

## 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 :

- ♣ ouvre le fichier ;
- ♣ prends une ligne ;
- ♣ additionne les trois éléments ;
- ♣ divise cette somme par trois ;
- ♣ garde en mémoire l'information de la ligne et la moyenne ;
- ♣ recommence avec la ligne suivante autant de fois qu'il y a de lignes ;
- ♣ 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 corrigeons 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 calculatrice. 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 débiter

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 :

- ✦ prenez le temps de faire les petits exercices d'application ;
- ✦ ne visez pas une compréhension complète du langage avant de commencer à l'utiliser : commencez par faire, en copiant et en modifiant des exemples ;
- ✦ 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> ;
- ✦ 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 et 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.

---

2. 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.

### 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 à 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.

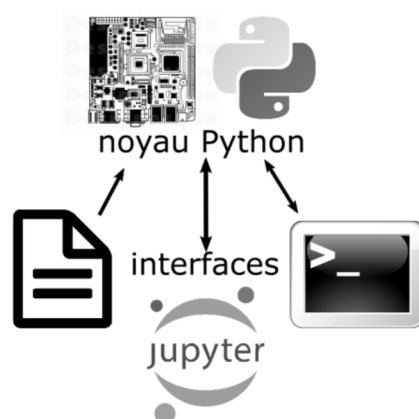


Fig. 2.1 – Relation entre l'ordinateur, le langage et les interfaces

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

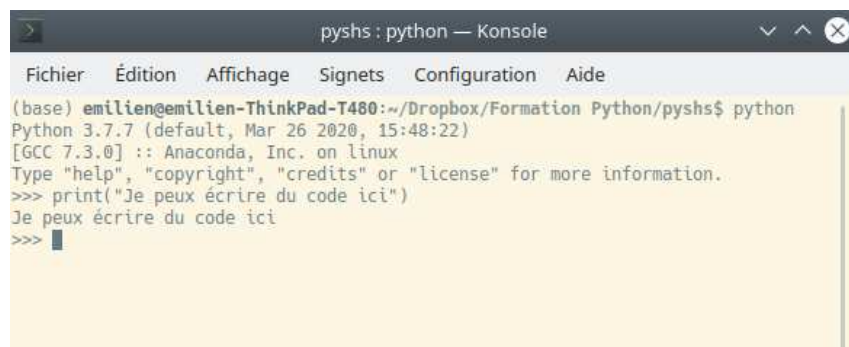
Le *terminal*, ou la *ligne de commande* est souvent l'interface privilégiée des utilisateurs plus avancés. Il est aussi appelé *shell* (coquille), car c'est une fine couche au dessus des capacités de l'ordinateur qui permet de dialoguer avec lui par des instructions. Bien qu'intimidant avec son écran 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>.



```
pyshts : python — Konsole
Fichier  Édition  Affichage  Signets  Configuration  Aide
(base) emilien@emilien-ThinkPad-T480:~/Dropbox/Formation Python/pyshts$ 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 d'aucun outil d'accompagnement.

### 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.



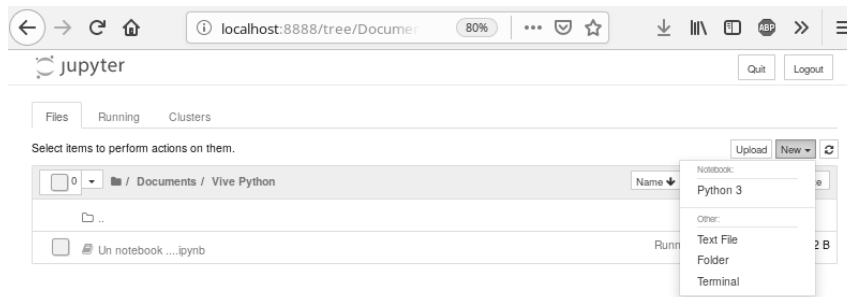


Fig. 2.3 – Page de présentation de Jupyter Notebook

### ? Exercice

#### Créez un nouveau Notebook Jupyter.

Une fois l'interface lancée, cliquez sur le bouton **New** en haut à droite et sélectionnez **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 exécutez 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.

### **i** 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.

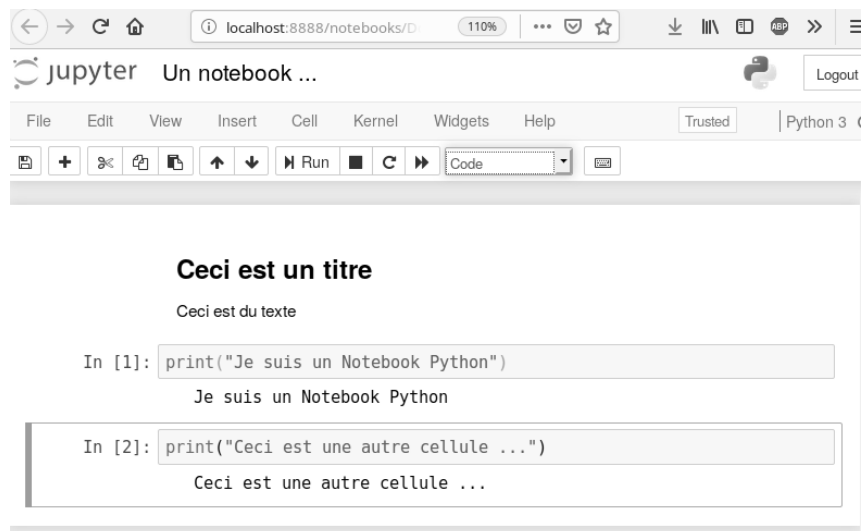


Fig. 2.4 – Un Notebook pour programmer avec ses cellules de texte et de code

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'éditer, 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 :

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

- ♣ En mode *sélection*, **b** crée une cellule sous la cellule sélectionnée ;
- ♣ 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.

### 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<sup>17</sup>.

#### ? 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>18</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.

---

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 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.

18. 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.

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

Sauvegardez le fichier.

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 :

- ✦ `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 » ;

- ✦ `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.

#### 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 **SyntaxeError**. 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>19</sup>. Il vous faudra adapter en fonction de vos besoins.

19. 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'appellez pas toutes vos variables `r`, `variable` ou `v1`, vous allez vous perdre. Ensuite, utilisez des minuscules et des majuscules, ou des tirets bas « \_ » 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 sont différentes des minuscules<sup>20</sup>.

### ❓ Exercice

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

```
pop_france_M = 67.19
```

20. 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.

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

```
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 :

- affiche la valeur contenue dans la variable `pop_france` ;
- calcule la croissance de la population pendant un an et la stocke dans la même variable ;
- 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>21</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>22</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)
```

21. 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.

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

```
"'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
```

### 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 :

- ♣ Le dernier élément de la liste `liste` peut être obtenu avec `liste[-1]` ;
- ♣ L'avant dernier élément avec `liste[-2]` ;
- ♣ 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>23</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.

---

23. 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>24</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>24</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 :

- ✦ définit une variable `exemple` qui prend comme valeur une liste ;
- ✦ transforme la liste en set avec la fonction `set`<sup>25</sup> et stocke l'information dans la variable `exemple_set` ;
- ✦ affiche le nombre d'éléments dans le set, correspondant au nombre d'éléments uniques de la liste.

---

25. 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 :

- ✦ définit une variable `exemple` qui prend comme valeur un tuple avec deux éléments ;
- ✦ affiche le deuxième élément du tuple ;
- ✦ 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>26</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>26</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 :

- ♣ *tester une condition* : par exemple, vérifier qu'un pays a une population supérieure à un autre ;
- ♣ *répéter une opération* : par exemple, afficher pour chaque pays son nom et sa population à partir d'une liste ;
- ♣ *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>27</sup>.

La particularité de Python est que ces blocs sont identifiés par un décalage de quatre espaces<sup>28</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 :

---

27. 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.

28. 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.



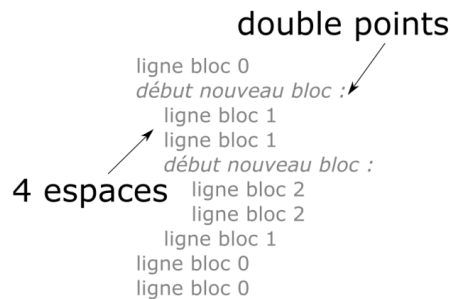


Fig. 2.5 – Organisation en bloc du code Python

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 :

- ✦ la condition évalue si la population est plus grande que dix millions ;
- ✦ si c'est le cas :
  - ★ afficher « Plus de dix millions » ;
- ✦ 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 :

- ✿ définit une variable `pop_france` avec une valeur 67 ;
- ✿ si cette variable est supérieure ou égale à 10 :
  - ★ exécute la ligne suivante qui affiche avec `print` un texte ;
- ✿ 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>29</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 :

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

---

29. 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 :

- ✦ définit une liste ;
- ✦ 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`.

#### ❓ 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` prend 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 :

- ✦ définir un nouveau dictionnaire (ou liste) qui va contenir la variable recodée ;
- ✦ prendre une à une les valeurs de l'ancien dictionnaire (ou liste) ;
- ✦ 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 :

- ♣ définit une variable `pop_reco` vide pour stocker le résultat ;
- ♣ 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` ;
- ♣ 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.

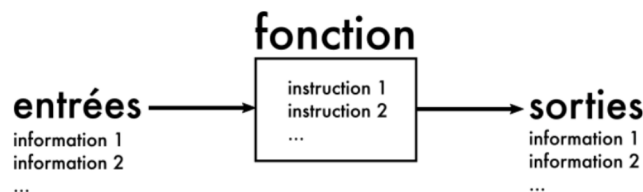


Fig. 2.6 – Structure d’une fonction

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 :

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

Une fois définie, vous pouvez utiliser cette fonction dans votre code<sup>30</sup>. Exécuter une fonction se fait en marquant les parenthèses et en donnant les arguments nécessaires.

---

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

```
recoder_population(11)
```

```
'>=10'
```

Ce code appelle la fonction `recoder_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.

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

```
pop_reco = []
for pop in liste_populations:
    pop_reco.append(recoder_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"

```

### ? 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.

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 :

- ♣ la fonction `print` pour afficher que nous avons déjà vue ;
- ♣ `input` pour une entrée au clavier ;
- ♣ 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 ;
- ♣ 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 ;
- ♣ rajoutons `sum` qui permet de faire la somme d'une liste, `max` et `min` pour le maximum et le minimum d'un ensemble ;
- ♣ `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.

### i 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>31</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 :

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

---

31. 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 des actions précises 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*.

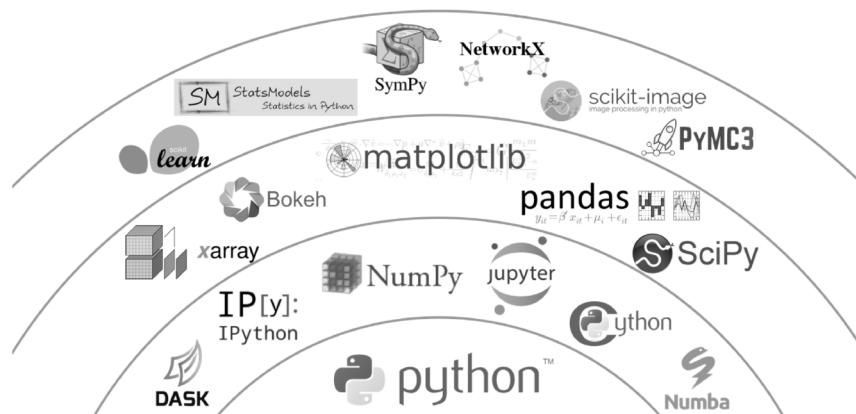


Fig. 3.1 – Schéma de l'écosystème Python (création : Jake Vanderplas)

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 ;
- ♣ *Smtplib* pour envoyer un mail par SMTP.

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.

### 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 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* représenter, 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.

### 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. Voici une petite liste des fonctions les plus souvent utilisées dans nos codes.

- ♣ `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)`.

---

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.

### 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. 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.

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**. Si vous rencontrez un problème avec une bibliothèque, pensez à la désinstaller et la réinstaller.

---

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 les bibliothèques existantes à disposition de tout le monde <https://pypi.org/>.

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 certaines dépendances indirectes. `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 pourrions 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 :

- ❖ définir les mots recherchés et les pages à consulter ;
- ❖ accéder aux pages à partir de leurs adresses ;
- ❖ récupérer le contenu de ces pages ;
- ❖ voir si les mots d’intérêt sont présents ;
- ❖ 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...

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,

---

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).



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 crochet 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.relpreload={};if(e.support=function(){var
e;try{e=t.document.createElem
```

Ce code :

- ❖ récupère la page du Monde avec `get` et stocke le contenu dans la variable `reponse` ;
- ❖ récupère uniquement le texte de la variable `reponse` avec `reponse.text` puis le stocke dans une nouvelle variable `contenu` ;
- ❖ 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 ».

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

#### 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
```

### ❓ 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][m]=presence_mot(m,p)
print(resultats)
```

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.

9. 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.

## 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.

**i 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.

**i 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.