La troisième est de faire la boucle non pas sur la liste directement mais sur une liste qui comprend les positions construites avec la fonction `range(min,max)` qui renvoie les nombres entre min et max-1.

```
longueur_liste = len(liste_populations)
for i in range(0,longueur_liste):
    print(i,liste_populations[i])
```

Une boucle sur une liste permet de récupérer un à un les éléments de la liste. Pour un dictionnaire, la boucle `for` prends les entrées une à une.

```
for pays in dictionnaire_populations:
    print(pays,dictionnaire_populations[pays])
```

```
Allemagne 82
Italie 60
Espagne 46
Belgique 11
Suisse 8
Luxembourg 0.5
Royaume-Uni 65
```

Ce code fait une boucle sur le dictionnaire, prend à chaque fois la clé de l'entrée et la met dans la variable temporaire `pays`. À chaque fois, il affiche ensuite la clé et la valeur associée.

## ?

### Exercice

N'affichez que les pays de plus de 50 millions d'habitants.

```
for pays in dictionnaire_populations:
    if dictionnaire_populations[pays] >= 50:
        print(pays,dictionnaire_populations[pays])
```

Maintenant que vous savez faire des conditions et des boucles, vous pouvez commencer à traiter des données. Vous avez ainsi toutes les cartes en main pour recoder des données existantes, en particulier pour transformer des valeurs numériques en catégories. Pour ce faire, les étapes sont les suivantes :

- ➊ définir un nouveau dictionnaire (ou liste) qui va contenir la variable recodée ;
- ➋ prendre une à une les valeurs de l'ancien dictionnaire (ou liste) ;
- ➌ faire des conditions permettant de sélectionner la nouvelle valeur à associer.

```

liste_recodee = []
for p in liste_populations:
    if p >=10:
        liste_recodee.append(">=10")
    else:
        liste_recodee.append("<10")
print(liste_recodee)

```

[ '>=10', '>=10', '>=10', '>=10', '>=10', '<10', '<10' ]

Ce code :

- ➊ définit une variable `pop_reco` vide pour stocker le résultat ;
- ➋ fait une boucle qui prend chacune des données de la `liste_populations` dans la variable temporaire `p` ;
  - \* si la valeur est plus grande que 10 :
    - ajoute « `>=10` » dans `liste_recodee` ;
  - \* sinon :
    - ajoute « `<10` » dans `liste_recodee` ;
- ➌ affiche la valeur de `liste_recodee`.

### 2.7.3 Construire ses fonctions

Les *fonctions* sont des morceaux de code qui ont pour but de traiter une entrée et de produire un résultat en sortie, que l'on peut ensuite exécuter quand on en a besoin avec des entrées différentes. De manière très schématique, nous pouvons les envisager comme un mécanisme avec des entrées et des sorties. Toute la question est alors de définir son contenu qui réalise une action.

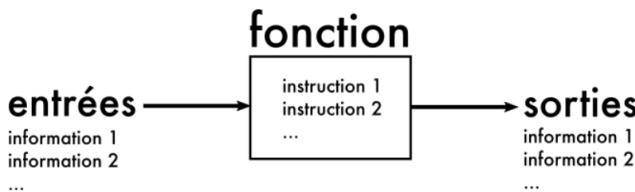


Fig. 2.6 – Structure d'une fonction

Quand le programme rencontre une fonction, il va chercher le code correspondant à cette fonction (qui doit donc avoir été chargée), lui donne les informations entre les parenthèses comme entrée et attend qu'elle fasse ses opérations et lui donne des informations en sortie. Pour `print` l'opération est d'afficher l'entrée. Pour `len` c'est de compter le nombre d'éléments de la liste et de donner ce nombre en sortie.

Appeler une fonction, c'est concrètement demander au programme d'utiliser des lignes de codes définies ailleurs. Il y a donc deux moments : le premier est la *déclaration* de la fonction, qui définit en général ce qu'elle doit faire et comment elle le fait ; le deuxième est son *exécution* dans un programme particulier avec des données spécifiques.

Pour reprendre le cas précédent, ce serait bien de ne pas avoir à chaque fois besoin de réécrire cette étape qui teste si un nombre est plus grand ou plus petit que 10 millions. C'est une opération identifiée et ce serait bien de pouvoir réutiliser cette étape plus facilement que de copier/coller le code.

Déclarer une fonction, c'est identifier une action que l'on voudrait faire, définir ce qu'on veut en entrée pour pouvoir le réaliser, et ce qu'on veut en sortie au final et décrire comment arriver de l'entrée à la sortie. Deux éléments sont importants : **def** pour définir le bloc d'une nouvelle fonction et **return** pour renvoyer une valeur. Une fois que le mot-clé **return** est atteint, le programme arrête la fonction, renvoie la valeur et sort du bloc. Si le programme arrive à la fin de la fonction, et qu'il n'a pas vu **return**, il quitte la fonction en retournant la valeur **None**.

La déclaration d'une fonction amène à définir les éléments qu'on lui donne en entrée. C'est vous qui définissez le nom de la fonction et ses arguments. Vous mettez ce que vous voulez, tant que vous utilisez ensuite ce nom à chaque fois que vous voulez utiliser la fonction. Pour prendre l'exemple précédent, recoder la population nécessite d'avoir un nombre en entrée (la population) et de produire une chaîne de caractères en sortie «  $\geq 10$  » , «  $<10$  »). Notez que comme pour les boucles ou les conditions, le contenu de la fonction est décalé de 4 espaces.

```
def recoder_population(population_entree):
    if population_entree >=10:
        return ">=10"
    else:
        return "<10"
```

Ce code :

- ➊ définit une nouvelle fonction avec **def** qui prend comme nom **recoder\_population**, avec une variable **population\_entree**;
- ➋ si **population\_entree** est supérieure ou égale à 10 :
  - \* la fonction retourne en sortie un texte «  $\geq 10$  » ;
- ➌ sinon :
  - \* retourne «  $<10$  ».

Une fois définie, vous pouvez utiliser cette fonction dans votre code<sup>29</sup>. Exécuter une fonction se fait en marquant les parenthèses et en donnant les arguments nécessaires.

29. Il faut que la fonction soit chargée, c'est-à-dire que le code de définition ait été exécuté dans le programme.

```
recodec_population(11)
```

```
'>=10'
```

Ce code appelle la fonction `recodec_population` qui a été définie avant avec une valeur spécifique d'entrée 11. En faisant cela, le programme va utiliser le code de la fonction pour traiter l'entrée et renvoyer une valeur de sortie.

## ?

### Exercice

**Écrivez une fonction qui renvoie la somme des valeurs d'une liste.**

```
def somme_liste(liste_entree):
    total = 0
    for nombre in liste_entree:
        total = total + nombre
    return total

exemple_liste = [10,23,43,54]
somme_liste(exemple_liste)
```

Ce code commence par définir une nouvelle fonction qui prend une liste en entrée. Cette fonction crée une variable `total` à 0 puis fait une boucle `for` sur les éléments pour les additionner à cette variable interne. À la fin de la boucle, la fonction renvoie la valeur de `total`. Ensuite, on définit une variable `exemple_liste` qui prend une liste de nombres comme valeur. On applique enfin la fonction `somme_liste` en donnant cette variable en entrée. Le résultat est alors affiché.

En combinant fonction et boucle, notre code précédent peut être compressé :

```
pop_reco = []
for pop in liste_populations:
    pop_reco.append(recodec_population(pop))
print(pop_reco)
```

```
['>=10', '>=10', '>=10', '>=10', '>=10', '<10', '<10']
```

Ce code :

- ⊕ définit une variable `pop_reco` avec comme valeur une liste vide ;
- ⊕ fait une boucle sur tous les éléments de la liste de population, mettant une des valeurs dans la variable `pop` à chaque étape :

- \* ajoute dans notre liste `pop_reco` avec la fonction `append` de la liste le résultat renvoyé par la fonction `recoder_population` appliquée sur la valeur `pop` (il y a deux étapes : d'abord la fonction renvoie un résultat, ensuite celui-ci est ajouté à la liste) ;
- ⊕ affiche le résultat de la variable recodée `pop_reco`.

## ?

### Exercice

Utilisez la fonction précédente sur un nombre écrit en toutes lettres `recoder_population("dix-neuf")` et proposez une solution pour corriger l'erreur.

L'erreur vient du fait que la fonction est prévue pour des nombres. Si l'entrée n'est pas un nombre, Python signale qu'il y a un problème au moment de la comparaison. Pour résoudre le problème, il faut d'abord tester avant de comparer si `population_entree` est bien un nombre. Si ce n'est pas le cas, la fonction peut par exemple renvoyer « Erreur d'entrée ».

```
def recoder_population(population_entree):  
    if type(population_entree) != int:  
        return "Erreur d'entrée"  
    if population_entree >=10:  
        return "Plus de 10 millions"  
    else:  
        return "Moins de 10 millions"
```

Python permet même de le faire de manière encore plus condensée, grâce à une formulation spécifique, la *list comprehension* ou compréhension de liste. Ne vous inquiétez pas si cela vous paraît un peu complexe et prenez le temps de jouer un peu avec cette formulation. Nous la retrouverons souvent par la suite car elle rend l'écriture du code plus condensée.

```
[recoder_population(pop) for pop in liste_populations]
```

```
['>=10', '>=10', '>=10', '>=10', '>=10', '<10', '<10']
```

Ce code crée en une seule ligne une nouvelle liste en faisant une boucle `for` sur `liste_populations` et en appliquant directement la fonction `recoder_population` sur l'élément de la boucle.

Pour le moment, nous n'avons donné qu'une variable en argument. Nous pouvons en donner plusieurs et nous pouvons même définir la valeur par défaut si jamais la valeur d'entrée n'était pas définie. Par exemple, le seuil de ce qu'est un grand

pays ou un petit pays peut varier suivant notre application, donc il est intéressant de définir une fonction qui prend à la fois la population à recoder et le niveau de séparation :

```
def recoder_population(pop, seuil=10):
    if pop >= seuil:
        return "Plus de "+str(seuil)+" millions d'habitants"
    else:
        return "Moins de "+str(seuil)+" millions d'habitants"
```

Ce code définit une nouvelle fonction `recoder_population` avec deux entrées, `pop` et `seuil`, qui prend une valeur par défaut. La fonction commence par tester une condition si la première entrée est supérieure ou égale à la seconde. Si c'est le cas, elle renvoie un texte, sinon elle renvoie l'autre. Et vous pouvez l'appeler de deux manières différentes, soit en ne donnant que la valeur à recoder (et donc `seuil` vaudra la valeur par défaut 10) ou en donnant aussi une deuxième information qui va alors changer le seuil :

```
print(recoder_population(20))
print(recoder_population(20,30))
```

Plus de 10 millions d'habitants  
 Moins de 30 millions d'habitants

Ce code appelle la même fonction, avec la même valeur de population d'entrée. Dans le premier cas, on ne définit pas le seuil et donc celui-ci est défini par défaut dans la fonction (`seuil=10`). Dans le deuxième cas, on donne explicitement une valeur de seuil.

### ?

### Exercice

**Définissez une fonction qui calcule la différence entre deux populations de pays.**

```
def distance(p1,p2):
    return p2-p1
```

## 2.7.4 Une multitude de fonctions

Maintenant vous savez ce que sont les fonctions et vous savez en écrire. Dans la pratique, de nombreuses fonctions existent déjà. Certaines sont intégrées dans le langage Python (*built-in*). Parmi celles que vous allez souvent utiliser, notons :

- ➊ la fonction `print` pour afficher que nous avons déjà vue ;
- ➋ `input` pour une entrée au clavier ;
- ➌ les fonctions liées aux types de base qui permettent de transformer (dans certaines conditions) des objets dans des types : `str` pour transformer en chaîne de caractères, `int` pour transformer en entier, `list` pour transformer en liste ;
- ➍ les fonctions de traitement : `len` pour le nombre d'éléments dans un ensemble, `range(min,max)` pour avoir une liste d'entiers de min à max-1 ;
- ➎ rajoutons `sum` qui permet de faire la somme d'une liste, `max` et `min` pour le maximum et le minimum d'un ensemble ;
- ➏ `enumerate` appliqué à une liste permet de produire une nouvelle liste qui comprend des paires d'éléments (position,valeur).

## ?

### Exercice

Générez et calculez la somme des 10 000 premiers entiers.

```
sum(range(1,10001))
```

Ce code commence par utiliser la fonction `range` pour créer une liste d'entier entre 0 et 10001 (le dernier étant non inclus), puis fait la somme de toutes ces valeurs.

Dans certains cas, nous allons définir nous-mêmes des fonctions, pour répondre à nos besoins, mais nous allons en fait souvent utiliser des fonctions déjà créées. Pour avoir accès à celles-ci, nous allons utiliser des *bibliothèques*. Le prochain chapitre présente comment installer et utiliser ces bibliothèques. Celles-ci comprennent à la fois de nouveaux objets (par exemple, des tableaux à deux dimensions) et des fonctions pour les traiter.

## !

### Information

#### Les méthodes

Dans beaucoup de situations, nous allons utiliser des fonctions qui sont en fait des *méthodes*, c'est-à-dire des fonctions qui sont contenues dans un objet (une entité qui existe et qui a un nom). Cette situation a été rencontrée dans le cas des listes avec la méthode `append` qui permet de rajouter un élément à une liste existante avec `liste.append(element)`.

Une méthode est liée à un objet : c'est une fonction à l'intérieur de cet objet, que l'on peut utiliser à partir de l'objet. C'est là le sens de l'écriture qui prend la forme « nom de l'objet » puis « . » puis « nom de la fonction

correspondante ». Pour se faire une idée claire, les fonctions sont des outils génériques, tandis que les méthodes sont des fonctions dépendantes d'un objet et qui agissent dans un contexte particulier.

La principale différence entre les fonctions et les méthodes est que, si la fonction fait une opération et retourne des valeurs, la méthode peut avoir pour objectif de modifier l'objet auquel elle est rattachée. Elles agissent souvent à partir de l'objet sur l'objet lui-même. La méthode `append` d'un objet liste permet de rajouter un élément à cette même liste : reliée à l'objet, elle modifie l'objet quand elle est exécutée. Une méthode peut à la fois modifier l'objet et renvoyer quelque chose : nous l'avons vu avec la méthode `pop` qui supprime un élément d'une liste.

Dans tous les cas, prêtez attention à cette différence entre fonction et méthode. Comme la méthode dépend d'un objet, elle dépend aussi de l'état de l'objet en question. Supprimer un élément d'une liste dépend de la présence ou non de cet élément dans la liste. Et donc de tout ce qui s'est passé avant dans votre programme.

### 2.7.5 Un mot sur l'espace des noms

En créant des variables et des fonctions, vous allez utiliser beaucoup de noms dans votre script. Certains noms sont déjà pris par le fonctionnement de Python et cela peut donner lieu à des messages d'erreur. Dans d'autres cas, vous allez utiliser les mêmes noms à différents moments pour des opérations différentes, ce qui peut donner lieu à des confusions.

L'*espace des noms* caractérise les entités qui existent à un moment donné pour votre programme (l'univers). Certains espaces de noms sont locaux. Quand vous êtes dans un bloc, il y a des variables qui n'existent que localement. En effet, maintenant que nous avons vu des blocs qui ajoutent de nouvelles variables spécifiques pour leur fonctionnement (les entrées de fonction ou encore l'itérateur de boucles), cela pose la question de la *portée* des variables dans un programme, c'est-à-dire jusqu'où elles sont visibles.

Quand vous déclarez directement une variable dans votre Notebook, elle devient disponible partout. Cependant, le comportement des variables peut changer quand elles sont dans des blocs.

En particulier, il existe une différence entre des variables *locales* et *globales*. Les variables que vous créez à l'intérieur d'une fonction n'existent que dans la fonction : elles sont *locales*. Par contre, à l'intérieur de la fonction, vous pouvez accéder aux variables définies en dehors, qui sont *globales* dans le script. Comme une boîte qui enferme son petit monde.

Pour bien le comprendre, prenons un petit exemple :

```
variable_globale = 20

# Définition d'une fonction
def test():
    variable_locale = 10
    print("variable globale : ",variable_globale)
    print("variable locale : ",variable_locale)

# Exécution de la fonction
test()
print(variable_locale)
```

```
variable globale : 20
variable locale : 10
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-42-4f1c89838731> in <module>
      9 # Exécution de la fonction
     10 test()
--> 11 print(variable_locale)

NameError: name 'variable_locale' is not defined
```

Ce code définit la variable globale `variable_globale`, puis une fonction `test`. La fonction `test` définit une variable locale en interne `variable_locale` et affiche la variable locale et globale. Il exécute ensuite cette fonction, puis essaye d'afficher la variable `variable_locale` définie dans la fonction. La fonction `test` peut bien afficher les deux variables, mais la variable locale n'est pas accessible de l'extérieur, ce qui génère une erreur.

La gestion de la portée des variables peut être complexe. Par exemple, une variable locale d'une fonction qui a le même nom qu'une variable globale va avoir la priorité dans la fonction. Une manière d'éviter les problèmes est de bien donner des noms spécifiques à vos variables pour éviter les collisions.

Pensez aussi à vérifier quand vous écrivez du script dans le Notebook que la variable que vous manipulez contient bien l'information qui vous intéresse. Cela se fait rapidement en visualisant le contenu.

## 2.8 Aller plus loin pour gérer les erreurs

Avant de conclure ce chapitre, nous voulons revenir sur les erreurs. Ces erreurs peuvent être dans l'écriture du code. Mais elles peuvent aussi venir de l'exécution du code, quand celui-ci demande de faire une opération impossible. Ces erreurs sont des exceptions (*Exceptions*), qui appartiennent à un type comme *ZeroDivisionError* ou *NameError*.

Généralement, ces erreurs apparaissent car vous avez mal conçu la logique de votre code. Vouloir utiliser un élément qui n'existe pas témoigne d'un problème dans la construction générale. Cependant, dans certains cas, des erreurs peuvent arriver car vous ne pouvez pas contrôler tous les éléments : soit que vous dépendez de données d'entrée que vous ne connaissez pas, soit que ce soit plus simple de partir du principe qu'il y aura des erreurs mais qu'elles ne sont pas problématiques.

Python permet de gérer ces exceptions pour qu'elles n'arrêtent pas le programme. L'idée est la suivante : définir un bloc dans lequel se trouve une série d'instructions à exécuter, et si une erreur arrive, exécuter à la place un autre groupe d'instructions. Il est possible de sélectionner le type d'exception qui donnera lieu à l'exécution du second bloc. La syntaxe pour essayer d'exécuter du code est `try` et le code à exécuter si le premier génère une erreur est `except`.

Essayons d'afficher une entrée du `dictionnaire_populations` défini plus haut :

```
print(dictionnaire_populations["Brésil"])
```

```
-----
KeyError                               Traceback (most recent call last)
<ipython-input-43-b7791435ce99> in <module>
----> 1 print(dictionnaire_populations["Brésil"])

KeyError: 'Brésil'
```

Cette valeur n'existe pas, nous avons donc une erreur. Maintenant, refaisons un code qui essaye d'afficher cette valeur, mais prévoit le fait qu'il peut y avoir une exception à gérer.

```
try:
    print(dictionnaire_populations["Brésil"])
except:
    print("Entrée absente")
```

**Entrée absente**

Comme la valeur est absente dans le dictionnaire, le code va utiliser le deuxième bloc `except` et donc afficher la ligne « Entrée absente ».

À quoi sert ce type d'écriture ? Dans un programme complexe, cela permet de prendre en compte la diversité des situations<sup>30</sup>. Nous allons peu utiliser ce type de code, sauf dans certaines fonctions permettant de traiter des données réelles afin de prendre en compte l'absence de certaines données.

## 2.9 Synthèse du chapitre

Dans ce chapitre, nous avons fait une plongée dans la programmation Python en présentant les formulations de base qui vous permettent de communiquer avec l'ordinateur. Vous avez pu vous familiariser avec la syntaxe de ce langage et apprendre les premiers mots. Nous avons aussi vu qu'un programme s'exécute ligne après ligne, mais peut être organisé en blocs qui réalisent certaines actions : des conditions, des boucles, ou encore des fonctions.

Pour le moment, n'essayez pas de tout apprendre de manière exhaustive. Savoir programmer en Python prend du temps et de la pratique. Nous allons rencontrer différentes applications. Ce n'est pas en lisant le dictionnaire de A à Z que l'on devient un grand écrivain, mais en lisant les œuvres d'autrui, en écrivant et en prenant en compte les retours.

Prenez donc le temps de digérer cette introduction et nous reviendrons régulièrement dans le manuel sur ces bases. Pour vous aider à progresser, voici quelques conseils :

- ➊ codez régulièrement ;
- ➋ prenez des notes sur ce qui fonctionne ;
- ➌ utilisez le mode interactif pour tester des formulations ;
- ➍ faites des pauses pour ne pas vous épuiser ;
- ➎ prenez le temps de comprendre les erreurs de votre code ;
- ➏ modifiez et adaptez des codes existants ;
- ➐ codez quelque chose vous-même !

---

30. La gestion des exceptions permet de définir vous-même de nouvelles catégories d'exceptions, pour permettre de gérer les situations problématiques dans l'exécution du programme.

# Chapitre 3

## Les bibliothèques

Ce chapitre présente l'utilisation des bibliothèques Python. En effet, programmer efficacement ne signifie pas écrire soi-même tout son code. Au contraire, vous serez d'autant plus efficace et rapide que vous vous appuierez sur du code écrit par d'autres.

### 3.1 Qu'est-ce qu'une bibliothèque ?

Au-delà des considérations techniques, l'utilité d'un langage repose sur l'existence d'une communauté d'utilisateurs qui développe de nouveaux outils et les rend disponibles. Pour Python, ce code est distribué sous la forme de *bibliothèques* dédiées, ou *packages* en anglais<sup>1</sup>, librement disponibles. Ces bibliothèques contiennent un ou plusieurs modules qui peuvent être chargés dans un script pour apporter de nouvelles fonctions et objets<sup>2</sup>.

Concrètement, les bibliothèques sont des fichiers qui contiennent du code Python mis sous une forme qui permet de faciliter leur utilisation, de la même façon que nous avons vu qu'une fonction permet de réutiliser du code déjà écrit. Dans une bibliothèque, vous trouvez généralement des fonctions permettant certaines actions et des objets permettant de manipuler certains types de données.

En fait, quand vous voulez réaliser un traitement, d'autres personnes se sont sûrement déjà posées des questions similaires. Et souvent, certaines d'entre elles ont pris le temps de développer une bibliothèque dédiée que vous pouvez facilement réutiliser. Par conséquent, il existe des milliers de bibliothèques pour des milliers

---

1. Souvent mal traduit par librairie en français. D'autres langages fonctionnent suivant ce principe, par exemple R.

2. Nous utilisons surtout le terme bibliothèque même si dans certains cas il s'agit de modules, une entité plus élémentaire.

de traitements différents : calculer la moyenne d'une série de nombres, faire une régression logistique ou encore automatiser la récupération d'informations sur internet. Dans un même programme, vous pouvez alors être amené à utiliser plusieurs bibliothèques.

Pour le traitement de données, il existe par exemple un ensemble de bibliothèques adaptées aux actions habituellement réalisées : mettre en forme des données, calculer des statistiques, visualiser des informations. Avec le langage Python, cet écosystème prend le nom de *SciPy* pour *Scientific Python*. Nous allons utiliser une partie des bibliothèques de l'écosystème *SciPy*<sup>3</sup>, car ces outils sont très puissants pour écrire des scripts et traiter des données de manière simple.

Pour le moment, les SHS n'ont pas encore beaucoup de bibliothèques dédiées. Mais celles-ci commencent à se développer, par exemple pour l'analyse textuelle ou pour la réalisation de cartes. Aucun doute que les prochaines années verront le développement de nouvelles bibliothèques répondant aux besoins des SHS.

## 3.2 Un écosystème

Faisons un petit tour de l'écosystème des bibliothèques Python. Le schéma 3.1 présente cet écosystème comme une succession de couches avec à la base des bibliothèques de bas niveau, proches du fonctionnement de l'ordinateur, qui permettent le fonctionnement de bibliothèques de plus haut niveau, proches des humains et souvent spécifiques à un domaine. Ces bibliothèques de haut niveau permettent de réaliser des tâches complexes en quelques lignes de code, mais sans voir les détails du traitement. Nous verrons dans le détail une partie de ces bibliothèques par la suite.

Tout au bas de notre diagramme se trouve le langage Python lui-même avec ses fonctions. Certaines de ces fonctions de base sont contenues dans des modules déjà disponibles qu'il suffit d'importer. C'est le cas par exemple des fonctions mathématiques disponibles dans le module `math` présenté ci-dessous.

Dans la deuxième couche, nous trouvons les bibliothèques permettant de faire du calcul numérique et l'interaction avec l'utilisateur. Par exemple, les données sont stockées et calculées rapidement grâce à *Numpy*.

Enfin, le niveau suivant correspond aux traitements génériques facilités pour un être humain. Nous manipulons directement des données comme un tableau avec *Pandas*, faisons des calculs scientifiques avec *SciPy* et des visualisations avec *Matplotlib*.

Le tout dernier niveau est celui des applications spécifiques à certains secteurs :

---

3. SciPy est un terme qui désigne un ensemble de bibliothèques (dont justement *Scipy*), une communauté de développeurs (principalement des chercheurs du monde entier dans des domaines très différents) et une série de conférences, dont la plus connue a lieu au Texas chaque année.

- ⊕ de l'analyse textuelle avec *Nltk* (*Natural Language Tool Kit*) ;
- ⊕ de l'analyse de réseaux avec *Networkx* ;
- ⊕ du *machine learning* avec *Scikit-learn* ;
- ⊕ du *Big Data* avec *Dask*.

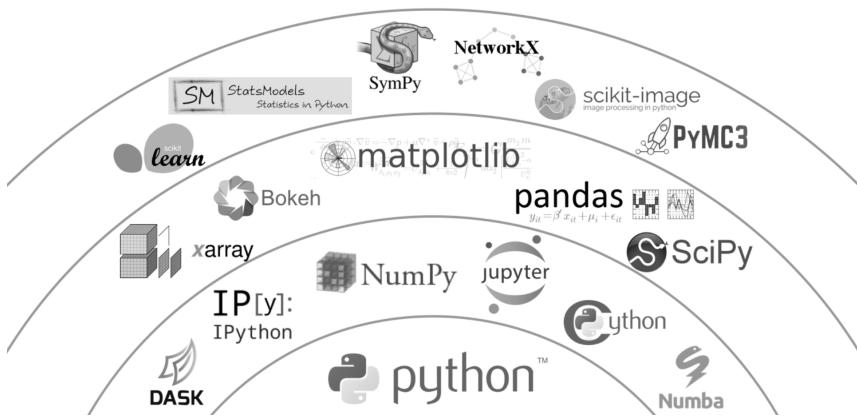


Fig. 3.1 – Schéma de l'écosystème Python (création : Jake Vanderplas)

Vous n'avez pas besoin de tout connaître pour pouvoir utiliser ces bibliothèques et leurs concepteurs ont beaucoup réfléchi pour que ces briques soient compatibles et faciles d'usage. La plupart des bibliothèques avancées utilisent de manière invisible les bibliothèques plus fondamentales et vous n'avez besoin que de quelques lignes pour obtenir votre résultat<sup>4</sup>.

Suivant vos besoins et votre domaine de spécialité, vous allez vous familiariser davantage avec certaines d'entre elles. Par exemple, si vous avez uniquement besoin de faire des statistiques descriptives, les bibliothèques d'analyse de réseaux ne vous intéresseront sûrement pas.

À côté de ces bibliothèques, nous avons aussi des outils complémentaires qui permettent de rédiger et partager ce code : c'est le cas de *Jupyter* ou encore la visualisation avec *Bokeh* permettant de mettre des images et graphiques directement sur internet. Certains de ces outils fonctionnent comme des bibliothèques, mais ne s'appuient pas uniquement sur le langage Python et impliquent d'autres ressources de l'ordinateur.

4. En fait les bibliothèques s'appuient les unes sur les autres. Pour faire certaines opérations, elles vont utiliser d'autres bibliothèques plus « fondamentales ». Cela conduit à générer un réseau de liens, qui est généralement pris en charge automatiquement lors de leur installation.

Au-delà de celles que nous avons mentionnées, il existe de nombreuses autres bibliothèques. Voici une petite liste de celles qu'on aime bien utiliser pour vous donner quelques idées de ce qui est possible :

- ❖ *Seaborn* pour visualiser des données, en particulier statistique ;
- ❖ *BeautifulSoup* pour manipuler des pages internet ;
- ❖ *Scrapy* pour collecter des données sur internet ;
- ❖ *Tweepy* fait l'interface avec Twitter ;
- ❖ *GeoPandas* pour faire des cartes ;
- ❖ *Smtpplib* pour envoyer un mail par SMTP.

D'autres bibliothèques, plus récentes, montrent que les SHS commencent à se saisir des outils de Python :

- ❖ *Tagette* pour coder du matériel qualitatif ;
- ❖ *Researchpy* pour des tests statistiques.

## ❶ Information

### Faire davantage connaissance avec quelques bibliothèques

Nous allons souvent les rencontrer par la suite, donc autant faire connaissance de manière un peu plus approfondie :

- ❖ *NumPy* (prononcé neum-paille) : Cette bibliothèque contient la majorité des fonctions numériques comme la fonction exponentielle ou logarithme, ainsi que la génération de nombres aléatoires. Elle fait partie de la base de l'écosystème scientifique. Nous ne la verrons que peu car pour nos usages nous n'avons pas besoin de directement de manipuler des grands ensembles de données numériques ;
- ❖ *SciPy* (prononcé s'aille-paille) : Cette bibliothèque regroupe des fonctions mathématiques avancées telles les tests statistiques et les constantes numériques ;
- ❖ *Pandas* : Cette bibliothèque permet la manipulation de données sous la forme de tableurs. Pour vous donner une idée, pensez à *Excel* mais en plus puissant et adaptable. Nous lui consacrons un chapitre dédié ;
- ❖ *Matplotlib* : Couteau suisse de la visualisation. Vous pouvez *tout* présenter, de la simple boîte à moustaches au rendu de trous noirs du centre de notre galaxie (ce qui est néanmoins peu utile en SHS). Beaucoup d'exemples sont disponibles pour commencer et il est possible de contrôler finement chaque graphique. Nous lui consacrons aussi un chapitre ;

- GeoPandas : Vous avez des données géographiques ? Des routes, des contours de pays ? Vous devez placer des points sur un fond de carte. Avec *GeoPandas* vous pouvez associer la puissance d'un tableur Excel avec le rendu cartographique.

Un dernier point important à présenter est la notion de *version*, déjà rencontrée pour le langage Python lui-même. Comme pour les logiciels, les bibliothèques évoluent en fonction des ajouts de fonctionnalités et des corrections des problèmes rencontrés. Cela conduit à des versions différentes qui correspondent à des changements plus ou moins importants.

Pourquoi s'intéresser aux versions des bibliothèques ? La première raison est que cela peut être la cause de problèmes, quand un code nécessite une version antérieure d'une bibliothèque qui a évolué. La seconde est que certains changements peuvent amener des transformations importantes dans la manière d'écrire le code.

Les dénominations de versions sont en général formées de trois nombres sous la forme X.y.z où X correspond à la version majeure, y et z sont les versions mineures et correctifs. La plupart du temps, l'utilisation d'une bibliothèque change peu – ou pas – pour une même valeur de version majeure. Le changement de y reflète souvent les corrections de bugs (problèmes divers) ainsi que des améliorations de performance.

Concrètement, cela peut poser des problèmes si vous utilisez des scripts anciens avec une version récente de Python, ou encore si pour certains usages vous avez besoin d'une version particulière. Si vous rencontrez un tel problème, essayez d'installer les versions des bibliothèques adaptées au programme que vous voulez utiliser pour être sûr de ne pas provoquer de « retours vers le futur » incontrôlés...

Certaines bibliothèques sont directement installées. D'autres nécessitent d'être ajoutées, donc d'avoir une connexion internet pour récupérer le contenu.

## 3.3 Utiliser des bibliothèques

Commençons par traiter le cas des bibliothèques déjà installées. Pour les utiliser, il faut cependant les charger car elles ne sont pas automatiquement disponibles dans l'exécution d'un programme spécifique. L'idée est de ne charger que les modules nécessaires. Comme en anglais la commande est `import`, le chargement est aussi souvent appelé importation et on dit qu'on importe une bibliothèque.

### 3.3.1 Charger un module déjà installé

L'installation de Python sur un ordinateur installe en même temps des modules. Par exemple, le module `math` qui permet d'utiliser des fonctions mathématiques est présent.

Pour utiliser cette bibliothèque, vous avez plusieurs possibilités : charger l'ensemble de son contenu, la charger comme une nouvelle entité, ou charger uniquement un nombre limité de fonctions contenues dans le module. Prenons l'exemple de la fonction `floor` qui renvoie la partie entière d'un nombre à virgule (donc sans les chiffres après la virgule).

Soit vous chargez d'abord la bibliothèque comme un module spécifique dont vous allez ensuite appeler des fonctions particulières en précisant explicitement qu'elles proviennent de `math` avec le point « `. .` ».

```
import math  
math.floor(10.53)
```

10

Ce code importe la bibliothèque `math` puis utilise une fonction de cette bibliothèque, `floor`, qui arrondit le nombre. Utiliser un élément de la bibliothèque se fait en nommant le module suivi d'un point et du nom de la fonction. Si le nom est trop long, vous pouvez donner un « surnom » (*alias*), ici `m`.

```
import math as m  
m.floor(10.53)
```

10

Ce code est similaire au précédent à la différence que la bibliothèque voit son nom abrégé en `m` plutôt que `math`. Cela permet d'écrire du code plus compact.

Enfin, vous pouvez charger uniquement une fonction de la bibliothèque de la manière suivante :

```
from math import floor  
floor(10.53)
```

10

Ce code n'importe pas toute la bibliothèque mais uniquement une fonction particulière, `floor`, que nous pouvons ensuite directement utiliser<sup>5</sup>.

Le module `math` est important car il arrive souvent que nous ayons besoin de transformations mathématiques. Voici une petite liste des fonctions les plus souvent utilisées dans nos codes pour certains calculs.

---

5. Vous pourrez lire des codes qui utilisent une forme que nous éviterons dans ce manuel : la construction `import *` qui importe tous les éléments d'une bibliothèque mais qui peut avoir des effets imprévisibles sur nos programmes : une fonction importée peut remplacer une fonction déjà utilisée et perturber le fonctionnement de votre code.

- ⊕ `math.sqrt(x)` pour la racine carrée de `x`;
- ⊕ `math.exp(x)` pour la valeur exponentielle de `x`;
- ⊕ `math.log(x)` pour le logarithme népérien de `x`;
- ⊕ `math.log10(x)` pour le logarithme en base 10 de `x`;
- ⊕ `math.pow(x, y)` pour mettre `x` à la puissance `y`;
- ⊕ `math.cos(x)` pour le cosinus de `x`, pareil avec `math.sin(x)` et `math.tan(x)`.

### 3.3.2 Installer une nouvelle bibliothèque

Si vous voulez utiliser une bibliothèque qui n'est pas encore sur votre ordinateur, il faut d'abord l'installer. Il existe plusieurs méthodes. Nous présentons ici les principales approches. Comme souvent, il existe plusieurs méthodes. Une remarque toutefois : utiliser une interface graphique peut sembler plus facile au début, mais peut présenter des limites sur le long terme. Si vous rencontrez des problèmes pour utiliser les lignes de commande, jetez un coup d'œil sur internet pour les résoudre, ça en vaut la peine car cela vous permettra par la suite d'être plus rapide.

La méthode la plus accessible, si vous avez *Anaconda* comme gestionnaire, est d'aller sélectionner la bibliothèque dans l'interface graphique.

La méthode la plus utilisée repose sur un gestionnaire dédié de bibliothèques appelé `pip`<sup>6</sup>. Il va chercher la bibliothèque que vous lui demandez dans des bases de données<sup>7</sup> en ligne pour l'installer sans que vous ayez besoin de la télécharger vous-même. Il va aussi vérifier que tous les autres éléments nécessaires pour utiliser la nouvelle bibliothèque (les dépendances) sont présents et installera automatiquement ceux nécessaires. `pip` s'utilise en ligne de commande et est automatiquement installé dans les nouvelles versions de Python.

L'installation d'une bibliothèque est alors simple. Par exemple, pour installer la bibliothèque *Pandas* permettant de traiter des tableaux de données, que nous verrons en détail dans un prochain chapitre, il faut rentrer dans un terminal :

```
python -m pip install pandas
```

Cette ligne dit au terminal de lancer dans *Python* le module `pip` avec la commande `install pandas`. Dans le cas où vous êtes déjà dans un environnement Python (par exemple un terminal lancé par *Anaconda*), vous pouvez directement écrire `pip install pandas`. Dans un Notebook Jupyter, vous pouvez écrire `!pip install pandas`, le signe « `!` » signifiant à Jupyter que c'est une commande à exécuter.

---

6. `pip` est un outil qui permet de sélectionner des versions particulières, de désinstaller des bibliothèques ou encore de les mettre à jour. Pour vérifier que tout est bien installé, entrez comme ligne de commande `pip --version` dans votre terminal (ou `!pip --version` sous *Jupyter*) pour avoir le numéro de version de `pip`.

7. Une de ces bases de données est le *Python Package Index* qui permet de mettre à disposition de tout le monde les bibliothèques <https://pypi.org/>.

Ce code va aller chercher la bibliothèque `pandas` sur internet et installer la dernière version. Si tout se passe bien, la dernière ligne qui s'affiche vous dit que la bibliothèque est bien installée.

Pour supprimer une bibliothèque, la commande est similaire : `pip uninstall pandas`. Dans certains cas, si vous rencontrez un problème avec une bibliothèque, pensez à la désinstaller et la réinstaller.

Dans le cas où vous avez installé *Anaconda*, vous avez une autre commande qui fonctionne presque comme `pip` : `conda`. *Anaconda* sert alors d'interface pour l'installation en évitant d'avoir à télécharger de nouveau les bibliothèques déjà présentes. Vous pouvez utiliser la ligne de commande suivante `conda install pandas`.

### Information

#### **pip ou conda ?**

Les commandes `pip` et `conda` ne sont pas en compétition car elles couvrent des besoins différents. `pip` installe des bibliothèques spécifiques et peut avoir du mal à installer des dépendances non-python souvent indirectement nécessaires. `conda` est plus robuste pour l'installation de bibliothèques, mais ne propose pas toutes les bibliothèques disponibles avec `pip`. `conda` va souvent mettre plus longtemps à installer vos bibliothèques que `pip` et refusera d'installer une bibliothèque si elle génère des incompatibilités avec les bibliothèques déjà installées. Nous vous conseillons d'utiliser d'abord `conda` puis `pip` si la première méthode n'aboutit pas.

### 3.3.3 Quelques problèmes que vous pouvez rencontrer

En installant une bibliothèque, vous installez un morceau de code qui a été construit par une ou plusieurs personnes. Or, nous sommes tous faillibles. Plusieurs problèmes peuvent arriver.

La bibliothèque provoque des conflits avec Python : les bibliothèques sont interdépendantes entre elles et il peut arriver que certaines versions ne soient pas compatibles entre elles. Vous pouvez essayer d'installer une version antérieure en précisant explicitement la version à installer. Par exemple, vous pouvez installer une version spécifique de *Pandas* avec la commande `pip install pandas==1.0.2`.

La bibliothèque n'est pas parfaite et il y a des erreurs : si cela arrive, commencez par chercher s'il n'y a pas une solution disponible sur internet. Ensuite, pensez à regarder s'il n'y a pas une autre stratégie possible : souvent il y a plusieurs bibliothèques qui remplissent des fonctions similaires.

Dans le cas d'une mise à jour générale impossible à réaliser par l'interface *Anaconda*, vous pouvez toujours mettre à jour une bibliothèque particulière par ligne de commande.

Suivant votre version de Python, certaines bibliothèques n'existent pas. Pensez à vérifier que la bibliothèque existe bien pour votre version si jamais vous n'arrivez pas à l'installer.

## 3.4 Mise en pratique

Vous l'aurez compris, une étape cruciale est d'identifier les outils adaptés. Dans ce manuel, nous vous présenterons les principales bibliothèques utiles pour le traitement de données. Par contre, nous ne pourrons épuiser leur diversité.

Une compétence en tant que telle est de savoir identifier, installer et se familiariser avec une nouvelle bibliothèque. Au cours de vos usages, vous allez davantage approfondir votre familiarité avec certaines tandis que vous n'utiliserez qu'une fonction d'autres.

### 3.4.1 Définir ses besoins, chercher l'information

Pour présenter ces étapes, nous allons suivre un exemple en amont du traitement de données qui se rencontre souvent : la récupération d'informations sur internet. Ce sont souvent ces étapes spécifiques qui nécessitent une certaine flexibilité.

Voici une situation : imaginez que vous avez besoin de compter chaque jour certains termes apparaissant sur la page principale des journaux en ligne. Vous le faites habituellement en vous connectant avec un navigateur, en utilisant la fonction « Rechercher » de celui-ci, puis en regardant si certains mots sont présents.

Mais si vous pouviez automatiser cette opération ? L'algorithme, au sens des étapes réalisées, serait :

- ➊ définir les mots recherchés et les pages à consulter ;
- ➋ accéder aux pages à partir de leurs adresses ;
- ➌ récupérer le contenu de ces pages ;
- ➍ voir si les mots d'intérêt sont présents ;
- ➎ afficher et/ou archiver l'information.

Chacune de ces étapes nécessite des outils. Vous savez déjà comment définir des listes (pour les adresses des sites internet et des mots recherchés) et aussi comment afficher l'information. Vous savez aussi organiser un script. Par contre, vous ne savez pas comment vous connecter à internet pour récupérer une information.

Or, accéder à une page sur internet est une opération à la fois générique et fréquente : il est fort probable qu'une solution ait déjà été trouvée. Un petit tour sur un moteur de recherche avec les mots clés « Python », « récupérer » et « pages

internet » vous fait tomber sur des explications que ce que vous cherchez à faire est du *scrapping* et qu'il existe une bibliothèque pour récupérer une page internet qui s'appelle `requests`.

Toutes les bibliothèques sont complétées par une documentation qui non seulement précise comment utiliser l'outil mais en plus donne des exemples. Vous pouvez donc consulter la page de la documentation de `requests` sur son site dédié. Bien que ces informations soient souvent en anglais, vous trouverez souvent des sites en français. *Requests* est une bibliothèque dédiée à un usage particulier : prendre en charge les échanges HTTP<sup>8</sup>. Son rôle central est de faciliter cette opération pour les humains comme nous.

### Information

#### Documentation

La page de la documentation <https://2.python-requests.org/en/master/> débute avec ce texte (notre traduction) : *Requests* est une bibliothèque élégante et simple pour des échanges HTTP, construite pour une utilisation humaine.

Après un exemple qui montre la simplicité du code permettant de se connecter à un site avec un mot de passe (la simplicité côté utilisateur, car pour réaliser cette opération, il faut de nombreuses opérations que la bibliothèque va gérer) et la mise en avant de son intérêt, un guide d'utilisateur détaille la logique de la bibliothèque. La première sous-partie de l'introduction est justement intitulée « Philosophie » : en effet, tout code a des principes fondateurs qui une fois compris permettent de développer un rapport intuitif.

La partie importante pour toute utilisation est « Quickstart » (démarrage rapide) qui donne des exemples. C'est le moment pour vous d'essayer directement les morceaux de code pour comprendre la logique, en regardant ce qui fonctionne, ce qui génère des erreurs, et le type de retour. Le premier usage est celui présenté ci-dessus, d'attraper un site. Une autre fonction (`post`) permet d'envoyer des informations à un site (par exemple, remplir un formulaire).

Vous vous rendrez compte en survolant le guide d'utilisation du nombre de variations possibles : en effet, interagir sur internet nécessite potentiellement de gérer beaucoup d'informations. Cette bibliothèque permet de le faire. À garder en tête si un jour vos besoins deviennent plus complexes...

---

8. L'Hypertext Transfer Protocol (HTTP, littéralement « protocole de transfert hypertexte ») est un protocole de communication client-serveur développé pour le World Wide Web. De fait, c'est le « format » d'internet que vous consultez habituellement avec votre navigateur (le fameux « `http://` » devant les sites).

Souvent, les exemples sont suffisants pour les applications standards et vous n'aurez pas à lire toute la documentation.

Les développeurs de la bibliothèque sont partis du constat qu'une opération très fréquente des programmes est de récupérer une page internet, ce qui peut avoir plein de variantes (entrer un mot de passe, télécharger un fichier, etc.). Ils ont donc développé un outil dédié à cet usage en 2011. Dans la logique du logiciel libre, constitutive de Python, il ont rendu ce code public. D'autres contributeurs ont alors amélioré cette bibliothèque, ont corrigé les *bugs* et ont ajouté des fonctions, ce qui a permis une évolution progressive.

### 3.4.2 Utiliser une nouvelle bibliothèque

La première étape est d'installer cette nouvelle bibliothèque. Dans notre Notebook Jupyter, exécutons !pip install requests (ou une autre méthode, voir ci-dessus). Une fois installée, nous pouvons la charger.

```
import requests as req
```

Ce code importe la bibliothèque **requests** et lui donne comme nom réduit **req**.

Pour obtenir une page internet, nous utilisons la fonction **get** qui prend un lien internet et va récupérer l'information. Pour obtenir la page du quotidien Le Monde, le code se résume à une ligne :

```
req.get("http://lemonde.fr")
```

```
<Response [200]>
```

Ce code utilise la fonction **get** de **req** avec comme lien à aller récupérer la page principale du Monde. Il renvoie comme information une réponse indiquant comment s'est passée cette opération. Le 200 entre crochets est un code particulier pour les requêtes internet qui indique que « tout s'est bien passé ». Vous avez peut-être déjà été confronté à d'autres codes, en particulier le fameux « 404 – Not found ».

La prochaine étape est de traiter le contenu. Nous allons stocker la réponse de la fonction dans une variable qui va contenir (entre autres) le contenu de la page (mais aussi plein d'informations sur la manière dont la connexion a eu lieu). Ce contenu textuel est un des éléments de la réponse qui se trouve dans la variable **text** de la réponse :

```
reponse = req.get("http://lemonde.fr")
contenu = reponse.text
print(contenu[1000:1500])
```

```
95fc97ca4d6c46a5a4a46cfac2775534bee1ff/css/icons.css">
```

```
<link rel="preload" href="/bucket/6795fc/css/fonts_last.css"
as="style" onload="this.onload=null;this.rel='stylesheet'">
<noscript><link href="/bucket/6795fc97ca4d6c46a5a4a46/css/font
s_last.css"></noscript>    <script>!function(t){"use
strict";t.loadCSS||(t.loadCSS=function(){});var
e=loadCSS.relppreload={};if(e.support=function(){var
e;try{e=t.document.createElement("div");e.innerHTML="t
est un test";var f=e.firstChild;t.loadCSS=function(n,s){n
.textContent=f.textContent}}catch(e){}});e();</script>
```

Ce code :

- ➊ récupère la page du Monde avec `get` et stocke le contenu dans la variable `reponse`;
- ➋ récupère uniquement le texte de la variable `reponse` avec `reponse.text` puis le stocke dans une nouvelle variable `contenu`;
- ➌ affiche un morceau du texte avec `print` (tout afficher prendrait trop de place, là on affiche les éléments entre 1000 et 1500).

La réponse est bien le contenu de la page. Mais il s'agit de son contenu internet, c'est-à-dire qu'il n'est pas mis en forme par le navigateur et contient toutes les balises HTML, le langage de présentation. Il y a donc à la fois le texte pour les humains et pour le navigateur. Si notre but était une analyse complète, il faudrait nettoyer ces informations pour ne garder que le contenu intéressant. Mais pour le moment, cela ne nous dérange pas car nous voulons juste vérifier la présence d'un mot dans la page, par exemple le mot « science ».

Voici le code qui permet de vérifier si un mot est présent dans la page :

```
import requests as req

# Récupérer la page
url = "http://lemonde.fr"
reponse = req.get(url)
contenu = reponse.text

# Tester la présence du mot
if "science" in contenu:
    print("Le mot est présent")
else:
    print("Le mot n'est pas présent")
```

**Le mot est présent**

Ce code refait les étapes précédentes et stocke le contenu de la page dans la variable `contenu`. Ensuite, il teste si le mot « science » est dans le texte avec l'opérateur `in` qui renvoie `True` si c'est vrai. Si c'est le cas, il affiche avec `print` « Le mot est présent », sinon il continue et affiche « Le mot n'est pas présent ».

## ?

### Exercice

#### Construire une fonction générique faisant cette opération.

Nous généralisons le code précédent pour le mettre dans une fonction qui prend deux entrées : le mot à chercher et l'adresse de la page.

```
def presence_mot(mot,url):
    reponse = req.get(url)
    contenu = reponse.text.lower()
    if mot in contenu:
        return True
    else:
        return False
```

À partir de nos connaissances de *Python* et la bibliothèque `requests`, nous venons d'écrire un script qui vérifie si une information est présente dans une page. Félicitation, vous venez d'écrire votre premier « robot » ! Nous n'allons pas rentrer dans le détail sur la récupération des données sur internet, qui dépasse l'objectif de ce manuel. Mais gardez cependant en tête que la collecte automatisée est réglementée et que certains usages sont problématiques. Dans certains cas, les serveurs peuvent vous bloquer si vous réalisez trop de requêtes sur une même page<sup>9</sup>.

## ?

### Exercice

#### Testez la présence de plusieurs mots sur plusieurs pages.

Pour cela, il faut définir deux listes, celle des pages à consulter et celle des mots à rechercher, puis utiliser la fonction `presence_mot` pour récupérer l'information. La dernière étape est de stocker le résultat dans un dictionnaire.

```
pages = ["http://lemonde.fr","https://www.liberation.fr/",
         "https://www.lefigaro.fr/"]
mots = ["science", "médecine"]
resultats = {}
for p in pages:
    resultats[p]={}
    for j in mots:
        resultats[p][j]=presence_mot(j,p)
print(resultats)
```

---

<sup>9</sup>. Si vous faites beaucoup de requêtes, il se peut que vous soyez très rapidement limité par le site. Dans ce cas, le code 200 que l'on a rencontré plus haut lors de notre requête changera probablement en 429. Si cela se passe, attendez une heure ou deux avant de recommencer.

Ce code utilise deux boucles sur les pages et sur les mots. Pour chaque page, il crée un nouveau dictionnaire dans la variable `resultats` et rajoute une entrée par mot en lui donnant une valeur `True` ou `False` en fonction du résultat de la fonction.

## 3.5 Les évolutions des bibliothèques

Les bibliothèques Python, sauf exception, sont le produit du grand mouvement de l'*open source* : le code peut être utilisé par tous. Si la bibliothèque est largement utilisée, il est probable que d'autres contributeurs vont participer à son enrichissement et à son intégration dans des programmes. Le développement de bibliothèques pour l'analyse des données scientifiques n'est pas jeu à somme nulle : la collaboration a un coût mais le gain sur le long terme est souvent plus important pour tous les participants lorsque les ressources sont partagées.

Vu de loin, l'écosystème Python est alors un ensemble de bibliothèques avec chacune sa vie et ses utilisateurs. Certaines sont utilisées par l'ensemble de la communauté et ont une certaine stabilité. Elles sont mises à jour pour suivre les évolutions du langage. D'autres vont cesser d'être développées parce qu'une nouvelle bibliothèque plus puissante la remplace, ou que son principal contributeur arrête de la maintenir. Tel est le cas des bibliothèques très spécifiques développées par un unique contributeur qui, au bout de quelques années, finissent par être dépassées et incompatibles avec les autres outils ayant eux-mêmes évolués.

Cette vision dynamique du code peut être inquiétante : comment être sûr que l'on utilise la bonne bibliothèque ? Est-ce que le code que l'on écrit maintenant sera toujours valable d'ici deux ans ? Est-ce que cela vaut vraiment la peine que je m'investisse dans l'apprentissage de cette bibliothèque si elle risque changer ?

Il est vrai que certaines connaissances peuvent devenir obsolètes. Mais apprendre à utiliser Python revient justement à être capable de s'orienter dans cet univers de l'*open source* pour identifier les outils les plus adaptés et s'en saisir. Les contributeurs facilitent d'ailleurs cet apprentissage en fournissant des exemples et des documentations précis.

Pour donner un exemple concret : dans les chapitres suivants, nous verrons comment calculer les principales statistiques utilisées en SHS. Pour cela, nous allons nous appuyer sur les bibliothèques les plus utilisées actuellement mais qui ne sont pas particulièrement adaptées pour présenter les résultats de la manière dont nous avons l'habitude. Or, une bibliothèque a récemment été développée pour faire des analyses spécifiques aux SHS (les analyses factorielles). Faut-il alors délaisser les autres bibliothèques ? À y regarder de près, cette bibliothèque est développée par une seule personne et n'a pas été mise à jour depuis plus d'un an. Si elle est très utile pour certains traitements et nous évite de réécrire le code, nous ne pouvons

être sûr qu'elle soit toujours d'actualité dans un an. Alors que faire ? Dans ce cas, nous allons privilégier des bibliothèques qui continuent à vivre, et donc à s'adapter aux évolutions du langage.

### ❶ Information

#### Citer une bibliothèque

Dans un travail où vous utilisez certaines bibliothèques spécifiques pour atteindre un résultat, il est souvent intéressant de citer explicitement cette bibliothèque. Cela permet non seulement de faciliter la reproduction du code mais aussi de apporter une reconnaissance envers les contributeurs. Pour faire référence à la bibliothèque *SciPy*, vous pouvez par exemple citer Virtanen *et al.* (2020). Pour les autres références, vous pouvez trouver les articles sur ce lien : <https://www.scipy.org/citing.html>.

Cependant, nous pensons que l'important n'est pas la bibliothèque utilisée mais l'approche qui consiste à comprendre l'objectif d'une bibliothèque et de l'intégrer dans notre démarche. Dans le cas précédent, nous sommes heureux de pouvoir présenter les outils spécifiques existants sans pour autant oublier qu'il existe d'autres manières de faire, afin de ne pas nous retrouver prisonniers d'un outil.

### ❶ Information

#### Utiliser d'anciennes versions d'une bibliothèque

Il y a toujours possibilité d'utiliser une ancienne bibliothèque, même si elle n'est pas maintenue. Si l'évolution du langage peut amener des incompatibilités entre du code « ancien » et les nouveaux modules, les gestionnaires d'environnement comme *Anaconda* vous permettent de contrôler les versions du code utilisé. Il est donc toujours possible d'utiliser le code que vous avez développé. La force de l'*open source* est que tout reste disponible, même si tout n'est pas nécessairement compatible. À la différence des logiciels propriétaires qui ne sont plus accessibles une fois que leur propriétaire cesse de les développer, le code libre peut être conservé et réutilisé.

Ne vous découragez pas : la majorité des bibliothèques est bien en place. Mais gardez à l'esprit qu'il est souvent plus pertinent de comprendre la démarche générale que de devenir un expert d'un outil spécifique.

## 3.6 Synthèse du chapitre

L'intérêt de Python réside pour beaucoup dans les nombreuses bibliothèques existantes, qui sont autant d'outils que nous pouvons utiliser. Certaines sont très spécialisées tandis que d'autres sont presque toujours nécessaires.

L'installation de ces bibliothèques est facilitée par leur centralisation dans une base de données sur internet et des outils de gestion comme pip.

Si vous avez un besoin, une bibliothèque existe presque toujours : prenez le temps de chercher un peu sur les moteurs de recherche. Une fois identifiée, lisez la documentation et trouvez des exemples pour comprendre la philosophie de la bibliothèque, cela en facilitera l'utilisation.

Et puis, commencez toujours par faire un petit test pour vérifier que tout fonctionne et prendre en main les fonctions.

# Chapitre 4

## Manipuler les données

Dans ce chapitre, nous entrons un peu plus dans les logiques de programmation pour manipuler les données, ce qui va nous amener à approfondir certaines notions.

### 4.1 Pourquoi manipuler des données ?

Dans le chapitre précédent, nous avons récupéré du contenu sur des pages internet. Cependant, cette information n'est pas directement utilisable pour une analyse statistique ou une visualisation. Avant de pouvoir la traiter, il est nécessaire de la nettoyer et la stocker dans un format adapté. Ces étapes souvent peu visibles dans les résultats finaux sont pour autant nécessaires.

Nous partons ici du principe que des informations ont déjà été collectées et numérisées. Enquête par questionnaire, téléchargement de pages internet ou de fichiers déjà disponibles, vous avez sur votre ordinateur des informations que vous voulez traiter<sup>1</sup>.

Avant de pouvoir obtenir un résultat interprétable de vos fichiers, il faut les transformer. Très concrètement, cela signifie créer un script qui réalise différentes opérations : ouvrir des fichiers, lire l'information, puis les mettre dans des formats adaptés à leur visualisation et modification pour enfin les sauvegarder de nouveau dans un format plus facile à utiliser. Maîtriser ces étapes vous permet de prendre la main sur le traitement de l'information numérique et de ne plus dépendre uniquement des jeux de données déjà constitués.

---

1. À l'ère de l'*open data*, les sources de données sont très nombreuses. Si vous souhaitez trouver des informations à traiter, vous pouvez consulter le site des données publiques françaises [data.gouv.fr](http://data.gouv.fr) ou encore le site de l'Insee. Mais de nombreuses autres sources existent, que ce soit sur le climat ou sur l'utilisation des vélos en libre-service.

Il existe deux catégories de données dans un programme : les données *externes* que nous voulons traiter, et les données *internes* qui servent au bon fonctionnement du script. Les données externes sont souvent les plus visibles car ce sont elles qui vous intéressent<sup>2</sup>. Les données internes sont nécessaires comme intermédiaires : par exemple, si vous avez besoin de répéter une opération un certain nombre de fois, vous allez avoir besoin d'une variable qui joue le rôle de compteur. Pour cela, il faut maîtriser un ensemble d'opérations fondamentales pour lire, copier et transformer de l'information.

Une partie de ces manipulations peuvent être prises en charge par des bibliothèques. Cependant, cela n'est pas toujours le cas. Nous présenterons dans ce chapitre les approches bas niveau qui non seulement permettent de comprendre la logique des traitements mais aussi d'être capable de s'adapter dans le cas où des bibliothèques de haut niveau n'existent pas.

La philosophie que nous suivons est alors la suivante : partir des données brutes et construire les différentes étapes permettant d'obtenir les données utilisables, par exemple pour faire des statistiques. L'ensemble de ces étapes constitue un *workflow*, ou séquence d'actions. L'idée est de construire chaque étape du traitement (collecte, préparation, analyse, visualisation...) et de les faire fonctionner ensemble.

### Information

#### Données brutes et données raffinées

Avec certains logiciels, vous pouvez être tenté de modifier les données brutes « à la main » pour les mettre en forme ou les corriger. Nous vous recommandons de ne jamais modifier directement les données brutes mais d'écrire le code permettant de faire ce traitement, même si cela peut sembler plus long au début. Ainsi, vous gardez une trace de chacune des étapes depuis les données initiales, et cela vous permettra de refaire le traitement si les données changent ou bien si vous avez besoin de corriger une erreur.

## 4.2 Charger et sauvegarder des fichiers

L'accès à des fichiers sur l'ordinateur est une étape importante pour que votre code puisse lire des informations existantes et stocker durablement des résultats. Dans cette partie, nous allons nous concentrer sur cette étape consistant à lire de l'information et la stocker. Cette information est contenue dans des fichiers

---

2. Dans ce cas, parler de données est souvent trompeur compte tenu du temps nécessaire pour les obtenir et les mettre en forme avant de réellement les utiliser. Rien n'est véritablement *données* sans un traitement préalable.

qui peuvent être organisés de différentes manières. Cette manière d'organiser l'information porte le nom de *format*. Savoir reconnaître un format de fichier est important pour pouvoir récupérer l'information qu'il contient.

### 4.2.1 Quelques mots sur les fichiers et leurs formats

Un fichier correspond à un emplacement de la mémoire de l'ordinateur où se trouve de l'information : le contenu d'un morceau de musique, des données textuelles ou une image. Ce lien entre un fichier et le stockage physique comme le disque dur est géré par le système d'exploitation de votre ordinateur.

Concrètement, un fichier est un ensemble de 0 et 1. Ce type de fichier est qualifié de binaire. Si vous l'ouvrez, vous ne récupérez que des 0 et des 1, sans savoir si ceux-ci correspondent à une image, à un texte, ou à un morceau de musique.

Pour passer de ces 0 et 1 à l'information spécifique contenue, il est nécessaire de savoir de quel type d'information il s'agit, autrement dit, comment regrouper ces 0 et 1 pour en faire quelque chose. L'extension d'un fichier indique le type de format dans lequel l'information est codée.

Par exemple, un fichier de texte est un format **.txt**, assez facile à lire pour un ordinateur car il correspond à un fichier ne contenant que des caractères. L'ordinateur peut alors regrouper les 0 et 1 en paquets qui correspondent aux caractères. Cependant, un même texte peut aussi être stocké en format **.doc** voire **.pdf** qui, en plus de l'information sur les caractères, vont ajouter des informations sur la mise en page, la place dans le document, etc. L'ordinateur a donc besoin de savoir que ces 0 et 1 correspondent à une telle organisation. Sans savoir *a priori* le format, il est difficile de retrouver l'information contenue dans un fichier.

Chaque format de fichier impose alors des contraintes, facilite certaines opérations et peut être plus ou moins adapté à certaines manipulations. Ainsi, chaque format de fichier a alors son histoire. Par exemple, les fichiers **.pdf** sont très compliqués à utiliser pour faire du traitement de données sauf comme produit final à générer. Ils ne sont pas faits pour stocker temporairement de l'information mais plutôt pour produire une présentation finale stable à regarder sur un écran. C'est un format pensé pour les humains et non pas pour l'ordinateur.

Dans ce manuel, nous privilégierons des formats ouverts, libres et facilement manipulables.

#### Information

##### Formats de fichiers

Voici une petite liste des formats que nous vous conseillons d'utiliser :

- ⊕ .txt pour les informations textuelles de base ;
- ⊕ .csv pour les tableaux de données brutes ;
- ⊕ .png pour les images ;
- ⊕ .shp pour les fichiers géographiques ;
- ⊕ .docx pour les textes mis en formes par Word ou Libre Office ;
- ⊕ .xlsx pour les tableurs Excel et Libre Office ;
- ⊕ .json pour stocker de l'information structurée ;
- ⊕ .html pour les pages internet ;
- ⊕ .svg pour un format d'images dites vectorielles.

Il existe des formats qui ont surtout pour objectif d'informer un usage plutôt qu'un contenu. Par exemple, les formats .md pour *markdown*, .html ou .json sont en fait des fichiers textes, donc faciles à ouvrir. De même, le format .csv est en fait un fichier texte. Plus compliqués, certains fichiers textes n'ont pas d'extension marquée.

Dans certains cas, une étape peut être nécessaire pour convertir votre source initiale dans un format lisible par l'ordinateur. Cette étape peut impliquer de la numérisation d'image avec reconnaissance de texte ou l'exploitation d'interfaces dédiées. Certaines opérations sont automatisables mais dans certains cas vous ne pourrez pas éviter une collecte « à la main ». Si cela devait arriver, prenez bien le temps de réfléchir en amont des données dont vous avez effectivement besoin.

#### 4.2.2 Ouvrir un fichier

Les fichiers sont stockés sur un ordinateur. Ce contenu n'est pas directement accessible : le programme doit récupérer leur contenu pour pouvoir l'utiliser. Pour cela, une étape importante est l'étape de lecture. Inversement, l'étape d'écriture permet de conserver durablement l'information en la stockant sur le disque dur. L'information qui n'est pas écrite sur le disque dur n'existe que dans la mémoire vive de l'ordinateur et disparaît à la fin de l'exécution d'un programme<sup>3</sup>.

Lire et écrire dans un fichier sont des opérations génériques de Python. Elles se font en deux étapes : d'abord la création d'un lien vers le fichier correspondant avec la fonction `open`, qui permet d'indiquer que le fichier est actuellement utilisé et interdire son utilisation par d'autres programmes, suivi de l'action proprement dite de lecture ou d'écriture de contenu. Une fois l'opération finie, il faut fermer le fichier pour permettre à d'autres utilisateurs d'y accéder.

---

3. Elle peut même être supprimée « automatiquement » dès que le programme juge que l'information n'est plus utile, par exemple si elle n'est plus connectée à une variable utilisée.

Ces opérations peuvent se faire explicitement (ouvrir, puis fermer). Une autre solution est d'utiliser un bloc que nous verrons juste après qui gère automatiquement l'ouverture et la fermeture du fichier. Commençons par voir comment lire un fichier<sup>4</sup>.

### 4.2.3 Lire des informations dans un fichier

Lire dans un fichier correspond au mode *read*, ou « r ». Nous allons ici ouvrir le fichier des données de l'Insee sur la démographie dans les départements.

La première manière comme nous l'avons dit se fait en déclarant explicitement l'ouverture (*open*) et la fermeture (*close*) du fichier.

```
fichier = open("./data/base-pop-2015-reduite.csv", "r")
contenu = fichier.read()
fichier.close()
print(contenu[0:40])
```

DEP,P15\_POP,P15\_POPF,P15\_POP0014,P15\_POP

Ce code :

- ➊ ouvre un fichier avec la fonction **open** en donnant le chemin du fichier et le mode lecture « r » puis stocke le lien vers le fichier ouvert dans la variable **fichier**;
- ➋ lit tout le contenu de **fichier** avec la méthode **read** et le stocke dans un objet associé à la variable **contenu**;
- ➌ ferme le fichier avec la méthode **close**;
- ➍ affiche les 40 premiers éléments de **contenu**.

Cependant, nous préférons utiliser le bloc **with** qui gère l'ouverture et la fermeture. Le code ci-dessous réalise la même opération que le précédent :

```
with open("./data/base-pop-2015-reduite.csv", "r") as fichier:
    contenu = fichier.read()
print(contenu[0:40])
```

DEP,P15\_POP,P15\_POPF,P15\_POP0014,P15\_POP

Ce code :

---

4. Dans la suite de ce chapitre nous utiliserons des fichiers de données que vous pouvez télécharger sur le site du manuel. Nous supposerons que les fichiers sont situés dans un sous dossier **data** du dossier courant, avec leur noms originels, et que vos scripts sont dans le dossier courant. Les données sont dans un format **.csv**. Il s'agit d'un fichier très simple en format texte, chaque ligne correspond à un département, et chaque information est séparé par une virgule.

- ⊕ ouvre un fichier avec le bloc `with`, qui déclare l'ouverture avec `open` en donnant le chemin du fichier et le mode lecture « `r` » puis stocke le lien vers le fichier ouvert dans la variable `fichier` avec l'opérateur `as` :
  - \* lit tout le contenu de `fichier` avec la méthode `read` et le stocke dans un objet associé à la variable `contenu`, puis sort du bloc `with` qui ferme alors automatiquement le fichier ;
- ⊕ affiche les 40 premiers éléments de `contenu`.

Il existe en outre plusieurs fonctions différentes pour lire des fichiers. La fonction `readlines` permet de directement séparer les lignes du fichier dans une liste. Rappel : une ligne est un ensemble de lettres avec à la fin un « saut de ligne », qui correspond au caractère spécial `\n`. Si vous utilisez `read` vous allez voir dans votre contenu des caractères `\n`.

Une fois les données chargées, le programme peut les traiter et les manipuler dans la mémoire vive de l'ordinateur<sup>5</sup>.

Une fois le traitement réalisé par le script, l'étape finale est souvent la sauvegarde des données sur le disque dur pour les conserver durablement. Cette étape correspond à une opération d'écriture du programme vers l'ordinateur.

### ?

### Exercice

Affichez les trois premières lignes d'un fichier.

```
with open("./data/base-pop-2015-reduite.csv", "r") as fichier:  
    lignes = fichier.readlines()  
    print(lignes[0:3])
```

#### 4.2.4 Écrire dans un fichier texte

Pour écrire dans un fichier il suffit de préciser que le mode est *écriture* : « `w` » (*write*) pour remplacer ou créer et « `a` » (*append*) pour ajouter. Le mode écriture crée le fichier s'il n'existe pas et s'il existe efface celui existant. Le mode « ajout » ajoute l'information à la fin du fichier s'il existe déjà, sinon le crée. Des fonctions spécifiques comme `write` permettent ensuite d'écrire. La forme d'ensemble est similaire à la lecture que nous venons de voir.

Par exemple, si nous voulons faire une sauvegarde d'un résultat de calcul dans un fichier (ici une ligne de texte) :

---

5. Suivant la quantité de ces données, il peut être nécessaire de découper la lecture en petit blocs pour ne pas saturer l'ordinateur (on parle de morceaux, ou *chunks*). Généralement, cette situation ne se rencontre pas en SHS. Cependant, la fonction `read` permet aussi d'indiquer une lecture par morceau si nécessaire.

```
with open("./resultat.txt", "w") as fichier:
    fichier.write("Moyenne sur la population : 25")
```

Ce code :

- ✿ déclare l'ouverture d'un fichier avec le bloc `with` et la fonction `open` avec le mode écriture et l'associe à la variable `fichier` :
  - ★ ajoute une information dans le fichier avec `write`.

Attention, la fonction `write` ne finit pas la ligne par un saut de ligne. Si vous voulez qu'un saut de ligne apparaisse dans le fichier, il faut l'indiquer dans la chaîne de caractères : "Moyenne sur la population : 25\n".

## ❶ Information

### Manipuler les dossiers et les fichiers

Nous avons déjà vu les chemins de fichier. Dans certains cas, vous pourrez avoir à créer ou à rechercher des dossiers et des fichiers. La bibliothèque `os` contient la majorité des outils permettant de faire ces opérations sur l'ordinateur.

Par exemple, la fonction `mkdir` crée un nouveau répertoire. Pour créer un nouveau répertoire « Résultats » contenant un fichier texte :

```
import os
os.mkdir("Résultats")
with open("Résultats/resultat1.txt", "w") as fichier:
    fichier.write("Le résultat est : 10")
```

Autre exemple : la fonction `glob` de la bibliothèque `Glob` permet de trouver tous les fichiers d'un dossier qui contiennent un motif donné. Pour avoir la liste de tous les fichiers Excel d'un dossier :

```
import glob
fichiers_excel = glob.glob('*.xls')
```

L'étoile signifie « tous ».

## ❷ Exercice

Télécharger la page d'accueil du journal Le Monde et sauvegardez-la dans un fichier.

Nous utilisons la bibliothèque `Requests` vue dans le chapitre précédent :

```
import requests as req
```

```
info = req.get("http://lemonde.fr")
with open("page-lemonde-save.txt", "w") as fichier:
    fichier.write(info.text)
```

Remarque : le contenu de la page se trouve dans l'attribut `text` de l'objet renvoyé par `Request` auquel on accède par `info.text`.

Ensuite, ouvrez le fichier pour regarder à quoi ressemble l'information sauvegardée. Il devrait être dans le dossier courant. Son contenu est difficilement compréhensible.

#### 4.2.5 Ouvrir d'autres fichiers

Si vous utilisez la fonction `open` pour ouvrir un fichier `.pdf` ou `.xls` (ou d'autres formats) vous vous apercevrez que ce que vous obtenez est illisible. En effet, le format Excel nécessite des informations supplémentaires en plus du contenu pour organiser les données – couleurs des cellules ou largeur des colonnes. Le fichier comprend à la fois le contenu et des informations non affichées qui servent au logiciel effectuant la lecture pour savoir comment les données sont structurées. Si vous l'ouvrez avec `open`, tous ces éléments seront affichés tels que l'ordinateur les voit. Et vous ne comprendrez rien.

##### Information

#### Le format binaire

La fonction `open` vous permet de lire et d'écrire des informations sous forme de texte. C'est un format de bas niveau mais qui part du principe que le contenu correspond à une suite de caractères. Comme nous l'avons mentionné, les fichiers sont codés sur l'ordinateur en format binaire. Par exemple, une image est une information comme l'est un texte, et peut donc être représentée par un codage binaire.

Il est possible de lire tout fichier comme un fichier binaire en précisant lors de l'ouverture ou de l'écriture que le niveau d'information n'est pas du texte mais du binaire. L'écriture a pour mode « `wb` » et la lecture « `rb` », avec « `b` » pour binaire (*binary*). Si vous croisez une telle notation, vous saurez que ce qui est lu est vraiment l'information non mise en forme.

Pour manipuler ces fichiers structurés, il vous faut utiliser des outils capables de mettre en forme le contenu. C'est d'ailleurs pour cela que les logiciels de bureau-tique sont plus compliqués qu'un bloc-notes : garder le format d'un texte mis en forme avec des tableaux, des images et des couleurs nécessite de stocker un nombre important d'informations supplémentaires.

Avec Python, nous allons utiliser des bibliothèques dédiées qui ont leurs propres fonctions d'ouverture pour ces fichiers structurés. Elles ont été conçues en respectant la manière dont ces fichiers sont construits pour récupérer correctement toutes ces informations de contenu et de mise en forme.

Dans nos besoins, nous allons surtout utiliser la bibliothèque *Pandas* construite pour lire et manipuler les tableaux comme les fichiers Excel. Quand vous le pouvez, il est toujours recommandé d'utiliser des bibliothèques qui évitent d'avoir à utiliser directement les fonctions de base comme `open`. Cela évite les erreurs et rend le code plus court. Nous verrons comment faire dans le prochain chapitre.

Charger les informations n'est que le début de la manipulation des données. Nous sommes amenés à les transformer mais aussi à organiser le programme. Pour cela, il nous faut améliorer notre familiarité avec les types de données les plus fréquents, en particulier les ensembles d'éléments et les textes.

## 4.3 Manipuler les ensembles d'éléments

Prenons le temps d'approfondir notre familiarité avec les listes et les dictionnaires. Ce type de données est important dans la mesure où il permet de créer des ensembles : ensembles de nombres, ensembles de textes, ensembles d'ensembles... Nous allons les utiliser en permanence, soit parce que nos données sont elles-mêmes des ensembles, soit pour organiser notre programme.

### ❶ Information

#### Approfondissement

Si vous vous familiarisez avec le langage Python pour la première fois, cette fin de chapitre peut être passée pour directement lire le prochain chapitre davantage centré sur des opérations pratiques. Cependant, les notions vues ci-dessous sont importantes pour utiliser au mieux les outils que nous allons voir dans les prochains chapitres.

### 4.3.1 Rappel sur les listes

Une liste est une série ordonnée d'éléments. Chaque élément a deux attributs : un indice (*index*), qui est sa position dans la liste, et une valeur (*value*), qui correspond à l'information contenue. Le premier élément a l'indice 0, le dernier N-1 si la liste à N éléments. Si votre liste contient 10 éléments, les indices vont de 0 à 9.

Voici une série d'opérations que nous utiliserons de manière courante sur des listes comme ici la variable `contenu` :

- ➊ la longueur de la liste : `len(contenu)` ;

- ➊ la valeur de l'élément au rang X : `contenu[X]` ;
- ➋ le premier indice de la valeur V : `contenu.index(V)` ;
- ➌ les valeurs entre l'indice X et l'indice Y (non compris) : `contenu[X:Y]` ;
- ➍ le dernier élément de la liste : `contenu[-1]`.

### ?

## Exercice

À partir de la liste [11,2,32,34,2] créez une nouvelle liste avec les indices des valeurs supérieures à 10.

Voici une solution compacte avec la compréhension de liste. Prenez le temps de comprendre sa logique.

```
liste1 = [11, 2, 32, 34, 2]
liste2 = [liste1.index(i) for i in liste1 if i > 10]
```

Voici quelques manières de modifier la liste `contenu` :

- ➊ ajouter la valeur V à la fin de la liste : `contenu.append(V)` ;
- ➋ enlever la valeur stockée à la position X et retourner la valeur enlevée : `contenu.pop(X)` ;
- ➌ trier la liste par ordre alphabétique/numérique : `contenu.sort()` ;
- ➍ changer la valeur stockée à l'indice X : `contenu[X] = V`.

### ?

## Exercice

Faites une boucle qui affiche l'indice et la valeur de chaque élément.

```
for element in contenu:
    print(contenu.index(element), element)
```

Pour aller plus loin, vous pouvez utiliser la fonction `enumerate` qui permet de récupérer en même temps l'indice et la valeur avec une boucle `for` qui prend deux éléments.

```
for (index, element) in enumerate(contenu):
    print(index, element)
```

Pour le moment, nous avons surtout vu des listes composées d'éléments du même type. En fait, les listes peuvent contenir des éléments beaucoup plus variés. Très souvent, les données que vous allez charger sont des tableaux ou des ensembles d'éléments.

Une liste peut contenir tout type d'élément. Ils n'ont pas besoin d'être de la même sorte (des nombres et des lettres par exemple). En plus, une liste peut contenir d'autres listes : c'est une *liste emboîtée*.

### 4.3.2 Les listes emboîtées

Le document `base-pop-2015-reduite.csv` contient des données que nous voulons traiter sur la population de départements. Comme tous les fichiers tableurs, c'est un ensemble de lignes, et chaque ligne a plusieurs informations. Nous ne voulons pas lire l'information comme un bloc mais au contraire avoir un tableau qui sépare les informations.

Une première étape est de lire les lignes du fichier :

```
with open("data/base-pop-2015-reduite.csv", "r") as f:
    contenu = f.readlines()
print(contenu[0][0:40])
```

DEP,P15\_POP,P15\_POPF,P15\_POP0014,P15\_POP

Ce code :

- ➊ ouvre un fichier avec le bloc `with` et `open` en mode lecture `r`;
- ➋ lit les lignes avec `readlines` et met la liste des lignes dans la variable `contenu`;
- ➌ affiche avec `print` les quarante premiers caractères du premier élément de la variable `contenu`.

Dans notre exemple, la première ligne de notre fichier contient les informations de description des colonnes.

Si nous voulons connaître le nombre de lignes décrivant des départements dans notre fichier, on peut supprimer le premier élément qui contient juste le nom des colonnes et afficher le nombre d'éléments.

```
print("La taille avant pop est", len(contenu))
premier_element = contenu.pop(0)
print("La taille de la liste est maintenant ", len(contenu))
```

La taille avant pop est 101  
 La taille de la liste est maintenant 100

Ce code :

- ➊ affiche la taille initiale du nombre de lignes ;
- ➋ utilise la méthode `pop` pour à la fois supprimer l'indice 0 de la liste `contenu` (la liste est modifiée) mais permet aussi de récupérer ce premier élément que nous stockons dans la variable `premier_element` (nous faisons deux étapes en une) ;
- ➌ affiche la taille de la variable `contenu`.

Dans notre cas, le fichier que nous avons chargé est un tableau : chaque ligne du fichier initial comprend des informations différentes, séparées par des virgules, que nous avons envie de récupérer indépendamment. Pour séparer ces éléments, nous pouvons utiliser la méthode `split` qui coupe une chaîne de caractères au niveau des délimiteurs voulus (nous verrons plus la manipulation du texte dans la partie suivante).

Pour transformer chaque entrée en une liste, nous pouvons faire une boucle sur chaque ligne et appeler la méthode `split` :

```
donnees = []
for ligne in contenu:
    ligne_decoupe = ligne.split(",")
    donnees.append(ligne_decoupe)
print(donnees[0][0:4])
```

[‘01’, ‘631877’, ‘319925.910897514’, ‘128088.557317217’]

Ce code :

- ➊ crée une nouvelle variable `donnees` qui est une liste vide ;
  - ➋ fait une boucle `for` sur tous les éléments de la variable `contenu` (les lignes du fichier). À chaque étape :
    - \* `ligne` prend une valeur de la liste donc une ligne du fichier ;
    - \* applique la méthode `split(",")` à `ligne` pour couper le texte à chaque virgule et renvoie une liste des éléments, stockée dans la variable `ligne_decoupe` ;
    - \* ajoute cette liste à la liste `donnees` avec `append` ;
  - ➌ affiche les 4 premiers éléments du premier élément de cette liste de listes.
- Insistons : l’écriture `[0:4]` sélectionne une tranche d’une liste incluant l’élément 0 jusqu’à l’élément avant 4 (donc `contenu[4]` n’est pas inclus).

Dans code précédent, nous avons enlevé la première ligne de `contenu`, avec les entêtes de chaque colonne. Les noms des colonnes étaient `DEP,P15_POP,P15_POPF,P15_POP0014,...`. Nous pouvons donc interpréter la ligne ci-dessus comme le département 01, ayant une `P15_POP` valant ‘631877’.

Cette liste a maintenant deux dimensions, comme un tableau. Il est alors possible d’accéder à un élément de ce tableau avec deux coordonnées<sup>6</sup>. Pour afficher la population féminine de la cinquième ligne :

```
print(donnees[4][2])
```

72026.7773473066

---

6. Le numéro de la ligne est souvent appelé `i` et le numéro de la colonne est souvent appelé `j`.

La cinquième ligne est à l'indice 4, et le nombre de femmes vivant dans le département est sur la troisième colonne, donc d'indice 2. Cependant, pour l'instant cette information est encore du texte. Il n'est pas possible de faire directement des opérations mathématiques. Il est nécessaire de transformer le texte pour avoir le bon format : creusons un peu plus la manipulation du texte.

## 4.4 Traiter du texte

Le texte est le principal support d'échange de l'information pour les humains. Il est fréquent de le manipuler : le découper, mettre une majuscule, trouver une information précise dans un grand texte, réunir plusieurs petits morceaux de textes ensembles, ou le mettre en forme de manière plus élaborée pour être lu par un humain.

### 4.4.1 Il faut qu'on parle... d'encodage

Un problème récurrent avec les textes est l'*encodage*. C'est-à-dire ? Nous l'avons dit, une information est une série de 0 et de 1 sur votre ordinateur. Une lettre est un groupe de 0 et de 1. Mais où commence une lettre et où finit-elle ? Comment l'ordinateur repère que ce groupe de 0 et 1 correspond à une lettre ?

Les règles d'encodage définissent comment lire cette information afin de la faire correspondre à un texte. En fonction des systèmes d'exploitation et les logiciels, et historiquement les régions du monde, les encodages ne sont pas les mêmes. Par exemple en `latin1` le 236ème caractère est `é`, mais en `windows-1252` (cyrilique) le 236ème caractère est un caractère qui ne s'affiche pas dans ce manuel.

Si un fichier est ouvert avec le mauvais encodage, vous allez vous retrouver avec un texte où les accents auront disparu et avec l'apparition de lettres bizarres. Parfois, cela empêchera même l'ouverture du fichier. Si cela vous arrive, pensez tout de suite : problème d'encodage !

De manière automatique, Python est en encodage dit `utf-8` ou `unicode` (par abus de langage). Utf-8 supporte dans le même format plus d'un million de symboles différents (`latin`, `arabe`, `grec`, `cyrilic...`) et permet de les intégrer dans le même fichier. Windows est généralement en `latin1`. Il existe d'autres formats d'encodage usuels : `ascii` par exemple. Si cela vous paraît compliqué, c'est normal : les questions d'encodage sont très prise de tête.

Le moment important pour préciser l'encodage est généralement quand vous ouvrez des fichiers. Pour préciser l'encodage (si vous le connaissez), la fonction `open` a une option spécifique. Par exemple, pour essayer de lire le fichier avec l'encodage `latin1` :

```
with open("./data/base-pop-2015-reduite.csv", "r",
          encoding="latin1") as fichier:
    contenu = fichier.read()
```

Ce code :

- ➊ utilise le bloc `with` pour ouvrir le fichier avec `open` comme précédemment, en précisant l'encodage de lecture du texte `encoding="latin1"` (on rajoute un saut de ligne pour la présentation du code, cela ne change pas l'exécution) ;
- ➋ lit le contenu du fichier puis l'associe à la variable `contenu`.

Si vous ne connaissez pas l'encodage utilisé, une bonne manière est d'ouvrir le fichier, voir si le texte est « propre » (tous les éléments sont bien représentés, les lettres avec des accents peuvent être affichées). Si ce n'est pas le cas, et que vous ne connaissez pas l'encodage du texte, essayez différents encodages par essais/erreurs<sup>7</sup>.

#### 4.4.2 Mettre en forme le texte

Un texte est avant tout un ensemble de symboles (lettres, chiffres, ponctuation...). Vous pouvez donc sélectionner une lettre, ou un ensemble de lettres, à partir de sa place dans la liste avec `l'index`. Nous l'avons déjà croisé dans des exemples où on sélectionnait une tranche dans un texte. Ainsi, prendre uniquement les 100 premières lettres d'un texte revient à prendre les 100 premiers éléments d'un tableau, ce qui peut s'écrire : `contenu[0:100]`.

Il est possible de transformer d'autres formats de données, et en particulier des nombres, en texte avec la fonction `str`. À l'inverse, il est possible dans certains cas de transformer certains textes en nombres, dans la mesure où la forme dans laquelle ils sont écrits correspond à l'écriture d'un nombre.

#### ?

#### Exercice

Transformez le texte « **56.45** » en nombre.

```
nombre_en_texte = "56.45"
nombre_float = float(nombre_en_texte)
```

Deux méthodes sont utiles pour prendre en charge les majuscules/minuscules (la casse). Pour mettre un texte entièrement en majuscule on utilise les fonctions `upper` ou en minuscule `lower`. La fonction `capitalize` permet d'avoir uniquement la première lettre en majuscule.

---

7. Dans certains cas, les problèmes d'encodage peuvent être *vraiment* pénibles. Si cela devait vous arriver, des solutions de contournement peuvent être utilisées : convertir en enregistrant le document avec un logiciel de traitement de texte directement dans le bon format par exemple.

## ?

### Exercice

Mettez le contenu du fichier département en minuscule.

```
contenu = [i.lower() for i in contenu]
```

Enfin, certaines méthodes permettent d'agir sur les chaînes de caractères. Nous avons vu précédemment la méthode `split` qui permet de couper le texte à chaque fois que se trouve un élément séparateur. Une autre méthode très utile est la méthode `replace` qui remplace un ensemble de lettres par un autre. Par exemple, nous voulons éviter les espaces dans les noms composés des villes et les remplacer par des tirets bas pour en faciliter l'utilisation.

```
contenu = [i.replace(" ","_") for i in contenu]
```

Ce code utilise la fonction `replace` dans une compréhension de liste pour remplacer dans chaque élément de `contenu` les espaces blancs par un tiret bas.

### 4.4.3 Intégrer des variables

Comment intégrer l'information de variables dans un texte ? Par exemple, pour présenter le résultat d'un traitement. Une première idée est de transformer ce contenu (par exemple des nombres) en texte avec `str()` puis le concaténer (ajouter) au reste de l'information textuelle.

Il existe des méthodes de *formatage* dédiées qui permettent de répondre au besoin suivant : j'ai des informations dans différentes variables et je veux les intégrer dans un texte pour avoir un résultat final, indépendamment de la valeur spécifique de la variable. La fonction de formatage `format` est bien utile pour cela et permet de décrire où mettre l'information d'une variable dans un texte sans préciser au moment de la programmation la valeur de cette information<sup>8</sup>. C'est un peu comme laisser un espace qui sera ensuite rempli par la valeur finale au moment où le programme va être lancé.

L'idée est d'écrire un texte en mettant des accolades « {} » à l'endroit où vous voulez mettre le contenu d'une variable, puis de renseigner dans la fonction `format` le nom des variables à utiliser.

```
nb_region_fr = 18
print("Il y a {} régions en France".format(nb_region_fr))
```

Il y a 18 régions en France

<sup>8</sup>. Il existe plusieurs stratégies de mise en forme des textes. L'ancienne méthode utilise %, progressivement remplacée par `format`. Plus récemment, Python intègre les chaînes de caractères littérales présentées dans un encadré par la suite. Ne soyez pas surpris de voir différentes méthodes. Les trois fonctionnent.

Ce code :

- ➊ définit une variable `nb_region_fr` ;
- ➋ affiche une chaîne de caractères en utilisant la fonction `format` pour mettre dans le texte l'information du nombre de régions au bon endroit.

Vous pouvez non seulement mettre plusieurs variables différentes dans le texte, mais aussi indiquer le numéro de la variable à mettre si vous voulez mettre plusieurs fois la même (ainsi, `{1}` correspond à la deuxième variable passée en argument).

Pour plus de flexibilité, vous pouvez aussi préciser le type de la variable à afficher. Si vous avez un nombre à virgule et que vous voulez l'afficher comme un nombre entier (ce qui peut arriver quand vous n'avez pas d'intérêt à mettre les décimales dans un texte), il suffit de préciser le type entier (avec le mot-clé `d`) : `{0:d}`. Pour afficher un nombre à virgule en limitant à deux décimales, ce sera `{0:.2f}`, `f` étant le mot clé pour les nombres à virgules, et `.2` indiquant que nous souhaitons 2 chiffres significatifs.

### ❶ Information

#### f-strings vs .format()

Les versions les plus récentes de Python introduisent les chaînes de caractères littérales, ou *f-strings*, permettant de rendre plus fluide le formatage du texte. Au lieu d'utiliser la méthode `.format()` vous pouvez précédé la chaîne de caractères de `f` et insérer les variables directement en les nommant à l'intérieur d'accolades.

```
nb_region_fr = 18
print(f"Il y a {nb_region_fr} régions en France")
```

Dans ce manuel, nous allons utiliser `format` pour l'instant plus répandue. Mais libre à vous de préférer une autre méthode, d'autant que les *f-string* sont plus faciles à lire.

Formater un texte est très utile pour restituer des informations à partir de votre analyse de données. Cela permet par exemple de créer une fonction qui affiche le résultat d'un traitement de manière claire. Voyons maintenant une application concrète.

#### 4.4.4 Traiter des données réelles

Combien de chefs-lieux français ont des noms composés (comme Charleville-Mézières) ? Cette opération nécessite plusieurs étapes : il faut ouvrir un fichier comprenant ces informations, récupérer le nom des chefs-lieux, identifier la présence d'un marqueur de nom composé (un tiret ou un espace), créer un compteur pour compter cette information. Pour cela, il faut manipuler des données tex-

tuelles, des listes et créer des données internes pour structurer le programme. Tout d'abord, chargeons les données sur les départements car nous voulons travailler sur le nom des chefs-lieux de département<sup>9</sup>.

```
with open("./data/departements-francais.csv", "r") as file:
    contenu = file.readlines()
donnees = [ligne.split(",") for ligne in contenu]
print(donnees[1][0:5])
```

['01', 'Ain', 'Auvergne-Rhône-Alpes', 'Bourg-en-Bresse', '5']

Ce code :

- ➊ ouvre le fichier `departements-francais.csv` avec le bloc `with` et la fonction `open`;
- ➋ lit les lignes avec `readlines` et stocke l'information dans `contenu`;
- ➌ fait une boucle (avec une compréhension de liste) sur l'ensemble des lignes, et pour chaque ligne coupe au niveau des virgules pour créer une nouvelle liste de liste, `donnees`;
- ➍ affiche les 5 premiers éléments de la deuxième entrée de la liste.

Le résultat n'est pas exactement sous la forme dont nous le voulons : des guillemets « " » dérangent et des caractères spéciaux de sauts de ligne « \n » sont encore là à la fin. Ce serait bien de créer une fonction qui enlève ces éléments dérangeants. Cette fonction serait une fonction de nettoyage qui permettrait de bien préparer nos données. Pour cela, nous allons utiliser la fonction `.replace()` qui remplace une chaîne de caractères par une autre. Ici, la fonction doit remplacer le guillemet et le saut de ligne par « rien ».

```
# Définir une fonction de nettoyage
def nettoyage(ligne):
    nouvelle_ligne = ligne.replace('"', '')
    nouvelle_ligne = nouvelle_ligne.replace("\n", "")
    return nouvelle_ligne

# Charger les données
with open("./data/departements-francais.csv", "r") as file:
    contenu = file.readlines()

# Appliquer et afficher
donnees = [nettoyage(ligne).split(",") for ligne in contenu]
print(donnees[1][0:5])
```

['01', 'Ain', 'Auvergne-Rhône-Alpes', 'Bourg-en-Bresse', '5']

---

9. Ce fichier est aussi disponible sur le dépôt du manuel.

Ce code :

- ➊ définit une fonction qui prend un texte en entrée :
  - ★ remplace les guillemets de l'entrée par « rien » puis stocke le résultat dans `nouvelle_ligne`;
  - ★ remplace les sauts de ligne par « rien » puis stocke le résultat dans `nouvelle_ligne`;
  - ★ renvoie la ligne nettoyée;
- ➋ ouvre le fichier des départements en lecture :
  - ★ lit les lignes du fichier et les stocke dans `contenu`;
- ➌ construit une variable `donnees` qui est la liste des lignes nettoyées puis découpées au niveau des virgules;
- ➍ affiche les 5 premiers éléments de la deuxième entrée de la liste.

L'information sur le chef-lieu est le quatrième élément de la ligne. Nous pouvons maintenant compter le nombre de noms de villes qui contiennent soit un tiret « - » soit un espace « » pour savoir combien de chefs-lieux ont des noms composés. Pour cela, nous utiliserons la fonction `count` qui compte le nombre de fois où un motif est présent :

```
nb_compose = 0

for ligne in donnees:
    nom = ligne[3]
    if (nom.count("-") > 0) or (nom.count(" ") > 0):
        nb_compose += 1

print("Il y a {} noms composés".format(nb_compose))
```

Il y a 18 noms composés

Ce code :

- ➊ définit une variable `nb_compose` qui va servir de compteur et qui est initialisée à 0 ;
- ➋ fait une boucle sur la liste `donnees` avec la variable `ligne` :
  - ★ met le nom de la ville dans la variable `nom`;
  - ★ teste si le nom du chef-lieu contient un tiret ou un espace en regardant si le nombre de ces éléments est supérieur à 0 :
    - si c'est le cas, le compteur augmente avec `+ = 1` (sinon, le compteur ne bouge pas);
- ➌ affiche la valeur du compteur en utilisant la méthode `format`.

Nous commençons à voir ici la complexité du traitement des données réelles : que se passerait-il si un des champs avait une apostrophe ? une virgule ? un guillemet mal placé ? Pour cette raison, il est important de bien maîtriser ses données et tester régulièrement si elles sont dans le bon format.

À partir des fonctions de base, nous avons donc ouvert un fichier et manipulé les données en utilisant les propriétés des textes et des listes. Bien sûr, nous n'avons pas épuisé tous les traitements possibles. Ces éléments de base permettent de faire beaucoup de traitements, et de nombreuses fonctions ont été créées pour manipuler les listes et les textes. Pour donner un exemple de ces traitements avancés, regardons rapidement le cas de la recherche d'informations dans un texte.

#### 4.4.5 Introduction aux expressions régulières

Un texte peut être composé de beaucoup d'informations différentes, et nous souhaitons souvent identifier la présence d'un élément particulier dans une chaîne de caractères. Plusieurs solutions existent pour chercher des éléments. Par exemple, l'opérateur `in` que nous avons déjà croisé permet de tester si un élément est présent dans un texte et la méthode `count` que nous avons déjà croisée permet de compter le nombre de fois où un élément est présent.

##### ?

#### Exercice

##### Identifiez les départements dont le nom mentionne le Rhin.

Nous pouvons utiliser l'opérateur `in` : le code "`Rhin`" `in` `texte` renvoie `True` si le « Rhin » est dans la variable `texte` et `False` sinon. La ligne prendra la forme suivante `if "rhin" in ligne:`.

Pour des recherches plus complexes, nous pouvons utiliser les *expressions régulières* aussi parfois appelées *expressions rationnelles* ou encore *regex* pour *regular expression*. Nous présentons ici leur existence pour souligner que le traitement du texte peut reposer sur des outils très efficaces facilitant largement l'automatisation des traitements.

En effet, trouver un élément déterminé est assez facile. Mais comment faire quand vous ne savez pas exactement l'information que vous cherchez mais que vous êtes capable de la décrire en général : « c'est 4 chiffres », « c'est un mot qui se trouve après un tiret », « il y a une virgule avant et une virgule après » ?

Une bibliothèque dédiée permet d'utiliser des expressions régulières permettant de décrire la forme de ce que l'on cherche<sup>10</sup>. L'idée est de décrire la forme que prend l'élément cherché et puis analyser le texte pour faire ressortir toutes les séquences de symboles qui respectent cette forme.

*Attention, utiliser des expressions régulières peut être compliqué dès que l'on dépasse des cas simples. C'est normal de ne pas trouver cette approche évidente.*

<sup>10</sup>. Notons que les expressions régulières ne sont pas uniques à Python et les règles existantes dans d'autres langages sont similaires.

Prenons un exemple : dans les articles de presse, les lieux des événements sont souvent suivis de la mention du département entre parenthèses, comme pour « À Marseille (Bouches-du-Rhône) la pratique du vélo se développe difficilement, surtout en comparaison avec Strasbourg (Bas-Rhin) ». Après avoir constaté que les départements sont mentionnés entre parenthèses, notre objectif est de récupérer toutes les informations entre parenthèses. Dit autrement, nous voulons dire à notre code : *prend toutes les lettres entre le début et la fin de la parenthèse, y compris les tirets*. Cela s'écrit : `\((.*?)\)`.

Construire des *masques* d'expression régulière (le terme *masque* désigne cette forme qu'on décrit) tient presque de la magie noire. Donc cela nécessite un peu d'expérience et beaucoup d'essais/erreurs. S'il est assez facile de trouver des nombres ou des mots, cela peut être un peu plus subtil pour des formes complexes, avec des conditions.

Le masque pour prendre le texte entre des parenthèses `\((.*?)\)` peut être un peu compliqué car il mélange plusieurs éléments : il y a des parenthèses qui disent ce que l'on veut garder au final, et les parenthèses dites « échappées » qui sont les éléments du texte à analyser permettant d'identifier l'information dans le texte, et qui doivent être déclarées comme caractère spécial pour ne pas faire des interférences avec les autres.

Décrivons cette expression régulière spécifique :

- ➊ tout d'abord il y a « \(`` qui correspond à une parenthèse dans le texte, qui est un caractère spécial échappé avec « \` ;
- ➋ ensuite, il y a la parenthèse simple ( qui marque le début de l'information que l'on souhaite garder à la fin de la recherche ;
- ➌ ensuite il y a le point . qui signifie « n'importe quel élément » ;
- ➍ puis l'étoile \* qui dit « autant de fois que possible » ;
- ➎ puis le point d'interrogation ? qui dit de s'arrêter dès que possible ;
- ➏ ensuite on ferme la parenthèse de l'expression régulière pour dire : « c'est bon, on garde l'information jusqu'ici » ;
- ➐ puis on met la parenthèse du texte qui permet de marquer la fin de ce qu'on cherche, aussi échappée comme au début « \)` .

Pour construire des expressions régulières, le plus simple est d'abord de définir les règles que vous voulez appliquer, puis d'aller consulter la documentation spécifique pour expérimenter sur des exemples et petit à petit construire le masque. De très bons tutoriaux vous permettent de trouver les astuces pour construire ces masques. Et les forums sont pleins d'exemples. Si vous avez une application particulière, prenez le temps de construire le masque adapté. Ça vous simplifiera la vie ensuite.

Pour utiliser l'expression régulière, le module dédié déjà installé dans Python est `re` pour *regular expression*. Certaines fonctionnalités plus avancées nécessitent d'installer `regex`, qui est une bibliothèque plus étendue. Pour faire le traitement mentionné précédemment avec *Regex* :

```
import regex as re

texte = "A Marseille (Bouches-du-Rhône) la pratique du vélo\
         se développe difficilement, surtout en comparaison\
         avec Strasbourg (Bas-Rhin)."
departements = re.findall("\((.*?)\)",texte)
print(departements)
```

[ 'Bouches-du-Rhône' , 'Bas-Rhin' ]

Ce code :

- ➊ charge la bibliothèque d'expressions régulières `regex` abrégée en `re` ;
- ➋ définit le texte à analyser dans `texte` en forçant le saut de ligne pour la présentation ;
- ➌ utilise la fonction `findall` pour rechercher tous les éléments qui correspondent au motif donné en premier argument dans le texte `texte` donné en deuxième argument puis stocke le résultat dans `departement` ;
- ➍ affiche le résultat avec `print`.

Bien entendu, nous n'avons qu'effleuré la puissance des expressions régulières. Ce qui est important à retenir est que vous pouvez construire des masques dès que vous avez besoin de rechercher certains éléments dans un texte et que vous pouvez décrire les conditions de cette recherche. Si cela peut prendre un peu de temps pour des cas compliqués, pensez à cette solution dès que vous avez à faire cette opération de manière systématique ou sur des grands textes. Nous ne développerons pas plus la technique des expressions régulières, mais nous retrouverons son usage dans des exemples au cours du manuel.

### ?

### Exercice

Faites un masque d'expression régulière qui récupère tous les groupes de quatre chiffres.

Un chiffre est un caractère de 0 à 9, qui se marque [0-9]. Comme on veut quatre chiffres, le masque est le suivant ( $[0-9]^{\{4\}}$ ). Vous pouvez donc rechercher ces éléments avec `re.findall("[0-9]^{\{4\}}",texte)`.

## 4.5 Complément : attention aux attributions

Avant de conclure ce chapitre, nous souhaitons attirer votre attention sur un point lié à la manipulation des données. Celle-ci peut être compliquée avec les objets dans la mesure où ils peuvent avoir plusieurs noms et être appelés de différentes manières. Ainsi, utiliser une variable peut soit conduire à accéder au contenu de

l'objet, soit faire un lien avec l'objet rattaché. Pour le dire autrement, une variable peut pointer vers des informations ou vers une autre variable. Cela peut modifier vos opérations surtout quand vous faites appel aux méthodes. Nous mentionnons cette possibilité car elle peut être la source d'erreurs dans votre programme.

Pour comprendre, prenons le cas de la modification d'une liste.

```
a = ["mange"]
b = a
b = b + ["moi"]
print(a, b)
```

```
['mange'] ['mange', 'moi']
```

Ce code :

- ➊ crée une liste avec un élément dans la variable **a**;
- ➋ crée la variable **b** comme égale à **a**;
- ➌ ajoute la valeur de **b** à une autre liste et stocke de nouveau dans **b** ;
- ➍ affiche les deux listes.

Les deux listes sont différentes. En effet, la troisième ligne ajoute le contenu de **b** à **moi** pour créer une nouvelle liste stockée dans **b**. La valeur de **b** est évaluée puis additionnée puis de nouveau stockée.

Par contre, le code suivant a un effet différent :

```
a = ["mange"]
b = a
b.append("moi")
print(a, b)
```

```
['mange', 'moi'] ['mange', 'moi']
```

Il ressemble au code précédent sauf que l'ajout se fait avec la méthode **append**. En fait, l'opération **b = a** ne copie pas l'objet de **a** dans une nouvelle variable **b** mais fait que **b** est en lien avec l'objet de **a**. Quand on appelle la méthode **append**, c'est celle du même objet que celui stocké dans **a**. Et donc les deux variables continuent à être associées au même objet.

Certaines méthodes, comme **.append()** vont modifier l'objet auquel elles font référence. D'autres, comme **.drop()**, vont renvoyer une copie. Comme cela dépend des méthodes. Le mieux reste d'être attentif à ces opérations. Dans la mesure où cela va dépendre de la manière dont sont construits les objets, une stratégie est souvent de vérifier que les objets que vous manipulez contiennent bien ce que vous voulez.

## 4.6 Synthèse du chapitre

Ce chapitre traite d'une étape importante et souvent nécessaire : la manipulation des données de bas niveau.

Bien comprendre le comportement de ces entités (listes, textes, nombre) est une étape importante pour pouvoir aborder sereinement les traitements plus avancés. Vous aurez toujours besoin de manipuler ces types de données.

Les informations externes sont regroupées dans des fichiers à charger et à mettre en forme pour pouvoir les utiliser. Dans de nombreux cas, ces informations sont du texte ou des listes qu'il faut nettoyer et mettre en forme dans un tableau. Nous allons les retrouver dans les prochains chapitres avec d'autres bibliothèques plus avancées.



# Chapitre 5

## Traitement de données avec *Pandas*

Ce chapitre présente la bibliothèque *Pandas* dédiée à la manipulation des données. Cette bibliothèque a été créée pour traiter de larges quantités de données organisées sous la forme de tableaux. Elle permet, entre autres, de faciliter le nettoyage des données et les traitements statistiques.

### 5.1 La bibliothèque *Pandas*

Nous l'avons mentionnée rapidement dans le chapitre sur les bibliothèques : *Pandas* est un peu « l'Excel » de Python. Si cette comparaison peut déranger certains, tant comparer une bibliothèque de programmation gratuite à un logiciel payant utilisé pour la bureautique peut sembler étrange, cela vous donne une idée de son usage : produire, modifier et analyser des tableaux de données variés<sup>1</sup>. Les fonctions de *Pandas* permettent d'accéder aux données, les transformer, et produire des statistiques ou des graphiques de manière souple et, osons le mot, intuitive.

Il n'y a que des bonnes raisons d'utiliser *Pandas* : une structure adaptée pour les tableaux et les fichiers issus de tableurs, des fonctions pensées pour l'exploration des données et leurs visualisations et une très forte compatibilité avec les autres bibliothèques Python<sup>2</sup>. Ne nous croyez pas sur parole : nous allons vous le montrer.

---

1. Nous n'avons rien contre les tableurs. Au contraire, savoir bien utiliser un tableur pour faire certains traitements peut être complémentaire avec la maîtrise de la programmation.

2. *Pandas* sert souvent de référence et des bibliothèques spécifiques vont utiliser un vocabulaire et des conventions de nommage similaire.

## ❶ Information

### Histoire de *Pandas*

*Pandas* est une bibliothèque développée par Wes McKinney à partir de 2008 pour répondre à ses besoins d'analyse financière et de traitement d'évolutions temporelles. La bibliothèque ne vient donc pas directement de la communauté scientifique mais est très vite devenue une brique importante de l'arsenal du traitement de données en raison de sa souplesse et de sa facilité d'intégration. Elle est maintenant utilisée par des millions de personnes dans de nombreux domaines avec des évolutions progressives pour intégrer de nouvelles fonctions.

*“Scientists unnecessarily dealing with the drudgery of simple data manipulation tasks makes me feel terrible”* [Wes McKinney]

Wes McKinney a écrit un livre de référence sur Pandas McKinney (2012).

Une première étape est d'installer la bibliothèque<sup>3</sup>, puis de la charger :

```
import pandas as pd
```

Il est commun d'abréger **pandas** en **pd**, nous utiliserons donc cette convention dans la suite. Pour simplifier l'affichage des tableaux *Pandas* dans ce manuel, nous allons changer quelques paramètres afin de limiter la quantité d'information affichée sur ces pages. Nous indiquerons de ne pas afficher plus de 80 caractères de large sur une limite de 6 colonnes et en se limitant à 1 chiffre après la virgule pour les valeurs numériques<sup>4</sup>.

```
pd.set_option('display.width', 80)
pd.set_option('display.max_columns', 5)
pd.set_option('precision', 1)
pd.set_option('max_colwidth', 12)
```

Ces options définissent la largeur totale du tableau, le nombre de colonnes affichées, le nombre de chiffres après la virgule, et la taille de chaque colonne.

Nous présenterons un sous-ensemble de tous les outils de *Pandas* dans la mesure où il existe beaucoup d'opérations possibles. La bibliothèque est en effet très vaste. Notre philosophie est de vous rendre capable de faire les traitements utiles pour

3. !pip conda install pandas dans Jupyter.

4. D'autres options permettent de paramétriser les affichages. Nous ne travaillerons pas avec les dates, mais il est par exemple possible de changer le format d'affichage des dates car les Anglo-saxons préfèrent le format Année/Jour/Mois qui peut être déroutant. Nous présentons ces options pour que vous ayez en tête que vous pouvez configurer l'affichage. Par la suite, elles seront cachées dans les prochains chapitres pour ne pas alourdir le contenu.

les SHS. Nous faisons le choix de nous concentrer sur un sous-ensemble de ces opérations qui permettent les manipulations tout en limitant le nombre d'éléments à mémoriser (il y a plus de 240 fonctions et attributs associés aux tableaux *Pandas*). Cela signifie aussi qu'il existe, comme souvent, plusieurs manières de réaliser la même opération. L'important est de trouver celle qui vous convient. Soyez juste averti qu'il existe des philosophies différentes.

## 5.2 Charger des données à partir d'un fichier

La bibliothèque *Pandas* permet de charger différents formats de fichier, en particulier les fichiers Excel (extension `.xlsx`, `.xls` et `.csv`)<sup>5</sup>. Pour charger le fichier des données sur les villes, nous allons utiliser un des outils de la bibliothèque<sup>6</sup>.

Le code suivant va charger des données d'un fichier *Excel* (format `.xls`) pour permettre de pouvoir manipuler les données.

```
tableau = pd.read_excel("./data/base-pop-2015-communes.xls")
```

Une ligne suffit donc pour charger les données. La fonction `read_excel` nécessite le chemin du fichier, qui se trouve dans le dossier `data` et qui a pour nom `base-pop-2015-communes.xls`. Cette fonction va lire le fichier (en faisant toutes les étapes de `open` et de lecture du fichier) et renvoyer le contenu. Celui-ci est ensuite stocké dans la nouvelle variable `tableau`, qui va donc contenir l'information du fichier.

### Exercice

#### De quel type est la variable `tableau` et quelle est sa taille ?

Utiliser la fonction `type` pour connaître le type de la variable. Utiliser la fonction `len` pour connaître la taille du tableau.

Il est important de noter qu'en faisant cette opération vous lisez uniquement le fichier. Cette opération ne le modifiera pas. Pour sauvegarder les modifications que vous allez faire aux données, il faudra réaliser une opération d'écriture, soit dans le même fichier, soit dans un nouveau.

5. Pandas permet aussi de lire des bases de données SQL, HTML, SPSS, JSON, CSV ou TSV. Par exemple, pour lire un fichier SPSS (.sav), la fonction est `read_spss`, avec un module spécifique à installer `pyreadstat`. Pandas permet aussi de lire et d'écrire des fichiers compressés, il suffit de rajouter le suffix `.zip` (ou `.gz`) sur les fichiers.

6. Dans certains cas, l'ouverture de fichier `.xls` demande une bibliothèque supplémentaire. Si vous avez un message d'erreur, lisez le texte et si ce message dit qu'il manque quelque chose, essayez d'abord de l'installer. Aussi, le chemin vers le fichier suppose que vous avez mis le dossier `data` dans le même endroit que votre code.

## 5.3 Le format des données *Pandas*

La bibliothèque *Pandas* permet d'utiliser un nouveau type d'objet, le *DataFrame*, que l'on va appeler aussi par la suite tableau *Pandas* pour plus de familiarité et moins d'anglicisme. Concrètement, il s'agit d'un tableau réunissant des colonnes. Chaque colonne est une série de type *Series* (un autre objet de *Pandas*). Notez que vous pouvez avoir une série indépendamment d'un tableau.

Pourquoi utiliser de nouveaux types d'objets et pas simplement des listes ? Pour avoir des outils spécifiques qui nous intéressent et qui vont nous éviter de programmer nous-mêmes toutes les opérations : compter le nombre de fois où apparaît un élément, faire des sommes, des moyennes ou des graphiques. Chaque objet est doté de ses méthodes permettant de réaliser ces opérations qui vont rendre interactif et intuitif le traitement des données, surtout si vous utilisez le *Notebook Jupyter*.

En utilisant les fonctions de lecture de *Pandas* vous obtenez donc directement vos données sous un format *DataFrame*. Construire « à la main » un tableau est possible mais assez peu pratique. Nous le verrons plus loin dans le chapitre. Notre conseil : il est souvent plus facile de mettre les données en forme dans un fichier pour ensuite l'ouvrir avec les méthodes `read` de *Pandas*.

Un tableau *Pandas* est un objet de type « *DataFrame* », chaque colonne est un objet de type « *Series* » et chaque ligne est une rangée (ou *row*). Le schéma suivant résume la structure de cet objet.

Structure d'un tableau *Pandas*

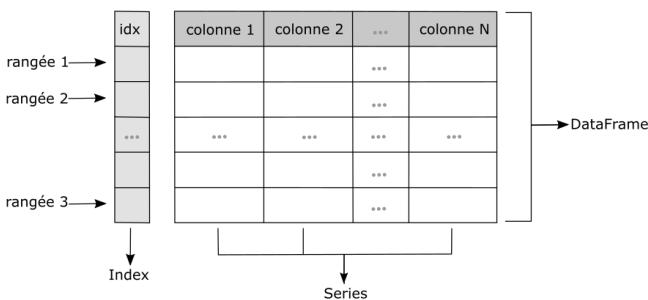


Fig. 5.1 – Structure d'un objet *DataFrame*

La première étape est de regarder à quoi ressemblent les données que nous avons chargées dans la partie précédente.

```
tableau.head()
```

```
CODGEO REG ... C15_POP55P_CS7 C15_POP55P_CS8
0 01001 84 ... 140.0 10.0
1 01002 84 ... 64.5 0.0
2 01004 84 ... 2864.2 244.6
3 01005 84 ... 310.0 30.0
4 01006 84 ... 24.8 0.0
```

[5 rows x 108 columns]

Nous utilisons la méthode `head` pour afficher uniquement le début du tableau<sup>7</sup>. Sous Jupyter, le tableau est affiché avec une mise en forme facilitant la lecture<sup>8</sup>.

Ce tableau est constitué de colonnes et de lignes. Comme pour les listes, la numérotation des lignes et des colonnes commence à 0. La forme générale du tableau peut être obtenue par le paramètre `shape` qui donne le nombre de lignes et le nombre de colonnes

```
tableau.shape
```

(35399, 108)

Le premier nombre représente le nombre de lignes, le second le nombre de colonnes<sup>9</sup>. En plus du contenu du tableau, il y a deux autres informations qui permettent de caractériser le tableau : le nom des lignes et le nom des colonnes.

Comme pour les listes, chaque ligne est repérée par un *index*. Par défaut, cet index est la numérotation, mais peut être aussi une autre information (un mot, ou une date). Vous pouvez regarder l'index d'un tableau avec la propriété `index` de la manière suivante :

```
list(tableau.index)[0:10]
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Nous utilisons la fonction `list` pour transformer l'index en liste car *Pandas* conserve l'information de l'index sous un format spécifique plus pratique pour lui mais moins facilement lisible pour un être humain. Et nous précisons que nous voulons uniquement voir les éléments du rang 0 au rang 10 (non inclus), donc les 10 premiers éléments, avec la sélection `[0:10]`. Si vous ne mettez pas cette sélection, vous allez afficher l'ensemble de l'index.

---

7. Une autre manière serait de sélectionner avec des crochets : par exemple `tableau[0:5]` pour sélectionner les 5 premiers éléments.

8. Ce qui se passe, c'est que l'objet *DataFrame* a une représentation spécifique qui s'affiche joliment dans un *Notebook* quand on exécute une ligne contenant un tel tableau.

9. Nous pouvons avoir plus de dimensions, pour des données plus complexes, mais nous limiterons à deux dans ce livre.

De la même manière, chaque colonne a un nom. Vous pouvez accéder au nom des colonnes avec l'attribut `columns`, ce qui donne pour les 8 premiers éléments du cas précédent :

```
list(tableau.columns)[0:8]
```

```
['CODGEO',
 'REG',
 'DEP',
 'LIBGEO',
 'P15_POP',
 'P15_POP0014',
 'P15_POP1529',
 'P15_POP3044']
```

Vous pouvez accéder à tout élément du tableau en mentionnant l'index et/ou la colonne correspondant via l'attribut `loc`, ce qui, pour accéder au premier élément de la première colonne, s'écrit de la manière suivante :

```
tableau.loc[0,"DEP"]
```

```
'01'
```

Ce code correspond à la sélection d'un élément du tableau *Pandas* `tableau` avec `loc` en indiquant entre crochets l'index (la ligne) et la colonne : dans notre cas, l'index commence à 0, et la première colonne s'appelle « DEP ». Si vous ne mettez pas le nom de la colonne `tableau.loc[0]`, toute la ligne est sélectionnée.

## ➊ Exercice

Affichez la dernière ligne du tableau.

```
total = len(tableau)
tableau.loc[total-1]
```

Pour sélectionner une colonne, il faut utiliser le nom de celle-ci entre crochets. Il n'est pas nécessaire d'utiliser la méthode `loc`, ce qui simplifie l'écriture. Par exemple pour afficher la première colonne (en se limitant uniquement aux premiers éléments) :

```
colonne_a_afficher = tableau.columns[0]
tableau[colonne_a_afficher].head()
```

```

0    01001
1    01002
2    01004
3    01005
4    01006
Name: CODGEO, dtype: object

```

Pour afficher plusieurs colonnes, il suffit de donner la liste des colonnes à afficher entre crochets.

### ?

### Exercice

Affichez la première et la dernière colonne, puis inversez l'ordre des colonnes dans le tableau.

```
colonnes_a_afficher = [tableau.columns[0], tableau.columns[-1]]
tableau[colonnes_a_afficher]
```

Pour inverser, il suffit de sélectionner les colonnes dans l'ordre inverse.

```
nombre_colonnes = len(tableau.columns)
colonnes_a_afficher = [tableau.columns[nombre_colonnes-i-1]
    for i in range(0,nombre_colonnes)]
tableau[colonnes_a_afficher]
```

Attention au -1 : l'indice commence à 0 dans une liste.

Nous savons jusqu'à maintenant charger nos données et les sélectionner. La prochaine étape est de pouvoir les modifier.

## 5.4 Modifier un tableau *Pandas*

Notre tableau *Pandas* est associé à une variable. Il est alors possible de le modifier de plusieurs manières : soit sa forme générale, soit des éléments spécifiques contenus dans le tableau.

### 5.4.1 Sélectionner des colonnes

Le tableau initial est très volumineux. Une première étape peut être de ne sélectionner que les colonnes qui nous intéressent. Il suffit de définir une liste des colonnes et sélectionner les colonnes.

```
colonnes = ['DEP', 'LIBGEO', 'P15_POP', 'P15_POPF', 'P15_POP0014',
    'P15_POP1529', 'P15_POP3044', 'P15_POP4559',
    'P15_POP6074', 'P15_POP7589', 'P15_POP90P',
```

```
'P15_F0014', 'P15_F1529', 'P15_F3044', 'P15_F4559',
'P15_F6074', 'P15_F7589', 'P15_F90P']
tableau = tableau[colonnes]
```

Ce code sélectionne 18 colonnes qui nous intéressent par la suite car elles contiennent des données sur la population.

### 5.4.2 Manipuler l'index

Si vous chargez un tableau, vous avez peut-être envie de changer le nom des colonnes et l'index. Par défaut, quand vous chargez un fichier Excel, l'index est le numéro de ligne du fichier et la première ligne du fichier est utilisée comme nom des colonnes<sup>10</sup>.

Dans notre cas, nous voulons que l'index qui définit chaque ligne soit le numéro de département. Pour cela, il existe une méthode dédiée, `set_index`, qui permet de définir qu'une des colonnes du tableau soit son index. L'intérêt est de pouvoir ensuite sélectionner les lignes en fonction des valeurs de cette colonne. La colonne doit avoir certaines propriétés comme avoir des valeurs distinctes : c'est par exemple le nom des villes si jamais vos données sont des villes ou l'adresse courriel dans le cas d'un sondage en ligne<sup>11</sup>.

```
tableau = tableau.set_index("DEP")
```

La méthode `set_index` est spécifique aux tableaux *Pandas*. Elle ne modifie pas l'objet original mais renvoie la copie modifiée : il faut donc la stocker dans une nouvelle variable. Dans notre cas, plutôt que de définir une nouvelle variable, nous remplaçons l'ancien tableau par le nouveau, ce qui est possible car le code à droite du signe `=` est exécutée (`set_index`) en premier, puis le résultat est assigné à la partie gauche.

#### ?

#### Exercice

##### Quelle est la taille du nouveau tableau ?

Vérifiez que le nouveau tableau a moins de colonnes que le précédent. Pour cela, utilisez l'attribut `.shape`

<sup>10</sup>. Vous pouvez aussi dire de ne pas prendre la première ligne comme noms des variables avec l'option `header = None`.

<sup>11</sup>. Si plusieurs lignes ont le même index, elles seront juste sélectionnées en même temps. C'est l'usage que nous pourrons faire de cet index en sélectionnant les villes par département.

À l'inverse, vous pouvez aussi vouloir que l'index redevienne une colonne du tableau, par exemple vous voulez récupérer la colonne des départements du tableau précédent. Pour cela vous avez la méthode `reset_index`, qui renvoie le tableau modifié.

### ?

#### Exercice

**Retrouvez le tableau initial à partir du tableau précédent.**

```
tableau_initial = tableau.reset_index()
```

### 5.4.3 Renommer les colonnes

Il est possible de donner directement le nom des colonnes à partir d'une nouvelle liste (avec le même nombre d'éléments que le nombre de colonnes). Par exemple, les noms actuels sont peu compréhensibles. En regardant la notice, on identifie par exemple que *P15\_POPF* correspond au nombre de femmes dans la population totale. On peut donc renommer les colonnes :

```
tableau.columns = [
    "Ville", "Total", "Nombre femme", "Nombre 0-14 ans",
    "Nombre 15-29", "Nombre 30-44", "Nombre 45-59",
    "Nombre 60-74", "Nombre 75-89", "Nombre 90 et +",
    "Nombre femme 0-14 ans", "Nombre femme 15-29",
    "Nombre femme 30-44", "Nombre femme 45-59",
    "Nombre femme 60-74", "Nombre femme 75-89",
    "Nombre femme 90 et +"]
```

Ce code modifie l'information du noms des colonnes en donnant une nouvelle liste de noms qui vient modifier le tableau *Pandas* contenu dans `tableau`.

### ?

#### Exercice

**Modifiez uniquement le nom de la première colonne.**

On procède en trois temps : récupérer les anciens noms dans une liste, changer le premier élément de cette liste, puis changer les noms du tableau.

```
nouveaux_noms = list(tableau.columns)
nouveaux_noms[0] = "Total"
tableau.columns = nouveaux_noms
```

### 5.4.4 Créer une nouvelle colonne

Il est fréquent d'avoir des opérations qui se font sur toutes les valeurs d'une colonne (voire du tableau entier). *Pandas* gère souvent ces opérations de manière transparente, évitant d'avoir à dire explicitement de réaliser l'opération sur chaque case.

Pour créer une nouvelle colonne, il suffit de la définir directement en lui donnant son contenu initial. Soit vous donnez une liste contenant le même nombre d'éléments que la taille de la colonne (définie par le tableau initial), soit vous pouvez donner une valeur unique qui sera appliquée sur chaque rangée. Par exemple, nous pouvons créer une colonne vide visant à recevoir ensuite le nom des départements, appelée « noms\_departements » de la manière suivante : `tableau["noms_departements"] = None`.

Ainsi, si vous souhaitez créer une colonne qui stocke les valeurs de l'index, pour l'instant les numéros de départements :

```
tableau["numeros_departements"] = tableau.index
```

#### ?

#### Exercice

Créez une colonne de zéros.

```
tableau["nouvelle_colonne"] = 0
```

### 5.4.5 Modifier une case du tableau

Tout élément du tableau peut être modifié. Pour modifier un élément particulier, il est possible de lui attribuer la valeur en le localisant par `loc` et en lui attribuant la nouvelle valeur. Plutôt qu'un numéro, nous voulons mettre le nom du département. Par exemple, « 01 » correspond à l'Ain.Modifier dans la nouvelle colonne la valeur se fait de la manière suivante :

```
tableau.loc["01", "noms_departements"] = "Ain"
```

Comme nous avons mis les numéros en index, nous pouvons sélectionner la case à partir du nom de la ligne « 01 » et du nom de la colonne « noms\_departements », et lui donner la valeur que nous voulons.

#### ?

#### Exercice

Recodez tous les noms de départements.

Plusieurs stratégies sont possibles. Une est de coder directement dans le document Excel pour avoir la colonne avec les noms. Une autre méthode est d'écrire 100 lignes similaires à la précédente pour chaque département. Enfin, une stratégie plus efficace est présentée dans la section suivante.

#### 5.4.6 Supprimer des lignes ou des colonnes

La méthode `drop` permet de supprimer une ligne (axe 0) ou une colonne (axe 1) d'un tableau existant. Si nous voulons supprimer la colonne vide que nous avions créée :

```
nouveau_tableau = tableau.drop("noms_departements", axis=1)
```

Ce code demande la suppression de la colonne (axe 1 à indiquer) `noms_departements` et stocke le nouveau tableau dans une nouvelle variable `nouveau_tableau`.

#### ?

#### Exercice

**Supprimer la région parisienne du tableau.**

Utilisez la ligne précédente avec le numéro de la région correspondante et `axis=0`.

#### 5.4.7 Répéter une opération sur un tableau

Il est souvent nécessaire d'appliquer un traitement identique aux différentes cases d'une colonne ou à toutes les lignes d'un tableau : recoder, vérifier, filtrer, etc. Par exemple, la phase de nettoyage d'un jeu de données nécessite souvent d'appliquer différentes conditions sur chaque valeur pour vérifier si elle est dans le bon format.

Pour ce faire, il existe une méthode `apply` qui permet d'appliquer une fonction à chacune des lignes (ou des colonnes). Nous procédons alors en deux étapes :

- ➊ définir la fonction qui réalise l'opération sur une case ;
- ➋ utiliser `apply` pour l'appliquer à l'ensemble des cases.

Par exemple, nous avons la population de chaque ville. Nous aimerais séparer les villes en catégories : population faible (moins de 1000 habitants), population moyenne (moins de 50 000), forte population (plus de 50 000). Nous allons procéder suivant ces deux étapes en définissant d'abord une fonction qui renvoie le codage souhaité pour une valeur d'entrée puis en l'appliquant à toutes les rangées de la colonne « Total ».

Pour commencer, nous définissons une fonction qui prend en entrée un nombre et renvoie en sortie la catégorie correspondante. Cette fonction réalise le traitement que nous voudrions appliquer sur chaque case du tableau.

```
def recodage_population(pop):
    if not isinstance(pop, int):
        return None
    if pop < 1000:
        return "Population faible"
    if pop < 50000:
        return "Population moyenne"
    return "Population forte"
```

Ce code :

- ⊕ définit une nouvelle fonction qui prend une variable en entrée *pop* ;
- ⊕ teste le type de *pop* avec la fonction `isinstance`. Si ce n'est pas de type `int`, la fonction s'arrête et retourne `None`, sinon elle continue ;
- ⊕ teste si la valeur de *pop* est moins de 1000 :
  - \* si oui, la fonction renvoie « Population faible » et s'arrête ;
- ⊕ teste si la valeur de *pop* est moins de 50 000 :
  - \* si oui, la fonction renvoie « Population moyenne » et s'arrête ;
- ⊕ retourne « Population forte » si elle ne s'est pas arrêtée avant.

Pour appliquer maintenant cette fonction à toutes les cases de la colonne « Total », qui contient la valeur de la population, nous pouvons utiliser la fonction `apply`. Celle-ci dit à l'ordinateur « prend cette fonction et applique-la à chaque ligne en lui donnant le contenu de la case en entrée » :

```
recodage = tableau["Total"].apply(recodage_population)
tableau["Total_reco"] = recodage
```

Ce code :

- ⊕ prend la colonne *Total* et utilise `apply` pour lui dire d'appliquer la fonction `recodage_population` sur chaque cellule de la colonne ce qui renvoie à une nouvelle colonne stockée dans la variable `recodage` ;
- ⊕ crée une nouvelle colonne dans le tableau *Pandas* `tableau` avec pour nom « *Total\_reco* » et lui donne la valeur de `recodage`.

Cette opération permet d'appliquer une opération sur chaque case d'une colonne. Il est aussi possible d'appliquer une opération sur chaque ligne (ou colonne) d'un tableau entier. La méthode porte le même nom `apply`, avec pour différence qu'elle s'applique sur un tableau *Pandas* et que l'opération doit prendre en entrée une ligne (ou une colonne).

Par exemple, nous avons le nombre de femmes dans chaque ville, mais sans avoir la proportion que cela représente par rapport à la population d'ensemble. Pour calculer cette proportion, nous avons besoin de deux informations pour chaque ville : le nombre de femmes (colonne « Nombre femme ») et le total (colonne « Total »).

## ❶ Information

### Deux fonctions apply

Suivant que vous utilisez `apply` sur une colonne ou sur l'ensemble du tableau, son fonctionnement va être différent. Si le nom et l'idée générale restent les mêmes, le fonctionnement diffère : sur une colonne, `apply` prend comme entrée la valeur des cases tandis que sur un tableau elle prend toute la ligne ou toute la colonne. Concrètement, ce sont deux fonctions différentes mais dont l'utilisation est similaire. Cela peut être un peu perturbant : c'est à vous de penser à quel usage servira la fonction de traitement, et de l'adapter.

À partir de là, il faut diviser le nombre de femmes par la population d'ensemble et multiplier par 100 pour calculer la proportion en pourcentage.

```
def calculer_proportion(ligne):
    if pd.isnull(ligne["Total"]) or ligne["Total"]==0:
        return None
    pourcentage = 100 * ligne["Nombre femme"]/ligne["Total"]
    return pourcentage
```

Ce code :

- ➊ définit une fonction `calculer_proportion` qui prend en entrée une variable `ligne`;
- ➋ teste si le total d'habitant est nul ou égal à 0, et renvoie « `None` » si c'est le cas, sinon continue (pour éviter de diviser par zéro);
- ➌ calcule le pourcentage à partir des informations de la variable `ligne` et le stocke dans la variable `pourcentage`;
- ➍ retourne la valeur de `pourcentage`.

Comme précédemment, la deuxième étape est d'appliquer cette fonction, ici au tableau. Pour préciser que l'opération se fait ligne par ligne, il faut ajouter l'option `axis=1`. Par défaut, la fonction sera appliquée sur chaque colonne, qui correspond à l'option `axis=1`.

```
proportions = tableau.apply(calculer_proportion,axis=1)
tableau["Proportion femmes"] = proportions
```

Ce code applique la fonction `calculer_proportion` ligne-par-ligne au tableau `Pandas tableau` avec la fonction `apply` et renvoie les informations sous la forme d'une série stockée dans la variable `proportions`. Ensuite, nous créons une nouvelle colonne du tableau `tableau` appelée « Proportion femmes » avec comme valeur l'information contenue dans la variable `proportions`<sup>12</sup>.

Nous allons utiliser par la suite un raccourci bien pratique pour fabriquer des fonctions de recodage : les lambda fonctions (*lambda functions*), ou *fonction anonymes*, qui sont des fonctions qui n'ont pas de nom. Au lieu de les définir par un nom avec `def`, il est possible de les définir avec le mot clé `lambda`. Par exemple, `lambda x : 2*x` définit une fonction qui prend une variable `x` en entrée et renvoie le nombre multiplié par 2. Elle a le même effet que la fonction suivante, définie avec un nom :

```
def fonction_x2(x):
    return 2*x
```

Les fonctions anonymes sont particulièrement utiles quand l'opération que nous avons à faire est simple et n'apparaît qu'à un moment dans le programme<sup>13</sup>. Par exemple, si vous avez besoin de compter le nombre de caractères de la case, vous pouvez définir une fonction anonyme qui transforme l'entrée en chaîne de caractères et compte le nombre de caractères avec la fonction `lambda lambda x : len(str(x))`.

Nous voulons maintenant créer une colonne qui indique si la proportion de femmes est supérieure ou inférieure à la moyenne nationale de 51,45 %. Nous allons pour cela utiliser une fonction anonyme :

```
tableau["Proportion femmes sup"] = tableau["Proportion femmes"]\n    .apply(lambda x : True if x > 51.45 else False)
```

Dans `apply` nous définissons une fonction `lambda` qui renvoie `True` si la valeur dans la case est plus de 51.45, sinon `False`. Une fois cette opération faite, le résultat est stocké dans une nouvelle colonne du tableau `tableau` appelée « Proportion femmes sup » (pour des raisons de présentation, nous mettons le code sur deux lignes avec le symbole \).

---

12. Nous le faisons en deux étapes pour des questions de présentation du code, mais vous pouvez aussi l'écrire en une seule ligne.

13. Pour exécuter la fonction `lambda` précédente, il faut écrire `(lambda x : 2*x)(2)`. Il est possible d'utiliser des conditions dans l'écriture d'une fonction anonyme comme dans l'exemple suivant. Si cela vous semble compliqué, vous pouvez aussi définir à chaque fois une fonction normale. Vous n'êtes pas obligé d'emprunter tous les raccourcis.

Avec cette méthode, vous avez tous les éléments pour rapidement recoder des colonnes, même avec les opérations les plus complexes<sup>14</sup>. Le nettoyage des données peut en effet demander de faire plusieurs opérations à la suite (par exemple, pour un texte : supprimer les accents, mettre en minuscule, etc.), qui peuvent être regroupées dans une fonction adaptée.

### ?

### Exercice

Réalisez la même opération de recodage en définissant explicitement la fonction.

```
def sup_moyenne(x):
    if x > 51.45:
        return True
    else:
        return False
```

Et puis `tableau["Proportion femmes"].apply(sup_moyenne)`.

Dans certains cas, la méthode `apply` ne suffira pas et vous aurez besoin de faire une boucle sur les lignes d'un tableau avec `for`. Pour cela il faut parcourir le tableau ligne par ligne. Il existe une méthode des tableaux *Pandas* qui le permet, `iterrows`, et qui renvoie une à une les lignes d'un tableau sous la forme (`index, contenu`). Il suffit alors de faire une boucle qui prend un à un ces éléments. L'écriture des boucles permet de prendre en compte que ce sont deux éléments, et non pas un seul, qui sont parcourus.

Par exemple, pour afficher la population de chaque ville, vous pouvez faire une boucle qui définit deux variables d'itération `i` et `l` pour parcourir le tableau :

```
for i,l in tableau.head().iterrows():
    print("{} a {} habitants".format(l["Ville"],l["Total"]))
```

```
L'Abergement-Clémenciat a 767.0 habitants
L'Abergement-de-Varey a 241.0 habitants
Ambérieu-en-Bugey a 14127.0 habitants
Ambérieux-en-Dombes a 1619.0 habitants
Ambléon a 109.0 habitants
```

À chaque étape, la boucle prend l'index de la ligne dans `i` et la valeur de la ligne dans `l` en se limitant ici au début du tableau.

---

<sup>14</sup> Vous aurez peut-être remarqué que certains exemples ci-dessus auraient pu être réalisés sans l'utilisation de `.apply`. Par exemple, avec *Pandas*, il est possible de faire directement `tableau["Proportion femmes"]>51.45` pour avoir la colonne des `True` et `False`. Cependant, dans un grand nombre de cas `.apply` est nécessaire.

### 5.4.8 Gérer les valeurs manquantes

Un fichier peut contenir des valeurs manquantes. Celles-ci peuvent poser problème dans le calcul et il est important de les prendre en compte. Pour cela, deux méthodes sont utiles : `dropna` et `fillna`, permettant prendre en charge les valeurs nulles qui pourraient exister dans votre tableau pour les remplacer par une valeur spécifique ou pour supprimer la ligne correspondante.

Pour remplir les cases vides avec une valeur spécifique, ici « donnée absente » :

```
nouveau_tableau = tableau.fillna("donnée absente")
```

Et si vous préférez supprimer les lignes dans lesquelles une valeur est absente :

```
nouveau_tableau = tableau.dropna()
```

Une situation qui peut arriver lors d'un recodage est l'envie de remplacer une catégorie par une autre, par exemple s'il y a deux manières différentes de l'écrire dans le tableau. La méthode `replace` prend comme argument un dictionnaire composé de la valeur à remplacer et de la valeur pour la remplacer.

Par exemple, si le nom « Paris » est écrit avec et sans majuscules et avec un « s » qui manque, on peut vouloir tout remplacer de la même manière en définissant un dictionnaire `correction = {"paris": "Paris", "pari": "Paris", "Pari": "Paris"}` et l'appliquer avec la fonction `replace : tableau.replace(correction)`. Une autre manière plus générique est de définir une fonction qui fait cette opération et d'utiliser `apply`.

## 5.5 Filtrer des informations

Une des grandes forces des tableaux *Pandas* est la facilité de sélectionner une partie du tableau en définissant les conditions que doivent remplir les lignes ou les valeurs. Ces conditions correspondent à un filtre. La démarche se fait en deux temps : définir un filtre qui pour chaque case contient « Vrai » ou « Faux », puis sélectionner les éléments du tableau avec ce filtre.

Prenons un cas simple : vous voulez uniquement les villes avec plus de cent mille habitants.

```
filtre = tableau["Total"] > 100000
echantillon = tableau[filtre]
print(len(echantillon))
```

42

Ce code :

- ⊕ crée un filtre à partir de la condition « est-ce que la valeur “Total” est plus grande que 100 000 » stockée dans la variable `filtre`;
- ⊕ prend un sous-ensemble du tableau `tableau` en appliquant le filtre `filtre`;
- ⊕ affiche la dimension de ce tableau.

Il est possible de définir plus finement notre sélection et de combiner des conditions avec les opérateurs ET « & » et OU « | ». Par exemple, pour avoir les villes avec une proportion de femmes supérieure à 50 % et ayant plus de cent mille habitants, le filtre sera `filtre = (tableau["Total"] > 100000) & (tableau["Proportion femmes"] > 50)`.

Les tableaux *Pandas* ont aussi une série de fonctions plus spécifiques qui peuvent être utiles pour créer des filtres et manipuler facilement vos données. Voici une petite liste de ces fonctions spécifiques :

- ⊕ `isin` permet de tester la présence de une ou plusieurs valeurs dans la colonne : par exemple, `colonne.isin([24,45])` renvoie `True` si la valeur de la case est 24 ou 45, et `False` sinon ;
- ⊕ `str.contains` permet de tester la présence d'un morceau de texte dans la colonne : par exemple `colonne.str.contains("coucou")` renvoie `True` si le morceau de texte « coucou » est dans la case, sinon `False` ;
- ⊕ `isnull` renvoie `True` si la case est nulle et `False` sinon ;
- ⊕ `where` renvoie les valeurs uniquement si elles respectent la condition sinon les remplace par `None` ;
- ⊕ `mask` remplace les valeurs qui vérifient la condition par `None`.

En fait, de manière plus générale, vous pouvez filtrer un tableau *Pandas* à partir d'une liste ou d'une colonne de `True` et de `False` qui permet d'indiquer les lignes que vous voulez garder ou enlever. Vous pouvez donc construire une colonne spécifique pour faire des filtres.

### ?

### Exercice

**Utilisez la fonction `str.isin` pour filtrer tous les départements du Grand Est.**

Départements du Grand Est : 08, 10, 51, 52, 54, 55, 57, 67, 68 et 88.

```
grand_est = [08, 10, 51, 52, 54, 55, 57, 67, 68, 88]
filtre = tableau["numeros_departements"].isin(grand_est)
data_grand_est = tableau[filtre]
```

Vous savez maintenant modifier un tableau, recoder des colonnes et filtrer des informations d'un même tableau. Sauf que dans les opérations pratiques, les données sont rarement dans un même fichier. Comment réunir des tableaux différents ?

## ?

### Exercice

**Filtrez uniquement les départements dont le numéro finit par 0.**

Pour commencer, on construit une fonction qui renvoie `True` ou `False` si le dernier numéro est un 0 ou pas. On l'applique à la colonne des numéros de départements, que l'on stocke dans une nouvelle colonne et que l'on peut ensuite utiliser comme filtre.

```
def finit_par_0(x):
    texte = str(x)
    if texte[-1]=="0":
        return True
    else:
        return False
```

## 5.6 Relier des informations de tableaux différents

Souvent, vous voulez utiliser certaines données avec des informations complémentaires récupérées ailleurs. Cela signifie mettre en relation des données différentes, contenues dans des tableaux différents. Cette opération s'appelle un *appariement*.

Deux stratégies sont alors possibles. La première consiste à créer un intermédiaire sous la forme d'un dictionnaire qui permet de relier une information de votre tableau avec une information complémentaire. La seconde consiste à utiliser les fonctions spécifiques des tableaux *Pandas* permettant de faire des opérations similaires au traitement des bases de données<sup>15</sup>.

Dans les deux cas, une étape importante va être de vérifier que les éléments qui permettent de relier les deux ensembles de données sont écrits de la même manière : si d'un côté l'information est un nombre, elle doit aussi être codée comme un nombre de l'autre côté et non pas une chaîne de caractères. Par exemple, si vous avez des informations sur la population d'une ville d'un côté et d'autres informations sur l'industrie, il faut que le nom de la ville soit écrit *exactement* de la même manière des deux côtés. S'il manque un accent, l'appariement va échouer. Pour cela, une étape importante est de connaître ses données et de vérifier leur qualité.

Prenons un exemple. Plus haut, nous avions les numéros des départements, mais pas les noms. Il s'agit d'une information courante, qui peut être trouvée sur internet et vous pouvez télécharger un fichier avec cette information<sup>16</sup>.

15. Ces fonctions ont un air de famille avec celles utilisées dans les bases de données, si vous avez déjà eu l'occasion de vous y frotter. Pour une introduction, voir Cellier & Coaud (2012).

16. Le fichier utilisé est sur le dépôt du manuel.

### 5.6.1 Passer par un dictionnaire

La première stratégie consiste à créer un dictionnaire, et puis écrire une fonction qui va chercher l'information correspondante pour chaque élément dans le dictionnaire. Cette approche est un peu lourde, mais elle marche dans presque tous les cas en ne nécessitant pas de mettre l'ensemble des données dans un même format.

Dans le cas de l'exemple des noms de départements, nous avons le fichier des informations complémentaires. La première colonne donne le numéro du département et la deuxième son nom. Nous allons créer un dictionnaire qui prend le numéro en index et le nom en variable. Ce dictionnaire sert en quelque sorte de traducteur : à telle clé (numéro de département) correspond telle valeur (nom complet) :

```
donnees_comp = pd.read_excel("data/departements-francais.xls")
donnees_comp["NUMÉRO"] = donnees_comp["NUMÉRO"].astype(str)
donnees_comp = donnees_comp.set_index("NUMÉRO")
noms_departements = donnees_comp["NOM"]
dictionnaire_num_nom = dict(noms_departements)
```

Ce code :

- ✿ charge le nouveau fichier `departements-francais.xls` avec *Pandas* en créant un nouveau tableau `donnees_comp` pour « données complémentaires » ;
- ✿ met ce numéro de département en index du tableau ;
- ✿ crée une nouvelle variable qui prend comme valeur la colonne « NOM » ;
- ✿ transforme cette colonne en dictionnaire où les valeurs de l'index deviennent les clés et les valeurs de la colonne les valeurs du dictionnaire.

Ce code fait plusieurs étapes : n'hésitez pas à regarder les résultats intermédiaires de chaque étape.

Maintenant que nous avons ce dictionnaire, nous pouvons soit créer une fonction qui va prendre comme entrée une chaîne de caractères et le dictionnaire et renvoyer « rien » si l'entrée n'est pas dans le dictionnaire, ou sinon l'information associée dans le dictionnaire, ou utiliser la fonction `replace` de *Pandas*.

Prenons la première option, qui est plus compliquée, mais qui permet de mieux maîtriser l'opération. Définissons une fonction qui utilise le dictionnaire pour recoder une entrée.

```
def associer(x, dic):
    if x in dic:
        return dic[x]
    return None
```

Ce code :

- ➊ crée une fonction qui prend deux éléments en entrée : `x` et `dic` ;
- ➋ teste si l'élément `x` est dans le dictionnaire :
  - \* si oui, retourne la valeur correspondante `dic[x]` ;
- ➌ retourne rien sinon : `None`.

Appliquons alors cette fonction sur la colonne à recoder des numéros de département pour construire une nouvelle colonne. Ce code utilise la fonction `apply` pour utiliser la fonction `associer`. Cependant, comme il faut plusieurs informations en argument, nous utilisons le mot-clé `args` pour préciser que la fonction `associer` va non seulement avoir comme entrée la valeur de la cellule mais aussi les arguments renseignés. On met alors la nouvelle colonne dans le tableau avec comme nom `noms_departements`. Puis nous visualisons le début des données.

```
tableau["noms_départements"] = tableau["numeros_departements"]\ 
    .apply(associer,args=[dictionnaire_num_nom])
tableau["noms_départements"].head()
```

```
DEP
01    None
01    None
01    None
01    None
01    None
Name: noms_départements, dtype: object
```

Horrreur. Ça n'a pas marché. Enfin si : presque. Quand on regarde plus en détail les données, et pas seulement le début, nous pouvons voir que seules les lignes correspondant à un index qui comprend d'abord un 0 posent problème.

En fait, dans un des fichiers, les numéros de 1 à 9 ont un 0 devant, et pas dans l'autre. Comment faire ? Vous pouvez corriger le fichier à la main ou modifier le dictionnaire.

### ?

### Exercice

#### Créez un nouveau dictionnaire qui corrige le problème des 0.

Attention : ce sont des chaînes de caractères. On crée donc un nouveau dictionnaire identique pour toutes les entrées sauf celles qui vont de 1 à 9, qu'on modifie. Pour cela, nous faisons une boucle `for` sur le dictionnaire pour créer un nouveau dictionnaire en faisant une opération différente suivant que la valeur est un chiffre de 1 à 9, ou autre (il y a aussi des lettres).

```
nouv_dic_num = {}
for i in dictionnaire_num_nom:
```

```

if len(i) == 1:
    nouv_dic_num["0"+i] = dictionnaire_num_nom[i]
else:
    nouv_dic_num[i] = dictionnaire_num_nom[i]

```

## 5.6.2 Joindre des tableaux

L'autre méthode pour réaliser une opération similaire utilise directement les fonctions de la librairie *Pandas*, et en particulier la fonction `join` qui permet de joindre des colonnes à un tableau, généralement à partir de l'index. Joindre un tableau signifie faire correspondre les lignes d'un deuxième tableau à celles d'un premier en suivant l'ordre d'une colonne particulière qui joue le rôle de lien.

Comme précédemment, nous avons nos données `tableau` avec des informations supplémentaires. Après avoir chargé et corrigé les données pour l'absence de 0 pour les départements comme précédemment, nous allons joindre les deux tableaux ensemble.

```

donnees_comp = pd.read_excel("data/departements-francais.xls")
donnees_comp["NUMÉRO"] = donnees_comp["NUMÉRO"].astype(str)
donnees_comp["NUMÉRO"] = donnees_comp["NUMÉRO"]\n
    .apply(lambda x : "0"+x if len(x)==1 else x)
donnees_comp = donnees_comp.set_index("NUMÉRO")
donnees_comp = donnees_comp[["NOM", "CHEF LIEU"]]

nouveau_tableau = tableau.join(donnees_comp,
                               on="numeros_departements")
nouveau_tableau.head()

```

	Ville	Total	...	NOM	CHEF LIEU
DEP			...		
01	L'Aberge...	767.0	...	Ain	Bourg-en...
01	L'Aberge...	241.0	...	Ain	Bourg-en...
01	Ambérieu...	14127.0	...	Ain	Bourg-en...
01	Ambérieu...	1619.0	...	Ain	Bourg-en...
01	Ambléon	109.0	...	Ain	Bourg-en...

[5 rows x 25 columns]

Ce code :

- ♣ charge les données supplémentaires dans le tableau *Pandas* `donnees_comp` ;
- ♣ transforme les numéros de département en texte ;
- ♣ rajoute un 0 quand le numéro est composé d'un seul nombre (une version compacte du code précédent) ;

- ⊕ met le numéro de département en index ;
- ⊕ sélectionne uniquement les colonnes qui nous intéressent à ajouter ;
- ⊕ relie les deux tableaux `tableau` et `donnees_comp` pour créer un nouveau tableau `Pandas nouveau_tableau` avec toutes les informations. Cette jointure se fait en précisant à la fonction `join` le tableau à joindre et la colonne du premier tableau qu'il faut faire correspondre à l'index du deuxième ;
- ⊕ affiche le début du tableau.

Il existe plusieurs subtilités pour la fonction `join` de *Pandas*, que vous pouvez regarder dans la documentation. En effet, il est possible de dire quel index garder entre les deux tableaux (celui de droite ou celui de gauche), quel tableau a la « priorité » pour imposer ses lignes, etc. Une bonne solution est de ne pas hésiter à tester plusieurs éléments. Surtout, ne renoncez pas à la première tentative : si deux tableaux partagent une même colonne, il est toujours possible de les relier.

## 5.7 Créer un tableau *Pandas*

Un point que nous n'avons pas encore traité est la création d'un tableau *Pandas*. Si une telle opération est possible, elle n'est pas toujours facile ni adaptée. Cette situation est assez rare. Nous avons rarement l'occasion de rentrer nos données à la main dans un programme.

En effet, nous ne saurions que trop vous conseiller de séparer les données du script utilisé pour les analyser. Une manière de ne pas avoir à créer le tableau est d'écrire ses données dans un fichier Excel (ou texte) et de le charger avec la bibliothèque *Pandas*. Cela permet de séparer les sources d'erreurs et de ne pas avoir tout à changer s'il y a un problème.

Si vous souhaitez cependant vraiment créer vos tableaux, vous pouvez utiliser les fonctions `DataFrame` et `Series` qui permettent de transformer respectivement des listes et des listes de listes en `Series` et `DataFrame`. Cela peut être utile dans certaines opérations de recodage. Prenons un exemple simple. Dans le cadre d'un cours, vous avez fait passer une interrogation à votre audience. Vous pouvez alors créer une nouvelle colonne :

```
notes = pd.Series([12,14,11,8,19,13,3])
notes
```

0	12
1	14
2	11
3	8
4	19
5	13
6	3

```
dtype: int64
```

### ?

### Exercice

Vérifiez le type de la variable `notes` et le nombre d'éléments qu'elle contient.

Utiliser les fonctions `type` et `len`.

Si maintenant, vous avez fait un deuxième test, et que vous avez une nouvelle série de notes, vous pouvez créer un tableau *Pandas* :

```
notes = pd.DataFrame([
    [12,14],[14,15],[11,18],[8,16],[19,11],[13,14],[3,7]],
    columns = ["notes 1","notes 2"])
notes
```

	notes 1	notes 2
0	12	14
1	14	15
2	11	18
3	8	16
4	19	11
5	13	14
6	3	7

Ce code définit un tableau *Pandas* avec `DataFrame` à partir d'une liste des notes de chaque individu, et précisant le nom à donner aux colonnes. Vous savez maintenant aussi créer directement vos tableaux. Vous pouvez alors rajouter d'autres colonnes si besoin. Cela peut être utile dans certaines conditions, par exemple pour faire certaines manipulations. Mais avouez que ce serait plus simple de les rentrer dans un fichier texte, de le sauvegarder en format `.csv`, et de le charger avec *Pandas* pour ensuite les traiter.

## 5.8 Sauvegarder ses données

Une fois que vous avez transformé vos données et nettoyé le tableau comme vous le souhaitiez, pensez à le sauvegarder dans un nouveau fichier afin de pouvoir le rouvrir quand vous en aurez besoin. Souvent, vous voulez conserver l'ensemble du tableau, et dans ce cas-là utilisez les fonctions de *Pandas* pour exporter les données.

Ces méthodes de *Pandas* permettent de sauvegarder sous différents formats : pour les plus fréquentes, `to_excel` pour sauvegarder sous un format `.xls` et `to_csv` pour un format `.csv`. L'utilisation se fait alors de la manière suivante, par exemple pour le tableau de notes de la section précédente à sauvegarder dans le dossier « Data » :

```
notes.to_excel("./data/fichier_notes.xls")
```

Dans d'autres cas, vous avez envie de sauvegarder vos données sous un autre format, généralement pour les ouvrir avec un autre logiciel ou pour les utiliser dans d'autres contextes. Vous pouvez aussi utiliser les méthodes d'écriture de bas niveau.

Rappelez-vous : le code se plie à vos besoins, mais pour cela il est important de savoir ce que l'on souhaite faire.

### ?

### Exercice

**Créez un fichier par ville avec dans chaque fichier l'information de la ville.**

```
for i,ligne in tableau.head().iterrows():
    with open("data/"+i,"w") as f:
        f.write(str(dict(ligne)))
```

Ce code boucle sur le début du tableau, ouvre un bloc `with` pour écrire dans un fichier et écrit dans le fichier la ligne transformée d'abord en dictionnaire puis en texte.

Toutes les étapes de manipulation des données que nous avons vues depuis le début sont importantes : ce sont elles qui vous permettent de passer d'informations diverses à des données utilisables puis à des interprétations.

L'utilisation d'un Notebook Jupyter permet de conserver les étapes de cette transformation pour pouvoir les exécuter à nouveau. En effet, vous pouvez être amené à refaire ces étapes si les informations changent ou si vous voulez modifier certaines étapes. Dans ce cas, vous pouvez construire un script dédié du type `nettoyage.py` que vous pouvez relancer à chaque fois. Ce code devient une des étapes de votre processus de travail. Et si vous travaillez en collaboration, garder la trace des étapes de traitement des données permettra à vos collègues de pouvoir refaire vos traitements.

## 5.9 Synthèse du chapitre

Ce chapitre a présenté l'utilisation de la bibliothèque *Pandas* pour traiter des données structurées sous la forme d'un tableau. Cette bibliothèque simplifie énormément les opérations de chargement, de recodage et de mise en forme de ces données. Nous verrons aussi par la suite qu'elle comprend de nombreux outils d'analyse.

Cependant, cette utilisation se fait en mobilisant aussi des opérations plus fondamentales vues dans les chapitres précédents : construire des fonctions, manipuler des chaînes de caractères, etc. Vous êtes amené en permanence à circuler entre le niveau plus fondamental des opérations sur le contenu (nombres, textes, listes) et les fonctions plus génériques des tableaux, vous permettant d'obtenir un maximum de flexibilité.



# Chapitre 6

## Statistiques descriptives et inférentielles

Nous présentons dans ce chapitre comment réaliser les traitements statistiques fréquemment rencontrés en SHS<sup>1</sup> : moyennes, écarts-types, tableaux croisés et corrélation, ainsi que les tests statistiques classiques.

### 6.1 Passer de la programmation aux statistiques

Python est un langage de programmation qui permet aussi de réaliser les traitements statistiques usuels des SHS. Il permet alors de relier l'approche générale de traitement de données aux opérations spécifiques de statistiques.

Ce terme de *statistiques* s'utilise souvent pour décrire une diversité d'opérations pratiques allant du calcul de moyenne à des modélisations avancées. Pour la majorité des situations, faire des statistiques revient surtout à réaliser un ensemble de calculs sur les données. Ces calculs peuvent ensuite être faits à la main, en utilisant une calculette, un logiciel dédié, un langage statistique comme R<sup>2</sup> ou encore Python.

---

1. Référence d'un manuel de statistiques pour les SHS : Chanvril-Ligneel & Le Hay (2014).

2. R est encore à l'heure actuelle plus riche que Python en termes d'outils statistiques, dans la mesure où il est plus ancien et spécifiquement dédié à cette tâche. Suivant vos besoins, il peut être intéressant d'utiliser les deux langages, Python pour la mise en forme des données et R pour l'analyse statistique. Il existe d'ailleurs des manuels d'introduction aux statistiques qui relient ces deux langages Bruce *et al.* (2017).

La force d'utiliser le langage Python plutôt qu'un logiciel dédié est d'intégrer ce traitement dans la continuité des autres opérations de collecte et d'analyse, favoriser son automatisation, gérer les grands volumes de données et contrôler l'ensemble des paramètres. L'interopérabilité de Python facilite l'intégration des calculs avec les autres traitements.

Dans ce chapitre, nous nous concentrerons plus spécifiquement sur les calculs classiques<sup>3</sup>. En fait, ces traitements sont assez spécifiques : les tableaux croisés ou les régressions logistiques, très utilisés en SHS, sont peu utilisés en physique par exemple. La présentation des résultats peut elle aussi être spécifique.

### Information

#### Adapter les outils

Beaucoup de bibliothèques existantes en Python sont génériques. Ce sera à nous d'adapter le résultat final pour le mettre sous la forme habituelle des SHS, qui peut différer même entre disciplines. Cette opération de mise en forme est souvent nécessaire car chaque travail nécessite des adaptations pour la restitution des résultats. La différence avec des logiciels statistiques est qu'il est possible de directement obtenir le résultat sous la forme souhaitée. Cela peut nécessiter un investissement supplémentaire suivant le résultat que vous souhaitez.

Pour simplifier, car dans la réalité les aller-retours sont nombreux, les étapes à suivre pour réaliser un traitement statistique sont :

0. trouver les données ;
1. nettoyer les données ;
2. explorer les données ;
3. identifier les analyses nécessaires ;
4. produire les sorties finales et les visualisations diffusables.

Nous allons surtout traiter des points 1, 2 et 3. Le nettoyage des données a été présenté dans le chapitre précédent, même si nous allons nécessairement rencontrer de nouveau cette étape. La visualisation sera par contre traitée dans le chapitre suivant, même si nous allons déjà rencontrer quelques graphiques pour illustrer la démarche.

Comme un traitement purement formel de ces questions est à la fois austère et peu pédagogique, nous allons par la suite montrer comment réaliser les opérations à partir du jeu de données issu de l'Insee sur les communes françaises en posant une question : *existe-t-il des villes en France dans lesquelles la proportion de femmes est différente de la moyenne nationale ? Et si oui, est-ce que cela est*

---

3. Les statistiques utilisées en SHS diffèrent ainsi largement de celles utilisées dans le domaine de la prédiction, où l'analyse des nombres sert à agir plutôt qu'à décrire des situations.

en lien avec la variable d’âge ou d’occupation professionnelle ? Une telle question pourrait se rencontrer dans différentes disciplines (sociologie, démographie, géographie) et utilise des traitements communs à l’ensemble des spécialités (statistiques descriptives, inférentielles, modèles, cartes). Tout d’abord, remettons en perspective la préparation des données en vue d’un traitement statistique, étape qui occupe généralement une partie importante du travail.

## 6.2 Construire et nettoyer ses tableaux

### 6.2.1 Le tableau de données

La première étape pour faire des statistiques est de constituer un tableau de données. Nous allons utiliser les tableaux *Pandas* présentés dans le chapitre précédent qui permettent de facilement manipuler des fichiers de tableurs. Les tableaux sont généralement organisés de la manière suivante :

- ✚ chaque ligne correspond à une entité ou individu : dans le cas de notre exemple ce sera une ville, identifiée par un numéro unique appelé *index* ;
- ✚ chaque colonne correspond à une variable : dans notre cas, ce sera par exemple le nombre d’habitants (en anglais, on parle de *feature*) ;
- ✚ chaque case contient l’information de l’entité pour la colonne correspondante : cela peut être dans notre cas le nombre d’habitants d’une ville particulière.

Dans le langage habituel des statistiques, chaque case contient une *valeur* de la variable *colonne* pour l'*entité* de la ligne. Cette valeur peut être de différentes natures : numérique, catégorie, date ou texte.

Les sources des données dépendent de la question. Dans certains cas, vous pourrez directement obtenir des données mises en forme dans un tableau. Dans d’autres cas, ce sera à vous de collecter ces données par une enquête et de remplir à la main un tableau avant de pouvoir les analyser. Enfin, dans certaines situations, vous pourrez utiliser la programmation pour mettre en forme des données numériques existantes sous la forme d’un tableau. Les outils vus dans les chapitres précédents permettent d’automatiser une partie de ces tâches. Dans notre cas, les données issues de l’Insee sont déjà mises sous la forme d’un tableau<sup>4</sup>.

---

4. Les données sont disponibles sur le dépôt du manuel. Si vous téléchargez les données directement depuis le site de l’Insee, il est nécessaire de faire une petite opération. Il faut ouvrir le fichier et supprimer les deux premières lignes, qui ne sont pas sous la forme d’un tableau mais informatives sur le fichier. Si vous ne faites pas cette opération, *Pandas* n’arrivera pas à ouvrir le fichier. Une autre option est de préciser à *Pandas* lors de la lecture d’éviter les deux premières lignes avec l’option `skiprows=2`.

## 6.2.2 Vérifier la qualité des données

Avant d'avancer dans le traitement statistique, il est nécessaire de se poser quelques questions : est-ce que mon tableau à la bonne taille ? Le bon nombre d'entités ? Le bon nombre de variables ? Quelles sont les valeurs des variables ? Par exemple, si ce sont des âges, est-ce que je trouve bien des valeurs numériques ? Ou s'il s'agit de nom, est-ce que j'ai bien des modalités sous la forme de chaînes de caractères ?

Nous allons travailler sur les données des villes. Chargeons donc le tableau pour regarder sa forme :

```
import pandas as pd
%matplotlib inline

tableau = pd.read_csv("./data/base-pop-2015-communes.csv")
print(tableau.shape)
```

(35399, 109)

Ce code :

- ➊ charge la bibliothèque *Pandas* avec l'alias `pd` ;
- ➋ ajoute une option pour que les visualisations s'affichent directement dans le Notebook Jupyter ;
- ➌ lit le fichier Excel en précisant le chemin pour y accéder et le stocke dans la variable `tableau` ;
- ➍ affiche la forme de ce tableau avec son attribut `shape`.

### ?

#### Exercice

Affichez dans le Notebook le début et la fin du tableau.

Utilisez les méthodes `head` et `tail`.

Une question importante à se poser dans toute analyse est : est-ce qu'il manque des données ? La fonction `dropna` permet de supprimer les lignes dans lesquelles il manque une information (présence de la valeur `None`). Comparons la taille d'ensemble du tableau avec la taille du tableau sans les informations manquantes :

```
print("Toutes les données : ",tableau.shape)
print("Sans les valeurs manquantes : ",tableau.dropna().shape)
```

Toutes les données : (35399, 109)

Sans les valeurs manquantes : (35399, 109)

Ce code affiche d'abord la forme du tableau général puis enlève les lignes avec une valeur nulle et affiche la forme du tableau. Les deux tableaux ont la même taille, ce qui signifie qu'il n'y a pas de données manquantes. Par contre, il peut toujours y avoir des données mal codées.

### 6.2.3 Sélectionner les données utiles

Nous constatons que ce tableau est très grand : il a plus de trente-cinq mille lignes, pour 108 colonnes. Regarder la documentation n'est pas superflu pour se repérer dans ces colonnes. En faisant ça, nous identifions les colonnes dont nous allons vraiment avoir besoin :

- ➊ P15\_POP pour « Population en 2015 » ;
- ➋ P15\_POPF pour « Nombre total de femmes » ;
- ➌ P15\_POP7589 pour « Nombre de personnes entre 75 et 90 ans » ;
- ➍ P15\_POP90P pour « Nombre de personnes de 90 ans ou plus » ;
- ➎ C15\_POP15P\_CS6 pour « Nombre de personnes de 15 ans ou plus Ouvriers ».

Nous allons aussi garder les informations sur le nom de la ville, le département et la région : CODGEO, REG, DEP, LIBGEO. Retenir uniquement les variables pertinentes pour notre étude nous permet de créer un jeu de données plus petit, qui va donc prendre moins de place dans la mémoire de l'ordinateur et plus facile à visualiser.

```
colonnes = [ "CODGEO", "REG", "DEP", "LIBGEO", "P15_POP",
             "P15_POPF", "P15_POP7589", "P15_POP90P", "C15_POP15P_CS6"]
donnees = tableau.[colonnes]
donnees[0:5]
```

	CODGEO	REG	DEP	...	P15_POP7589	P15_POP90P	C15_POP15P_CS6
0	1001	84	1	...	53.00	5.00	125.00
1	1002	84	1	...	16.86	1.98	9.92
2	1004	84	1	...	1059.01	149.63	1800.54
3	1005	84	1	...	99.00	10.00	230.00
4	1006	84	1	...	9.91	0.99	14.86

[5 rows x 9 columns]

Ce code :

- ➊ définit une liste des colonnes à garder dans la variable `colonnes` ;
- ➋ définit un nouveau tableau en sélectionnant toutes les lignes et certaines des colonnes du tableau initial `tableau` stocké dans `donnees` ;
- ➌ affiche les 5 premières lignes de `donnees`.

Nous pouvons ensuite vérifier que ce sont bien des nombres dans les colonnes, à part le nom de la ville « LIBGEO », en prenant une case en particulier.

```
type(donnees.loc[10, "P15_POP"])
```

```
numpy.float64
```

Ce code affiche le type de la case à la 11ième (le rang 10 correspond au 11e élément, car le comptage commence à 0) ligne de la colonne « P15\_POP », qui est un nombre à virgule (*float*).

Nous l'avons déjà souligné, mais il faut le redire : *l'étape de nettoyage des données est cruciale*. Prenez le temps de bien vous saisir de vos données, comme de leur donner des noms plus compréhensibles. Si vous passez trop rapidement sur cette étape, il est fort probable que cela va conduire à des problèmes ultérieurs.

### ➊ Exercice

**Changez le nom des colonnes du tableau pour des noms plus compréhensibles pour vous.**

Utilisez l'attribut `columns` du tableau.

## 6.3 Recoder les variables

Une étape de *recodage* est nécessaire pour transformer les données disponibles en données utiles.

### 6.3.1 Construire des indicateurs

Une démarche consiste à construire des *indicateurs* à partir des données brutes. En effet, les données brutes ne sont souvent pas directement adaptées pour répondre à la question de recherche. Construire des indicateurs nécessite alors de faire des choix, qui peuvent être basés sur l'expérience ou sur les approches théoriques. Ces choix ne sont pas neutres et conditionnent les résultats obtenus : ce n'est pas la même chose de garder l'âge en année ou de classer les individus dans des catégories qui introduisent alors un découpage.

Le nouvel indicateur peut être obtenu par le recodage d'une seule variable : la présence ou l'absence d'un mot dans un texte, une valeur supérieure ou égale à un seuil, une valeur numérique comme l'âge transformée en classes par regroupement. Il peut aussi être plus complexe et nécessiter d'utiliser plusieurs variables, par exemple l'âge et le niveau de vie, pour construire une typologie de profils.

Dans notre cas, nous voulons poser des questions sur la proportion de femmes. Comme cette information n'est pas directement disponible dans le tableau, nous devons construire une nouvelle variable indicatrice. De la même manière, l'infor-

mation qui va nous intéresser est la part de personnes âgées (plus de 75 ans) et la part d'ouvriers<sup>5</sup>, qui ne sont pas directement dans les données. Nous avons vu dans le chapitre sur la manipulation des données comment recoder une variable et faire des opérations sur un tableau *Pandas* :

- ➊ définir une fonction dédiée qui prend comme entrée une ligne (ou une cellule) et permet de la recoder ;
- ➋ appliquer la fonction ligne par ligne à notre tableau de données ;
- ➌ mettre le résultat dans une nouvelle colonne du tableau.

Calculons donc la proportion de femmes dans la population. Pour plus de facilité nous allons utiliser la proportion en pourcentage. Commençons par définir une fonction qui prend comme entrée une ligne du tableau et renvoie la proportion de femmes en sortie :

```
def calcul_prop_f(ligne):
    if (ligne["P15_POP"] == None) or (ligne["P15_POP"] == 0):
        return None
    proportion = 100*ligne["P15_POPF"]/ligne["P15_POP"]
    return round(proportion,1)
```

Ce code :

- ➊ crée une nouvelle fonction `calcul_prop_f` qui prend en entrée une variable `ligne` qui correspond à une ligne du tableau ;
- ➋ dans la fonction, teste si la valeur `P15_POP` est nulle ou non définie :
  - \* si c'est le cas, la fonction renvoie `None` car il n'est pas possible de diviser par un nombre inexistant ou nul ;
- ➌ calcule la proportion de femmes en divisant le nombre de femmes `P15_POPF` par la population totale `P15_POP`, multipliée par 100 puis la stocke dans la variable `proportion` ;
- ➍ renvoie la valeur `proportion` arrondie à une décimale avec la fonction `round`.

Une fois cette fonction définie, nous pouvons l'appliquer sur chacune des lignes du tableau à l'aide de la fonction `apply` des tableaux *Pandas* :

```
donnees["prop_f"] = donnees.apply(calcul_prop_f, axis=1)
donnees.head()
```

	CODGEO	REG	DEP	...	P15_POP90P	C15_POP15P_CS6	prop_f
0	1001	84	1	...	5.00	125.00	48.90
1	1002	84	1	...	1.98	9.92	49.40

5. Nous utilisons la part des ouvriers pour avoir une information sur la distribution socio-économique de la population, et les plus de 75 ans pour avoir une idée de la part de personnes âgées. Ces informations sont évidemment limitées et pourraient être précisées. Dans le cas de ce chapitre, nous sommes dans une démarche exploratoire.

2	1004	84	1	...	149.63	1800.54	51.70
3	1005	84	1	...	10.00	230.00	49.60
4	1006	84	1	...	0.99	14.86	43.60

[5 rows x 10 columns]

Ce code utilise la fonction `apply` du tableau `donnees` pour appliquer la fonction `calcul_prop_f` définie plus haut. L'option `axis=1` précise que la fonction `apply` avance ligne par ligne. Le résultat de ce calcul est stocké dans une nouvelle colonne du tableau appelée `prop_f`. Le tableau a maintenant une nouvelle colonne (qui correspond à une nouvelle variable) qui contient pour chaque ligne (ville) l'information sur la proportion de femmes.

### Exercice

**Calculez de la même manière la part d'ouvriers puis la part de personnes de plus de 75 ans.**

Il suffit de faire une fonction qui s'applique sur la colonne du nombre d'ouvriers et de créer avec elle une nouvelle colonne du tableau :

```
def cal_prop_ouvr(ligne):
    if (ligne["P15_POP"] == None) or (ligne["P15_POP"] == 0):
        return None
    return round(100*ligne["C15_POP15P_CS6"]/ligne["P15_POP"],1)
donnees["prop_ouvriers"] = donnees.apply(cal_prop_ouvr,axis=1)
```

La différence est que le nombre de personnes de plus de 75 ans se calcule en additionnant le nombre entre 75 et 89 et celle de ceux de plus de 90 ans. Il faut donc rajouter une étape.

```
def cal_prop_sup75(ligne):
    if (ligne["P15_POP"] == None) or (ligne["P15_POP"] == 0):
        return None
    plus_75ans = ligne["P15_POP7589"]+ligne["P15_POP90P"]
    return round(100*plus_75ans/ligne["P15_POP"],1)
donnees["prop_sup75"] = donnees.apply(cal_prop_sup75,axis=1)
```

Après avoir effectué ces opérations, vous pouvez vérifier que le tableau a bien maintenant trois colonnes de plus.

Les valeurs contenues dans ces colonnes sont numériques. Pour une première analyse, il est intéressant d'avoir des variables qualitatives (des catégories) qui permettent de s'affranchir de variations qui n'ont pas nécessairement beaucoup de sens et de se concentrer sur les structures des données. Dans notre cas, nous voulons savoir quelles villes ont une proportion au dessus ou en dessous de la moyenne

nationale. Cela signifie classer les villes dans trois catégories : « au-dessus », « dans la moyenne », « en dessous ». Cela correspond à un recodage du quantitatif, des nombres, vers du qualitatif, des catégories.

Tout d'abord, calculons la proportion de femmes en France :

```
total_pop = donnees["P15_POP"].sum()
total_f = donnees["P15_POPF"].sum()
prop_f_nat = 100*total_f/total_pop
print(round(prop_f_nat,2))
```

## 51.6

Ce code :

- ➊ calcule la variable `total_pop` en faisant la somme de la colonne des populations des villes ;
- ➋ calcule la variable `total_f` en faisant la somme de la colonne des populations de femmes des villes ;
- ➌ calcule la proportion de femmes à partir des deux variables précédentes ;
- ➍ arrondit et affiche.

Nous voulons maintenant définir les conditions pour être au dessus ou en dessous de la moyenne. Pour simplifier ici, on va prendre une variation d'un point (1 %, de manière arbitraire pour explorer les données) comme seuil de différent : si la proportion d'une ville est plus grande que la proportion nationale plus un pourcent, la ville est considérée comme au-dessus, si elle est inférieure de moins d'un pourcent, elle est en dessous, et sinon elle est similaire. Effectuons ce recodage :

```
def reco_prop_f(proportion):
    if proportion > prop_f_nat+1:
        return "Supérieur"
    if proportion < prop_f_nat-1:
        return "Inférieur"
    return "Similaire"

donnees["prop_cat"] = donnees["prop_f"].apply(reco_prop_f)
```

Ce code :

- ➊ définit une fonction `reco_prop_f` qui prend comme entrée une valeur :
  - \* compare cette valeur pour voir si elle est plus grande que `prop_f_nat + 1` :
    - si oui, la fonction retourne « Supérieur » et sort de la fonction, sinon continue ;
  - \* compare cette valeur pour voir si elle est inférieure à `prop_f_nat - 1` :

- si oui, la fonction retourne « Inférieur » et sort de la fonction, sinon continue ;
- \* retourne « Similaire » et sort de la fonction ;
- ⊕ applique la fonction à la colonne `prop_f` avec `apply` et stocke le résultat dans une nouvelle colonne `prop_cat`.

Ces opérations mettent en forme les données. Cette étape n'est en fait jamais vraiment finie. Attendez-vous à devoir créer de nouvelles colonnes ou faire d'autres modifications au fil de l'analyse. Il est rare de penser à tout dès le début, et des aller-retours sont généralement nécessaires comme nous l'avons déjà mentionné.

Pour conserver les modifications que nous avons faites si vous arrêtez votre ordinateur, pensez à sauvegarder le tableau transformé dans un fichier. Ainsi, vous aurez le choix soit de réexécuter l'ensemble des lignes de mise en forme, soit juste charger le tableau modifié.

### ?

### Exercice

#### Sauvegarder le tableau modifié.

Nous l'avons vu précédemment, le code pour écrire un tableau *Pandas* est :

```
donnes.to_csv("./data/data.csv")
```

### 6.3.2 Recodage des variables numériques en intervalles

Un autre type de recodage très fréquent est de changer de types de variables. Les deux cas fréquents sont de passer de nombres à des grandes catégories pour avoir une idée générale et inversement, d'avoir des catégories mais de vouloir des nombres pour faire des calculs.

Le passage de catégories à nombres n'est possible que si ces catégories sont données. Vous pouvez alors leur associer un ordre (1,2,3...) en ayant en tête que calculer une moyenne dans ces conditions a finalement peu de sens.

Le passage de nombres à catégories dépend des objectifs. Cela passe par définir des intervalles qui ont un sens pour les données que vous analysez.

Dans notre cas, nous avons à la fois des populations de villes, des proportions (de femmes, de personnes âgées et d'ouvriers) qui sont des données numériques. Nous voulons à chaque fois créer une nouvelle variable avec des catégories. Nous allons utiliser la fonction de *Pandas* `cut` qui permet de recoder une colonne numérique en définissant les bornes des différents intervalles. Faisons-le pour la population des villes en prenant les villes de moins de 1000 habitants, de moins de 10 000, de moins de 100 000 et au-dessus.

```
q = [0,1000,10000,100000,10000000]
donnees["P15_POP_C"] = pd.cut(donnees["P15_POP"], q)
donnees["P15_POP_C"].head()
```

```
0      (0, 1000]
1      (0, 1000]
2    (10000, 100000]
3    (1000, 10000]
4      (0, 1000]
Name: P15_POP_C, dtype: category
```

Ce code :

- ➊ définit les bornes d’intervalles pour le découpage ;
- ➋ recode la colonne « P15\_POP » avec `cut` en précisant dans notre cas les intervalles de séparation avec le résultat stocké dans une nouvelle colonne `P15_POP_C` ;
- ➌ affiche le début de la variable recodée (vous constatez que le type de la colonne est catégoriel avec des catégories définies).

Les noms des catégories sont automatiquement donnés par la fonction, mais vous pouvez aussi les définir en donnant une liste en plus avec l’argument `labels`. Les intervalles sont définis avec une borne inclue (parenthèse) et une borne exclue (crochet).

## 6.4 Analyse univariée des données

Nous avons un tableau de données sur les villes françaises. Comme il est très grand, nous ne pouvons pas juste regarder chacune des lignes pour nous faire une idée de son contenu. Le début de l’analyse correspond généralement à la production de statistiques dites *univariées* : cela consiste à regarder les variables une à une, c’est-à-dire colonne par colonne. Nous réalisons en priorité les analyses avec les méthodes des tableaux *Pandas*.

### 6.4.1 Le tri à plat des variables

Le tri à plat correspond à la distribution d’une variable catégorielle, c’est-à-dire le nombre de fois où chaque modalité apparaît dans nos données. Ce tri à plat peut être exprimé en effectif absolu (nombre de fois où la modalité apparaît) ou en proportion (pourcentage d’une modalité par rapport à l’ensemble). Nous allons utiliser la méthode `value_counts` de *Pandas*<sup>6</sup>.

---

6. Il est possible aussi de faire une fonction dédiée.

Débutons avec notre variable d'intérêt : la proportion de femmes des villes recensées dans la variable `prop_cat`. Combien de villes ont une proportion de femmes supérieure, ou inférieure, à la proportion nationale ?

```
donnees["prop_cat"].value_counts()
```

```
Inférieur    20517
Similaire    10508
Supérieur    4374
Name: prop_cat, dtype: int64
```

Ce code utilise la fonction `value_counts` qui compte le nombre d'occurrences de chaque élément de la colonne et l'affiche dans une nouvelle série *Pandas*. Il est possible d'avoir la proportion de chaque modalité entre 0 et 1 :

```
donnees["prop_cat"].value_counts(normalize=True)
```

```
Inférieur    0.58
Similaire    0.30
Supérieur    0.12
Name: prop_cat, dtype: float64
```

## ❶ Information

### Construire une fonction qui donne directement le tableau avec les valeurs absolues et les pourcentages

Les outils ne sont pas dédiés aux SHS et ne donnent pas le résultat dans la forme habituelle, qui dépend de votre domaine et de vos besoins. Il est possible de définir un affichage dédié. Pour construire le tableau habituel en sociologie avec l'effectif et le pourcentage, nous pouvons définir une fonction de mise en forme :

```
def construire_tableau(colonne):
    effectif = colonne.value_counts()
    pourcentage = round(100*colonne.value_counts(normalize=True),1)
    tableau = pd.DataFrame([effectif,pourcentage]).T
    tableau.columns = ["Effectif","Pourcentage (%)"]
    return tableau
```

Ce code définit une fonction qui prend une colonne en entrée, calcule le tri à plat en effectif et le tri à plat en pourcentage (arrondi à la première décimale), réunit ces deux colonnes dans un même tableau avec T pour transposer (renverser) les lignes et les colonnes. Il renomme et renvoie ce tableau.

Nous avons ici un bon exemple de l'utilisation de la souplesse de la programmation : définir ses propres fonctions pour répondre à ses besoins.

Attention, s'il existe des cases vides dans votre tableau, elles ne vont pas être comptées par la fonction `value_counts`. Pour remplacer les cases vides par une valeur (par exemple « Absent ») il faut faire un petit recodage avec la fonction `fillna` (remplir les cases vides) :

```
donnees["prop_f_ab"] = donnees["prop_f"].fillna("Absent")
```

Ce code remplace les valeurs absentes `None` par le mot « Absent ». Ce faisant, elles seront comptées dans les tris à plat. Par contre, si vous faites ensuite des calculs, cela peut provoquer des erreurs si la colonne contient des nombres car il s'agit d'une chaîne de caractères.

Une manière rapide de regarder les données de votre tableau est de faire des petites visualisations. Nous reviendrons plus en détail sur les différentes manières de construire des graphiques dans le chapitre suivant. Pour le moment, appuyons-nous sur les outils directement disponibles de la bibliothèque *Pandas*<sup>7</sup>. Pour juste jeter un coup d'œil « visuel » sur un tri à plat :

```
comptage = donnees["prop_cat"].value_counts()
ax = comptage.plot(kind="barh", color="lightgray")
```

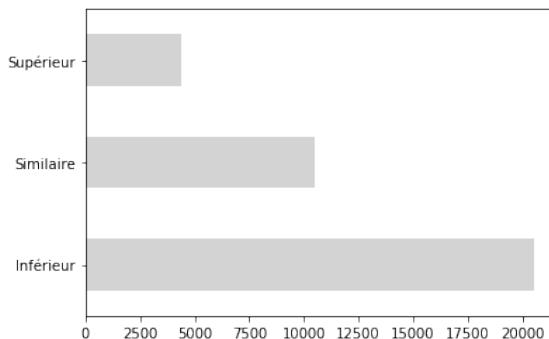


Fig. 6.1 – Exemple d'une représentation graphique exploratoire

Ce code :

- ➊ fait le tri à plat de la colonne `prop_cat` et le stocke dans `comptage`;
- ➋ utilise la fonction `plot` pour faire la visualisation 6.1 en précisant le type de diagramme en barre avec `kind="barh"` (`kind` veut dire type) qui permet d'avoir les barres horizontales.

7. Les fonctions permettant de construire des graphiques sont généralement appelées `plot`.

Les distributions permettent d'avoir une vision d'ensemble qui peut être davantage quantifiée par des indicateurs spécifiques permettant des comparaisons<sup>8</sup>.

Il n'est que très rarement intéressant de faire un tri à plat pour des valeurs quantitatives (si vous en doutez, faites le tri à plat de `prop_f` avant recodage). D'autres méthodes existent pour résumer l'information, présentées plus bas : soit les indicateurs de dispersion, soit le recodage.

## 6.4.2 Les indicateurs de tendance centrale et de dispersion

Ces indicateurs classiques se retrouvent dans tous les manuels de statistiques et sont aisés à calculer avec un tableau *Pandas*.

### Les indicateurs de tendance centrale

Pour caractériser rapidement des valeurs, les indicateurs de tendance centrale (qui correspondent généralement à la *médiane*, *moyenne* et *mode*) permettent d'avoir un résumé en *une* valeur d'un ensemble de données. Comme dans tout résumé, il est pour cela nécessaire de simplifier la complexité des données<sup>9</sup>.

Prenons la variable du nombre d'habitants par ville. La moyenne se calcule avec la méthode `mean` d'un tableau *Pandas*. La population moyenne dans les villes françaises est :

```
donnees["P15_POP"].mean()
```

1869.8347410943811

La médiane se calcule avec la méthode `median` d'un tableau *Pandas*. La médiane de la population est alors :

```
donnees["P15_POP"].median()
```

450.0

Le mode se calcule avec la méthode `mode`. Il n'est pas très intéressant dans le cas de données numériques, mais il peut aussi être calculé :

```
donnees["P15_POP"].mode()
```

---

8. Si rien ne s'affiche dans Jupyter, pensez à rajouter la ligne `%matplotlib inline` au début de votre script qui indique à votre navigateur d'afficher directement les visualisations.

9. Rappelons que la moyenne ne peut être calculée que si la variable est numérique. Le mode peut être calculé pour tout type de variable. Ainsi, vous aurez un message d'erreur si vous voulez calculer une moyenne de catégories.

```
0    107.00
dtype: float64
```

La différence entre ces indicateurs de tendance montre l'importance des choix de traitements.

### ?

### Exercice

**Calculez la médiane et la moyenne des autres colonnes du tableau.**

### Les indicateurs de dispersion

De la même manière que les indicateurs de tendance, nous pouvons calculer les indicateurs de dispersion qui donnent une information sur les variations des valeurs. Pour cela, on utilise généralement l'*écart-type* et les *quantiles*.

L'écart-type caractérise la dispersion des données<sup>10</sup>. Dans le cas d'un tableau *Pandas* il se calcule avec la méthode `std` signifiant *standard-error*. Pour notre exemple de la population des villes contenue dans la colonne « P15\_POP » du tableau :

```
donnees["P15_POP"].std()
```

```
15028.280107001483
```

Les *quantiles* sont une autre manière d'avoir la dispersion. En fait, il s'agit d'une généralisation de la médiane, qui est le quantile d'ordre 2 : la valeur qui divise l'ensemble de nos valeurs en deux groupes de même taille. Les *quartiles* correspondent aux quantiles d'ordre 4 et permettent de calculer 3 valeurs qui divisent les données en quatre groupes d'effectifs égaux. Vous croisez aussi souvent des déciles, correspondant aux quantiles d'ordre 10, et qui permettent de calculer 9 valeurs qui découpent nos données en 10 groupes.

Les quantiles se calculent grâce à la méthode `quantile` qui prend en paramètre une liste des proportions, exprimées entre 0 et 1. La médiane est à 0.5. Calculons alors les quartiles de la population des villes françaises :

```
donnees["P15_POP"].quantile([0,0.25,0.5,0.75,1])
```

---

10. L'écart-type est une manière de calculer la dispersion en référence à une distribution souvent utilisée en statistique, la courbe de Gauss ou loi normale. Cet indicateur est utilisé pour des raisons historiques et conventionnelles, car il simplifie certains traitements. N'hésitez pas à jeter un coup d'œil à un manuel de statistiques si vous voulez plus d'information.

```

0.00      0.00
0.25    198.00
0.50    450.00
0.75   1131.00
1.00  2206488.00
Name: P15_POP, dtype: float64

```

Ce code calcule les quartiles avec la fonction `quantile`. Il indique qu'on divise l'ensemble des valeurs (100 %, ou 1), en quatre intervalles [0,0.25[, [0.25,0.5[, [0.5,0.75[, [0.75,1]. Les valeurs rentrées correspondent au minimum, à Q1, Q2, Q3 et au maximum. Ainsi, 25 % des valeurs de la colonne sont inférieures à la valeur associée à 0.25.

À l'inverse, il est possible de fixer les bornes de ces quartiles et de recoder une donnée numérique en fonction de sa position dans la distribution. La fonction `qcut` permet de recoder une variable numérique en fonction de ses quantiles.

Il est donc possible de recoder nos variables quantitatives en quartiles afin de faciliter leur analyse.

```

q = [0,0.25,0.5,0.75,1]
l = ["Q1","Q2","Q3","Q4"]

donnees["P15_POP_C"] = pd.qcut(donnees["P15_POP"],q,l)
donnees["prop_f_C"] = pd.qcut(donnees["prop_f"],q,l)
donnees["prop_sup75_C"] = pd.qcut(donnees["prop_sup75"],q,l)
donnees["prop_ouvriers_C"] = pd.qcut(donnees["prop_ouvriers"],q,l)

donnees.to_csv("./data/data-chap6.csv")

```

Ce code :

- ➊ définit les bornes souhaitées des quartiles et leurs noms ;
- ➋ fait le découpage en quartiles pour chacune des variables et crée une nouvelle colonne avec les modalités recodées ;
- ➌ sauvegarde de nouveau nos données.

### ?

### Exercice

Faites une boucle pour éviter d'avoir 4 lignes similaires dans le code précédent.

Il suffit de définir la liste des variables concernées :

```
variables = ["P15_POP_C", "prop_f_C", "prop_sup75_C",
```

```

        "prop_ouvriers_C"]
q = [0,0.25,0.5,0.75,1]
l = ["Q1","Q2","Q3","Q4"]
for i in variables:
    donnees[i+"_C"] = pd.qcut(donnees[i],q,l)

```

La *distance interquartile* est l'écart entre le premier et le troisième quartile. Il est facile à obtenir à partir de l'information sur les quartiles :

```

Q = donnees["P15_POP"].quantile([0.25,0.75])
print(Q[0.75]-Q[0.25])

```

933.0

Ce code calcule Q1 et Q3 puis fait la différence entre les deux pour obtenir la distance interquartile.

### ?

### Exercice

**Définissez une fonction qui calcule la distance interdécile Q2-Q9.**

Il suffit de définir q = [0.1,0.9] et renvoyer la différence des deux valeurs.

Enfin, la fonction `describe` permet de faire un résumé de plusieurs colonnes numériques.

```
donnees["P15_POP"].describe()
```

count	35399.00
mean	1869.83
std	15028.28
min	0.00
25%	198.00
50%	450.00
75%	1131.00
max	2206488.00
Name:	P15_POP, dtype: float64

### ?

### Exercice

**Décrivez les différentes colonnes de votre tableau.**

Appliquez `describe` au tableau en entier.

Nous avons pour le moment uniquement utilisé les méthodes de *Pandas* pour des raisons pratiques. Pour autant, d'autres fonctions permettent de réaliser les mêmes opérations sans passer par une bibliothèque dédiée. Python 3 a un module statistique chargé automatiquement qui a les fonctions `mean()`, `median()`, `mode()` et `std()` pouvant être appliquées sur des listes. Suivant les cas, vous utiliserez donc les méthodes *Pandas* si vous êtes dans un tableau *Pandas* et les fonctions de base si vous travaillez directement sur des listes.

### ?

### Exercice

**Écrivez une fonction qui calcule l'écart moyen (moyenne des écarts à la moyenne).**

Il faut utiliser la fonction `abs` qui donne la valeur absolue car une distance est toujours positive.

```
def ecart_moyen(serie):
    moyenne = serie.mean()
    distance_moyenne = serie.apply(lambda x : abs(x-moyenne))
    return distance_moyenne.mean()
```

Ce code :

- ⊕ définit une fonction qui prend une série en entrée;
- ⊕ calcule la moyenne de la série;
- ⊕ calcule une nouvelle série qui est obtenue en faisant la distance à la moyenne de chaque élément de la série `serie`;
- ⊕ retourne la moyenne de cette série de distance à la moyenne.

## Identifier les valeurs extrêmes

Enfin, regardons comment faire pour identifier des valeurs extrêmes dans notre jeu de données. Une première manière de les identifier est d'utiliser les méthodes `max` et `min` des tableaux *Pandas*.

Une autre manière est de calculer les *outliers*. Ce sont les valeurs qui s'écartent beaucoup de la tendance centrale (mode ou moyenne). Que signifie beaucoup ? Cela dépend de ce que vous cherchez. Cela peut être deux fois l'écart-type. Ou trois. Ou cinq.

### ?

### Exercice

**Identifiez la ville qui a la population la plus élevée.**

Utilisez `donnees["P15_POP"].max()` pour identifier la valeur puis un filtre.

Quelle que soit la règle que vous prenez pour définir un *outlier*, il est alors facile de faire une fonction qui renvoie vrai ou faux si cette règle est respectée. Par exemple, pour identifier les villes qui sont à plus de `nb_std` fois l'écart-type :

```
# Définition de la fonction
def identifie_outliers(valeur, nb_std, m, std):
    distance = abs(valeur-m)/std
    if distance > nb_std :
        return True
    return False

# Calcul des indicateurs
moyenne = donnees["P15_POP"].mean()
ecarttype = donnees["P15_POP"].std()

# Identification des outliers
outliers = donnees["P15_POP"].apply(identifie_outliers,
                                      args = (10,moyenne,ecarttype))
outliers.value_counts()
```

```
False    35382
True      17
Name: P15_POP, dtype: int64
```

Ce code :

- ➊ définit une fonction qui prend pour argument une valeur, la distance de détection en nombre d'écart-types `nb_std`, la moyenne `m` et l'écart-type `std` :
  - \* calcule la distance entre la valeur et la moyenne en nombre d'écart-types ;
  - \* si cette distance est plus grande que le seuil, renvoie `True` ;
  - \* sinon renvoie `False` ;
- ➋ applique la fonction `identifie_outliers` sur la série `donnees["P15_POP"]` avec `apply` en précisant les arguments supplémentaires ;
- ➌ affiche le nombre d'outliers avec `value_counts`.

Il y a 17 villes qui sont des *outliers* par rapport à la définition que nous avons prise. Nous avons étudié les variables une à une : comment les étudier dans leurs relations ?

## 6.5 Relation dans les données

Expliquer conduit nécessairement à relier des éléments ensemble et donc à s'intéresser aux relations. Après l'étape de l'analyse individuelle de chaque variable, voici venu le moment de *croiser* les variables, c'est-à-dire de regarder comment elles sont liées. L'analyse *bivariée* procède de différentes manières (tableaux croisés ou corrélations) suivant la nature des variables (quantitatives ou catégorielles).

### 6.5.1 Construire des tableaux croisés

D'un point de vue pratique, un tableau croisé revient à prendre deux variables avec leurs modalités et compter chacune des combinaisons qui existent quand on prend une modalité de la première variable et une modalité de la seconde. Le *tableau croisé* est alors un tableau avec en ligne les modalités d'une variable et en colonne les modalités de l'autre. Une fois ce tableau construit, il est alors possible de calculer des pourcentages par ligne, par colonne, ou sur le tableau entier pour avoir la présentation habituelle aux SHS. Cela permet de regarder les différences de distribution d'une propriété suivant les catégories.

Une des hypothèses que nous pouvons faire pour expliquer la variation de la proportion des femmes dans les villes françaises est qu'elle est liée à la proportion des personnes plus âgées, car l'espérance de vie n'est pas la même pour les femmes et pour les hommes. Nous avons donc envie de croiser la variable `prop_f` et la variable `prop_sup75_C`. Pour cela, on va utiliser la fonction de *Pandas crosstab* qui prend en argument les variables à croiser.

```
pd.crosstab(donnees["prop_f_C"], donnees["prop_sup75_C"])
```

prop_sup75_C	Q1	Q2	Q3	Q4
prop_f_C				
Q1	2649	2064	2146	2055
Q2	2943	2423	2070	1542
Q3	2359	2515	2390	1655
Q4	1207	1567	2305	3503

Ce code utilise la fonction `crosstab` pour croiser les deux colonnes correspondant aux deux variables qui nous intéressent. Elle renvoie un tableau *Pandas* qui a en lignes les quartiles de `prop_f_C` et en colonnes les quartiles de `prop_sup75_C`. L'ensemble des villes est alors réparti dans ce tableau (si on fait la somme de toutes les valeurs on retrouve la population totale). Rajoutez l'option `margins=True` pour avoir les totaux par ligne et par colonne.

Pour interpréter un tableau croisé, il est souvent préférable d'utiliser des pourcentages (par ligne ou par colonne) pour être plus à même de distinguer des variations dues à la proportion de personnes âgées. Il est d'usage de faire les pourcentages sur les modalités de la variable dite « indépendante » ou « explicative ». Dans notre cas, la variable explicative est l'âge, qui est en colonne. Nous calculons donc le tableau avec les pourcentages en colonnes en utilisant l'option `normalize`.

```
tab = pd.crosstab(donnees["prop_f_C"],
                   donnees["prop_sup75_C"],
                   normalize='columns',
                   margins=True)
round(100*tab,2)
```

	prop_sup75_C	Q1	Q2	Q3	Q4	All
prop_f_C						
Q1	28.93	24.09	24.08	23.47	25.19	
Q2	32.14	28.28	23.23	17.61	25.37	
Q3	25.76	29.35	26.82	18.90	25.20	
Q4	13.18	18.29	25.87	40.01	24.25	

Ce code utilise deux paramètres de la fonction `crosstab` pour rajouter les *marges* du tableau avec `margins` (c'est-à-dire la colonne et la ligne qui font les totaux), et le paramètre `normalize='columns'` qui permet de calculer les pourcentages par colonnes (pour les lignes, il faut mettre `normalize='index'`). Il affiche ensuite les valeurs arrondies de ce tableau.

## ?

### Exercice

Existe-t-il un lien entre la taille des villes et la proportion de femmes ?

Pour y répondre, construisez le tableau croisé entre `prop_f_C` et `P15_POP_C`.

## !

### Information

**Différentes manières de calculer les pourcentages.**

Il y a plusieurs manières de calculer les pourcentages par ligne ou par colonne. Une manière de faire est d'utiliser la fonction `apply` avec une fonction anonyme qui permet de mieux contrôler le format de l'affichage (arrondir ou pas, en pourcentage ou en fréquence, etc.) :

```
tab = pd.crosstab(donnees["prop_f_C"],
```

```
donnees["prop_sup75_C"])
tab.apply(lambda x : 100*x/sum(x), axis=1)
```

Ce code commence par créer le tableau croisé et lui applique une fonction anonyme qui divise la ligne x par la somme de la ligne et multiplie le résultat par 100. `axis=1` permet de faire l'opération sur les lignes, et `axis=0` sur les colonnes.

Risquons une rapide interprétation : pour les villes avec la population la plus jeune (première colonne correspondant au premier quartile), la proportion des villes avec la plus forte proportion de femmes est de 13 %, contrairement à 25 % si les données étaient réparties aléatoirement. À l'inverse, pour les villes avec la population la plus âgée, 40 % des villes sont dans le dernier quartile. Il existe donc un lien entre l'âge de la population et la proportion de femmes. Pour autant, cela ne signifie pas qu'il y ait une causalité, d'autres facteurs pouvant intervenir. Il faut continuer l'analyse.

Et voilà, si vous avez besoin de regarder le lien entre des colonnes contenant des catégories, vous savez maintenant faire des tableaux croisés.

## ❶ Information

### Définir une fonction qui met en forme un tableau

Comme pour le tri à plat, il est plus agréable d'avoir une mise en forme adaptée à nos besoins car nous faisons beaucoup de tableaux croisés présentant effectifs et pourcentages.

```
def tableau_croise(c1,c2):
    t_absolu = pd.crosstab(c1,c2,margins=True)
    t_pourcentage = pd.crosstab(c1,c2) \
        .apply(lambda x: 100*x/sum(x),axis=1)
    t = t_absolu.copy()
    for i in range(0,t_pourcentage.shape[0]):
        for j in range(0,t_pourcentage.shape[1]):
            t.iloc[i,j] = str(t_absolu.iloc[i,j]) \
                +" ("+str(round(t_pourcentage.iloc[i,j],1))+ "%)"
```

return t

Ce code définit une fonction qui prend deux colonnes comme entrée. Il commence par calculer les deux tableaux (en valeurs absolues et pourcentages), puis crée un nouveau tableau en copiant celui des valeurs absolues pour le compléter avec les pourcentages. Pour cela, il fait une boucle sur toutes les cases du tableau. Pour chaque case, il modifie le contenu en créant une chaîne de caractères qui contient à la fois l'effectif et le pourcentage mis en forme.

Et puis, dans certains cas, vous avez envie de construire un tableau croisé à plus de deux variables. Par exemple, pour contrôler l'effet d'une variable. La fonction `crosstab` permet de gérer des tableaux à plus de deux dimensions. Voici un exemple de tableau croisant les trois variables de proportion :

```
tab3 = pd.crosstab([donnees["prop_sup75_C"],
                    donnees["prop_ouvriers_C"]],
                   donnees["prop_f_C"],
                   rownames=["prop vieux", "prop ouvriers"],
                   colnames=["prop femmes"])
tab3
```

		Q1	Q2	Q3	Q4
		prop ouvriers			
		prop vieux	prop ouvriers		
Q1	Q1	588	638	583	380
	Q2	507	794	698	292
	Q3	635	813	616	255
	Q4	919	698	462	280
Q2	Q1	402	423	556	458
	Q2	413	564	692	398
	Q3	502	703	703	400
	Q4	747	733	564	311
Q3	Q1	511	384	501	658
	Q2	429	447	619	649
	Q3	505	585	687	600
	Q4	701	654	583	398
Q4	Q1	779	519	514	1210
	Q2	395	330	410	990
	Q3	379	346	424	867
	Q4	502	347	307	436

Ce code utilise la fonction `crosstab` avec la seule différence que l'argument donné est d'un côté une colonne, de l'autre une liste de colonnes. Un tel tableau n'est pas forcément très facile à lire, mais permet dans certains cas de montrer l'effet d'une troisième variable.

## 6.5.2 Les corrélations

Les coefficients de corrélation permettent de caractériser le lien entre deux variables numériques ou ordonnées. Le plus rencontré est celui dit de Bravais-Pearson, qui donne une indication sur la relation linéaire entre deux séries de données numériques. Pour le dire rapidement, cela indique si la variation d'une variable est dépendante de la variation de l'autre : si j'augmente X, est-ce que Y varie dans le même sens (corrélation positive) ou dans l'autre sens (corrélation négative).

Comme souvent déjà, il existe plusieurs stratégies pour calculer le coefficient de corrélation de Bravais-Pearson. Des fonctions existent dans la bibliothèque scientifique *Scipy*. Et enfin, *Pandas* a directement une fonction intégrée pour calculer des corrélations.

À partir de la bibliothèque *Scipy*, nous pouvons utiliser la fonction `pearsonr` du module `stats`, qui prend en entrée deux listes de nombres de *même taille*. Faisons-le pour la corrélation entre la taille des villes et la proportion d'ouvriers.

```
from scipy.stats import pearsonr

variable1 = list(donnees["prop_sup75"].dropna())
variable2 = list(donnees["prop_ouvriers"].dropna())

round(pearsonr(variable1,variable2)[0],2)
```

-0.13

Ce code :

- ✿ charge la fonction `pearsonr` du module `scipy.stats` ;
- ✿ définit les deux listes de nombres à partir des colonnes du tableau, en pensant bien à enlever les valeurs nulles qui vont poser problème sinon (attention, vérifiez que la taille des listes reste la même) ;
- ✿ applique la fonction `pearsonr` sur ces deux listes pour obtenir le coefficient de corrélation linéaire qui est le premier élément.

Il est possible de faire exactement le même traitement, voire plus, à partir des outils de *Pandas*. Pour obtenir les coefficients de corrélation linéaire d'un ensemble de colonnes, il existe la fonction `corr`.

```
donnees[["prop_sup75","prop_ouvriers","prop_f"]].corr()
```

	prop_sup75	prop_ouvriers	prop_f
prop_sup75	1.00	-0.13	0.16
prop_ouvriers	-0.13	1.00	-0.11
prop_f	0.16	-0.11	1.00

Ce code prend une partie du tableau `donnees` en sélectionnant une liste de ses variables (d'où les doubles crochets) et calcule les coefficients de corrélation linéaire de chaque couple de colonnes. Cela donne un tableau des corrélations.

Nous constatons qu'il n'existe pas de corrélation forte entre ces variables numériques, elles sont toutes assez faibles et proches de 0.

### 6.5.3 Grouper des catégories

Une situation différente des deux précédentes, pour autant fréquente, est d'avoir à la fois des données numériques et des données en catégories. La question se pose si les groupes décrits par des catégories différentes vont avoir des valeurs numériques différentes. Par exemple, regrouper les villes par département pour calculer la moyenne des populations de villes par département.

La fonction `groupby` de *Pandas* permet de réorganiser un tableau en utilisant les valeurs d'une colonne pour grouper les informations d'autres colonnes. Ainsi, si une colonne contient des informations sur les groupes à faire (le département), et les autres colonnes des valeurs numériques (la taille de la ville), cela permet de grouper par l'information des groupes pour ensuite calculer des moyennes par groupes.

Par exemple, pour voir s'il y a un effet aléatoire spécifique pour les petites villes, regroupons les villes par catégories de proportion de femmes et regardons quelle est la taille médiane des villes de chaque groupe.

```
donnees.groupby("prop_f_C")["P15_POP"].mean()
```

```
prop_f_C
Q1    382.62
Q2    886.30
Q3   1961.62
Q4   4349.42
Name: P15_POP, dtype: float64
```

Ce code utilise la fonction `groupby` pour regrouper le tableau `donnees` par les catégories de la colonne `prop_f_C`. Il sélectionne ensuite une colonne particulière, `P15_POP` et utilise la fonction `mean` pour avoir la moyenne par catégorie.

Il est possible d'appliquer en même temps plusieurs fonctions avec `agg` :

```
donnees.groupby("prop_f_C")["P15_POP"].agg(["median", "mean"])
```

	median	mean
<code>prop_f_C</code>		
Q1	226.00	382.62
Q2	500.00	886.30
Q3	720.00	1961.62
Q4	673.00	4349.42

Ce code fait la même opération que précédemment mais utilise en plus la fonction `agg` pour en même temps appliquer deux traitements, la moyenne `mean` et la médiane `median`. Nous pourrions en ajouter d'autres.

Vous pouvez directement donner un nom aux colonnes avec un dictionnaire en argument plutôt qu'une liste. Dans le cas précédent, essayez avec `.agg({"Médiane": "median", "Moyenne": "mean"})`

Le résultat permet de constater que les villes les plus éloignées de la moyenne nationale, en particulier du premier quartile Q1, correspondent à des petites villes. Une plus grande proportion masculine est liée à ces petites villes.

Plusieurs hypothèses peuvent être faites : d'une part, les petites villes sont souvent associées à la ruralité, qui offre moins d'activité aux femmes et, d'autre part, un effet aléatoire d'échantillonnage peut produire une plus grande dispersion.

La fonction `groupby` est très importante pour l'utilisation des tableaux *Pandas*. Elle permet de faire des opérations beaucoup plus générales que celles que nous verrons dans ce chapitre, en particulier appliquer des fonctions sur des sous-ensembles de tableaux. Gardez en tête que grâce à cette fonction vous pouvez rassembler des groupes dans votre tableau pour les traiter ensuite.

## 6.6 Réaliser des tests statistiques

Le calcul de statistiques est généralement associé au calcul de leur significativité statistique pour vérifier que les différences obtenues ont bien un sens au-delà des variations aléatoires des données.

L'utilisation des tests statistiques dépend des domaines : dans certains, ils sont systématiquement présentés, tandis que dans d'autres leur usage est plus souple. Dans tous les cas, il est important de savoir comment les calculer.

### 6.6.1 Quelques rappels

Les données que nous traitons généralement<sup>11</sup> sont le résultat de phénomènes en partie aléatoires : cela signifie que les données que l'on peut obtenir et les valeurs que l'on calcule intègrent une partie de hasard.

#### ❶ Information

##### Générer des données aléatoires

Pour tester une fonction ou pour certaines applications spécifiques, il est intéressant de générer des données aléatoires. Pour cela, il existe un module de génération d'éléments aléatoires, le module `random`, qui a plusieurs fonctions dédiées. Quelques fonctions intéressantes :

---

11. Nous disons généralement car dans le cas où vous travaillez non pas sur un échantillon (un sous-ensemble des individus) mais en population complète, il peut arriver que le calcul de tests statistiques n'ait pas beaucoup de sens.

- ➊ `random.randint(min,max)` renvoie un nombre entier aléatoire entre le min et le max ;
- ➋ `random.random()` renvoie un nombre aléatoire entre 0 et 1 ;
- ➌ `random.choice(liste)` renvoie aléatoirement un élément de la liste liste ;
- ➍ `random.normalvariate(moyenne, ecarttype)` renvoie un nombre aléatoire suivant une distribution normale avec la moyenne et l'écart-type.

Par exemple, imaginons que nous voulons générer un tableau de données aléatoires pour faire des tests, avec une variable qui représente des âges aléatoires et une variable qui représente aléatoirement des hommes ou des femmes.

```
import random
nombre = 100
age_max = 100
donnees = pd.DataFrame([
    [random.randint(0,age_max), random.choice(["H","F"])]
    for i in range(0,nombre)])
donnees.columns = ["Age", "Genre"]
```

Selon vos besoins, vous pouvez trouver des fonctions plus spécifiques pour générer des informations aléatoires dans les bibliothèques *NumPy* et *Scipy*. Il est aussi possible de fixer la graine (*seed*) du générateur avec `random.seed` et un nombre, ce qui permet d'avoir du hasard reproductible.

Une fois les coefficients statistiques calculés (une moyenne ou encore un paramètre de régression), une étape souvent nécessaire est d'évaluer la part de ce hasard dans nos analyses. Ce n'est pas la même chose d'avoir une variation de 1 % sur cent individus que sur cent millions : dans le premier cas, il suffit qu'on change une personne pour avoir une telle différence, dans l'autre cas un million de personnes. Pour rendre compte de la part de cet aléatoire, nous sommes amenés à régulièrement calculer des tests statistiques en fonction de l'analyse.

## ❶ Information

### Petit rappel sur les tests statistiques

De manière schématique, un *test statistique* permet de poser la question : est-ce que la situation que j'observe est différente de celle que j'aurais obtenue juste avec le hasard ? Très souvent, la question ressemble à : est-ce que ce que j'observe aurait pu se passer s'il n'y avait pas un comportement différent entre deux groupes, ou entre deux variables ?

Pour aborder cette question, la démarche est de faire l'hypothèse que l'on souhaite interroger, calculer la situation qu'on aurait dû avoir (en fait, comme il y a du hasard, les situations probables que l'on aurait pu attendre) et ensuite regarder la probabilité de la situation qu'on a vraiment.

La robustesse statistique de résultats se fait généralement à partir de la probabilité critique, ou *p-value*, qui décrit la probabilité d'avoir la situation observée par rapport à une hypothèse initiale. Plus cette valeur est petite (et donc plus la probabilité est faible) moins l'hypothèse initiale est probable, donc plus grande est la probabilité que la différence constatée ne soit pas due au hasard.

Quel que soit le logiciel ou le langage utilisé, ces tests d'hypothèses se font à partir de fonctions déjà programmées, basées sur des lois de probabilités. Même s'il est préférable d'avoir une idée des mathématiques derrière un test, nous sommes généralement dans la situation de les utiliser comme un outil déjà existant pour résoudre un problème spécifique : la difficulté est alors d'identifier le test adapté (qui dépend des situations) et dans quelle bibliothèque celui-ci est disponible<sup>12</sup>.

Nous allons ici présenter brièvement les tests disponibles dans la bibliothèque *Scipy* pour des situations fréquentes en SHS. Généralement, les cas rencontrés dans les approches non expérimentales des SHS sont assez classiques : tableau croisé, corrélation ou comparaison de moyennes. D'autres tests sont directement associés aux modèles présentés plus loin. Ce manuel ne visant à présenter une liste exhaustive de tests, et nous n'aborderons pas toutes les variantes, corrections, et particularités des situations expérimentales<sup>13</sup>.

### 6.6.2 Intervalle de confiance

Tout d'abord, faisons un petit détour sur les intervalles de confiance pour préciser les usages des tests. Les intervalles de confiance permettent de bien comprendre le lien entre les données réelles et variables statistiques. Dans notre situation où nous travaillons sur les données en population complète des villes françaises, cela permet d'insister sur le lien entre échantillon et population d'ensemble.

Le cas le plus simple est d'interroger : quelle est la valeur de la proportion de femmes des villes françaises ? Comme nous avons toutes les données, nous pouvons facilement la calculer :

---

12. Dans certains cas particuliers, disposer d'un langage de programmation permet de programmer directement sa fonction de test quand elle n'est pas disponible. Beaucoup d'informations existent sur les forums en ligne si vous devez vous retrouver dans ce cas.

13. Si vous devez utiliser un test d'hypothèse qui n'est pas couvert dans ce chapitre, vous pouvez identifier dans un manuel de statistiques la configuration des données que vous devez traiter (par exemple, Saporta (2006)), puis consulter la documentation de la bibliothèque *Scipy* pour voir tous les tests disponibles. Dans certains cas (assez rares) le test spécifique peut ne pas exister et vous amener à écrire vous-même la fonction nécessaire.

```
m = donnees["prop_f"].mean()
et = donnees["prop_f"].std()
print("Moyenne : {:.2f} [{:.2f}]".format(m, et))
```

Moyenne : 5e+01 [2.7]

Dans ce cas, cela n'a pas de sens de poser la question de l'intervalle de confiance : cette moyenne et cet écart sont descriptifs sur l'ensemble des villes et ne sont donc pas liés à un phénomène aléatoire. Il ne pourrait pas être autrement.

Par contre, nous pouvons nous poser la question : est-ce que la répartition de cette proportion suit une distribution normale. Pour cela, il y a un test dédié dans la bibliothèque *Scipy* qui est **normaltest** :

```
from scipy.stats import normaltest
test = normaltest(donnees["prop_f"].dropna())
print("La p-value du test est {:.e}".format(test[1]))
```

La p-value du test est 0.000000e+00

Ce code :

- ✿ charge la fonction de test **normaltest** ;
- ✿ applique cette fonction à la distribution des valeurs de proportion de femmes ;
- ✿ affiche la probabilité critique du test avec **print** en formatant le texte pour que la valeur soit exprimée dans la notation scientifique.

La valeur est très faible (proche de 0). À ce titre, la répartition s'éloigne de l'hypothèse initiale de la loi normale. Les données que nous testons ont très peu de probabilité d'apparaître si le modèle était celui d'une distribution normale.

La situation serait potentiellement différente si nous n'avions qu'une partie des données des villes. Par exemple, on peut utiliser la méthode **sample** qui permet de sélectionner aléatoirement un échantillon d'un tableau *Pandas*, en indiquant la proportion par rapport à la taille totale. À ce moment-là, si nous calculons la moyenne sur cet échantillon, ce n'est plus la moyenne nationale. Nous pouvons par contre nous dire que cette valeur correspond à une estimation de cette moyenne nationale avec une erreur aléatoire liée à l'échantillon.

```
echantillon = donnees["prop_f"].sample(frac=0.01)
m = echantillon.mean()
et = echantillon.std()
print("Moyenne : {:.2f} [{:.e}]".format(m, et))
```

Moyenne : 5e+01 [2.930897e+00]

Ce code :

- ⊕ extrait un échantillon aléatoire représentant 1 % des données de la colonne avec la méthode `sample` (comme l'échantillon est aléatoire, vous n'allez pas obtenir les mêmes résultats que dans le livre car votre échantillon sera différent) ;
- ⊕ calcule la moyenne puis l'écart-type ;
- ⊕ affiche ces valeurs.

Dès lors, la question se pose : comme cette valeur peut varier suivant l'échantillon que l'on prend, dans quel intervalle va-t-elle se situer ? Dans la mesure où nous faisons l'hypothèse que la variable suit une distribution normale (ce qui n'est pas toujours le cas, par exemple ici, mais souvent la situation se rapproche d'une telle distribution), nous pouvons calculer un intervalle de confiance de la moyenne de la proportion de femmes. Nous utilisons alors l'objet `norm` qui décrit le comportement d'une loi normale avec une fonction `interval` qui donne l'intervalle correspondant.

## ❶ Information

### L'erreur standard

Dans ce cas, il faut utiliser l'erreur standard et non pas l'écart-type. Ces deux indicateurs ont des similarités, mais sont légèrement différents dans leur calcul. Il est possible d'obtenir l'erreur standard sur un échantillon avec la fonction `sem`.

```
from scipy.stats import sem, norm
m = echantillon.mean()
es = sem(echantillon)
intervalle = norm.interval(0.95, loc=m, scale=es)
print(round(intervalle[0],2),round(intervalle[1],2))
```

48.31 49.95

Ce code :

- ⊕ importe deux fonctions pour calculer l'erreur standard et la loi normale ;
- ⊕ calcule la moyenne de notre échantillon ;
- ⊕ calcule l'erreur standard de notre échantillon ;
- ⊕ calcule l'intervalle à 95 % à partir d'une fonction dédiée qui nécessite deux paramètres : la moyenne et l'erreur standard ;
- ⊕ affiche le résultat arrondi.

Nous avons pris le temps dans un cas particulier de faire le lien entre les données et les tests. Cela peut sembler plus ardu que de cliquer directement sur un bouton. Suivre ces étapes permet de s'assurer du sens de l'analyse. Cependant, comme pour les logiciels, la majorité des traitements ne nécessite pas de comprendre toutes ces étapes et de nombreuses fonctions sont directement disponibles.

### 6.6.3 Significativité d'un tableau croisé

Le test classiquement utilisé pour un tableau croisé est le test du *chi-deux* qui donne une information sur la proximité du tableau croisé obtenu avec les données par rapport à celui qu'on aurait eu si les deux variables croisées étaient indépendantes. Ce test peut être obtenu avec la fonction `chi2_contingency` de *Scipy*. Pour avoir l'information sur la significativité statistique de ce qui est observé dans le tableau croisé précédent entre la proportion de femmes et la proportion de personnes âgées :

```
from scipy.stats import chi2_contingency
tableau_croise = pd.crosstab(donnees["prop_f_C"],
                             donnees["prop_sup75_C"])
chi2 = chi2_contingency(tableau_croise)
print(round(chi2[0],1), chi2[1])
```

2188.9 0.0

Ce code :

- ➊ importe la fonction `chi2_contingency` du module statistique de *Scipy*;
- ➋ construit le tableau croisé avec `crosstab` que stocké dans la variable `tableau_croise`;
- ➌ applique la fonction `chi2_contingency` sur ce tableau qui renvoie le résultat du test statistique mis dans la variable `chi2`;
- ➍ affiche les deux premiers éléments de la variable `chi2`.

Le retour de cette fonction comprend 4 informations : la valeur de la statistique du chi2, la *p-value*, le degré de liberté du tableau et enfin la situation obtenue si les variables étaient indépendantes. Dans notre cas, l'information importante est que la *p-value* est très proche de zéro, ce qui signifie que l'hypothèse initiale (que les deux variables sont indépendantes) est très peu probable.

Nous pouvons même aller plus loin, et soustraire au tableau empirique de nos données le tableau théorique qu'on aurait eu à l'indépendance pour savoir où les différences sont les plus importantes :

```
tableau_croise - chi2[3]
```

	Q1	Q2	Q3	Q4
prop_f_C				
Q1	342.49	-94.17	-98.30	-150.01
Q2	619.93	249.34	-190.42	-678.85
Q3	51.19	355.62	144.44	-551.25
Q4	-1013.61	-510.79	144.28	1380.11

Ce code soustrait le tableau croisé avec le tableau théorique pour directement identifier les cases dans lesquelles la différence est la plus importante. Dans ce cas, cela concerne le dernier quartile de la proportion de femmes.

### ?

### Exercice

**Obtenez les écarts non pas absolus mais relatifs à partir des données précédentes.**

Il faut diviser chaque case du tableau par rapport aux valeurs de référence de `chi2[3]`.

#### 6.6.4 Le V de Cramer

Pour caractériser l'association entre deux variables qualitatives, un test souvent utilisé est le V de Cramer. Il s'agit d'un indicateur pour les tableaux croisés qui varie entre 0 et 1.

Cependant, cet indicateur n'est pas implémenté dans les bibliothèques habituelles comme *Scipy*. Dans ce cas, vous avez deux solutions : identifier une bibliothèque spécifique qui a la fonction, ou bien implémenter le test vous-même.

La bibliothèque *Researchpy* que nous avons mentionnée dans le chapitre 3 implémente directement le test de Cramer dans son calcul des tableaux croisés.

```
import researchpy
tableau = researchpy.crosstab(donnees["prop_f_C"],
                               donnees["prop_sup75_C"], test= "chi-square")
tableau[1]
```

```
Chi-square test  results
0  Pearson Chi-square ( 9.0) =    2188.87
1                  p-value =      0.00
2                  Cramer's V =     0.14
```

La fonction renvoie deux éléments : le tableau croisé et les tests du tableau croisé. Nous affichons ces tests.

### ?

### Exercice

**Écrivez la fonction qui permet de calculer le V de Cramer pour un tableau croisé.**

Voici un exemple qui utilise la bibliothèque *Numpy* (calcul sur des tableaux) pour faire le calcul de la racine carrée `sqrt`.

```
import numpy as np

def cramers_V(tableau_croise):
    chi2 = chi2_contingency(tableau_croise)[0]
    n = tableau_croise.sum().sum()
    return np.sqrt(chi2 / (n*(min(tableau_croise.shape)-1)))
```

Écrire son test permet en outre d'intégrer les corrections spécifiques pour prendre en compte les particularités de certains échantillons.

### 6.6.5 Différence entre deux groupes

Une manière de tester la différence entre les valeurs des deux groupes est de comparer la moyenne des valeurs de chaque groupe. Pour déterminer si cette différence est significative, nous utilisons le test de comparaison de moyennes (test de Student<sup>14</sup>). Ce test dans la bibliothèque *Scipy* s'appelle `ttest_ind`.

#### Exercice

**Vérifiez si les deux échantillons ont la même variance.**

Utilisez la fonction `std` sur les deux échantillons.

Dans notre cas, comme nous travaillons sur la population complète des villes françaises, nos échantillons ne sont pas indépendants. Mais pour l'exemple, nous allons tester si la moyenne des proportions de personnes âgées est statistiquement différente entre les petites villes et les grandes villes.

```
from scipy.stats import ttest_ind

med = donnees["P15_POP"].median()
pv = donnees[donnees["P15_POP"] < med][["prop_sup75"]].dropna()
gv = donnees[donnees["P15_POP"] >= med][["prop_sup75"]].dropna()

print("Moyenne groupe 1 : {}\nMoyenne groupe 2 : {}".format(round(pv.mean(),2),round(gv.mean(),2)))

test = ttest_ind(pv,gv)
print("La p-value du test est {:.e}".format(test[1]))
```

14. Les hypothèses pour faire ce test sont que les échantillons sont indépendants, distribués de manière identique suivant une loi normale de même variance. Suivant les domaines, plus ou moins d'énergie est investie à vérifier ce genre d'hypothèses...

Moyenne groupe 1 : 10.4

Moyenne groupe 2 : 9.44

La p-value du test est 1.587731e-82

Ce code :

- ⊕ importe la fonction `ttest_ind`;
- ⊕ calcule la médiane des populations;
- ⊕ crée le groupe avec uniquement les villes dont la population est plus petite que la médiane, prend la variable de la proportion des personnes âgées, enlève les valeurs nulles;
- ⊕ crée le groupe avec uniquement les villes dont la population est plus grande que la médiane, prend la variable de la proportion des personnes âgées, enlève les valeurs nulles;
- ⊕ affiche la moyenne des deux groupes;
- ⊕ calcule le test de Student pour ces deux groupes;
- ⊕ affiche la probabilité critique.

La différence entre ces deux échantillons est de 1 point environ. Cette différence est statistiquement significative.

Dans le cas où il y a plus de deux échantillons, la méthode à utiliser est l'ANOVA (et le test de Fisher). De la même manière, l'hypothèse de l'indépendance des échantillons va être faux dans notre cas. La fonction correspondante est `f_oneway`.

## ?

### Exercice

#### Faites une ANOVA pour les quartiles des populations.

Nous avons déjà codé les quartiles, on peut donc construire 4 échantillons :

```
from scipy.stats import f_oneway
Q1 = donnees[donnees["P15_POP_C"]=="Q1"]["prop_sup75"]
Q2 = donnees[donnees["P15_POP_C"]=="Q2"]["prop_sup75"]
Q3 = donnees[donnees["P15_POP_C"]=="Q3"]["prop_sup75"]
Q4 = donnees[donnees["P15_POP_C"]=="Q4"]["prop_sup75"]
print(Q1.mean(), Q2.mean(), Q3.mean(), Q4.mean())
f_oneway(Q1, Q2, Q3, Q4)
```

La fonction `f_oneway` retourne la valeur du test statistique et la *p-value*, ici significative.

Une autre fonction que vous pouvez utiliser pour tester la différence entre deux échantillons est le test de rang-somme de Wilcoxon `ranksums`. Elle teste si les deux échantillons de données proviennent de la même distribution. Pour l'appliquer à notre cas précédent, on peut vouloir tester si les deux derniers quartiles Q3 et Q4 sont vraiment si différents en termes de répartition.

```
from scipy.stats import ranksums

Q3 = donnees[donnees["P15_POP_C"]=="Q3"]["prop_sup75"].dropna()
Q4 = donnees[donnees["P15_POP_C"]=="Q4"]["prop_sup75"].dropna()
test = ranksums(Q3,Q4)
print("La p-value est {:.e}".format(test[1]))
```

La p-value est 1.387835e-43

Ce code :

- ➊ importe la fonction `ranksums`;
- ➋ définit la liste des valeurs pour le troisième quartile;
- ➌ définit la liste des valeurs pour le quatrième quartile;
- ➍ calcule le test pour ces deux listes;
- ➎ affiche la probabilité critique.

Le résultat indique que ces deux distributions sont statistiquement différentes.

Nous n'avons pas pu traiter de l'ensemble des tests statistiques existant : la tâche serait sans fin. Suivant votre discipline, vous allez devoir analyser des données d'expérience ou vous conformer à des règles spécifiques. Pour certains traitements classiques comme les modèles présentés dans le prochain chapitre, ces statistiques sont calculées dans le cadre de modèles plus généraux. Dans les autres cas, le plus simple est de faire une recherche dans la documentation de la bibliothèque *SciPy* avec le nom du test, ou directement sur internet. Il serait très étonnant que d'autres utilisateurs n'aient pas déjà eu la même question.

## 6.7 Synthèse du chapitre

Les bibliothèques de Python permettent de calculer les statistiques descriptives et inférentielles rencontrées en SHS.

Une fois les données mises en forme sous un tableau, la première étape est de vérifier la nature des données et de construire les variables nécessaires. Le Notebook Jupyter permet facilement d'explorer un tableau de données.

Les tableaux *Pandas* permettent rapidement d'obtenir les indicateurs classiques comme la médiane, la moyenne ou l'écart-type, tout en facilitant le recodage et l'obtention de tri à plat. Les tests statistiques classiques existent sous la forme de fonctions dédiées disponibles dans la bibliothèque *SciPy*.

Avant de voir des traitements plus complexes, regardons comment visualiser nos données.



# Chapitre 7

## Visualiser

Ce chapitre montre comment produire des visualisations. L'opération de visualisation est présente tout au long du travail sur des données, de l'étape d'exploration pour avoir une idée des informations existantes, à la restitution, pour produire une synthèse diffusable auprès d'un public plus large.

### 7.1 Pourquoi programmer des visualisations ?

Comme pour calculer des statistiques, des logiciels existent pour produire des visualisations standardisées. Cependant, dans certaines situations, vous souhaitez produire des représentations spécifiques, comme mélanger des informations temporelles avec des informations spatiales pour saisir une dynamique d'évolution. Vous pouvez aussi avoir envie de systématiser la production de visualisations en définissant les étapes par un script plutôt qu'en cliquant sur des boutons dans un logiciel. Ou tout simplement, vous souhaitez produire des visualisations intermédiaires de manière interactive dans un Notebook au cours de votre analyse, réalisée en Python, pour vérifier certaines intuitions.

Pourquoi utiliser la programmation pour faire une visualisation ? Une première réponse est la possibilité d'avoir un plus grand contrôle sur les paramètres.

Une seconde réponse est la possibilité d'automatiser et de reproduire ces visualisations, ce qui est très utile par exemple quand vos données changent et que vous voulez reproduire exactement le même type de figure. Il suffit alors d'exécuter le même code avec les nouvelles données.

De nombreuses bibliothèques Python permettent de faire des visualisations destinées à des usages différents. Vous trouverez des bibliothèques qui vous permettent de faire des visualisations interactives, des visualisations 2D, 3D, certaines sont

spécialisées pour la représentation de données statistiques, médicales, d'astrophysique, ou encore géographiques. Choisir le bon outil nécessite de se poser la question des opérations à réaliser.

Par exemple, si vous avez besoin de tracer un diagramme en barres pour avoir une idée de la proportion de certaines caractéristiques de vos données, il s'agit d'un traitement standard et les outils dits de haut niveau existent pour rapidement effectuer cette opération. Par contre, si vous voulez construire un nuage de points qui se superpose avec des évolutions temporelles, peut-être faudra-t-il faire appel à des outils de plus bas niveau.

Pour le dire simplement, une bibliothèque de haut niveau sera plus facile à manier mais permettra moins de flexibilité qu'une bibliothèque de bas niveau qui permettra souvent de modifier les éléments fondamentaux d'une image : dans la mesure où une image est un ensemble de points (les pixels), le plus bas niveau est de modifier les pixels un par un. Comme cela est fastidieux, il est préférable d'éviter de se retrouver dans cette situation. Mais cela peut être utile pour des besoins spécifiques.

Dans ce chapitre, nous allons d'abord regarder comment produire rapidement des visualisations avec une bibliothèque de haut niveau permettant d'aller très vite vers des rendus visuels de bonne qualité, et ensuite vous familiariser avec la grammaire plus générique de la construction de graphiques.

Nous n'aborderons pas les fonctions les plus avancées, dont nous ne dirons que quelques mots en fin de chapitre. Nous ne présenterons pas non plus une théorie de la représentation des données qui vous dirait comment choisir le bon graphique, mais nous vous invitons si cela vous intéresse à jeter un coup d'œil aux travaux d'Edward Tufte<sup>1</sup>.

## Information

### Pixels vs vectoriel

Les images se répartissent en deux types de formats. Soit l'image est un tableau de pixels qui sont autant de points (aussi appelée image raster), soit elle est définie par une superposition d'éléments géométriques (des ronds, des lignes, etc.). Dans ce dernier cas, la structure de l'image est une série de formes « mathématiques » définies par leurs coordonnées qui sont ensuite représentées : il s'agit d'une image vectorielle. Vous pouvez agrandir une image vectorielle autant que vous le voulez sans perdre de précisions ou de qualité.

---

1. À la croisée entre les statistiques et l'art, la réflexion d'Edward Tufte porte sur la philosophie de la représentation de l'information, permettant de rentrer perceptible à un lecteur à la fois la richesse des données et le message central. Par exemple, n'hésitez pas à feuilleter *The Visual Display of Quantitative Information* Tufte (1973).

En effet, l agrandissement porte d abord sur les fonctions mathématiques et seul le rendu visuel est fait en pixel. Si vous le faites sur une image raster, vous finirez par voir les pixels individuels car l image initiale est définie par un nombre limité de pixels.

Le choix d une visualisation adaptée aux données et aux résultats à communiquer est loin d être évident. Les images peuvent facilement trahir les données. Suivant les formes et les couleurs, le spectateur verra un résultat différent. Pour cette raison, des règles existent sur le choix des représentations. Ce chapitre se concentre sur des formats de visualisation classique que vous connaissez sûrement. Selon nous, le choix des formes et des couleurs devrait respecter une règle de simplicité en privilégiant le fond sur la forme.

Une dernière remarque que nous avons déjà faite, mais qui prend toute son importance ici : ce livre est en noir et blanc. Les visualisations utilisent généralement des couleurs pour transmettre l information. Nous avons donc dû trouver un compromis et essayer de rendre au mieux les visualisations présentées<sup>2</sup>.

## 7.2 Petit tour des lieux

Pour vous permettre de choisir, nous vous présentons trois manières de représenter des données de notre exemple sur les villes françaises : d abord directement avec la bibliothèque *Pandas* que nous avons déjà rencontrée, ensuite à partir de la bibliothèque de bas niveau *Matplotlib* et enfin avec une bibliothèque dédiée *Seaborn*. Avoir trois approches conduira à quelques aller-retours, mais cela permet d ouvrir les possibilités.

Commençons donc par charger le jeu de données utilisé dans le chapitre précédent avec comme objectif de visualiser le nombre de villes par départements.

```
import pandas as pd
tableau = pd.read_csv("./data/data-chap6.csv")
donnees = tableau[["CODGEO", "REG", "DEP", "LIBGEO",
                    "P15_POP", "P15_POPF", "P15_POP90P",
                    "C15_POP15P_CS6", "P15_POP7589",
                    "prop_f", "prop_sup75", "prop_ouvriers"]]
```

---

<sup>2</sup>. Pour cette raison, nous précisons souvent une option de couleur dans les visualisations (généralement du gris ou du noir). Vous pouvez enlever cette option pour avoir les couleurs automatiques de Python, et nous verrons par la suite comment configurer ces couleurs. Dans tous les cas, les visualisations devraient être plus jolies chez vous.

### 7.2.1 Représenter les données avec *Pandas*

Comme vous avez pu le voir dans les chapitres précédents, *Pandas* permet de manipuler les tableaux de données. Comme la visualisation est une étape importante de l'analyse, la bibliothèque *Pandas* intègre des fonctions comme la méthode `plot` déjà rencontrée permettant de construire des visualisations à partir des données du tableau. Voici un exemple de visualisation directement produit à partir d'un tableau :

```
# Mise en forme des données
distribution = donnees["DEP"].value_counts()
# Création de la figure
ax = distribution.plot(kind="bar", figsize=(20,5),
                       title="Nombre de villes par département")
# Options et visualisation
ax.set_xlabel("Départements")
ax.set_ylabel("Nombre")
```

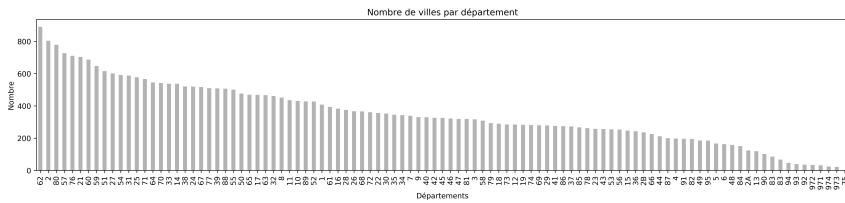


Fig. 7.1 – Exemple de représentation d'un graphique produit avec Pandas

Ce code :

- ➊ prend la distribution des villes par département avec `value_counts` stockée dans la variable `distribution` (qui est aussi un tableau *Pandas*) ;
- ➋ utilise la méthode `plot` déjà rencontrée des tableaux *Pandas* en précisant que le type de visualisation est un diagramme en barres, en précisant la taille de la figure avec `figsize` et en lui donnant un titre avec `title`. Ce graphique est stocké dans la variable `ax` ;
- ➌ ajoute le nom des axes X et Y avec les méthodes spécifiques de manipulation des graphiques et affiche la figure 7.1.

En une ligne, complétée de deux pour la mise en forme, nous obtenons une visualisation correcte. Elle apparaît petite dans ce manuel pour tenir sur la page mais vous pouvez changer ses dimensions. D'autres caractéristiques pourraient être facilement modifiées : les couleurs, la taille des barres, etc. Pour construire ces visualisations, *Pandas* s'appuie en fait sur une bibliothèque de plus bas niveau, *Matplotlib*.

### 7.2.2 *Matplotlib*, une bibliothèque de bas niveau

Créée en 2002 par John Hunter et s'inspirant d'autres interfaces existantes à l'époque pour la grammaire de programmation, cette bibliothèque centrale pour Python permet à la fois de faire des graphiques de manière assez rapide et de contrôler tous les paramètres.

Les autres bibliothèques de plus haut niveau sont souvent basées sur *Matplotlib* ce qui permet alors une compatibilité. Cela permet souvent de commencer une figure avec une bibliothèque de haut niveau et de modifier les détails avec *Matplotlib*. Pour refaire le diagramme précédent uniquement avec *Matplotlib* :

```
import matplotlib.pyplot as plt
# Mise en forme des données
distribution = donnees["DEP"].value_counts()
x = range(0,len(distribution))
y = list(distribution)
# Création de la figure
fig,ax = plt.subplots()
ax.bar(x,y)
# Options et visualisation
ax.set_title("Nombre de villes par département")
ax.set_xlabel("Départements")
ax.set_ylabel("Nombre")
plt.show()
```

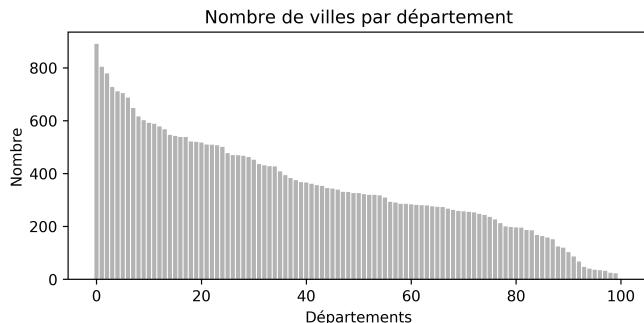


Fig. 7.2 – Exemple d'un graphique produit avec *Matplotlib*

Ce code :

- ✿ charge le module `pyplot` de *Matplotlib* avec l'alias `plt`;
- ✿ crée la distribution des données et transforme la distribution en deux listes, les abscisses `x` et les ordonnées `y`;

- ⊕ initie un nouveau graphique en donnant la taille qui comprend une figure `f` et un axe `ax` (nous verrons par la suite la signification de ces entités) ;
- ⊕ affiche sur l'axe `ax` un diagramme en barres avec la méthode `bar` qui prend comme argument deux listes, pour les abscisses et les ordonnées ;
- ⊕ ajoute le titre et les informations sur les axes ;
- ⊕ trace le graphique de la figure 7.2.

Comme vous pouvez le voir, il y a plus d'étapes, en particulier celle de déclarer la figure. L'utilisation de *Pandas* dans le cas précédent nous a permis de limiter le nombre de lignes de code. Toutefois, en comparant les deux graphiques, nous pouvons constater quelques différences qui peuvent justifier par exemple de privilégier l'une ou l'autre approche.

### 7.2.3 *Seaborn*, une bibliothèque de haut niveau

Enfin, une dernière manière de faire est d'utiliser une bibliothèque dédiée aux visualisations complexes. *Seaborn*, abrégée généralement en `sns`, a été développée par un étudiant de l'université de Californie de Stanford lassé de faire encore et encore les mêmes visualisations. Elle est orientée vers la représentation graphique des statistiques comme les tableaux de corrélation ou les barres d'erreurs en ajustant facilement les niveaux de coloration avec un minimum de lignes de code. Cela en fait un outil très utile et complet pour l'exploration des données en première approche.

Pour réaliser la visualisation précédente, après avoir chargé la bibliothèque avec `import seaborn as sns`, le code est alors `sns.countplot(x="DEP", data=donnees)`. La fonction `countplot` permet directement de faire le comptage de chaque catégorie et d'afficher le diagramme avec un choix automatique de couleur. Il n'est même plus besoin de calculer un tableau intermédiaire de données.

Il y a des limites pour ce type de visualisation automatique : la taille de la figure n'est généralement pas celle que nous souhaiterions, la légende n'est souvent pas lisible, il n'y a pas de titre. Pour compléter ou ajuster, l'idée est alors d'utiliser la bibliothèque de bas niveau *Matplotlib*. Elle fait le lien entre les différentes visualisations.

#### ?

#### Exercice

Construisez un graphique similaire avec les régions en utilisant les trois méthodes précédentes.

## 7.3 La grammaire des visualisations de *Matplotlib*

Pour le traitement de données, *Matplotlib* représente la bibliothèque de référence sur laquelle se construisent de nombreuses fonctions.

### 7.3.1 Quelques éléments généraux

La bibliothèque *Matplotlib* est spécialisée dans les représentations graphiques à deux dimensions<sup>3</sup>. Parmi plusieurs de ses points forts, notons la facilité de construire rapidement un graphique, d'ajuster l'ensemble des caractéristiques (taille des polices, place des légendes, etc.) et de sauvegarder le rendu dans de nombreux formats.

Dans les faits, nous utilisons pour faire les représentations le module `pyplot` abrégé usuuellement `plt` qui permet d'écrire les éléments graphiques que nous voulons, par exemple : « afficher », « faire un histogramme », « mettre ce texte comme titre » (les autres modules sont les fonctions permettant de les tracer). Pour l'importer :

```
import matplotlib.pyplot as plt.
```

La diversité des usages possible de *Matplotlib* conduit à une documentation dense et s'y orienter peut être un peu difficile<sup>4</sup>. Nous présenterons les principaux éléments pour la visualisation des données.

La démarche pour construire un graphique peut être décomposée en six étapes :

1. préparer les données dans le bon format ;
2. choisir le type de représentation souhaité ;
3. définir une nouvelle figure dans Python ;
4. afficher les données avec la fonction de représentation souhaitée ;
5. ajouter des informations de contexte comme le titre ou le nom des axes ;
6. afficher et sauvegarder la figure.

Pour avoir une idée des différents éléments mobilisés par la bibliothèque, le schéma ci-dessous représente la figure générale `Figure` qui est composée de graphiques plus ou moins nombreux et/ou superposés, les `Axes` et enfin les caractéristiques de chaque graphique (comme le titre `title`, les noms des axes `xlabel` et `ylabel`, etc.).

---

3. Cela signifie que pour dessiner autre chose que des graphiques (donc des visualisations pour le traitement de données), il faut se tourner vers d'autres outils.

4. Pour donner un ordre de grandeur, la bibliothèque représente un total de 70 000 lignes de code et se trouve utilisée dans de très nombreuses bibliothèques.

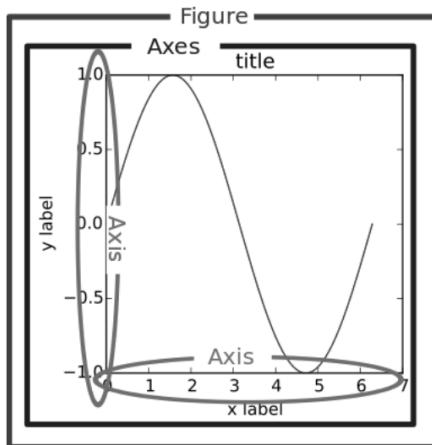


Fig. 7.3 – Structure d'une visualisation avec Matplotlib

Chaque élément peut être modifié : cela signifie que le nombre de graduations sur les axes, ou la police de caractère des axes, ou leur orientation, leur couleur, ou encore leur taille, peuvent être paramétrés. À chaque fois, cela signifie rajouter une ligne de code qui fait l'opération souhaitée sur l'objet concerné.

Par exemple, les graduations de l'axe des abscisses (x) s'appellent les `xticks` et paramètrent à la fois l'écartement entre les graduations (la localisation) et le texte (label). À partir d'un axe existant d'une figure, il est possible de récupérer l'information de ces abscisses avec des fonctions dédiées, comme `get_xticks` et `get_xticklabels`, de la modifier et de l'appliquer avec `set_xticks` et `set_xticklabels`. Nous verrons ces modifications dans le cadre d'exemples. Ne soyez pas surpris de découvrir de nouvelles manières de faire.

### 7.3.2 Définir une figure

Comme souvent en programmation, il existe plusieurs façons de procéder pour créer une nouvelle figure. Voici un petit récapitulatif des situations que vous pouvez rencontrer, même si nous avons une préférence pour la dernière qui déclare explicitement la création de la figure :

- ✚ implicitement lors de l'utilisation d'une fonction de visualisation comme `plot`;
- ✚ en amont de la visualisation avec la fonction `figure` qui permet d'indiquer des paramètres, comme la taille ;

- ⊕ avec la fonction **subplots** qui permet de définir des figures complexes avec plusieurs graphiques côté à côté.

Donnons un exemple. Nous avons le nombre d'habitants par ville et une question qui se pose alors est : quelle est la répartition des villes en France ? Pour cela, l'histogramme permet de représenter rapidement une distribution. La fonction **hist** de *Matplotlib* permet de tracer les histogrammes.

```
import matplotlib.pyplot as plt

# Création de la figure
fig, ax = plt.subplots()
ax.hist(donnees["P15_POP"], bins=20)

# Options et visualisation
ax.set_title("Histogramme de la taille des villes")
ax.set_xlabel("Nombre d'habitants")
ax.set_ylabel("Nombre de villes")
plt.show()
```

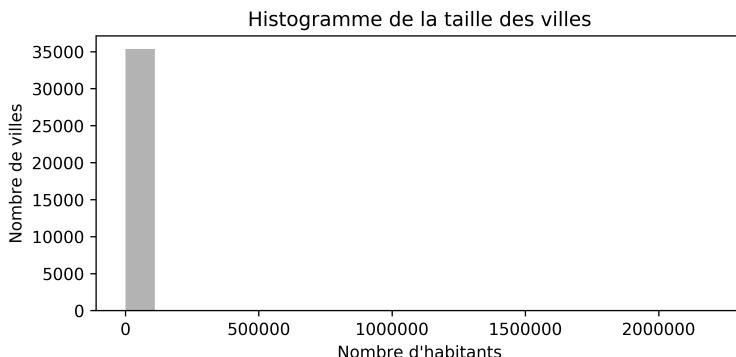


Fig. 7.4 – Exemple d'une figure définie avec subplots

Ce code :

- ⊕ charge le module **pyplot** permettant de définir des visualisations ;
- ⊕ définit explicitement une nouvelle figure qui crée en même temps l'objet figure **fig** et ses composantes d'axes **ax**.
- ⊕ utilise la fonction **hist** associée à l'axe pour afficher l'histogramme des valeurs de la colonne P15\_POP de nos données, en précisant que cet histogramme doit être découpé en 20 intervalles avec **bins=20** ;
- ⊕ met un titre avec la fonction **set\_title** ;
- ⊕ met une légende pour les axes des abscisses avec **set\_xlabel** ;

- ⊕ met une légende pour les axes des ordonnées avec `set_ylabel` ;
- ⊕ affiche la figure 7.4.

Nous constatons que l'histogramme se résume à une barre... en effet, la distribution est très asymétrique, avec une ville (Paris) qui a beaucoup d'habitants ce qui crée un intervalle de variation très grand. Cela contribue à rendre ce type de visualisation peu efficace et nous amène à réfléchir sur la meilleure manière de présenter les données.

## ?

### Exercice

#### Filtrer pour enlever les villes les plus peuplées avant de tracer l'histogramme.

Pour réduire l'asymétrie, on va filtrer les villes les plus peuplées (moins de cent mille habitants) :

```
donnees_filtrees = donnees[donnees["P15_POP"]<100000][["P15_POP"]]
plt.hist(donnees_filtrees), bins=20)
```

Essayez avec plusieurs valeurs différentes de filtre pour voir comment cela fait changer la visualisation.

Ne soyez pas surpris si vous rencontrez une autre manière de faire qui n'utilise pas exactement les mêmes fonctions. Plutôt que de créer l'histogramme sur l'axe `ax`, il est possible de le faire avec la ligne `plt.hist`. De même, vous pouvez mettre un titre avec la fonction `plt.title` et le titre des abscisses avec `plt.xlabel`. Le lien entre l'axe et la figure est fait de manière implicite. Comme dans notre cas, il n'y a qu'un axe, cela marche sans souci... mais les soucis commencent quand il y a plusieurs axes différents dans les figures complexes.

Les fonctions que vous allez le plus utiliser sont :

- ⊕ `ax.set_title` (ou `plt.title`) pour mettre un titre ;
- ⊕ `ax.set_xlabel` (ou `plt.set_xlabel`) pour mettre le nom des abscisses ;
- ⊕ `ax.set_ylabel` (ou `plt.set_ylabel`) pareil, pour les ordonnées ;
- ⊕ `ax.set_xlim` (ou `plt.set_xlim`) pour définir la limite de l'axe des abscisses (par exemple, `plt.xlim(0,100)` va limiter la visualisation entre 0 et 100) ;
- ⊕ `ax.set_ylim` (ou `plt.set_ylim`) a un effet similaire pour l'axe des ordonnées ;
- ⊕ `plt.savefig` pour sauvegarder la figure, vous pouvez rajouter `dpi=400` (ou plus que 400) pour augmenter la résolution de l'image.

Ces fonctions permettent de paramétrier des options générales de la figure, quelle que soit la représentation. Il est donc temps de voir les différentes visualisations possibles.

### 7.3.3 Les différents types de figures

Nous venons juste de présenter l'histogramme, très utile pour avoir une information sur la distribution d'une variable quantitative. D'autres visualisations reviennent régulièrement :

- ➊ des diagrammes en barres ;
- ➋ des diagrammes circulaires ;
- ➌ des courbes ;
- ➍ des nuages de points (*scatterplot*) ;
- ➎ des *boxplots* (boîtes à moustaches).

Ces visualisations ont chaque fois des paramètres spécifiques et il est compliqué de rentrer dans le détail de chacune. Plutôt que de faire une présentation théorique, nous allons nous concentrer sur des exemples détaillés que vous pouvez adapter pour débuter puis compléter avec la documentation de Python.

#### Diagramme en barres

Les diagrammes en barres font correspondre à une liste d'éléments des barres dont la hauteur correspond à une liste de valeurs. Pour cette raison, la fonction prend deux listes en argument.

Commençons par afficher le diagramme en barres du nombre de villes par région en France avec la fonction `bar` de `pyplot` qui prend comme arguments une première liste qui donne la position des barres (les abscisses) et une deuxième qui donne la hauteur (les ordonnées).

```
# Mise en forme des données
distribution = donnees["REG"].value_counts()
x = range(0,len(distribution))
y = list(distribution)

# Création de la figure
fig, ax = plt.subplots()
ax.bar(x,y, tick_label=distribution.index)

# Options et visualisation
ax.set_title("Nombre de villes par région")
ax.set_xlabel("Régions")
ax.set_ylabel("Nombre de villes")
plt.show()
```

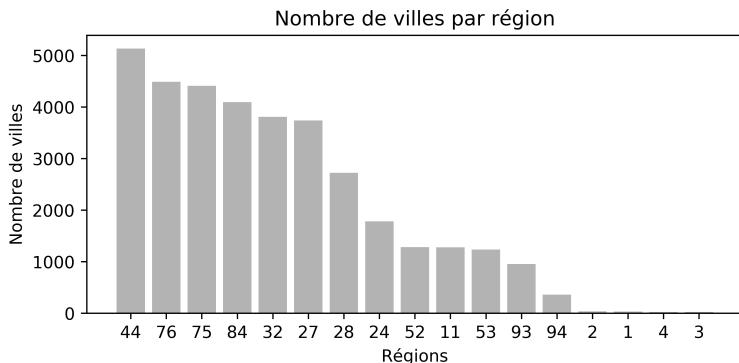


Fig. 7.5 – Exemple d'un diagramme en barres

Ce code :

- ➊ calcule la distribution des villes par région ;
- ➋ crée deux listes, `x` qui va de 1 au nombre de région, et `y` avec les valeurs ;
- ➌ déclare une nouvelle figure ;
- ➍ rajoute un graphique `bar` sur l'axe avec `x` et `y` comme valeurs, l'option `tick_label` pour indiquer le nom de chaque barre contenu dans l'index de la série `Pandas` et l'option indiquant la couleur `color="gray"` ;
- ➎ ajoute des informations de présentation du graphique : titre général et des axes, et affiche le graphique 7.5.

## Diagramme circulaire

Le diagramme circulaire est obtenu avec la fonction `pie`, qui prend comme argument une liste et indique la proportion de chaque élément par rapport à l'ensemble. Dans le cas de nos données, cela donne :

```
# Mise en forme des données
distribution = donnees["REG"].value_counts()
x = list(distribution)

# Création de la figure
fig, ax = plt.subplots()
ax.pie(x, labels=distribution.index)

# Options et visualisation
plt.title("Nombre de villes par région")
plt.show()
```

Nombre de villes par région

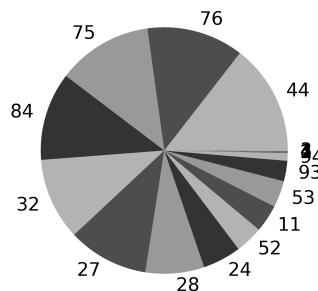


Fig. 7.6 – Exemple d'un diagramme circulaire

Ce code crée la liste des valeurs, utilise la fonction `pie` en indiquant les valeurs et le nom des labels pour produire la figure 7.6. Comme certains labels sont trop proches, cela donne une superposition qui est peu jolie. Pour l'éviter, il est possible de mieux regrouper les catégories ou de modifier les labels.

### Exercice

#### Mettez uniquement les labels sur les 10 premiers départements.

Au lieu de mettre directement `distribution.index` pour l'index, nous allons créer une liste adaptée qui comprend les 10 premiers éléments de l'index, puis une chaîne vide.

```
n = len(distribution)
numeros_departements = list(distribution.index)[0:10]+[""]*(n-10)
ax.pie(x, labels=numeros_departements)
```

Comme ça, vous pouvez contrôler tous les éléments que vous affichez.

Pour le moment, les couleurs sont générées automatiquement, ici entre le blanc et le gris, mais généralement suivant une palette colorée. Il existe des paramètres permettant de contrôler les couleurs et nous revenons par la suite sur comment les utiliser.

### Courbe de points

Les courbes sont des points reliés les uns aux autres. Elles peuvent être tracées avec la fonction `plot` et sont surtout utilisées en SHS pour représenter des évolutions temporelles. Comme nous n'avons pas d'évolution temporelle dans notre jeu de données sur les villes, utilisons d'autres données : l'évolution de la population de Paris (données Insee).

```

# Mise en forme des données
annees = range(1999,2014)
population_paris = [2.123, 2.129, 2.137, 2.145, 2.154, 2.161,
                    2.172, 2.181, 2.193, 2.211, 2.234, 2.243,
                    2.249, 2.240, 2.229]

# Création de la figure
fig, ax = plt.subplots()
ax.plot(annees,population_paris,"o-", label='Paris')

# Options et visualisation
ax.set_title("Population de Paris en millions d'habitants")
ax.set_xlabel("Années")
ax.set_ylabel("Millions d'habitants")
plt.legend()
plt.show()

```

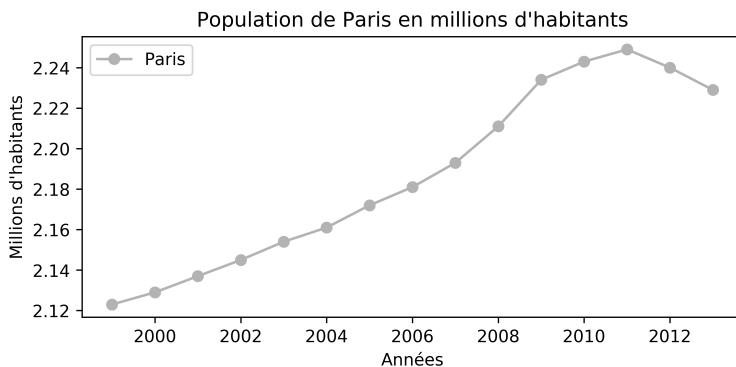


Fig. 7.7 – Exemple de représentation d'une courbe

Ce code :

- ➊ utilise `range` pour générer les années de 1999 à 2013 ;
- ➋ définit une liste avec le nombre d'habitants par année à Paris ;
- ➌ utilise la fonction `plot` pour afficher les points, en précisant la forme avec "`o-`" qui indique afficher les points et traits pleins et `label` pour donner un nom à la courbe ;
- ➍ ajoute les informations de titrage ;
- ➎ utilise `plt.legend` pour ajouter une légende ;
- ➏ trace la figure 7.7.

## ?

### Exercice

**Changez les ordonnées pour faire commencer à 0.**

Utilisez `plt.ylim(0, 2.4)` et regardez comment l'interprétation change de la variation

Il est possible de changer facilement la propriété des lignes avec `plot`. Ainsi, "`o-`" indique que l'on veut des points et des traits. Il est possible d'indiquer d'autres types de courbes : des pointillés avec "`:`", des carrés plutôt que des points avec "`s`", voire de mettre directement la couleur avec "`r`" par exemple pour rouge. Si vous voulez juste des pointillés rouges avec des petits points, indiquez "`r.:`". Pour plus d'information, jetez un coup d'œil à la documentation.

## Nuage de points

Les nuages de points permettent de représenter avec plus de facilité des couples d'informations afin de voir leur relation. La fonction `scatter` permet de tracer un nuage de point avec deux informations : la liste des abscisses et la liste des ordonnées.

Suite au traitement du chapitre sur les statistiques où nous interrogions sur la proportion des femmes dans les villes françaises, nous pouvons représenter cette distribution et celle des personnes âgées de plus de 75 ans.

```
# Mise en forme des données
x = list(donnees["prop_f"])
y = list(donnees["prop_sup75"])

# Création de la figure
fig, ax = plt.subplots(figsize=(20,10))
ax.scatter(x,y, s=1, alpha=0.3)

# Option et visualisation
ax.set_title("Nuage de points")
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion des plus de 75 ans (%)")
plt.show()
```

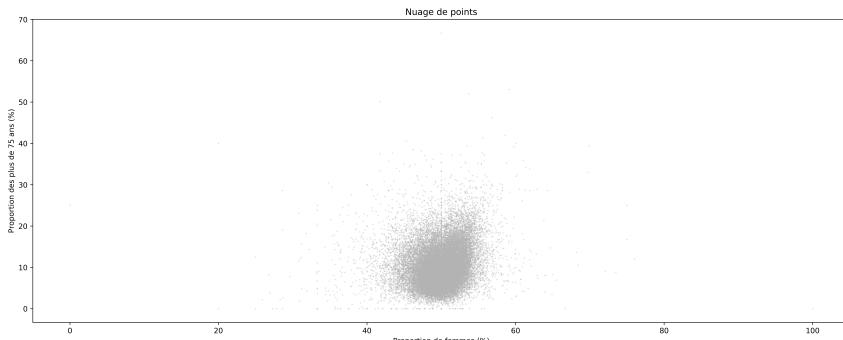


Fig. 7.8 – Exemple de représentation d'un nuage de points

Ce code :

- ➊ définit les deux listes d'abscisses (x) et d'ordonnées (y) ;
- ➋ déclare une figure comme précédemment ;
- ➌ utilise la fonction `scatter` pour représenter un point par ville avec suivant l'axe des abscisses la proportion de femmes et en ordonnée la proportion des plus de 75 ans, en indiquant que la taille des points est de 1 avec `s=1` ;
- ➍ déclare les titres et affiche la figure 7.8 où chaque point est une ville.

Nous le verrons dans la suite, la fonction `scatter` nous permet aussi de faire varier la couleur, la taille et la transparence des points en fonction de nos données.

### ?

### Exercice

**Représentez la proportion de femmes par rapport à la proportion d'ouvriers.**

Il suffit de déclarer différemment la variable y.

### Boîtes à moustaches (boxplot)

Enfin, un dernier type de représentation utile pour visualiser des données quantitatives est le *boxplot* ou, en français, l'indémodable « boîtes à moustaches ». Il donne une information sur les quartiles d'une variable quantitative, renseignant donc sur sa distribution<sup>5</sup>. Par exemple, pour représenter la proportion de femmes par ville, plutôt que de faire un histogramme, il est possible de faire un *boxplot* avec la fonction `boxplot`.

5. Les boxplots sont assez simples, pour aller plus loin vous pouvez regarder les diagrammes en essaims (*swarm plots*) et les diagrammes en violons (*violin plot*) disponibles dans les bibliothèques que nous présenterons ensuite.

```
# Mise en forme des données
dep = ["12", "13", "14", "15", "16", "17"]
data = [donnees[donnees['DEP']==str(x)]["prop_f"].dropna()
        for x in dep]

# Création de la figure
fig, ax = plt.subplots()
ax.boxplot(data, labels=dep, notch=True)

# Option et visualisation
ax.grid(axis='x')
ax.set_title("Proportion de femmes par département")
ax.set_ylabel("Pourcentage")
plt.xticks(rotation=30)
plt.show()
```

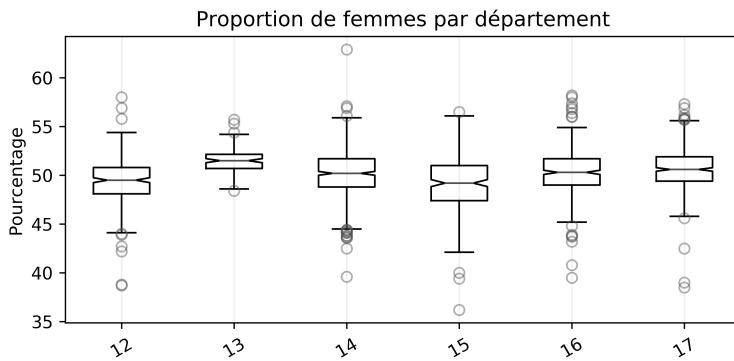


Fig. 7.9 – Exemple de visualisation d'un boxlot

Ce code :

- ➊ définit une liste de départements ;
- ➋ crée une liste avec pour chaque département les données correspondant à la proportion de femmes ;
- ➌ crée une nouvelle figure
- ➍ crée les boîtes à moustaches pour les données, mets les noms de départements avec `label` et rajoute une option de visualisation ;
- ➎ ajoute différentes options : ligne verticale, titres et rotation des labels, puis affiche la figure 7.9.

Le trait central correspond à la médiane, la boîte démarre au premier quartile et s'arrête au troisième quartile (les moustaches représentent par défaut 95 % du total, il est possible de spécifier la proportion avec l'option `whis=[5, 95]`). Nous utilisons `notch=True` pour rendre la médiane plus visible et avoir une indication de l'intervalle de confiance sur celle-ci (les triangles).

### ?

### Exercice

**Tracez une boîte à moustaches sur l'ensemble des villes françaises**

```
fig, ax = plt.subplots()  
ax.boxplot(donnees["prop_f"].dropna())  
plt.show()
```

Les quelques figures présentées ici n'épuisent pas tout ce que vous pouvez obtenir comme visualisations. Nous verrons dans des cas concrets d'autres visualisations possibles. De manière générale, il est toujours intéressant de jeter un coup d'œil sur les galeries existantes<sup>6</sup>.

### ?

### Exercice

**Tracez l'histogramme de la proportion des femmes.**

```
fig, ax = plt.subplots()  
ax.hist(donnees["prop_f"].dropna(), bins=100)  
plt.show()
```

## 7.3.4 Tracer plusieurs figures

Il est possible de tracer plusieurs figures en même temps en créant une visualisation qui a des sous-éléments avec la fonction `subplots`. Celle-ci permet de définir un quadrillage d'éléments.

Ce qui va se passer est que la déclaration des axes ne va plus correspondre à un axe unique, mais à une liste d'axes, chacun renvoyant à un graphique. Il est alors possible d'accéder à chaque sous-axe avec un numéro pour la ligne et un numéro pour la colonne. Par exemple, nous voulons tracer quatre histogrammes pour quatre départements différents. Et nous voulons que ce soit sur deux lignes et sur deux colonnes. Cela ajoute un niveau de représentation et donc nécessite de distinguer différents niveaux de titres.

La déclaration de `subplots` est la suivante :

---

6. Pour vous donner une idée : <https://python-graph-gallery.com/>.

```

# Mise en forme des données
b = [1000*i for i in range(0,100)]
data67 = donnees[donnees["DEP"]=="67"]["P15_POP"]
data68 = donnees[donnees["DEP"]=="68"]["P15_POP"]
data55 = donnees[donnees["DEP"]=="55"]["P15_POP"]
data57 = donnees[donnees["DEP"]=="57"]["P15_POP"]

# Création de la figure
fig,axes = plt.subplots(2, 2, sharex=True, sharey=True)
axes[0,0].hist(data67, bins=b, ec='w')
axes[0,0].set_title("Département 67")
axes[1,0].hist(data68, bins=b, ec='w')
axes[1,0].set_title("Département 68")
axes[0,1].hist(data55, bins=b, ec='w')
axes[0,1].set_title("Département 55")
axes[1,1].hist(data57, bins=b, ec='w')
axes[1,1].set_title("Département 57")

# Options et visualisation
plt.xlim(0,8000)
fig.suptitle("Histogramme par départements")

```

Histogramme par départements

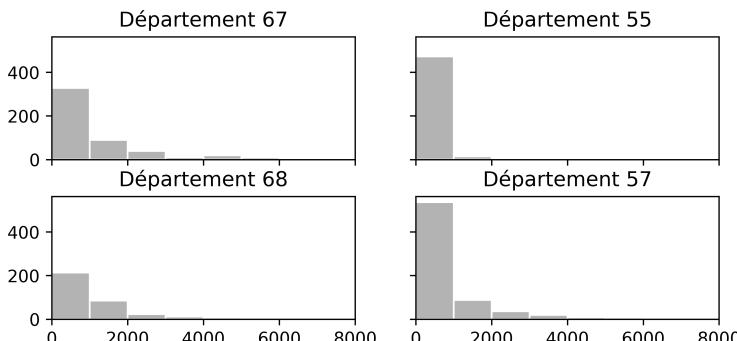


Fig. 7.10 – Exemple d'une figure avec plusieurs graphiques

Ce code :

- ➊ crée une liste qui définit les bornes des intervalles de l'histogramme à l'aide de `range` ;

- ⊕ crée 4 tableaux avec les données pour chaque département ;
- ⊕ crée une nouvelle figure avec la fonction `subplots` qui précise qu'il va y avoir 2 lignes et 2 colonnes de figures et donne la taille de la figure d'ensemble, en précisant que les axes des abscisses (`sharex`) et ordonnées (`sharey`) doivent être les mêmes. Cela crée deux variables `fig` qui stocke la figure et `axes` qui contient les informations sur les axes dans un tableau de deux lignes et deux colonnes. Notez que dans la figure finale les abscisses/ordonnées ne sont graduées qu'une seule fois.
- ⊕ prend chacun des quatre axes repérés respectivement par les indices [0,0] jusqu'à [1,1] (la numérotation commence à 0) et :
  - \* trace un histogramme avec les données du département correspondant, avec les intervalles de `b` et avec des bords blancs avec `ec=`;
  - \* met un titre avec `set_title`;
- ⊕ fixe l'amplitude des abscisses pour la visualisation ;
- ⊕ met un titre général avec la fonction `suptitle` de la figure `fig` ;
- ⊕ affiche la figure 7.10.

## ?

### Exercice

Faites une figure avec 8 histogrammes sur une seule colonne et ajoutez pour chaque histogramme de la figure le nom des abscisses et des ordonnées avec `set_xlabel` et `set_ylabel`.

Pour la déclaration des 8 figures : `plt.subplots(8, 1)`.

Pensez à modifier la taille pour que ce soit harmonieux. Comme pour la fonction `set_title`, il faut prendre un des éléments (par exemple, première ligne première colonne avec `ax[0,0]`) et lui donner le texte à indiquer avec `ax[0,0].set_xlabel("Nombre d'habitants")`.

Pour mettre les mêmes axes d'ordonnées et d'abscisses, il est soit possible d'utiliser les méthodes `set_ylim` et `set_xlim` sur chaque graphique avec les mêmes valeurs. Il existe d'autres manières de faire : une option `sharex=True` et `sharey=True` de `subplots` permettent de forcer des échelles communes.

Si vous avez du mal avec le tableau à deux dimensions et les notations du type `ax[0,1]`, vous pouvez aplatiser le tableau. Par exemple, dans notre cas précédent, où le tableau avait six cases, il est possible d'utiliser la méthode `flatten` pour obtenir six variables différentes `ax1` à `ax6` stockant chacune un des graphiques. Vous pouvez alors soit récupérer chaque axe avec un nom avec `ax1, ax2, ax3, ax4, ax5, ax6 = ax.flatten()` soit les récupérer dans une liste avec `axes = ax.flatten()`.

## ?

### Exercice

**Utilisez une boucle pour faire le graphique à 6 histogrammes sans écrire à chaque fois le même code.**

Vous pouvez utiliser la méthode `ax.flatten()` pour obtenir une liste de tous les axes. Avec ceci vous devriez pouvoir faire une boucle sur plusieurs départements sans écrire le même code plusieurs fois :

```
fig,axes = plt.subplots(4, 3)

for i, ax in enumerate(axes.flatten(), start=1):
    # i:02 rajoute un zéro si besoin pour avoir deux chiffres
    ax.hist(donnees[donnees["DEP"]==f"{i:02}"]["P15_POP"], bins=20)
    ax.set_title(f"Histogramme département {i:02}")

fig.suptitle("Histogrammes par départements")
plt.show()
```

Il est aussi possible de créer un nouvel axe dans une figure déjà existante avec la fonction `nouveau_axe = plt.axes((position_gauche, position_bas, largeur, hauteur))`. Cela permet par exemple de rajouter un élément spécifique dans une figure complexe.

Dans certains cas, vous ne voulez pas séparer vos visualisations mais au contraire les *superposer* pour voir les différences entre plusieurs courbes, plusieurs histogrammes ou nuages de points. Pour reprendre l'exemple précédent sur les histogrammes : Est-ce que les départements ont des distributions comparables ? Pour répondre à cette question, nous allons tracer deux histogrammes sur le même graphique.

Il nous faut deux éléments de plus pour permettre une véritable comparaison : comme chaque département n'a pas le même nombre de villes, ni de la même taille, nous voulons un histogramme *normalisé* qui indique les proportions (ici rapportées à 1) avec un découpage similaire ; il faut mettre en transparence pour voir les différents histogrammes quand ceux-ci se chevauchent.

```
decoupage = range(0, 50000, 250)

# Création de la figure
fig,ax = plt.subplots(figsize=(10,5))
ax.hist(donnees[donnees["DEP"]=="67"]["P15_POP"], label='67',
        bins=decoupage, alpha=0.4, density=True)
ax.hist(donnees[donnees["DEP"]=="68"]["P15_POP"], label='68',
        bins=decoupage, alpha=0.2, density=True, hatch='//')
```

```
# Options et visualisation
ax.set_xlim(0,8000)
ax.set_title("Distribution des données pour deux départements")
ax.set_xlabel("Nombre d'habitants")
ax.set_ylabel("Proportion de villes (entre 0 et 1)")
plt.legend()
plt.show()
```

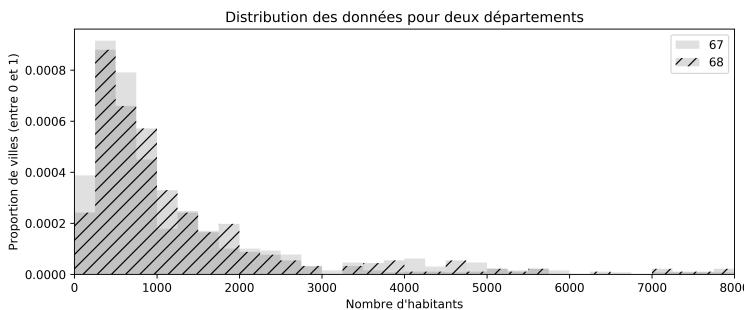


Fig. 7.11 – Exemple d'une superposition de graphiques

Ce code :

- ➊ définit avec la fonction `range` une liste d'éléments en précisant qu'ils doivent partir de 0, aller jusqu'à 50000 et augmenter de 1000 à chaque fois ;
- ➋ trace un histogramme, en précisant le découpage avec `bins=decoupage` défini plus haut, la transparence `alpha` et la normalisation avec `density=True` ;
- ➌ trace un deuxième histogramme avec les mêmes options qui vient se superposer car nous utilisons le même axe avec cette fois-ci des rayures avec `hatch='//'` ;
- ➍ fait un zoom sur le début de l'histogramme de 0 à 8000 ;
- ➎ met le titrage général et des axes, active la légende et affiche la figure 7.11.

Il est important de rappeler que les graphiques vont par défaut se superposer sur la même figure et que c'est à nous de faire attention à leur superposition correcte. Dans ce cas, en définissant le même découpage des histogrammes, nous nous assurons la superposition.

Un dernier exemple peut être intéressant, qui reprend le nuage de points précédent et rajoute un zoom sur une partie. L'idée est alors d'avoir deux figures, l'une étant un zoom de l'autre.

```
x = list(donnees["prop_f"])
y = list(donnees["prop_sup75"])
```

```

fig, (ax,zoom) = plt.subplots(1,2)

ax.set_title("Nuage de points")
fig.suptitle("Figure avec un zoom")
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion des plus de 75 ans (%)")

ax.scatter(x, y, s=5, alpha=0.1)
zoom.scatter(x, y, s=20, alpha=0.2, marker='+')

zoom.set_xlim(45, 55)
zoom.set_ylim(0, 40)

ax.indicate_inset_zoom(zoom, edgecolor='k')
plt.show()

```

Figure avec un zoom

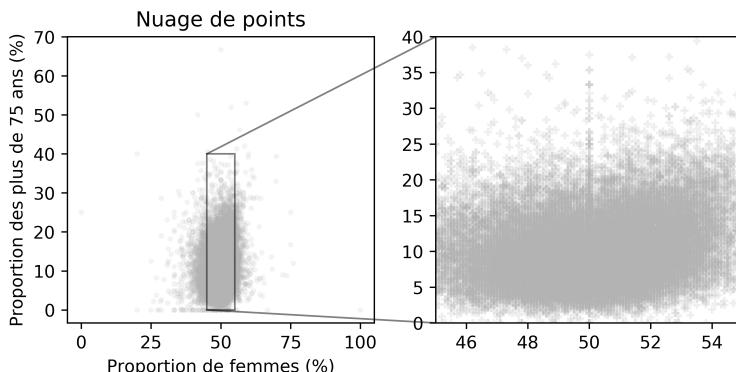


Fig. 7.12 – Figure composée d'un zoom dans un nuage de points

Ce code produisant la figure 7.12 ressemble à une figure composée comme précédemment, avec deux axes ici `ax` et `zoom`. La même figure est tracée sur les deux, avec un zoom sur la deuxième. La particularité est d'utiliser une fonction spécifique pour mettre en forme ce zoom `indicate_inset_zoom` qui est une option avancée de *Matplotlib*. Il en existe d'autres que nous vous laissons explorer au gré des exemples disponibles sur internet.

### 7.3.5 Couleurs et légendes

Une couleur est automatiquement attribuée aux visualisations. Dans certains cas, cela peut vous suffire. Dans d'autres, vous pouvez souhaiter la contrôler, en particulier pour des rendus finaux.

Définir une couleur n'est pas chose aisée<sup>7</sup>. Plusieurs stratégies existent : soit choisir des couleurs qui ont un nom, donc ont été codifiées, soit les décrire à partir de coordonnées. Le plus simple est d'utiliser un nom de couleur déjà défini. Certaines couleurs très utilisées ont un nom qui se résume à une lettre ("b" : bleu, "g" : vert, "r" : rouge, "c" : cyan, "m" : magenta, "y" : jaune, "k" : noir, "w" : blanc).

*Matplotlib* fonctionne aussi avec des thèmes et charge 10 couleurs allant de "C0" à "C9". Enfin, si vous souhaitez choisir une couleur particulière dans l'ensemble des variations imaginables, vous pouvez la décrire comme un élément de l'ensemble des variations par des coordonnées. La forme la plus courante est RGBA (pour *Red*, *Green*, *Blue* et *Alpha*) correspondant à des coordonnées (proportion de rouge, vert, bleu et transparence), soit résumé sous la forme hexadécimale<sup>8</sup>.

Une fois la couleur obtenue, il faut l'indiquer avec l'option dédiée dans la figure que vous tracez. Suivant la fonction, cette option s'appelle `c` (dans notre cas), `col` ou `color`. Reprenons le nuage de points précédent qui permet de voir sur un même graphique la proportion de femmes et la proportion d'habitants de plus de 75 ans avec un point par ville. Regardons les deux nuages de points pour deux départements, le 67 et le 13.

```
fig,ax = plt.subplots()

ax.scatter(donnees[donnees["DEP"]=="67"] ["prop_f"],
           donnees[donnees["DEP"]=="67"] ["prop_sup75"],
           label='67', s=20, c="C0", alpha=0.5, marker='^')

ax.scatter(donnees[donnees["DEP"]=="13"] ["prop_f"],
           donnees[donnees["DEP"]=="13"] ["prop_sup75"],
           label='13', s=20, c="C1", alpha=0.5, marker='o')

ax.set_title("Deux nuages de points")
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion de plus de 75 ans (%)")
plt.legend()
```

7. Mais c'est quoi une couleur ? Il existe plusieurs réponses à cette question suivant qu'on y répond en psychologue, en physicien ou en biologiste. Il y a une différence entre caractéristique de la lumière et perception par l'œil humain qui va dépendre de nombreux éléments.

8. Pour connaître le code correspondant à une couleur, utilisez un logiciel de dessin ou un site comme <https://www.color-hex.com/> qui vous permet de choisir une couleur et d'obtenir son code.

```
plt.show()
```

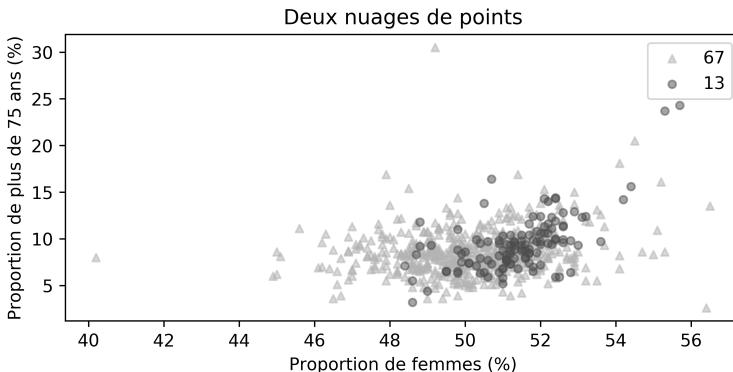


Fig. 7.13 – Exemple de représentation de deux nuages de points colorés

Ce code trace un premier nuage de points pour les données du département 67, avec la couleur par défaut `c="C0"` et celui du 13 avec la couleur `c="C1"`, puis rajoute un titre, une légende et le nom des axes pour produire la figure 7.13., nous utilisons aussi des formes de marqueurs différents pour la lisibilité.

Dans certains cas où des fonctions tracent directement plusieurs informations, vous pouvez donner non pas une couleur mais un ensemble de couleurs : les cartes de couleurs, ou *colormaps*. Des *colormaps* déjà existantes ont des noms et ont été conçues pour optimiser les représentations. Il existe deux types de *colormaps* : celles qui sont continues et varient d'une couleur à une autre, et celles séquentielles qui regroupent des couleurs différentes. Vous trouverez toutes les *colormaps* existantes dans la documentation en ligne de *Matplotlib*.

Enfin, avec plusieurs couleurs se pose la question des légendes. Nous avons déjà croisé la fonction `legend`. Il est possible de préciser sa position avec le paramètre `loc` (avec un numéro de 1 à 10, correspondant à différentes positions comme en haut à gauche ou en bas à droite, ou donnant les coordonnées précises). Nous retrouverons par la suite différents usages des couleurs et des légendes dans des exemples précis qui vous permettront de vous faire la main.

### 7.3.6 Sauvegarder une figure

Déjà rencontrée, la fonction `savefig` permet de sauvegarder la figure en cours dans un fichier. Pour cela il faut lui donner un chemin et un nom de fichier, dont l'extension indiquera à Python le format à respecter. Souvent nous utilisons

le format PNG (donc `.png`), mais cela peut être autre chose comme du JPEG (`.jpg`), les formats vectoriels comme postscript (`.eps`) et PDF (`.pdf`) sont aussi recommandés.

Pour augmenter la résolution, l'option `dpi` permet d'indiquer le nombre de points par pouces de l'image (*dot per inch*). Pour ajuster les bords blancs, vous pouvez utiliser l'option `bbox_inches=tight`. Une autre manière est d'utiliser `plt.tight_layout()` avant la sauvegarde de la figure.

Une sauvegarde de figure prend alors la forme suivante à la fin d'une visualisation : `plt.savefig("./ma-jolie-figure.png", dpi=200, bbox_inches='tight')`<sup>9</sup>.

### 7.3.7 Affiner une visualisation

Souvent, pour finaliser un graphique, vous avez envie de rajouter une information textuelle (par exemple, un mot près d'un point). La fonction `text` permet de rajouter du texte en précisant sa position. Si vous voulez juste rajouter un mot au niveau de l'abscisse 1 et de l'ordonnée 2, `plt.text(1, 2, "ajout d'un mot")`.

Vous pouvez encore mieux contrôler les annotations avec la fonction `annotate` qui permet de rajouter des flèches en précisant le point à annoter, l'endroit où le texte doit se trouver, et les propriétés de la flèche.

La fonction `hline` permet de tracer une ligne horizontale, tandis que `vline` permet de tracer une ligne verticale. Pour avoir une ligne horizontale blanche à hauteur de 1, il suffit d'écrire `plt.hline(1, c="w")`. Il est aussi possible de le faire sur un axe existant avec `ax.axvline` et `ax.axhline`.

La fonction `grid` permet de tracer une grille derrière les visualisations pour mieux lire les grandeurs.

Utilisons tout ça pour améliorer la présentation de l'histogramme de la proportion de femmes :

```
# Création de la figure
fig, ax = plt.subplots()
ax.hist(donnees["prop_f"].dropna(), bins=105, edgecolor='w',
        alpha=0.9, zorder=3, label='Découpage bins=105')

# Annotation et ligne supplémentaires
ax.axvline(donnees["prop_f"].mean(), linestyle='--',
            linewidth=2, zorder=4)
```

---

9. Il est souvent tentant de vouloir rajouter des détails en augmentant `figsize`, cependant ceci risque de rendre la police de caractère beaucoup plus petite à l'écran ainsi que de rendre les lignes plus fines. Dans ce cas, pensez à garder `figsize` constant et augmentez sa résolution avec `dpi`.

```

moyenne_f = round(donnees["prop_f"].mean(),2)
ax.annotate("Moyenne : {}".format(moyenne_f),
            (0.7, 0.5), xycoords='figure fraction')

# Options et visualisation
ax.grid(True)
ax.legend()
ax.set_xlim(40,60)
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Nombre de villes")
ax.set_title("Distribution de la proportion des femmes")
plt.show()

```

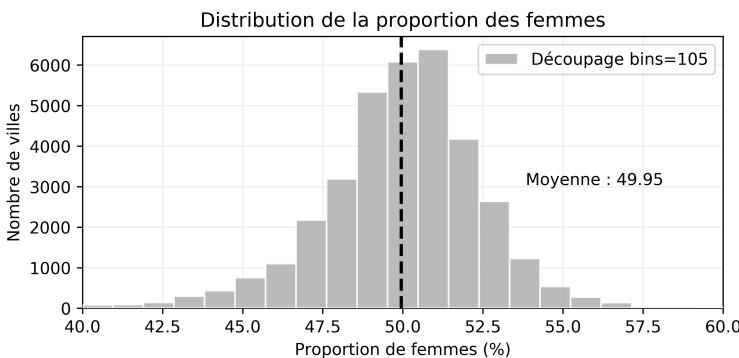


Fig. 7.14 – Exemple d'une figure complétée par différentes options

Ce code utilise les différentes fonctions vues précédemment pour tracer un histogramme découpé en 105 intervalles, une ligne verticale sur la moyenne et une annotation pour indiquer la moyenne sur le graphique 7.14. Remarquez ici l'utilisation de `zorder` pour choisir quelle distribution est au-dessus de l'autre (une figure sera devant l'autre si sa valeur est supérieure) et l'utilisation de `xycoords='figure fraction'` pour indiquer le positionnement de l'annotation avec des valeurs entre 0 et 1 plutôt qu'en coordonnées age/nombre de villes.

Nous pourrions encore aller plus loin en mettant une flèche sur l'annotation ou en modifiant les couleurs. Mais nous allons plutôt voir comment simplifier la visualisation avec des bibliothèques de plus haut niveau.

## 7.4 Visualiser avec *Pandas*

Maintenant que vous avez une idée assez précise de comment produire une visualisation avec les outils de base, vous allez à la fois apprécier la puissance de *Pandas* et savoir comment ajuster la visualisation.

### 7.4.1 Les différents types de visualisation

Nous avons déjà utilisé les possibilités de *Pandas* pour visualiser des données dans nos démarches exploratoires des chapitres précédents. Les tableaux *Pandas* permettent de produire des visualisations avec la fonction `plot` en précisant plusieurs options : le type de visualisation avec l'option `kind` ainsi que des paramètres comme la couleur. Les formats de visualisations directement disponibles sont les suivants, similaires à ceux déjà vus :

- ✚ `bar` pour un diagramme en barres ;
- ✚ `barh` pour un diagramme en barres horizontales ;
- ✚ `pie` pour un diagramme circulaire ;
- ✚ `hist` pour un histogramme, avec les mêmes options que *Matplotlib* ;
- ✚ `scatter` pour un nuage de points avec les options `x` et `y` pour préciser les colonnes à mettre en abscisses et ordonnées.

#### ➊ Exercice

**Tracez un nuage de points similaire à celui déjà vu précédemment.**

```
ax = donnees.plot(kind="scatter", x="prop_f", y="prop_sup75",
                   s=3, title="Nuage de points",
                   c="gray", edgecolor="gray", alpha=0.7)
```

Ce code permet de tracer un nuage de points en une ligne à partir du tableau en précisant les abscisses `x`, les ordonnées `y`, la taille des points avec `s`, la couleur avec `c`, la transparence avec `alpha`, et le titre.

Toutes ces visualisations avec *Pandas* sont des méthodes sur les tableaux de données et vont utiliser *Matplotlib* en arrière-fond pour créer concrètement l'image. *Pandas* fonctionne comme un intermédiaire pour éviter d'avoir à écrire tout le code.

### 7.4.2 Combiner *Pandas* et *Matplotlib*

Nous avons déjà utilisé des options de *Matplotlib* pour compléter une visualisation de *Pandas*. Il est ainsi possible de tracer une figure à partir d'un tableau *Pandas* et puis de l'ajuster au plus près pour obtenir ce que l'on souhaite. Voici quelques exemples de ce que vous souhaitez pouvoir faire :

- ⊕ zoomer sur un morceau de l'axe des abscisses en définissant un intervalle avec `plt.xlim` ou des ordonnées avec `plt.ylim`;
- ⊕ ajouter des titres, modifier les légendes ;
- ⊕ modifier le nom des graduations de l'axe des abscisses ou des ordonnées avec l'option `set_xticklabels` ou `set_yticklabels` des axes (ou plus généralement les positions d'abscisses avec `xticks` ou `yticks`), le terme `ticks` désigne les graduations ;
- ⊕ modifier la position de la légende.

Nous verrons par la suite des situations combinant une visualisation avec une bibliothèque avancée et des précisions avec *Matplotlib*.

### Exercice

**Modifiez les abscisses d'un diagramme en barres du nombre de villes par région.**

Nous aimerais plutôt les noms des régions que leurs numéros. Traçons donc le diagramme en barres puis changeons les labels des abscisses. La région identifiée par le numéro 44 est ici « Grand Est ».

```
distribution = donnees[\"REG\"].value_counts()
ax = distribution.plot(kind=\"bar\")
plt.xticks([0,1], [\"Grand Est\", \"Autre\"], rotation=70)
plt.show()
```

Ce code trace d'abord le diagramme en barres de la distribution des villes par région comme précédemment. Ensuite, il utilise la fonction `xticks` qui prend en argument la position des annotations, ici en 0 et 1, et l'annotation correspondante avec une rotation de 70 degrés. Comme la liste des graduations est trop courte par rapport aux graduations, beaucoup de barres n'ont pas d'annotations sur la figure.

### 7.4.3 Combiner tableaux et visualisations

Les tableaux permettent de faire de multiples opérations de transformation des données. Combinés avec les fonctions de visualisation, ils facilitent l'exploration.

Précédemment, nous avons vu que la fonction `groupby` permet de regrouper les données suivant les valeurs d'une colonne. Cela permet par exemple de regrouper par région et d'afficher la moyenne et la médiane d'habitants par ville pour avoir l'information directement disponible visuellement dans la figure 7.15 :

```
tab = donnees.groupby(\"REG\")[\"P15_POP\"].agg([\"mean\", \"median\"])
```

```
plt.show()
```

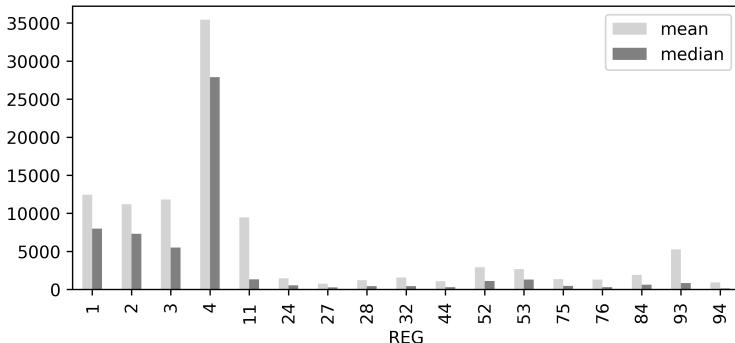


Fig. 7.15 – Exemple d'un diagramme en barres et groupby

Pour aller plus loin et voir la distribution d'une variable quantitative comme le nombre d'habitants des villes par rapport à des catégories définies dans une autre colonne du tableau, comme la région, il est possible de tracer un `boxplot`. Dans le cas précédent, cela se fait avec `donnees.boxplot(column = "P15_POP", by="REG")`. Ce code utilise la fonction `boxplot` qui prend comme argument la colonne des données numériques « P15\_POP » et la colonne des catégories à utiliser pour regrouper, ici « REG » donnée avec l'argument `by`.

Nous revoyons ici des graphiques déjà vus avec *Matplotlib*, et cela amène la question du choix. La réponse à une telle question ne peut pas être définitive : il s'agit de privilégier la méthode adaptée à votre pratique. Si vous utilisez déjà des tableaux *Pandas*, ce sera plus simple d'utiliser les visualisations associées. Par contre, si vos données sont dans un format plus basique, *Matplotlib* sera souvent plus adaptée. Le mieux est de connaître les deux possibilités et d'expérimenter.

## ?

### Exercice

**Pour une région donnée faites une analyse similaire par département.**

Pour choisir une région, vous pouvez faire une condition sur le tableau des données.

```
dpt67 = donnees[donnees["REF"]=="67"]
ax = dpt67.boxplot(column="P15_POP", by="DEP")
```

## 7.5 Utiliser la bibliothèque de visualisation avancée *Seaborn*

Nous avons vu les outils bas niveau avec *Matplotlib* et des outils haut niveau intégrés dans la bibliothèque *Pandas*. Faisons un petit tour maintenant des possibilités d'une bibliothèque dédiée à la visualisation de haut niveau permettant de construire des visualisations complexes. Nous ne pourrons épuiser les possibilités de *Seaborn* dans ce chapitre et nous avons retenu nos visualisations préférées.

### 7.5.1 Quelques visualisations utiles

#### Amélioration des visualisations habituelles

Tout d'abord, la bibliothèque permet de produire les visualisations habituelles de manière rapide et condensée. Commençons par charger la bibliothèque *Seaborn* avec `import seaborn as sns`.

La fonction `catplot` permet d'étendre les boîtes à moustaches pour représenter la distribution d'une variable numérique suivant des catégories. Pour représenter la distribution des proportions de femmes dans les départements de la région Grand Est, il est alors possible en une ligne d'obtenir les boîtes à moustaches (dont la longueur des traits est définie par défaut à 1.5 fois la largeur interquartile) dans la figure 7.16.

```
sns.catplot(x="DEP",y="prop_f", kind="box",
             data=donnees[donnees["REG"]==44])
```

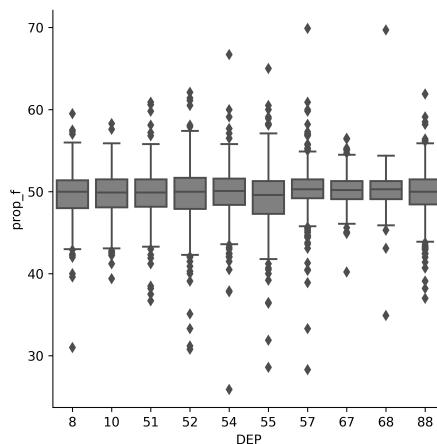


Fig. 7.16 – Exemple d'un boxplot avec Seaborn

Pour configurer plus finement la visualisation, vous pouvez alors jouer sur les couleurs ou sur la superposition de différents niveaux d'information. Par exemple, pour avoir une présentation horizontale plutôt que verticale, utilisez l'option `orient="h"`. Vous pouvez aussi regarder l'effet de l'option `kind="boxen"` ou `kind="violin"` qui changent la forme de la distribution.

Sans surprise, *Seaborn* facilite aussi les histogrammes avec la fonction `distplot`.

### ?

### Exercice

**Utilisez *Seaborn* pour produire un histogramme.**

```
plt.figure(figsize=(10,5))
sns.distplot(donnees["prop_f"].dropna(), bins=100)
plt.xlim(30,70)
```

Le code centre la figure sur l'intervalle 30-70. La distribution est complétée par une approximation en courbe de l'histogramme (il est possible de désactiver la courbe qui approxime l'histogramme avec `kde=False`).

*Seaborn* permet aussi d'améliorer les nuages de points en ajoutant les histogrammes de distribution avec la fonction `jointplot`.

### ?

### Exercice

**Tracez un nuage de points avec `jointplot` pour la proportion d'ouvriers et de la proportion de personnes âgées.**

```
sns.jointplot(x="prop_f", y="prop_sup75", data=donnees)
```

## Des nuages de points avec la droite de régression

Tracer un nuage de point c'est bien. Tracer directement en même temps la droite de régression c'est mieux. C'est ce que permet `lmplot`, que nous pouvons utiliser avec les données précédentes des proportions.

```
sns.lmplot(x="prop_f", y="prop_sup75", col="DEP",
            data=donnees[donnees["DEP"].isin(["67","68"])],
            sharex=True, sharey=True,
            scatter_kws={'alpha':0.8})
plt.show()
```

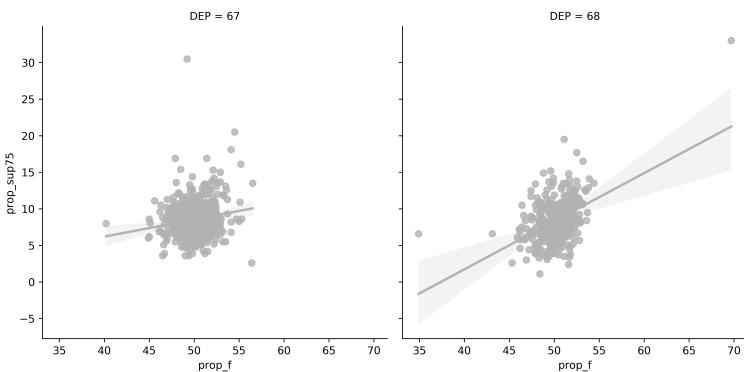


Fig. 7.17 – Exemple d'un nuage de points avec la droite de régression

Ce code utilise la fonction `lmplot` pour tracer le nuage de points avec la droite de régression complétée par la l'intervalle de confiance de la prédiction pour le jeu de données réduit aux départements 67 et 68, avec les mêmes abscisses et ordonnées. L'option `col` permet d'utiliser la colonne « DEP » pour séparer les deux graphiques dans la figure 7.17.

Dans de nombreux cas, vous pourriez avoir envie d'ajouter une troisième information à travers la couleur des points. L'option `hue` permet de décrire en plus de `x` et `y` une troisième variable qui sera représentée par une variation de couleur. Nous ne développerons pas cette option ici dans la mesure où le livre est en noir et blanc.

## Visualiser un tableau croisé

Une option intéressante pour voir rapidement les valeurs dans un tableau croisé est de colorer les cases. La fonction `heatmap` (carte des chaleurs) permet une telle visualisation. Prenons le tableau croisé entre les quartiles de la proportion d'ouvriers et la proportion de personnes âgées.

```
# Mise en forme des données
qs = [0,0.25,0.5,0.75,1]
labels = ["Q1", "Q2", "Q3", "Q4"]
quartiles_age = pd.qcut(donnees["prop_sup75"], qs, labels)
quartiles_fh = pd.qcut(donnees["prop_f"], qs, labels)
tableau_croise = pd.crosstab(quartiles_age,quartiles_fh)

# Création de la figure et affichage
plt.figure(figsize=(5,5))
sns.heatmap(tableau_croise, square=True)
```

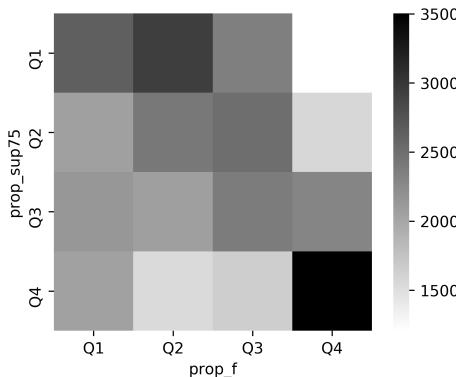


Fig. 7.18 – Exemple de visualisation d'un tableau croisé

Ce code :

- ➊ définit des paramètres ;
- ➋ construit deux variables `quartiles_age` et `quartiles_fh` à partir des quartiles des informations sur la proportion des personnes de plus de 75 ans et la proportion des femmes par ville, avec la fonction `qcut` de *Pandas* ;
- ➌ construit le tableau croisé `tableau_croise` à partir de ces deux variables catégorielles ;
- ➍ trace le tableau sous une forme colorée avec la fonction `heatmap` dans la figure 7.18.

### ➊ Exercice

Faites un tableau coloré avec le tableau en pourcentages par ligne.

Vous pouvez aussi :

- ➊ choisir le début et la fin de l'échelle des couleurs avec `vmin` et `vmax` ;
- ➋ centrer les couleurs autour d'une valeur avec `center` ;
- ➌ choisir les couleurs avec `cmap`, par exemple `cmap="YlGnBu"` (Yellow Green Blue) ;
- ➍ conserver l'information numérique des cases avec `annot=True` ;
- ➎ mettre des lignes blanches entre les cases avec `lineweights=.5`.

Nous n'avons pas épousé toutes les visualisations possibles. Nous ne saurions trop répéter de jeter un coup d'œil sur les catalogues d'exemples de visualisation pour ensuite vous demander quels sont vos besoins.

## 7.5.2 Stylisation des figures avec les thèmes

Une visualisation repose sur un ensemble de choix de couleurs, de police de caractères, de taille, de positionnement, d'espacement entre les graduations, de sens d'écriture... Le rendu final de votre visualisation et l'effet qu'elle peut produire vont donc être dépendant d'un ensemble de choix de style.

Le thème par défaut qui est utilisé par *Seaborn* fixe un style, mais vous pouvez choisir un des autres thèmes disponibles. Les thèmes par défaut sont : *darkgrid*, *whitegrid*, *dark*, *white*, et *ticks*. Chacun de ces thèmes produit un résultat différent, qui peut être plus ou moins adapté à votre usage.

Pour en sélectionner un, ajoutez au début de votre code la commande suivante :

```
sns.set_style("whitegrid")
```

Le style peut aussi être modifié en donnant les éléments de contexte généraux de la figure qui va influer sur la taille relative de l'écriture et du graphique. Quatre contextes existent : *paper*, *notebook*, *talk*, et *poster*. Leur utilisation se fait avec la fonction *set\_context*. Il est possible de changer d'autres paramètres comme la taille des polices directement avec l'option *font\_scale* :

```
sns.set_context("paper", font_scale=1)
```

## 7.6 Aller plus loin dans la maîtrise du visuel

Le contenu de ce chapitre vous permet déjà d'avoir des outils pour explorer et finaliser vos visualisations. Mais nous sommes loin d'avoir fait le tour de l'ensemble des possibilités. Plutôt que de rentrer dans des usages spécialisés, nous pensons plus important de revenir un peu sur la démarche générale d'abord d'un point de vue philosophique, puis pratique, pour mentionner les utilisations plus spécifiques.

### 7.6.1 Séparer exploration et finalisation

La visualisation est présente tout au long du traitement de données. Mais le moment d'exploration est très différent de celui de finalisation. L'exploration des données a pour principal objectif de vous familiariser avec les données et d'identifier des relations intéressantes. Pour cela, la rapidité de visualisation est plus importante que la finalisation ou que le choix de la taille des titres. Pensez alors à utiliser des visualisations simples à interpréter, en multipliant leur nombre si besoin pour explorer les différentes facettes de vos données. Et qu'importe si le choix des couleurs n'est pas tout à fait de bon goût.

Une exploration efficace des données repose sur deux règles principales : des données bien structurées et leur amélioration itérative. Prenez le temps avant de vous lancer dans l'analyse de construire un tableau *Pandas* bien organisé. Mais cela ne suffira pas. Au cours de votre exploration, vous aurez envie de recoder certaines variables. Acceptez que votre démarche est un processus continu.

La finalisation d'une visualisation signifie que vous souhaitez communiquer une information sous forme visuelle à un public extérieur. Dans ce cas, avant même de produire la visualisation, il est important que vous identifiez les éléments que celle-ci doit transmettre et la nature de ce public. Cela implique faire des choix (il est impossible de tout représenter) et de hiérarchiser les objectifs.

Ensuite, donnez-vous le temps de produire une visualisation satisfaisante. Consultez les exemples existants en ligne pour trouver des approches qui vous plaisent et jetez un coup d'œil au code. Une même information peut être codée visuellement de nombreuses manières. Faites plusieurs tentatives. Aussi, ne soyez pas captifs des outils : si vous avez envie de réaliser un certain effet, il est généralement possible de le faire même si ce n'est pas immédiat.

### 7.6.2 Exemple d'une figure complexe : le *bubble chart*

Un diagramme en bulles est une forme particulière de nuage de points où chaque point a une taille qui représente une information. Il peut être visuellement attrayant pour l'œil car il fait un peu *pop* et donc peut être une solution intéressante pour finaliser une analyse. Nous voulons par exemple rendre perceptible la diversité de deux départements, le 67 et le 68, en termes de villes sur la proportion de femmes et des personnes âgées.

Le cahier des charges est donc :

- ♣ représenter des points avec en abscisse la proportion de femmes et en ordonnées la proportion des plus de 75 ans ;
- ♣ avoir une couleur par département ;
- ♣ avoir un diagramme lisible.

Pour y répondre, nous allons combiner les fonctions des tableaux *Pandas* pour isoler l'information qui nous intéresse, puis construire une figure où nous allons représenter pour chaque département l'information souhaitée en changeant de couleur. Comme les points risquent de se superposer, nous allons les rendre en partie transparents et bien choisir leurs tailles.

Avant d'arriver au résultat final, nous faisons quelques tentatives : d'abord avec la fonction `scatter` de *Matplotlib*, puis avec la fonction `plot` de *Pandas*. Finalement, ce qui est le plus simple est de tracer un nuage de points par département en superposant tout sur le même graphique, en passant la liste des tailles de ville pour la taille des points.

Voici le résultat, qui est le produit de plusieurs tentatives (nous ne l'avons pas écrit d'un seul coup ! Jetez un coup d'œil à l'annexe sur les étapes invisibles du code) :

```
# Paramètres et données
colors = {"67": "C0", "68": "C1"}
donnees_filtrees = donnees[donnees["DEP"].isin(["67", "68"])]  
  

# Création de la figure
fig,ax = plt.subplots()
for dep_number,j in donnees_filtrees.groupby("DEP"):
    j.plot(kind="scatter",x="prop_f",
           y="prop_sup75",ax=ax,
           c=colors[dep_number],
           alpha=0.5,
           s=j["P15_POP"]/100, label=f"Departement {dep_number}")  
  

# Options et visualisation
ax.set_xlim(45,55)
ax.set_ylim(4,15)
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion >75 ans (%)")
ax.set_title("Diagramme en bulles (département 67 et 68)")
ax.legend(loc=2, markerscale=1/6, scatterpoints=2 )
plt.tight_layout()
plt.savefig("basrhin-bubblechart.png")
```

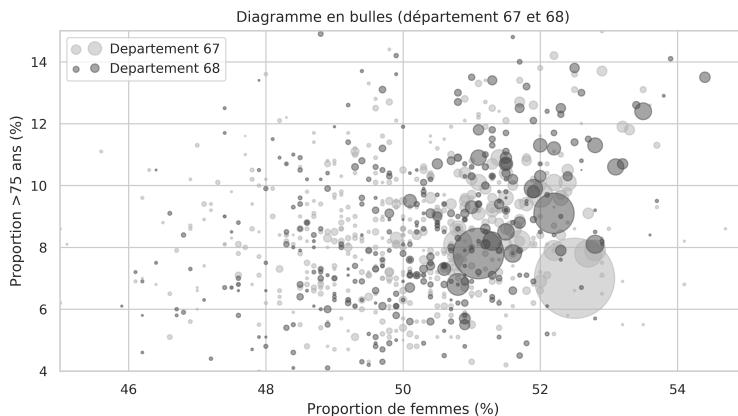


Fig. 7.19 – Exemple de représentation d'un diagramme en bulles

Ce code :

- ⊕ définit un dictionnaire qui associe chaque numéro de département à une couleur ;
- ⊕ filtre uniquement les départements 67 et 68 avec `isin` ;
- ⊕ crée une figure en lui donnant sa taille avec la fonction `subplots` pour récupérer l'axe `ax` et superposer les figures ;
- ⊕ groupe le tableau par département et fait une boucle sur ces groupes avec `dep_number` le numéro du département et `j` les données uniquement pour ce département :
  - \* trace un nuage de points pour chacun de ces sous-ensembles en définissant la couleur `c` à partir du dictionnaire des couleurs et la taille à partir de la variable sur la taille de la population, divisée par 100 pour que le rendu soit joli ;
- ⊕ fait les mises en forme pour que le rendu soit lisible et agréable à lire ;
- ⊕ ajoute une légende, trace la figure 7.19 et la sauvegarde dans un fichier.

### ?

### Exercice

**Rajouter une annotation pour mettre le nom des deux plus grandes villes sur le diagramme.**

Utilisez la fonction `annotate` vue précédemment.

## 7.6.3 Autres visualisations

### Les cartes

Les cartes représentent un domaine spécifique de la visualisation : appartenant à la grande famille des représentations visuelles des données, elles font l'objet d'une réflexion dédiée en géographie, discipline qui a développé ses propres outils et approches. Il est tout à fait possible de produire des cartes avec Python. Toutefois, et comme pour les statistiques, la question se pose d'utiliser Python ou un logiciel dédié. Suivant vos usages, nous vous conseillons de jeter un coup d'œil sur les outils existants en géographie dont une grande partie sont libres<sup>10</sup>.

Dans quelle situation est-il pertinent d'utiliser directement la programmation Python pour faire une carte ? Nous pensons que trois grandes raisons peuvent vous amener à privilégier cette approche :

- ⊕ vous avez besoin de générer des cartes avec des données qui peuvent changer et vous n'avez pas envie de refaire les manipulations ;

<sup>10</sup>. Deux logiciels en particulier peuvent répondre à vos besoins : Philcarto et QGIS. Ce dernier nécessite un apprentissage mais peut vous faciliter la vie ensuite.

- ⊕ vous avez besoin de mélanger plusieurs formes de données et de visualisation qui ne sont pas toutes de nature « géographique » ;
- ⊕ vous avez déjà tout votre traitement de données avec Python et vous voulez que ce soit cohérent ;

Vous ne serez pas étonné d'apprendre qu'il existe des approches bas niveau et haut niveau pour construire une carte. Au final, avec des points et des traits, il est possible de reproduire n'importe quelle image. Vous pouvez donc utiliser *Matplotlib* pour afficher des fonds de carte, et placer des points dessus comme un nuage de points. Par exemple, si vous avez récupéré des coordonnées GPS, vous pouvez utiliser la fonction `scatter` pour placer des points.

Pour le traitement de haut niveau, il existe un alter-ego de *Pandas* qui s'appelle *GeoPandas* dédié au traitement et à la visualisation de données géographiques. En particulier, *GeoPandas* permet de facilement lire les fichiers des cartes comme les *shape files* et les afficher. Le chapitre sur les usages avancés aborde l'utilisation de *GeoPandas* pour construire une carte avec les données de l'Insee.

## Les figures interactives

Que ce soit pour les intégrer à un site internet ou pour faciliter leur exploration, vous pourriez avoir envie de visualisations interactives. Par visualisation interactive nous parlons de visualisation qui réagit aux actions de l'utilisateur. Par exemple, un diagramme en barres qui affiche les valeurs au survol de la souris ou des diagrammes liés avec une sélection conjointe de points sur plusieurs visualisations.

Plusieurs solutions existent. La bibliothèque *Plotly* donne la possibilité de faire des visualisations de haut niveau. Plus simple que *Matplotlib* dont nous avons beaucoup parlé, cette bibliothèque est développée par une entreprise mais son usage est libre. Vous pouvez avoir une idée des résultats produits à partir des exemples en ligne et le code associé. Une autre bibliothèque libre permettant d'obtenir des figures interactives est *Bokeh*. Plus orientée vers la construction de figures pour des interfaces web, elle permet aussi d'être utilisée pour produire des rendus. Vous trouverez de nombreux exemples d'utilisation sur le site de la bibliothèque.

Mentionnons aussi l'existence de *Bqplot* avec une interface compatible avec *Matplotlib* et *Altair* comme bibliothèque de haut niveau avec des possibilités d'interactivité. Comme pour le choix des couleurs, du type de graphique et de tout autre aspect d'une visualisation, l'ajout d'interactivité doit se faire de manière parcimonieuse car elle peut devenir envahissante dans la lecture des informations.

## 7.7 Synthèse du chapitre

Dans ce chapitre, nous avons exploré les différentes stratégies de visualisation. La philosophie importante à retenir est qu'il existe des outils haut niveau et bas niveau. Comme toutes ces approches reposent sur *Matplotlib*, elles sont compatibles

et cohérentes. Dans l'idéal, maîtriser les deux niveaux ensemble est préférable car les outils de haut niveau permettent d'obtenir rapidement des visualisations complexes difficiles à construire avec les outils bas niveau tandis que ces derniers permettent les ajustements fins.

Concrètement, il est difficile de retenir toutes les visualisations possibles. Nous vous conseillons d'utiliser des visualisations simples (le diagramme en barres, l'histogramme) pour l'exploration, et de prendre le temps de réfléchir au type de résultats que vous voulez produire pour y aller étape par étape. Construire une visualisation de qualité peut prendre un peu de temps, ce qui est tout à fait normal.

# Chapitre 8

## Statistiques avancées

Ce chapitre présente des traitements statistiques avancés avec Python. Il aborde en particulier les analyses factorielles, les modèles de régression et les classifications.

### 8.1 Les traitements avancés en Python

Les traitements statistiques, entendus au sens large, regroupent de nombreuses traditions. Au moins trois approches peuvent être identifiées<sup>1</sup> :

- ➊ une approche *exploratoire et structurelle* des données : les outils, comme les analyses factorielles ou les classifications, permettent de décrire un ensemble de données, s'y repérer et situer la relation entre les entités étudiées sans nécessairement faire des hypothèses sur la causalité à l'œuvre ;
- ➋ une approche de *modélisation explicative* : les modèles de régression visent à capturer des relations avec pour objectif l'explication et l'interprétation de leurs paramètres ;
- ➌ une approche *prédictive* : les modèles servent en tant qu'outils pour prédire ou réaliser une action, ils n'ont pas vocation à rendre compte de la réalité mais d'être efficace.

Ces perspectives engagent des considérations philosophiques et épistémologiques autour de la notion de modèle qui conditionnent non seulement l'interprétation des résultats mais aussi les contextes dans lesquels ils vont être utilisés. L'interprétation dépend aussi beaucoup des questions abordées. Analyser un corpus de

---

1. Nous vous invitons à revenir à un manuel de statistiques dédié pour rentrer dans les détails des méthodes présentées si vous ne les connaissez pas. En effet, chacune se rattache à une tradition méthodologique avec ses règles. De plus, suivant votre spécialité, les utilisations peuvent varier.

textes pour extraire des thématiques pose des questions radicalement différentes de s'intéresser à la mobilité sociale dans la société française ou les phénomènes de clusters dans une épidémie.

Ce chapitre a pour but de montrer concrètement comment mettre en œuvre ces traitements dans la perspective des SHS. En effet, un même outil statistique peut être utilisé de différentes manières. Les outils existants en Python pour les statistiques avancées sont rarement dédiés aux SHS et leurs résultats peuvent ne pas être facilement interprétables à premier abord, contrairement aux logiciels dédiés. Nous vous montrons comment réaliser ces traitements en insistant sur leur interprétation. La contrepartie à cet effort supplémentaire nécessaire pour mettre en forme les résultats est une plus grande flexibilité<sup>2</sup>.

## 8.2 Les analyses factorielles

Les analyses factorielles visent à rendre compte de la structure des données. Elles sont utilisées différemment suivant les disciplines et le type de données, mais partagent néanmoins des similarités. Revenons d'abord sur la logique sous-jacente aux analyses factorielles avant de voir comment les réaliser.

### 8.2.1 Une grande famille

L'*analyse factorielle* (ou analyse en composantes) regroupe une famille d'outils avec plusieurs usages possibles. Alors que l'analyse en composantes principales est très fréquemment utilisée dans le domaine des sciences des données comme prétraitement pour simplifier des données numériques, les SHS sont davantage portées à utiliser cette analyse en composantes pour décrire les particularités du monde social, comme la proximité des individus entre eux compte tenu de leurs caractéristiques dans une perspective à visée explicative<sup>3</sup>.

#### ❶ Information

##### C'est quoi l'analyse factorielle ?

Pour le dire en quelques mots, chaque individu peut être associé à un point dans l'espace des variables. Ce point a des coordonnées (les modalités des variables). Deux points sont similaires s'ils ont des coordonnées proches, c'est-à-

---

2. Et puis, espérons que dans les prochaines années des bibliothèques dédiées aux SHS voient le jour et que la version 2 de ce manuel puisse être réduite.

3. Rappelons-le dès le début : donner un sens précis à des distances entre des points sur une visualisation des analyses factorielles est au minimum complexe et souvent assez faux. Les visualisations souvent utilisées permettent de rendre compte des tendances comme des regroupements de points, mais les interprétations plus au cas par cas doivent souvent s'appuyer sur des analyses complémentaires.

dire qu'ils ont globalement les mêmes modalités pour chaque variable. Comme l'espace est grand, c'est-à-dire composé de beaucoup de variables, l'idée est de chercher un plus petit espace qui conserve au mieux cette proximité entre les points. Cette étape est la projection qui vise à réduire l'espace initial sur un espace plus petit afin de diminuer le nombre de variables utilisées.

Pour pouvoir extraire de l'information pertinente, le traitement de l'analyse factorielle va combiner et transformer les variables initiales dans de nouvelles variables (les *facteurs*) qui ont la particularité d'être identifiées par leur plus ou moins grande importance dans leur capacité de résumer l'information générale. Ces nouvelles variables sont aussi appelées les *composantes* et facilitent des représentations graphiques en 2 ou 3 dimensions.

Suivant les données disponibles, la manière d'y parvenir va être différente, et trois procédures sont assez fréquemment rencontrées. L'*analyse en composantes principales* (ACP) est utilisée pour les données quantitatives. L'*analyse des correspondances multiples* (ACM) est adaptée pour les variables qualitatives. Enfin, les sociologues aiment bien, pour des raisons historiques et leur goût pour les tableaux croisés, les *analyses factorielles des correspondances* (AFC). Si ces approches ont des points communs, elles se distinguent cependant dans la manière d'interpréter les résultats.

Pour résumer :

- ✿ j'ai des données quantitatives et je voudrais les résumer en quelques indicateurs synthétiques ou voir à quel point elles sont différentes les unes d'entre elles : analyse en composante principale ;
- ✿ pareil, mais j'ai des données qualitatives : analyse des correspondances multiples ;
- ✿ je veux étudier la relation entre les modalités de variables dans un tableau croisé : analyse factorielle des correspondances.

Nous allons vous présenter comment réaliser ces analyses. Deux points sont importants à noter. D'une part, des étapes de préparation sont souvent nécessaires. Si cela peut vous paraître compliqué au début à retenir, ne vous inquiétez pas pour autant. Ces approches sont toujours les mêmes, et vous pouvez revenir aux exemples.

D'autre part, comme nous l'avons dit, la situation actuelle est que les bibliothèques Python actuelles ne sont pas pensées pour les SHS. Certains calculs ne sont donc pas automatiquement faits. Pour cette raison, nous présentons certaines opérations pour compléter les traitements réalisés par les bibliothèques existantes.

## 8.2.2 Analyse en composantes principales

Débutons par l'analyse la plus couramment rencontrée dans les manuels mais surtout utilisée par les disciplines ayant des données quantitatives : l'analyse en composantes principales.

En termes pratiques, une *Analyse en Composantes Principales* (ACP en français, et PCA en anglais pour *Principal Components Analysis*) est une opération qui prend en entrée un tableau numérique (lignes pour les individus, colonnes pour les variables) et calcule à la sortie un nouveau tableau de même taille, avec comme différence que les nouvelles colonnes (les facteurs ou composantes) sont classées de la plus importante (composante principale) à la moins importante. Chacune est associée à un poids différent définissant son importance dans la capacité à résumer les données initiales. Pour ces raisons, l'ACP est souvent utilisée pour ne garder que les composantes les plus importantes dans une logique de réduction des données pour ensuite construire des typologies ou réaliser des calculs<sup>4</sup>.

Pour faire ce traitement, nous allons utiliser la bibliothèque *Scikit-learn* qui regroupe tous les outils statistiques liés à l'apprentissage automatique, dont fait partie l'ACP. Elle a l'avantage d'être largement utilisée et maintenue.

Une étape intermédiaire est souvent nécessaire pour mettre les données en forme, autrement dit les *normaliser*. Cela signifie « centrer » les données et les « réduire » afin de rendre les colonnes comparables entre elles. En effet, comme chacune relève d'une variable différente, elles codent numériquement des informations de variances différentes<sup>5</sup>.

Nous allons réaliser une ACP sur l'ensemble des données numériques décrivant la population des villes sur les données déjà rencontrées. Précédemment, nous avons retenu un petit nombre de variables numériques qui caractérisent les villes afin de poser une question précise. Nous aurions aussi pu nous demander comment les villes se distinguent en termes de structure de population à partir de toutes les variables disponibles, sans faire de choix *a priori*. Nous allons donc utiliser l'ensemble des caractéristiques des villes et procéder à une ACP pour dégager les trois principaux facteurs.

```
# Chargement des bibliothèques
from sklearn.preprocessing import StandardScaler
```

---

4. Par exemple, un indicateur de défavorisation des villes peut être calculé en prenant le premier facteur d'une ACP sur un ensemble de variables socio-économiques (chômage, délinquance, accès à la propriété, imposition, etc.), voir par exemple Pernet *et al.* (2012).

5. Centrer signifie soustraire la moyenne de la colonne pour ramener chaque variable autour de zéro. Réduire signifie diviser par l'écart-type, ce qui permet de rendre les variations de chaque colonne comparables. Cependant, cela peut dépendre de ce que vous voulez obtenir comme résultats et de la nature de vos données, donc ne soyez pas étonné que dans certains cas cette étape n'ait pas lieu. Un des avantages de la programmation est de pouvoir contrôler chacune de ces étapes.

```

from sklearn.decomposition import PCA
import pandas as pd

# Chargement et mise en forme des données
tableau = pd.read_csv("./data/base-pop-2015-communes.csv")
tab_num = tableau.drop(["CODGEO", "REG",
                       "DEP", "LIBGEO", "P15_POP"], axis=1)
tab_norm = StandardScaler().fit_transform(tab_num)

# Application de l'ACP
pca = PCA(n_components=3)
decomposition = pca.fit_transform(tab_norm)

# Mise en forme et affichage
decomposition = pd.DataFrame(data = decomposition,
                               columns = ['facteur 1', 'facteur 2', 'facteur 3'])
decomposition.head()

```

	facteur 1	facteur 2	facteur 3
0	-0.677155	0.306759	0.383459
1	-1.146110	-1.271362	-0.034671
2	8.578975	0.284749	-4.194704
3	-0.242592	-0.786465	-0.241956
4	-1.222772	-1.264867	-0.054123

Ce code :

- ✿ importe deux fonctions de la bibliothèque Scikit-learn `StandardScaler` pour normaliser les données et `PCA` pour réaliser l'ACP ;
- ✿ construit un tableau `tab_num` contenant uniquement les données numériques enlevant les informations comme le département ;
- ✿ crée un objet de type `StandardScaler` permettant de normaliser et l'applique à nos données avec `fit_transform`, puis stocke le résultat dans la variable `tab_norm` ;
- ✿ crée un objet de type `PCA` qui est l'outil pour réaliser l'ACP en lui disant en option qu'on veut ne garder que trois dimensions ;
- ✿ applique l'outil `PCA` avec la fonction `fit_transform` sur le tableau puis stocke le résultat dans `decomposition` ;
- ✿ met en forme dans un tableau `Pandas` avec des noms de colonnes ;
- ✿ affiche le début du tableau.

L'ACP permet de récupérer les trois principaux facteurs de décomposition, chacun caractérisé par une valeur propre<sup>6</sup>. Chaque facteur représente une partie de la variance totale de l'information contenue dans les données, la *proportion de la variance expliquée*. L'idée est alors de garder uniquement les principaux facteurs pour résumer l'information.

Ces informations sont calculées lors de l'initialisation de l'ACP sur les données. Vous pouvez accéder aux valeurs singulières avec `singular_values_` (qui correspondent à la racine carrée des valeurs propres) et à la proportion de variance expliquée avec `explained_variance_ratio_`. Cela permet de se poser la question de la proportion totale d'information que l'on garde si on prend uniquement l'axe 1, ou l'axe 1 et l'axe 2, etc. Le code suivant affiche la proportion de variance expliquée par chaque composante.

```
print([round(i,2) for i in pca.explained_variance_ratio_])
```

```
[0.87, 0.04, 0.03]
```

Le premier facteur est le plus important, il correspond à 87% de la variance<sup>7</sup>. Dans notre cas, le premier axe s'impose et nous pouvons par commodité garder un deuxième.

Les facteurs sont des combinaisons des variables initiales. Vous pouvez accéder à leur composition avec `pca.components_` : une ligne correspond à un des nouveaux axes et chaque colonne a ses coordonnées dans les anciennes valeurs. Si la valeur est proche de 0 pour une case, c'est que le facteur ne dépend pas vraiment de la variable correspondante. Et inversement, si une valeur est importante, l'ancienne variable participe à définir le facteur.

## ?

### Exercice

**Calculez le pourcentage de variance capturée avec les deux premiers axes.**

```
100*pca.explained_variance_ratio_[0:2].sum()
```

---

6. Une valeur propre est une grandeur mathématique qui est associée à la structure du tableau. Si vos variables décrivent des valeurs différentes, un tableau avec N colonnes a généralement N valeurs propres, qui indique que ces N colonnes ne sont pas identiques et représentent des informations différentes. L'identification des valeurs propres permet de représenter la structure de ce tableau d'un point de vue mathématique. L'information donnée ici par la bibliothèque est les valeurs singulières.

7. La sélection du nombre de facteurs à garder se fait souvent à partir de cette distribution de valeurs propres, en regardant la « courbe des éboulis », c'est-à-dire l'évolution de ces valeurs, pour prendre toutes les valeurs avant le moment de cassure où ces valeurs commencent à beaucoup diminuer ou ne plus représenter beaucoup de la variance totale (le coude, même si cette méthode est aussi sujette à discussion).

Enfin, une fois la transformation effectuée, la visualisation aide souvent à voir comment les individus se répartissent. Nous utilisons la fonction `plot` vue dans le chapitre précédent pour présenter les deux premiers axes sur la figure 8.1.

```
ax = decomposition.plot(kind="scatter", color="gray", alpha=0.5,
                        x='facteur 1', y='facteur 2', marker = "v")
```

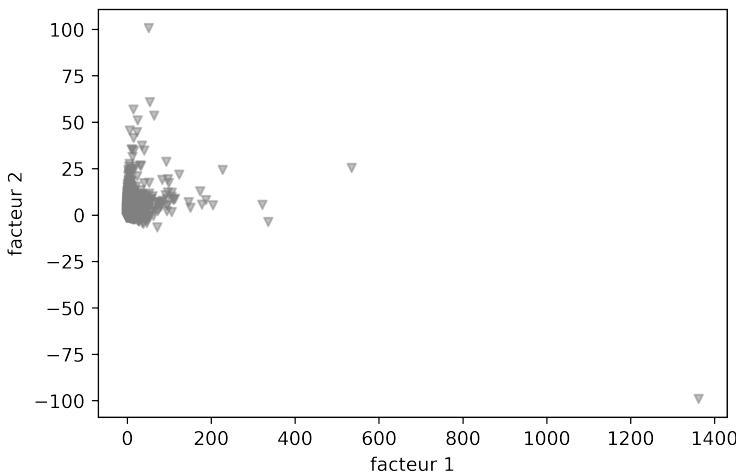


Fig. 8.1 – Représentation des deux premiers axes d'une ACP

Ce graphique n'est pas très beau car il y a une accumulation de points. On observe cependant des points extrêmes sur les deux axes, qui doivent largement participer à les définir.

Trouvons à quelles villes ils correspondent. On peut le faire facilement car l'ordre des lignes reste le même tout au long de la transformation et donc les tableaux `tableau` et `decomposition` partagent le même index. Il est donc possible de filtrer l'un avec l'autre.

```
tableau[decomposition["facteur 1"]>400]["LIBGEO"]
```

```
4385      Marseille
29631      Paris
Name: LIBGEO, dtype: object
```

### Exercice

## Quelles sont les villes les plus extrêmes sur l'axe 2 ? Et l'axe 3 ?

Pour cela, il suffit de filtrer le tableau de décomposition pour ne prendre que les lignes les plus extrêmes et voir à quel index elles correspondent.

La projection déforme les lignes initiales. Le problème est que *Scikit-learn* n'intègre pas les fonctions pour calculer ces indicateurs de déformation<sup>8</sup>.

### Information

#### Calcul de la qualité de projection

Pour interpréter, il est souvent utile d'avoir une information sur la déformation produite par la projection sur quelques axes. Il est possible d'analyser cette déformation (qu'est-ce que je perds de mes données initiales quand je ne regarde que les deux principaux facteurs ?) et les données qui contribuent le plus à rendre ces axes importants (à quelles données correspondent mes axes ?). Cela se fait avec les cosinus carrés mesurant la qualité de la projection des données. Une autre mesure utile est la contribution des individus aux facteurs, généralement appelée *CTR*.

Il est possible de faire les quelques lignes de Python pour calculer ces indicateurs. La qualité de la projection des données ( $\cos^2$ ) d'un point sur un des facteurs se calcule comme le carré de la coordonnée sur ce facteur divisé par la distance à l'origine du point ( $d^2$ ).

```
d2 = (tab_norm**2).sum(axis=1)
cos2 = decomposition**2
cos2["d2"] = d2
cos2 = cos2.apply(lambda x : x/x["d2"],axis=1)
cos2 = cos2.drop("d2",axis=1)
cos2.head()
```

	facteur 1	facteur 2	facteur 3
0	0.103090	0.021156	0.033058
1	0.220244	0.271013	0.000202

8. Les logiciels de statistiques calculent généralement automatiquement ces valeurs. À ce titre, ils sont plus simples à utiliser (comme la bibliothèque FactoMineR pour R). Python permet de suppléer à ces manques en permettant de construire ses propres outils. Dans ce cas, nous avons un bon exemple de la manière de procéder : identifier les besoins, trouver le calcul à réaliser, puis le coder. L'étape suivante, qui ne manquera pas d'arriver avec le développement de l'utilisation de Python, est que ces outils vont être intégrés automatiquement dans les prochaines versions de la bibliothèque. Cependant, cela n'enlèvera pas la logique qui est de pouvoir résoudre les limites que l'on rencontre.

```

2   0.708399  0.000780  0.169360
3   0.014261  0.149880  0.014186
4   0.244118  0.261215  0.000478

```

Ce code :

- ➊ calcule le carré de la longueur de chaque point et le stocke dans `d2`;
- ➋ crée une variable `cos2` avec les coordonnées au carré de chaque ligne;
- ➌ rajoute la colonne du carré de la distance à ce tableau nommé `d2` pour faciliter les calculs par la suite qui auront besoin de cette valeur;
- ➍ pour chaque ligne, divise chaque élément par cette valeur de la colonne « `d2` » en utilisant la méthode `apply`;
- ➎ supprime la colonne « `d2` » qui n'était qu'un intermédiaire du calcul (tous les éléments sont égaux à 1 maintenant);
- ➏ affiche les premières lignes avec la méthode `head`.

Nous avons maintenant la déformation de chaque point dans la projection. Pour la première composante, la quatrième ligne (index 3) est très peu projetée. Si on ne prend que cette composante, l'information de cette ligne sera finalement peu conservée. À l'inverse, le troisième élément (index 2) est bien conservé sur l'axe.

### ?

## Exercice

### Identifiez les villes les plus « déformées » par la projection.

Elles correspondent pour chaque composante aux lignes pour lesquelles le  $\cos^2$  est proche de zéro. Si la projection retient deux composantes, la qualité de projection est la somme des  $\cos^2$  de ces composantes, à comparer à 1.

### !

## Information

### Calculer la contribution des individus aux axes

La contribution des individus aux facteurs n'est pas non plus calculée automatiquement par la bibliothèque. Ce CTR permet de comprendre le lien entre un facteur et les données des lignes. Le coefficient pour un individu `i` pour le facteur `j` est calculé en divisant la coordonnée `(i,j)` de cet individu au carré divisé par la valeur propre du facteur correspondant multiplié par le nombre d'individus. Cela se calcule de la manière suivante, code que nous donnons pour information :

```

valeurs_propres = pca.singular_values_**2
n = len(decomposition)
ctr = decomposition**2

```

```

for col, val in zip(ctr.columns, valeurs_propres):
    ctr.loc[:,col]=ctr.loc[:,col]/(n*val)

ctr.columns = ["CTR facteur 1", "CTR facteur 2", "CTR facteur 3"]
ctr.head()

```

Dans de nombreux cas, l'étude nécessite de rajouter des données supplémentaires illustratives : celles-ci ne participent pas à produire les facteurs de la décomposition, mais à être comparées et situées. Cela peut être par exemple le cas d'informations permettant de contrôler l'interprétation. La méthode est alors simple : il suffit d'utiliser la méthode `transform` des deux objets `normaliser` et `pca` (à la place de `fit_transform` qui fait en même temps le calcul des paramètres de la transformation et la transformation elle-même) en donnant en entrée les nouvelles données.

## Exercice

### Rajoutez une nouvelle ville fictive dans la transformation.

Il faut générer une ligne de même taille que celles du tableau. Pour cela, on utilise les fonctions aléatoires déjà vues.

```

ville = [10000]+[random.randint(0,10000) for i in range(0,102)]
villes_supplémentaires = normaliser.transform([ville])
projection = pca.transform(tab_norm)

```

On construit une ligne de 103 valeurs, dont la première est le nombre total d'habitants et la suivante est une série aléatoire, puis la transformation et la projection se font avec les outils précédents.

Il existe d'autres bibliothèques qui facilitent certains traitements. Cependant, plus récentes et moins soutenues par la communauté, elles ont potentiellement une durée de vie plus courte. Par exemple, la bibliothèque *Prince* a une fonction PCA qui calcule directement la contribution des lignes. Pour refaire l'ACP précédente dans cette bibliothèque, le code est un peu différent :

```

import prince

# Application du modèle
pca = prince.PCA(n_components=3,
                  rescale_with_mean=True,
                  rescale_with_std=True)
pca = pca.fit(tab_num.values)
decomposition = pca.transform(tab_num.values)

```

```
# Affichage de paramètres
print([round(i,2) for i in pca.explained_inertia_])
```

[0.87, 0.04, 0.03]

Ce code importe la bibliothèque, définit la transformation en précisant qu'on veut normaliser les données (réduire/centrer), puis calcule l'analyse sur les données numériques du tableau `tab_num.values` avant d'afficher les paramètres correspondant à la variance expliquée par chacun des axes (pour les valeurs propres, ce serait : `pca.eigenvalues_`). Dans certains cas, cette bibliothèque sera plus facile à utiliser pour obtenir certaines informations : par exemple, la corrélation entre les composantes et les variables initiales peuvent être obtenues avec la fonction `pca.column_correlations(tableau_numerique.values)` et la contribution de chaque ligne par `pca.row_contributions(tableau_numerique.values)`. De même, une visualisation peut être facilement obtenue<sup>9</sup>.

### 8.2.3 Analyse des correspondances multiples

Le principe des *Analyse des correspondances multiples* (ACM) est similaire à celui de l'ACP à la différence que les variables sont catégorielles plutôt que numériques. Pour cette raison, cette analyse est particulièrement utilisée par les sociologues confrontés à des données qualitatives<sup>10</sup>. L'objectif est cependant le même : identifier les dimensions les plus importantes dans des données.

En pratique, vous avez un tableau d'individus (lignes) décrits par des variables qualitatives (colonnes). Comme il n'est pas possible de faire un traitement sur des catégories, ce tableau est transformé : chaque variable avec N modalités donne N colonnes, dont chaque case prend la valeur 0 si ce n'est pas la modalité de la ligne, ou 1 si c'est la modalité de la ligne. L'analyse en composante se fait alors sur ce nouveau tableau, appelé *tableau disjonctif complet*.

#### Information

##### Tableau disjonctif complet

Il s'agit d'un tableau où chaque variable catégorielle est séparée en N colonnes, N étant le nombre de modalités de cette variable, indiquant pour chaque ligne si cette modalité est présente (1) ou absente (0). Par exemple, une

9. Toutes les informations sont sur la documentation en ligne. Pour définir la visualisation des deux premiers axes : `ax = pca.plot_row_coordinates(X, figsize=(6, 6), x_component=0, y_component=1, ellipse_fill=True, show_points=True)` puis pour l'afficher `ax.get_figure().savefig('images/pca_row_coordinates.svg')`.

10. Pour une présentation de l'usage en sociologie, voir Renisio & Belin (2014).

variable V où les réponses sont Oui/Non donnera deux colonnes, « V-Oui » et « V-Non ». Il existe plusieurs fonctions pour obtenir facilement un tableau disjonctif complet à partir d'un tableau initial, en particulier une fonction *Pandas* qui prend le tableau X en argument : `pd.get_dummies(X)`. Nous utiliserons les deux méthodes dans les exemples.

Nous allons utiliser la bibliothèque *Prince* que nous venons de présenter pour faire ces analyses, car *Scikit-learn* ne fait pas ce type de traitement. Vous pouvez lui fournir soit le tableau disjonctif déjà calculé, soit le tableau initial (le tableau disjonctif sera alors calculé).

Réalisons une ACM sur les variables qualitatives que nous avons créées sur les villes lors de notre recodage en catégories, en particulier « prop\_f\_C », « prop\_sup75\_C », « prop\_ouvriers\_C » et « P\_POP15\_C ». Nous allons nous concentrer sur un département, le Bas-Rhin, qui a pour numéro 67. Nous utiliserons les données de `./data/data-chap6.csv` qui contient en plus des données brutes les variables calculées dans le chapitre précédent.

```
import prince

# Chargement et mise en forme des données
tableau = pd.read_csv("./data/data-chap6.csv")
colonnes = ["P15_POP_C", "prop_f_C",
            "prop_sup75_C", "prop_ouvriers_C"]
X = tableau[tableau["DEP"]=="67"][colonnes]

# Application de l'ACM
acm = prince.MCA(n_components=2)
acm = acm.fit(X)

# Affichage
print([round(i,3) for i in acm.explained_inertia_])
```

[0.134, 0.113]

Ce code :

- ➊ importe la bibliothèque *Prince*;
- ➋ crée un tableau X ne gardant que les variables qualitatives d'intérêt ;
- ➌ crée un objet *MCA* à partir de la bibliothèque en précisant le nombre de composantes à garder (d'autres paramètres peuvent être rajoutés) ;
- ➍ applique cet objet sur les données pour adapter le modèle ;
- ➎ affiche la proportion de variance expliquée en arrondissant.

Les valeurs propres sont obtenues avec `acm.eigenvalues_`. Il est aussi possible de récupérer les coordonnées des lignes (et des colonnes) avec les deux méthodes `mca.row_coordinates(X)` et `mca.column_coordinates(X)`.

La bibliothèque permet aussi facilement de tracer la figure 8.2<sup>11</sup>.

```
X.columns = [i.replace("_","") for i in X.columns]
ax = acm.plot_coordinates(X=X, figsize=(4, 4),
                           row_points_size=10,
                           show_row_labels=False,
                           show_column_points=False,
                           column_points_size=30,
                           show_column_labels=False)
ax.get_figure().savefig('mca.svg')
```

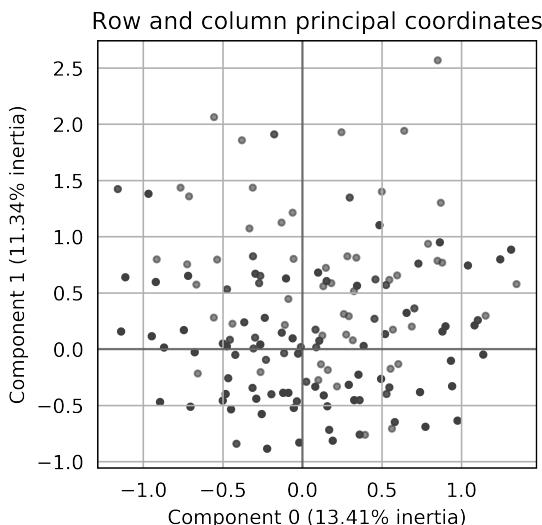


Fig. 8.2 – Visualisation des deux premiers axes de l'ACM

Ce code :

- ➊ change le nom des colonnes pour éviter un problème de légende sur la figure ;
- ➋ affiche la projection des individus (ou des lignes) avec différentes options ;
- ➌ sauvegarde la figure.

11. Remarque : il faut enlever le signe tiret-bas des colonnes, car sinon ça génère un problème et les labels ne s'affichent pas. Nous nous en sommes rendus compte en essayant et en comparant notre résultat aux exemples de la documentation. C'est typiquement un bug qui sera probablement bientôt corrigé, nous le mentionnons pour illustration.

## ?

### Exercice

Rajoutez les colonnes et les labels.

Modifiez les options de `False` à `True`

Cette bibliothèque permet en quelques lignes d'obtenir une analyse et par ailleurs de récupérer les projections pour des traitements ultérieurs.

## ?

### Exercice

Quelles sont les villes les plus extrêmes sur les deux axes ?

Récupérez les coordonnées avec `mca.row_coordinates(X)` et filtrez les villes les plus extrêmes.

Comme pour l'ACP, le  $\cos^2$  et le  $CTR$  ne sont pas directement calculés. À vous de les calculer, sachant que la méthode est la même que pour les ACP. Nous mettons le code ci-dessous si nécessaire.

## !

### Information

Calculer le  $\cos^2$  de l'ACM avec *Prince*

```
mca_prince = prince.MCA(n_components=12)
mca_prince.fit(X)
T = mca_prince.row_coordinates(X)
d2 = (T**2).sum(axis=1)
intermediaire = (T**2)
intermediaire["d2"] = d2
cos2 = intermediaire.apply(lambda x : x/x["d2"],axis=1)
cos2 = cos2.drop("d2",axis=1)
```

Ce code :

- ➊ fait une décomposition complète du tableau avec *Prince* (autant de composantes que la dimension initiale du tableau qui est le nombre total de modalités moins le nombre de variables) et stocke la projection dans la variable T ;
- ➋ calcule à partir de cette décomposition la distance au carré de chaque vecteur dans d2 ;
- ➌ construit un tableau intermédiaire qui met au carré les composantes de la décomposition, rajoute la distance carrée en colonne ;

- ✚ construit le tableau des  $\cos^2$  en divisant chaque ligne par la distance carrée de la ligne ;
- ✚ supprime la dernière ligne intermédiaire.

### 8.2.4 Analyse factorielle des correspondances

L'*analyse factorielle des correspondances* (AFC) est un cas particulier de la décomposition en composantes dans le cas où le tableau des données est un tableau croisé. Largement popularisées par les travaux de Pierre Bourdieu et de Jean-Paul Benzécri Benzécri *et al.* (1973), les AFC utilisent avec efficacité la symétrie présente dans un tableau croisé pour analyser dans un même mouvement les modalités de deux variables afin de faire ressortir la structure des affinités du tableau. Pour cette raison, la représentation la plus popularisée est un graphique à deux dimensions avec à la fois les modalités des deux variables.

La bibliothèque *Prince* permet aussi de faire ce type de traitement<sup>12</sup>.

L'AFC s'applique sur des tableaux croisés. Nous allons prendre pour exemple le tableau croisé entre les départements et les proportions de femmes par ville.

```
import prince

# Chargement et mise en forme des données
X = pd.crosstab(tableau["DEP"], tableau["prop_f_C"])
X.columns.rename('Quartiles proportion femmes', inplace=True)
X.index.rename('Départements', inplace=True)

# Application de l'AFC
afc = prince.CA(n_components=2)
afc = afc.fit(X)

# Affichage
afc = afc.plot_coordinates(X=X, figsize=(5, 5),
                            show_row_labels=False,
                            x_component=0, y_component=1)
```

---

12. Peu de bibliothèques permettent de faire ce traitement. Celles qui existent deviennent vite obsolètes car elles ne sont pas maintenues. Une bibliothèque française *fanalysis* permet de faire les traitements en composantes principales et des correspondances multiples que nous venons de voir, ainsi que les AFC, mais elle n'est plus maintenue et se retrouve incompatible avec les évolutions récentes de Python.

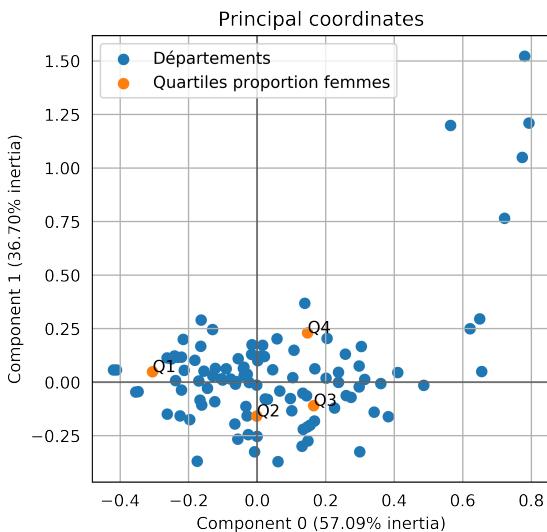


Fig. 8.3 – Visualisation des deux premiers axes de l’AFC

Ce code :

- ➊ importe la bibliothèque ;
- ➋ crée le tableau croisé entre les départements de chaque ville et la proportion de femmes en quartiles, renomme le nom général ligne/colonne ;
- ➌ construit l’objet CA avec le nombre de composantes à garder ;
- ➍ adapte le modèle aux données avec la méthode `fit` ;
- ➎ trace le résultat de cette transformation avec à la fois les lignes et colonnes dans la représentation 8.3.

Le graphique rend visible alors la proximité de chaque département par rapport aux quartiles de la proportion de femmes, calculé à partir de sa distribution de villes. Les points les plus à droite sont alors les départements qui comprennent des villes ayant une proportion davantage élevée de femmes.

De la même manière que l’ACM vue précédemment, il est possible d’obtenir les différentes informations sur les données transformées : les valeurs propres de chaque axe avec `afc.eigenvalues_` et la proportion d’inertie expliquée avec `afc.explained_inertia_`. Comme pour les cas précédents, certains paramètres ne sont pas calculés. Si vous en avez besoin, vous devez les calculer à partir des données de la projection avec une logique similaire à celle présentée pour l’ACM.

Cette partie a présenté différents outils d’analyse factorielle. Il en existe cependant d’autres. Si vous êtes amené à rencontrer d’autres formes de projection, pensez à faire une petite recherche car les outils peuvent évoluer et certaines opérations

difficiles à réaliser avec certaines bibliothèques prennent une ligne dans d'autres. Vous êtes cependant équipé pour traiter vos données. Passons maintenant à une autre manière d'analyser les relations dans des données : les modèles de régression.

## 8.3 Les modèles de régression

Les modèles de régression représentent un traitement statistique largement formalisé. Ils sont facilement mis en œuvre avec les bibliothèques Python.

### 8.3.1 Quelques éléments sur la modélisation

L'idée générale d'une régression est de faire correspondre au mieux un modèle<sup>13</sup> aux données en calculant des paramètres qui permettent cet ajustement.

Ces modèles peuvent être utilisés pour étudier la relation dans des données ou prédire de nouvelles valeurs. En SHS, ces modèles sont généralement utilisés pour montrer l'existence de relations dans les données et quantifier la force de ces relations, en intégrant des formes de dépendances plus complexes que juste la corrélation.

La modélisation distingue :

- ✚ la variable *dépendante* (ou à expliquer) que l'on cherche à relier à un ensemble d'autres variables ;
- ✚ les variables *indépendantes* (ou explicatives), qui sont susceptibles d'agir sur la variable dépendante ;
- ✚ le type de *modèle* retenu pour représenter les données ;
- ✚ les *données* qui permettent d'estimer les coefficients du modèle.

Il y a donc souvent trois étapes après l'identification des données : la première est de sélectionner un modèle adapté, la deuxième est d'estimer les paramètres du modèle, la troisième est de vérifier que ces paramètres et le modèle en général ont un sens. Souvent, le choix du modèle se fait sur la base des habitudes ou des modèles disponibles. On parle de modélisation statistique pour souligner qu'on accepte qu'il existe une part d'aléatoire. Ce faisant, un enjeu est non seulement de mesurer la distance entre le modèle et les données réelles, mais aussi de mesurer l'effet de l'aléatoire.

D'un point de vue pratique, nous allons utiliser une nouvelle bibliothèque *Statsmodels*. En effet, la bibliothèque *Scikit-learn* dont nous avons déjà parlé et que vous rencontrerez souvent est plutôt orientée vers la prédiction et *l'apprentissage automatique*, pouvant être un peu plus aride pour une utilisation en SHS davantage orientée vers la description des relations dans les données.

---

13. Un modèle correspond à une forme, comme une droite, qui relie les variables.

### 8.3.2 Régressions linéaires

La régression linéaire signifie que l'on fait l'hypothèse que la variable dépendante est reliée aux variables indépendantes par une relation linéaire (une droite s'il y a une seule variable indépendante). L'objectif est alors de calculer au mieux les paramètres qui relient des variables indépendantes  $X_1, X_2, \dots, X_N$  à la variation d'une variable dépendante Y, en faisant l'hypothèse que cette variation est linéaire (si  $X_1$  augmente, Y doit varier linéairement aussi). S'il y a une seule variable X, on parle de régression simple et multiple s'il y en a plusieurs.

Si  $X_1, X_2, \dots, X_N$  sont les variables indépendantes, et Y la variable dépendante, la régression linéaire des paramètres  $a_1, a_2, \dots, a_N$  pour que la relation  $Y = a_0 + a_1.X_1 + a_2.X_2 + \dots + a_N.X_N$  s'adapte au mieux aux données que l'on a.

Attention, « au mieux » ne signifie pas « bien » et encore moins « parfaitement » : souvent, des modèles sont utilisés sans aucun rapport avec la variation réelle des données. Le résultat d'une régression linéaire est donc l'estimation des paramètres liés à chaque variable et la capacité du modèle de bien représenter les données.

Pour l'exemple, nous allons faire une régression de la proportion de femmes par rapport aux variables d'intérêt. L'idée est similaire aux cas déjà rencontrés : il existe un objet régression linéaire défini dans une bibliothèque, on renseigne les paramètres et il calcule la régression.

```
import statsmodels.api as sm

# Chargement et mise en forme des données
donnees = tableau[["prop_ouvriers", "prop_sup75",
                   "P15_POP", "prop_f"]].dropna()
X = donnees[["prop_ouvriers", "prop_sup75", "P15_POP"]]
Y = donnees["prop_f"]

# Appliquer la régression
modele = sm.OLS(Y, X)
regression = modele.fit()

# Affichage
print('Paramètres: ', regression.params)
print('R2: ', round(regression.rsquared, 2))
```

```
Paramètres: prop_ouvriers    1.761921
prop_sup75      2.396768
P15_POP        0.000093
dtype: float64
R2:  0.91
```

Ce code :

- ⊕ charge le module `api` de la bibliothèque `statsmodels` ;
- ⊕ enlève les lignes ayant au moins une valeur manquante ;
- ⊕ définit le tableau des variables indépendantes `X` ;
- ⊕ définit la variable dépendante `Y` ;
- ⊕ construit la régression avec la fonction `OLS` en donnant directement les données dépendantes et indépendantes à part. Le modèle est stocké dans la variable `modele` ;
- ⊕ ajuste le modèle aux données avec la fonction `fit` ;
- ⊕ affiche les paramètres de la régression.

Vous pouvez avoir la présentation de l'ensemble des éléments du modèle de régression avec la méthode `summary`.

### ?

#### Exercice

Affichez l'ensemble des résultats de la régression.

```
print(regression.summary())
```

Le nom de la régression est OLS, pour *Ordinary Least Square*, le nom générique de ce type de régression. Cette régression linéaire semble montrer que l'âge a une influence importante, en accord avec notre hypothèse initiale d'une longévité différenciée entre homme et femme. La taille des villes n'apparaît pas avoir un poids important, mais il faut avoir en tête que la variation du nombre d'habitants peut être importante donc un paramètre faible va être multiplié par un effectif important<sup>14</sup>.

### ?

#### Exercice

Retirez l'information sur la taille des villes et regardez comment les paramètres changent.

Il suffit de modifier le tableau des variables indépendantes

```
X = tableau[["prop_ouvriers", "prop_sup75"]].dropna()
```

Le modèle est caractérisé par son  $R^2$  qui donne une information sur l'adéquation du modèle aux données des variables dépendantes (la variance expliquée). Cette valeur est directement accessible avec la variable `regression.rsquared`. Il est possible d'obtenir d'autres informations du modèle. Par exemple, pour juste avoir les paramètres afin de tracer la droite de régression, vous pouvez récupérer les paramètres avec : `regression.params`.

14. L'étape ensuite est de vérifier la significativité statistique de ces paramètres. Vous pouvez avoir les *p values* de la régression avec `regression.pvalues()` et différentes fonctions permettent de faire des tests d'hypothèses sur les paramètres.

### 8.3.3 Régressions logistiques

La régression logistique est une régression qui utilise un modèle dit *logistique* (une courbe en forme de S) de la même manière que la régression linéaire utilise un modèle *linéaire* (une droite). La différence fondamentale est que la régression porte non pas directement sur la variable dépendante elle-même (sa valeur) mais sur la probabilité d'avoir l'une ou l'autre de ses valeurs par rapport aux variables indépendantes.

La plus connue est la régression logistique binaire qui permet de modéliser une variable avec deux modalités (Oui/Non, Présent/Absent, 1/0, etc.) et de s'interroger sur l'effet propre d'un ensemble de variables indépendantes. Les variables indépendantes peuvent être indifféremment catégorielles ou quantitatives. Comme pour la régression linéaire, l'objectif est de trouver les coefficients de ces variables indépendantes qui permettent d'ajuster au mieux le modèle aux données que nous avons<sup>15</sup>.

#### Information

#### Les deux usages de la régression logistique en SHS

Deux usages prédominent en SHS pour la régression logistique. La première est de construire un modèle explicatif en identifiant les variables qui participent à la production d'un phénomène. Cela peut amener à la construction de plusieurs modèles emboîtés pour vérifier les effets d'interdépendance des variables et affiner la réflexion sur les relations causales.

Un autre usage est celui d'un tableau croisé généralisé. Il permet de contrôler l'effet de corrélation entre les variables pour dégager la corrélation « toute chose égale par ailleurs ». En effet, en utilisant plusieurs variables indépendantes, la modélisation va prendre en compte les corrélations multiples et ainsi rendre plus visibles des phénomènes qu'un simple tableau croisé ne permettait pas de montrer.

Disons-le tout de suite : un usage maîtrisé de la régression logistique s'appuie sur une compréhension détaillée du modèle et de ce qu'il est possible de faire. Suivant votre domaine, l'usage des régressions logistiques peut varier. Nous nous contenterons ici de montrer comment il est possible de calculer cette régression logistique, mais nous vous laissons la partie de construction du modèle et d'interprétation.

---

15. Pour une présentation détaillée de la régression logistique en anglais Hosmer *et al.* (2013). Nous ne présentons que la régression binaire par la suite, mais les bibliothèques permettent de faire des régressions logistiques à plus de modalités.

La régression logistique permet d'interroger l'importance des facteurs dans la probabilité d'avoir une proportion de femmes en dessous ou au-dessus de la médiane nationale. Pour cela, la variable dépendante est la proportion de femmes et les variables indépendantes le recodage en quartile des variables d'intérêt utilisées précédemment.

Nous allons utiliser la bibliothèque *Statsmodels*. La démarche est similaire à celle de la régression linéaire vue dans la partie précédente. Une étape supplémentaire est cependant nécessaire pour créer le tableau disjonctif.

### **❶ Information**

#### Créer le tableau de données

La régression logistique modélise l'effet de chacune des modalités (par exemple, l'effet du genre va être découpé en l'effet d'être un homme, une femme, ou autre). Le codage de ces valeurs peut avoir un effet majeur dans l'interprétation des résultats de la régression. Une règle classique qui permet d'interpréter facilement les coefficients de la régression logistique est de transformer une variable avec plusieurs modalités en autant de sous-variables correspondant à chaque modalité, codée avec 0 (absence) et 1 (présence), en enlevant une des modalités qui servira de référence. Ce faisant, les *odds ratios* obtenus par rapport aux coefficients seront à chaque fois ceux de la modalité correspondante par rapport à la référence.

Nous devons donc construire une colonne pour la variable dépendante qui prend 0 ou 1 suivant les situations et un tableau des variables indépendantes sous la forme d'un tableau où chaque colonne correspond à une modalité, en enlevant la colonne correspondant à la modalité de référence pour chaque variable et en ajoutant une colonne de valeur uniforme de 1 jouant le rôle de constante (appelée interception ou *intercept* en anglais).

Vous pouvez préparer les données de deux façons : construire vous-même ce tableau ou utiliser une bibliothèque spécifique qui génère le tableau correspondant.

### **❶ Information**

#### Mettre en forme un tableau disjonctif avec Patsy

La fonction `dmatrices` de la bibliothèque *Patsy* permet de mettre en forme un tableau suivant une formule préétablie. S'inspirant du langage R, elle prend pour argument un tableau global de données et cette formule de composition pour produire le tableau de sortie correspondant.

Dans le cas suivant, la formule sépare les variables indépendantes des variables dépendantes en précisant dans le modèle que `prop_f_dich` va dépendre (signe tilde) de `prop_ouvriers_C`, `prop_sup75_C` et `P15_POP_C` où on précise avec `C()` que ces modalités sont des catégories. La syntaxe avec le tilde et les modalités en catégories `C()` est spécifique à `patsy` et non à Python. Elle rajoute aussi automatiquement une colonne spécifique pour l'interception.

```
import statsmodels.api as sm
from patsy import dmatrices

# Mise en forme des données
tableau["prop_f_dich"] = pd.qcut(tableau["prop_f"],
                                  [0,0.5,1],[0,1])
tab_reg = tableau[["prop_f_dich", "prop_ouvriers_C",
                   "prop_sup75_C", "P15_POP_C"]].dropna()

# Création des variables dépendantes et indépendantes
y, X = dmatrices(
    'prop_f_dich~C(prop_ouvriers_C)+C(prop_sup75_C)+C(P15_POP_C)',
    tab_reg, return_type="dataframe")
y = y[y.columns[1]]

# Application de la régression
modele = sm.Logit(y,X)
regression = modele.fit()

# Affichage
print(regression.pvalues)
```

```
Optimization terminated successfully.
    Current function value: 0.625686
    Iterations 5
Intercept                      7.396532e-153
C(prop_ouvriers_C) [T.Q2]       3.574940e-03
C(prop_ouvriers_C) [T.Q3]       7.968558e-14
C(prop_ouvriers_C) [T.Q4]       2.311467e-45
C(prop_sup75_C) [T.Q2]          3.492900e-25
C(prop_sup75_C) [T.Q3]          5.644946e-82
C(prop_sup75_C) [T.Q4]          5.653602e-184
C(P15_POP_C) [T.Q2]             5.633297e-16
C(P15_POP_C) [T.Q3]             6.912931e-73
C(P15_POP_C) [T.Q4]             0.000000e+00
dtype: float64
```

Ce code :

- ⊕ charge *Statsmodels* et la fonction `dmatrices` de la bibliothèque *Patsy* ;
- ⊕ construit une nouvelle variable `prop_f_dich` dans le tableau qui prend 0 si la valeur est sous la médiane et 1 sinon avec la fonction `qcuts` de *Pandas* ;
- ⊕ prend un sous-ensemble des données pour enlever toutes les lignes où il y a une valeur manquante avec `dropna` afin de ne pas poser problème ensuite ;
- ⊕ utilise `dmatrices` pour fabriquer le tableau disjonctif avec une formule spécifique ;
- ⊕ définit le modèle logistique en précisant que la variable dépendante est la première colonne de `y` (où la valeur vaut 1) et les variables indépendantes sont dans `X`, puis applique le modèle avec `fit` ;
- ⊕ affiche les significativités des coefficients (nous affichons plus bas l'ensemble des paramètres).

### ?

### Exercice

Affichez l'ensemble des résultats de la régression.

```
print(regression.summary())
```

Plutôt que d'utiliser la bibliothèque *Patsy* pour construire les tableaux, il est possible de les construire directement avec le code suivant :

```
X = pd.get_dummies(tab_reg[["prop_ouvriers_C", "prop_sup75_C", ↴
    "P15_POP_C"]])
X["Intercept"] = 1
X = X[[i for i in X.columns if not "Q1" in i]]
y = tab_reg["prop_f_dich"]
```

Ce code :

- ⊕ crée un tableau disjonctif des variables indépendantes ;
- ⊕ rajoute une colonne correspondant à l'interception, important pour le modèle logistique ;
- ⊕ enlève toutes les colonnes qui correspondent au premier quartile des variables et qui vont jouer le rôle de référence ;
- ⊕ définit la colonne de la variable dépendante.

Le résumé de la régression logistique permet d'avoir les coefficients de la régression, l'intervalle de confiance et la significativité de chaque coefficient. Le modèle de régression logistique que nous avons calculé permet de récupérer les différents éléments qui peuvent nous intéresser.

- ⊕ `regression.params` permet d'avoir les paramètres ;
- ⊕ `regression.conf_int()` permet d'avoir les intervalles de confiance ;

- ✿ `regression.pvalues` permet d'avoir les probabilités critiques (*p-values*).

Nous constatons que si toutes les modalités semblent avoir un effet (la significativité indiquée par la *p-value* est nulle ou presque pour toutes), les coefficients sont à la fois positifs et négatifs indiquant des effets dans des sens différents.

La présentation habituelle en SHS est généralement sous la forme d'*odds ratios* et leur intervalle de confiance plus facilement interprétables. Les coefficients étant à chaque fois calculés par rapport à la valeur de référence (ici, le premier quartile), ils permettent d'interpréter le gain de vraisemblance par rapport à cette catégorie de référence.

Dans une régression logistique, en raison de la forme *logistique* du modèle, le passage des coefficients aux *odds ratios* se fait en calculant la valeur exponentielle des coefficients. La fonction exponentielle est disponible dans la bibliothèque *Numpy*. Nous pouvons adapter nos résultats pour construire un nouveau tableau qui se prête plus aux habitudes en SHS à partir des éléments que renvoie la régression. La valeur renvoyée par `np.exp(regression.conf_int())` est un tableau *Pandas* avec deux colonnes pour les bornes de l'intervalle de confiance :

```
import numpy as np

df = np.exp(regression.conf_int())
df['odd ratio'] = round(np.exp(regression.params), 2)
df["p-value"] = round(regression.pvalues, 2)
df[["IC"]] = df.apply(lambda x : "%.{2f} [%.{2f}-{.2f}]" \
                     % (x["odd ratio"],x[0],x[1]),axis=1)
df = df.drop([0,1], axis=1)
df
```

	odd ratio	p-value	IC
Intercept	0.39	0.0	0.39 [0.36-0.42]
C(prop_ouvriers_C) [T.Q2]	0.91	0.0	0.91 [0.85-0.97]
C(prop_ouvriers_C) [T.Q3]	0.79	0.0	0.79 [0.74-0.84]
C(prop_ouvriers_C) [T.Q4]	0.63	0.0	0.63 [0.59-0.67]
C(prop_sup75_C) [T.Q2]	1.40	0.0	1.40 [1.31-1.49]
C(prop_sup75_C) [T.Q3]	1.85	0.0	1.85 [1.74-1.97]
C(prop_sup75_C) [T.Q4]	2.60	0.0	2.60 [2.44-2.77]
C(P15_POP_C) [T.Q2]	1.30	0.0	1.30 [1.22-1.38]
C(P15_POP_C) [T.Q3]	1.80	0.0	1.80 [1.69-1.92]
C(P15_POP_C) [T.Q4]	6.09	0.0	6.09 [5.68-6.52]

Ce code :

- ✿ charge la bibliothèque *Numpy* pour ensuite la fonction exponentielle ;

- ⊕ récupère les intervalles de confiance de la régression avec `regression.params` puis calcule leur exponentiel avec `np.exp` en arrondissant l'ensemble et stocke l'ensemble dans la variable `df` ;
- ⊕ rajoute une colonne `odd_ratio` en appliquant de la même manière la fonction exponentielle aux paramètres de la régression ;
- ⊕ rajoute une colonne `p-value` de la même manière ;
- ⊕ rajoute une colonne `intervalle de confiance` qui met en forme de manière plus lisible les informations ;
- ⊕ supprime les deux premières colonnes du tableau qui sont maintenant résumées dans la colonne `IC` et affiche le tableau ainsi construit.

Tous les coefficients sont significatifs, mais celui qui apparaît avoir le poids le plus important est la taille, avec un facteur de 6 pour le dernier quartile. Ainsi, les villes appartenant au dernier quartile ont, toutes choses égales par ailleurs, 6 fois plus de chance d'avoir une proportion de femmes au-dessus de la médiane.

## ?

### Exercice

#### Refaites le modèle en dichotomisant par rapport à la proportion nationale de femmes.

Le seul changement à faire est dans la variable dépendante `y`, qui dans ce cas est 51.59%. On va utiliser alors non pas la fonction `qcuts` pour les quantiles mais `cut` qui permet de définir des intervalles

```
tableau["prop_f_dich"] = pd.cut(tableau["prop_f"],
[0,51.59,100],labels=[0,1])
tableau["prop_f_dich"].value_counts()
```

La distribution n'est plus moitié/moitié pour les villes et l'interprétation des résultats sera alors différente.

Il est possible d'interpréter le modèle de manière similaire à la régression linéaire avec un pseudo- $R^2$  donné par `regression.prsquared`, permettant aussi, sous certaines conditions, de comparer les modèles entre-eux.

Nous n'avons présenté ici que la régression logistique binaire. Il existe bien d'autres modèles de régression logistique et, au-delà, d'autres modèles de régression tout court. Par exemple, les modèles multiniveaux, ou les modèles de survie. Chacun de ces modèles est généralement lié à une communauté de pratique, qui a souvent pris le temps de développer une bibliothèque. Avant d'utiliser un nouvel outil, prenez le temps de regarder qui l'utilise pour savoir si la mise en forme des résultats correspond bien à vos besoins. Bien souvent, c'est le cas et passé l'étrangeté d'un nouvel outil, vous serez en mesure de retrouver une présentation plus familière.

Passons maintenant à un troisième type de traitement statistique largement généralisé et répondant à une autre logique : les classifications.

## 8.4 Les classifications

### 8.4.1 Quelques éléments généraux

Le but des méthodes de classification est de construire une partition des données en groupes, ou *clusters*, soit pour simplifier un problème, soit pour rendre la structure des données visible. Elles permettent ainsi de construire des typologies en prenant en compte la complexité des informations.

Par exemple, les stratégies de classification peuvent être utilisées pour regrouper des aires géographiques avec des propriétés similaires, identifier des proximités de mots dans des textes ou encore repérer des individus aux propriétés sociales proches.

Comme beaucoup de méthodes, la classification peut être utilisée soit dans une perspective exploratoire en amont d'une analyse, soit confirmatoire pour mettre à l'épreuve des hypothèses.

Ainsi, la classification hiérarchique ascendante se retrouve dans de nombreux traitements : réunir des documents similaires sur la base des mots qui les composent, des individus par rapport à leurs caractéristiques sociales, etc. Elle peut par exemple compléter une analyse en composantes pour rendre les effets de distorsion de la projection visibles.

La situation est cependant complexe car les méthodes existantes se basent sur des principes différents. Non seulement il y a un enjeu concernant la sélection des informations à utiliser (sélection des variables), mais ensuite les méthodes de regroupement dépendent de la manière dont on calcule la distance entre deux individus (la métrique) et les règles de sélection pour associer ou séparer deux individus. La conséquence principale est qu'il existe peu de règles en SHS sur les bonnes méthodes à utiliser ainsi que sur la manière d'évaluer une classification<sup>16</sup>.

Python permet facilement d'utiliser de nombreuses fonctions de classification. Nous allons cependant nous concentrer sur deux approches couramment utilisées : la méthode des k-moyennes (ou *K-means*) et la classification hiérarchique ascendante. D'une certaine manière, une fois la logique de la démarche classificatoire présentée, changer un modèle pour un autre se fait assez facilement.

### 8.4.2 Partitionnement en k-moyennes

La logique des k-moyennes (*k-means*) ou nuées dynamiques est simple : après avoir défini le nombre de groupes à rechercher (défini par un nombre *k*), l'algorithme découpe l'ensemble des données en *k* groupes avec comme objectif de réduire la distance entre les éléments contenus au sein de chaque groupe. Un des avantages de cette méthode est sa rapidité sur de très grands ensembles de données, contrai-

---

16. Il existe différentes manières d'évaluer une classification. Elles dépendent cependant des domaines et nous ne les développerons pas dans ce manuel.

rement à d'autres approches qui peuvent être davantage gourmandes en puissance de calcul. La bibliothèque *Scikit-learn* contient une fonction dédiée `KMeans`, qui a une utilisation similaire aux autres modèles précédents.

## ❶ Information

### Choisir le nombre de groupes

Combien de groupes ? Cette question n'a pas de réponse définitive. Plusieurs stratégies existent (en fait, plus d'une trentaine), certaines basées sur des informations de contextes (vous avez de bonnes raisons de penser qu'il y a un nombre particulier de groupes), d'autres sur une approche plus calculatoire (calculer l'évolution de la somme des carrés des distances interclusters et choisir le nombre tel qu'ajouter un cluster en plus ne modifie pas beaucoup cette somme).

On peut donc essayer de regrouper les villes françaises dans 3 catégories, avec l'idée intuitive que cela pourrait nous aider à réfléchir sur la distribution de la proportion de femmes si on avait une typologie. Les données utilisées sont les mêmes que celles de la régression logistique précédemment.

```
from sklearn.cluster import KMeans

# Sélection des données
tableau_cluster = tableau[["prop_f","prop_ouvriers",
                           "prop_sup75","P15_POP"]].dropna()

# Application de la classification
kmeans = KMeans(n_clusters=3)
kmeans.fit(tableau_cluster)

# Mise en forme et affichage
moyennes_groupes = pd.DataFrame(kmeans.cluster_centers_)
moyennes_groupes.columns = tableau_cluster.columns
moyennes_groupes
```

	prop_f	prop_ouvriers	prop_sup75	P15_POP
0	49.948236	12.221370	9.922756	1.581008e+03
1	52.900000	3.900000	7.700000	2.206488e+06
2	52.327907	8.767442	8.167442	1.882963e+05

Ce code :

- ➊ importe la fonction `KMeans` ;

- ➊ crée un tableau de données à classifier ;
- ➋ initialise le classifieur en précisant que l'on souhaite 3 groupes à la fin ;
- ➌ applique ce classifieur à nos données ;
- ➍ récupère le centre de chacun de ces 3 groupes défini comme la moyenne des variables, disponible dans `cluster_centers_` et le transforme en tableau `Pandas` ;
- ➎ met le nom des variables sur les colonnes ;
- ➏ affiche le tableau.

Il apparaît sur les trois groupes qu'un des groupes se caractérise par une proportion de femmes basse, un taux d'ouvriers et un taux de personnes âgées plus important, accréditant l'hypothèse qu'il peut exister un lien entre ces variables. Combien a-t-on de villes dans chaque groupe ?

```
pd.Series(kmeans.labels_).value_counts()
```

```
0    35349
2      43
1      1
dtype: int64
```

La distribution est très particulière et isole Paris dans un groupe en raison de sa population très différente. Il n'est pas évident de comprendre pourquoi les deux autres groupes ont été constitués. Dans certains cas, une forme d'évidence peut apparaître. Dans d'autres cas un travail interprétatif est nécessaire. Une manière de procéder est de voir la proportion d'autres caractéristiques dans ces groupes. Par exemple, si je prends les régions, est-ce que je trouve la même proportion de chaque groupe ?

Une étape souvent intéressante ensuite est de rajouter cette information du groupe dans nos données pour l'intégrer à un autre traitement, par exemple une régression logistique. Dans notre cas, il suffit de rajouter une colonne à nos données initiales : `donnees_cluster["groupe"] = kmeans.labels_`.

### ?

### Exercice

**Regardez l'évolution de la classification en changeant le nombre de groupes, en enlevant la variable de la population et en utilisant la normalisation de StandardScaler.**

Les résultats sont amenés à beaucoup varier et posent la question du « bon » réglage.

### 8.4.3 Classification hiérarchique ascendante

La classification hiérarchique ascendante est une autre approche de la classification. Le principe est simple : à la première étape, les deux éléments les plus proches sont regroupés et deviennent un nouvel élément caractérisé par la moyenne des valeurs. À l'étape suivante, on recommence. Et ainsi de suite. À la fin, tous les éléments sont regroupés et cela forme un arbre inversé, chaque feuille étant un regroupement. Un des avantages de cette approche est la possibilité de choisir le nombre de groupes une fois l'analyse réalisée en coupant l'arbre à un niveau déterminé. Une de ses limites est qu'elle est moins adaptée à de très grands ensembles de données.

La démarche pratique est la suivante : une première étape réalise la mise en relation des éléments. Il est important de définir ce que veut dire « être proche » car il existe plusieurs manières de mesurer les distances.

Reprendons le même cas que précédemment avec les quelques variables qui nous intéressent. Cependant, comme le jeu de données est très grand, nous allons le faire uniquement sur un échantillon aléatoire de 1000 villes<sup>17</sup>.

La première étape est de construire ces associations de proximité entre les éléments. La fonction `linkage` permet de calculer le tableau de la manière dont les différents éléments vont être progressivement réunis dans des groupes (chaque ligne du tableau indiquant quels éléments sont réunis).

La manière de réunir les éléments se fait suivant une méthode de choix. Nous allons utiliser la méthode dite de « ward », qui vise à minimiser la variation à l'intérieur du cluster suivant une distance euclidienne, mais d'autres options peuvent être envisagées et testées suivant vos données<sup>18</sup>.

```
from scipy.cluster.hierarchy import linkage

tableau_cluster = tableau[["prop_f","prop_ouvriers",
                           "prop_sup75","P15_POP"]].dropna().sample(n=1000)

arbre_associations = linkage(tableau_cluster.values, 'ward')
```

Ce code :

- ➊ importe la fonction `linkage` ;
- ➋ crée un jeu de données à partir des données en prenant 1000 villes aléatoirement ;
- ➌ construit l'arbre des associations ;

17. Si votre ordinateur est puissant, n'hésitez pas à prendre l'ensemble du jeu de données.

18. Les approches les plus récentes dans l'analyse des textes mobilisent différentes métriques pour calculer les distances, pour un exemple récent d'une application sur les pièces de Molière Cafiero & Camps (2019).

- ✿ affiche la première association (les deux premiers éléments sont les deux villes associées entre elles, le troisième la distance qui les séparait).

L'étape suivante est de représenter ce regroupement :

```
from scipy.cluster.hierarchy import dendrogram
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
dendrogram(arbre_associations, distance_sort='ascending',
           truncate_mode = 'lastp', p=100,
           color_threshold = 50000, leaf_rotation=90.,
           leaf_font_size=8., ax=ax)
fig.set_figwidth(12)
fig.set_figheight(8)
```

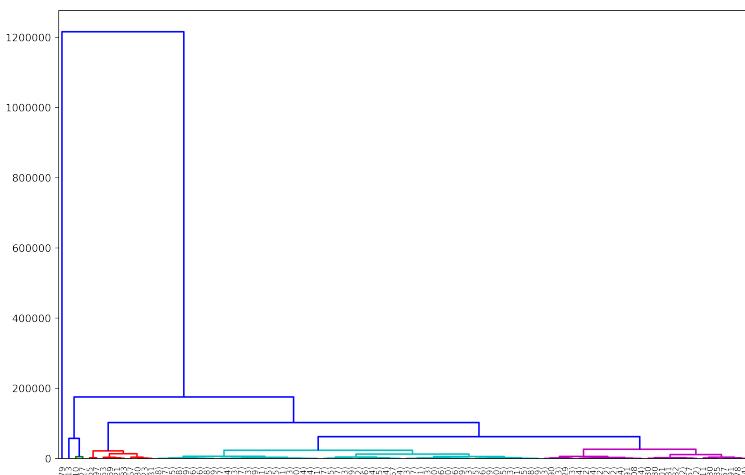


Fig. 8.4 – Visualisation du dendrogramme de la CHA

Ce code utilise la fonction `dendrogram` pour représenter cette association successive. Avec l'option `truncate_mode` nous représentons uniquement les `p=100` derniers niveaux. Nous choisissons aussi un seuil pour définir et colorier les classes avec `color_threshold = 50000` après avoir vu une première fois le graphique. Sur la figure 8.4, les abscisses représentent les différents regroupements et les ordonnées représentent la distance qui sépare deux regroupements à franchir pour les rassembler. Plus cette distance est grande, plus les groupes sont différents.

La dernière étape est de « couper » cet arbre pour attribuer chaque élément à un groupe. Il n'existe pas de critère absolu qui vous dirait où couper, cela dépend de votre analyse - même si certaines règles peuvent vous guider. Pour cela, une dernière fonction est utile : `fcluster`.

```
from scipy.cluster.hierarchy import fcluster
distance = 50000
clusters = fcluster(arbre_associations, distance,
                     criterion='distance')
pd.Series(clusters).value_counts()
```

```
4    929
5     57
3     10
1      2
6      1
2      1
dtype: int64
```

Ce code :

- ➊ importe la fonction `fcluster` qui sert à « couper » l'arbre ;
- ➋ définit une distance max `distance` à laquelle se fait cette coupure ;
- ➌ coupe le tableau avec cette distance en précisant que le critère est bien celui de la distance (d'autres critères existent, vous pouvez aussi déterminer directement le nombre de clusters que vous voulez : `fcluster(linked, 5, criterion='maxclust')` pour 5 clusters) ;
- ➍ affiche la distribution des différents groupes.

### ➊ Exercice

**Couper l'arbre pour faire 3 groupes différents.**

Utilisez l'option `maxclust`.

Comme précédemment, une fois les groupes constitués, le travail commence pour les interpréter. Limitons-nous à regarder la proportion dans chaque groupe de la proportion de femmes.

```
tableau_cluster["cluster"] = clusters
tableau_cluster.groupby("cluster")["prop_f"].agg(["mean",
                                                 "std", "count"])
```

cluster	mean	std	count
1	52.600000	0.282843	2
2	50.800000	NaN	1
3	51.800000	0.986577	10
4	49.832831	2.641873	929
5	51.356140	1.766253	57
6	52.600000	NaN	1

Ce code :

- ✚ ajoute une colonne avec le numéro des clusters dans le tableau des données ;
- ✚ regroupe le tableau par cluster avec `groupby`, sélectionne la colonne d'intérêt et calcule trois indicateurs : la moyenne, l'écart-type et le nombre d'éléments.

Comme nous ne travaillons que sur un échantillon, il n'y a pas vraiment d'analyse définitive à faire. Mais nous retrouvons la tendance d'un groupe avec une proportion moyenne plus faible qui se caractérise aussi par une proportion d'ouvriers et de personnes âgées plus forte.

Avant de conclure ce chapitre, nous voulons encore présenter une autre approche de statistiques avancées qui gagne beaucoup en visibilité à l'époque des promesses du « Big Data » et de « l'Intelligence Artificielle » : l'apprentissage automatique sur des données.

## 8.5 Les modèles d'apprentissage automatique

Les évolutions récentes nécessitent de faire un point sur l'apprentissage automatique (ou *machine learning* en anglais) dans le cadre d'un chapitre de statistiques avancées, même si l'usage en SHS est encore assez limité<sup>19</sup>.

### 8.5.1 Quelques éléments généraux

Pour le dire schématiquement, l'apprentissage automatique revient à utiliser des modèles entraînés sur un jeu de données initial pour ensuite *prédirer* des résultats sur d'autres données. Vous connaissez déjà une partie de ces modèles : par exemple, la régression logistique peut être utilisée pour faire de l'apprentissage automatique. La vraie différence de l'apprentissage automatique avec l'approche exploratoire ou explicative vue précédemment se fait dans l'approche même et le rapport aux données.

---

19. Pour un exemple récent utilisant l'apprentissage automatique et les réseaux de neurones en sciences politiques : Boelaert & Ollion (2020). Pour une réflexion plus générale par les mêmes auteurs : Boelart & Ollion (2018).

L'apprentissage automatique a largement bénéficié de l'expansion de la puissance de calcul et de l'utilisation d'un type particulier d'algorithmes, les réseaux de neurones. Si nous n'allons pas aborder ces modèles particuliers, leur usage en Python se fait de la même manière que les autres modèles, régression logistique ou les arbres de décisions.

Comme la prédiction n'est pas au cœur du projet scientifique des SHS, ces approches sont pour le moment moins visibles. Il est rare que vous soyez en situation de faire une recommandation de produit ou de publicité adaptée comme les algorithmes des sites en lignes. Pourtant, dans certains cas, cela peut être intéressant d'utiliser ces approches pour construire vos outils.

Par exemple, de nombreux outils d'annotation automatique du texte sont développés grâce à des approches d'apprentissage automatique pour permettre ensuite que vous puissiez annoter un nouveau texte. Imaginez que vous êtes dans une situation où vous devez coder un grand corpus : peut-être est-il plus intéressant d'entraîner un modèle pour qu'il puisse ensuite le faire à votre place ?

Nous ne rentrerons pas dans les problématiques spécifiques de l'apprentissage automatique, non seulement parce qu'il existe des livres très bien faits sur l'utilisation de Python pour ces traitements mais surtout que nous sortirions des objectifs de ce manuel. Par exemple, il existe un enjeu important autour de la validation d'un modèle sur des corpus différents de celui d'apprentissage avec des indicateurs spécifiques. Typiquement, la notion de test statistique disparaît en apprentissage automatique.

Dans tous les cas, une des grandes forces du langage Python réside dans sa souplesse pour l'apprentissage automatique et le traitement des grands corpus de données associées. Quel que soit le modèle que vous allez utiliser, la démarche est sensiblement la même. Pour vous montrer comment procéder, nous allons prendre le cas de la régression logistique.

### 8.5.2 Régression logistique pour prédire

Nous avons déjà vu la régression logistique comme modèle pour représenter les données. Mais une fois ce modèle ajusté, nous pouvons aussi l'utiliser pour prédire la sortie avec une nouvelle entrée.

Une question typique de l'apprentissage automatique est la classification : suivant les données que j'ai, est-ce que je suis dans telle ou telle catégorie ? Par rapport à notre exemple, suivant les caractéristiques de la ville, est-ce que la proportion de femmes est supérieure ou inférieure à la médiane ? Poser une telle question repose bien sur une logique de prédiction.

Une première étape est de définir notre jeu de données qui va permettre d'entraîner le modèle et le jeu de données qui va permettre de le tester. Ces deux opérations doivent se faire sur des données différentes. Pour cela, il existe avec la bibliothèque *Scikit-learn* un ensemble de fonctions dédiées qui permettent de rendre très rapide ces étapes.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Chargement des données et mise en forme
tableau_totales = tableau[["prop_f_dich","prop_ouvriers_C",
                           "prop_sup75_C","P15_POP_C"]].dropna()

X_train, X_test, y_train, y_test = train_test_split(
    tableau_totales[["prop_ouvriers_C",
                      "prop_sup75_C","P15_POP_C"]],
    tableau_totales['prop_f_dich'], test_size=0.20)

X_train = pd.get_dummies(X_train)
X_test = pd.get_dummies(X_test)

# Application du modèle
modele_logistique = LogisticRegression()
modele_logistique.fit(X_train.values, y_train.values)
prediction = modele_logistique.predict(X_test)

# Affichage des résultats
score = modele_logistique.score(X_test, y_test)
print(round(score,2))
```

0.65

Ce code :

- ⊕ charge la fonction permettant de séparer un corpus en deux sous-ensembles pour configurer le modèle et le tester ;
- ⊕ charge la fonction de la régression logistique ;
- ⊕ sélectionne les données que l'on souhaite utiliser comme corpus total ;
- ⊕ sépare ce corpus en quatre sous-groupes : le corpus d'entraînement, en séparant variable dépendante et variables indépendantes et le corpus de test, séparé de la même manière ;
- ⊕ transforme les tableaux en tableaux disjonctifs (une colonne par modalité nécessaires pour la fonction logistique de *Scikit-learn* (une approche différente de celle utilisée avec la régression logistique vue précédemment) ;
- ⊕ crée le modèle logistique ;

- ⊕ l'entraîne sur les données d'entraînement avec la fonction `fit` ;
- ⊕ prédit quelles seraient les sorties avec le corpus de test ;
- ⊕ calcule le score de « réussite » avec la fonction `score` : à quel point on a bien réussi à prédire avec le modèle, comparé aux données réelles que nous connaissons aussi.

On a un score autour de 64 %. Sachant que la variable dépendante est divisée avec la médiane (moitié/moitié), on aurait un score de 50 % si on choisissait systématiquement l'une ou l'autre des valeurs de sortie. Le modèle permet donc bien de prédire (un peu) la valeur de sortie.

### Information

#### Les courbes ROC et l'indicateur AUC

Vous les croiserez sûrement si vous vous intéressez à l'apprentissage automatique. La courbe ROC (pour *Receiver Operating Characteristic*) représente le taux de prédiction positive en fonction du taux de prédiction négative suivant la variation des paramètres du modèle et permet de caractériser par l'aire sous la courbe (AUC pour *Area Under Curve*) l'efficacité du modèle.

Dans notre cas, la régression logistique permet de calculer une probabilité d'avoir la catégorie 1 (« être au-dessus de la moyenne »). C'est l'utilisateur qui définit ce seuil à partir duquel la valeur est effectivement mise dans la catégorie 1. Or notre modèle est forcément imparfait. Si nous décidons de ne classer en 1 que les entrées qui ont une probabilité très haute, nous allons manquer certaines prédictions et faire baisser le taux de prédiction positive. À l'inverse, si nous considérons qu'une prédiction est codée 1 même si la probabilité n'est pas très forte, nous allons avoir beaucoup de faux positifs et faire augmenter le taux de prédiction négative. L'objectif étant de trouver le seuil qui optimise le modèle.

L'enjeu central dans l'apprentissage automatique est d'améliorer le modèle pour améliorer la prédiction. Cela passe soit en transformant le corpus d'entrée (faire une analyse factorielle, sélectionner les bonnes variables, etc.) soit en changeant le modèle lui-même pour des modèles plus adaptés. En testant à chaque fois la performance du modèle (et il existe plusieurs stratégies pour le faire) l'objectif est d'obtenir à la fin la meilleure combinaison qui pourra ensuite fonctionner sur d'autres données.

Chaque modèle est finalement très dépendant des données utilisées pour l'apprentissage. Dans notre cas, le corpus est français sur les villes françaises. Nous pouvons douter de l'efficacité de ce modèle pour prédire la proportion de femmes en prenant en entrée des données de villes chinoises, par exemple. L'apprentissage automatique, dans toutes ses formes, est donc intrinsèquement lié aux corpus utilisés pour

l'apprentissage. Cette affirmation est vraie pour toute analyse de données, mais la tendance avec l'apprentissage automatique à utiliser des grands ensembles de données peut faire oublier cette dépendance. Il faut le garder en tête. Demandez-vous toujours : quelles données ont été utilisées ?

## 8.6 Synthèse du chapitre

Ce chapitre propose de mettre en œuvre des traitements statistiques avancés avec Python. Nous avons vu en particulier l'analyse en composantes, la modélisation et la classification.

Comme ces traitements statistiques relèvent de domaines très différents, la priorité est de réfléchir leur pertinence aux questions explorées. La mise en œuvre est ensuite toujours possible avec Python. Elle peut être plus ou moins immédiate suivant les besoins.

La limite de la facilité de mise en œuvre de ces traitements est l'existence d'une bibliothèque dédiée : dans certains cas, des bibliothèques parfaitement adaptées permettent en quelques lignes d'entraîner un modèle statistique et d'afficher les résultats souhaités. Dans d'autres cas, il est nécessaire de compléter les traitements avec un peu de code supplémentaire. Nous avons essayé de donner dans ce chapitre les principaux outils dont vous aurez besoin en vous montrant comment adapter les outils existants au mieux pour les intégrer dans votre analyse.

Un dernier mot : nous n'avons bien sûr pas présenté toutes les analyses et toutes les bibliothèques existantes. Chaque domaine a ses spécificités, avec souvent des outils dédiés en Python. Pensez à interroger la communauté Python si vous avez des besoins spécifiques.

# Chapitre 9

## Usages avancés

Dans ce chapitre nous souhaitons vous montrer comment récupérer des données plus complexes et mettre en œuvre de nouvelles analyses. Nous abordons ces applications à travers des études de cas pour donner un exemple concret d'utilisation en présentant différentes bibliothèques.

### 9.1 Récupérer des données complexes

Dans le cas du calcul de statistiques, nous nous sommes surtout concentrés sur le traitement de données déjà mises en forme dans un fichier. Nous avons cependant rencontré la possibilité de transformer d'autres sources d'informations en données, par exemple en récupérant des pages internet.

Beaucoup d'informations se présentent d'abord comme du matériau « qualitatif », c'est-à-dire du contenu à interpréter et à recoder avant de pouvoir procéder à une analyse. En combinant l'interprétation humaine avec la programmation, vous êtes en mesure de développer des outils adaptés à une multitude de sources d'information pour récolter les données. Dans cette partie, nous présentons comment indexer des entretiens pour les analyser à l'aide de scripts puis comment traiter du contenu récolté sur internet.

#### 9.1.1 Transformer du matériau qualitatif en données

En SHS, et plus particulièrement en sociologie ou en anthropologie, les enquêtes produisent souvent des données dites qualitatives : entretiens, images, fichiers divers d'archive. Il n'est pas toujours possible de faire les statistiques habituelles qui nécessitent un tableau d'individus et de variables. Cependant, ces informations peuvent être transformées soit pour se mettre dans une situation plus habituelle, soit pour aider l'interprétation en mobilisant les possibilités interactives de la pro-

grammation Python. Ainsi, un texte ou une image peuvent être codés par un observateur pour ensuite effectuer des repérages, des croisements ou des analyses de ce codage.

## Principe d'indexation des entretiens

Comme humain, vous serez toujours plus efficace pour traiter le sens contenu dans vos entretiens ou vos archives, surtout quand le nombre de textes est limité. Les textes sont des objets visant à être compris et pour le moment rien ne remplace une lecture humaine. Un être humain est généralement capable de comprendre le sens d'un texte ce qu'un ordinateur n'arrive (pas vraiment) à faire.

Un script informatique sera par contre toujours meilleur que vous pour retrouver des éléments dans un document, comparer des documents entre eux, ou faire des traitements sur la fréquence de présence conjointe de certains éléments. Toutes ces opérations sont automatisables. Il est possible de tirer parti du meilleur de ces deux mondes en créant des outils adaptés à l'analyse qualitative.

### Information

#### Exemple de programmation en analyse qualitative

Émilien vient de faire des entretiens avec des médecins et des chercheurs sur la crise du coronavirus. Ces entretiens ont été retranscrits. Au fil de leur lecture, il code les passages qui l'intéressent. Il utilise ensuite un script Python pour présenter la proportion des différentes catégories dans les entretiens. Comme il a aussi associé chaque entretien à différentes informations (âge du médecin, hôpital d'appartenance, etc.) il peut faire des analyses croisées. Il utilise aussi Python pour facilement extraire des passages contenant certains termes spécifiques pour illustrer son rapport. Comparé à d'autres logiciels dédiés, ce traitement vient en parallèle de sa lecture et permet de construire au gré de l'enquête des analyses exploratoires de son matériau.

Si vous réalisez une étape de codage en repérant dans vos documents les passages qui vous intéressent, vous pouvez ensuite utiliser l'analyse de données pour rechercher, croiser et visualiser vos données. Il convient alors de :

- ➊ construire une grille de codage de vos documents (des catégories pour coder le texte) ;
- ➋ lire le texte en marquant dans celui-ci les zones qui correspondent aux catégories ;
- ➌ utiliser un script pour extraire ces informations, par exemple :
  - ★ compter la fréquence des catégories ;
  - ★ identifier les cas où plusieurs catégories apparaissent ensemble.

Pour reprendre l'exemple précédent, vous avez réalisé des entretiens avec des médecins. Vous avez retroussé ces entretiens et vous voulez pouvoir les analyser autour des enjeux liés à l'urgence. Comment faire ?

### ❶ Information

#### Les logiciels de QDA

Les logiciels d'analyse des données qualitatives (ou *Qualitative Data Analysis*) réalisent les traitements que nous présentons ici avec une interface visuelle. Pourquoi alors le faire en Python ? Comme pour beaucoup de cas dans ce manuel, cela dépend effectivement de vos besoins. Une approche reposant sur Python :

- ✚ est très souple et adaptable ;
- ✚ est gratuite ;
- ✚ est ouverte (travail directement dans les textes, facile à échanger) ;
- ✚ peut se combiner avec d'autres traitements.

#### Coder ses entretiens

La première étape est de coder les entretiens : cela signifie identifier dans le texte les passages qui vous intéressent. Si votre problématique est celle de la comparaison du vécu des personnes par rapport à d'autres situations, vous avez peut-être envie d'identifier les passages qui correspondent à des comparaisons qu'ils font (catégorie *comparaison*) et des différences qu'ils soulignent (catégorie *différence*). Cela fait deux catégories à coder.

En nous inspirant librement du format XML<sup>1</sup>, l'idée est de définir des balises, c'est-à-dire des symboles qui vont encadrer les passages nous intéressants. Pour la catégorie comparaison, nous allons utiliser la balise `<comparaison>` pour signifier qu'un extrait commence et `</comparaison>` pour signifier qu'un extrait finit. Pour la catégorie différence, nous allons utiliser la balise `<difference>` pour signifier qu'un extrait commence et `</difference>` pour signifier qu'un extrait finit. L'idée est de définir autant de balises que de catégories, puis de les utiliser pour coder le texte de vos documents<sup>2</sup>.

1. Pour le dire rapidement, le XML est un métalangage informatique sous la forme de balises qui permet de structurer des informations dans un document.

2. Un point important est le format des fichiers dans lequel vous allez travailler. S'il est toujours plus simple de traiter des informations textuelles, nous avons souvent des documents Word ou associé. Pour plus de facilité, utilisez le format `.docx` quitte à sauvegarder votre fichier codé dans ce format. Il est plus simple à ouvrir avec Python. Pour vous donner une idée, le code ressemblerait à : « blabla blabla `<comparaison>` blabla intéressant ici `</comparaison>` blablabla ».

Vous pouvez définir les balises comme vous le souhaitez. En fait, vous pouvez même trouver d'autres symboles pour débuter et finir les passages. Nous vous conseillons de faire comme pour les variables en Python : éviter les espaces et les caractères spéciaux. Dans la pratique, il est rare d'avoir des catégories définitives au début du codage. Comme pour toute typologie, c'est au cours du codage qu'on adapte ses catégories, et il est fréquent de créer progressivement de nouvelles catégories. Cela signifie aussi de devoir recoder les premiers documents. Une fois cette étape réalisée, il faut récupérer et analyser ces codages.

### Récupérer les informations dans un fichier docx

L'objectif est de récupérer les informations codées dans les fichiers. Si vous utilisez des fichiers textes .txt vous savez déjà le faire. Pour les fichiers Word il faut utiliser une nouvelle bibliothèque qui gère l'ouverture de ces documents. La première étape est d'ouvrir les fichiers en format .docx pour récupérer le texte.

Cette bibliothèque est `python-docx`. Elle permet de gérer la lecture et l'écriture de ce type de fichier avec un nouveau type d'objet, les *Documents*, qui représentent un fichier Word. Cet objet est organisé en paragraphes, chacun pouvant être mis en forme.

Dans notre cas, nous nous concentrerons sur le contenu du document : nous voulons récupérer le texte de ces paragraphes. Si vous êtes curieux, jetez un coup d'œil sur les autres fonctions dont elle dispose<sup>3</sup>. Nous avons mis dans les données un fichier « entretien.docx » un morceau d'entretien avec un médecin publié par France 3 Régions. Nous avons mis quelques balises. Vous trouverez ce fichier sur le dépôt du manuel.

```
from docx import Document

doc = Document("./data/entretien.docx")
paragraphes = [i.text for i in doc.paragraphs]
print(paragraphes[0])
```

Début entretien médecin

Ce code :

- ➊ importe l'objet *Document* de la bibliothèque ;
- ➋ charge le fichier dans un objet *Document* ;
- ➌ récupère le texte des paragraphes du document dans la variable *paragraphes* ;
- ➍ affiche le premier paragraphe.

---

3. Vous pouvez ajouter des tableaux, des images, mettre en forme les zones de textes... bref programmer votre document. Comme toujours, si cela vous intéresse, jetez un coup d'œil à la documentation. Cela peut aussi être fait avec des slides.

Nous pouvons maintenant définir une fonction qui prend un fichier quelconque en entrée et qui renvoie le contenu textuel brut.

```
def extraire_texte(fichier):
    doc = Document(fichier)
    paragraphes = [i.text for i in doc.paragraphs]
    return "\n".join(paragraphes)
```

Cette fonction fait la même opération que précédemment, sauf qu'elle renvoie tous les paragraphes réunis ensemble et séparés par un saut de ligne \n. Maintenant que nous avons le texte, nous voulons récupérer les extraits entre balises. Autrement dit, nous voulons une fonction qui prend en entrée un texte et une balise et nous renvoie tous les passages codés. Pour cela, nous allons utiliser les expressions régulières abordées dans le chapitre 2 : elles permettent de repérer dans un texte tous les passages qui ont une forme particulière. La bibliothèque correspondante est `regex`.

```
import regex as re

def extraire_balise(texte,balise):
    debut = '<'+balise+'>'
    fin = '</'+balise+'>'
    texte = texte.replace("\n", " ")
    passages = re.findall(debut+'.*?'+fin,texte)
    return passages
```

Ce code :

- ➊ importe la bibliothèque `regex`;
- ➋ définit une fonction qui construit les éléments de début et de fin à partir de la syntaxe que nous avons définie et enlève les sauts de ligne dans le texte qui coupent sinon la détection du motif;
- ➌ utilise la fonction `findall` pour trouver tous les passages qui sont encadrés par la valeur de `début` et de `fin`. L'expression régulière s'explique de la manière suivante : . signifie n'importe quel caractère, \* signifie autant de fois qu'il le faut et ? indique de s'arrêter dès que l'on peut.

### ?

### Exercice

Modifiez la fonction pour que les balises n'apparaissent pas dans les extraits.

Une manière est d'utiliser la fonction `replace`. Une autre est d'ajouter deux parenthèses bien placées : `passages = re.findall(debut+"(.*?)"+fin, texte)`.

Vous avez maintenant deux fonctions qui permettent de récupérer les informations que vous avez croisées. Rien ne vous arrête maintenant d'en écrire d'autres pour les besoins spécifiques que vous pouvez avoir :

- ➊ lister toutes les balises présentes dans un texte ;
- ➋ tester la présence ou l'absence d'une balise ;
- ➌ regardez si des balises sont comprises dans d'autres balises.

### ➊ Exercice

**Créez une fonction qui liste toutes les balises présentes dans un texte.**

```
def lister_balises(texte):
    balises = re.findall("<.*?>", texte.replace("\n", " "))
    return set(balises)
```

Cela arrive fréquemment de faire des erreurs dans le codage des entretiens : balises mal écrites, oubli d'une balise de fin. Une bonne idée est de faire une petite fonction qui compte la présence des balises et vérifie qu'elles sont bien en nombre pair.

## Analyser les données

Vous avez tous les outils pour faire le lien entre vos entretiens qualitatifs et de l'analyse quantitative. Les utilisations vont dépendre de vos besoins.

Une première utilisation est de recourir à la recherche : vous pouvez facilement construire un corpus de tous vos entretiens et filtrer ceux qui correspondent aux conditions de votre recherche (présence de mots, présence ou absence de balise). Pour ainsi dire, cela joue un rôle de moteur de recherche.

Par exemple, si nous voulons uniquement les entretiens qui contiennent à la fois des balises « comparaison » et des balises « différence » :

```
import os

liste_docs = [i for i in os.listdir("data") if "docx" in i]
corpus = {i:extraire_texte("data/"+i) for i in liste_docs}
corpus_f = [i for i in corpus if ("<difference>" in corpus[i])
            and ("<comparaison>" in corpus[i])]
print(corpus_f)
```

```
['entretien.docx']
```

Ce code :

- ➊ importe la bibliothèque `os` pour pouvoir lister le contenu du dossier et ne garder que les documents qui contiennent l'extension `.docx`;
- ➋ construit un dictionnaire `corpus` avec comme clé le nom du fichier et comme contenu le texte extrait avec la fonction précédente `extraire_texte`;
- ➌ sélectionne les clés des éléments qui contiennent à la fois la balise `<difference>` et la balise `<comparaison>` et l'affiche (il y a un unique élément dans cet exemple, mais vous pouvez en ajouter d'autres).

Une dernière étape qui permet de faciliter ce traitement des données est de rassembler toutes ces informations dans un tableau *Pandas*, avec en index le nom du fichier (par exemple) et autant de colonnes que d'informations à étudier (présence des balises, proportion de mots, etc). Pour notre très petit exemple d'un unique fichier (mais qui fonctionne pour plus de fichiers) :

```
import pandas as pd

corpus = [[i,extraire_texte("data/"+i)] for i in liste_docs]
corpus = pd.DataFrame(corpus)
corpus.columns = ["fichier","texte"]
corpus = corpus.set_index("fichier")
corpus["<difference>"] = corpus["texte"].str.
    ↪contains("<difference>")
corpus["<comparaison>"] = corpus["texte"].str.
    ↪contains("<comparaison>")
corpus.head()
```

	texte	<difference>	<comparaison>
fichier			
entretie...	Début e...	True	True

Ce code commence par construire une liste avec le nom du fichier et le contenu, le passe dans un tableau *Pandas*, renomme le nom des colonnes, met le nom des fichiers en index puis construit deux nouvelles colonnes qui indiquent pour chaque ligne si le texte contient la balise correspondante avec la méthode `str.contains` pour vérifier la présence ou l'absence d'un mot.

### ?

### Exercice

**Rajoutez une colonne qui indique le nombre de mots de l'entretien et une autre pour indiquer le nombre de balises présentes.**

Utilisez la fonction `split` pour couper le texte et `len` pour compter les mots. Et `count` pour les occurrences.

Nous étions donc partis d’entretiens sous format Word et nous avons maintenant un tableau qui permet d’accéder au contenu, aux informations codées dans les entretiens et à différentes dimensions supplémentaires que nous pouvons rajouter progressivement. D’une certaine manière, nous avons construit notre petite base de données.

La méthode utilisée ici est adaptée pour des données de taille raisonnable (de quelques dizaines à quelques milliers de textes). Au-delà, il est préférable de faire appel à d’autres méthodes mobilisant des bases de données.

Cette approche a de nombreux avantages : flexible, incrémentale, vous pouvez facilement naviguer dans vos données. Elle tire parti du caractère interactif de Python et permet d’accompagner votre processus de réflexion. Avec un peu de dextérité, elle vous permet de mieux circuler dans votre matériau d’enquête. Elle demande toutefois d’écrire un peu de code. Si nous avons fait le choix de vous faire écrire ces fonctions plutôt que de les mettre dans une bibliothèque, c’est pour insister sur la souplesse et l’adaptation aux cas particuliers qui font la force d’une telle approche. Maîtriser Python permet ainsi de s’adapter à la diversité des formats. Un bon exemple de cette flexibilité est le traitement de données issues d’internet.

### 9.1.2 Récupérer des données sur internet

#### Les données d’internet

Une autre source d’information utile pour les SHS provient d’internet. La majorité de cette information correspond à du texte mais mis en forme pour internet. Les images sont une autre source d’information, mais plus difficile à traiter<sup>4</sup>. Cela signifie que le format est d’abord destiné à un navigateur et contient de l’information de formatage. Cette information n’est pas destinée à être lue par un humain. Pour pouvoir effectuer un traitement automatisé, il est important d’extraire l’information utile.

Certaines de ces données sont directement accessibles sur des sites. D’autres sont protégées, qu’elles soient payantes ou spécifiques à certaines organisations. Dans tous les cas, la collecte, l’utilisation et la diffusion d’informations disponibles sur internet peuvent être soumises à différentes règles. Certaines données sont libres

---

4. Traiter des images en SHS n’est pas une évidence, au-delà de quelques approches dites « visuelles ». D’un point de vue informatique, une image est une information comme une autre, qu’il est possible de traiter. Cependant, peu d’utilisations concrètes existent au-delà des interprétations faites par les humains. Bien que le traitement automatique d’image progresse, ce n’est pas encore un outil très intéressant pour les SHS. La programmation vous sera utile pour automatiser leur collecte, mais peu pour les analyser.

d'accès et d'utilisation. D'autres peuvent être consultées mais pas collectées automatiquement par un script. Dans le doute, renseignez-vous sur l'usage de ces données.

Nous présentons dans cette partie l'intégration des données d'Europresse<sup>5</sup> dans un script pour illustrer comment ponctuellement et pour un usage privé vous pouvez créer un script facilitant l'utilisation de données collectées sur internet. Europresse est un exemple de ces informations accessibles par internet permettant de constituer un corpus, mais dont l'automatisation d'une partie de la procédure simplifie la vie pour constituer des revues de presse.

Nous prenons cet exemple pour deux raisons : l'analyse des médias est une étape de beaucoup de travaux en SHS et il nous permet de présenter l'utilisation d'une bibliothèque permettant de traiter le format HTML obtenu quand on récupère des pages internet. Dans le chapitre 3, nous avons vu la bibliothèque *Requests* qui permet de récupérer des pages internet : leur traitement suit ensuite une logique similaire à celle présentée ci-dessous.

## Traiter des données en format HTML

L'interface d'Europresse, mais cela pourrait être un autre site d'information, permet de télécharger une page internet avec les articles nous intéressent. Comment utiliser les données d'une page internet ?

### ❶ Information

#### Récupérer les informations sous un format HTML

Pour ceux qui pourraient être amenés à utiliser des données Europresse, une petite manipulation est nécessaire pour se faciliter la vie. La version classique de l'interface d'Europresse (à sélectionner dans le menu) permet de rechercher des articles de presse avec certains mots-clés, journaux et dates et d'afficher le résultat. Il est alors possible de sauvegarder les articles sélectionnés (icône sauvegarder) dans un fichier HTML. De la même manière, si vous récupérez un article sur le site d'un journal, le format sera aussi du HTML. Nous sélectionnons les 50 premiers articles sur une recherche avec le mot-clé *coronavirus* dans la presse française nationale. Nous sauvegardons ce fichier.

Pour d'autres pages internet, un clic droit et sauvegarder permet d'avoir la page HTML.

5. Europresse est une base de données d'informations accessible sur abonnement par Internet qui compose une archive de presse. Son usage est donc payant. De nombreuses universités et organismes de recherche ont des abonnements qui permettent à leurs membres d'accéder à Europresse pour faire de la veille documentaire. Pour autant, cela ne signifie pas que nous pouvons faire ce que nous voulons des données collectées.

Vous êtes donc en face d'un fichier HTML et vous voulez pouvoir le nettoyer pour obtenir l'information que vous souhaitez. Pour cela, il faut avoir une petite idée de comment est organisé un tel fichier. Voici à quoi ressemble le contenu de ce qu'Europresse vous permet de sauvegarder (on coupe une partie au niveau des [...] sinon le fichier est trop long).

```
<!DOCTYPE html> <html> <head> <meta charset="utf-8"/><title>
</title><style type="text/css">html,body{font-family:Arial
[...]
<body> <article><header> <span
class="" sourcecode="TR"></span>
<div class="rdp_DocPublicationName">
<span class="DocPublicationName"> La Tribune (France),
no. 6880 </span> </div> <div class="rdp_DocHeader">
<span class="DocHeader"> Focus, samedi 21 mars 2020 4592 mots, p. 32
</span> </div> <div class="titreArticle">
<p class="titreArticleVisu rdp_articletitle">
L'UE suspend les règles de discipline budgétaire, une mesure inédite
</p> </div> <div class="docAuthors">Latribune.fr</div>
<b> <p>Tout ce qu'il faut savoir sur
l'épidémie de <mark>coronavirus</mark>, heure par heure, en ce
vendredi 20 mars.</p> </b> </header> <section>
<div class="DocText clearfix"> <div class='doc0currContainer'>
<p> - 18h50 - En un jour, l'Italie recense 627 décès et
franchit un nouveau record</p> <p> - 18h45 - Au Royaume-Uni, les
pubs vont fermer dès ce soir </p>
[...]
```

Ce fichier HTML contient trois types d'informations : des informations de structure qui organisent le document, des informations de mise en forme et du contenu à proprement parler. L'ensemble des éléments de structure sont indiqués à travers des balises, c'est-à-dire ces mots clés entre < et >. Ces balises organisent le document. Elles permettent aussi de retrouver des éléments.

## Information

### Structure d'un fichier HTML

Sans rentrer dans tous les détails, une page HTML est contenue dans une balise `<html></html>`. Elle est ensuite décomposée dans une première partie contenue entre `<head>` et `</head>` qui donne des informations de mise en forme de la page, sans contenu. Le contenu proprement dit est généralement

entre les balises `<body>` et `</body>`. Une bonne manière de se repérer dans une telle page est de rechercher où sont les informations que vous souhaitez avec l'option rechercher de votre éditeur de texte ou de votre navigateur après avoir fait un clic droit et choisir d'afficher la source du document.

Les balises que nous rencontrons souvent sont :

- ⊕ `<div></div>` qui délimite une partie d'information ;
- ⊕ `<p></p>` qui délimite un paragraphe ;
- ⊕ `<li></li>` qui délimite une liste.

Ces balises ont aussi différents paramètres, elles peuvent par exemple appartenir à une classe particulière qui indique au navigateur comment les afficher (en couleur, etc.) .

Dans le cas qui nous intéresse, nous regardons la structure du document et nous voyons que chaque article est contenu entre les balises `<article></article>` qui permettent de les délimiter. Le titre de l'article est entre la balise `<div class="titreArticle"></div>`. Ensuite, chaque article est relié à un journal dont l'information est dans la balise `<span class="DocPublicationName"></span>`. L'information sur la date est dans la balise `<span class="DocHeader"></span>`.

Il est possible de faire un script avec les outils que nous connaissons déjà : les expressions régulières. Si vous cherchez un passage spécifique, cela peut être la solution la plus rapide. Cependant, une bibliothèque dédiée existe pour manipuler des pages HTML et les balises : *Beautifulsoup*, que vous pouvez installer avec le nom `bs4` (`pip install bs4`).

### ?

### Exercice

**Identifiez la balise qui délimite le contenu du texte proprement dit.**

La balise `<section>` délimite le contenu.

Comme pour d'autres bibliothèques rencontrées, elle rend disponible un nouvel objet spécifique `Beautifulfoup` qui permet de lire le fichier HTML et de facilement le manipuler, comme faire une recherche de balise. Pour ouvrir le fichier HTML et le mettre sous la forme d'un objet `BeautifulSoup`, l'opération est simple :

```
import bs4

file = "data/fichier_europresse.HTML"
with open(file, "r", encoding="utf8") as f:
    html = bs4.BeautifulSoup(f, "lxml")
type(html)
```

### bs4.BeautifulSoup

Ce code charge le fichier dans un objet `BeautifulSoup` et affiche son type. Notez que l'objet `BeautifulSoup` prend comme argument soit le lien vers le fichier (ici `f`) soit le contenu lui-même. Il est maintenant possible d'accéder aux différents éléments de cette page. Nous allons utiliser la méthode `findAll` qui permet de récupérer l'ensemble d'un type de balise. Pour récupérer toutes les balises de type `<article>` (il y a 50 articles dans la page téléchargée) :

```
corpus = html.findAll("article")
print(len(corpus))
```

50

La variable `corpus` contient le contenu correspondant aux 50 balises `<article>`. Chaque occurrence de balise est sous la forme d'un objet `BeautifulSoup` que l'on peut continuer à manipuler. Il ne reste plus qu'à écrire une fonction qui prend une de ces balises et extrait l'information qui nous intéresse, basée sur le repérage que nous avons fait plus haut.

Il peut arriver que notre programme rencontre des erreurs. Par exemple, quand nous voulons faire une opération impossible, car il manque un élément. Cela est particulièrement vrai pour des données réelles souvent incomplètes. La gestion des exceptions en Python avec les mots-clés `try` (essaye) et `except` (en cas d'exception) vue dans le chapitre 2 permet d'indiquer l'opération à réaliser si jamais le code rencontre une erreur.

Dans ce cas, nous utilisons les exceptions pour gérer l'absence d'éléments. S'il est préférable d'éviter ce genre de code, qui ne fait pas très sérieux aux yeux des vrais programmeurs, il est particulièrement adapté ici pour traiter des données « réelles » qui peuvent avoir des défauts, des éléments qui manquent, etc. Dans le cas où le code fait une erreur, cela évite d'arrêter l'ensemble du programme et de plutôt considérer que l'information n'existe pas et continuer.

```
def extraire_contenu(article):
    try:
        titre = article.find("div",
                             {"class": "titreArticle"}).text
    except:
        titre = None

    try:
        publication = article.find("span",
                                    {"class": "DocPublicationName"}).text
    except:
```

```

publication = None

try:
    text = article.find("div",
                         {"class": "DocText clearfix"}).text
except:
    text = None

return [titre, publication, text]

```

Cette fonction prend en entrée un objet *BeautifulSoup* correspondant à un article. Elle va ensuite essayer de chercher des éléments particuliers que nous avons repérés plus haut avec la méthode `find`, qui précise la balise à chercher mais aussi la classe si nécessaire pour préciser la recherche et ensuite cherche à extraire l'attribut texte (le `.text`). Si cette opération est impossible, et qu'il y a une erreur, on considère que l'information n'existe pas et on lui donne la valeur nulle. La fonction renvoie enfin les trois informations recherchées. Et voilà, vous avez en quelques lignes une fonction qui permet d'extraire les données par article. Il ne reste plus qu'à tout mettre ensemble pour créer un tableau *Pandas* avec les informations. Voici une idée de code complet :

```

file = "data/fichier_europresse.HTML"
with open(file, "r", encoding="utf8") as f:
    html = bs4.BeautifulSoup(f, "lxml")

corpus = html.find_all("article")
donnees = [extraire_contenu(article) for article in corpus]
donnees = pd.DataFrame(donnees,
                        columns = ["titre", "journal", "texte"])
donnees.head()

```

	titre	journal	texte
0	L'UE s...	La Tri...	- 18..
1	L'UE v...	La Tri...	- 18..
2	Le co...	La Tri...	Alors...
3	Corona...	La Tri...	- 18..
4	Plus d...	La Tri...	- 17..

Et voici vos données chargées. Vous remarquerez si vous regardez en détail le tableau que le contenu n'est pas très bien mis en forme : par exemple, on n'a pas vraiment le nom du journal bien séparé mais un ensemble d'informations combinées. Du recodage est alors nécessaire. Mais maintenant, vous avez tous les outils à votre disposition pour le faire vous-même.

Dans ce cas, nous sommes passés par le portail internet d'Europresse pour collecter une page web et ensuite la traiter. Cependant, de nombreuses autres données peuvent être récoltées à partir d'internet sans passer par une page internet. L'idée est alors similaire : récupérer l'information avec *Requests*, potentiellement la stocker sur l'ordinateur, puis la transformer dans un objet *BeautifulSoup*. Explorer cette page pour identifier où est l'information qui vous intéresse, puis utiliser les balises pour la récupérer. Simple non ?

Bien entendu, le passage consistant à explorer la page pour comprendre sa structure peut prendre un peu de temps, mais souvent vous savez ce que vous cherchez donc vous pouvez vous concentrer sur ces éléments. Si vous avez de la chance, l'information que vous cherchez peut être récupérée autrement qu'en interprétenant une page internet grâce à une interface dédiée.

### ?

### Exercice

#### Faites une fonction de nettoyage pour récupérer l'information du journal.

Utilisez la méthode pour découper les textes afin de prendre le nom du journal avant la virgule : `.split(",") [0]`.

### 9.1.3 Utiliser des API : Twitter, Google et les autres

Pour le moment, nous avons croisé des informations sous la forme de pages internet destinées à être consultées par un navigateur pour une lecture par un être humain. Il existe de nombreuses autres sources de données sur internet qui ne se résument pas à des pages internet, tout en étant accessibles par internet. Prenons trois exemples parmi les nombreux existants : Twitter, *Open Street Maps* et *Google Trends*. Ces portails spécifiques se sont largement développés avec l'internet des applications<sup>6</sup>.

Suivant vos centres d'intérêt, vous pouvez vouloir suivre les discussions autour d'un sujet sur Twitter, l'évolution médiatique d'un thème sur *Google Trends*, ou à géolocaliser des adresses avec *Open Street Map*. Chacun de ces services a plusieurs interfaces pour échanger les données. Une est souvent destinée aux utilisateurs sur internet. Et souvent une autre est destinée au programmeurs pour qu'ils puissent développer des logiciels utilisant ces données. Cette interface est habituellement appelée interface de programmation d'application ou API pour *Application Programming Interface*.

---

6. Les applications sont des services dédiés contenus dans des « jardins fermés » accessibles par internet. Elles servent à des services particuliers, suivant les conditions fixées par leurs créateurs. Entre internet et le logiciel, elles transforment notre rapport au numérique. Si le sujet vous intéresse, jetez un coup d'œil au blog d'Olivier Ertzscheid [www.affordance.info](http://www.affordance.info).

*gramming Interface*. Dit simplement, c'est une manière d'échanger les données avec l'interface. Cela permet au programme d'interroger les services et de recevoir des données sous un format facilement utilisable.

Très souvent, des développeurs de la communauté Python ou même les entreprises qui mettent à disposition ces services développent une bibliothèque Python pour gérer les échanges avec ces interfaces. *Open Street Maps* a une bibliothèque *Geocoder* qui permet de faire des requêtes sur des adresses, Twitter a plusieurs bibliothèques officielles et non officielles, comme *Tweepy* ou *GetOldTweets*, et *Google Trends* a plusieurs bibliothèques non officielles comme *PyTrends*.

Dans certains cas, il est nécessaire de demander l'autorisation d'utiliser les données pour obtenir un mot de passe spécifique. C'est le cas pour les bibliothèques Twitter qui nécessitent de d'abord créer un compte développeur chez Twitter (cela n'est pas particulièrement compliqué, ne vous inquiétez pas). Dans certains cas, il y a des limites sur le nombre d'utilisations possibles. Et dans tous les cas, l'usage des données est réglementé et il est important de jeter un coup d'œil aux conditions d'utilisation. Nous n'allons pas rentrer en détail dans l'utilisation de chaque bibliothèque et nous limiter à un exemple précis. À vous ensuite de regarder comment les utiliser au mieux par rapport à vos besoins.

Tout d'abord, utilisons *Google Trends* pour connaître d'où vient l'intensité des requêtes de Google sur certains mots-clés. Nous pouvons vouloir récupérer l'intensité de requêtes sur un mot-clé particulier ou l'actualité générale (ces notions sont définies par Google). Récupérons les requêtes sur le terme « cancer » avec *PyTrends* (pensez à l'installer).

```
import pandas as pd
from pytrends.request import TrendReq

pytrends = TrendReq(geo="FR", tz=360)
pytrends.build_payload(kw_list=["cancer"])
requetes = pytrends.related_queries()

requetes["cancer"]["top"][0:10]
```

	query	value
0	le cancer	100
1	cancer ...	74
2	cancer ...	60
3	cancer ...	45
4	cancer ...	44
5	symptome	41
6	symptom...	40
7	cancer ...	34

8	prostate	33
9	colon c...	33

Ce code :

- ⊕ charge la bibliothèque *Pandas* et un objet de la bibliothèque *PyTrends* ;
- ⊕ initialise l’interface avec *Google Trends* en précisant que la zone géographique est la zone française ;
- ⊕ crée la requête avec un seul mot-clé, « cancer » ;
- ⊕ récupère les requêtes associées dans la variable `requetes` (le type de ces données est un dictionnaire de dictionnaires) ;
- ⊕ affiche les 10 premières requêtes liées au « cancer » et qui sont les plus fréquentes (« top », sinon aussi celles en augmentation, « rising »).

C'est un exemple de requêtes. D'autres sont possibles, pour cela il faut se plonger dans des tutoriaux disponibles sur internet ou directement dans la documentation de la bibliothèque.

### ?

### Exercice

Regardez l'aide de la fonction `interest_over_time` et utilisez-la.

```
d = pytrends.interest_over_time()  
d[ "cancer" ].plot()
```

Comme *Pandas* gère automatiquement les dates, nous avons directement un joli graphique.

Deuxième exemple, nous souhaitons géolocaliser une adresse que nous avons récupérée. Pour cela, nous pouvons utiliser la bibliothèque *Geocoder* qui permet d’interroger la base de données d'*Open Street Maps* et de récupérer les informations, en particulier les coordonnées GPS qui peuvent permettre de situer l’adresse sur une carte. Cette bibliothèque est très simple à utiliser :

```
import geocoder  
g = geocoder.osm('Cours Julien, Marseille')  
infos = g.json  
print(infos[ "address" ])
```

Cours Julien, Noailles, 6e Arrondissement, Marseille,  
Bouches-du-Rhône, Provence-Alpes-Côte d'Azur, France, 13001

Nous affichons l'information de l'adresse contenue dans l'ensemble des informations retournées par la requête sous un format JSON<sup>7</sup>, stockées dans la variable `infos`. Vous pouvez circuler dans un tableau JSON comme dans un dictionnaire, à partir des clés. JSON est un format de fichier permettant de stocker de l'information sous une forme organisée (des entrées comme un dictionnaire) souvent utilisé pour les échanges sur internet.

### ?

### Exercice

**Regardez les informations contenues dans les données JSON récupérées puis trouvez la latitude/longitude de l'adresse où vous vous trouvez.**

Affichez avec `print` le contenu de la variable `infos`.

Enfin, un dernier exemple pour récupérer des données Twitter. Plusieurs bibliothèques existent construites à partir de l'API Twitter. Elles nécessitent de vous connecter à l'espace développeur de Twitter. C'est le cas pour la bibliothèque *Tweepy*.

Une fois ces codes obtenus, vous pouvez utiliser la bibliothèque pour récupérer des informations sur Twitter à partir d'une recherche de mots-clés, soit en remontant dans le temps (avec une limite de quelques jours), soit en continu avec une limite maximale de connexions.

Mettre en place un script qui fait de la veille sur Twitter dépasse l'ambition de ce chapitre. Cependant plusieurs exemples sont disponibles sur internet et vous avez maintenant assez d'éléments pour les comprendre et les adapter.

Nous allons présenter les étapes pour mettre en place une recherche de tweets. Une telle situation se rencontre si par exemple vous souhaitez faire de la veille sur un sujet, ou récupérer l'ensemble des informations sur un thème.

Pour pouvoir accéder à Twitter, il faut demander un accès développeur à <https://developer.twitter.com/en/apps> et donc initialement avoir un compte Twitter. Cela permet d'obtenir des codes autorisant la connexion : la clé API et le jeton d'accès<sup>8</sup>.

---

7. JavaScript Object Notation (JSON) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.

8. Si vous rencontrez des difficultés, vous pouvez consulter des tutoriaux comme <https://cran.r-project.org/web/packages/rtweet/vignettes/auth.html>.

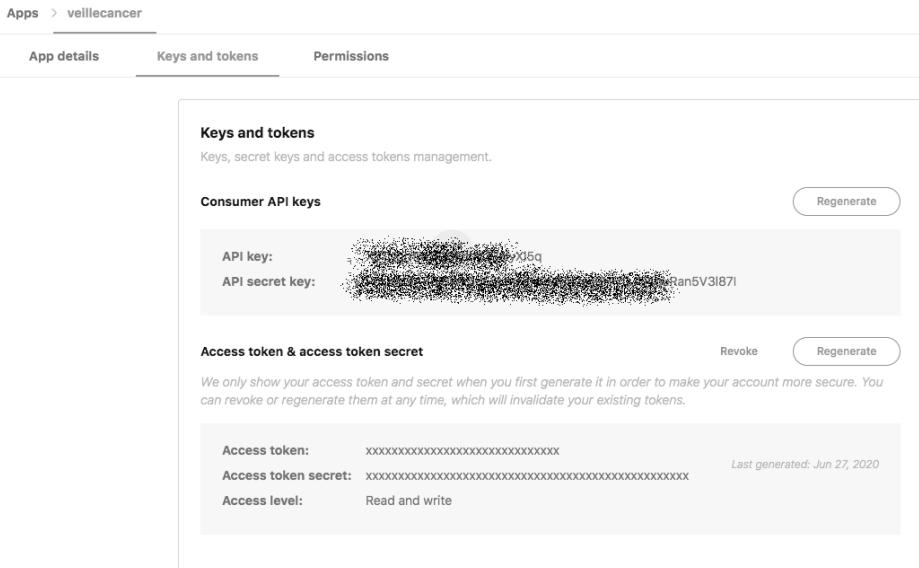


Fig. 9.1 – Page de l’API de Twitter pour obtenir les codes développeur

Ces codes sont à renseigner dans le script pour permettre la connexion entre votre ordinateur et l’interface Twitter. Dans les scripts suivants, nous avons défini un dictionnaire avec ces codes qui a la forme suivante `codes = {'api_key':XXXXXX, 'api_secret_key':XXXXXX, 'access_token':XXXXXX, access_token_secret=XXXXXX}`. Si nous voulons faire une recherche sur un mot-clé à partir d’une date donnée et récupérer les 5 premiers tweets, voici le code qui nécessite plusieurs étapes.

```
import tweepy as tw
import json

# Indique les codes Twitter (à remplacer par les vôtres)
api_key = codes['api_key']
api_secret_key = codes['api_secret_key']
access_token = codes['access_token']
access_token_secret = codes['access_token_secret']

# Initialise la connexion à l'API
auth = tw.OAuthHandler(api_key, api_secret_key)
auth.set_access_token(access_token, access_token_secret)
api = tw.API(auth, wait_on_rate_limit=True)
```

```
# Réalise la recherche
mot = "#cancer"
date = "2020-06-01"
tweets = tw.Cursor(api.search, q=mot,
                    lang="fr", since=date).items(10)

# Affiche le résultat
tweets = [t._json for t in tweets]
print(json.dumps(tweets[0], indent=2, sort_keys=True)[1000:2000])

{
    "retweeted": false,
    "retweeted_status": {
        "contributors": null,
        "coordinates": null,
        "created_at": "Sun Oct 11 15:00:06 +0000 2020",
        "entities": {
            "hashtags": [
                {
                    "indices": [
                        9,
                        16
                    ],
                    "text": "Cancer"
                }
            ],
            "symbols": [],
            "urls": [],
            "user_mentions": []
        },
        "favorite_count": 73,
        "favorited": false,
        "geo": null,
        "id": 1315306185264955393,
        "id_str": "1315306185264955393",
        "in_reply_to_screen_name": null,
        "in_reply_to_status_id": null,
        "in_reply_to_status_id_str": null,
        "in_reply_to_user_id": null,
        "in_reply_to_user_id_str": null,
        "is_quote_status": false,
        "lang": "fr",
        "metadata": {
            "iso_language_code": "fr",

```

```
    "result_type": "recent"
},
"place": null,
"retweet_count": 24,
"retweeted": false,
```

Ce code :

- ♣ importe la bibliothèque ;
- ♣ déclare les 4 informations nécessaires pour se connecter à l'API de Twitter. Comme nous n'allons pas donner nos informations, nous les avons chargées de manière cachée dans le dictionnaire `code`, à vous de les remplacer par vos valeurs récupérés sur Twitter ;
- ♣ crée un objet d'authentification `auth` avec le code utilisateur ;
- ♣ valide l'accès avec les codes du token ;
- ♣ crée l'objet API proprement dit avec l'authentification ;
- ♣ définit le mot à rechercher et une date ;
- ♣ fait la recherche à partir de l'API avec la fonction `Cursor` qui utilise l'API pour aller chercher le mot (`q=` pour *query*) dans les tweets français à partir de la date et sélectionne 10 éléments avec la méthode `items` ;
- ♣ passe les tweets récupérés sous la forme d'une liste contenant leur mise en forme dans un dictionnaire grâce à l'attribut `_json` ;
- ♣ affiche une partie du premier tweet de manière lisible avec une fonction de la bibliothèque `Json`.

De nombreux éléments peuvent être modifiés. Déjà dans la recherche, vous pouvez indiquer plusieurs mots, ou des options spécifiques définies par l'API Twitter comme `#cancer -filter:retweets` pour ne pas rechercher dans les retweets. Ensuite il est possible dans la recherche de paramétrier plusieurs types d'informations comme la dates des tweets ou la langue. Par exemple, il est possible de spécifier le numéro d'identifiant limite des tweets récupérés avec `since_id` ou `max_id`.

Nous n'avons vu que la manière de faire une recherche, avec l'entité `api.search`. Il est en fait possible de faire toutes les manipulations que vous faites d'habitude avec un compte Twitter, comme mettre à jour un statut, récupérer vos propres Tweets avec `api.home_timeline`, récupérer les informations d'un compte avec `api.user_timeline`, etc.

Pour récupérer un tweet particulier avec son identifiant, vous pouvez utiliser la ligne `api.get_status(1276847867878088705)`.

Bien sûr, chacune de ces opérations est limitée par Twitter. Ainsi, il n'est pas possible de récupérer plus de 100 tweets à la fois. Mais vous pouvez récupérer des tweets par groupe et avec un peu d'astuce réussir à collecter ce qui vous intéresse<sup>9</sup>.

---

9. Vous pouvez connaître l'état actuel du nombre de requêtes restantes pour différents types de demandes avec la ligne `api.rate_limit_status()` qui retourne un contenu JSON avec de nombreux paramètres.

## ?

### Exercice

#### Mettez en forme l'information des tweets dans un tableau *Pandas*.

À partir de la liste des tweets récupérés, construisez un tableau avec les principales informations (texte, utilisateur, nombre de retweets) en regardant le contenu du JSON, puis utilisez l'objet *DataFrame* pour construire un tableau *Pandas*.

Il existe d'autres bibliothèques pour interagir avec Twitter. Cependant, une grande partie sont non-officielles, c'est-à-dire qu'elles cherchent à exploiter différentes caractéristiques de l'API pour parvenir à leur fin (comme récupérer des tweets). À chaque fois que Twitter décide de modifier son API, ou de mieux la protéger, ces bibliothèques peuvent arrêter de fonctionner. Vous pouvez donc souvent vous retrouver avec des bibliothèques existantes mais qui ne fonctionnent plus avec la version actuelle de Twitter. Ne soyez donc pas surpris.

## !

### Information

#### Autre bibliothèque

La bibliothèque non officielle *GetOldTweets3* permet de récupérer les tweets affichés sur l'interface internet. Cependant, à l'heure où nous écrivons ces lignes, elle ne fonctionne plus suite à un changement de l'API de Twitter. La communauté des contributeurs cherche une solution et il est probable que bientôt elle fonctionnera de nouveau.

Ainsi, si vous voulez récupérer tous les tweets de Matthias pour suivre l'actualité Python, vous pouvez faire un petit script :

```
import GetOldTweets3 as got

tweetCriteria = got.manager.TweetCriteria()\
    .setUsername("Mbussonn")\
    .setMaxTweets(10)

tweets = got.manager.TweetManager.getTweets(tweetCriteria)
un_tweet = tweets[0]
print([un_tweet.author_id, un_tweet.username, un_tweet.date,
       un_tweet.text, un_tweet.retweets, un_tweet.favorites,
       un_tweet.permalink])
```

Ce code charge la bibliothèque puis paramètre l'objet qui servira à se connecter à Twitter en précisant le nom du compte à regarder et le nombre de tweets à collecter. Ensuite, il cherche ces tweets, puis affiche certains des attributs du premier tweet.

Nous avons fait un tour de quelques usages permettant de collecter des données que nous n'avons pas encore eu l'occasion de manipuler. Souvent, des bibliothèques existent pour manipuler ces données particulières, voire les collecter quand elles sont reliées à des services spécifiques. Suivant vos besoins, il peut alors être intéressant de prendre le temps de comprendre comment fonctionnent Twitter, la structure d'un tweet, ou encore ce que signifie la tendance de *Google Trends*, pour ensuite l'analyser. Justement, passons à l'analyse de ces textes.

## 9.2 Analyse de corpus textuels

Les données textuelles nécessitent des traitements spécifiques pour pouvoir être traitées quantitativement. Nous avons vu au début de ce chapitre une stratégie pour lier l'interprétation humaine aux statistiques. Dans d'autres cas, en particulier les grands volumes d'information, il est intéressant de procéder à une analyse statistique du texte.

### 9.2.1 Traiter des textes pour en dégager de l'information

L'analyse des données textuelles est un domaine de recherche à part entière en lien avec la linguistique et l'informatique. Les approches structurelles historiques ont récemment cédé leur place au traitement massif permis par l'apprentissage automatique et les réseaux de neurones. Nous allons balayer quelques traitements qui peuvent vous être utiles, en nous concentrant sur la logique générale.

Avant d'entrer dans les traitements proprement dits, attirons votre attention sur l'utilisation de ces outils en SHS. Beaucoup d'entre eux ont historiquement été développés pour rendre compte de la forme des textes, dans une logique d'étude littéraire ou linguistique. Des livres récents existent si vous voulez aller plus loin sur ces sujets<sup>10</sup>. Le passage d'une analyse des textes à d'autres sujets n'est pas si évident.

À notre avis, la limite centrale de l'analyse textuelle pour les SHS, en particulier pour les sciences sociales, est la difficulté d'obtenir des résultats interprétables dans le cadre d'une réflexion plus générale. Bien entendu, beaucoup de méthodes permettent d'obtenir des classifications, des nuages de mots, ou encore des résumés automatiques de textes. Mais deux situations se rencontrent souvent : la première est que ces résultats sont peu interprétables ; la seconde est qu'ils ne font que redoubler ce qu'une connaissance pratique du domaine permet d'obtenir.

Au cœur de ce paradoxe se trouve le problème de la sémantique des textes : l'ordinateur fait un traitement statistique et toutes les interprétations de sens reposent en fait sur ces associations statistiques. À moins d'avoir de très grands corpus ou

---

10. Par exemple, si vous voulez un peu creuser ce domaine, nous ne pouvons que trop vous conseiller Lebart *et al.* (2019) et pour des applications avec R Bécue-Bertaut & Lebart (2018).

de poser des questions assez générales, par exemple le caractère genré des textes en analysant l'accord des participes passés. Il est généralement difficile de calculer le sens des textes.

L'enjeu est alors d'intégrer de manière pertinente l'analyse textuelle dans la réflexion en SHS. Cela dépend bien entendu de votre objet. À moins que vos objectifs se limitent à rendre compte d'un matériau textuel, l'articulation avec vos réflexions va être importante. Cela peut se traduire par une attention particulière à une partie de ces textes ou des évolutions chronologiques. La priorité doit être donnée à la pertinence des traitements plutôt que leur complexité.

De nombreux outils permettent de produire des analyses à partir de données textuelles. Pour autant, la classification des textes, l'identification des mondes lexicaux ou des réseaux d'entités posent la même question : comment ont-ils été construits ? Comment justifier les choix qui ont été faits ? Très souvent, il existe de nombreux paramètres à régler de manière plus ou moins arbitraire et dont la modification change les résultats. Pour éviter les boîtes noires, privilégiez les traitements simples que vous contrôlez, ou prenez le temps de comprendre la théorie sous-jacente aux outils que vous utilisez.

Voici quelques questions que vous pourriez vous poser à propos d'un corpus de texte :

- ♣ est-ce que les textes sont similaires ou différents ?
- ♣ de quoi parlent les textes ?
- ♣ quels sont les thèmes présents dans les textes ?
- ♣ quelles sont les entités nommées (noms communs et noms propres) ?

Dans tous les cas, les étapes de l'analyse sont généralement les suivantes :

1. constituer un corpus ;
2. mettre en forme le texte en unités de base ;
3. sélectionner les éléments pertinents ;
4. appliquer un traitement statistique.

## Constituer un corpus

Le corpus représente le « monde » sur lequel nous allons travailler. Sa composition est cruciale pour la suite des analyses. Suivant vos besoins, il peut être déjà constitué ou nécessiter un travail à part entière d'extraction et de mise en forme.

Pour cette partie, nous allons constituer un corpus en utilisant ce que nous avons appris dans la partie précédente sur les API. Nous allons nous interroger sur la manière dont on parle du cancer dans une encyclopédie en ligne comme *Wikipédia*<sup>11</sup>. Ces informations sont libres et disponibles sur internet.

---

11. Wikipédia correspond à un corpus particulier disposant d'une API, qui a donné lieu à de nombreuses réflexions à la fois conceptuelles et méthodologiques, comme dans Rogers (2013).

La question que nous nous posons ici pour Wikipédia peut se poser pour d'autres sources : existe-t-il une manière de facilement récupérer l'information ? Comme souvent, une solution est de collecter les pages une à une comme utilisateur. Cependant, une petite recherche nous permet de trouver une bibliothèque Python *Wikipedia*. Instalions-là et utilisons sa fonction de recherche pour trouver toutes les pages qui concernent le cancer.

```
import wikipedia as wp
wp.set_lang("fr")
pages = wp.search("cancer", results=100)
pages[0:8]
```

```
['Cancer',
 'Cancer (astrologie)',
 'Cancer du sein',
 'Cancer du poumon',
 'Cancer du pancréas',
 'Cancer colorectal',
 'Tropique du Cancer',
 "Cancer de l'œsophage"]
```

Ce code utilise la bibliothèque pour faire une recherche sur Wikipédia. Il est maintenant possible de récupérer le contenu de ces pages et de créer notre corpus. Pour alléger notre analyse, nous allons nous limiter à 8 pages, mais n'hésitez pas à en prendre davantage pour approfondir l'analyse.

```
corpus = []
for i in pages[0:8]:
    p = wp.page(i)
    corpus.append([i,p.content])

corpus = pd.DataFrame(corpus,columns = ["page","contenu"])
corpus.to_csv("data/corpus_cancer_wiki.csv")
```

Ce code utilise la méthode `page` qui permet de récupérer une page *Wikipedia*, puis ajoute son contenu dans une liste, la met sous une forme de tableau *Pandas* puis sauvegarde ce corpus. Le corpus de textes est constitué et nous pouvons l'analyser.

## Décomposer les textes en mots

Un texte est un contenu complexe, qui peut être décomposé en plusieurs éléments. L'unité élémentaire la plus évidente est souvent le mot. La phrase peut aussi être un niveau pertinent dans certains cas. Les groupes de mots peuvent aussi être importants : les 2-grams (suite de 2 mots), les 3-grams (suite de 3 mots), etc.

Suivant la question qui vous intéresse, certains niveaux seront plus pertinents que d'autres. Et puis, il est aussi possible de rajouter des informations comme la place du mot dans la phrase : est-ce un nom ? un verbe ? La catégorie morphosyntaxique de ces mots peut en effet aussi être intéressante.

Nous avons déjà vu quelques méthodes de base en début de chapitre pour commencer à manipuler des textes. Le comptage du nombre de mots ou la fréquence d'un mot dans un texte peut se faire facilement.

## ❶ Exercice

**Rajoutez deux colonnes au corpus avec le nombre de mots et la fréquence du terme ‘cancer’.**

```
corpus["contenu_min"] = corpus["contenu"].str.lower()
corpus["nombre_mots"] = corpus["contenu_min"]\ 
    .apply(lambda x : len(x.split()))
corpus["frequence_cancer"] = corpus.apply(lambda x :\ 
    x["contenu_min"].count("cancer")/x["nombre_mots"],\ 
    axis=1)
```

Vous pouvez aussi utiliser des logiciels extérieurs dédiés pour faire ce traitement des données textuelles. C'est le cas du logiciel libre Iramuteq qui a su trouver sa place en SHS. Dans ce cas, maîtriser Python permet de jouer un rôle de liant entre des données hétérogènes et un logiciel spécialisé.

## ❷ Information

**Exporter les données pour Iramuteq**

Il suffit de mettre en forme le corpus de vos textes de la bonne manière : un fichier .txt où chaque texte est séparé par une ligne qui a la forme « \*\*\* \*VARIABLE1 \*VARIABLE2 ». Trois (petites) lignes de code permettent de créer un tel fichier en ouvrant un nouveau fichier, puis en faisant une boucle sur nos données, et en écrivant chaque texte mis en forme :

```
with open("data/fichier-iramuteq.txt","w") as f:
    for i,j in corpus.iterrows():
        f.write("*** *id_{} \n\n {} \n\n".format(i,
            j["contenu"].replace("*","")))
```

Attention à bien penser à enlever toutes les « \* » qui sont des caractères spéciaux et vous rappeler que l'encodage de base de Python est de l'utf-8 quand vous l'ouvrez avec Iramuteq. D'autres informations sur l'installation et l'utilisation d'Iramuteq : <http://www.iramuteq.org/>.

Sauf dans des analyses plus complexes, dont nous nous contenterons de dire quelques mots, un texte est avant tout vu comme un sac de mots. Une étape importante de l'analyse est la décomposition en mots. Cette étape s'appelle la *tokenisation*. Bien souvent, l'information qui nous intéresse n'est pas le mot en lui-même mais sa forme : qu'importe que le mot soit au singulier ou au pluriel, ou que le verbe soit conjugué ou pas.

Prendre un mot pour revenir à sa forme de base est une étape appelée *lemmatisation*. Dans certains cas, il peut y avoir de grandes différences entre l'usage d'un mot au singulier ou au pluriel. De même, l'ordre des mots peut avoir une importance. Gardez en tête que ces opérations, si elles permettent un traitement quantitatif, font perdre de l'information.

Ces opérations sont assez bien identifiées. Plusieurs bibliothèques permettent de le faire. La plus connue et complète en Python est *Nltk*. Une difficulté de ces traitements de la langue est l'étape de lemmatisation et d'identification morpho-syntaxique (appelée POS pour *Part-of-speech tagging*) qui dépend de la langue. Si l'anglais est pris en charge de base, les autres langues peuvent rencontrer des limites. Pour cette raison, nous serons amenés à utiliser une autre bibliothèque, *SpaCy*. Enfin, pour la modélisation des thèmes dans les textes, une bibliothèque dédiée existe, *Gensim*.

Certaines opérations peuvent être réalisées avec les trois bibliothèques : à vous de creuser pour trouver la manière qui vous semble la plus évidente. *Nltk* a été principalement développé pour un usage universitaire d'analyse linguistique. *SpaCy* est pensé pour être un outil opérationnel à destination du développement de services basés sur l'analyse du langage, donc pour des informaticiens devant développer des applications rapides et fonctionnelles.

Débutons d'abord avec les opérations de base. Nous allons travailler sur un texte unique. La première étape est de le décomposer en mots avec la fonction `word_tokenize` :

```
import nltk
texte = corpus.loc[0,"contenu"]
texte_tokenise = nltk.word_tokenize(texte)
print(texte_tokenise[0:10])
```

```
[ 'Le', 'cancer', 'est', 'une', 'maladie', 'provoquée', 'par', 'la',
 'transformation', 'de']
```

Vous pouvez avoir besoin de la commande `nltk.download('punkt')` pour compléter l'installation. Le texte est devenu une liste de mots. L'étape suivante est de lemmatiser ces mots (réduire les mots à leur « racine ») avec un module dédié au français, car il faut pouvoir reconnaître le mot pour le réduire :

```
from nltk.stem.snowball import FrenchStemmer

stemmer = FrenchStemmer()
texte_lemmatise = [stemmer.stem(m) for m in texte_tokenise]
print(texte_lemmatise[0:10])

['le', 'canc', 'est', 'une', 'malad', 'provoqu', 'par', 'la',
 'transform', 'de']
```

Ce code crée un objet de réduction (*stem* pour souche) dédié au français et l'applique à chaque mot du texte. Nous remarquons que beaucoup de mots ne sont pas très intéressants. Ces mots sont couramment appelé *stopwords* et correspondent des mots qui ne portent pas beaucoup d'information. Chaque langue a ses propres *stopwords*. À l'inverse, les mots « utiles » sont appelés formes actives. *Nltk* a une liste de ces mots peu utiles, que vous pouvez compléter en fonction de vos besoins. Les filtrer permet de réduire le nombre de mots. Cela vaut aussi pour des éléments de ponctuation, ou des nombres.

```
from nltk.corpus import stopwords
from string import punctuation

stop_words = stopwords.words('french') + list(punctuation)
texte_lemmatise_filtre = [i for i in texte_lemmatise
                           if i not in stop_words]
print(texte_lemmatise_filtre[0:5])

['canc', 'malad', 'provoqu', 'transform', 'cellul']
```

Ce code importe à la fois les *stopwords* de la langue française, définis par *Nltk* et les symboles de ponctuation. Il crée alors une liste de ces éléments et filtre les mots à partir de cette liste<sup>12</sup>.

Ce code ne garde que les mots qui ne sont pas dans la liste `stop_words`. Cependant, certaines ponctuations ne semblent pas prises en compte : il faut alors soit les rajouter dans les *stopwords* soit les enlever directement dans le corpus initial. Dans certains cas, vous allez avoir envie d'enlever des adresses web qui sont dans le texte, ou des nombres. Et vous pouvez rajouter des règles spécifiques, en fonction de votre connaissance du corpus. Dans les faits, on construit souvent ces règles au fur et à mesure de l'analyse. Cela représente une étape importante de construction du corpus.

Nous avons donc réduit le texte à ce qu'on appelle des *formes actives* : les entités qui vont nous servir pour l'analyse. En fonction de votre connaissance du sujet, vous pouvez considérer que certains mots doivent encore être filtrés. Au

<sup>12</sup>. Vous aurez besoin d'exécuter la commande `nltk.download('stopwords')` la première fois que vous exécutez `from nltk.corpus import stopwords` afin de compléter l'installation.

contraire, vous pouvez vous rendre compte que certains mots composés ont été séparés. Trouver les n-grams de votre texte peut se faire en amont en remplaçant ces combinaisons de mots par un mot unique. Par exemple, si l'expression « le président de la république » est importante, elle peut se mettre sous la forme « le président\_de\_la\_république » dans un traitement initial, évitant que ces mots soient séparés.

Vous pouvez maintenant avoir une idée du contenu de votre texte sur la base des mots les plus fréquents :

```
pd.Series(texte_lemmatise_filtre).value_counts()[0:10]
```

```
canc      291
cancer    165
dan       152
plus      96
cellul    85
d'un      78
trait     76
tumeur    73
certain   62
'
dtype: int64
```

Maintenant que vous savez décomposé un texte en mots, vous pouvez l'appliquer à l'ensemble de vos documents. Et puis rechercher les mots spécifiques à certains documents et pas à d'autres. Sans rentrer dans les détails, une des manières de calculer un score de spécificité pour chaque mot est la transformation *tf-idf*. Pour cela, nous allons utiliser (encore) une autre bibliothèque, plus récente que *Nltk* et qui permet de facilement faire des traitements complexes.

Le *tf-idf* (de l'anglais *term frequency-inverse document frequency*) est une méthode de pondération souvent utilisée en recherche d'information et en particulier dans la fouille de textes. Le score calculé pour chaque mot permet de représenter l'importance d'un terme contenu dans un document dans le cadre d'un corpus. Plus le mot en question est fréquent dans le document et pas dans les autres, plus il aura un score élevé. Cela permet de filtrer, par exemple, les mots qui sont fréquents dans tous les documents et donc moins intéressants pour caractériser un document spécifique.

Tout d'abord, calculons les 10 mots les plus fréquents de nos 10 documents :

```
# Définition de la fonction de lemmatisation
def lem(texte):
    texte = nltk.word_tokenize(texte)
```

```

texte = [stemmer.stem(m) for m in texte]
texte = [i for i in texte if
         (i not in stop_words) and (len(i)>2)]
return texte

# Transformation du corpus
corpus["contenu_min"] = corpus["contenu"].str.lower()
corpus["contenu_lem"] = corpus["contenu_min"].apply(lem)

# Sélection des 10 premiers mots pour chaque page
top_10_mots = [list(pd.Series(i).value_counts()[0:10]).index
                for i in list(corpus["contenu_lem"])]
tab = pd.DataFrame(top_10_mots, index = list(corpus["page"]))
tab.head()

```

	0	1	...	8	9
Cancer	canc	cancer	...	certain	chez
Cancer (...	sign	canc	...	symbol	attach
Cancer d...	sein	canc	...	trait	d'un
Cancer d...	canc	poumon	...	carcinom	cancer
Cancer d...	tumeur	canc	...	trait	m-2

[5 rows x 10 columns]

Ce code :

- ➊ définit une fonction qui lemmatise un texte en enlevant les stopwords et les mots de moins de 2 lettres (toutes les étapes que nous venons de voir) ;
- ➋ applique cette fonction sur les textes et crée une nouvelle colonne dans le corpus ;
- ➌ transforme chaque liste en *Series* puis compte la fréquence des mots pour ne garder que les 10 plus fréquents ;
- ➍ les met dans un tableau *Pandas* et affiche le début.

Maintenant, vous voulons calculer pour chaque mot de chaque document l'indice *tf-idf* pour savoir quels sont les mots les plus caractéristiques. Pour cela, nous allons utiliser la bibliothèque *Gensim* adaptée pour l'analyse des thématiques. Elle permet d'introduire de nouvelles fonctions, mais aussi un objet particulier qu'est le *Dictionary* et qui permet de remplacer un mot par un numéro unique pour alléger les calculs et compter le nombre de fois où chaque élément est présent dans le corpus.

L'idée est la suivante : prendre chaque document transformé en groupes de mots, remplir le dictionnaire avec les mots du corpus, puis utiliser le dictionnaire pour transformer chaque groupe de mots en groupe de numéros correspondants. Ensuite, nous pouvons calculer le score *tf-idf* de chaque mot dans chaque document car le dictionnaire contient l'information de la présence du mot dans tout le corpus.

```
from gensim.models import TfIdfModel
from gensim.corpora import Dictionary

dictionnaire = Dictionary(corpus["contenu_lem"])
corpus_convert = [dictionnaire.doc2bow(line)
                  for line in list(corpus["contenu_lem"])]
modele_tfidf = TfIdfModel(corpus_convert)

vectors = [modele_tfidf[i] for i in corpus_convert]

def top_10(vec,dic):
    serie = pd.Series({dic[i[0]]:i[1] for i in vec})
    return serie.sort_values()[0:10].index

tab = pd.DataFrame([top_10(vec,dictionnaire) for vec in vectors],
                   index=corpus["page"])
tab.head()
```

	0	1	...	8	9
page			...		
Cancer	temp	indiqu	...	l'absenc	posit
Cancer	(... trop	peut	...	associ	typ
Cancer	d... nord	puis	...	bibliog...	meilleur
Cancer	d... fois	humain	...	deux	donc
Cancer	d... donc	lor	...	humain	jusqu

[5 rows x 10 columns]

Ce code :

- ➊ importe deux objets, un dictionnaire et le convertisseur *tf-idf* ;
- ➋ construit un dictionnaire avec l'ensemble du corpus lemmatisé ;
- ➌ convertit le corpus à l'aide de ce dictionnaire avec la méthode `doc2bow` qui associe un identifiant à chaque mot et crée un vecteur ;
- ➍ construit le modèle *tf-idf* sur le corpus avec l'objet `TfidfModel` ;
- ➎ applique ce modèle sur notre corpus converti ;
- ➏ définit une fonction pour afficher uniquement les 10 mots avec le score le plus élevé pour chaque document ;

- ✿ affiche le début du tableau de notre corpus.

Le nouvel objet *Dictionary* s'utilise comme un dictionnaire. Par exemple, pour avoir le mot associé à l'identifiant 10, il suffit de faire `dictionary[10]`. Considérez-le comme un passage entre la langue humaine et la langue de l'ordinateur.

### ?

#### Exercice

**Trouver le mot du corpus qui a le score tf-idf le plus élevé.**

Faites un tableau *Pandas* avec les mots et les scores puis triez ce tableau.

Nous pouvons encore complexifier un tout petit peu notre analyse. Pour le moment, nous avons décomposé le texte en mots, puis les avons lemmatisé. Dans certains cas, nous voulons pouvoir rajouter un traitement qui est d'associer les groupes de 2 ou 3 mots qui sont généralement ensemble. Pour cela, soit vous procédez à la main, en repérant les groupes qui vous intéressent, soit vous pouvez utiliser les outils des bibliothèques.

Un objet de *GenSim*, *Phrases*, permet de repérer les mots qui apparaissent fréquemment ensemble et permet de les réunir. Cependant, il faut lui donner deux paramètres : le premier est la fréquence minimale dans tout le corpus (`min_count`) et le deuxième est le seuil (`threshold`) qui définit la sensibilité. Plus ce dernier est haut, moins le regroupement est fréquent. Il n'y a pas vraiment de règle pour le déterminer, à vous de faire des tests.

```
from gensim.models.phrases import Phrases

phrases = Phrases(list(corpus["contenu_lem"]),
                  min_count=1, threshold=1)
corpus["contenu_lem_bi"] = corpus["contenu_lem"]\
    .apply(lambda x : phrases[x])
```

Ce code crée un objet qui permet d'associer les mots ensemble (`phrases`) et l'applique sur chaque corpus lemmatisé pour créer une nouvelle colonne. Vous pouvez vérifier que certains mots sont maintenant composés de deux mots reliés par des tiret-bas.

### ?

#### Exercice

**Recommencez l'analyse de tf-idf en intégrant les bigrammes dans votre analyse.**

Pour aller encore plus loin dans l'analyse, il est aussi possible de s'intéresser à la place des mots dans la phrase : est-ce que ce mot est un verbe, un nom, un adverbe ? Identifier le rôle grammatical d'un mot n'est pas évident. La possibilité de le faire rapidement a largement bénéficié des évolutions de l'apprentissage automatique, qui permettent d'entraîner des modèles sur des grands corpus déjà annotés pour ensuite analyser de nouveaux textes. Attention, cela signifie qu'il y a des erreurs (ce codage n'est pas infaillible) qui augmentent quand les textes codés sont différents des textes utilisés pour l'apprentissage.

La bibliothèque *Spacy* a des modèles pour différentes langues dont le français. Une étape est donc de télécharger ces modèles spécifiques. Par ailleurs, comme elle est prévue pour être opérationnelle, elle intègre directement les opérations que nous avons vues précédemment (tokenisation et lemmatisation) en rajoutant aussi la position dans la phrase (POS) sans qu'il soit nécessaire de tout écrire. Nous n'allons pas développer l'utilisation complète de la bibliothèque, mais voici un exemple qui illustre le caractère très opérationnel de cette approche :

```
import spacy
try:
    convertisseur = spacy.load('fr_core_news_sm')
except Exception:
    spacy.cli.download('fr_core_news_sm')
    convertisseur = spacy.load('fr_core_news_sm')

phrase = "Ceci est une phrase qui va servir d'exemple"
phrase_convertie = convertisseur(phrase)
for mot in phrase_convertie:
    print(mot,mot.pos_)
```

Ceci PRON  
est AUX  
une DET  
phrase NOUN  
qui PRON  
va VERB  
servir VERB  
d' ADP  
exemple NOUN

Ce code :

- ✿ charge la bibliothèque, télécharge le module spécifique français et le charge ;
- ✿ définit une phrase d'exemple et la convertit avec le modèle ;
- ✿ affiche chaque mot de la phrase et sa position dans la phrase.

De là à écrire un script qui ne garde que les noms, il n'y a qu'un pas que nous vous laissons franchir. Avec les opérations précédentes, nous sommes donc en mesure de passer d'un texte à un ensemble plus restreint de mots. Cela nous permet de faire des traitements déjà vus, comme des regroupements de textes sur la base de la présence de mots similaires dans le même texte (voir le chapitre sur les statistiques avancées). Il existe aussi d'autres stratégies statistiques qui permettent de réfléchir à la décomposition d'un corpus dans un ensemble de thématiques. Une thématique est définie par des mots qui sont fréquemment ensemble.

## Créer un nuage de mots

Il arrive souvent de vouloir visualiser le corpus de mots. Cela se fait très facilement avec la bibliothèque *Wordcloud* qui permet de nombreux paramétrages. L'objet *Wordcloud* prend en entrée une chaîne de caractères et génère un nuage de mots avec différents paramètres. Par exemple, avec notre corpus lemmatisé :

```
from wordcloud import WordCloud

# Mise en forme des données
texte = [" ".join(i) for i in corpus["contenu_lem"].values]
texte = " ".join(texte)

# Création et visualisation
wordcloud = WordCloud(background_color="white",
                      max_words=5000, contour_width=3)
wordcloud.generate(texte)
wordcloud.to_image()
```



Fig. 9.2 – Représentation d'un nuage de mots

Ce code :

- ✚ charge la bibliothèque ;
  - ✚ crée une longue chaîne de caractères en reliant tous les mots du corpus lemmatisé ;
  - ✚ crée un objet *Wordcloud* avec différents paramètres ;

- ♣ l'applique sur le texte ;
- ♣ le visualise dans la figure 9.2.

Pour le sauvegarder, la commande est `wordcloud.to_file("nuage.png")`.

Il est possible d'aller beaucoup plus loin dans le paramétrage d'un nuage de mots. Vous pouvez définir aussi la forme ou les couleurs. Encore une fois, un petit coup d'œil à un tutoriel ou à l'aide peut vous donner des idées.

## Analyser les thèmes d'un corpus

Nous allons présenter l'utilisation d'une des méthodes les plus utilisées pour analyser des thématiques : le modèle *Latent Dirichlet Allocation*, ou LDA. Pour le présenter schématiquement, le modèle cherche à identifier des thématiques dans un texte sur la base d'associations statistiques. Ces thématiques sont identifiées sur la base de la proximité des mots. Une fois le modèle obtenu sur un corpus, il est possible de calculer à quelles thématiques appartient un texte donné. Un thème est alors un ensemble de mots-clés caractéristiques.

Très utilisées pour des applications pratiques, ces stratégies trouvent cependant des limites en raison de la difficulté d'expliquer les thématiques obtenues ou le faible apport que l'identification apporte à la réflexion. Néanmoins, dans certains cas elles peuvent vous être utiles pour explorer un ensemble très large de documents ou trier des éléments.

Dans tous les cas, l'extraction des thèmes va aussi dépendre de l'ensemble des étapes en amont du modèle. Ces étapes, que nous avons vues dans la partie précédente, visent à ne garder que les éléments vraiment signifiants. Concrètement, la qualité de l'analyse va dépendre de :

- ♣ la nature et la taille du corpus ;
- ♣ le travail en amont de nettoyage du corpus ;
- ♣ le type de modèle ;
- ♣ les paramètres de ce modèle.

Vous vous en doutez, il existe d'autres modèles qui améliorent les résultats ou permettent de trouver d'autres thématiques dans un corpus. Par exemple, une variation du LDA est le modèle *LDA Mallet*<sup>13</sup>. Le monde de l'analyse textuelle reste encore un peu un terrain d'aventure où plusieurs approches coexistent sans que des règles claires de choix s'imposent.

Appliquons ce modèle sur notre petit corpus : sa taille limitée fait aussi que les associations statistiques vont être potentiellement moins fortes que sur un corpus plus vaste. Nous partons du corpus déjà lemmatisé.

---

13. Un autre type de modèle est le *Latent Semantic Analysis* (ou *Indexing*) LSI, qui correspond à faire une décomposition en facteurs principaux de la matrice documents / formes actives et de garder les principaux axes.

```

from gensim.corpora import Dictionary
from gensim.models.phrases import Phrases, Phraser
from gensim.models.ldamodel import LdaModel

# Générer les bigrams
phrases = Phrases(list(corpus["contenu_lem"]),
                  min_count=1, threshold=1)
phraser = Phraser(phrases)
corpus["contenu_lem_bi"] = corpus["contenu_lem"]\ 
    .apply(lambda x : phraser[x])

# Crédation du corpus
dictionnaire = Dictionary(corpus["contenu_lem_bi"])
corpus_convert = [dictionnaire.doc2bow(line) for
                  line in list(corpus["contenu_lem_bi"])]

# Modélisation
lda_model = LdaModel(corpus=corpus_convert,id2word=dictionnaire,
                      num_topics=5,random_state=100,
                      update_every=1,chunksize=100,
                      passes=10, alpha='auto',
                      per_word_topics=True)

# Affichage
categories = lda_model.print_topics()
print(categories[0])

```

(0, '0.005\*canc\_poumon' + 0.005\*poumon' + 0.004\*dan'  
+ 0.004\*cellul' + 0.004\*==== + 0.004\*canc' + 0.004\*plus'  
+ 0.003\*tumeur' + 0.003\*cancer' + 0.003\*risqu'')

Ce code :

- ➊ charge les objets nécessaires au traitement ;
- ➋ crée un modèle d'association des bigrammes, en utilisant une étape en plus que précédemment avec l'objet Phraser qui a pour but d'accélérer le traitement ;
- ➌ construit les bigrammes dans notre corpus (les mots associés ensemble) en rajoutant une colonne à notre corpus ;
- ➍ crée un dictionnaire de tous les termes du corpus, étape nécessaire pour convertir le corpus en vecteur ;
- ➎ convertit le corpus en vecteur ;

- ⊕ crée le modèle LDA en définissant le corpus d'analyse avec l'option `corpus`, le dictionnaire de conversion avec `id2word`, le nombre de thèmes à chercher avec `num_topics` et différents paramètres ;
- ⊕ exporte les thèmes identifiés ;
- ⊕ affiche le premier thème.

Vous pouvez afficher tous les thèmes identifiés pour avoir une idée du résultat final. Certains des mots ou des termes n'ont pas de sens ou sont trop génériques : peut-être il faudrait les inclure dans les *stopwords* à retirer, pour avoir des classes plus significatives. Par ailleurs, est-ce que 5 catégories suffisent ?

### ?

## Exercice

**Retirer les mots les moins signifiants du corpus avant traitement et faites varier le nombre de classes.**

Les différentes classes obtenues n'ont pas toujours de sens et posent des questions sur l'interprétation possible.

Une fois ce modèle entraîné, il est possible de l'utiliser pour classer des articles (ceux du corpus, mais aussi d'autres). L'utilisation est très simple et renvoie trois informations différentes : à quelle classe principale appartient un document avec la probabilité et à quelles classes chaque mot du document appartient en priorité. Pour avoir la classe du deuxième document (la page sur les signes astrologiques) :

```
page = corpus_convert[1]
attribution = lda_model[page]
print("La classe attribuée à la page est : ",attribution[0])
```

La classe attribuée à la page est : [(3, 0.9989762)]

Dans la même logique, il est possible de calculer phrase par phrase le thème principal, de calculer la proportion relative de chaque thème dans chaque document, etc. Un usage concret par exemple de ce modèle est de détecter les passages qui parlent en fait du cancer comme signe astrologique, par exemple pour s'en débarrasser si notre centre d'intérêt est le cancer comme maladie.

Quels mots garder ? Combien de thèmes ? Il n'y a pas de réponses définitives à ces questions, mais des stratégies pour essayer d'y répondre. La priorité doit être la capacité d'interpréter ces classes. Pour un modèle donné, il est possible de calculer un indice de cohérence<sup>14</sup>. À ce moment-là, une stratégie est de choisir le modèle avec le nombre de classes le plus petit qui a le meilleur indice en calculant des modèles pour différentes classes et regardant la variation de cet indice.

14. Pour plus d'information, car le sujet est compliqué, vous pouvez jeter un coup d'œil sur ce tutoriel : <https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/>.

## ❶ Information

### Visualiser les thèmes

*pyLDAvis* est un outil de visualisation graphique des thèmes de LDA qui fonctionne bien dans le Notebook Jupyter. Vous pouvez installer cette bibliothèque dédiée et la lancer de la manière suivante avec `lda_model` le modèle précédent et `corpus` le corpus correspondant :

```
import pyLDAvis
import matplotlib.pyplot as plt
%matplotlib inline
pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim.prepare(lda_model, corpus, id2word)
vis
```

La taille représentée des thèmes dépend de leur importance et leur superposition indique leur degré de proximité par rapport à leur composition en termes de mots.

Une autre stratégie possible pour analyser les données est d'appliquer les méthodes vues dans les statistiques avancées précédentes : soit une analyse factorielle des correspondances sur les formes actives (les mots retenus après tokenisation et lemmatisation) soit des méthodes de classification comme la classification hiérarchique ascendante. Chaque élément (par exemple un texte) peut être représenté comme une ligne d'une matrice où chaque colonne est une forme active. Calculer les composantes de cette matrice permet d'identifier les proximités entre ces formes actives pour rendre compte des thèmes ou des tendances. De même, la classification permet de créer des regroupements.

Bien entendu, la présentation est trop brève pour rentrer dans le détail d'une utilisation concrète. Cependant, vous avez pu vous rendre compte de la possibilité de progressivement mettre en place le modèle, contrôler chaque étape et d'analyser les résultats. Dans le cas où vous avez besoin de mettre en place une analyse thématique poussée, vous disposez donc de tous les outils et de la précision nécessaire pour obtenir le résultat souhaité. Passons maintenant à une autre forme de données : les données relationnelles, ou réseaux.

### 9.2.2 L'information relationnelle dans les réseaux

Les réseaux sont des entités que l'on rencontre dans beaucoup de domaines qui se définissent par une information relationnelle : réseaux sociaux, réseaux de mots ou réseaux de transports.

## Voir et mesurer les relations

Par définition, un réseau est un ensemble d'éléments (souvent qualifiés de nœuds) et de relations entre ces éléments (les liens). À partir de là, beaucoup de situations peuvent être relationnelles :

- ♣ les liens de connaissance ou d'interaction entre personnes ;
- ♣ des documents reliés par leurs thématiques ;
- ♣ les liens entre domaines de recherche par les publications scientifiques ;
- ♣ les liens entre pays par les échanges commerciaux ;
- ♣ les liens entre des acteurs qui jouent dans un même film.

Des données relationnelles sont moins définies par la nature des informations (textes, images...) que par la manière de coder une information relationnelle se rapportant à des données. Par exemple, à partir de peintures, il est possible de coder des personnages présents dans la même scène et progressivement construire un réseau. Cela signifie que, quelles que soient vos données initiales, il est envisageable d'y réfléchir sous la forme d'un réseau.

Comme souvent, une première étape est alors la construction du corpus de données sous la forme d'un réseau. Ensuite, la question se pose du traitement de ce corpus. Plusieurs domaines, en particulier issus des sciences sociales, ont développé des outils et des mesures spécifiques pour rendre compte des propriétés des réseaux. Cette science des réseaux se déploie par exemple de manière très quantitative dans les sciences de la complexité avec des applications en biologie mais aussi en SHS comme l'utilisation des données de télécommunication.

Voici quelques idées de traitements possibles sur les réseaux :

- ♣ identifier les entités centrales et plus marginales ;
- ♣ calculer des distances sur le réseau (combien d'étapes pour passer d'une entité à l'autre) ;
- ♣ trouver les communautés ;
- ♣ visualiser les relations et les proximités.

Ces informations peuvent ensuite être utilisées pour elles-mêmes ou réintégrées dans une réflexion plus générale. Il est assez facile de construire et d'analyser des réseaux en Python. De même, des outils existent pour la visualisation. Cependant, nous vous conseillons d'utiliser un logiciel dédié et libre, *Gephi*, pour les manipulations graphiques de vos réseaux.

## Construire un réseau

La bibliothèque *Networkx* permet de construire et d'analyser des réseaux. Comme souvent, cela passe par l'introduction d'un nouvel objet, le *Graph*.

Nous allons continuer à utiliser les données de Wikipédia, sur la thématique du cancer, comme illustration. Chaque page Wikipédia contient des relations vers de nouvelles pages. On peut donc envisager d'étudier le réseau des relations entre les pages. Ainsi, il est envisageable que les pages qui traitent de sujets similaires renvoient vers des pages similaires. Cela nous permettrait par exemple de filtrer les pages sur une même thématique.

Comme précédemment, nous allons nous concentrer sur un sous-ensemble de quelques pages. N'hésitez pas à le faire sur un plus grand réseau. D'abord, récupérons les pages Wikipédia et les liens de chaque page. Nous avions récupéré le nom de 100 pages contenant « cancer » dans la variable `pages`. Nous allons uniquement traiter 8 de ces pages :

```
corpus = []
for i in pages[0:8]:
    p = wp.page(i)
    corpus.append([i,p.links])
print("Nombre de pages : {}".format(len(corpus)))
for i in corpus:
    print("La page {} a {} liens".format(i[0],len(i[1])))
```

```
Nombre de pages : 8
La page Cancer a 504 liens
La page Cancer (astrologie) a 81 liens
La page Cancer du sein a 447 liens
La page Cancer du poumon a 381 liens
La page Cancer du pancréas a 212 liens
La page Cancer colorectal a 340 liens
La page Tropique du Cancer a 658 liens
La page Cancer de l'œsophage a 164 liens
```

Ce code utilise l'attribut `links` pour récupérer les liens vers des pages de chaque page, puis fait une boucle pour afficher l'information.

L'idée étant de faire un réseau des relations entre les pages. Deux pages sont reliées entre elles si elles ont un lien. Ce lien est un lien orienté (d'une page vers une autre). Nous allons construire ce réseau grâce à l'objet `Graph` de la bibliothèque `Networkx` en rajoutant progressivement les noeuds et les liens. Pour cela, à chaque fois, on regarde si l'élément existe dans le réseau et, suivant le cas, augmente leur poids (le nombre d'apparitions) ou le crée. Comme ce code est un peu long, nous commentons chaque étape.

```
import networkx as nx

# Définition d'un réseau
```

```

G = nx.Graph()

# Boucle sur nos données
for p in corpus:
    p_ini = p[0]

    # Ajouter la page ou augmenter son poids
    if not p_ini in G:
        G.add_node(p_ini,size=1)
    else:
        G.nodes[p_ini]["size"]+=1

    # Faire une boucle sur les liens
    for p_ext in p[1]:
        # Ajouter la page du lien ou augmenter son poids
        if not p_ext in G:
            G.add_node(p_ext,size=1)
        else:
            G.nodes[p_ext]["size"]+=1

        # Ajouter le lien ou augmenter son poids
        if G.has_edge(p_ini,p_ext):
            G[p_ini][p_ext]["weight"]+=1
        else:
            G.add_edge(p_ini,p_ext,weight=1)

print("{} nœuds et {} liens".format(G.number_of_nodes(),
                                      G.number_of_edges()))

```

2043 nœuds et 2777 liens

Ce code :

- ➊ charge la bibliothèque et crée l'objet `Graph`;
- ➋ prend chaque page initiale `p_ini`;
- ➌ commence par voir si c'est déjà un nœud dans le réseau avec la méthode `has_node` :
  - ★ si oui, augmente son attribut poids de 1, sinon la crée avec la méthode `add_node`;
- ➍ fait une boucle sur toutes les pages citées dans la page initiale :
  - ★ si le nœud `p_ext` n'existe pas, le crée, sinon augmente son poids de 1 ;
  - ★ si le lien entre la page initiale et la page extérieure n'existe pas, le crée, sinon augmente son poids de 1.

Avec ce code, nous venons de construire un objet de type réseau. Celui-ci permet ensuite d'utiliser les différents outils de la bibliothèque.

Cette démarche peut être utilisée avec de nombreuses autres sources de données. Par exemple, il est envisageable d'utiliser tous les mots dans la page pour faire la même chose, en créant un lien entre deux mots quand ils appartiennent à la même page. À partir du moment où vous commencez à y réfléchir, presque tout peut donner lieu à un réseau. Pour le créer, la démarche sera assez similaire à celle qu'on vient de suivre : ajouter des noeuds, ajouter des liens.

Nous l'avons vu : un réseau est constitué de noeuds et de liens, mais chaque noeud ou lien peut aussi avoir des attributs. Nous avons utilisé les attributs pour coder l'information du poids. Mais d'autres informations peuvent être mises, par exemple si vous avez des noeuds renvoyant à des entités différentes (des classes).

Une autre forme spécifique d'attribut est l'orientation. Le réseau précédent est orienté mais nous n'avons pas gardé cette orientation dans l'analyse. Il existe un autre objet pour définir des réseaux orientés : `DiGraph`. Cet objet est similaire à `Graph`, à la différence que les liens vont dans un sens spécifique et donc il y a des différences entre les chemins.

### ?

### Exercice

**Réalisez la même analyse mais avec un réseau orienté.**

Au lieu de déclarer un `graph`, on déclare `G = nx.DiGraph()`. La création des liens prendra en compte l'orientation. Et il est alors possible de calculer des degrés de liens sortants et entrants (avec par exemple la méthode `G.out_degree(NOEUD)`.

## Statistiques de réseaux

Une étape intermédiaire est de filtrer ce réseau pour ne garder que l'information pertinente. Par exemple, si nous ne voulons garder que les pages qui apparaissent au moins deux fois dans nos données, il faut récupérer l'information des noeuds et supprimer ceux qui sont trop petits.

```
G_filtre = G.copy()
for n in list(G.nodes(data="size")):
    if n[1]==1:
        G_filtre.remove_node(n[0])
print("{} nœuds et {} liens".format(
    G_filtre.number_of_nodes(),
    G_filtre.number_of_edges()))
```

268 nœuds et 830 liens

Ce code :

- ◆ copie le réseau initial, pour ne pas perdre les données ;
- ◆ récupère la liste des noeuds du réseau avec l'attribut taille avec `G.nodes(data="size")` ;
- ◆ si la taille est égale à un, supprime le noeud du réseau avec la méthode `remove_node` ;
- ◆ affiche les caractéristiques du réseau.

## ?

### Exercice

**Utilisez la méthode edges du réseau pour regarder la distribution du poids des liens.**

Utilisez `G.edges(data="width")` pour récupérer les informations sur les liens et créez ensuite un tableau *Pandas*.

Les mesures classiques dans un réseau sont le degré des nœuds (le nombre de connexions), la densité (le nombre de liens par rapport au nombre total possible) ou encore le diamètre<sup>15</sup>. *Networkx* permet de calculer ces différentes informations.

```
import matplotlib.pyplot as plt

# Calculs
densite = nx.density(G)
diametre = nx.diameter(G)
hist = nx.degree_histogram(G)

# Affichage
fig,ax = plt.subplots(figsize=(10,5))
ax.bar(range(0,len(hist[0:10])),hist[0:10])
plt.title("Distribution des degrés d'un réseau")
plt.subplots_adjust(left=0.3, right=0.7, top=0.7, bottom=0.3)

print("La densité du réseau est {}".format(round(densite,4)))
print("Le diamètre du réseau est de {}".format(diametre))
```

La densité du réseau est 0.0013

Le diamètre du réseau est de 5

---

15. Pour avoir une vision d'ensemble de l'analyse des réseaux, vous pouvez jeter un coup d'œil au manuel Scott & Carrington (2011).

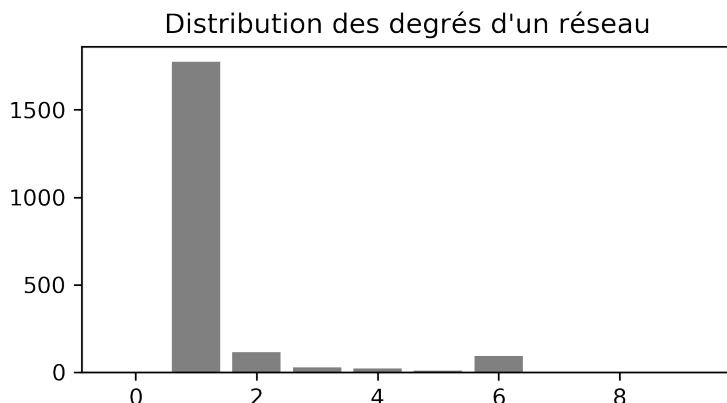


Fig. 9.3 – Visualisation des propriétés d'un réseau

Ce code calcule la densité, le diamètre et affiche l'histogramme des degrés de chaque noeud avec la figure 9.3. Encore une fois, l'idée n'est pas de détailler la démarche. Soulignons cependant que les traitements nécessaires, comme les différentes formes de centralité pour chaque noeud, ou la détection de communauté, peuvent être réalisés. La force de travailler directement avec Python sur ces réseaux est la faciliter de modifier le réseau, construire des opérations complexes et de calculer rapidement certains indicateurs pour préparer le moment d'interprétation. Un moment souvent symbolique est la visualisation du réseau.

## Visualiser un réseau

Visualiser un réseau est toujours une opération... difficile. Surtout quand le réseau est particulièrement grand. Pour cela, différentes stratégies de spatialisation existent, avec leurs avantages et défauts. La bibliothèque *Networkx* a un ensemble de fonctions pour produire ces visualisations. Elles sont cependant plus adaptées à de petits réseaux. Voici le résultat avec le réseau que nous avons produit sans essayer de le mettre en forme :

```
import matplotlib.pyplot as plt
fig,ax = plt.subplots(figsize=(5,5))
nx.draw(G_filtre, pos=nx.spring_layout(G_filtre),
        node_size=15, width=0.1, alpha=0.5)
```

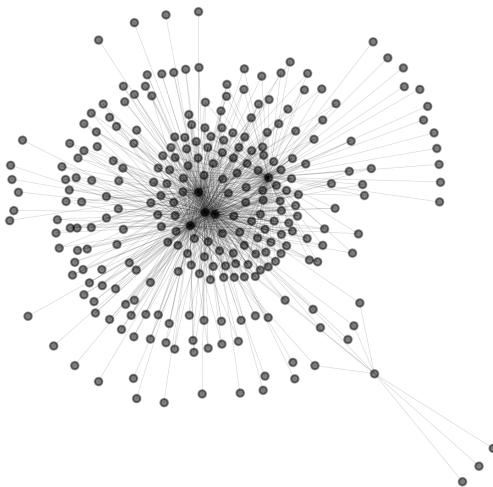


Fig. 9.4 – Visualisation d'un réseau

Ce code initialise une figure puis utilise la fonction `draw` pour produire une visualisation du réseau dans la figure 9.4. L'option `pos` permet de renseigner un positionnement des points dans l'espace. Pour calculer ce positionnement, on a recours à une fonction de spatialisation `spring_layout` qui calcule la distance entre chaque point en fonction des liens entre eux, comme si c'était des ressorts entre des particules qui s'écartent. D'autres spatialisations sont possibles. Ce réseau peut être configuré : changer la couleur des noeuds, la taille des liens, mettre les labels avec la fonction `draw_networkx_labels`. Cela peut cependant être laborieux pour obtenir un résultat agréable à regarder.

Si vous voulez juste explorer votre réseau, nous vous conseillons de recourir au logiciel *Gephi* qui est dédié à l'analyse des réseaux. Ses options de visualisation sont assez faciles à prendre en main et permettent d'interagir avec le réseau : sélectionner des parties, changer les couleurs, produire une belle visualisation et aussi calculer certaines statistiques. Nous retrouvons une utilisation de Python pour la mise en forme et le traitement de données, permettant ensuite de construire des données pour des logiciels spécialisés. C'est un rôle de liant !

Pour permettre cette visualisation, il suffit d'exporter le réseau que vous avez construit avec Python vers un fichier extérieur que *Gephi* est capable de lire. Ce fichier peut être dans différents formats. Nous vous conseillons d'utiliser le format GRAPHML adapté pour décrire un réseau et facilement lu avec *Gephi*.

L'export de Python vers un fichier GRAPHML se fait en une ligne :

```
nx.write_graphml(G_filtre,"mon_reseau.graphml")
```

Voilà, un fichier réseau « mon\_reseau.graphml ». Vous pouvez maintenant l'ouvrir avec *Gephi*. Avec les éléments présentés dans cette partie, vous êtes en mesure de créer un réseau à partir de vos données, le manipuler (électionner certains éléments, en supprimer d'autres), puis l'exporter dans un format facile à analyser dans des logiciels dédiés. Si vous avez besoin de fonctions spécifiques, il y a de grandes chances qu'elles existent dans la bibliothèque *Networkx*, donc pensez à regarder la documentation.

### ❶ Information

#### Gephi pour la visualisation

Gratuit et *Open source*, *Gephi* est un logiciel largement utilisé dans différentes communautés de pratique. Comme il est facile d'exporter un réseau construit avec Python dans différents formats, l'importer dans *Gephi* se fait très rapidement.

Pour finir sur une application avancée de visualisation, intéressons-nous à la cartographie avec Python.

## 9.3 Visualiser des données géographiques

Dans le cas précédent, nous avons délégué la visualisation d'un réseau à un logiciel extérieur. Cette solution est souvent disponible. Mais comme nous l'avons souligné en début de manuel, une des justifications à la programmation est de maîtriser chacune des étapes. C'est en particulier vrai pour la construction de cartes.

### 9.3.1 La construction de cartes

Les phénomènes humains sont presque toujours spatialisés. Cela nous amène à devoir représenter des zones géographiques, associer des éléments à des coordonnées, faire des opérations statistiques prenant en compte des caractéristiques comme la distance entre des entités et puis visualiser les résultats sous la forme de cartes en choisissant des symboles comme les marqueurs ou les couleurs.

D'excellents logiciels permettent de réaliser des cartes. Certains se rapprochent largement de logiciels de dessin, d'autres permettent de presque programmer directement vos cartes. C'est en particulier le cas du logiciel libre *QGIS*, largement utilisé par les géographes, qui allie une interface graphique avec la possibilité de faire des opérations avec Python. Pour autant, dans bien des cas, l'analyse géogra-

phique n'est qu'une partie d'un traitement plus général. Une bibliothèque Python permet d'allier la faciliter de manipuler des données de *Pandas* avec les représentations graphiques : *GeoPandas*.

### 9.3.2 Introduction à *GeoPandas*

*GeoPandas* est une bibliothèque qui permet de manipuler des informations géographiques. Pour la présenter simplement, il s'agit de manipuler des tableaux qui contiennent en plus une information géographique : des zones (polygones), des lignes ou des points.

Les géographes penseront directement aux logiciels SIG (Système d'Information Géographique, dont QGIS fait partie). Et en effet, cette bibliothèque permet de relier le traitement sur une base de données (un tableau, où il est possible de faire des calculs, de rajouter des colonnes, de modifier les lignes) et la visualisation de ces données. Fondamentalement, cela revient à relier une base de données à des outils de visualisation.

Comme souvent, il est plus facile de comprendre sur un cas concret. Comme pour *Pandas*, construire à la main des données *GeoPandas* est possible, mais souvent ces données sont déjà créées : ce sont des cartes vectorielles. Le format le plus courant est le *Shapefile* ou *.shp*.

Pour prendre la France, un petit tour sur [data.gouv.fr](http://data.gouv.fr) permet de récupérer un fichier qui contient le contour des régions françaises à partir des données libres d'*Open Street Map*. Suivant vos besoins, vous trouverez beaucoup de données géographiques librement disponibles grâce au travail de la communauté des géographes.

```
import geopandas as gpd
fichier = "data/regions-20180101-shp/regions-20180101.shp"
france = gpd.read_file(fichier)
list(france.columns)

['code_insee', 'nom', 'nuts2', 'wikipedia', 'surf_km2', 'geometry']
```

Ce code charge la bibliothèque *GeoPandas* (pensez à l'installer), puis lit le fichier *.shp* que nous avons téléchargé et mis dans le dossier *data*. Il affiche ensuite le contenu de ce nouvel objet *france* qui est un *GeoDataFrame*.

Celui-ci ressemble beaucoup à un tableau *Pandas*, n'est-ce pas ? Avec des colonnes avec différentes informations. La seule différence majeure est cette colonne « *geometry* ». Celle-ci contient cette information géographique. Il est possible de directement afficher cette information, avec la méthode *plot* en précisant la couleur, ce qui donne la figure 9.5.

```
france.plot()
```

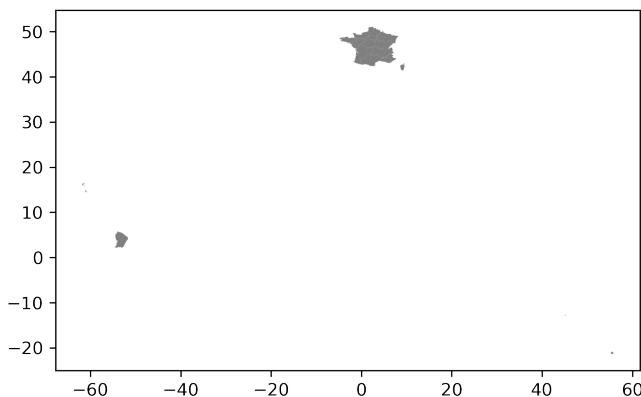


Fig. 9.5 – Exemple d'une visualisation de données *GeoPandas*

Comme la France compte des territoires d'outre-mer, ce qui peut être compliqué à représenter sur une même échelle, la visualisation n'est pas très parlante. De plus, le système de projection n'est pas nécessairement adapté. Cependant, comme il s'agit d'un tableau, il est possible de sélectionner uniquement quelques régions. Par exemple, la Corse :

```
corse = france[france["code_insee"]=="94"]
ax = corse.plot()
print(type(corse.loc[1,"geometry"]))
```

<class 'shapely.geometry.multipolygon.MultiPolygon'>

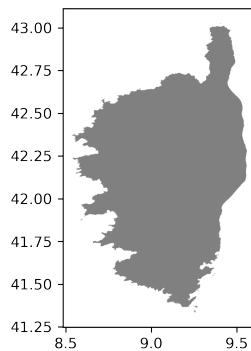


Fig. 9.6 – Visualisation d'un sous-ensemble des données géographiques

Ce code sélectionne une ligne avec une condition et affiche l'information correspondante dans la figure 9.6 en précisant que les axes doivent être dans la même unité. On constate que la colonne géométrie contient un objet particulier : un polygone. Celui-ci définit les frontières d'une surface.

### ➊ Exercice

#### Supprimer les DOMTOMs de la carte pour permettre une visualisation.

Le tableau est comme un tableau *Pandas* avec la méthode `drop`.

Il est alors possible d'avoir différentes informations, par exemple la taille de cette surface avec `corse.geometry.area` ou encore son centre avec `corse.geometry.centroid`. Ces éléments permettent de faire de nombreuses opérations. Par exemple, si nous voulons afficher le centre de chaque région pour la métropole :

```
france = france.set_index("code_insee")
france_metropole = france.drop(["01","02","03","04","06"])
france_metropole = france_metropole.to_crs("EPSG:2154")
ax = france_metropole.plot(figsize=(5,10), edgecolor="white")
france_metropole.centroid.plot(ax=ax, color="black")
```

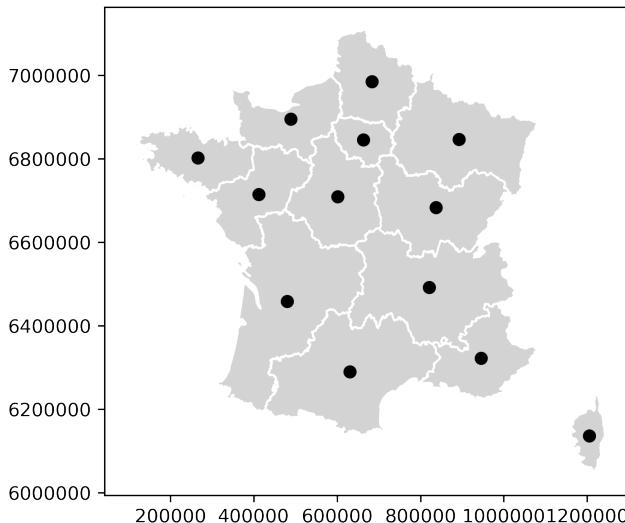


Fig. 9.7 – Visualisation d'une carte avec des données superposées

Ce code :

- ⊕ modifie le tableau pour enlever les DOMTOMs et ne garder que la France métropolitaine ;
- ⊕ projette la carte des coordonnées GPS initiales (`epsg:4326`) à une projection Lambert mieux adaptée à une visualisation habituelle (`epsg:2154`) ;
- ⊕ crée une figure en affichant la carte ;
- ⊕ affiche sur la même figure 9.7 les centres de chaque région. Ces centres sont des objets de type `Point` et sont représentés par des points.

### Information

#### Construire vos propres polygones

Vous pouvez vous aussi construire des objets géométriques pour les afficher. Un point est défini par une position dans l'espace. Une ligne par deux points. Et un polygone par un ensemble de points. Si vous voulez créer vos objets, par exemple pour les placer sur une carte, la bibliothèque *Shapely* permet de construire ces objets.

```
from shapely.geometry import Point, Polygon, LineString
mon_point = Point(1,1)
ma_ligne = LineString([(0,0),(3,2)])
mon_carre = Polygon([(0,0),(0,1),(1,1),(1,0)])
```

Vous pouvez afficher chaque élément et faire des calculs : par exemple, utiliser l'attribut `area` pour avoir la surface du carré. Pour construire vous-même un objet *GeoDataFrame*, cela ressemble à un *DataFrame Pandas* mais en rajoutant une colonne `geometry` qui contient des objets `Point`, `LineString` ou `Polygon`. Par exemple :

```
gpd.GeoDataFrame({
    'geometry': [Point(1, 1), Point(2, 2)],
    'donnees': [1, 2]})
```

Une fois ces différents objets géographiques définis, *GeoPandas* permet de réaliser l'ensemble des traitements que vous avez envie de faire : vérifier si un point est contenu dans une surface, réunir ensemble une surface ou encore calculer la distance entre deux points.

Quelle est la distance de La Réunion par rapport au centre de la France ? Réunissons toute la France dans un même polygone, calculons son centre et calculons ensuite la distance entre ce centre et La Réunion. Il est possible d'utiliser la méthode `dissolve` pour dissoudre ensemble différents polygones et la méthode `distance` pour avoir l'écart entre un polygone et un point.

```
france_metropole["unite"] = 1
france_metropole = france_metropole.dissolve(by="unite")
distance = france.loc["04"].geometry.distance(
    france_metropole.centroid.loc[1])
round(distance,2)
```

6634984.0

Ce code définit une nouvelle colonne avec la valeur 1 partout pour définir l'appartenance à une même entité puis demande de réunir l'ensemble des polygones qui ont la même valeur avec la méthode `dissolve`. Cela fait donc un unique bloc. Ensuite, il sélectionne La Réunion, prend le polygone avec l'attribut `geometry`, puis la méthode `distance` appliquée au centre du bloc. Par contre, la distance n'est pas en kilomètre. La relation entre distance calculée et distance géographique dépend du système de coordonnées utilisé.

Dans notre cas, les opérations d'affichage et de superposition se passent bien car les différentes informations sont bien rentrées dans le même système de coordonnées. Malheureusement, ce n'est souvent pas le cas. Dans les applications concrètes, certaines cartes sont dans une projection et vos données dans une autre. Des ajustements sont alors nécessaires. De même, savoir ce que représente une distance sur la carte par rapport à la distance réelle nécessite de connaître ce système de projection. Le système de projection d'un objet *GeoPandas* est renseigné dans l'attribut `.crs`.

Il est bien entendu possible de changer de système de projection. Une méthode `to_crs` permet de projeter un tableau *GeoPandas* dans un autre système. À vous de vous renseigner sur la projection que vous allez devoir utiliser.

Les outils permis par *GeoPandas* vont bien plus loin que ce que nous venons de présenter : il est possible de rechercher les éléments qui sont contenus, contiennent, se touchent, sont à une certaine distance, se croisent, se superposent, etc. Encore une fois, si cela vous intéresse, jetez un coup d'œil sur la documentation et les exemples existants. Pour illustrer l'utilisation de *GeoPandas* pour la visualisation, nous prendrons un exemple à partir des données que nous avons déjà eu l'occasion de traiter dans le chapitre.

### 9.3.3 La carte des populations françaises

Nous avons les données sur la population française et nous voulons faire des cartes de la distribution de la population dans les villes (aussi appelées des cartes choroplèthes) et de la proportion de femmes que nous avons étudiées dans les chapitres précédents. La première étape est de trouver une carte de France à l'échelle des villes. Une petite recherche sur internet permet de trouver une carte libre de droits

issue, encore une fois, d'*Open Street Map*<sup>16</sup>. Nous prenons le fichier le plus léger pour l'exemple, mais suivant vos besoins peut-être aurez-vous besoin d'une plus grande précision.

Il faut aussi charger les données de population française que nous avons déjà utilisées et recalculer la proportion de femmes. Puis réunir les deux types d'informations (géographiques et démographiques) à partir du code Insee qui identifie chaque commune française. Ces opérations sont très concrètement de la manipulation de données sous la forme d'un tableau.

```
tableau = pd.read_csv("./data/data-chap6.csv")
carte = gpd.read_file("./data/communes-20190101.json")
tableau["CODGEO"] = tableau["CODGEO"].apply(str)
d_sup = tableau.set_index("CODGEO")[["P15_POP","prop_f"]]
carte = carte.join(d_sup, on="insee")
```

Ce code utilise les différents outils que nous avons pu voir dans les chapitres précédents. Un passage un peu compliqué est l'utilisation de la méthode `join` pour ajouter les données d'un tableau à l'autre : il faut bien identifier un numéro unique.

Dans notre cas, ce numéro est mis en index du tableau à ajouter `d_sup`, nous avons bien mis les codes en chaînes de caractères des deux côtés et nous précisons avec l'attribut `on` la colonne du tableau initial à utiliser pour faire cette jointure.

Vous pouvez regarder la variable `carte` qui contient maintenant toutes les informations. Il ne reste plus qu'à procéder à une visualisation. Pour ne pas avoir à regarder toute la France, regardons le département du Bas-Rhin, dont le code postal commence par 67 :

```
basrhin = carte[carte["insee"]]\n    .apply(lambda x : True if x[0:2]=="67" else False)\nbasrhin.plot(column='prop_f', legend=True, figsize=(10,5))\nplt.axis('off')\nplt.title("Proportion de femmes dans le Bas-Rhin")\nplt.grid(False)
```

---

<sup>16</sup> Le fichier téléchargé en `.json` est sur le dépôt Github du manuel. Attention, ce fichier est particulièrement volumineux. Sinon ici : <https://www.data.gouv.fr/en/datasets/decoupage-administratif-communal-francais-issu-d-openstreetmap/>

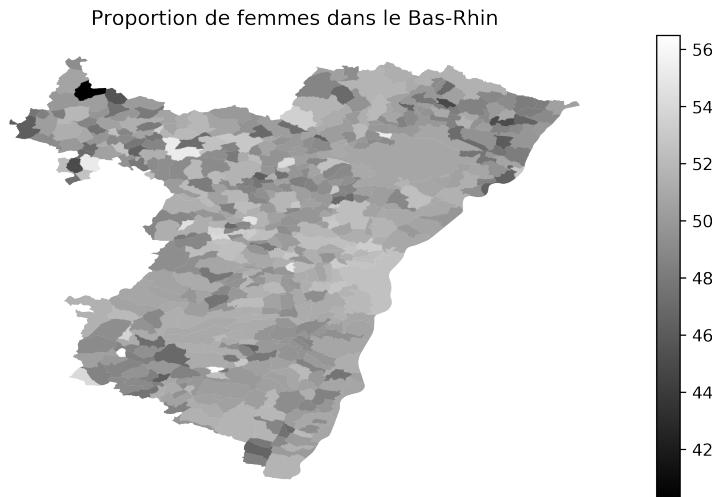


Fig. 9.8 – Visualisation des données statistiques sur une carte

Ce code :

- ➊ extrait un sous-ensemble de notre carte en ne prenant que les lignes où le code postal commence par 67 ;
- ➋ affiche la carte avec la méthode `plot` sur la figure 9.8 en indiquant le nom de la colonne à afficher et en mettant la légende ;
- ➌ enlève les axes de la carte et met un titre.

Une option de *GeoPandas* permet d'optimiser la visualisation d'une variable quantitative en présentant les quantiles calculés de telle sorte à avoir un rendu optimal. Il suffit de rajouter l'option `scheme = "quantiles"` dans la méthode `plot`.

Nous voulons vous montrer une dernière opération pour illustrer la flexibilité de *GeoPandas*. Traçons la carte de la population des Bouches-du-Rhône mais en plaçant en plus un point à l'endroit où ce manuel a été en partie écrit, près du Cours Julien, à Marseille. Nous avons tous les éléments à notre disposition : géocoder une adresse, la mettre sous la forme d'un objet *GeoPandas* puis la mettre sur une carte. Les différentes bibliothèques ont déjà été utilisées dans ce chapitre, donc nous ne les importons pas une nouvelle fois.

```
from shapely.geometry import Point

# Mise en forme des données
adresse = geocoder.osm('Cours Julien, Marseille')
point = Point(adresse.json["lng"], adresse.json["lat"])
donnees_points = gpd.GeoDataFrame({'geometry': [point]},
```

```

    'info':["Cours Julien"]},crs="epsg:4326")
carte = carte[carte["insee"]]\n
    .apply(lambda x : True if x[0:2]=="13" else False)]\n\n
# Cr ation de la carte
ax = carte.plot(column='P15_POP',legend=True,
                  figsize=(10,10),scheme = "quantiles",
                  edgecolor="black")
donnees_points.plot(ax=ax,color="red",edgecolor="white",
                      markersize=500)\n\n
# Mise en forme de la l gende
labels = ["Q1","Q2","Q3","Q4","Q5"]
legend_labels = ax.get_legend().get_texts()
for i, legend_label in enumerate(legend_labels):
    legend_label.set_text(labels[i])\n\n
# Affichage
plt.grid(False)
plt.axis('off')
plt.show()

```

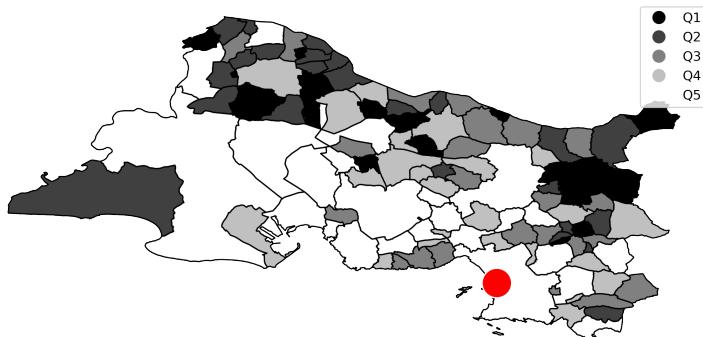


Fig. 9.9 – Visualisation de donn es sur une carte

Ce code :

- ➊ fait du g ecodage de l'adresse, r cup re l'information et construit un objet `Point`;
- ➋ fabrique un tableau `GeoPandas` avec un seul ´lement `donnees_points`;
- ➌ extrait la carte correspondant aux Bouches-du-Rh ne (code postal commen ant par 13);

- ⊕ affiche la carte avec comme information la population, en quartiles, en précisant la couleur du fond et des bords ;
- ⊕ affiche sur la même figure les données du tableau `donnees_points` : il devrait être rouge sur votre ordinateur.
- ⊕ met en forme la légende (sinon ce sont des nombres) en récupérant la légende générée puis en remplaçant à chaque fois le texte avec `set_text` par l'étiquette correspondante ;
- ⊕ affiche la carte 9.9.

Avec la possibilité de construire des objets géographiques, de les manipuler et de les représenter, vous êtes maintenant capable de réaliser une bonne partie des analyses géographiques envisageables.

## 9.4 Synthèse du chapitre

Ce chapitre proposait de faire le tour des possibilités ouvertes par l'usage de Python.

Les nombreuses bibliothèques existantes permettent de réaliser des opérations complexes dans le flux des autres traitements. La possibilité de paramétrier ces traitements et de créer les morceaux de codes spécifiques à vos problèmes permettent une flexibilité incroyable. Et dans le cas où des logiciels dédiés existent, travailler avec Python permet de faire du liant entre le corpus et le traitement.

Bien entendu, le temps passé sur chaque exemple était trop bref pour en saisir toute la portée. À chaque fois, de nombreuses options auraient pu être précisées. N'hésitez pas à compléter les éléments présentés avec les exemples disponibles sur internet, maintenant que vous savez que vous pouvez récolter des données, analyser des textes, construire des réseaux et des cartes.

Nous espérons que ce chapitre a permis d'ouvrir les perspectives des analyses envisageables et fournir des morceaux de codes et des manières de faire réutilisables dans vos propres traitements. Et rappelez-vous, généralement tout est possible. Donc si vous avez besoin de faire un traitement, ne vous découragez pas et décomposez le problème en étapes.

# Chapitre 10

## Organiser son code et le partager

Ce dernier chapitre a pour but de prendre un peu de recul sur la programmation afin d'améliorer et réutiliser son code. En effet, la logique même derrière la programmation est souvent de construire de nouveaux codes, de les réutiliser et de les partager avec d'autres utilisateurs.

### 10.1 De la ligne au logiciel

Dans ce manuel, nous avons rencontré de nombreuses manières d'utiliser la programmation. Dans certains cas, les outils sont déjà disponibles. Les bibliothèques correspondantes s'installent facilement et le traitement des données se fait en quelques lignes de code. Dans d'autres cas, au contraire, il nous a fallu écrire nous-mêmes du code supplémentaire, définir des fonctions dédiées et créer une multitude de variables et fonctions intermédiaires pour parvenir à nos fins.

Dans certains cas, ce code ne sert qu'une fois, pour une seule tâche. C'est le cas des analyses de données sur un corpus particulier. Dans d'autres, vous avez réalisé un traitement que vous voudrez pouvoir reproduire facilement. Vous serez probablement amené à réutiliser ces fonctions d'un projet à l'autre car elles correspondent à des traitements que vous réalisez souvent ou même de partager ces fonctions avec vos collaborateurs. Il serait dommage de réécrire à chaque fois ces traitements, ou qu'ils soient inutilisables par d'autres.

Dans ce chapitre, nous allons revisiter du code déjà vu et discuter comment le rendre réutilisable afin de voir quels sont les avantages, les inconvénients et les bonnes pratiques à suivre. Concrètement, nous allons voir comment passer progressivement de l'utilisation interactive de Python, par exemple dans un Notebook

Jupyter, à la production d'une nouvelle bibliothèque. En effet, il n'y a pas besoin d'être un expert de la programmation pour avoir envie de réutiliser le code qu'on a trouvé pour résoudre ses problèmes.

Passer de l'utilisation interactive de Python à l'écriture d'un script ou d'un module n'est pas toujours une action évidente. Cela demande souvent de prendre un surcroît de recul sur les problèmes pratiques et d'anticiper les besoins futurs. Un dicton commun dans la programmation est le suivant : « *Make it work, make it right, make it fast* », que l'on peut traduire par « faites-le fonctionner, puis rendez-le correct, et enfin rapide ».

Pour la plupart des bibliothèques que vous avez utilisées précédemment, les auteurs sont partis de quelques dizaines de lignes qu'ils utilisaient régulièrement et ont progressivement enrichi leur code jusqu'à leurs tailles et popularités actuelles. Dans tous les cas, il est important de partir de besoins concrets.

Pour le présenter schématiquement, l'unité de base est la ligne de code. Plusieurs lignes de code peuvent donner un script qui réalise une action. Il est alors souvent intéressant de regrouper les fonctions et les objets créés dans un *module* séparé, correspondant à un fichier .py qu'il est possible de charger en début de script pour ne pas avoir à écrire l'ensemble du code à réutiliser. Ces modules, qui sont pour l'instant un simple fichier, peuvent donner naissance à une bibliothèque spécifique, entendue comme un ensemble d'éléments permettant de réaliser certaines tâches, avec une documentation dédiée et partagée avec l'ensemble de la communauté. Bien entendu, tout code n'a pas destination à devenir un module, ni une bibliothèque.

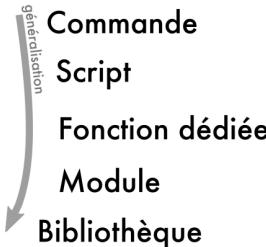


Fig. 10.1 – Étapes de la généralisation du code

Un code clairement écrit et documenté peut être partagé avec d'autres utilisateurs. Cela permet non seulement une réutilisation plus large, mais aussi la possibilité de vérifier les traitements réalisés. La reproduction des résultats d'un traitement de données est actuellement un des enjeux cruciaux pour la fiabilité des résultats.

## 10.2 Organiser son script dans le Notebook

Souvent, le premier code est réalisé de manière interactive dans le Notebook ou dans un petit script. Durant ce moment de création, il est important de documenter son code pour faciliter toute réutilisation. Il existe plusieurs manières de le faire, et nous rencontrerons un cas plus bas. Au minimum, utilisez les commentaires pour décrire les opérations que vous écrivez. Si votre script vous paraît maintenant évident, pensez au temps que vous allez prendre dans cinq mois pour vous rappeler pourquoi vous avez créé ces dix fonctions spécifiques.

La deuxième étape est de faire des copies de sauvegarde de votre travail, en particulier lorsque vous allez commencer à rendre un projet réutilisable. Nommez vos fichiers avec des noms explicites. En effet, il peut arriver qu'une modification introduise des erreurs dans votre script qui cesse de fonctionner. Avoir une version antérieure peut aider à résoudre certains problèmes.

Enfin, la troisième étape est de faire une phase de réorganisation (*refactoring*). Dans tout projet, il y a forcément un moment où votre code est un peu brouillon et mélangé. Certaines variables sont déclarées dans une cellule qui vient après leur utilisation, leurs noms ne sont plus très pertinents car vous avez changé la logique de votre démarche ou encore vous vous retrouvez avec plusieurs versions de la même fonction. Cette situation ne doit pas rester chronique au risque un jour de devenir irréversible.

Pour cela, un peu de nettoyage du Notebook est nécessaire. L'objectif est de pouvoir le ré-exécuter complètement de haut en bas sans erreurs. Si vous rencontrez des erreurs, essayez de les corriger et recommencez jusqu'à ce que l'ensemble du Notebook soit exécuté sans erreur. Cela signifie aussi que vous devez organiser les différentes étapes dans un ordre logique : les variables et fonctions sont définies avant d'être utilisées. En effet, quand on bricole avec des cellules, il arrive un moment où tout est mélangé.

Une fois le script réorganisé dans le Notebook, il est plus facile de hiérarchiser ce qui correspond à des fonctions génériques utilisées dans tout le code, et ce qui au contraire relève d'étapes précises. Il est aussi plus facile de regrouper ensemble certaines variables. C'est ainsi le moment de se rendre compte qu'il est peut-être plus intéressant d'utiliser une liste de variables que plusieurs variables séparées, ou que certaines informations pourraient être regroupées dès le début dans un tableau. Ce moment de réorganisation conduit souvent à repenser la logique d'ensemble du code et à le raccourcir drastiquement<sup>1</sup>.

Nous reprenons pas à pas dans la partie suivante une visualisation pour montrer la logique à l'œuvre dans la construction d'une fonction plus générique réutilisable.

---

1. Si cela peut être décourageant de voir un code se réduire à quelques lignes, dites-vous que l'étape initiale était souvent nécessaire pour obtenir le résultat souhaité.

## ❶ Information

### Gestionnaires de version

Il arrive qu'une modification du code amène des problèmes sur l'exécution d'un script et vous donne envie de revenir à une version antérieure pour corriger le problème. La gestion des versions permet ainsi de revenir en arrière. Il existe plusieurs manières de gérer les versions. Les services de Cloud gèrent les versions et vous permettent de revenir sur des versions précédentes de votre code. Cependant des outils de versionnement dédiés à la programmation existent. Dans ce livre nous ne discuterons pas l'utilisation de gestionnaire de version (*Git*, *Mercurial*, *Svn*) mais nous vous recommandons fortement d'apprendre au minimum les bases de *Git*. Un gestionnaire de version est bien plus puissant et sûr que de faire des copies à la main. Cependant, une autre manière de procéder peut être d'utiliser des systèmes comme *Google Drive*.

## 10.3 Retravailler une visualisation

Cette étude de cas vise à montrer les arbitrages qui sont faits entre des morceaux de codes génériques et d'autres plus spécifiques en vue de la construction d'un module dédié permettant la réutilisation d'une fonction.

### 10.3.1 Identifier les éléments génériques dans un code

Dans le chapitre sur la visualisation, nous avons présenté un graphique en bulles. Le code pour réaliser cette visualisation était le suivant, qui utilisait les différentes fonctions présentées au cours du manuel pour produire le graphique 10.2 :

```
import pandas as pd
tableau = pd.read_csv("./data/data-chap6.csv")
donnees = tableau[["CODGEO", "REG", "DEP", "LIBGEO",
                    "P15_POP", "P15_POPF", "P15_POP90P", "C15_POP15P_CS6",
                    "P15_POP7589", "prop_f", "prop_sup75"]]

import matplotlib.pyplot as plt
# Paramètres et données
colors = {"67": "C0", "68": "C1"}
donnees_filtrees = donnees[donnees["DEP"].isin(["67", "68"])]

# Création de la figure
fig,ax = plt.subplots()
for dep_number,j in donnees_filtrees.groupby("DEP"):
```

```
j.plot(kind="scatter",x="prop_f",
       y="prop_sup75",ax=ax,
       c=colors[dep_number],
       alpha=0.5,
       s=j["P15_POP"]/100, label=f"Departement {dep_number}")

# Options et visualisation
ax.set_xlim(45,55)
ax.set_ylim(4,15)
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion >75 ans (%)")
ax.set_title("Diagramme en bulles (département 67 et 68)")
ax.legend(loc=2, markerscale=1/6, scatterpoints=2 )
fig.savefig("basrhin-bubblechart.png")
plt.show()
```

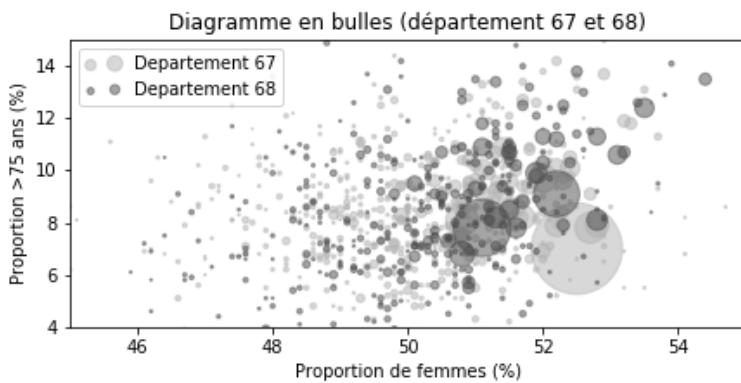


Fig. 10.2 – Diagramme en bulles

Imaginons que nous ayons régulièrement besoin de tracer ce type de diagramme, tout en devant modifier certains paramètres. L'idée générale reste la même mais nous voulons apporter des modifications mineures. Comment produire à la fois un code suffisamment générique qui permet sa réutilisation et suffisamment spécifique pour qu'il remplisse aussi le problème que l'on souhaite résoudre ?

Une première étape est de mieux réfléchir à la structure de ce code et de repérer les éléments structurels (génériques) et ceux qui relèvent plutôt du contexte d'usage (spécifiques). Ce sont des choix que nous devons faire. Dans ce cas, l'idée serait de comparer de la même manière certaines variables définies pour différents départements.

Si nos objectifs avaient été différents, cette séparation entre générique et spécifique aurait été autre. Par exemple si nous travaillons uniquement sur le Haut-Rhin (68) et Bas-Rhin (67), nous allons peut-être considérer que les départements sont génériques. Et que ce sont les variables comparées qui vont changer, et seront spécifiques à chaque visualisation.

Ainsi, si nous faisons un peu la dissection du code précédent avec l'idée que nous fixons les variables comparées et que l'objectif est de pouvoir faire varier les départements :

- ➊ `import pandas as pd` est relativement générique – nous utiliserons presque toujours cet alias ;
- ➋ `tableau = pd.read_csv("./data/data-chap6.csv")` est spécifique pour deux raisons :
  - ★ le nom de fichier changera probablement ;
  - ★ de même que le type de fichier ;
- ➌ les deux lignes qui calculent `prop_f` et `prop_sup75` semblent très spécifiques à notre opération ;
- ➍ l'importation de `Matplotlib` est aussi générique ;
- ➎ l'association de couleur au numéro de département est assez spécifique, liée au choix esthétique du moment.

Une seconde étape est de constater que les lignes sont assez mélangées : on importe des bibliothèques à deux endroits, on définit des variables à différents moments : n'est-il pas possible de mieux organiser la répartition du code ? Cela nous permettra en tout cas de mieux voir la structure générale.

Bien sûr, tout pourrait bouger. Mais nous pouvons faire des choix, généralement en fonction des usages. Dans notre cas, nous faisons l'hypothèse que la structure de nos données va peu ou pas changer, dans le sens où le nom des colonnes est une règle. Par contre, dans certains cas les fichiers sont en Excel, d'autres en CSV. La comparaison se fait toujours avec deux départements, mais pas forcément les mêmes. Cela nous permet de modifier l'ordre du code pour distinguer ce qui est générique de ce qui est plus spécifique. Pour rendre visible l'opération, nous indiquons en commentaire ce que nous considérons comme générique ou spécifique.

```
# générique
import pandas as pd
import matplotlib.pyplot as plt
fig,ax = plt.subplots()

# spécifique car le format peut changer
tableau = pd.read_csv("./data/data-chap6.csv")

# spécifique pour le département mais pas pour les couleurs
colors = {"67": "C0", "68": "C1"}
```

```

# générique (toujours les mêmes colonnes)
donnees = tableau[["CODGEO", "REG", "DEP", "LIBGEO",
                  "P15_POP", "P15_POPF", "P15_POP90P", "C15_POP15P_CS6",
                  "P15_POP7589", "prop_f", "prop_sup75"]]
donnees_filtrees = donnees[donnees["DEP"].isin(["67", "68"])]  

# générique, c'est l'opération de traitement des données
for dep_number,j in donnees_filtrees.groupby("DEP"):
    j.plot(kind="scatter", x="prop_f", y="prop_sup75",
           ax=ax, c=colors[dep_number],
           alpha=0.5,s=j["P15_POP"]/100,
           label=dep_number)  

# spécifique, car cela pourrait éventuellement varier
ax.set_xlim(45,55)
ax.set_ylim(4,15)  

# générique
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion >75 ans (%)")  

# générique dans la formulation mais spécifique pour les numéros
ax.set_title("Département 67 et 68")  

# générique
ax.legend(loc=2, markerscale=1/6, scatterpoints=2 )  

# spécifique, le nom peut changer
fig.savefig("basrhin-bubblechart.png")
plt.show()

```

### 10.3.2 Aller vers une fonction

Pour avancer dans la formalisation du code, nous allons séparer clairement ces éléments génériques et spécifiques en créant de nouvelles variables (génériques) qui prennent les valeurs (spécifiques) du cas particulier. Écrivons juste les morceaux qui vont changer par rapport à la situation précédente :

```

#...
  

dep1 = "67"
dep2 = "68"

```

```

colors = {dep1:"C0",dep2:"C1"}

#...

minx, maxx = 45, 55
miny, maxy = 4, 15

ax.set_xlim(minx, maxx)
ax.set_ylim(miny, maxy)

#...

```

(4, 15)

Pour rendre le code plus réutilisable, nous pouvons maintenant construire une fonction générique, qui pourra être utilisée avec des données spécifiques en paramètres. Nous prenons aussi le soin de décrire à quoi sert cette fonction. Cette aide pourra s'afficher quand l'utilisateur utilisera « ? » :

```

# générique pour l'ensemble du script
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.lines import Line2D

# générique, définition d'une fonction
def draw_bubble_chart(tab, dep1, dep2, minx, maxx, miny, maxy):
    """
    Cette fonction trace un diagramme en bulle.
    """
    colors = {dep1:"C0",dep2:"C1"}
    donnees = tab[["CODGEO", "REG", "DEP", "LIBGEO",
                  "P15_POP", "P15_POPF", "P15_POP90P", "C15_POP15P_CS6",
                  "P15_POP7589", "prop_f", "prop_sup75"]]
    donnees_filtrees = donnees[donnees["DEP"].isin([dep1,dep2])]

    fig,ax = plt.subplots()
    el = []
    for dep_number,j in donnees_filtrees.groupby("DEP"):
        j.plot(kind="scatter", x="prop_f", y="prop_sup75",
               ax=ax, c=colors[dep_number], alpha=0.5,
               s=j["P15_POP"]/100,label=dep_number)

    ax.set_xlim(minx, maxx)

```

```

ax.set_xlim(miny, maxy)
ax.legend(loc=2, markerscale=1/6, scatterpoints=2)
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion >75 ans (%)")
ax.set_title("Département {} et {}".format(dep1,dep2))
return fig, ax

```

Ce code commence par importer les bibliothèques génériques nécessaires au code, puis définit une fonction générique qui trace le diagramme à partir de paramètres. Quelques notes sur le code ci-dessus :

- ➊ `isin` a été mis à jour : `.isin([dep1,dep2])` ;
- ➋ le texte utilisé pour le titre est mis en forme dans `ax.set_title` à partir des variables ;
- ➌ on retourne de la fonction les éléments du graphiques `fig` et `ax` pour pouvoir les utiliser plus tard.

Notre fonction est maintenant réutilisable facilement et peut être appliquée à d'autres départements :

```
tableau_ = pd.read_csv("./data/data-chap6.csv")
```

```
draw_bubble_chart(tableau_, '22', '35',
                   minx=40, maxx=60, miny=0, maxy=20 )
```

```
(<Figure size 504x288 with 1 Axes>,
<matplotlib.axes._subplots.AxesSubplot at 0x7f7673d26c90>)
```

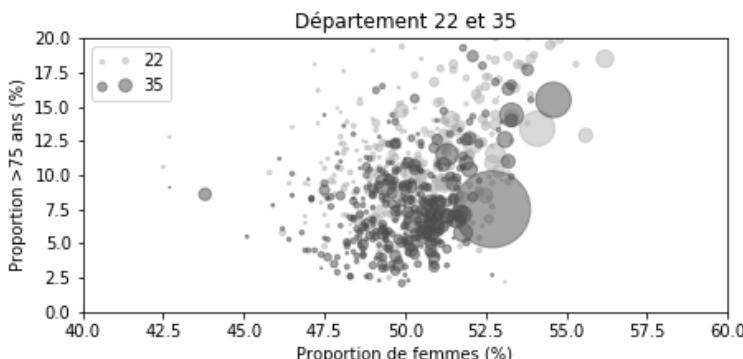


Fig. 10.3 – Diagramme en bulles sur d'autres départements

Ce code permet de représenter les données de deux autres départements dans la figure 10.3.

Entre l'idée initiale, le moment interactif d'écriture du premier code permettant d'avoir une visualisation pour deux départements et la production d'une fonction spécifique, de nombreuses étapes ont permis de préciser ce que nous voulions et de faire des choix (de couleurs, de formes, etc.). Dans la mesure où la comparaison de départements peut être importante pour la poursuite de l'étude, disposer d'une fonction dédiée à nos besoins pourra accélérer notre réflexion et faciliter la production de résultats intermédiaires. Pour autant, le choix d'écrire une fonction reste bien souvent basé sur votre propre estimation de vos besoins de réutilisation.

Pour le moment, nous sommes partis d'une manipulation et nous avons progressivement stabilisé une fonction dans un Notebook. L'étape suivante est d'autonomiser du Notebook et du code interactif pour en faire un module que nous pouvons charger dans d'autres analyses.

### 10.3.3 Un fichier à soi : créer un module dédié

À partir de cette étape de formalisation, nous pouvons envisager de séparer notre fonction du Notebook en la mettant dans un fichier dédié. Ce fichier est un script Python avec une extension .py. Par exemple, créez un fichier nommé `mes_fonctions_shs.py` que nous plaçons dans le dossier courant (celui dans lequel est exécuté le script). Le contenu de ce fichier est alors le suivant :

```
"""
Ce fichier contient ma boîte à outils de fonctions utiles
Fonctions disponibles :
- bubble chart
"""

__version__ = '0.0.1'

import matplotlib.pyplot as plt

def draw_bubble_chart(tableau,dep1,dep2,minx,maxx,miny,maxy):
    """
    Cette fonction trace un diagramme en bulle
    """
    colors = {dep1:"gray",dep2:"lightgray"}

    donnees = tableau[["CODGEO", "REG","DEP", "LIBGEO",
                      "P15_POP","P15_POPF","P15_POP90P", "C15_POP15P_CS6",
                      "P15_POP7589","prop_f","prop_sup75"]]

    donnees_filtrees = donnees[donnees["DEP"].isin([dep1,dep2])]
```

```

fig,ax = plt.subplots()
el = []
for dep_number,j in donnees_filtrees.groupby("DEP"):
    j.plot(kind="scatter", x="prop_f", y="prop_sup75",
           ax=ax, c=colors[dep_number], alpha=0.5,
           s=j["P15_POP"]/100, label=dep_number)

ax.set_xlim(minx, maxx)
ax.set_ylim(miny, maxy)

ax.legend(loc=2, markerscale=1/6, scatterpoints=2)
ax.set_xlabel("Proportion de femmes (%)")
ax.set_ylabel("Proportion >75 ans (%)")
ax.set_title(f"Diagramme en bulle (dep. {dep1} et {dep2})")
return fig, ax

```

Ce fichier est un nouveau module. Le commentaire entre deux """ au début est l'aide de ce module et sera affiché si un utilisateur demande de l'aide. La variable `__version__` permet d'indiquer la version pour garder une trace des évolutions. À partir de maintenant, je peux importer ce nouveau module `mes_fonctions_shs` de la même manière que nous importons des bibliothèques avec le terme `import`.

Puisque la fonction est désormais dans un fichier séparé, Python la charge au moment de l'importation puis l'utilise. Les fonctions définies dans le module sont appelées avec le nom du module, un point, puis le nom de la fonction, comme nous l'avons déjà utilisé pour les bibliothèques rencontrées au cours de ce manuel.

Si vous souhaitez apporter des changements à la fonction du module, vous devez relancer votre script. Généralement les fonctions placées dans des fichiers séparés changent peu. C'est donc une étape assez avancée de la stabilisation de vos outils. Une dernière étape peut encore exister pour votre script : le transformer en bibliothèque.

## ?

### Exercice

**Importez le module et tracez un graphique.**

```

import mes_fonctions_shs
import pandas as pd
tableau_ = pd.read_csv("./data/data-chap6.csv")
dep1_ = "67"
dep2_ = "68"
fig, ax = mes_fonctions_shs.draw_bubble_chart(tableau_,
                                               dep1_, dep2_, minx=40, maxx=60, miny=0, maxy=20)

```

```
plt.subplots_adjust(left=0.2, right=0.8, top=0.8, bottom=0.2)
```

## 10.4 Créer une bibliothèque

Dans la partie précédente, nous avons vu comment monter en généralité pour rendre une opération indépendante du contexte initial afin de la réutiliser. Tout d'abord nous avons construit une fonction, puis nous l'avons mis dans un module à importer. Cependant, cela présente des limites. En particulier, le fichier doit être dans le dossier courant, ce qui n'est pas toujours adéquat. Il faut copier/coller à la main ce fichier si nous souhaitons l'utiliser ailleurs. N'est-il pas possible d'intégrer directement notre code à Python ?

Dans le cas où vous utilisez régulièrement les mêmes fonctions dans différents contextes, une étape intéressante peut être de construire une bibliothèque dédiée. On parle de créer un *paquet* afin que celui-ci soit installable et importable au même titre que toute autre bibliothèque que vous avez installée, ce qui facilite ensuite sa distribution sur internet.

Il est probable que vous rencontriez rarement cette situation, qui se limite à quelques usages avancés. Cependant, savoir que c'est une possibilité peut être intéressant dans le cadre d'un travail collaboratif, ou tout simplement pour comprendre la logique derrière la communauté Python qui vit grâce à la production de bibliothèques par ses nombreux contributeurs. Nous allons présenter une des manières de construire un paquet Python. D'autres méthodes existent, mais celle-ci a l'avantage d'être simple et rapide. Attention : pour utiliser cette méthode, il est nécessaire d'utiliser le logiciel de versionnement *Git*.

Un paquet est plus complexe qu'un simple fichier Python. Il correspond à une architecture de fichiers et de dossiers. La structure de projet la plus simple est la suivante, composée de fichiers (comme `LICENSE`) et de dossiers, comme `mes_fonctions` :

```
mes_fonctions
LICENSE
pyproject.toml
readme.md
mes_fonctions
    __init__.py
```

Des outils Python facilitent la production de bibliothèques. Nous utilisons ici *Flit* pour gérer le paquet, qui va créer automatiquement les fichiers `LICENSE` et `pyproject.toml`. Ces deux fichiers contiennent des informations de description. L'utilisateur doit définir le contenu des deux fichiers :

- ❖ `Readme.md` contient souvent les instructions de base sur la manière d'installer et utiliser le paquet ;

- ⊕ `__init__.py` est un nom spécial reconnu par Python et contient le code qui précédemment était le fichier Python.

L'utilisation de Flit se fait en ligne de commandes. Voici le déroulé (`mkdir` crée un nouveau dossier ; `touch` un nouveau fichier texte) :

```
$ mkdir mes_fonctions
$ cd mes_fonctions
/mes_fonctions $ mkdir mes_fonctions
/mes_fonctions $ touch mes_fonctions/__init__.py
/mes_fonctions $ mes_fonctions/__init__.py
/mes_fonctions $ flit init
Module name [mes_fonctions]:<enter>
Author: Votre Nom<enter>
Author email: votre-email@mail.fr<enter>
Home page: https://optionnel.fr/<enter>
Choose a license (see http://choosealicense.com/ for more info)
1. MIT - simple and permissive
2. Apache - explicitly grants patent rights
3. GPL - ensures that code based on this is shared with the same terms
4. Skip - choose a license later
Enter 1-4: 1 <nous suggérons 1>
```

Written `pyproject.toml` ; edit that file to add optional extra info.

```
$ flit install --symlink
Extras to install for deps 'all': {'none'}
I-flit.install
Symlinking mes_fonctions ->
...../python3.x/site-packages/mes_fonctions I-flit.install
```

Après l'exécution de ce script, Flit a créé l'architecture décrite précédemment. Vous pouvez à ce moment-là modifier le fichier `__init__.py`, et voilà ! Vous pouvez ensuite compléter cette bibliothèque en ajoutant de nouveaux modules qui eux-mêmes contiennent de nouvelles fonctions et objets. Il serait maintenant possible de mettre cette nouvelle bibliothèque en ligne sur `pypi.org` et pouvoir ensuite l'installer avec `pip`. Envoyer une nouvelle bibliothèque est facile avec ce site. Et la boucle est bouclée. Vous pouvez installer votre bibliothèque.

Cette présentation a surtout pour objectif de vous montrer que toutes les étapes sont faisables. Dans certains cas, cela peut être pertinent d'avoir pour objectif le développement d'une nouvelle bibliothèque. Actuellement, les besoins en SHS sont encore loin d'être tous remplis.

## 10.5 Collaborer autour du code

À chacune des étapes, du Notebook à la bibliothèque, vous pouvez être amené à collaborer avec d'autres personnes, soit sur une analyse de données en cours, soit sur l'écriture de code. Deux situations peuvent se rencontrer.

La première est de vouloir mettre en place un travail collaboratif. De nombreuses solutions permettent de travailler collectivement sur du code, la plus connue dans le domaine de l'*Open Source* est *Github* qui relie le gestionnaire de versionnement *Git* à une interface internet permettant d'accéder au code et à la documentation. Gratuit si vous rendez votre code public, Github permet de créer un dépôt de votre code. Celui-ci peut être téléchargé localement par chaque utilisateur (cloné), être modifié. Ces modifications peuvent ensuite être proposées (envoyées, ou *push*) pour modifier le code commun sur le dépôt et ainsi faire progresser collectivement un projet.

De manière schématique, un contributeur peut proposer une *pull request*, qui correspond à une modification du code du dépôt. Cela peut être par exemple l'ajout d'une fonction, la correction d'un bug, ou encore la réécriture d'un fichier. Le propriétaire du dépôt peut vérifier cette proposition avant de l'accepter ou la rejeter. Ce mode de travail collaboratif permet de faire avancer rapidement des projets. Cette logique contributive permettant à quiconque de proposer une solution est une des forces de l'*Open Source*.

La deuxième situation est celle dans laquelle vous voulez vous-même contribuer à un code existant, car plutôt que de développer une nouvelle bibliothèque, vous souhaitez y ajouter une fonction. La majorité des projets Python ont un dépôt *Github*, que vous pouvez cloner et modifier afin de proposer votre *pull request*. Vous pouvez aussi ouvrir un billet (*issue*) sur la page de la bibliothèque pour poser la question de la possibilité d'une modification. Prenez cependant le temps de vous renseigner en amont afin de ne pas poser une question déjà existante. L'ensemble de ces contributions et usages fait la force de Python et participe à constituer une communauté accueillante.

## 10.6 Synthèse du chapitre

Ce chapitre fait un retour rapide sur la manière d'organiser le code pour le rendre réutilisable. La gestion du code est cruciale à l'échelle individuelle pour ne pas se perdre dans ses analyses. Les commentaires ou l'organisation des différentes parties sont importants pour ne pas se perdre d'un jour à l'autre. Ils sont aussi indispensables pour collaborer avec d'autres personnes ou partager son code.

Vous pouvez construire des fonctions, des modules voire des bibliothèques dédiées facilitant la réutilisation du code. La construction d'un module dédié ou d'une bibliothèque permet ainsi de stabiliser progressivement les opérations les plus fréquentes.

# Conclusion

Ce manuel propose une introduction à la programmation Python pour les sciences humaines et sociales. Nous y présentons dans un même mouvement l'intérêt à maîtriser les rudiments de programmation, les avantages du langage Python et une application de la programmation dans le traitement des données. Nous insistons sur une philosophie minimaliste de la programmation au service de la pratique. Loin de proposer une nouvelle approche, nous nous sommes attachés à montrer la compatibilité des traitements existants avec les outils de la programmation. Apprendre à programmer ne signifie pas abandonner l'usage des logiciels existants ou de réviser entièrement sa pratique mais d'ouvrir l'espace des possibilités en ce qui concerne l'utilisation d'un ordinateur pour le traitement des données.

Dans nos activités respectives, nous avons régulièrement constaté que les approches des SHS, en dehors de leurs spécialités plus computationnelles, reposent sur d'autres priorités que le calcul numérique et la formalisation logique des procédures. Les ressources existantes de programmation sont peu adaptées et développent rarement des applications utiles pour un étudiant ou chercheur en SHS. L'algorithme a tendance à occuper une place importante au détriment de la mise en situation. Pour cette raison, nous avons souhaité montrer une approche de la programmation qui insiste davantage sur la manipulation des données et des ordinateurs que sur la formalisation logique et algorithmique. Plus encore, dans un contexte où les ressources existantes sont encore largement en anglais, nous avons souhaité privilégier une introduction en français afin de faciliter l'apprentissage.

Programmer devient une ressource qui ouvre l'espace des possibles. Une comparaison peut être faite entre la programmation et le bricolage d'un meuble. Bien entendu, il est toujours possible d'aller acheter une armoire conçue spécifiquement pour ranger ses affaires, et dans de nombreux cas cela peut être plus facile ou fiable. Mais qu'en est-il quand vous devez adapter un meuble à une pièce dont les murs ne sont pas vraiment droits ? La pratique des SHS revient souvent à devoir adapter son approche aux saillances et irrégularités du monde réel. Bien entendu, dans certains cas, il est plus facile d'adapter une armoire existante que d'en construire une nouvelle. Mais dans d'autres cas, construire son propre meuble est la seule solution. À ce moment-là, fabriquer un coin pour stabiliser l'armoire

ou un nouveau meuble nécessite d'avoir quelques rudiments de bricolage. Il en est de même pour la programmation. Cette activité créative permet aussi d'envisager de nouvelles solutions qui ne seraient pas disponibles en magasin. Et si vous êtes très bon en bricolage, vous finissez par ne presque plus acheter de meubles du tout et tout faire vous-même, voire en faire pour d'autres.

Le traitement des données est une des applications de la programmation, qui devient centrale dans un monde numérique où de nombreuses informations peuvent devenir des données à analyser. Le langage Python ouvre aussi d'autres usages au-delà du traitement des données. Des applications plus avancées sont le développement de logiciels dédiés à certaines tâches ou encore de routine pour des dispositifs embarqués comme des capteurs. Pour cela, il peut être nécessaire de compléter certains aspects du langage Python ou des dispositifs informatiques existants. Mais la familiarisation que vous avez acquise avec le traitement des données vous donnera des bases solides pour réfléchir à ces usages.

Nous devons faire une confidence dans cette conclusion : nous n'avons pas pu couvrir l'ensemble des notions de la programmation Python. Nous n'avons pas abordé la notion de classe sinon en les mentionnant rapidement, ni celle d'héritage, ou de décorateur, ou encore de tests unitaires. Souhaitant garder une trame lisible et un apprentissage progressif, nous avons dû faire des choix, et hiérarchiser la priorité des notions sur la base de notre expérience. Ne soyez donc pas surpris si vous découvrez de nouvelles manières de programmer. Si cela devait arriver, rappelez-vous qu'il existe de nombreux styles de programmation, et que ce manuel a insisté sur une approche de script scientifique destiné à faciliter le traitement des données dans une perspective de programmation interactive. Pour autant, nous pensons avoir mis dans ce manuel le nécessaire pour que vous ayez une idée de la démarche de programmation en Python.

Alors, est-ce que tout le monde doit savoir programmer ? À l'heure où la grande majorité des transferts d'information prend place dans des dispositifs informatisés, la connaissance des bases d'un langage de programmation et du fonctionnement d'un ordinateur semble importante. Cette connaissance permet d'être en mesure de rendre compte de ces phénomènes sociaux et aux évolutions de la société. Avoir une idée de ce qu'est un programme permet de dissiper un peu l'obscurité des algorithmes qui tissent de plus en plus notre quotidien. Nous aimerions donc répondre qu'une connaissance de base de ce qu'est la programmation est utile. Pour autant, cela ne signifie pas que tout le monde doit être virtuose, ou même auto-nome. Ce manuel peut aussi être lu comme une présentation rapide de l'utilité de la programmation afin de se faire une idée générale.

La force d'un langage, et nous avons longuement insisté dessus, repose pour beaucoup sur sa communauté d'utilisateurs. À ce titre, les outils existants sont encore largement façonnés par les domaines les plus proches de l'informatique, et les SHS restent encore à l'heure actuelle les parents pauvres. Pour certains usages comme les statistiques en SHS, le langage R reste plus complet. Mais la communauté

Python se développe. Nous n'avons aucun doute que cela va continuer dans les prochaines années. En effet, ce langage est enseigné maintenant dans les lycées français et son utilisation ne cesse de progresser dans de nombreux domaines. Nous espérons que ce manuel permettra d'inciter les étudiants et praticiens des SHS à identifier de nouveaux usages possibles et de compléter les bibliothèques existantes en Python.

Cela est une bonne chose. En raison de son caractère libre, gratuit et pédagogique, Python permet d'être approprié par tous, facilitant le partage et la reproduction des résultats obtenus. Cela est particulièrement important dans le monde scientifique où la reproductibilité des analyses devient un enjeu crucial. Mais cela est aussi important pour former des étudiants à des outils disponibles partout facilitant l'existence d'une langue commune. Pour les enseignants, il existe de nombreux outils pédagogiques qui permettent de faciliter l'utilisation de Python en classe. Nous faisons le vœu que ce premier manuel soit suivi de nombreux autres, certains spécialisés sur des usages et d'autres plus transversaux qui intégreront les prochaines évolutions de la communauté Python en SHS afin de rendre encore plus facile et souple l'usage des outils existants.

Pour finir, et dans cet esprit collaboratif, nous sommes sensibles aux retours que vous pourriez faire sur les besoins rencontrés dans votre pratique. Nous vous invitons à échanger sur le site du manuel pour réfléchir ensemble aux outils dont les SHS ont besoin. En programmation, les outils s'abîment seulement si nous ne les utilisons pas.



## Annexe A

# Annexe : les coulisses du code

Dans ce manuel et partout ailleurs, vous êtes confronté à du code déjà écrit. Il ne reste plus aucune trace de toutes les étapes intermédiaires qui ont conduit à la version que vous avez sous les yeux. Même sur les vidéos qui montrent l'ensemble des opérations, l'intervenant a déjà une bonne idée du résultat à obtenir et peut aller directement au but.

Or, la réalité est souvent beaucoup moins linéaire. Le code est le produit final d'un ensemble d'échanges entre celui ou celle qui écrit et l'interface qui lui renvoie des erreurs. Bien souvent, le code est constitué étape par étape en résolvant des petites difficultés puis en combinant les solutions trouvées. Car s'il existe des manières plus ou moins efficaces d'écrire du code, un usage concret revient à résoudre des problèmes toujours spécifiques nécessitant votre ingéniosité.

### A.1 Règles à avoir en tête

*La première règle n° 1 :* vous allez forcément avoir des erreurs dans votre code. Bien sûr, vous en ferez moins au fur et à mesure que vous allez développer votre virtuosité, mais nous sommes désolés de vous l'apprendre, vous allez encore en faire. Même si vous êtes très bons. La conséquence logique est qu'il est important d'apprendre à trouver les erreurs et les corriger, et non pas de détourner ses yeux des messages d'erreur comme on aimerait bien pouvoir le faire.

*La deuxième règle n° 1 :* testez les opérations de votre code morceau par morceau. Un code est un assemblage plus ou moins long d'opérations. Il est généralement impossible de tout écrire d'un seul coup. Et si vous le faites, trouver les erreurs

et les corriger est plus difficile. Prenez l'habitude dès le début à décomposer votre code en opérations élémentaires que vous pouvez tester une-à-une avec des données simplifiées.

La *troisième règle n° 1* : avancez du particulier au général. Débutez par résoudre des problèmes concrets, un cas particulier, un exemple. Puis à partir de là, vous pouvez envisager de généraliser votre stratégie à des ensembles de cas plus larges. Par exemple, avant de vouloir analyser automatiquement toutes vos données, prenez le temps d'obtenir des résultats sur une colonne de votre tableau.

La *quatrième règle n° 1* : il est normal d'aller regarder des exemples et chercher de l'aide. Très rares sont ceux qui ont une mémoire exhaustive du langage, des fonctions et des manières de faire. Une fois que vous avez une idée de l'objectif, n'hésitez jamais à aller regarder des codes déjà écrits pour vous inspirer. Nous ne connaissons personne qui écrive du code sans chercher en permanence des exemples ou consulter les aides pour voir comment utiliser les différents outils déjà existants.

## A.2 Écrire un code pas-à-pas

Programmer débute par l'envie de réaliser une opération. Nous allons retracer les différentes étapes qui partent d'une idée à l'obtention d'un morceau de code efficace.

Nous allons travailler avec Jupyter pour avoir plus d'interactivité<sup>1</sup>. À chaque fois, il est possible d'écrire du code dans une cellule différente, de tester si le résultat correspond à ce que l'on souhaite, puis de copier les morceaux qui fonctionnent dans une autre cellule pour progressivement construire notre code.

Pour travailler, nous utilisons les raccourcis. Un raccourci bien utile sous Jupyter Notebook est la touche « tabulation » qui permet de compléter le nom d'une variable ou d'une fonction que vous êtes en train d'écrire, ou de proposer les possibilités existantes. Pensez à l'utiliser pour gagner du temps. Si vous avez chargé le module `math` et que vous voulez voir toutes les fonctions, il suffit d'écrire `math.` puis d'appuyer sur la touche « tabulation » pour voir toutes les fonctions disponibles dans le module. Cela permet de gagner du temps, et cela fonctionne aussi pour les variables. Plus généralement, les raccourcis clavier rendent la programmation plus agréable et fluide. Essayez-en un ou deux.

---

1. Nous prenons comme point de départ une installation de Python 3, de Jupyter Notebook et de la bibliothèque *Pandas*. Si ce n'est pas le cas, n'hésitez pas à revenir à la fiche correspondante.

### A.2.1 Identifier un objectif

Nous voulons mettre en forme les tris à plats de nos données pour pouvoir ensuite les mettre dans un rapport. Cette mise en forme est présentée toute réalisée dans le chapitre sur les statistiques descriptives. Nous revenons ici sur la manière d'écrire ce code.

Cette situation est très fréquente : après un premier traitement exploratoire, nous avons généralement envie de mettre en forme l'analyse pour pouvoir l'afficher, la sauvegarder et puis ensuite l'utiliser ailleurs<sup>2</sup>. Nous avons vu dans le chapitre sur la manipulation des données que la bibliothèque *Pandas* a une fonction qui permet de facilement obtenir le comptage des valeurs d'une colonne d'un tableau. Le comptage est une opération de base qui permet d'avoir l'information *brute*. Cependant, en SHS, nous aimons avoir un tableau qui donne à la fois le nombre d'éléments et leur proportion pour faciliter l'analyse.

Nous savons donc ce que nous voudrions obtenir, il ne reste plus qu'à écrire une fonction en Python qui prend comme information une variable et affiche un tableau bien mis en forme. Comment faire ?

### A.2.2 Procéder par étapes

La première étape est de charger des données concrètes pour avoir un exemple permettant de faire des tests. On a déjà utilisé un fichier `base-pop-2015-communes.csv` que l'on peut charger rapidement avec *Pandas*. Essayons d'écrire la ligne.

```
tableau = pd.read_csv("base-pop-2015-communes.csv")
```

```
NameError                                Traceback (most recent call last)
<ipython-input-1-4fd937a84d04> in <module>
----> 1 tableau = pd.read_csv("base-pop-2015-communes.csv")

NameError: name 'pd' is not defined
```

Raté. Le message est relativement clair et succinct, Python ne sait pas ce qu'est `pd`. Il faut en effet d'abord penser à charger la bibliothèque *Pandas*. Nous avons oublié une étape.

```
import pandas as pd
tableau = pd.read_csv("base-pop-2015-communes.csv")
```

2. Comme les formats de présentation sont dépendants de vos besoins, même un logiciel statistique ne peut produire toutes les configurations possibles. La force de la programmation est de permettre de définir exactement la forme que vous souhaitez.

```
-----  
FileNotFoundError          Traceback (most recent call last)  
<ipython-input-2-b04da0c9f3e1> in <module>  
      1 import pandas as pd  
----> 2 tableau = pd.read_csv("base-pop-2015-communes.csv")  
  
~/anaconda3/lib/python3.7/site-packages/pandas/io/parsers.py  
    674         )  
    675  
--> 676         return _read(filepath_or_buffer, kwds)  
    677  
    678     parser_f.__name__ = name  
  
~/anaconda3/lib/python3.7/site-packages/pandas/io/parsers.py in  
_read(filepath_or_buffer, kwds)  
    446  
    447     # Create the parser.  
--> 448     parser = TextFileReader(fp_or_buf, **kwds)  
    449  
    450     if chunksize or iterator:  
  
~/anaconda3/lib/python3.7/site-packages/pandas/io/parsers.py in  
__init__(self, f, engine, **kwds)  
    878             self.options["has_index_names"] =  
kwds["has_index_names"]  
    879  
--> 880             self._make_engine(self.engine)  
    881  
    882     def close(self):  
  
~/anaconda3/lib/python3.7/site-packages/pandas/io/parsers.py in  
_make_engine(self, engine)  
   1112     def _make_engine(self, engine="c"):  
   1113         if engine == "c":  
-> 1114             self._engine = CParserWrapper(self.f, **self.  
options)  
   1115         else:  
   1116             if engine == "python":  
  
~/anaconda3/lib/python3.7/site-packages/pandas/io/parsers.py in  
__init__(self, src, **kwds)  
 1889         kwds["usecols"] = self.usecols  
 1890
```

```
-> 1891         self._reader = parsers.TextReader(src, **kwds)
1892         self.unnamed_cols = self._reader.unnamed_cols
1893

pandas/_libs/parsers.pyx in pandas._libs.parsers.
TextReader.__init__()

pandas/_libs/parsers.pyx in pandas._libs.parsers.
TextReader._setup_parser_source()

FileNotFoundException: [Errno 2] File base-pop-2015-communes.csv
does not exist: 'base-pop-2015-communes.csv'
```

Raté encore. L'erreur est plus longue, et nous voyons ce que Python a tenté de faire.

Nous avons essayé de faire `tableau = pd.read_csv`. Pour cela, Python a appelé la fonction `read_csv` dans `_base.py`, qui elle-même a appelé `TextFileReader.__init__(...)` (en haut du message d'erreur) qui génère une erreur nous indiquant que le fichier n'existe pas avec `FileNotFoundException` (en bas du message d'erreur). Ceci nous donne une idée des étapes que Python a tenté de faire et nous aide à trouver une solution. Après avoir jeté un coup d'œil sur l'endroit où est le fichier, il se trouve en fait dans le dossier `data`. Se tromper dans le chemin du fichier n'est pas un problème, quand on sait lire le message d'erreur et ensuite indiquer le bon chemin.

```
import pandas as pd
tableau = pd.read_csv("./data/base-pop-2015-communes.csv")
```

Super. Vérifions la taille du tableau pour se rassurer sur sa forme. Pour cela, créons une nouvelle cellule dans le Notebook Jupyter (raccourci avec la touche « b ») et écrivons :

```
tableau.shape
```

```
(35399, 109)
```

Une des informations que nous voudrions bien avoir est le nombre de ville par région. La colonne du tableau qui donne la région de chaque ville est `REG`. La fonction `value_counts` permet d'obtenir la distribution des modalités de la colonne. Recréons une nouvelle cellule, pour garder les éléments précédents :

```
tableau["REG"].value_counts
```

```
<bound method IndexOpsMixin.value_counts of 0          84
1      84
2      84
3      84
4      84
..
35394   4
35395   4
35396   4
35397   4
35398   4
Name: REG, Length: 35399, dtype: int64>
```

Erreur, nous avons bien identifié la fonction, mais il faut indiquer que nous voulons qu'elle s'exécute, ce qui nécessite de rajouter (). Sinon, Python nous dit juste « ok, c'est bien une fonction » (<bound method ...).

```
tableau["REG"].value_counts()
```

```
44    5136
76    4488
75    4413
84    4095
32    3809
27    3739
28    2722
24    1783
52    1281
11    1277
53    1233
93    951
94    360
2     34
1     32
4     24
3     22
Name: REG, dtype: int64
```

Cette fonction nous donne la distribution des effectifs des valeurs de la colonne. Mais elle ne donne pas la proportion du total que représente chaque région. Comment calculer la proportion ? Une manière est simplement de prendre les valeurs absolues, les diviser par le total, et les multiplier par 100.

La valeur totale est le nombre de lignes de la colonne ou du tableau : `len(tableau)`. On peut essayer directement de calculer les proportions, en partant du principe que Python gère bien les conversions (que multiplier un tableau se traduit par multiplier chaque élément du tableau...). Faisons le test dans la même cellule :

```
100*tableau["REG"].value_counts()/len(tableau)
```

```
44    14.509
76    12.678
75    12.466
84    11.568
32    10.760
27    10.562
28    7.689
24    5.037
52    3.619
11    3.607
53    3.483
93    2.687
94    1.017
2     0.096
1     0.090
4     0.068
3     0.062
Name: REG, dtype: float64
```

Une telle opération produit bien le résultat souhaité (dans certains cas, les opérations peuvent se faire autrement, il est donc important de tester sur des cas précis pour comprendre la logique). Par contre, il y a un peu trop de chiffre après la virgule pour notre goût, ce qui est compliqué à lire.

Après une petite recherche sur internet avec la requête « arrondir un nombre avec Python », nous trouvons l'existence de la fonction `round` qui arrondit à la décimale souhaitée. On fait d'abord un test sur un nombre particulier dans une nouvelle cellule à part pour bien comprendre comment ça fonctionne. La fonction prend le nombre à arrondir comme premier argument, et le nombre de décimales à conserver comme deuxième argument.

```
round(10.12345,1)
```

10.1

Appliquons-le maintenant à l'ensemble de notre tableau, en ne gardant que deux décimales. On modifie la cellule précédente.

```
round(100*tableau["REG"].value_counts()/len(tableau),3)
```

```
44    14.509
76    12.678
75    12.466
84    11.568
32    10.760
27    10.562
28    7.689
24    5.037
52    3.619
11    3.607
53    3.483
93    2.687
94    1.017
2     0.096
1     0.090
4     0.068
3     0.062
Name: REG, dtype: float64
```

Super, il est bien possible d'appliquer la fonction à tout le tableau. Nous savons maintenant comment avoir le nombre total des villes et leur proportion. Il faut tout réunir dans le même tableau. Comment faire ? Une première idée est de construire un nouveau tableau *Pandas* de zéro à partir des informations. Une autre idée est de réunir ces colonnes dans un même tableau.

Essayons la première idée : construisons un tableau *Pandas* en lui donnant comme contenu les deux tableaux précédents. Créer un tableau se fait avec la fonction *DataFrame* à laquelle on passe généralement une liste comprenant les données du tableau. Dans notre cas, on se dit qu'on peut essayer de lui donner en argument une liste avec les deux colonnes calculées précédemment.

```
tab_ab = tableau["REG"].value_counts()
tab_p = round(100*tab_ab/len(tableau),2)
nouveau_tab = pd.DataFrame([tab_ab, tab_p])
```

Ce code calcule les deux colonnes de l'effectif et des pourcentages comme précédemment, puis crée une troisième variable *nouveau\_tab* en définissant un tableau *Pandas* avec *DataFrame* avec comme contenu une liste composé des deux colonnes. Pas d'erreur... regardons ce qu'on a obtenu en exécutant juste la variable dans une nouvelle cellule pour voir ce que nous avons mis dedans :

```
nouveau_tab
```

```

44      76      75 ...   1     4     3
REG 5136.000 4488.000 4413.000 ... 32.000 24.000 22.000
REG    14.510   12.680   12.470 ...  0.090  0.070  0.060

```

[2 rows x 17 columns]

C'est presque ce qu'on veut mais en ligne et pas en colonne. Après une petite recherche sur internet à propos d'inverser lignes et colonnes d'un tableau *Pandas*, nous constatons qu'il suffit de *transposer* (renverser) le tableau avec la fonction T pour obtenir ce qu'on veut. Nous pouvons rajouter la ligne :

```
nouveau_tab = nouveau_tab.T
```

De nouveau, nous pouvons regarder la variable `nouveau_tab`. Nous constatons que le nom des colonnes n'est pas explicite. Il est possible de les modifier avec le paramètre adapté :

```
nouveau_tab.columns = ["Effectif", "Pourcentage"]
```

Regardons alors le résultat :

```
nouveau_tab
```

	Effectif	Pourcentage
44	5136.000	14.510
76	4488.000	12.680
75	4413.000	12.470
84	4095.000	11.570
32	3809.000	10.760
27	3739.000	10.560
28	2722.000	7.690
24	1783.000	5.040
52	1281.000	3.620
11	1277.000	3.610
53	1233.000	3.480
93	951.000	2.690
94	360.000	1.020
2	34.000	0.100
1	32.000	0.090
4	24.000	0.070
3	22.000	0.060

Pour avoir un tableau complet, ce serait bien d'avoir une ligne en bas du tableau avec le total des colonnes. Pour cela, il faut rajouter une ligne. Modifier un tableau *Pandas* n'est pas toujours facile. Quand on regarde sur les exemples disponibles, nous nous rendons compte qu'il existe une fonction `append` qui permet d'ajouter un tableau à un tableau. Pour l'utiliser, il faudrait d'abord définir un nouveau tableau à ajouter. Cela nous semble un peu trop compliqué à utiliser.

En regardant les autres manières de faire, un exemple montre qu'il est possible de rajouter une ligne en créant une nouvelle entrée dans l'index avec `loc` (qui permet de sélectionner une ligne), le nom de l'index que l'on souhaite, et les informations pour les différentes colonnes.

```
nouveau_tab.loc["Total"] = [len(tableau),100]  
nouveau_tab
```

	Effectif	Pourcentage
44	5136.000	14.510
76	4488.000	12.680
75	4413.000	12.470
84	4095.000	11.570
32	3809.000	10.760
27	3739.000	10.560
28	2722.000	7.690
24	1783.000	5.040
52	1281.000	3.620
11	1277.000	3.610
53	1233.000	3.480
93	951.000	2.690
94	360.000	1.020
2	34.000	0.100
1	32.000	0.090
4	24.000	0.070
3	22.000	0.060
Total	35399.000	100.000

Et voilà ! Ce tableau ressemble à ce que l'on veut. Le processus a été un ensemble d'étapes pour tester chacune des fonctions, avec des essais/erreurs, mais aussi des évolutions progressives. La visualisation des variables a joué un rôle important pour progresser. N'hésitez pas à faire ces étapes intermédiaires.

Une autre manière de faire aurait été de transformer le tri à plat qui est une seule colonne (c'est une *Series* et non pas un *DataFrame*) en tableau à une colonne, puis de lui ajouter une nouvelle colonne avec les pourcentages. C'est une petite astuce qui facilite la vie :

```
tab_ab = tableau["REG"].value_counts()
tab_p = round(100*tableau["REG"].value_counts()/len(tableau),2)
nouveau_tab = pd.DataFrame(tab_ab)
nouveau_tab["Pourcentages"] = tab_p
```

### A.2.3 Généraliser vers une fonction

Nous avons réussi, étape par étape, à écrire les lignes de codes permettant d'obtenir un résultat qui nous satisfait. Si l'objectif est juste de le réaliser une fois, il suffit maintenant de sauvegarder l'information dans un fichier avec `nouveau_tab.to_excel("sauvegarde_triplat.xls")`.

Mais en fait, nous allons avoir besoin d'utiliser cette présentation plusieurs fois. Plutôt que de copier/coller systématiquement ce code, pourquoi ne pas en faire une fonction qui peut s'appliquer à n'importe quelle colonne, et qui fait la même chose : l'effectif des valeurs, leurs pourcentages, et la mise en forme ? Pour cela, il suffit de reprendre les éléments que nous avons trouvé et les regrouper dans une fonction.

Appelons-là de manière originale `triplat`. Il suffit de remplacer toutes les fois où notre colonne spécifique `tableau["REG"]` apparaît par une variable générale, par exemple appelons-là `colonne` (n'importe quel nom peut être utilisé). Puis reprenons exactement le code précédent.

```
def triplat(colonne):
    tab_ab = colonne.value_counts()
    tab_p = round(100*colonne.value_counts()/len(colonne),2)
    nouveau_tab = pd.DataFrame([tab_ab, tab_p])
    nouveau_tab = nouveau_tab.T
    nouveau_tab.columns = ["Effectif","Pourcentage"]
    nouveau_tab.loc["Total"] = [len(colonne),100]
    return nouveau_tab
```

Ce code :

- ⊕ définit une fonction à une entrée `colonne` ;
- ⊕ calcule une variable interne `tab_ab` à partir de `colonne` ;
- ⊕ calcule une variable interne `tab_p` à partir de `colonne` ;
- ⊕ définit une variable interne `nouveau_tab` qui est un tableau *Pandas* avec `DataFrame` ;
- ⊕ le transpose pour le mettre en forme ;
- ⊕ donne un nom aux colonnes ;
- ⊕ rajoute une entrée « Total » ;
- ⊕ renvoie le tableau de `nouveau_tab`.

Nous pouvons alors vérifier que dans le cas de la colonne des régions, nous obtenons bien le même résultat que précédemment. Et aussi l'utiliser maintenant sur d'autres colonnes. Par exemple, la colonne « DEP » :

```
departements = triaplat(tableau["DEP"])
departements[0:5]
```

	Effectif	Pourcentage
62	891.000	2.520
2	804.000	2.270
80	779.000	2.200
57	727.000	2.050
76	711.000	2.010

## Exercice

### Rajouter une troisième colonne avec les pourcentages cumulés.

Souvent, une troisième colonne est rajoutée dans les tri à plat : les pourcentages cumulés. Une telle présentation est particulièrement utile pour les variables ordonnées.

#### A.2.4 Documentation

La dernière étape pour construire du code réutilisable est de le documenter. Ceci est relativement simple et se fait par l'ajout d'une *Docstring* après la définition de la fonction par des triple-double-guillemets<sup>3</sup> :

```
def triaplat(colonne):
    """
    Tri à plat d'une colonne Pandas.
    - Prends une colonne (`Series`) Pandas
    - Réalise un tri à plat.
    -> Renvoie un tableau DF (Effectif et Pourcentage)
    """
    tab_ab = colonne.value_counts()
    tab_p = round(100*colonne.value_counts()/len(colonne),2)
    nouveau_tab = pd.DataFrame([tab_ab,tab_p])
```

3. En Python 3, vous observerez aussi les annotations de types `def triaplat(colonne: pd.Series) -> pd.DataFrame:`. Les extras : `pd.Series` et `-> pd.DataFrame` sont des « annotations » indiquant à la fois aux humains et machines que votre fonction est sensée prendre en entrée une `Serie` et vous donner un `DataFrame`. Une telle notation permet de faciliter la recherche d'erreurs.

```
nouveau_tab = nouveau_tab.T  
nouveau_tab.columns = ["Effectif", "Pourcentage"]  
nouveau_tab.loc["Total"] = [len(colonne), 100]  
return nouveau_tab
```

Maintenant les utilisateurs peuvent utiliser `help(triaplat)` et `triplat?` pour obtenir de l'aide sur votre nouvelle fonction. Cela vous facilitera aussi la réutilisation si vous oubliez ce que vous vouliez faire avec ce morceau de code.

## A.3 Synthèse

Cette annexe avait pour objectif de vous montrer ce qu'un manuel tend à cacher : les étapes nécessaires d'essai et de tâtonnement. Rassurez-vous : ces moments de test sont normaux, et font partie de l'expérience. Ils peuvent être un peu perturbant au début, mais facilitent à la fois l'apprentissage et l'écriture d'un code robuste.



## Annexe B

# Ressources utiles pour l'apprentissage

### En français

- ✿ Mentionnons d'abord le tutorial de la fondation Python : <https://docs.python.org/fr/3/tutorial/index.html>.
- ✿ Ce manuel peut être complété par un manuel plus général : « Python 3 - les fondamentaux du langage » Chazallet (2019).
- ✿ Les cours de programmation d'Open Classrooms peuvent permettre de compléter les notions vues dans ce manuel avec une présentation plus formelle : <https://openclassrooms.com/>.
- ✿ Vous trouverez de nombreux cours en ligne plus ou moins spécialisés. Le suivant est par exemple à destination de la biologie mais il permet de revenir sur le fonctionnement du Notebook Jupyter : [https://python.sdv.univ-paris-diderot.fr/18\\_jupyter/](https://python.sdv.univ-paris-diderot.fr/18_jupyter/).
- ✿ Des articles en français à destination des historiens sont développés sur le site *Programming Historian* <https://programminghistorian.org/fr/>. Par exemple, un article propose de rentrer en profondeur dans la manipulation de chaînes de caractères.

### En anglais

- ✿ Des cours généraux sur l'informatique sont conçus et développés par le collectif *Software Carpentry* et disponibles sur leur site <https://software-carpentry.org/lessons/>. En plus d'une introduction à Python, vous trouverez par exemple un cours d'utilisation du terminal Unix.

- ❖ Chaque année, la conférence internationale *SciPy* rend disponible ses conférences et tutoriaux en ligne. Vous pourrez donc suivre les tutoriaux donnés par des membres de la communauté sur *Youtube* avec une recherche sur le mot-clé « *scipy2019* ».
- ❖ Vous trouverez des cours plus avancés sur le site Real Python <https://realpython.com/>.
- ❖ Une introduction « pour les nuls » à Python pour la science des données Mueller & Massaron (2019).
- ❖ Une présentation plus avancée de la programmation Python pour la science des données et le script scientifique peut être trouvé dans le livre VanderPlas (2016).
- ❖ Le blog de Jake Vanderplas <http://jakevdp.github.io/> est intéressant à regarder pour apprendre des tours de main, en particulier sur l'adaptation de visualisations.
- ❖ Enfin, jetez un coup d'œil à « Apprendre à automatiser les trucs ennuyant avec Python » : <https://automatetheboringstuff.com/>.

# Bibliographie

- Barnier J. (2010). *R pour les sociologues (et assimilés)*. Lyon.
- Bastin G. & Tubaro P. (2018). Le moment Big Data des sciences sociales. *Revue française de sociologie*, **59**(3), 375.
- Bécue-Bertaut M. & Lebart L. (2018). *Analyse textuelle avec R*. Presses universitaires de Rennes, Rennes.
- Benzécri J.P. et al. (1973). *L'analyse des données*, vol. 2. Dunod.
- Boelaert J. & Ollion E. (2020). Les sommets du Palais. Analyser l'espace parlementaire avec des cartes auto-organisatrices. *Revue Française de science politique*, **70**.
- Boelart J. & Ollion E. (2018). *The great regression. Machine learning, econometrics, and the future of quantitative social sciences*.
- Brooker P. (2019). *Programming with Python for Social Scientists*. SAGE Publications, New York.
- Bruce P., Bruce A. & Gedeck P. (2017). *Practical Statistics for Data Scientists : 50+ Essential Concepts Using R and Python*. O'Reilly Media, Sebastopol.
- Cafiero F. & Camps J.B. (2019). Why molière most likely did write his plays. *Science advances*, **5**(11), 54–89.
- Cellier J. & Coaud M. (2012). *Le traitement des données en histoire et sciences sociales*. Presses universitaires de Rennes, Rennes.
- Chanvril-Ligneel F. & Le Hay V. (2014). *Méthodes statistiques pour les Sciences Sociales*. Ellipses Marketing, Paris.
- Chazallet S. (2019). *Python 3 - les fondamentaux du langage*. ENI, Paris.
- Edelmann A., Wolff T., Montagne D. & Bail C.A. (2020). Computational Social Science and Sociology. *Annual Review of Sociology*, **46**(1).

- Hosmer D., Lemeshow S. & Sturdivant R. (2013). *Applied Logistic Regression Analysis. Third Edition.* Hoboken.
- Husson, Fran ois M.L., Eric A.G., Cornillon P.A., Josse J., Rouviere L., Klutchnikoff N., Thieurmel B., J gou N. & Le Pennec E. (2018). *R pour la statistique et la science des donn es.* Presses universitaires de Rennes, Rennes.
- Lebart L., Pincemin B. & Poudat C. (2019). *Analyse des donn es textuelles.* Presses de l'Universit  du Qu bec, Montr al.
- Marres N. (2017). *Digital sociology : The reinvention of social research.* John Wiley and Sons, Cambridge.
- McKinney W. (2012). *Python for data analysis : Data wrangling with Pandas, NumPy.* O'Reilly Media, Sebastopol.
- Mueller J.P. & Massaron L. (2019). *Python for Data Science For Dummies, 2nd Edition.* Wiley, Sebastopol.
- Plutniak S. (2018). A Precursor of Digital Humanities ? The First Automated Analysis of an Ancient Economic Network (Gardin & Garelli, 1961). Implementation, Theorization, Reception. *ARCS - Analyse de r SEAUX pour les sciences sociales / Network analysis for social sciences.*
- Pornet C., Delpierre C., Dejardin O., Grosclaude P., Launay L., Guittet L., Lang T. & Launoy G. (2012). Construction of an adaptable European transnational ecological deprivation index : the French version. *Journal of Epidemiology and Community Health,* **66**(11), 982-989.
- Renisio Y. & Belin S. (2014). L'analyse des correspondances multiples au service de l'enqu te de terrain. *Actes de la recherche en sciences sociales.*
- Rogers R. (2013). *Digital methods.* MIT press, Cambridge.
- Saporta G. (2006). *Probabilit s, analyse des donn es et statistique.* Editions Technip, Paris.
- Scott J. & Carrington P. (2011). *The Sage Handbook of Social Network Analysis.* SAGE publications, London.
- Tufte E.R. (1973). *The Visual Display of Quantitative Information.* CT : Graphics press, Cheshire.
- VanderPlas J. (2016). *Python Data Science Handbook.* O'Reilly Media, Sebastopol.
- Virtanen P., Gommers R., Oliphant T.E., Haberland M., Reddy T., Cournapeau D., Burovski E., Peterson P., Weckesser W., Bright J., van der Walt S.J., Brett M., Wilson J., Millman K.J., Mayorov N., Nelson A.R.J., Jones E., Kern R., Larson E., Carey C.J., Polat ., Feng Y., Moore E.W., VanderPlas J., Laxalde

D., Perktold J., Cimrman R., Henriksen I., Quintero E.A., Harris C.R., Archibald A.M., Ribeiro A.H., Pedregosa F. & van Mulbregt P. (2020). SciPy 1.0 : fundamental algorithms for scientific computing in Python. *Nature Methods*, **17**(3), 261–272.



# Index des fonctions

add_edge (networkx) .....	278	dissolve (geopandas) .....	288
add_node (networkx) .....	278	distplot (sns) .....	194
agg (pandas).....	151, 234	drop (pandas) .....	111
annotate (matplotlib) .....	188	dropna (pandas) .....	116
append (liste) .....	39	enumerate .....	50, 56
apply (pandas) .....	111, 120, 133	except .....	59
axes (matplotlib) .....	169	exp (math) .....	67
bar (matplotlib) .....	174	exp (numpy) .....	226
bar (pandas) .....	190	f_oneway (scipy) .....	160
barh (pandas) .....	190	False (dict) .....	43
bool (dict).....	43	fcluster (scipy) .....	232
boxplot (matplotlib) .....	178	fillna (pandas) .....	116, 139
boxplot (pandas).....	192	findall (regex) .....	243
bs4 .....	249	fit_transform (sklearn) .....	207
capitalize (str).....	90	float .....	90
catplot (sns).....	193	floor .....	66
chi2_contingency (scipy) .....	157	for .....	49
columns (pandas) .....	106, 109	format (str) .....	91
conda .....	68	FrenchStemmer (nltk) .....	265
corr (pandas) .....	150	geometry (geopandas) .....	286
cos (math) .....	67	get_dummies (pandas) .....	214, 225
count (str).....	94	get_xticks (matplotlib) .....	170
crosstab (pandas) .....	146	Graph (networkx) .....	278
crs (geopandas) .....	288	grid (matplotlib) .....	188
cut (pandas) .....	136	groupby (pandas) .....	151
DataFrame (pandas).....	104, 122	head (pandas) .....	105
def .....	52	heatmap (sns) .....	196
describe (pandas) .....	143	hist (matplotlib) .....	171, 181
dict .....	40	hist (pandas) .....	190
Dictionary (gensim) .....	268		

<b>hline</b> (matplotlib) . . . . .	188	<b>OLS</b> (statsmodel) . . . . .	220
<b>if-else</b> . . . . .	47	<b>open</b> . . . . .	80, 89
<b>in</b> . . . . .	72	<b>os</b> . . . . .	83
<b>index</b> (pandas) . . . . .	105	 	
<b>input</b> . . . . .	28	<b>PCA</b> (sklearn) . . . . .	207
<b>interval</b> (scipy) . . . . .	156	<b>pearsonr</b> (scipy) . . . . .	150
<b>isin</b> (pandas) . . . . .	117	<b>Phrases</b> (gensim) . . . . .	269
<b>isnull</b> (pandas) . . . . .	113, 117	<b>pie</b> (matplotlib) . . . . .	174
<b>items</b> (dict) . . . . .	43	<b>pie</b> (pandas) . . . . .	190
<b>iterrows</b> (pandas) . . . . .	115	<b>pip</b> . . . . .	68
 		<b>plot</b> (geopandas) . . . . .	285
<b>join</b> (geopandas) . . . . .	290	<b>plot</b> (matplotlib) . . . . .	175
<b>jointplot</b> (sns) . . . . .	194	<b>plot</b> (pandas) . . . . .	139
 		<b>Point</b> (shapely) . . . . .	287
<b>keys</b> (dict) . . . . .	41, 43	<b>Polygon</b> (Polygon) . . . . .	287
<b>KMeans</b> (sklearn) . . . . .	229	<b>pop</b> (list) . . . . .	86
 		<b>sort</b> (list) . . . . .	86
<b>LdaModel</b> (gensim) . . . . .	273	<b>pow</b> (math) . . . . .	67
<b>len</b> . . . . .	28, 42, 85	<b>print</b> . . . . .	22, 25
<b>linkage</b> (scipy) . . . . .	231	<b>pyplot</b> (matplotlib) . . . . .	169
<b>list</b> . . . . .	38	 	
<b>lmplot</b> (sns) . . . . .	195	<b>qcut</b> (pandas) . . . . .	142
<b>load</b> (spacy) . . . . .	270	 	
<b>loc</b> (pandas) . . . . .	106	<b>random</b> . . . . .	153
<b>log</b> (math) . . . . .	67	<b>range</b> . . . . .	56
<b>log10</b> (math) . . . . .	67	<b>ranksums</b> (scipy) . . . . .	160
<b>lower</b> (str) . . . . .	90	<b>read</b> . . . . .	81
 		<b>read_csv</b> (pandas) . . . . .	313
<b>mask</b> (pandas) . . . . .	117	<b>read_excel</b> (pandas) . . . . .	103
<b>math</b> . . . . .	66	<b>read_spss</b> (pandas) . . . . .	103
<b>max</b> . . . . .	56	<b>readlines</b> . . . . .	82
<b>max</b> (pandas) . . . . .	144	<b>regex</b> . . . . .	95
<b>MCA</b> (prince) . . . . .	214	<b>remove</b> (liste) . . . . .	39
<b>mean</b> . . . . .	144	<b>replace</b> (pandas) . . . . .	116
<b>mean</b> (pandas) . . . . .	140	<b>replace</b> (str) . . . . .	90, 94
<b>median</b> . . . . .	144	<b>requests</b> . . . . .	70
<b>median</b> (pandas) . . . . .	140	<b>requests.get</b> . . . . .	72
<b>min</b> . . . . .	56	<b>reset_index</b> (pandas) . . . . .	109
<b>min</b> (pandas) . . . . .	144	<b>return</b> . . . . .	52
<b>mkdir</b> . . . . .	83	<b>round</b> . . . . .	135
<b>mode</b> (pandas) . . . . .	140	 	
 		<b>sample</b> (pandas) . . . . .	155
<b>normaltest</b> (scipy) . . . . .	155	<b>savefig</b> (matplotlib) . . . . .	171, 188
 		<b>scatter</b> (matplotlib) . . . . .	177

---

<b>scatter</b> (pandas) .....	190, 199	<b>T</b> (pandas) .....	138, 319
<b>Series</b> (pandas) .....	122	<b>text</b> (matplotlib) .....	188
<b>set</b> .....	44	<b>TfidfModel</b> (gensim) .....	268
<b>set_index</b> (pandas) .....	108	<b>tight_layout</b> (matplotlib) .....	188
<b>set_title</b> (matplotlib) .....	171	<b>to_csv</b> (pandas) .....	123
<b>set_xticks</b> (matplotlib) .....	170	<b>to_excel</b> (pandas) .....	123
<b>shape</b> (pandas) .....	105	<b>True</b> (dict) .....	43
<b>show</b> (matplotlib) .....	171	<b>try</b> .....	59, 250
<b>sklearn</b> .....	206	<b>ttest_ind</b> (scipy) .....	159
<b>split</b> (str) .....	88	<b>tuple</b> .....	44
<b>sqrt</b> (math) .....	67	<b>type</b> .....	35
<b>StandardScaler</b> (sklearn) .....	207	 	
<b>std</b> .....	144	<b>upper</b> (str) .....	90
<b>std</b> (pandas) .....	141	 	
<b>stopwords</b> (nltk) .....	265	<b>value_counts</b> (pandas) .....	137, 316
<b>str</b> .....	36	<b>values</b> (dict) .....	43
<b>subplots</b> (matplotlib) .....	171, 180	 	
<b>sum</b> .....	56	<b>where</b> (pandas) .....	117
<b>summary</b> (statsmodel) .....	221	<b>word_tokenize</b> (nltk) .....	264
<b>suptitle</b> (matplotlib) .....	181	<b>WordCloud</b> (wordcloud) .....	271
		<b>write</b> .....	82



# Index

## A

- Abscisses ..... 191
- ACM ..... 213
- ACP ..... 206
- AFC ..... 217
- Aide ..... 31
- Aléatoire ..... 153
- Anaconda ..... 11
- Analyse en composantes ..... 204
- Analyse textuelle ..... 260
- ANOVA ..... 160
- API ..... 252
- Apprentissage automatique ..... 234
- Attribut ..... 33
- Attribution ..... 33
- Auto-complétion ..... 312

## B

- Balise ..... 241, 248
- Bash ..... 21
- BeautifulSoup ..... 249
- Bibliothèque ..... 56, 61, 74, 304
- Bibliothèque (importer) ..... 65
- Bibliothèque (installer) ..... 67
- Bigramme ..... 269
- Binaire ..... 79, 84
- Boucle (for) ..... 49
- Boxplot ..... 178, 193
- Bubble chart ..... 198

## C

- Calculer distance ..... 287
- Caractères spéciaux ..... 37

- Carte ..... 200, 283, 288
- Cellule (Notebook) ..... 23
- Chaîne de caractères ..... 36, 89
- Chemin d'accès ..... 27
- Chi2 ..... 157
- Classe ..... 32
- Classification ..... 228
- Classification hiérarchique ascendante ..... 228
- Clé (dictionnaire) ..... 41
- Collaborer ..... 306
- Colonne (Pandas) ..... 106
- Colormaps ..... 187
- Commentaire ..... 46, 303
- Comparaison ..... 43
- Condition ..... 43
- Condition (if-else) ..... 47
- Console ..... 20
- Corrélation linéaire ..... 149
- $\cos^2$  ..... 210
- Couleurs ..... 186
- Courbe ..... 175

## D

- DataFrame ..... 104, 122
- Data science ..... 2
- Déclaration ..... 52
- Décomposition en mots ..... 263
- Décrire (données) ..... 143
- Définir une variable ..... 33
- Dendogramme ..... 232
- Diagramme circulaire ..... 174, 190

Diagramme en barres ..... 173, 190  
 Dictionnaire ..... 38, 268  
 Documentation ..... 31, 70, 71, 322  
 Données (taille) ..... 246  
 Données internet ..... 246  
 Données réelles ..... 94, 239

**E**

Écart-type ..... 141  
 Échantillon ..... 155  
 Échapper un caractère ..... 96  
 Écrire (Pandas) ..... 123  
 Encodage ..... 89  
 Entier (variable) ..... 35  
 Environnements Python ..... 12  
 EOF ..... 30  
 Erreur ..... 29, 59  
 Espace ..... 33  
 Espace des noms ..... 57  
 Europresse ..... 247  
 Excel ..... 103  
 Exception ..... 29, 59, 250  
 Exécuter ..... 52  
 Expression régulière ..... 95, 243  
 Extension ..... 79

**F**

F-strings ..... 92  
 Facteur ..... 205  
 Fichier ..... 26, 79  
 Fichier (écrire) ..... 82  
 Fichier (ouvrir) ..... 80  
 Figure ..... 169, 170  
 Figure (sauvegarder) ..... 188  
 Figures multiples ..... 180  
 Filtrer (Pandas) ..... 116  
 Fonction ..... 22, 51, 250, 299, 321  
 Fonction anonyme ..... 114  
 Formatage (texte) ..... 90  
 Forme active ..... 266

**G**

Gensim ..... 268  
 GeoPandas ..... 65, 284

Gephi ..... 282  
 Gestionnaire de version ..... 295  
 GetOldTweets3 ..... 259  
 Git ..... 295, 304  
 Github ..... 306  
 Google Trends ..... 253  
 Graduation ..... 191  
 Grouper ..... 151  
 Guillemet ..... 36

**H**

Histogramme ..... 190, 194  
 HTML ..... 247  
 HTTP ..... 70

**I**

IDE ..... 21  
 Index (dictionnaire) ..... 40  
 Index (liste) ..... 38, 86  
 Index (Pandas) ..... 108  
 Indexer ..... 240  
 Indicateur ..... 132  
 Installation ..... 10  
 Interpréteur ..... 18  
 Intervalle ..... 136  
 Intervalle (liste) ..... 86  
 Intervalle de confiance ..... 154  
 IPython ..... 21  
 Iramuteq ..... 263

**J**

Joindre (Pandas) ..... 118  
 JSON ..... 255  
 Jupyter Notebook ..... 21

**K**

K means ..... 228

**L**

lambda function ..... 114  
 Langages de programmation ..... 6  
 Latent Dirichlet Allocation ..... 272  
 Latent Semantic Analysis ..... 272  
 Latin1 ..... 89  
 Légende ..... 187, 191

Lemmatisation .....	264
Lien .....	277
Lire fichier (Pandas) .....	103
List comprehension .....	54
Liste .....	38, 85
Liste emboîtée .....	87

**M**

Machine learning .....	234
Markdown .....	23
Masque .....	96
Matplotlib.....	64, 167, 169
Message d'erreur .....	29
Méthode .....	33, 56
Modalité.....	129
Mode .....	140
Module .....	61, 302
Moyenne.....	140
Mediane .....	140

**N**

N-grams .....	266
Nettoyage .....	93, 130, 132
Nettoyer .....	295
Networkx.....	63, 64, 276
Nltk .....	63, 264
Normaliser.....	206
Notebook Jupyter.....	25, 295
Nuage de mots .....	271
Nuage de points.....	177, 190, 195
Numpy .....	226
Noeud.....	277

**O**

Objet .....	32
Odd ratio .....	223, 226
Open Source .....	74
Open Street Map .....	254
Opérations.....	36
Orienté objet .....	7
Outlayer .....	144

**P**

Pandas.....	64, 101
-------------	---------

Patsy .....	223
pip .....	67
Polygone.....	287
Prédiction .....	203, 237
Prince .....	212
Procédural .....	7
Proportion .....	133
Pull request .....	306
Python (histoire) .....	7
Python-docx .....	242
PyTrends .....	253

**Q**

QDA .....	241
Qualitatif .....	240
Quantiles .....	141
Quartiles .....	141

**R**

R .....	8
R2 .....	221
Raccourci .....	23, 312
Rang .....	38
Raster .....	165
Recodage.....	112, 132, 136, 142
Recoder variable .....	50
Regex .....	243
Régression .....	219
Régression linéaire .....	220
Régression logistique .....	222, 235
Requests .....	70
Réseau (analyse) .....	275
Réseau (visualisation) .....	281
Réutiliser .....	301
ROC-AUC .....	237
Rotation labels .....	191

**S**

Scikit-learn .....	63, 206, 229
Scipy .....	62, 64, 161
Script .....	3, 16
Seaborn .....	168, 193
Sélectionner (tableau) .....	131
Series .....	104, 122

Ensemble .....	44
Shapely .....	287
Spacy .....	270
SPSS .....	103
Spyder .....	21
Statistiques bivariées .....	146
Statistiques univariées .....	137
Statsmodels .....	219, 223
String .....	36
Styles .....	197
Superposer figures .....	183
SyntaxeError .....	30

## T

Tableau croisé .....	146, 195
Tableau croisé théorique .....	157
Tableur .....	101
Taille (liste) .....	85
Terminal .....	20
Test de Student .....	159
Test normal .....	155
Test statistique .....	152
Texte .....	36, 89
Texte (figure) .....	188
tf-idf .....	266
Titre .....	191
Tokennisation .....	264
Trace d'erreur .....	29
Trait (figure) .....	188
Transposer .....	319
Tri à plat .....	137

Tuple .....	44
Tweepy .....	255
Twitter .....	252, 255
Type .....	35

## U

Unicode .....	89
Utf-8 .....	89

## V

V de Cramer .....	158
Valeur (liste) .....	86
Valeur manquante .....	116
Valeur nulle .....	36
Valeur propre .....	208
Variable .....	32, 97
Variable dépendante .....	219
Variable indépendante .....	219
Variance expliquée .....	208
Vectoriel .....	165
Version .....	7, 65
Violin plot .....	194
Visualisation .....	139
Visualisation (Pandas) .....	190
Visualisation interactive .....	201

## W

Wikipédia .....	262
Word (fichier) .....	242
Wordcloud .....	271
Workflow .....	78