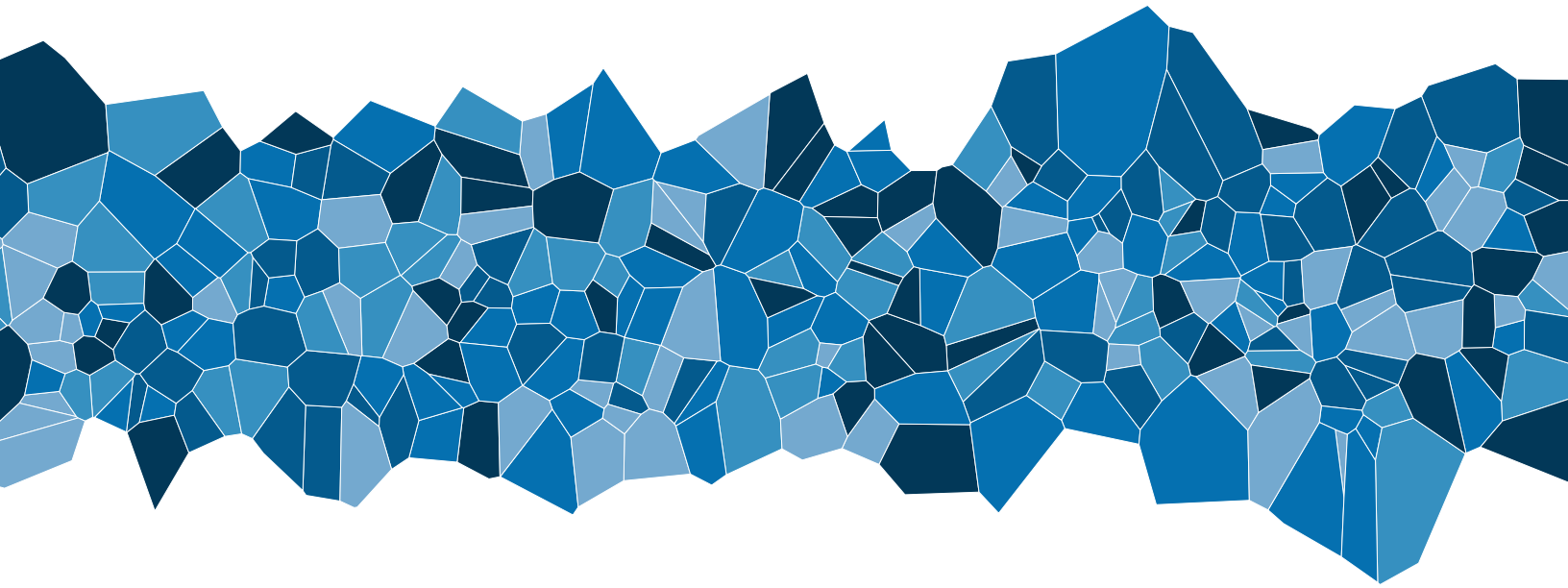
Science des données pour l'analyse criminelle avec Python

Andrew P. Wheeler



Science des données pour l'analyse criminelle avec Python

Andrew P. Wheeler

2025-10-25

Science des données pour l'analyse criminelle avec Python

ISBN 979-8-9903770-4-2 (version française ebook), ISBN 979-8-9903770-5-9 (version française broché).

Vous pouvez consulter mes autres travaux à l'adresse https://crimede-coder.com/

Table of Contents

Préface

Python est un langage de programmation informatique open source et gratuit. Il vous permet d'écrire des programmes simples (ou complexes) afin que votre ordinateur accomplisse des tâches. À titre d'exemple bref, voici un extrait de code Python qui indique la différence en nombre de jours entre deux dates. Les lignes qui commencent par `#` sont des commentaires en Python, les autres lignes effectuent différentes opérations dans le code Python. Le texte dans la zone grisée ci-dessous correspond au code Python, et le texte dans la section bleue est la sortie du programme.

```
# importing library to calculate times
from datetime import datetime

# creating two datetime objects
begin = datetime(2022,1,16)
end = datetime(2023,1,16)

# calculating the difference
dif = end - begin

# printing the result
print(dif.days)
```

```
365
```

Il s'agit donc d'un programme trivial — vous pourriez déterminer le nombre de jours entre les deux dates à l'aide de divers outils. La puissance de la programmation en Python réside dans le fait que vous pouvez écrire du code informatique pour effectuer (presque) n'importe quel calcul que vous souhaitez. Un exemple courant pour un analyste en criminalité peut consister à interroger une base de données et à créer un tableau des dénombrements de crimes cumulés depuis le début de l'année pour cette année par rapport à l'année dernière. Vous pouvez ensuite exécuter le code à volonté, et il mettra à jour les statistiques cumulatives depuis le début de l'année à la fréquence de votre choix. En pratique, un tel rapport consistera simplement à enchaîner de courts exemples de code comme ci-dessus en des séries d'opérations plus complexes.

À qui s'adresse ce livre ?

Ce livre s'adresse aux personnes n'ayant aucune expérience (ou seulement des notions de base) en programmation, mais qui souhaitent utiliser du code pour mener des analyses quantitatives et automatiser

des tâches. Le public principal visé par l'ouvrage est celui des analystes de la criminalité, mais toute personne souhaitant se lancer dans la programmation et l'analyse de données devrait trouver le contenu du livre utile. Outre les analystes de la criminalité, celles et ceux qui souhaitent faire progresser leur carrière vers des fonctions en science des données ou entreprendre des recherches de niveau master ou doctorat (et qui disposent d'une formation en justice pénale) trouveront également le livre et ses exemples utiles.

Il existe de nombreuses ressources actuelles sur l'utilisation de python sur Internet — on peut utiliser un moteur de recherche pour trouver diverses ressources entièrement gratuites en ligne. Je blogue fréquemment sur l'informatique technique sur andrewpwheeler.com, qui est gratuit et accessible à tous. Ces ressources gratuites sont toutefois souvent décousues et il est très difficile pour les débutants de les comprendre et de se lancer. Des questions comme « Comment exécuter un script python simple » ou « Comment installer une bibliothèque python » ne sont généralement pas abordées, même dans les supports d'initiation à python en ligne. Cet ouvrage vise à constituer une ressource unique pour permettre aux personnes travaillant en analyse criminelle de se lancer.

Je conçois ce livre non seulement pour présenter des exemples de code en Python, mais aussi pour décrire d'autres étapes nécessaires pour les débutants, telles que la configuration d'environnements Python et l'automatisation de tâches à l'aide de scripts shell. Ne vous inquiétez pas si vous ne comprenez pas encore de quoi il s'agit – tout sera expliqué ! Je prends même le temps de décrire une structure de projet typique, assez standard dans le développement logiciel professionnel. Ces éléments ne sont pas directement liés au code, mais ils sont nécessaires pour pouvoir se lancer avec Python et l'utiliser efficacement.

Ainsi, ce livre comble une niche — une introduction à la réalisation de tâches en Python utiles aux analystes en criminalité. Le livre contient ensuite :

- installation et création d'environnements Python
- une introduction à la programmation en Python
- manipulation de données tabulaires à l'aide de bibliothèques scientifiques
- utilisation de SQL pour interroger des bases de données
- automatisation de la création de rapports et création de tableaux et de graphiques de haute qualité

Voici les éléments indispensables, tant en matière de programmation que pour la réalisation de projets plus réalistes, qui permettent d'améliorer sa productivité dans ses tâches courantes avec Python.

Ce que n'est pas ce livre

Lorsqu'on aborde l'apprentissage de l'écriture de code et de la réalisation d'analyses de données, de nombreux livres incluent *les deux* en même temps. C'est souvent une erreur, car cela peut accroître considérablement la charge pour ceux qui souhaitent apprendre la matière. Ce livre *n'est pas* destiné à servir d'introduction à l'analyse criminelle en tant que sujet général. Pour ceux qui souhaitent apprendre les statistiques de base et les analyses réalisées par les analystes criminels, je vous suggère de consulter les supports de cours sur mon site personnel, ainsi que les ressources de l'International Association of Crime Analysts (IACA). Si la demande est suffisante, je pourrais créer de futurs livres pour couvrir plus en détail les statistiques d'introduction destinées aux analystes criminels ; faites-moi savoir si cela vous intéresse !

Ce livre a pour objectif de vous permettre de commencer à écrire du code et de l'appliquer à des tâches concrètes que les analystes en criminalité doivent réaliser. Je m'appuie sur des exemples réalistes susceptibles d'intéresser un analyste en criminalité, tels que l'envoi d'e-mails automatisés, la création de tableaux cumulés depuis le début de l'année et la réalisation de graphiques en courbes. En revanche, je n'aborde pas en détail des sujets comme la distribution de Poisson pour l'analyse des taux de criminalité ou les raisons pour lesquelles les analyses des points chauds sont importantes.

Bien que le contenu soit sans aucun doute pertinent pour *tout le monde* devant réaliser des analyses de données avec Python, j'espère que l'utilisation d'exemples plus réalistes d'analyse de la criminalité aidera à mieux illustrer l'utilité de Python pour effectuer des analyses dans vos tâches quotidiennes d'analyste en criminalité.

Pourquoi apprendre à coder ?

Les analystes criminels effectuent une grande partie de leur travail quantitatif dans des feuilles de calcul (p. ex. Excel), et un plus petit nombre utilise des outils supplémentaires, tels que des bases de données (p. ex. Access, SQLServer), des documents formatés (Word, Powerpoint, PDF) et des outils SIG (comme ESRIs ArcMap). Pourquoi s'embêter à apprendre python ? Je reconnais qu'Excel peut être utilisé pour faire des choses étonnantes avec les données, et de nombreuses tâches sont *échangeables* entre python et un (voire plusieurs) autres outils.

Les avantages de l'utilisation de la programmation, par opposition aux outils qui utilisent une interface graphique (par exemple, pointer-cliquer dans la *GUI*), sont :

- les tâches peuvent être entièrement automatisées
- les tâches sont entièrement documentées

Le premier point à puce est un argument fondé sur les économies de temps potentielles liées à l'automatisation des tâches. Supposons qu'il vous faille 30 minutes pour effectuer une tâche au quotidien. Si vous passez 100 heures à écrire du code Python pour automatiser entièrement la tâche, vous vous êtes fait gagner du temps au bout de 50 jours grâce au processus automatisé.

Pour de nombreux rapports réguliers sur lesquels travaillent les analystes de la criminalité, cet argument d'économies de temps n'est toutefois pas très convaincant. Par exemple, lorsque je travaillais comme analyste, j'avais un rapport CompStat mensuel avec divers graphiques et cartes. En utilisant des outils à interface graphique (GUI), cela me prenait peut-être 24 heures (trois jours ouvrables) pour le terminer. Une fois que j'ai écrit du code pour créer automatiquement les graphiques, c'était une tâche de moins d'une journée. Mais j'ai peut-être passé plus de 160 heures à écrire le code pour automatiser cette tâche. Il faudrait plus d'un an pour amortir cet investissement en temps.

Beaucoup de rapports réguliers rédigés par les analystes en criminalité ressembleront au second cas; ils ne seront que semi-réguliers, et donc l'argument du gain de temps en faveur de l'automatisation par le code n'est pas aussi convaincant. (L'automatisation par le code a davantage de sens, en termes de gain de temps, pour les tâches à effectuer plus fréquemment.) Même dans ces cas de rapports semi-réguliers, toutefois, je pense qu'il vaut toujours la peine d'écrire du code pour automatiser autant que possible.

Cela est dû au deuxième point de la liste — les tâches, lorsqu'elles sont écrites en code, sont par nature entièrement documentées. Cela permet à un analyste de dire rétrospectivement des choses comme «

ce chiffre a l'air étrange, comment l'ai-je calculé ? », ou, lorsqu'un nouvel analyste arrive et reprend le poste, vous pouvez dire « allez simplement voir les scripts dans le dossier X ». Disposer d'un code normalisé offre un environnement bien plus professionnel et transparent, ce qui est utile pour vous en tant qu'analyste ainsi que pour l'organisation dans son ensemble.

Cela vous permet également de faire évoluer votre travail. Si vous devez consacrer durablement du temps à une tâche particulière, même si ce n'est qu'une seule journée par mois pour un rapport donné, l'ampleur du travail que vous pouvez accomplir ne peut augmenter que jusqu'à un certain point. Pouvoir automatiser les tâches fastidieuses est une étape nécessaire pour libérer du temps afin de vous consacrer à d'autres projets. Cela vous permet même de partir en vacances, tout en continuant à satisfaire aux exigences de reporting. En fin de compte, apprendre à coder vous rendra probablement plus productif lors d'analyses de données ad hoc et améliorera votre employabilité dans un éventail plus large d'emplois (comme des postes en science des données dans le secteur privé).

Pourquoi Python ?

La section ci-dessus explique uniquement pourquoi on voudrait écrire du code pour automatiser des tâches ; elle ne détaille pas pourquoi utiliser Python spécifiquement (plutôt que, par exemple, R ou un autre logiciel statistique). En plus de Python, j'ai utilisé SPSS (un logiciel payant) et R (un autre logiciel statistique open source) assez largement au cours de ma carrière. J'ai un package R, ptools, pour des fonctions courantes utiles aux analystes en criminalité, par exemple.

J'ai presque entièrement migré mon codage personnel vers python et je n'utilise plus très souvent ces autres outils. Encore une fois, python est largement interchangeable avec R pour de nombreuses tâches, mais je préfère python à ce stade de ma carrière en raison de sa capacité à gérer des projets entiers, et pas seulement à exécuter une tâche unique. De plus, de nombreux postes de data science dans le secteur privé se concentrent presque entièrement sur python (et moins sur R). Je pense donc que, en termes de développement professionnel, surtout si vous avez pour objectif d'élargir vos compétences pour viser des postes de data science dans le secteur privé, python est un meilleur choix que R.

Il existe aussi des situations où des outils payants sont appropriés. Les logiciels statistiques comme SPSS et SAS ne stockent pas l'intégralité du jeu de données en mémoire, ce qui peut être très pratique pour certaines tâches sur de grands volumes de données. Les outils SIG d'ESRI (*système d'information géographique*) peuvent être plus pratiques pour des tâches cartographiques spécifiques (comme le calcul de distances sur réseau ou le géocodage) que de nombreuses solutions open source. (Et les outils d'ESRI peuvent également être automatisés à l'aide de code Python, donc ce n'est pas exclusif.) Cela étant, je peux m'appuyer sur Python pour près de 100 % de mes tâches quotidiennes. C'est particulièrement important pour les analystes criminels du secteur public, car vous n'avez peut-être pas de budget pour acheter des logiciels propriétaires. Python est 100 % gratuit et open source.

Comment lire ce livre

Je pense que la meilleure manière d'assimiler le contenu de ce livre est de suivre un processus en deux étapes. Votre niveau d'expérience avec Python (qu'il soit faible ou nul) influencera les contenus sur lesquels vous vous concentrerez et ceux que vous pourrez probablement ignorer. Pour tout le monde, je

suggère de *parcourir rapidement* chaque chapitre dès le départ et de comprendre les objectifs généraux que chaque chapitre cherche à enseigner.

Voici comment je consomme personnellement du contenu technique. Vous devez comprendre les objectifs de haut niveau qu'une portion de code donnée cherche à accomplir avant de pouvoir en saisir les détails techniques plus fins. Si vous ne comprenez pas les objectifs de haut niveau, il sera très difficile de comprendre les détails techniques. Il est également utile de savoir ce qui est possible — vous n'avez pas besoin de pointer-cliquer dans Excel pour régénérer ce rapport CompStat chaque mois ; vous pouvez écrire du code pour l'automatiser (voir le chapitre 10).

La deuxième partie, après le survol, dépend du fait que vous soyez néophyte en Python ou que vous ayez déjà une certaine expérience en programmation. Pour les néophytes sans expérience, je vous suggère d'étudier en détail les chapitres d'introduction 1 à 4 du livre. L'un des principaux obstacles lorsqu'on apprend à exécuter du code est le problème du "démarrage avec un exemple simple" : télécharger un programme et exécuter des commandes est difficile pour ceux qui ne l'ont jamais fait auparavant.

Cette partie — comprendre comment installer Python et exécuter une commande simple — peut être l'obstacle le plus difficile à franchir pour se lancer. Une partie de la difficulté, en tant qu'auteur, vient du fait que les systèmes de chacun sont légèrement différents et évoluent avec le temps. Les instructions pour démarrer ont tendance à être propres à votre ordinateur personnel. Si j'écris ce livre, c'est en partie parce que la plupart des ressources pour débutants n'essaient même pas d'aborder ce problème et utilisent des astuces (comme des plateformes en ligne) pour aider les gens à démarrer.

Pour accomplir des tâches réelles dont les analystes en criminalité ont besoin pour leur travail, vous ne pouvez cependant pas utiliser les plateformes en ligne. Beaucoup de personnes à qui l'on enseigne Python dans des cours universitaires utilisent ces plateformes en ligne (par exemple, si vous n'avez d'expérience qu'avec les notebooks Jupyter ou uniquement avec les notebooks Google Collab). Vous devez savoir comment télécharger Python et l'exécuter localement sur votre ordinateur personnel afin de pouvoir l'utiliser pour des tâches liées au travail. Mais ne désespérez pas si vous avez des difficultés à démarrer ! Une technique qu'utilisent les ingénieurs logiciels professionnels s'appelle la programmation en binôme — faites appel à un ami qui sait exécuter du code Python (cela peut être moi, ou quelqu'un d'autre dans votre réseau), regardez par-dessus son épaule, puis demandez-lui de regarder par-dessus la vôtre. Cela vous aidera à démarrer au chapitre 1.

Les chapitres 2 à 4 présentent les objets de base (chaînes de caractères, nombres, listes, dictionnaires) et les actions (instructions conditionnelles, boucles, substitution de chaînes). Ce sont des notions de base de Python très ennuyeuses – un peu comme l'étude de la loi normale est ennuyeuse dans un cours d'introduction aux statistiques, ou comme l'apprentissage de l'algèbre est ennuyeux en mathématiques. Elles constituent toutefois les éléments fondamentaux nécessaires pour comprendre comment écrire du code Python efficacement. Celles et ceux qui ont déjà une certaine expérience de base pourront se sentir à l'aise de survoler rapidement les chapitres 2 à 4 ; je suggère néanmoins de les examiner au moins de manière superficielle – il y a probablement quelques points que vous ne connaissez pas encore et qui seront introduits.

Le chapitre 5 est une section à laquelle même ceux qui ont une expérience introductive ne sont souvent pas exposés. Rédiger vos propres fonctions et comprendre comment importer ces fonctions constituent une étape importante pour passer d'un code de loisir à la création d'un environnement professionnel afin de développer des projets de travail sur la durée. Là encore, de nombreuses personnes qui ont suivi un cours universitaire de programmation en Python n'y sont pas exposées.

Les chapitres 6 à 9 sont axés sur la présentation d'exemples concrets de traitement et de présentation des données qui susciteront un large intérêt, non seulement pour les professionnels de l'analyse criminelle, mais aussi pour toute personne exerçant un rôle axé sur les données. Le chapitre 6 présente les deux principales bibliothèques pour travailler avec des données tabulaires – `numpy` et `pandas`. Comprendre en particulier la bibliothèque pandas est une compétence importante pour ceux qui utilisent python pour effectuer des analyses de données.

Le chapitre 7 montre comment utiliser Python pour générer des requêtes SQL. Pour ceux qui ne sont pas familiers avec SQL, *Structured Query Language*, SQL est utilisé pour extraire des données d'une base de données externe et les charger dans un DataFrame pandas. Dans ce chapitre, je présenterai également différentes instructions SQL, car, dans certains scénarios, il est préférable d'effectuer certaines tâches d'analyse des données dans la base de données *avant* de charger les données dans un DataFrame pandas en mémoire.

Le chapitre 8 présente la bibliothèque de visualisation `matplotlib` en Python. Créer des graphiques au rendu professionnel est une compétence importante pour les analystes de données. Produire des graphiques de haute qualité envoie un signal aux destinataires quant à la qualité du travail (pour les analystes criminels, il peut s'agir de policiers, de membres de l'état-major ou du grand public). Générer de tels graphiques par du code en Python est un bon moyen de maîtriser l'apparence et la cohérence des graphiques que vous produisez.

Le chapitre 9 présente les notebooks Jupyter – les notebooks offrent un environnement différent du terminal pour exécuter du code. Les notebooks Jupyter peuvent mêler des descriptions en texte brut, des cellules exécutables pour le code, et les résultats de ces exécutions de code (par exemple des graphiques et des tableaux). Ce livre, sous le capot, est compilé à partir d'une série de notebooks Jupyter. Je présente Jupyter, car c'est un moyen pratique de créer des rapports standardisés qui contiennent différents éléments d'analyse de données.

Le chapitre 10, le dernier, *organisation du projet*, aborde des aspects de gestion de projet et d'automatisation des flux de travail – les derniers éléments nécessaires pour, à partir de projets simples, tirer pleinement parti de Python afin de vous aider à accomplir votre travail d'analyste en criminalité. Maintenant que vous savez écrire du code, à quoi ressemble un projet ? Il existe des façons standard d'organiser votre projet, afin que, lorsque vous devez relancer le code, ou que d'autres doivent le faire, ils puissent en comprendre les éléments nécessaires. Cela implique notamment de créer un README contenant les informations permettant de reproduire l'environnement nécessaire à l'exécution du code, d'avoir des fonctions documentées et stockées à un emplacement spécifique, et de définir un point d'entrée clair qui exécute le code de manière automatisée.

L'ensemble du contenu du livre vise à aller au-delà de « comment écrire du code Python », pour offrir aux lecteurs une expérience de bout en bout de la création de projets réalistes qui peuvent aider les analystes de la criminalité à faire leur travail. Cela implique plus que l'exécution d'un simple script, mais aussi de savoir comment les professionnels font des choses comme interroger une base de données, créer des fonctions réutilisables et configurer des projets afin d'automatiser, au fil du temps, différentes tâches d'analyse de données.

Ce sont précisément les parties qui ne traitent pas de l'écriture de code qui font cruellement défaut dans les tutoriels actuels de programmation Python, et c'est la principale motivation pour écrire ce livre.

Mon parcours

Pendant que je préparais mon doctorat en justice pénale à SUNY Albany (entre 2008 et 2015), j'ai occupé plusieurs postes d'analyste. D'abord, j'ai travaillé au sein de la Division of Criminal Justice Services de l'État de New York. Ce poste consistait principalement à rédiger des rapports standardisés à partir de la base de données de l'historique des arrestations pénales de l'État de New York. Ensuite, pendant plusieurs années, j'ai travaillé en interne au sein du service de police de Troy (NY) en tant que leur seul analyste de la criminalité. Enfin, j'ai travaillé comme analyste de recherche au Finn Institute for Public Safety, une organisation à but non lucratif qui collaborait à des projets de recherche avec des services de police du nord de l'État de New York.

J'ai ensuite été professeur de criminologie pendant plusieurs années à l'Université du Texas à Dallas, de 2016 à 2019. Durant cette période, j'ai rédigé environ 40 publications évaluées par des pairs et j'ai collaboré à des projets quantitatifs avec des services de police à travers les États-Unis. Je présentais régulièrement ces travaux lors de la conférence de l'IACA et, pendant une brève période, j'ai été responsable du comité des publications de l'IACA.

Actuellement (depuis fin 2019), je travaille comme data scientist (dans le secteur privé) pour une entreprise du secteur de la santé. Mon travail consiste désormais à développer des logiciels, en mettant l'accent sur l'utilisation de modèles prédictifs appliqués aux données de demandes de remboursement de soins de santé. Bien que le domaine de la santé puisse sembler très différent de l'analyse de la criminalité, beaucoup de problèmes sont fondamentalement les mêmes (travailler sur des demandes de remboursement d'assurance santé n'est pas très différent du travail sur des rapports de criminalité). Parmi les exemples de ce que j'ai réalisé dans mon poste actuel dans le secteur privé figurent des modèles prédictifs permettant d'identifier quand des demandes de remboursement sont trop remboursées, ou quand des personnes présentent un risque élevé de subir un infarctus ultérieur.

Les compétences nécessaires pour construire ces modèles prédictifs ne diffèrent en rien du travail que j'ai effectué pour prévoir la criminalité dans différentes zones, ou pour identifier les délinquants chroniques présentant un risque élevé de commettre de futures violences. Ainsi, mon expérience personnelle en tant qu'analyste criminel, chercheur, puis data scientist dans le secteur privé est ce qui m'a motivé à écrire ce livre. Je souhaite que certains des travaux les plus avancés réalisés dans le milieu universitaire, ainsi que les pratiques d'ingénierie logicielle du secteur privé, se diffusent plus largement dans la profession de l'analyse criminelle. Je crois qu'un ouvrage d'introduction est la meilleure manière d'atteindre cet objectif.

Commentaires sur le livre

Pour tout retour sur le contenu de ce livre, vous pouvez m'envoyer un message à https://crimede-coder.com/contact. N'hésitez jamais à m'envoyer vos remarques, à proposer des sujets supplémentaires ou à me signaler des erreurs. Si vous êtes intéressé par des services plus directs, tels que des formations en présentiel pour vos analystes ou du conseil direct sur les projets sur lesquels vous travaillez, n'hésitez pas à m'écrire par e-mail également. Parmi les travaux que j'ai réalisés par le passé pour diverses agences de justice pénale figurent l'évaluation de programmes, le redécoupage des circonscriptions, l'automatisation de différents processus, le conseil en contentieux civil et l'élaboration de modèles prédictifs.

J'ai l'intention de produire à l'avenir du contenu Python plus avancé. Cela inclut des livres sur :

- programmation plus avancée, gestion des données et des paquets
- modélisation par régression
- applications d'apprentissage automatique en analyse criminelle
- analyse SIG avec Python

Vos retours sur le contenu et le fait de me dire ce qui vous intéresse m'aident à établir des priorités pour mes travaux futurs. Et davantage de ventes de ce livre en particulier me motivent aussi à écrire davantage – alors parlez-en à vos amis si vous l'aimez.

Les traductions française et espagnole du livre ont été principalement générées à l'aide du modèle GPT-5 d'OpenAI, puis corrigées pour corriger des erreurs grammaticales mineures. Tout commentaire sur la grammaire anglaise, française ou espagnole est apprécié.

Merci

Des remerciements s'imposent à plusieurs amis pour avoir relu les premières versions du livre et fourni des retours critiques. Renee Mitchell m'a donné des retours précoces sur les brouillons et a été l'une des impulsions décisives pour lancer cette idée et la mener à bien. Dae-Young Kim a fait preuve d'une grande assiduité en me fournissant des commentaires très détaillés sur chaque chapitre, surtout lorsque je devais expliquer le code plus en détail ou lorsque mon propos n'était pas cohérent avec les exemples de code. Et je remercie ma femme pour ses premiers retours sur les chapitres introductifs consacrés à l'écriture de code Python. (De loin, la partie la plus difficile de la programmation est de comprendre comment exécuter le code, et non d'écrire le code lui-même. Un bon test pour voir si votre documentation est suffisante consiste à demander à quelqu'un qui n'est *pas* programmeur s'il parvient à comprendre.)

Merci à tous pour votre soutien, ainsi qu'à tous ceux qui ont acheté des versions préliminaires du livre.

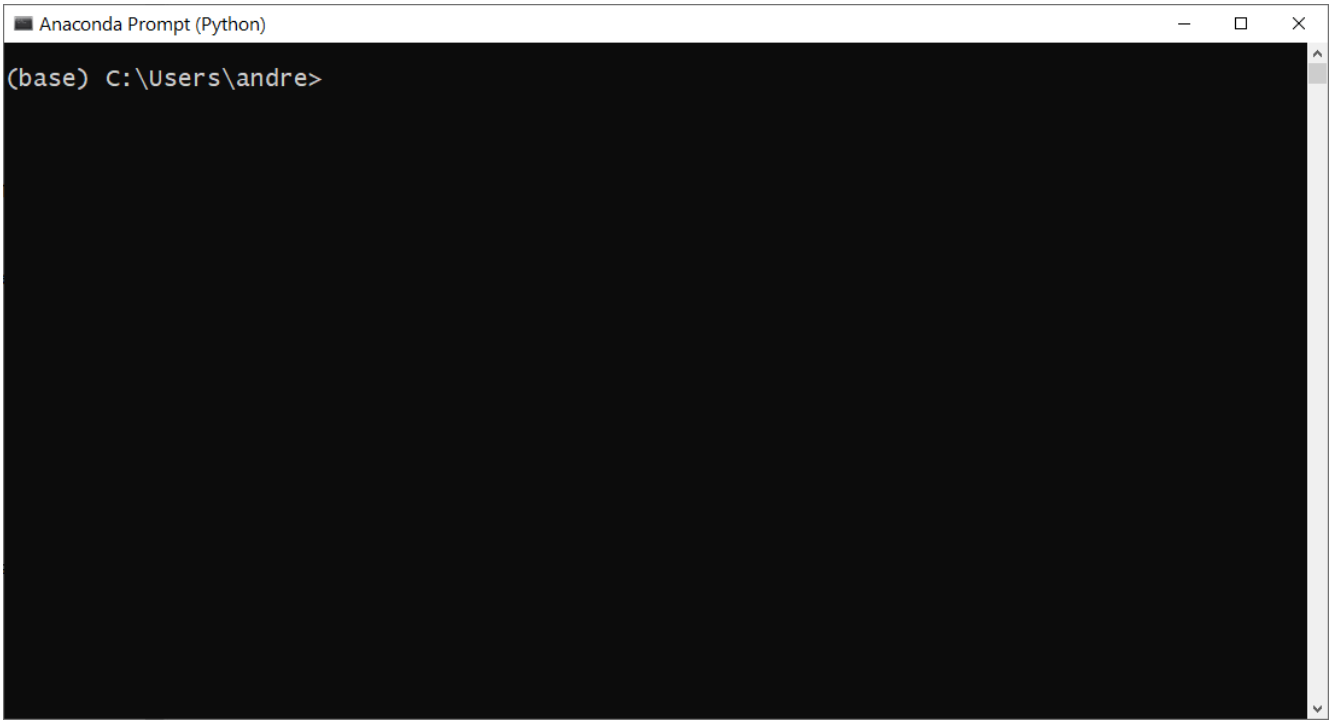
Ce livre n'est possible que grâce à diverses contributions open source. J'utilise le moteur Quarto pour générer ce livre dans différents environnements (PDF et EPUB), lequel utilise, sous le capot, LaTeX et Pandoc. Python lui-même est open source, et j'utilise abondamment des outils fournis par Anaconda. Mes remerciements à toutes celles et ceux qui, dans les coulisses, contribuent à faire tourner le monde.

1 Configuration de Python

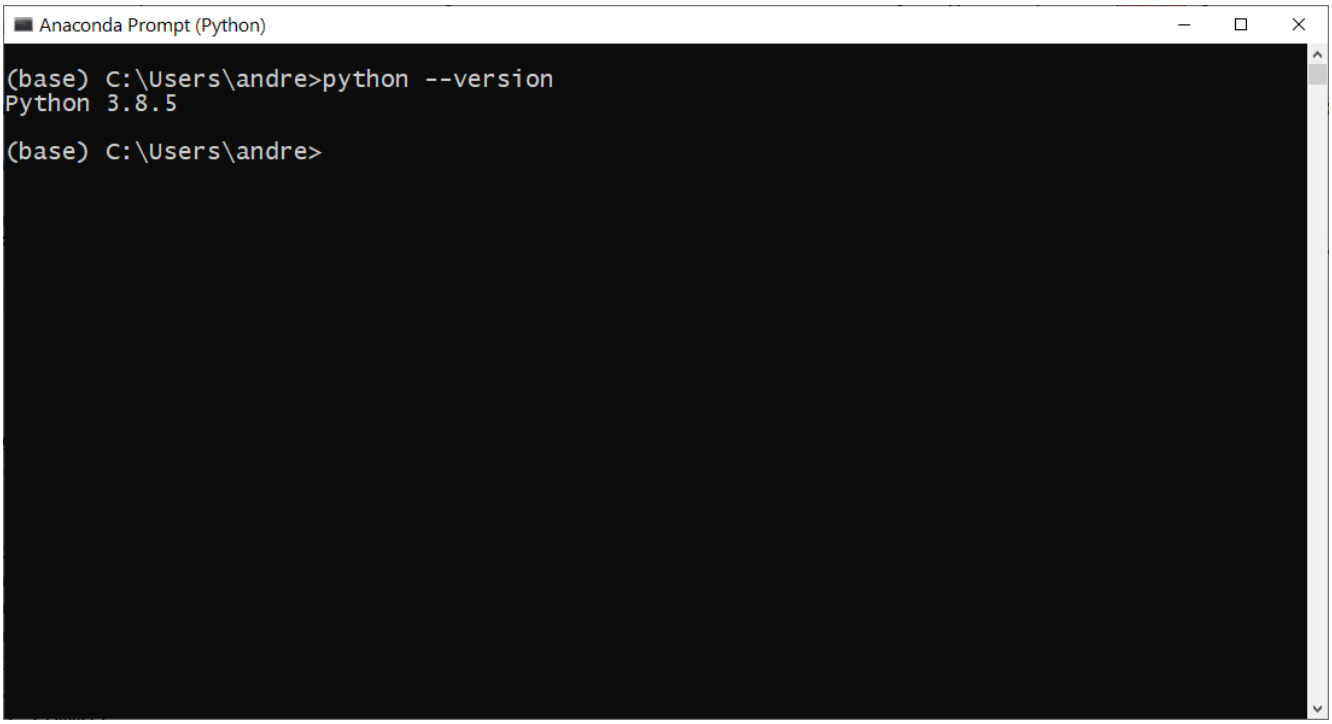
Tout d'abord, avant de pouvoir commencer à *écrire du code*, nous devons installer Python sur votre machine locale. Ma suggestion pour les analystes du crime est d'installer la version Anaconda de Python, que vous pouvez télécharger pour votre machine à l'adresse https://www.anaconda.com/download. J'écris ce livre sous Windows, mais la plupart de mes conseils devraient également s'appliquer aux utilisateurs de Mac et d'Unix (vous pouvez télécharger Anaconda et exécuter Python sur n'importe lequel de ces systèmes d'exploitation). Lorsque cela peut faire une différence, je fournirai une note en encadré. Par exemple :

> Note
>
> Lors de l'utilisation de chemins différents vers des emplacements de fichiers, les machines Windows utilisent des barres obliques inverses, p. ex. `C:\Users\andre`, tandis que les machines Mac et Unix utilisent des barres obliques, `/users/andre`.

Une fois Anaconda installé, ouvrez l'*Invite de commandes Anaconda* (traitez-la comme n'importe quel programme installé sur votre machine; sous Windows, vous pouvez parcourir les programmes qui s'affichent lorsque vous cliquez sur l'icône Windows en bas à gauche). Une fois l'Invite de commandes Anaconda ouverte, elle devrait ressembler à quelque chose comme ci-dessous.

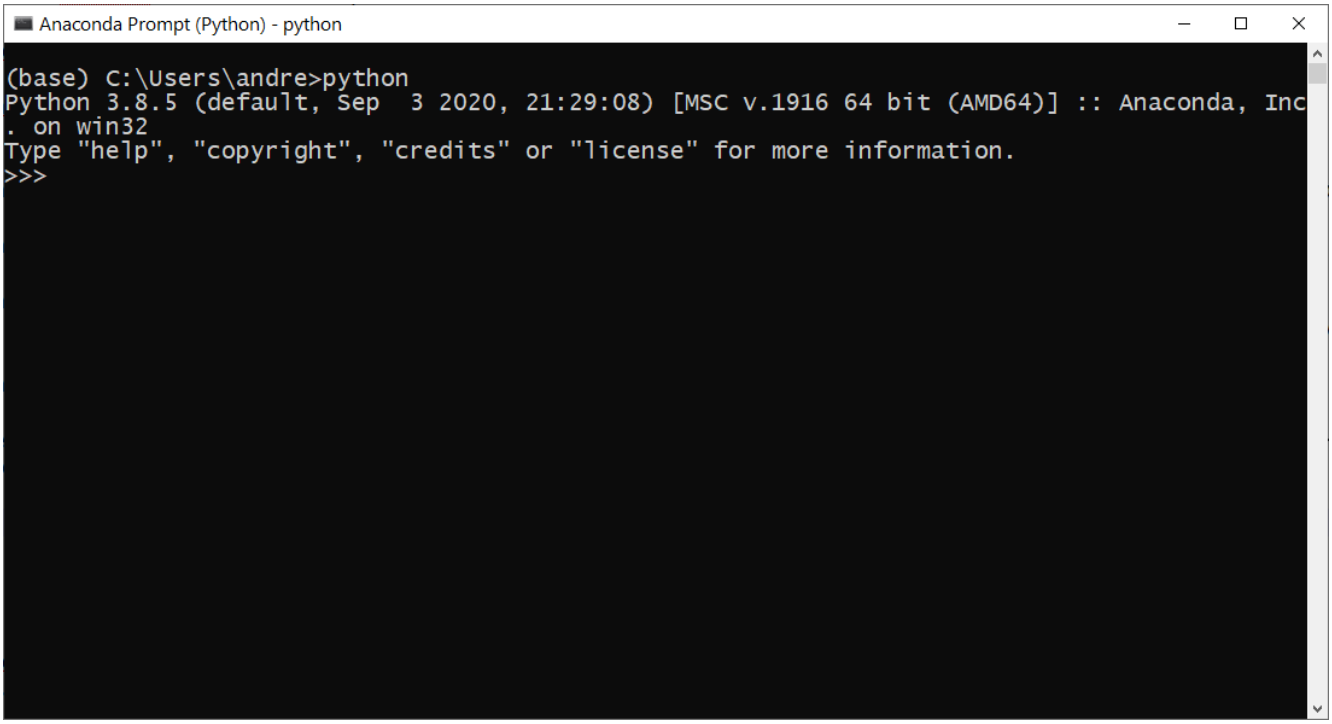


Allez-y et tapez `python --version` dans l'invite de commandes, appuyez sur Entrée, et voyez le résultat. Cela vous indiquera si vous avez installé Python correctement ! Ma version de Python est `3.8.5`. Votre version peut être différente de celle avec laquelle j'ai écrit ce livre, mais pour le contenu que je vais couvrir dans ce livre, cela ne fera aucune différence.

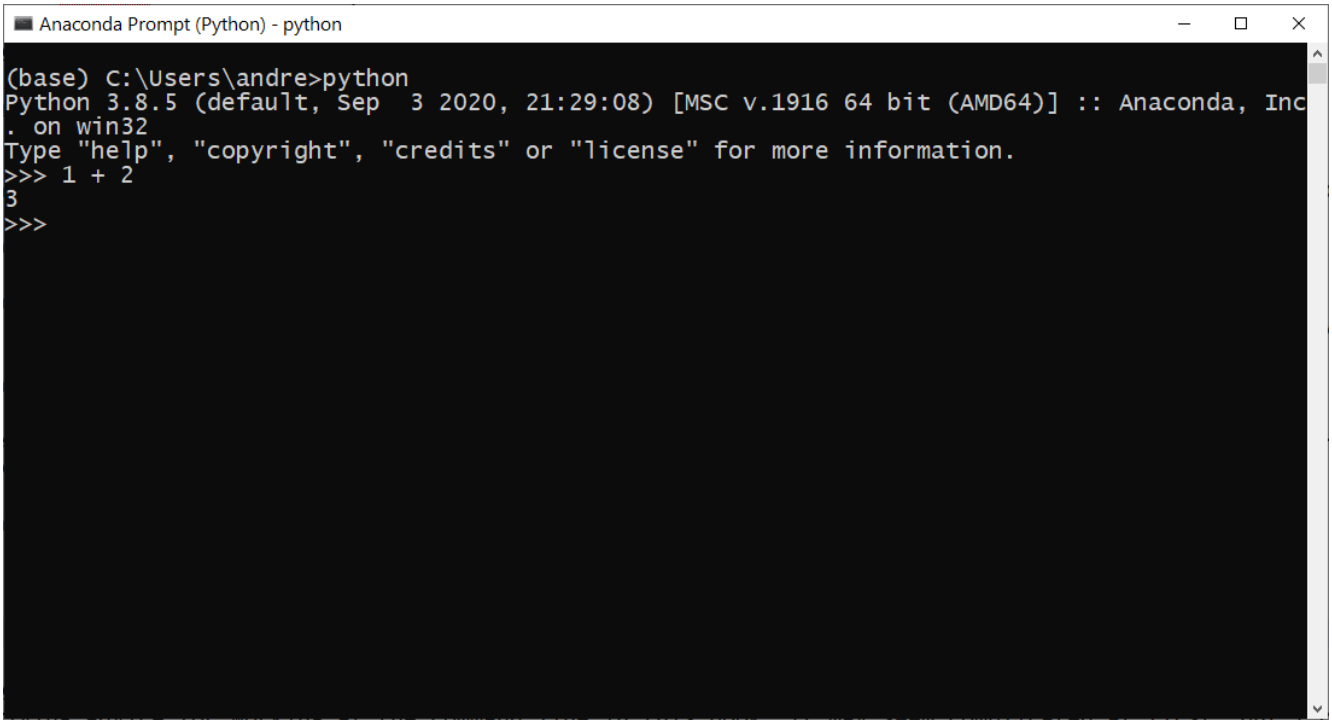


1.1 Exécution dans le REPL

Maintenant, nous allons lancer une session Python interactive, que certains appellent le REPL, boucle lire-évaluer-afficher. Tapez simplement `python` dans l'invite de commande et appuyez sur Entrée. Vous verrez alors cet écran et vous serez dans une session Python.



Le curseur dans le terminal doit se trouver à l'emplacement `>>>` de l'écran. Maintenant, à l'invite, tapez simplement `1 + 2`, appuyez sur Entrée, et il affichera la réponse :



Félicitations, vous avez maintenant écrit du code Python.

> **Note**
>
> Notez que l'emplacement du curseur dans le terminal, à ce stade, doit se trouver sur la ligne `>>>`. Vous ne pouvez pas remonter à une ligne précédente dans le terminal, comme vous pouvez le faire dans un éditeur de texte, mais vous pouvez modifier des éléments sur une seule ligne dans le terminal. Ainsi, vous pouvez taper `1 + 2`, puis, avant d'appuyer sur Entrée, appuyer sur Retour arrière, et modifier la ligne pour qu'elle devienne `1 + 3`.

1.2 Exécution d'un script Python

Créons maintenant un script Python simple et exécutons ce script depuis la ligne de commande. Tout d'abord, accédez à n'importe quel dossier de votre machine dans lequel vous pouvez ajouter un fichier (voir la note ci-dessous pour naviguer vers différents emplacements via la ligne de commande). Ici, je me suis rendu dans le dossier sur ma machine `B:\code_examples`, mais l'emplacement de votre dossier sera probablement différent. Créez un fichier simple, appelez-le `hello.py`, dans ce même dossier (vous pouvez initialiser le fichier en tapant simplement `echo "" > hello.py` à l'invite de commande, ou `touch hello.py` est plus facile sur une machine Mac/Unix. Ou, sous Windows, créez un fichier texte puis renommez l'extension en `.py` au lieu de `.txt`).

Note

Dans ce livre, je donnerai des conseils pour travailler en ligne de commande. Cela peut sembler compliqué au début, mais je n'ai mémorisé qu'une poignée de commandes. Il est important pour la gestion de projet de savoir exactement *où* vous exécutez les commandes. Voici quelques notes sur l'utilisation de `cd` pour se déplacer dans différents dossiers :

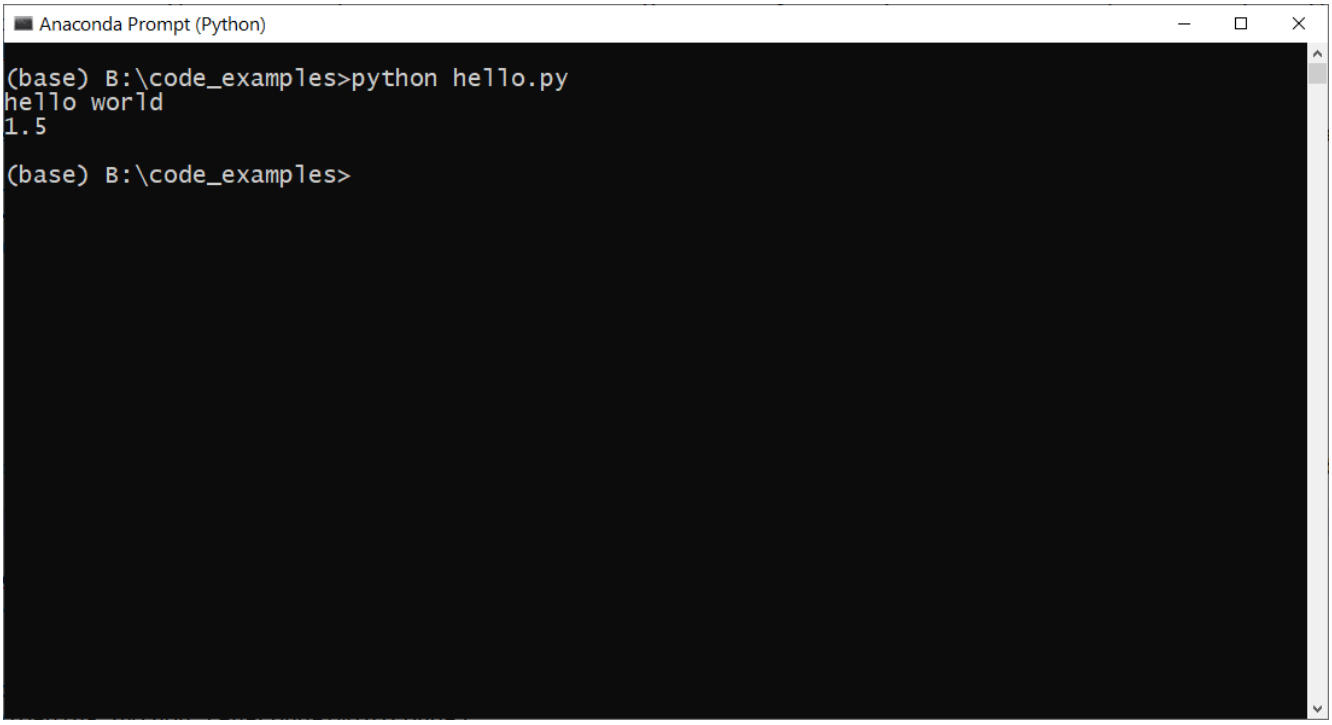
- utilisez `cd YourPath\YourFolder` pour vous déplacer entre différents dossiers, par exemple, sous Windows cela peut être `cd D:\Dropbox\Project`, tandis que sous Unix cela peut être `cd /Project/sub_folder`. Sous Windows, pour changer de lecteur, vous pouvez simplement taper la lettre de ce lecteur (pas de `cd` devant; par exemple, si vous tapez simplement `D:`, l'invite de commandes passera au lecteur `D:`).
- Si votre chemin contient un espace, il doit être entouré de guillemets lors de l'utilisation de `cd`, par exemple `cd "C:\OneDrive\OneDrive - Uni\Folder"`. Sans les guillemets, la commande `cd` renverra une erreur.
- utilisez `cd ..\` (ou `cd ../` sur Unix/Mac) pour remonter d'un dossier; par exemple, si vous êtes dans `D:\Project\sub_project` et que vous tapez `cd ../`, vous serez maintenant dans `D:\Project`.
- Il n'est pas nécessaire de taper le chemin complet pour descendre d'un seul dossier; ainsi, si vous êtes dans `D:\Project` et que vous tapez `cd .\sub_project`, vous descendrez dans le dossier `D:\Project\sub_project`.
- utilisez la commande `pwd` pour afficher le répertoire courant.

Dans ce fichier `hello.py` (qui n'est qu'un fichier texte), ouvrez-le avec l'éditeur de texte de votre choix. Saisissez ensuite ces lignes de code dans ce fichier, puis enregistrez le fichier :

```
# This is a comment line
x = 'hello world'
print(x)

y = 3/2
print(y)
```

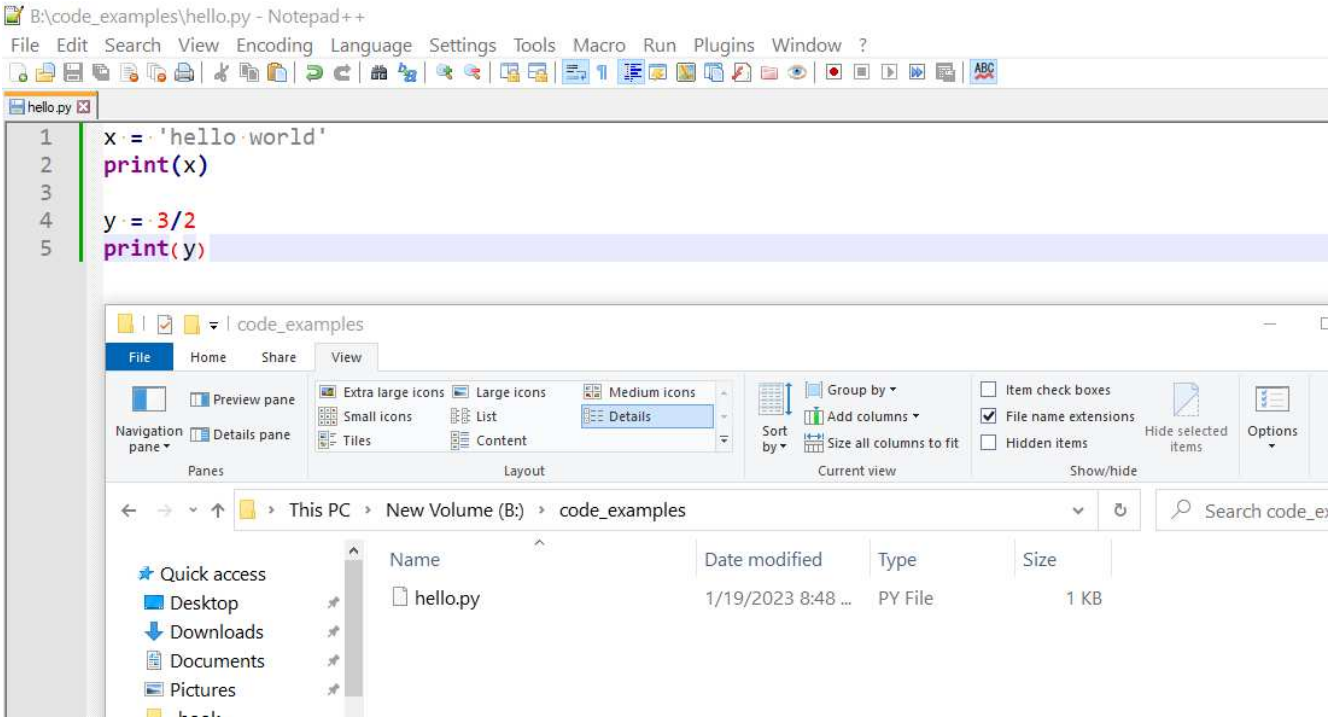
De retour à l'invite de commande, saisissez `python hello.py` puis appuyez sur Entrée. Vous devriez voir qu'il a exécuté votre fichier Python et affiché les résultats.



Certains scénarios détermineront si vous écrivez du code dans une session REPL interactive ou si vous exécutez du code via des scripts. La plupart du temps, je débogue le code initial à l'aide du REPL, puis j'enregistre le code finalisé et exécute l'ensemble des procédures au moyen d'un script.

1.3 Quelques notes supplémentaires

J'utilise Notepad++ pour écrire une grande partie de mon code. Voici à quoi ressemble le bloc de code précédent, montrant Notepad++ et le fichier lui-même :



Notepad++ sait qu'un fichier `.py` est un fichier Python et offre donc une belle mise en forme du code. Il permet aussi d'activer une option pour afficher les espaces blancs, ce qui deviendra plus important plus tard lors de l'écriture de blocs de code conditionnels en Python. (Vous ne devriez pas écrire du code dans un programme qui met en forme votre texte, comme Microsoft Word, car sa mise en forme automatique peut provoquer des erreurs dans votre code.)

Mais il existe d'autres options pour vous aider à écrire du code Python. À l'occasion, au travail, j'utilise Visual Studio Code (VS Code), qui est un IDE complet (environnement de développement intégré). Cela signifie simplement qu'il offre des fonctionnalités supplémentaires (comme un terminal intégré et la prise en charge de GitHub). VS Code propose des extensions pour Python ainsi que pour plusieurs autres langages. Un autre IDE populaire pour Python est PyCharm. Ces outils sont en grande partie interchangeables; j'aime Notepad++ pour sa simplicité.

Note

Voici quelques commandes supplémentaires de la ligne de commande que j'utilise régulièrement:

- Sous Windows, pour effacer le terminal, vous pouvez utiliser `cls`, sous Unix/Mac vous pouvez utiliser `clear`
- vous pouvez utiliser `cat file.py` pour afficher le contenu d'un fichier dans le terminal
- vous pouvez utiliser `mkdir` pour créer un nouveau dossier
- vous pouvez rediriger la sortie vers un fichier, par exemple `python hello.py > log.txt` enregistrera la sortie de la commande ci-dessus dans le fichier `log.txt` au lieu de l'afficher dans le terminal.
- si vous utilisez `python script.py >> log.txt`, cela *ajoute* la sortie au fichier journal. Ce qui est utile pour des tâches répétées qui se mettent à jour au fil du temps.

Une autre option pour écrire du code python (surtout pour les personnes qui font du calcul scientifique) est d'utiliser Jupyter Notebooks. De nombreux tutoriels python pour débutants le suggéreront. J'aurai un chapitre ultérieur montrant comment réaliser un rapport standardisé avec Jupyter Notebooks, mais je ne recommande pas cet outil aux débutants lorsqu'ils se lancent. Bon nombre des informations sur la gestion de projet que j'aborderai dans ce livre sont difficiles à gérer avec Jupyter.

Maintenant que vous disposez des bases pour exécuter du code Python, le prochain chapitre abordera plus en détail la manière d'écrire du code Python.

2 Premiers pas pour écrire du code Python

Alors que le chapitre précédent vous a montré comment exécuter du code, soit dans le REPL, soit via des scripts, ce chapitre vous initiera à l'écriture de code Python. Je vous suggère de saisir les exemples de code présentés dans les encadrés dans une session REPL pour suivre, mais vous pouvez aussi les enregistrer dans des scripts `.py` et les exécuter directement. Les parties grises correspondent à ce que vous taperiez dans le terminal, et les encadrés bleus à ce qui devrait s'afficher.

Ce chapitre présentera une introduction à la manipulation des nombres, des chaînes de caractères, des booléens, des listes et des dictionnaires. Ces *objets* constituent la base de pratiquement tout le code en Python.

> Note
>
> À la première lecture, ce chapitre (peut-être tous les chapitres) semblera contenir beaucoup d'informations et sera probablement ennuyeux. Je vous suggère de suivre en parallèle dans le REPL, mais de *survoler* les chapitres assez rapidement (en particulier les chapitres 2, 3 et 4 sur les notions de base des objets, des chaînes de caractères et des boucles). J'ai délibérément essayé d'être assez exhaustif sur les notions de base. Lorsque vous travaillerez sur des projets réels, vous devrez peut-être revenir sur certains chapitres pour comprendre et vous réapproprier les sujets. On ne devient pas expert en lisant un chapitre une seule fois, mais en écrivant du code à plusieurs reprises pour ses propres projets au fil du temps.

2.1 Valeurs numériques

Pour commencer, vous pouvez considérer le code Python comme des *objets* et des *opérations* appliquées à ces objets. Par exemple, si vous exécutez le code Python :

```
x = 1
y = x + 1
print(y)
```

```
2
```

Ici, nous commençons par créer un objet, `x`, auquel on *attribue* une valeur de 1 via le symbole `=`. Nous créons ensuite un deuxième objet, `y`, auquel est attribuée la valeur `x + 1`. Et enfin, nous `print` la valeur

de l'objet `y`. `print` est une fonction, dont le seul but est d'afficher la valeur (ou, plus précisément, la représentation sous forme de chaîne de cet objet) dans le terminal (ou tout autre emplacement que vous indiquez à Python pour afficher ses résultats).

> **Note**
>
> Dans le REPL, il est également possible de taper simplement un seul élément sur une ligne, et cela affichera la sortie dans le terminal. Ainsi, au lieu de `print(y)`, dans une session REPL, vous pouvez simplement taper `y` et cela accomplira la même chose. En revanche, dans un script, seul `print(y)` affichera la sortie dans le terminal.

Dans l'exemple ci-dessus, les objets étaient des valeurs entières. Vous pouvez également avoir des objets numériques à virgule flottante. Python, contrairement à certains langages de programmation, ne vous oblige pas à définir le type d'objet à l'avance.

```
x = -1
y = 3.2
z = x/y
print(z)
```

```
-0.3125
```

Lorsqu'on manipule des valeurs numériques, python est intelligent et convertit ici `z` en une valeur flottante, même s'il utilise un entier en entrée (même avec deux entiers, p. ex. `z = 1/2`, en python `z` sera un flottant). Vous pouvez voir qu'ici j'ai effectué une division, la plupart des opérations mathématiques sont similaires à ce que vous saisiriez sur n'importe quelle calculatrice, à l'exception que les puissances utilisent `**` au lieu de `^` :

```
# Showing off the different
# math operations
x = 2
print(x - 1)
print(x + 1)
print(x/2)
print(x*2)
print(x**3)      # x to the third power
print(x**(1/2))  # x to the 1/2 power (square root)
```

```
1
3
1.0
4
8
1.4142135623730951
```

Un opérateur spécial en Python permet de modifier la valeur numérique d'un objet. Par exemple, pour incrémenter la valeur d'un objet d'une unité, vous pourriez faire :

```
x = 1
x = x + 1
print(x)
```

```
2
```

Mais il est plus facile d'utiliser la notation spéciale `x += 1`:'

```
x = 1
x += 1
print(x)
```

```
2
```

Notez que vous pouvez également effectuer d'autres opérations mathématiques, telles que la soustraction `x -= 1`, la multiplication `x *= 2`, la division `x /= 2`, ou l'exponentiation `x **= 2`.

Les derniers exemples numériques de base à présenter sont `//`, la division entière, et `%`, l'opérateur modulo. La division entière ne renvoie que la partie entière du quotient, et le modulo correspond au reste de la division.

```
x = 5
print(x//2)
print(x % 2)
```

```
2
1
```

Dans les chapitres ultérieurs consacrés aux bibliothèques destinées à travailler avec des données tabulaires (NumPy et pandas), je traiterai plus en détail le traitement numérique des données. En effet, on ne souhaite souvent pas faire des calculs sur un seul objet, mais sur un vecteur comportant plusieurs objets.

2.2 Chaînes de caractères

Un autre objet de base que vous utiliserez souvent en Python, ce sont les chaînes de caractères. Si vous placez un ensemble de caractères entre des guillemets, cela donne un objet de type chaîne de caractères. Ici, j'effectue même une addition de chaînes, ce qui concatène les deux chaînes.

```
x = "ABC"
y = "de"
z = x + y
print(z)
```

```
ABCde
```

Les chaînes peuvent contenir n'importe quel caractère alphanumérique, elles peuvent donc contenir des représentations textuelles de nombres. Notez qu'ajouter deux chaînes n'est pas la même chose qu'ajouter deux valeurs numériques ! Cela concatène les deux chaînes.

```
x = "1"
y = "3"
z = x + y
print(z)
```

```
13
```

Vous pouvez convertir des chaînes de caractères en valeurs numériques à l'aide des fonctions `int` et `float`.

```
x = "1"
xi = int(x)
xf = float(x)
print(xi,",",xf) # can print multiple objects at once
```

```
1 , 1.0
```

Et inversement, vous pouvez convertir des valeurs numériques en chaînes à l'aide de la fonction `str`.

```
xi = 1
xf = 1.0 # if you use decimal, will be float

xsi = str(xi)
xsf = str(xf)

print(xsi + "|" + xsf) #concat the strings
```

```
1|1.0
```

> **Note**
>
> J'ai présenté trois fonctions jusqu'à présent, `print`, `int`, `float` et `str`. En Python, les fonctions prennent la forme `function(input)`, c'est-à-dire un nom particulier suivi de deux parenthèses.

Les chaînes peuvent être encadrées par des guillemets simples ou doubles. Notez toutefois que les caractères spéciaux dans les chaînes peuvent poser des problèmes. Les barres obliques inverses (antislash) dans les variables de chemin sous Windows en sont un exemple courant :

```
project_path = "C:\Project\SubFolder"
project_path # Note no print statement
```

```
'C:\\Project\\SubFolder'
```

> **Note**
>
> Ce qui est affiché dans la console n'est pas nécessairement la même représentation textuelle que l'objet lui-même. Par exemple, essayez `x = "Line1\nLine2"` puis tapez simplement `x` dans le terminal et voyez ce qui s'affiche, par rapport à taper `print(x)`. Dans cet exemple, print interprète les retours à la ligne dans la chaîne, tandis que taper simplement `x` dans une session REPL ne le fait pas.

Vous pouvez constater que Python a inséré des barres obliques supplémentaires ! Si vous voulez que la chaîne soit exactement telle que vous l'avez saisie, vous pouvez utiliser l'option de chaîne `r""`, qui signifie chaîne *réelle*.

```
project_path = r"C:\NoExtra\BackSlashes"
print(project_path)
```

```
C:\NoExtra\BackSlashes
```

Si vous voulez une chaîne multiligne en python, vous pouvez utiliser des guillemets triples. Notez que cette chaîne comporte des retours à la ligne dans l'objet chaîne de caractères.

```
long_note = '''
This is a long
string note
over multiple lines
'''

print(long_note)
```

```
This is a long
string note
over multiple lines
```

Si vous avez une chaîne très longue, par exemple une URL, dans laquelle vous ne souhaitez pas insérer de sauts de ligne, vous pouvez mettre la chaîne entre parenthèses. L'interpréteur Python la transforme simplement en une chaîne unique à la fin :

```
long_url = ("Pretend I am a super "
            "long url on a single line")

print(long_url)
```

```
Pretend I am a super long url on a single line
```

J'ai un chapitre entier, le chapitre 3, consacré à une utilisation plus avancée des chaînes de caractères.

2.3 Booléens

Les objets booléens ne peuvent avoir que deux valeurs, `True` ou `False`.

```
print(1 == 1)
print(2 == 3)
```

```
True
False
```

Vous pouvez voir que j'utilise `==` pour effectuer une comparaison d'égalité. Rappelez-vous qu'un seul `=` est destiné à l'affectation. Pour exprimer « n'est pas égal », le symbole est `!=` :

```
print('a' != 'a')
print('b' != 'c')
```

```
False
True
```

Et l'on peut également utiliser la logique « inférieur à » et « supérieur à » :

```
print(1 < 1)
print(1 <= 1)
print(2 > 1)
print(2 >= 1)
```

```
False
True
True
True
```

Notez qu'on peut aussi effectuer des comparaisons « inférieur à » avec des chaînes, par exemple, `'a' < 'b'` est `True`, mais je ne m'en sers pas très souvent. L'exemple ci-dessous montre un cas probablement non intentionnel, car il compare par accident des représentations de nombres sous forme de chaîne au lieu des nombres directement.

```
print('10' < '2')
```

```
True
```

Souvent, vous utilisez un booléen pour effectuer une logique conditionnelle dans du code Python. Vous pouvez donc avoir une instruction `if` comme ci-dessous :

```
a = 1

if a == 1:
  print(a + 1)
```

```
2
```

Une particularité de la syntaxe de Python est que l'indentation est significative. Pour indiquer que la ligne `print` se trouve à l'intérieur de l'instruction if, on ajoute plusieurs espaces au début. Le nombre d'espaces n'a pas d'importance, il doit toutefois être cohérent. (Et techniquement, vous pouvez aussi utiliser des tabulations au lieu des espaces, mais je le déconseille fortement, car cela peut rendre l'édition des fichiers pénible.)

Python utilise `if`, `elif`, `else` pour les instructions conditionnelles. Voici un exemple avec if et else :

```
a = 1

if a != 1:
  print('a does not equal 1')
  print(a + 1)
```

```
else:
  print('I am in the else part')
```

```
I am in the else part
```

La façon dont ces instructions fonctionnent est la suivante : on vérifie si la première instruction `if` est vraie. Si cette instruction est vraie, on exécute le code imbriqué dans la partie if, *puis on quitte le bloc de code*. Si l'instruction `if` s'évalue à False, on exécute alors le bloc d'instructions `else` pour tous les autres cas.

Parfois, vous souhaitez enchaîner plusieurs tests, p. ex. « si A, faire Y, sinon si B, faire X ». Pour faire cela en code Python, vous utiliseriez `if` puis `elif`.

```
a = 1

if a != 1:
  print('a does not equal 1')
  print(a + 1)
elif a == 1:
  print('I am in the elif part')
else:
  print('I am in the else part')
```

```
I am in the elif part
```

Si un `elif` est vrai, il exécute ce bloc puis quitte, comme l'instruction `if`. Ainsi, dans l'extrait de code ci-dessus, puisque l'instruction `elif` est vraie, il n'atteint jamais la partie `else` de la logique conditionnelle.

Il n'y a rien qui relie les différentes instructions conditionnelles entre elles, donc toutes les sections n'ont pas besoin de faire référence au même élément :

```
a = 1
b = 'X'

if a != 1:
  print('a does not equal 1')
elif a == 2:
  print('a equals 2')
elif b == 'X':
  print('I am in the b check elif part')
else:
  print('I am in the else part')
```

```
I am in the b check elif part
```

Et vous pouvez aussi imbriquer d'autres conditions à l'intérieur d'une condition.

```
a = 1
b = 'X'

if a != 1:
  print('a does not equal 1')
  if b == 'X':
    print('b check in first if')
else:
  print('I am in the else part')
  if b == 'X':
    print('b check in elif')
```

```
I am in the else part
b check in elif
```

Parfois, dans une partie de la condition, vous ne voulez rien faire. Dans ces cas-là, vous pouvez utiliser `pass` à l'intérieur de la condition.

```
a = 1

if a == 1:
  pass # This snippet will print nothing
else:
  print('I am in the else part')
```

En général, il est préférable de placer d'abord les conditions les plus courantes dans une série d'instructions if, et les conditions moins courantes plus loin. Ainsi, si, pour la condition la plus fréquente, vous ne faites rien et que vous n'agissez que dans des cas plus rares, c'est une façon tout à fait raisonnable d'écrire vos instructions if.

Ces exemples ne comparent que deux objets, mais vous pouvez combiner plusieurs instructions conditionnelles à l'aide de `and` et `or`.

```
# and example
a = (1 == 1) and (2 == 2)
print(a)

b = (1 == 1) and (2 == 3)
print(b)

# or example
```

```
c = (1 == 1) or (2 == 3)
print(c)
```

```
True
False
True
```

Il existe cependant des opérateurs abrégés, l'esperluette pour `and` et la barre verticale pour `or` :

```
# ampersand for and
a = (1 == 1) & (2 == 2)
print(a)

b = (1 == 1) & (2 == 3)
print(b)

# pipe for or
c = (1 == 1) | (2 == 3)
print(c)
```

```
True
False
True
```

Techniquement, les parenthèses ne sont pas nécessaires dans les exemples ci-dessus, mais je trouve qu'il est beaucoup plus facile de lire le code de cette façon et de garder les différents termes ensemble :

```
#       This is false        But this is true
a = ((1 == 2) & (2 == 2)) | (4 == 4)
print(a)
```

```
True
```

Mais la plupart du temps, si possible, je le décompose dans le code et je fais en sorte que la ligne de l'instruction if finale soit aussi simple que possible.

```
check1 = (1 == 2) & (2 == 2) # This is false
check2 = (4 == 4)            # This is true

if check1 & check2:
  print('Both check1 and check2 are true')
```

```
elif check1 | check2:
  print('At least one of check1 or check2 are true')
else:
  print('Neither check1 or check2 are true')
```

```
At least one of check1 or check2 are true
```

Parfois, on n'utilise pas les opérateurs mathématiques pour générer des expressions booléennes, mais `is` ou `is not`. L'usage le plus courant est probablement avec l'objet `None`, que l'on peut considérer comme une valeur manquante en Python.

```
x = None

if x is None:
  print('x has no value')
else:
  print('x has some value')
```

```
x has no value
```

Techniquement, lorsque vous écrivez `a is b`, cela vérifie non seulement les valeurs des objets, mais aussi que cela fait référence au même objet exact en mémoire. Techniquement, `x == None` fonctionnera ci-dessus, mais il est d'usage de l'écrire sous la forme `x is None`. Vous pouvez également vérifier la condition inverse via `is not None`:

```
x = None

if x is not None:
  print('x has some value')
else:
  print('x is None')
```

```
x is None
```

Pour un dernier exemple, on peut se passer complètement des opérateurs de comparaison, voire des instructions if. Si vous passez un objet « vide » à une instruction if, Python vérifie si l'objet contient le moindre élément et renvoie `True` si c'est le cas. Ainsi, si vous passez une chaîne vide, l'instruction if renvoie `False` ici :

```
x = ''

if x:
  print('x has some value')
else:
  print('x is an empty string')
```

```
x is an empty string
```

Cela fonctionne pour d'autres objets Python, tels que les listes et les dictionnaires vides, ce qui sera illustré dans la section suivante.

2.4 Listes

Il est important pour les débutants de comprendre les différents objets qui sont des conteneurs pour d'autres objets. Le premier est un objet *liste*. Une liste contient plusieurs autres objets, et vous créez une liste en plaçant des éléments entre des crochets et en séparant les éléments par des virgules. C'est plus facile à montrer qu'à expliquer avec des mots !

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

```
[1, 2, 3]
```

Les listes peuvent contenir différents types d'objets ; elles peuvent, par exemple, rassembler à la fois des chaînes de caractères et des valeurs numériques dans une même liste. Ici, je montre aussi que les listes peuvent contenir d'autres objets Python définis.

```
a = 1
b = 'z'
x = [a,b,3]
print(x)
```

```
[1, 'z', 3]
```

Vous pouvez répartir les éléments d'une liste sur plusieurs lignes dans l'interpréteur Python ; il sait rechercher le crochet fermant pour savoir que la saisie de la liste est terminée. Ainsi, la liste obtenue ci-dessous est exactement la même que `x = [1,2,3]`, c'est simplement une autre manière de l'écrire (il peut être judicieux de répartir des listes très longues sur plusieurs lignes pour améliorer la lisibilité de votre code).

```
# it is ok to define lists
# over multiple lines
x = [1,
     2,
     3]

# it is the same as earlier
# on a single line
print(x)
```

```
[1, 2, 3]
```

Une erreur courante lorsqu'on travaille avec des listes est d'oublier la virgule. Avec des valeurs numériques, cela entraîne souvent une erreur, mais avec des chaînes de caractères, il peut parfois n'y avoir aucune erreur, car Python concatène simplement les chaînes de manière implicite. Par exemple :

```
# This is probably not what was intended
x = ['a' 'b', 'c']
print(x)
```

```
['ab', 'c']
```

Donc ici, la virgule entre les deux premières chaînes de caractères a été omise, donc la liste résultante ne comporte que deux éléments, le premier étant `'ab'`.

> **Note**
>
> L'un des avantages à écrire des listes sur plusieurs lignes est que vous pouvez utiliser l'édition *en colonne* dans divers éditeurs de texte. Dans Notepad++, vous pouvez maintenir la touche Alt enfoncée et sélectionner plusieurs lignes pour les modifier en même temps. La plupart des éditeurs de texte avancés proposent une fonctionnalité similaire.

Vous pouvez accéder à un élément d'une liste via son indice. Notez qu'en Python, les indices de liste commencent à 0 et non à 1 ; ainsi, le premier élément d'une liste a pour indice 0, le deuxième a pour indice 1, etc.

```
y = ['a','b','c']
print(y[0])
print(y[1])
print(y[2])
```

```
a
b
c
```

Vous pouvez également utiliser des indices *négatifs* pour accéder aux éléments d'une liste dans l'ordre inverse. Ainsi, -1 accède au dernier élément d'une liste, -2 à l'avant-dernier, etc.

```
y = ['a','b','c']
print(y[-1])
print(y[-2])
```

```
c
b
```

Enfin, pour accéder à plusieurs éléments d'une liste, vous pouvez utiliser la notation de tranche. Ainsi, `x[1:3]` permet d'accéder aux 2e et 3e éléments de la liste (c'est équivalent à la notation `[,)` en mathématiques, donc l'extrémité gauche de la tranche est fermée et l'extrémité droite est ouverte).

```
y = ['a','b','c']
print(y[1:3]) # note it returns a list!
```

```
['b', 'c']
```

Vous pouvez également utiliser `x[1:]` pour indiquer, "donne-moi le 2e élément de la liste jusqu'à la fin", ou `x[:3]` pour "donne-moi les 3 premiers éléments de la liste.

On peut créer une liste vide, simplement en assignant `[]` à une valeur. Une autre astuce utile à connaître est que vous pouvez effectuer un test booléen pour vérifier qu'une liste est vide.

```
x = []

if x:
  print('The x list is not empty')
else:
  print('x is empty')
```

```
x is empty
```

Une liste vide ne contient aucun objet, donc si vous essayez d'accéder à un objet, une erreur sera renvoyée. Comme indiqué plus haut, si vous essayez d'utiliser `x[0]`, vous obtiendrez l'erreur `index out of range`.

Une autre opération booléenne sur les listes consiste à vérifier si un élément est contenu dans cette liste.

```
x = ['a','b','c']

if 'd' in x:
  print('The list has a d element')
elif 'c' in x:
  print('The list has a c element')
else:
  print('x has neither d or c')
```

```
The list has a c element
```

Une information importante à connaître sur les listes est qu'elles sont *mutables*. Que signifie cela exactement ? Cela signifie que nous pouvons modifier le contenu d'une liste. Par exemple, nous pouvons remplacer un seul élément d'une liste.

```
y = ['a','b','c']
print(y)

y[1] = 'Z'
print(y)
```

```
['a', 'b', 'c']
['a', 'Z', 'c']
```

Pour un exemple plus complexe, les listes peuvent pointer vers d'autres listes. Comme les listes sont mutables, vous pouvez modifier la liste interne ici, et la liste externe reflète ce changement.

```
x = ['a','b','c']
y = [1, x]
print(y)

# if we alter x
# y points to
# the altered x list
x[1] = 'Z'
print(y)
```

```
[1, ['a', 'b', 'c']]
[1, ['a', 'Z', 'c']]
```

Voici une manière d'y penser : la liste `y` ici ne contient pas réellement le contenu de `x`; elle se contente de *pointer vers* l'objet `x`. Donc, si l'objet `x` est modifié, elle pointe vers ce nouvel objet `x`.

Cela peut sembler entrer dans les détails, mais c'est une caractéristique importante du langage de programmation Python. Elle permet d'écrire de nombreux algorithmes de façon plus simple lorsqu'on peut modifier les listes en place.

Vous pouvez concaténer deux listes en les additionnant :

```
x = ['a','b','c']
y = [1,2]
z = x + y
print(z)
```

```
['a', 'b', 'c', 1, 2]
```

Vous pouvez également créer une liste répétée en utilisant la multiplication :

```
x = ['a','b','c']
y = [1]
print(y*3 + x*2)
```

```
[1, 1, 1, 'a', 'b', 'c', 'a', 'b', 'c']
```

Les listes possèdent plusieurs *méthodes* pour modifier leur contenu ; deux des plus courantes sont le tri et l'inversion :

```
x = [3,1,2]

x.sort() # sorting the list
print(x)

x.reverse() # reverse ordering
print(x)
```

```
[1, 2, 3]
[3, 2, 1]
```

> Note
>
> Lorsque vous triez ou inversez une liste, l'opération est effectuée et la liste est modifiée *en place*. Ainsi, le code `y = x.sort()` est probablement incorrect, car `x.sort()` ne retourne rien. Donc `y` n'est pas égal à la liste triée, mais vaut `None`.

Les méthodes sont des fonctions spéciales liées à des objets particuliers. Elles s'écrivent donc sous la forme `object.method(input)`. Elles sont toujours séparées de l'objet de base par un point — cela signifie donc que vous ne pouvez pas utiliser de point dans un nom de variable. Par exemple, si vous tapez `q.i = 1` dans le terminal, il renverra une erreur indiquant que `q is not defined`. Ces méthodes peuvent accepter des arguments supplémentaires (elles sont comme des fonctions), mais ces exemples utilisent simplement les valeurs par défaut. Par exemple, vous pouvez trier par ordre décroissant :

```
x = [3,1,2]

x.sort(reverse=True) # passing arg
print(x)
```

```
[3, 2, 1]
```

> Note
>
> Ceci est le premier exemple que je présente d'une fonction qui possède un argument *nommé*, donc au lieu de `function(input)` c'est `function(keyword=input)`. Les fonctions peuvent prendre plusieurs arguments, comme `function(input1,input2)`. Dans ce cas, l'ordre des arguments compte, et vous pouvez utiliser des arguments nommés pour différencier les entrées. J'aborderai ce point plus en détail dans un chapitre ultérieur consacré à la définition de vos propres fonctions.

Vous pouvez également ajouter des éléments à des listes ou en supprimer :

```
x = [3,1,2]

x.append('a') # appending an item to end of list
print(x)

x.remove(1) # removing an item
print(x)
```

```
[3, 1, 2, 'a']
[3, 2, 'a']
```

Pour trouver l'emplacement précis d'un élément dans une liste, vous pouvez utiliser la méthode index :

```
x = ['a','b','c']

# will be 1, the 2nd item in a list
bindex = x.index('b')
print(bindex)
```

```
1
```

Il existe d'autres méthodes pour les listes que je n'ai pas montrées ici ; si vous exécutez la commande `dir` sur un objet, elle affichera toutes les méthodes disponibles. Je vous encourage à expérimenter par vous-même et à voir comment les autres méthodes, comme `count` ou `pop`, fonctionnent.

```
x = ['a','b','c']

# you can look at the methods
# for a object using dir(object)
me = dir(x)

# there are many more! only
# printing a few to save space
print(me[-6:])
```

```
['index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Parfois, vous avez besoin d'un objet similaire à une liste, mais vous ne voulez pas qu'il soit mutable (vous ne pouvez donc pas effectuer des opérations comme modifier une seule valeur, ajouter un élément ou trier la liste). Cela peut se produire si vous avez un ensemble de constantes dans votre script et que vous savez qu'elles ne doivent jamais être modifiées. Les placer dans un *tuple* est un moyen de s'assurer qu'elles ne sont pas modifiées par inadvertance. Les tuples ressemblent globalement à des listes, mais utilisent des parenthèses au lieu de crochets :

```
y = (1,2,3)
y[1] = 5 # this will give an error
```

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

Vous pouvez convertir une liste en un tuple via la commande `tuple` :

```
y = [1,2,3]
z = tuple(y)
```

```
y[1] = 5 # this is ok
print(y) # can see y list is updated
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

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

Ou inversement, convertir un tuple en liste via la commande `list` :

```
z = (1,2,3)
y = list(z)
y[1] = 5 # this is ok
print(y)
z[1] = 5 # this is not, again cannot modify tuple
```

```
[1, 5, 3]
```

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

Une dernière remarque au sujet des tuples : vous pouvez affecter plusieurs objets en même temps. Vous pouvez donc faire :

```
x, y = 1, 2
print(x,y)
```

```
1 2
```

Ceci s'appelle *déballage de tuples*. La raison pour laquelle on l'appelle ainsi est que, lorsque vous *ne déballez pas* les valeurs séparées par une virgule, cela renvoie un tuple.

```
t = 1, 2
print(t)
```

```
(1, 2)
```

Et vous pouvez affecter plus de deux valeurs :

```
a,b,c,d = (1,'a',6,-1.2)
print(a,b,c,d)
```

```
1 a 6 -1.2
```

Cela peut être pratique dans divers exemples de boucles for (présentés au chapitre 4), ainsi que lorsque des fonctions renvoient plusieurs valeurs (présenté au chapitre 5). En dehors de cela, toutefois, les tuples ne sont pas aussi couramment utilisés que les listes, selon mon expérience. Mais un exemple d'utilisation est illustré dans la section suivante, où l'on a besoin d'utiliser des tuples immuables pour les dictionnaires.

2.5 Dictionnaires

Le deuxième grand conteneur d'éléments en Python est un dictionnaire. Ainsi, pour accéder aux éléments d'une liste, il suffit de suivre l'ordre fixé : le premier élément est `mylist[0]`, le deuxième élément est `mylist[1]`, etc. Parfois, vous souhaitez pouvoir accéder aux éléments par des noms plus simples ; par exemple, imaginez que vous ayez une liste destinée à contenir les informations d'une personne :

```
# using a list to hold data
x = ['Andy Wheeler','Data Scientist','2019']
```

Pour accéder à l'élément nom, vous devez savoir qu'il se trouve à l'indice 0, l'élément titre se trouve à l'indice 1, etc. Il est probablement plus simple de se référer à ces données via un dictionnaire.

```
# using a list to hold data
d = {'name': 'Andy Wheeler',
     'title':'Data Scientist',
     'start_year': 2019}

print(d)
```

```
{'name': 'Andy Wheeler', 'title': 'Data Scientist', 'start_year':
2019}
```

Maintenant, il est plus facile d'accéder à un élément individuel via `dict[key]`, donc si je veux uniquement le nom, il me suffit de le spécifier explicitement :

```
# Grabbing the specific name element
print(d['name'])
```

```
Andy Wheeler
```

La terminologie des dictionnaires est qu'ils possèdent des *clés* qui référencent des *valeurs*. Les valeurs peuvent être n'importe quoi : des valeurs numériques, des chaînes, des listes, d'autres dictionnaires, etc. Les clés, en revanche, doivent être *immuables*, même si les dictionnaires eux-mêmes sont mutables (vous ne pouvez donc pas utiliser une liste comme clé, mais vous pouvez utiliser un tuple). Ici, nous pouvons modifier les valeurs du dictionnaire `d` original que j'ai créé. Vous pouvez également ajouter un nouvel élément une fois le dictionnaire créé.

```
# Modifying the start_year element
d['start_year'] = 2020

# Adding in a new element tenure
d['tenure'] = 3

print(d['name'])
print(d['title'])
print(d['start_year'])
print(d['tenure'])
```

```
Andy Wheeler
Data Scientist
2020
3
```

La plupart du temps, on utilise des chaînes de caractères comme clés, puisque l'avantage principal des dictionnaires par rapport aux listes est de pouvoir nommer des objets spécifiques pour s'y référer. Mais une clé peut aussi être un nombre. Ainsi, si vous disposez d'identifiants numériques dans une autre base de données qui font référence à des emplacements spécifiques, il peut être judicieux de rédiger votre dictionnaire en utilisant ces mêmes identifiants numériques.

```
# You can have numeric values
# as a dictionary key
d = {} # can init a dict as empty
d[101] = {'address': 'Penny Lane', 'tot_crimes': 1}
d[202] = {'address': 'Outer Space', 'tot_crimes': 42}

# These show a dictionary inside of a dictionary
print(d[101])
print(d[202])
```

```
{'address': 'Penny Lane', 'tot_crimes': 1}
{'address': 'Outer Space', 'tot_crimes': 42}
```

Et, comme pour les listes vides, un dictionnaire vide renverra `False` dans une instruction if booléenne :

```
d = {} # can init a dict as empty

if d:
  print('The dictionary d is not empty')
else:
  print('The dictionary d is empty')
```

```
The dictionary d is empty
```

Il existe *de nombreux* types d'objets plus complexes en Python, le tout premier exemple dans la préface présentait un exemple utilisant des objets *datetime*. Mais sous le capot, ils ne sont souvent que des conteneurs permettant d'effectuer différentes opérations sur les objets que j'ai énumérés ci-dessus : valeurs numériques, chaînes de caractères, listes et dictionnaires.