O'REILLY®

Python pour Excel

Automatisation et analyse des données avec un langage moderne



Felix Zumstein

Python pour Excel

Felix Zumstein

FIRST INTERACTIVE

O'REILLY

Table des matières

Préface xi
Pourquoi j'ai écrit ce livre xii
À qui s'adresse ce livre xiii
Organisation de ce livre xiii
Partie I : Introduction à Python xiii
Partie II : Introduction à pandas xiv
Partie III : Lecture et écriture de fichiers Excel sans Excel xiv
Partie IV : Programmation de l'application Excel avec xlwings xiv
Versions de Python et d'Excel xv
Conventions utilisées dans ce livre xvi
Utilisation des exemples de code xvi
Remerciements xvii

1. Pourquoi Python pour Excel ? 1
Excel est un langage de programmation 2
Excel dans l'actualité 3
Bonnes pratiques de programmation 4
Excel moderne 9
Python pour Excel 11
Lisibilité et maintenabilité 11
Bibliothèque standard et gestionnaire de packages 13
Informatique scientifique 14
Caractéristiques des langages modernes 15
Compatibilité multiplateforme 16
Conclusion 17

2. Environnement de développement 19
Distribution Python Anaconda 20
Installation 20
Anaconda Prompt 21
Python REPL : une session Python interactive 24
Gestionnaires de packages : Conda et pip 25

Environnements Conda 27
Notebooks Jupyter 28
Exécution de notebooks Jupyter 29
Cellules de notebook 30
Mode édition et mode commande 32
L'ordre d'exécution des cellules est important 33
Arrêt des notebooks Jupyter 34
Visual Studio Code 35
Installation et configuration 37
Exécution d'un script Python 39
Conclusion 44

3. Prise en main de Python 45
Types de données 45
Objets 46
Types numériques 47
Booléens 50
Chaînes 51
Indexation et découpage 52
Indexation 53
Découpage 53
Structures de données 54
Listes 55
Dictionnaires 57
Tuples 58
Ensembles 59
Structures de contrôle 60
Blocs de code et instruction pass 60
L'instruction if et les expressions conditionnelles 61
Boucles for et while 62
Compréhensions de listes, de dictionnaires et d'ensembles 65
Organisation du code 66
Fonctions 66
Modules et instruction import 68
Classe datetime 70
PEP 8 : Guide de style du code Python 72
PEP 8 et VS Code 74
Indicateurs de type 75
Conclusion 75

4. Bases de NumPy 79
Prise en main de NumPy 79
Tableau NumPy 79
Vectorisation et diffusion 81
Fonctions universelles (ufunc) 83
Création et manipulation de tableaux 84
Lecture et définition d'éléments de tableau 84
Constructeurs de tableau utiles 85
Vue ou copie ? 86
Conclusion 86

5. Analyse de données avec pandas 87
DataFrames et séries 87
Index 90
Colonnes 92
Manipulation des données 94
Sélection des données 95
Sélection à l'aide d'un MultiIndex 100
Modification des données 101
Données manquantes 104
Données en double 106
Opérations arithmétiques 107
Utilisation des colonnes de texte 109
Application d'une fonction 110
Vue ou copie ? 112
Combinaison de DataFrames 112
Concaténation 113
Jointure et fusion 114
Statistiques descriptives et agrégation des données 117
Statistiques descriptives 117
Regroupement 118
Tableaux croisés dynamiques et fusion 119
Réalisation de tracés 121
Matplotlib 121
Plotly 123
Importation et exportation de DataFrames 126
Exportation de fichiers CSV 127
Importation de fichiers CSV 128
Conclusion 130

6. Analyse de séries chronologiques avec pandas.....131
DatetimeIndex 132
Création d'un DatetimeIndex 132
Filtrage d'un DatetimeIndex 134
Utilisation des fuseaux horaires 136
Manipulations courantes sur les séries chronologiques 137
Décalages et variations en pourcentage 137
Rebasement et corrélation 140
Rééchantillonnage 143
Fenêtres glissantes 145
Limites de pandas 146
Conclusion 147

7. Manipulation de fichiers Excel avec pandas 151
Étude de cas : génération d'un rapport Excel 151
Lecture et écriture de fichiers Excel avec pandas 155
Fonction read_excel et classe ExcelFile 155
Méthode to_excel et classe ExcelWriter 161
Limites lors de l'utilisation de pandas avec des fichiers Excel 163
Conclusion 163

8. Manipulation de fichiers Excel avec des packages de lecture et d'écriture.....165
Packages de lecture et d'écriture 165
Savoir quel package utiliser 166
Module excel.py 167
OpenPyXL 169
XlsxWriter 173
pyxlsb 175
xlrd, xlwt et xlutils 176
Fonctionnalités avancées pour la lecture et l'écriture de fichiers Excel 179
Exploitation de fichiers Excel volumineux 179
Mise en forme des DataFrames dans Excel 183
Étude de cas (revisitée) : création d'un rapport Excel 189
Conclusion 190

9. Automatisation d'Excel 193
Prise en main de xlwings 194
Utilisation d'Excel pour visualiser des données 194
Modèle objet d'Excel 196
Exécution du code VBA 202
Convertisseurs, options et collections 203
Exploitation des DataFrames 204
Convertisseurs et options 205

Graphiques, images et noms définis 207
Étude de cas (revisitée) : rapports Excel 211
Approfondissement de xlwings 212
Sur quoi repose xlwings ? 213
Amélioration des performances 215
Comment pallier les fonctionnalités manquantes 216
Conclusion 217

10. Outils Excel dopés par Python 219
Utilisation d'Excel comme frontend avec xlwings 219
Complément Excel 220
Commande quickstart 222
Run main 222
Fonction RunPython 223
Déploiement 228
Dépendance Python 228
Classeurs autonomes : comment se débarrasser du complément xlwings 229
Hiérarchie de configuration 230
Paramètres 231
Conclusion 233

11. Python Package Tracker 235
Ce que nous allons construire 235
Fonctionnalités de base 238
API Web 238
Bases 242
Exceptions 251
Structure de l'application 253
Frontend 254
Backend 258
Débogage 262
Conclusion 263

12. Fonctions définies par l'utilisateur (UDF) 265
Prise en main des UDF 266
Commande quickstart pour les UDF 266
Étude de cas : Google Trends 271
Introduction à Google Trends 271
Utilisation de DataFrames et de tableaux dynamiques 273
Récupération de données à partir de Google Trends 278
Traçage avec des UDF 283
Débogage des UDF 284

Sujets avancés sur les UDF 286
Optimisation des performances de base 287
Mise en cache 289
Décorateur sub 291
Conclusion 292

Annexe A Environnements Conda 295
Création d'un nouvel environnement Conda 295
Annulation de l'activation automatique 297

Annexe B Fonctionnalités avancées de VS Code 299
Débogueur 299
Notebooks Jupyter dans VS Code 301
Exécution de notebooks Jupyter 301
Scripts Python avec des cellules de code 303

Annexe c Concepts Python avancés 305
Classes et objets 305
Exploitation d'objets datetime prenant en charge les fuseaux horaires 307
Objets Python mutables ou immuables ? 308
Appel de fonctions avec des objets mutables comme arguments 309
Fonctions avec des objets mutables comme arguments par défaut 311

Index 313

Préface

Microsoft anime un forum consacré à Excel sur UserVoice[1] où tout le monde peut soumettre au vote de nouvelles idées afin d'améliorer cette application. La demande de fonctionnalité la plus populaire est que Python devienne le langage de programmation d'Excel, et cette suggestion recueille environ deux fois plus de suffrages que la deuxième demande la plus plébiscitée. Bien qu'il ne se soit rien passé depuis que l'idée a été suggérée en 2015, les utilisateurs d'Excel ont repris espoir à la fin de l'année 2020 lorsque Guido van Rossum, le créateur de Python, a tweeté[2] qu'il s'ennuyait en retraite et qu'il allait rejoindre Microsoft. Je ne suis pas certain que son embauche ait une quelconque influence sur l'intégration d'Excel et de Python, mais je sais parfaitement comment vous pouvez profiter de la combinaison de ces deux produits, et c'est d'ailleurs le sujet de ce livre.

C'est le fait que nous vivons dans un monde composé de données qui a motivé l'écriture de *Python pour Excel*. De nos jours, d'énormes bases de données sont disponibles en accès libre sur à peu près tous les sujets. Souvent, ces données sont si volumineuses qu'elles ne tiennent plus dans une feuille de calcul. Il y a quelques années, on a pu évoquer le terme de *big data*, mais de nos jours, un ensemble de données de plusieurs millions de lignes n'a vraiment rien d'exceptionnel. Excel a évolué pour faire face à cette tendance et Power Query est arrivé à la rescousse pour charger et nettoyer les jeux de données qui ne tiennent pas dans une feuille de calcul, accompagné de Power Pivot, un complément qui effectue une analyse sur ces jeux de données et présente les résultats. Power Query est basé sur le langage de formules nommé Power Query M, tandis que Power Pivot définit les formules à l'aide d'expressions DAX (*Data Analysis Expressions*). Si vous souhaitez également automatiser quelques éléments dans votre fichier Excel, vous devez utiliser le langage de programmation d'Excel, Visual Basic pour Applications (VBA). Autrement dit, si vous voulez réaliser quelque chose d'assez simple, vous pouvez au bout du compte utiliser VBA, M et DAX. La difficulté supplémentaire est que tous ces langages ne servent que dans le monde Microsoft, notamment Excel et Power BI (que je présenterai brièvement au Chapitre 1).

---

1 https://excel.uservoice.com/forums/304921-excel-for-windows-desktop-application/filters/top

2 https://twitter.com/gvanrossum/status/1326932991566700549

En revanche, Python est un langage de programmation polyvalent qui est devenu l'un des choix les plus populaires parmi les spécialistes de l'analyse des données. Si vous utilisez Python avec Excel, vous allez employer un langage de programmation qui est performant sur tous les plans, qu'il s'agisse d'automatiser Excel, d'accéder à des jeux de données et de les préparer, ou d'effectuer des tâches d'analyse et de visualisation. Cerise sur le gâteau : vous pouvez réutiliser vos compétences Python en dehors d'Excel ; par exemple, si vous avez besoin d'augmenter votre puissance de calcul, vous pouvez facilement faire migrer votre modèle quantitatif, votre simulation ou votre application d'apprentissage automatique dans le cloud, où des ressources informatiques pratiquement sans limites vous attendent.

Pourquoi j'ai écrit ce livre

Grâce à mon travail sur xlwings, le package d'automatisation d'Excel que je présenterai dans la Partie IV de ce livre, je suis en contact étroit avec de nombreux utilisateurs qui se servent de Python avec Excel, que ce soit *via* le suivi des problèmes sur GitHub[3], une question sur StackOverflow[4] ou lors d'une conférence.

On me demande régulièrement si je peux recommander des ressources pour commencer l'apprentissage de Python. Bien qu'il n'y ait certainement aucune pénurie d'ouvrages d'introduction à Python, ces derniers sont souvent trop généraux (on n'y trouve rien sur l'analyse des données) ou trop spécifiques (et ils nécessitent une trop grande culture scientifique). Les utilisateurs d'Excel se situent cependant entre ces deux approches : ils travaillent assurément avec des données, mais une introduction scientifique complète peut se révéler trop technique. Ils ont également souvent des exigences et des questions spécifiques auxquelles aucun des ouvrages existants ne répond. Voici quelques exemples de ces questions :

- De quel package Python-Excel ai-je besoin pour effectuer telle tâche ?
- Comment migrer ma connexion à la base de données Power Query vers Python ?
- Quel est l'équivalent du filtre automatique ou du tableau croisé dynamique d'Excel en Python ?

J'ai écrit ce livre pour ceux et celles qui n'ont aucune connaissance en Python et qui souhaitent automatiser leurs tâches réalisées sur Excel et tirer parti de l'analyse des données de Python ainsi que des outils scientifiques d'Excel.

3 https://github.com/xlwings/xlwings/issues
4 https://stackoverflow.com/

À qui s'adresse ce livre

Si vous êtes un utilisateur avancé d'Excel et que vous voulez repousser les limites de cette application grâce à un langage de programmation moderne, ce livre est fait pour vous. Plus généralement, cela signifie que vous passez des heures chaque mois à télécharger, nettoyer et copier-coller de grandes quantités de données dans des feuilles de calcul qui sont essentielles pour votre activité. Bien qu'il existe différentes façons de dépasser les limites d'Excel, ce livre se concentre sur l'emploi de Python pour cette tâche.

Vous devez avoir des connaissances de base en programmation : il est notamment utile d'avoir déjà écrit une fonction ou une boucle for (quel que soit le langage de programmation) et d'avoir une idée de ce qu'est un entier ou une chaîne de caractères. Vous tirerez également profit de ce livre si vous avez l'habitude d'écrire des formules complexes ou d'optimiser des macros VBA enregistrées. Vous n'êtes cependant pas censé avoir d'expérience en Python, car cet ouvrage propose des introductions à tous les outils dont nous nous servirons, y compris une introduction à Python lui-même.

Si vous êtes un développeur VBA chevronné, vous trouverez de nombreuses comparaisons entre Python et VBA qui vous permettront de contourner les pièges courants et d'être opérationnel rapidement.

Ce livre peut également être utile si vous êtes un développeur Python et que vous avez besoin d'en savoir plus sur les différentes façons dont Python peut gérer l'application Excel et ses fichiers afin de pouvoir choisir le bon package en fonction des exigences de vos utilisateurs professionnels.

Organisation de ce livre

Dans ce livre, divisé en quatre parties, je vais vous faire découvrir tout ce qui concerne *Python pour Excel* :

Partie I : Introduction à Python

Cette partie explique les raisons pour lesquelles Python est le compagnon idéal d'Excel ; ensuite je présente les outils que nous utiliserons dans ce livre : la distribution Python Anaconda, Visual Studio Code et les notebooks Jupyter. Cette partie vous enseignera également suffisamment de Python pour pouvoir maîtriser le reste de ce livre.

Partie II : Introduction à pandas

pandas est la bibliothèque de référence de Python pour l'analyse des données. Nous allons apprendre à remplacer les classeurs Excel par une combinaison de notebooks Jupyter et de pandas. Habituellement, le code pandas est à la fois plus facile à gérer et plus efficace qu'un classeur Excel, et vous pouvez travailler avec des jeux de données qui ne tiennent pas dans une feuille de calcul. Contrairement à Excel, pandas permet d'exécuter votre code où vous le souhaitez, y compris dans le cloud.

Partie III : Lecture et écriture de fichiers Excel sans Excel

Cette partie traite de la manipulation des fichiers Excel à l'aide des packages Python suivants : pandas, OpenPyXL, XlsxWriter, pyxlsb, xlrd et xlwt. Ces packages permettent de lire et d'écrire des classeurs Excel directement sur le disque et, en tant que tels, ils remplacent l'application Excel : comme vous n'avez pas besoin qu'Excel soit installé, ils fonctionnent sur n'importe quelle plateforme prise en charge par Python, notamment Windows, macOS et Linux. Parmi les scénarios d'utilisation typique de ces packages, on peut citer la lecture des données des fichiers Excel que vous recevez tous les matins en provenance d'une société ou d'un système externe et le stockage de leur contenu dans une base de données. Ces packages servent aussi à fournir les fonctionnalités d'export vers Excel que l'on trouve dans presque toutes les applications.

Partie IV : Programmation de l'application Excel avec xlwings

Dans cette partie, nous verrons comment nous pouvons utiliser Python avec le package xlwings afin d'automatiser l'application Excel plutôt que de lire et d'écrire des fichiers Excel sur le disque. Cette partie nécessite par conséquent que vous disposiez d'une installation locale d'Excel. Nous allons apprendre à ouvrir des classeurs Excel et à les manipuler. En plus de lire et d'écrire des fichiers *via* Excel, nous allons créer des outils Excel interactifs qui nous permettront de cliquer sur un bouton pour que Python effectue quelque chose que vous avez peut-être réalisé précédemment avec des macros VBA, comme un calcul coûteux en ressources système. Nous allons également apprendre à écrire des fonctions définies par l'utilisateur[5] en Python au lieu d'employer VBA.

5 Microsoft a commencé à utiliser le terme *fonction personnalisée* au lieu d'UDF (*user defined function*, fonction définie par l'utilisateur), mais dans ce livre, je continuerai à les appeler UDF.

Il est important de comprendre la différence fondamentale entre la lecture et l'écriture de *fichiers* Excel (Partie III) et la programmation de l'*application* Excel (Partie IV). La figure ci-dessous illustre cette différence.

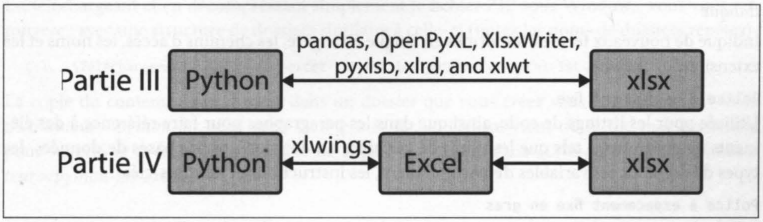


Figure P.1 : Différences entre la lecture et l'écriture des fichiers Excel (Partie III) et la programmation Excel (Partie IV).

Comme la Partie III ne nécessite pas d'installation d'Excel, le code fonctionne sur toutes les plateformes prises en charge par Python (Windows, macOS et Linux). En revanche, le code de la Partie IV ne fonctionnera que sur les plateformes prises en charge par Microsoft Excel, c'est-à-dire Windows et macOS, car le code repose sur une installation locale de Microsoft Excel.

Versions de Python et d'Excel

Ce livre est basé sur la version 3.8 de Python, qui est la version fournie avec la dernière version de la distribution Python Anaconda au moment de la rédaction de cet ouvrage. Si vous souhaitez utiliser une version plus récente de Python, suivez les instructions sur la page d'accueil du site Web de ce livre[6], mais assurez-vous que vous n'utilisez pas une version plus ancienne. Je ferai occasionnellement un commentaire s'il y a une différence avec la version 3.9 de Python.

Ce livre part également du principe que vous utilisez une version moderne d'Excel, c'est-à-dire au moins Excel 2007 pour Windows et Excel 2016 pour macOS. La version d'Excel installée en local et fournie avec l'abonnement Microsoft 365 fonctionnera également parfaitement, et c'est d'ailleurs celle que je recommande, car elle possède les dernières fonctionnalités que vous ne trouverez pas dans d'autres versions d'Excel. C'était aussi la version que j'ai utilisée pour écrire ce livre, de telle sorte que si vous utilisez une autre version d'Excel, vous pourrez parfois constater une petite différence dans le nom ou l'emplacement d'un élément de menu.

6 https://www.xlwings.org/book

Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique de nouveaux termes, les termes en langue étrangère, les chemins d'accès, les noms et les extensions de fichiers.

`Police à espacement fixe`

Utilisée pour les listings de code, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que les noms de variables ou de fonctions, les bases de données, les types de données, les variables d'environnement, les instructions et les mots-clés.

`Police à espacement fixe en gras`

Affiche les commandes ou tout autre texte qui doit être saisi tel quel par l'utilisateur.

`Police à espacement fixe en italique`

Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.



Cet élément indique un conseil ou une suggestion.



Cet élément indique une note.



Cet élément indique un avertissement ou une mise en garde.

Utilisation des exemples de code

J'ai créé un site Web[7] qui contient des informations supplémentaires destinées à faciliter la lecture de cet ouvrage. Assurez-vous de le consulter, surtout si vous rencontrez un problème.

Des documents supplémentaires (exemples de code, exercices, *etc.*) peuvent être téléchargés à l'adresse *https://github.com/fzumstein/python-for-excel*. Pour télécharger le code, cliquez sur le bouton vert Code pour dérouler le menu, puis sélectionnez Download ZIP. Une fois l'archive téléchargée, cliquez avec le bouton droit sur le fichier sous Windows et sélectionnez Extraire tout pour décompresser dans un dossier les fichiers. Sous macOS, faites simplement un double-clic sur le fichier pour le décompresser. Si vous savez comment travailler avec Git, vous

7 https://xlwings.org/book

pouvez également utiliser Git pour cloner le dépôt sur votre disque dur. Vous pouvez copier le dossier où vous voulez, mais j'y ferai référence de temps en temps de la manière suivante :

```
C:\Utilisateurs\nom d'utilisateur\python-for-excel
```

En téléchargeant et en décompressant simplement le fichier ZIP sous Windows, vous vous retrouverez avec une structure de dossiers similaire à celle-ci (notez les noms de dossiers répétés) :

```
C:\...\Téléchargements\python-for-excel-1st-edition\python-for-excel-1st-edition
```

La copie du contenu de ce dossier dans un dossier que vous créez sous *C:\Utilisateurs\<nom d'utilisateur>\python-for-excel* vous facilitera la tâche. Ce qui vient d'être dit vaut également pour macOS et il est donc préférable de copier les fichiers dans */Utilisateurs/<nom d'utilisateur>/python-for-excel*.

Remerciements

Comme il s'agit du premier livre que j'écris, de très nombreuses personnes m'ont apporté leur aide et je leur suis incroyablement reconnaissant, car elles m'ont grandement facilité la tâche.

Chez O'Reilly, j'aimerais remercier mon éditrice, Melissa Potter, qui a fait un excellent travail pour conserver intacte ma motivation et tenir les délais ; elle m'a aussi aidé à améliorer la lisibilité de cet ouvrage. Je souhaiterais également remercier Michelle Smith, qui a travaillé avec moi sur le projet initial de ce livre, et Daniel Elfanbaum, qui ne s'est jamais lassé de répondre à mes questions techniques.

J'adresse un grand merci à tous mes collègues, amis et clients qui ont pris le temps de lire les premières ébauches de mon manuscrit. Leurs commentaires ont été cruciaux pour rendre le livre plus facile à comprendre, et certaines études de cas sont inspirées par des problèmes Excel réels qu'ils ont partagés avec moi. Mes remerciements vont à Adam Rodriguez, Mano Beeslar, Simon Schiegg, Rui Da Costa, Jürg Nager et Christophe de Montrichard.

J'ai également reçu des commentaires utiles des lecteurs de la version Early Release qui a été publiée sur la plateforme d'apprentissage en ligne d'O'Reilly. Je remercie Felipe Maion, Ray Doue, Kolyu Minevski, Scott Drummond, Volker Roth et David Ruggles.

J'ai eu beaucoup de chance que le livre ait été révisé par des experts hautement qualifiés et j'apprécie vraiment le travail acharné qu'ils ont accompli dans des délais très serrés. Merci pour toute votre aide, Jordan Goldmeier, George Mount, Andreas Clenow, Werner Brönnimann et Eric Moreira.

J'adresse un merci spécial à Björn Stiel, qui n'a pas seulement été un critique technique, mais auprès de qui j'ai appris beaucoup de choses qui se retrouvent dans ce livre. J'ai aimé travailler avec toi au cours de ces dernières années.

Enfin, je voudrais exprimer ma gratitude à Eric Reynolds, qui a fusionné son projet ExcelPython dans la base de code xlwings en 2016. Il a également repensé l'ensemble du package en repartant de zéro, et transformé mon horrible API des premiers jours qui n'est plus qu'un mauvais souvenir. Merci beaucoup !

PARTIE I

Introduction à Python

CHAPITRE 1

Pourquoi Python pour Excel ?

Habituellement, les utilisateurs d'Excel commencent à remettre en question leurs outils lorsqu'ils en atteignent les limites. Un exemple classique est celui des classeurs Excel qui contiennent tellement de données et de formules qu'ils deviennent lents ou, dans le pire des cas, plantent. Il est logique, cependant, de remettre en question votre configuration avant que les choses ne se dégradent : si vous travaillez sur des classeurs critiques où les erreurs peuvent entraîner des dommages financiers et ruiner votre réputation, ou si vous passez des heures quotidiennement à mettre à jour manuellement des classeurs Excel, vous devriez apprendre à automatiser vos processus avec un langage de programmation. L'automatisation éliminera le risque d'erreur humaine et vous permettra de passer votre temps sur des tâches plus productives que celles qui consistent à copier et à coller des données dans une feuille de calcul Excel.

Dans ce chapitre, je vais vous donner quelques arguments afin de démontrer que Python combiné à Excel est un excellent choix et vous montrer ses avantages par rapport au langage d'automatisation intégré d'Excel, VBA. Après avoir présenté Excel en tant que langage de programmation et expliqué ses particularités, je soulignerai les fonctionnalités spécifiques de Python qui en font un bien meilleur outil que VBA. Mais pour commencer, examinons rapidement les origines de nos deux personnages principaux !

En termes de technologie informatique, Excel et Python sont tous deux présents sur scène depuis très longtemps : Excel a été lancé par Microsoft pour la première fois en 1985 (cela pourra en surprendre beaucoup, mais à l'époque il n'était disponible que sur Macintosh). Ce n'est qu'en 1987 que Microsoft Windows a obtenu sa première version sous le nom d'Excel 2.0. Microsoft n'était cependant pas le premier acteur sur le marché des tableurs, car VisiCorp avait sorti VisiCalc en 1979, suivi par Lotus Software en 1983 avec Lotus 1-2-3. Ce n'était d'ailleurs pas le premier tableur pour Microsoft, car trois ans plus tôt, avait été publié Multiplan, un tableur qui tournait sous MS-DOS et quelques autres systèmes d'exploitation, mais pas sous Windows.

Python est né en 1991, seulement six ans après Excel. Bien qu'Excel soit devenu populaire dès le début, il a fallu un peu plus de temps à Python avant qu'il ne soit adopté dans certains domaines

tels que le développement Web ou l'administration système. En 2005, Python a commencé à devenir une alternative sérieuse pour le calcul scientifique lorsque *NumPy*, un package pour l'informatique basée sur les tableaux et l'algèbre linéaire, a été publié pour la première fois. NumPy a combiné deux anciens packages et a donc rationalisé tous les efforts de développement autour de l'informatique scientifique en un seul projet. Aujourd'hui, il constitue la base d'innombrables packages scientifiques, y compris *pandas*, qui est sorti en 2008 et qui est en grande partie responsable de l'adoption généralisée de Python dans le monde de la science des données et de la finance, qui a commencé après 2010. Grâce à pandas, Python, associé à R, est devenu l'un des langages les plus couramment utilisés pour les tâches de science des données telles que l'analyse de données, les statistiques et l'apprentissage automatique.

Le fait que Python et Excel aient été tous les deux inventés il y a longtemps n'est pas la seule chose qu'ils ont en commun : Excel et Python sont également des langages de programmation. Si cela ne vous surprend probablement pas à propos de Python, cela peut nécessiter des explications pour Excel.

Excel est un langage de programmation

Cette section commence par présenter Excel en tant que langage de programmation, ce qui vous aidera à comprendre pourquoi les problèmes de feuille de calcul se retrouvent régulièrement dans l'actualité. Nous examinerons ensuite quelques bonnes pratiques qui ont émergé dans la communauté du développement de logiciels et qui peuvent vous éviter de nombreuses erreurs Excel typiques. Nous conclurons par une brève introduction à Power Query et Power Pivot, deux outils Excel modernes qui couvrent le type de fonctionnalité pour laquelle nous utiliserons plutôt pandas.

Si vous utilisez Excel pour autre chose que votre liste de courses, vous utilisez certainement des fonctions telles que `=SOMME(A1:A4)` pour faire la somme d'une plage de cellules. Si vous réfléchissez un instant à la façon dont cela fonctionne, vous remarquerez que la valeur d'une cellule dépend généralement d'une ou plusieurs autres cellules, qui peuvent à leur tour utiliser des fonctions qui dépendent d'une ou plusieurs autres cellules, *etc.* Effectuer de tels appels de fonction imbriqués n'est pas différent de la façon dont les autres langages de programmation fonctionnent, à ceci près que vous écrivez le code dans des cellules au lieu de fichiers texte. Et si cela ne vous a pas encore convaincu : à la fin de 2020, Microsoft a annoncé l'introduction de *fonctions lambda*, qui permettent d'écrire des fonctions réutilisables dans le propre langage de formule d'Excel, c'est-à-dire sans avoir à compter sur un autre langage comme VBA. Selon Brian Jones, chef de produit Excel, c'était la pièce manquante qui fait finalement d'Excel un « vrai » langage de programmation[1]. Cela signifie également que les utilisateurs d'Excel peuvent vraiment revendiquer le titre de programmeurs Excel !

1 Vous pouvez lire l'annonce concernant les fonctions lambda sur le blog Excel à : https://techcommunity.microsoft.com/t5/excel-blog/announcing-lambda-turn-excel-formulas-into-custom-functions/ba-p/1925546

Les programmeurs Excel ont cependant une particularité : la plupart d'entre eux sont des utilisateurs professionnels ou des experts de leur domaine, mais ils n'ont aucune formation initiale en informatique. Ce sont des traders, des comptables ou des ingénieurs, pour ne citer que quelques exemples. Leurs feuilles de calcul sont conçues pour résoudre un problème d'entreprise et ils ignorent souvent les meilleures pratiques en matière de développement de logiciels. En conséquence, ces feuilles de calcul mélangent souvent les entrées, les calculs et les sorties sur les mêmes feuilles ; elles peuvent nécessiter des étapes qui ne vont pas de soi pour qu'elles fonctionnent correctement, et les changements critiques sont effectués sans aucun filet de sécurité. En d'autres termes, les feuilles de calcul ne disposent pas d'une architecture d'application solide et souvent elles ne sont ni documentées ni testées. Parfois, ces problèmes peuvent avoir des conséquences désastreuses : si vous oubliez de recalculer votre classeur de trading avant de placer une transaction, vous pouvez acheter ou vendre le mauvais nombre d'actions, ce qui peut vous faire perdre de l'argent. Et si ce n'est pas seulement votre propre argent que vous placez, vous pourrez faire la une des journaux…

Excel dans l'actualité

Excel s'invite régulièrement dans l'actualité, et au cours de la rédaction de ce livre, deux nouvelles histoires ont fait la une des journaux. La première concernait le comité de nomenclature des gènes HUGO, qui a dû renommer quelques gènes humains afin qu'ils ne soient plus interprétés par Excel comme des dates. Par exemple, pour éviter que le gène MARCH1 ne soit transformé en 1er mars, il a été renommé en MARCHF1[2]. Dans la deuxième histoire, Excel a été impliqué dans la déclaration tardive de 16 000 résultats de tests COVID-19 en Angleterre. Le problème a été provoqué par les résultats des tests écrits dans l'ancien format de fichier Excel (*.xls*) qui était limité à environ 65 000 lignes. Cela signifiait que les ensembles de données plus volumineux étaient tout simplement tronqués au-delà de cette limite[3]. Bien que ces deux histoires montrent l'importance et la domination continues d'Excel dans le monde d'aujourd'hui, il n'y a probablement pas d'autre « incident Excel » plus célèbre que celui de « la baleine de Londres ».

London Whale est le surnom d'un trader dont les erreurs de trading ont contraint JP Morgan à annoncer une perte stupéfiante de 6 milliards de dollars en 2012. La source de la catastrophe était un modèle de valeur à risque basé sur Excel qui sous-estimait considérablement le risque réel de perdre de l'argent dans l'un de leurs portefeuilles. Le *Rapport du groupe de travail de la direction de JP Morgan Chase & Co. sur les pertes de 2012*[4] (publié en 2013) mentionne que « le modèle fonctionnait au moyen d'une série de feuilles de calcul Excel, qui devaient être remplies

2 Voir l'article de James Vincent, « Les scientifiques renomment les gènes humains pour empêcher Microsoft Excel de les interpréter à tort comme des dates », The Verge, 6 août 2020. https://www.theverge.com/2020/8/6/21355674/human-genes-rename-microsoft-excel-misreading-dates

3 Voir l'article de Leo Kelion, « Excel : pourquoi l'utilisation de l'outil de Microsoft a provoqué la perte des résultats de COVID-19 », BBC News, 5 octobre 2020. https://www.bbc.com/news/technology-54423988

4 Wikipédia renvoie à ce document dans l'une des notes de bas de page d'un de ses articles sur l'affaire : https://en.wikipedia.org/wiki/2012_JPMorgan_Chase_trading_loss#cite_note-35

manuellement, par un processus de copier-coller de données d'une feuille de calcul à une autre ». En plus de ces problèmes opérationnels, il y avait une erreur logique : dans un calcul, on divisait par une somme au lieu d'une moyenne.

Si vous voulez voir plus d'histoires de ce genre, jetez un œil à Horror Stories[5] : il s'agit d'une page Web gérée par EuSpRIG (*European Spreadsheet Risks InterestGroup*) qui recense les risques liés à l'usage des feuilles de calcul.

Pour éviter que votre entreprise ne se retrouve à la une des journaux avec une histoire similaire, nous allons étudier quelques bonnes pratiques qui rendent votre travail avec Excel beaucoup plus sûr.

Bonnes pratiques de programmation

Cette section présente les bonnes pratiques de programmation les plus importantes, notamment la séparation des problèmes, le principe DRY, les tests et le contrôle de version. Comme nous le verrons, suivre ces bonnes pratiques sera plus facile lorsque vous commencerez à utiliser Python avec Excel.

Séparation des problèmes

L'un des principes de conception les plus importants dans la programmation est la *séparation des problèmes*, parfois aussi appelée *modularité*. Cela signifie qu'un ensemble de fonctionnalités connexes doit être pris en charge par une partie indépendante du programme afin qu'il puisse être facilement remplacé sans affecter le reste de l'application. Au niveau le plus élevé, une application est souvent divisée en différentes couches[6] :

- couche de présentation ;
- couche métier ;
- couche de données.

Pour expliquer ces couches, examinez le convertisseur de devises simple qui est illustré à la Figure 1.1. Vous trouverez le fichier Excel *currency_converter.xlsx* dans le dossier *xl* des exemples de code.

Voici comment fonctionne l'application : saisissez respectivement le montant et la devise dans les cellules A4 et B4, et Excel convertira cela en dollars américains dans la cellule D4. De nombreux tableurs suivent une telle conception et sont utilisés par les entreprises tous les jours. Permettez-moi de décomposer cette application en couches :

5 http ://www.eusprig.org/horror-stories.htm

6 Cette terminologie est extraite du guide *Microsoft Application Architecture*, 2e édition, qui est disponible en ligne : https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650706(v=pandp.10)

- *Couche de présentation.* C'est celle que vous voyez et avec laquelle vous interagissez, c'est-à-dire l'interface utilisateur : les valeurs des cellules A4, B4 et D4 ainsi que leurs libellés constituent la couche de présentation du convertisseur de devises.
- *Couche métier.* Cette couche prend en charge la logique spécifique à l'application : la cellule D4 définit la façon dont le montant est converti en dollars. La formule `=A4 * RECHERCHEV(B4;F4:G11;2;FAUX)` se traduit par « montant multiplié par taux de change ».
- *Couche de données.* Comme son nom l'indique, cette couche se charge d'accéder aux données : la partie `RECHERCHEV` de la cellule D4 réalise cette tâche.

La couche de données accède aux données à partir de la table des taux de change qui commence à la cellule F3 et qui sert de base de données à cette petite application. Si vous avez été attentif, vous avez probablement remarqué que la cellule D4 apparaît dans les trois couches : cette application simple mélange les trois couches (présentation, métier et données) dans une seule cellule.

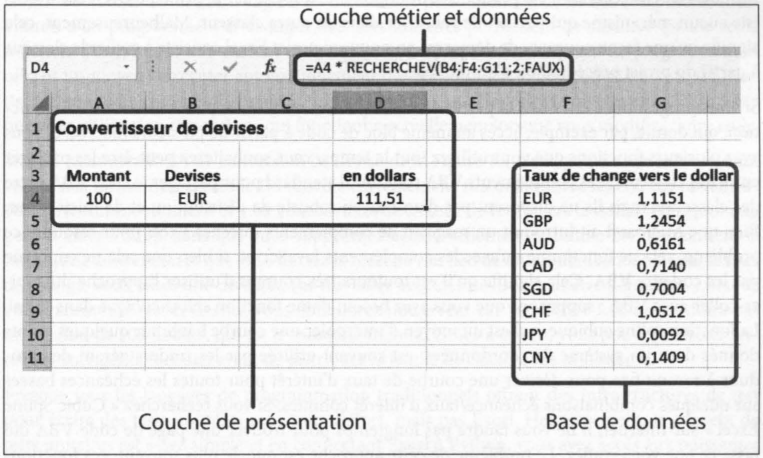


Figure 1.1 : Convertisseur de devises currency_converter.xlsx.

Ce n'est pas nécessairement un problème pour ce convertisseur de devises simple, mais souvent, ce qui commence comme un petit fichier Excel se transforme assez vite en une application beaucoup plus importante. Comment améliorer cette situation ? La plupart des ressources professionnelles consacrées au développement Excel conseillent d'utiliser une feuille séparée pour chaque couche, ce qui se traduit généralement dans la terminologie d'Excel en *entrées, calculs* et *sorties.* Souvent, on combine cela avec la définition d'un certain code couleur pour chaque

couche, par exemple, un arrière-plan bleu pour toutes les cellules d'entrée. Dans le Chapitre 11, nous allons créer une application réelle basée sur ces couches : Excel sera la couche de présentation, tandis que les couches métier et de données seront migrées vers Python, où il est beaucoup plus facile de structurer votre code correctement.

Maintenant que vous savez ce que signifie la séparation des problèmes, découvrons ce qu'est le principe DRY.

Principe DRY

Dans leur ouvrage, *The pragmatic programmer* (Pearson Education), Thomas David et Hunt Andrew ont popularisé le principe DRY (*Don't Repeat Yourself*, ne vous répétez pas). Le fait qu'aucun code ne soit dupliqué signifie qu'il y a moins de lignes de code et par conséquent moins d'erreurs, ce qui rend le code plus facile à gérer. Si votre logique métier se trouve dans les formules de vos cellules, il est pratiquement impossible d'appliquer le principe DRY, car il n'existe aucun mécanisme qui permet de réutiliser dans un autre classeur. Malheureusement, cela signifie qu'une façon courante de démarrer un nouveau projet Excel consiste à copier le classeur à partir du projet précédent ou d'un modèle.

Si vous programmez en VBA, la partie de code que l'on réutilise le plus souvent est une fonction, qui donne, par exemple, accès au même bloc de code à partir de plusieurs macros. Si vous avez plusieurs fonctions que vous utilisez tout le temps, vous souhaiterez peut-être les partager entre des classeurs. Les compléments VBA sont l'outil standard pour partager le code VBA entre des classeurs, mais ils ne disposent pas d'un moyen robuste de distribution et de mise à jour. Bien que Microsoft ait introduit un magasin de compléments internes Excel pour résoudre ce problème, cela ne fonctionne qu'avec les compléments JavaScript, si bien que cela ne concerne pas les codeurs VBA. Cela signifie qu'il est toujours très courant d'utiliser l'approche du copier-coller avec VBA : supposons que vous ayez besoin d'une fonction *spline cubique* dans Excel. La fonction spline cubique, qui est un moyen d'interpoler une courbe basée sur quelques points donnés dans un système de coordonnées, est souvent utilisée par les traders gérant des produits à revenu fixe pour dériver une courbe de taux d'intérêt pour toutes les échéances basées sur quelques combinaisons échéance/taux d'intérêt connues. Si vous recherchez « Cubic Spline Excel » sur Internet, il ne vous faudra pas longtemps pour trouver une page de code VBA qui fasse ce que vous voulez. Le problème de cette approche est que, le plus souvent, ces fonctions ont été écrites par une seule personne qui avait probablement de bonnes intentions, mais qui n'a rédigé aucune documentation et n'a réalisé aucun test. Ces fonctions marcheront peut-être pour la majorité des entrées, mais qu'en est-il de certains cas limites rares ? Si vous négociez un portefeuille à revenu fixe de plusieurs millions, il est souhaitable de disposer d'un outil dans lequel vous pouvez avoir confiance. C'est du moins ce que vous diront vos auditeurs internes lorsqu'ils découvriront la provenance du code.

Python facilite la distribution de code à l'aide d'un gestionnaire de packages, comme nous le verrons dans la dernière section de ce chapitre. Avant d'aborder ce sujet, poursuivons avec les tests, l'une des pierres angulaires du développement de logiciels robustes.

Test

Lorsque vous indiquez à un développeur Excel de tester ses classeurs, il effectuera probablement quelques vérifications aléatoires : il va cliquer sur un bouton et voir si la macro fait toujours ce qu'elle est censée faire ou modifier quelques entrées et vérifier si la sortie semble normale. Il s'agit cependant d'une stratégie risquée : Excel facilite l'introduction d'erreurs difficiles à repérer. Par exemple, vous pouvez remplacer une formule par une valeur codée en dur, ou oublier de modifier une formule dans une colonne masquée.

Lorsque vous dites à un développeur professionnel de tester son code, il écrit des *tests unitaires*. Comme son nom l'indique, il s'agit d'un mécanisme permettant de tester des composants individuels d'un programme. Par exemple, les tests unitaires permettent de s'assurer qu'une seule fonction d'un programme marche correctement. La plupart des langages de programmation offrent un moyen d'exécuter automatiquement des tests unitaires. L'exécution de tests automatisés augmentera considérablement la fiabilité de votre base de code et vous donnera l'assurance que vous n'allez rien casser de ce qui fonctionne actuellement lorsque vous modifiez votre code.

Si vous examinez l'outil de conversion de devise de la Figure 1.1, vous pouvez écrire un test qui vérifie si la formule de la cellule D4 renvoie correctement 105 dollars avec les entrées suivantes : 100 EUR comme montant et 1.05 comme taux de change de l'euro vers le dollar. Pourquoi cela peut-il aider ? Supposons que vous supprimiez accidentellement la cellule D4 qui contient la formule de conversion et que vous deviez la réécrire : au lieu de multiplier le montant par le taux de change, vous faites une division, car travailler avec des devises peut être déroutant. Lorsque vous exécutez le test ci-dessus, vous obtiendrez une erreur car 100 EUR / 1.05 ne produira pas un résultat de 105 dollars, qui est le montant attendu. Grâce à ce test, vous pouvez détecter l'erreur et corriger la formule avant de distribuer la feuille de calcul à vos utilisateurs.

Presque tous les langages de programmation traditionnels offrent des infrastructures de test pour écrire des tests unitaires sans trop d'effort, mais pas Excel. Heureusement, le concept de tests unitaires est assez simple et en connectant Excel à Python, vous avez accès aux puissantes infrastructures de tests unitaires de Python. Bien qu'une présentation plus approfondie des tests unitaires dépasse le cadre de ce livre, je vous invite à jeter un coup d'œil à mon article de blog[7], dans lequel j'explique ce concept et propose des exemples pratiques.

Les tests unitaires sont souvent configurés pour s'exécuter automatiquement lorsque vous enregistrez votre code sur un système de contrôle de version. La section suivante explique ce que sont les systèmes de contrôle de version et pourquoi ils sont difficiles à utiliser avec les fichiers Excel.

7 https://www.xlwings.org/blog/unittests-for-microsoft-excel

Contrôle de version

Les programmeurs professionnels utilisent des systèmes pour le *contrôle de version* que l'on appelle aussi *contrôle de code source*. Un système de contrôle de version (dont le sigle anglais est VCS pour *version control system*) suit les modifications apportées à votre code source au fil du temps, ce qui permet de voir qui a changé quoi, quand et pourquoi, et donne la possibilité de revenir aux anciennes versions à tout moment. Actuellement, le système de contrôle de version le plus populaire est Git. Créé à l'origine pour gérer le code source de Linux, il a aujourd'hui conquis le monde de la programmation et même Microsoft a adopté Git en 2017 pour gérer le code source de Windows. Dans le monde d'Excel, en revanche, le système de contrôle de version de loin le plus populaire se présente sous la forme d'un dossier où les fichiers sont archivés de la manière suivante :

```
currency_converter_v1.xlsx
currency_converter_v2_2020_04_21.xlsx
currency_converter_final_modifications_Bernard.xlsx
currency_converter_final_final.xlsx
```

Si, contrairement à cet exemple, le développeur Excel respecte une convention dans le nom des fichiers, il n'y a rien de fondamentalement mauvais dans cette approche. Mais en conservant l'historique des versions de vos fichiers en local, vous vous interdisez le bénéfice de fonctionnalités importantes du contrôle de code source, comme une collaboration plus facile, l'évaluation par les pairs, les processus d'approbation et les journaux d'audit. Et si vous voulez améliorer la sécurité et la robustesse de vos classeurs, il n'est pas souhaitable de se passer de ces fonctionnalités. Le plus souvent, les programmeurs professionnels utilisent Git conjointement avec une plateforme Web comme GitHub, GitLab, Bitbucket ou Azure DevOps. Ces plateformes permettent de travailler avec ce que l'on appelle des *pull requests* ou des *merge requests*. Elles permettent aux développeurs de demander officiellement que leurs modifications soient fusionnées dans la base de code principale. Une *pull request* offre les informations suivantes :

- qui est l'auteur des modifications ;
- quand les modifications ont été effectuées ;
- quel est l'objectif des modifications telles que décrites dans le message de commit ;
- quels sont les détails des modifications telles qu'elles apparaissent dans la vue diff, c'est-à-dire une vue qui met en évidence les modifications en vert pour le nouveau code et en rouge pour le code supprimé.

Cela permet à un collègue ou à un chef d'équipe d'examiner les modifications et de repérer les irrégularités. Souvent, un regard extérieur sera en mesure de détecter un problème ou de formuler des commentaires précieux au programmeur. Avec tous ces avantages, pourquoi les développeurs Excel préfèrent-ils utiliser le système de fichiers en local et leur propre convention de nommage des fichiers à la place d'un système professionnel comme Git ?

- De nombreux utilisateurs d'Excel ne connaissent tout simplement pas Git ou l'abandonnent rapidement, car son apprentissage n'est pas facile.
- Git permet à plusieurs utilisateurs de travailler en parallèle sur des copies locales du même fichier. Une fois qu'ils ont tous « commité » leur travail, Git peut généralement fusionner toutes les modifications sans aucune intervention manuelle, mais cela ne fonctionne pas pour les fichiers Excel : s'ils sont modifiés en parallèle sur des copies séparées, Git ne sait pas comment fusionner ces modifications en un seul fichier.
- Même si vous parvenez à traiter les problèmes précédents, Git n'a pas autant d'avantages avec les fichiers Excel qu'avec les fichiers texte : Git n'est pas en mesure d'afficher les modifications entre les fichiers Excel, ce qui empêche un processus d'examen par les pairs.

En raison de tous ces problèmes, ma société a mis au point xltrail[8], un système de contrôle de version basé sur Git qui sait comment gérer les fichiers Excel. xltrail cache la complexité de Git afin que les utilisateurs professionnels se sentent à l'aise et permet également de se connecter à des systèmes Git externes, au cas où vous suiviez déjà vos fichiers avec GitHub, par exemple. xltrail assure le suivi des différents composants d'un classeur, y compris les formules de calcul, les plages nommées, les requêtes Power Query et le code VBA, ce qui permet de tirer parti des avantages classiques du contrôle de version, y compris les évaluations par les pairs.

Pour faciliter le contrôle de version avec Excel, on peut aussi déplacer la logique métier d'Excel vers des fichiers Python, et c'est ce que nous ferons au Chapitre 10. Comme les fichiers Python sont simples à suivre avec Git, vous aurez la partie la plus importante de votre feuille de calcul sous contrôle.

Bien que cette section se nomme « Bonnes pratiques de programmation », elle met surtout l'accent sur les difficultés à les suivre avec Excel par rapport à un langage de programmation traditionnel comme Python. Avant de passer à Python, je voudrais présenter brièvement Power Query et Power Pivot, qui est la réponse de Microsoft dans sa tentative de modernisation d'Excel.

Excel moderne

L'ère moderne d'Excel a commencé avec Excel 2007 lorsque le système des menus déroulants a été remplacé par celui du ruban et que les nouveaux formats de fichier (*xlsx* à la place de *xls*) ont été introduits. Toutefois, la communauté Excel évoque la *modernité* d'Excel pour faire référence aux outils qui ont été ajoutés à Excel 2010, surtout Power Query et Power Pivot, qui permettent de se connecter à des sources de données externes et d'analyser des données trop volumineuses pour tenir dans une feuille de calcul. Comme les fonctionnalités de ces outils se recoupent avec ce que nous ferons avec pandas au Chapitre 5, je vais les présenter brièvement dans la première partie de cette section. La deuxième partie concerne Power BI, qui peut être décrit comme une

8 https://www.xltrail.com/

application autonome d'aide à la décision combinant les fonctionnalités de Power Query et de Power Pivot avec des possibilités de visualisation ; en outre, Power BI dispose d'une prise en charge intégrée de Python.

Power Query et Power Pivot

Avec Excel 2010, Microsoft a introduit un complément appelé *Power Query*. Power Query se connecte à une multitude de sources de données, notamment des classeurs Excel, des fichiers CSV et des bases de données SQL. Il offre également des connexions à des plateformes telles que Salesforce et peut même être étendu pour se connecter à des systèmes qui ne sont pas prévus à l'origine. La fonctionnalité principale de Power Query consiste à traiter les jeux de données qui sont trop volumineux pour tenir dans une feuille de calcul. Après avoir chargé les données, vous pouvez effectuer des étapes supplémentaires pour les nettoyer et les manipuler afin qu'elles arrivent sous une forme utilisable dans Excel. Vous pouvez, par exemple, fractionner une colonne en deux, fusionner deux tables ou filtrer et regrouper vos données. Depuis Excel 2016, Power Query n'est plus un complément, mais est accessible directement sous l'onglet du ruban Données *via* le bouton Obtenir des données. Power Query n'est que partiellement disponible sous macOS, mais son développement continue et il devrait donc être entièrement pris en charge dans une future version d'Excel.

Power Pivot va de pair avec Power Query : conceptuellement, c'est la deuxième étape après l'acquisition et le nettoyage de vos données avec Power Query. Power Pivot vous aide à analyser et à présenter vos données de manière attrayante directement dans Excel. Vous pouvez vous représenter Power Pivot comme un tableau croisé dynamique traditionnel qui, comme Power Query, peut traiter des jeux de données volumineux. Power Pivot permet de définir des modèles de données formels avec des relations et des hiérarchies, et d'ajouter des colonnes calculées *via* le langage de formule DAX. Power Pivot a également été introduit avec Excel 2010, mais il demeure un complément qui n'est jusqu'à présent pas disponible sous macOS.

Si vous aimez travailler avec Power Query et Power Pivot et que vous souhaitez créer des tableaux de bord à partir de ces outils, Power BI peut vous intéresser et nous allons voir pourquoi…

Power BI

Power BI est une application autonome qui est sortie en 2015. C'est la réponse de Microsoft aux outils d'aide à la décision comme Tableau ou Qlik. Power BI Desktop est gratuit, de telle sorte que si vous souhaitez le tester, il suffit d'accéder à la page d'accueil[9] de Power BI et de le télécharger ; notez toutefois que Power BI Desktop n'est disponible que pour Windows. L'objectif de Power BI est de donner un sens aux jeux de données volumineux en les visualisant dans des tableaux de bord interactifs. Fondamentalement, il s'appuie sur les mêmes fonctionnalités Power Query et Power Pivot disponibles dans Excel. Les offres commerciales vous permettent

9 https://powerbi.microsoft.com/fr-fr/

de collaborer et de partager des tableaux de bord en ligne, mais il s'agit de produits distincts de la version Desktop. Power BI nous intéresse particulièrement dans le contexte de ce livre, car il prend en charge les scripts Python depuis 2018. Python peut être utilisé pour la partie requête ainsi que pour la partie visualisation, grâce aux bibliothèques de traçage de Python. Selon moi, l'utilisation de Python dans Power BI semble un peu maladroite, mais le plus important est que Microsoft ait reconnu le rôle important que joue Python en matière d'analyse des données. En conséquence, on peut raisonnablement espérer qu'un jour Python trouve également une place officielle dans Excel.

Alors, qu'y a-t-il de si génial dans Python pour que Microsoft l'ait intégré à Power BI ? La section suivante apporte quelques réponses.

Python pour Excel

Excel permet de stocker, d'analyser et de visualiser des données. Comme Python est particulièrement doué dans le domaine de l'informatique scientifique, il s'associe naturellement à Excel. Python est également l'un des très rares langages qui plaît à la fois au programmeur professionnel ainsi qu'à l'utilisateur débutant qui écrit quelques lignes de code de temps en temps. Les programmeurs professionnels aiment travailler avec Python, car il s'agit d'un langage de programmation polyvalent qui permet donc de réaliser à peu près n'importe quoi sans se casser la tête. Les débutants, en revanche, aiment Python car il est plus facile à apprendre que les autres langages. En conséquence, Python est utilisé à la fois pour l'analyse de données et les tâches d'automatisation plus petites, ainsi que dans d'énormes bases de code de production comme le backend d'Instagram[10]. Cela signifie également que lorsque votre application Excel dopée par Python devient très populaire, il est facile d'ajouter un développeur Web au projet afin de transformer votre prototype Excel-Python en une application Web à part entière. L'avantage unique de Python est que la partie avec la logique métier n'a probablement pas besoin d'être réécrite, mais peut être migrée telle quelle à partir du prototype Excel vers l'environnement Web de production.

Dans cette section, je vais présenter les concepts de base de Python et les comparer avec Excel et VBA. J'aborderai la lisibilité du code, la bibliothèque standard et le gestionnaire de packages de Python, la pile de calcul scientifique, les fonctionnalités du langage moderne et la compatibilité multiplateforme.

Lisibilité et maintenabilité

Si votre code est lisible, cela signifie qu'il est facile à suivre et à comprendre, en particulier pour les personnes extérieures qui n'ont pas écrit le code elles-mêmes. Cela facilite la recherche d'er-

10 Vous pouvez en apprendre plus sur la façon dont Instagram utilise Python sur leur blog : https://instagram-engineering.com/static-analysis-at-scale-an-instagram-story-8f498ab71a0c

reurs et la maintenance du code. C'est la raison pour laquelle une des maximes du *Zen de Python* affirme que « la lisibilité compte ». Le Zen de Python est un résumé des principes de conception de base de Python, et nous apprendrons à l'afficher au chapitre suivant. Jetons un coup d'œil à l'extrait de code VBA suivant :

```
If < 5 Then
    Debug.Print "i est inférieur à 5"
ElseIf i < = 10 Then
    Debug.Print "i est compris entre 5 et 10"
Else
    Debug.Print "i est supérieur à 10"
End If
```

En VBA, vous pouvez reformater l'extrait de code précédent sous la forme suivante, qui est fonctionnellement équivalente :

```
If < 5 Then
    Debug.Print "i est inférieur à 5"
    ElseIf i < = 10 Then
    Debug.Print "i est compris entre 5 et 10"
    Else
    Debug.Print "i est supérieur à 10"
End If
```

Dans la première version, l'indentation s'aligne sur la logique du code, ce qui en facilite la lecture et la compréhension, ce qui, à son tour, simplifie la recherche d'erreurs. Dans la deuxième version, un développeur néophyte peut ne pas voir les conditions `ElseIf` et `Else` lorsqu'il voit le code pour la première fois, ce qui est évidemment encore plus vrai si le code fait partie d'une base de code plus grande.

Python ne tolère pas que le code soit mis en forme comme dans le deuxième exemple : il vous oblige à aligner l'indentation avec la logique du code, ce qui évite les problèmes de lisibilité. Python offre cette fonctionnalité, car il s'appuie sur l'indentation pour définir les blocs de code tels que vous les utilisez dans les instructions `if` ou les boucles `for`. À la place des indentations, la majorité des autres langages utilisent des accolades, et VBA utilise des mots-clés tels que `End If`, comme nous venons de le voir dans les extraits de code. Si Python emploie des indentations pour les blocs de code, c'est parce qu'en programmation, on consacre plus de temps à maintenir le code qu'à l'écrire. La lisibilité du code aide les programmeurs néophytes (ou vous-même, quelques mois après que vous avez écrit un programme) à vous replonger dans le code et à comprendre ce qui se passe.

Nous allons tout apprendre sur les règles d'indentation de Python au Chapitre 3, mais pour l'instant, nous allons étudier la bibliothèque standard qui est fournie avec Python.

Bibliothèque standard et gestionnaire de packages

Python est livré avec un riche ensemble de fonctionnalités natives fournies par sa *bibliothèque standard*. La communauté Python aime s'y référer en disant que Python est livré avec des « batteries incluses ». Que vous ayez besoin de décompresser un fichier ZIP, de lire les valeurs d'un fichier CSV ou de récupérer des données sur Internet, la bibliothèque standard de Python couvre vos besoins, et vous pouvez réaliser tout cela en quelques lignes de code. La même fonctionnalité dans VBA vous obligerait à écrire une quantité considérable de code ou à installer un complément. Et souvent, les solutions que vous trouvez sur Internet ne fonctionnent que sous Windows, mais pas sous macOS.

Bien que la bibliothèque standard de Python couvre une quantité impressionnante de fonctionnalités, il existe toujours des tâches fastidieuses à programmer ou lentes lorsque vous ne comptez que sur la bibliothèque standard. C'est à ce moment qu'entre en jeu PyPI (*https://pypi.org/*). PyPI, qui est l'abréviation de *Python Package Index*, est un dépôt géant où tout le monde (y compris vous-même) peut télécharger des packages Python open source qui ajoutent des fonctionnalités à Python.



PyPI se prononce « paille pie aille » afin de le différencier de PyPy qui se prononce « paille paille » et qui est une implémentation alternative de Python.

Par exemple, pour faciliter l'extraction de données à partir de sources sur Internet, vous pouvez installer le package Requests afin d'accéder à un ensemble de commandes puissantes, mais faciles à utiliser. Pour l'installer, vous devez utiliser le gestionnaire de packages *pip* de Python, que vous exécutez dans une invite de commandes ou un terminal. pip est un acronyme récursif signifiant *pip installe les packages*. Ne vous inquiétez pas si cela vous semble un peu abstrait pour le moment, car j'expliquerai comment cela fonctionne en détail dans le chapitre suivant. L'une des principales raisons est que tout package comprenant de nombreuses fonctionnalités ne dépendra pas seulement de la bibliothèque standard de Python, mais aussi d'autres packages open source également hébergés sur PyPI. Ces dépendances peuvent à leur tour dépendre de sous-dépendances et ainsi de suite. pip vérifie de manière récursive les dépendances et sous-dépendances d'un package, les télécharge et les installe. pip facilite également la mise à jour de vos packages afin que vous puissiez garder à jour vos dépendances. Cela rend le respect du principe DRY beaucoup plus facile, car vous n'avez pas besoin de réinventer ou de copier-coller ce qui est déjà disponible sur PyPI. Avec pip et PyPI, vous disposez également d'un mécanisme solide pour distribuer et installer ces dépendances, ce qui fait défaut à Excel avec ses compléments traditionnels.

Logiciels open source

À ce stade, j'aimerais dire quelques mots sur l'*open source*, car j'ai utilisé cette expression à quelques reprises dans cette section. Si le logiciel est distribué sous une licence open source, cela signifie que son code source est disponible gratuitement, ce qui permet à chacun de contribuer à de nouvelles fonctionnalités, à des corrections de bogues ou à sa documentation. Python, lui-même, et presque tous les packages Python tiers, sont open source et le plus souvent maintenus par des développeurs sur leur temps libre. Ce n'est pas toujours une situation idéale, car si votre entreprise s'appuie sur certains packages, vous avez intérêt que le développement et la maintenance de ces packages soient assurés de manière continue par des programmeurs professionnels. Heureusement, la communauté scientifique Python a reconnu que certains packages sont trop importants pour confier leur sort entre les mains de quelques bénévoles qui travaillent le soir et le week-end.

C'est la raison pour laquelle en 2012, NumFOCUS (*https://numfocus.org/*), une organisation à but non lucratif, a été créée pour parrainer divers packages et projets Python dans le domaine de l'informatique scientifique. Parmi les packages Python les plus populaires sponsorisés par NumFOCUS, on peut citer pandas, NumPy, SciPy, Matplotlib et Project Jupyter, mais aujourd'hui, cette organisation prend également en charge différents packages d'autres langages, notamment R, Julia et JavaScript. Il y a quelques grandes entreprises qui sont des sponsors, mais tout le monde peut rejoindre gratuitement NumFOCUS en tant que membre de la communauté.

Avec pip, vous pouvez installer des packages pour à peu près n'importe quelle discipline, mais pour les utilisateurs d'Excel, les plus intéressants sont certainement les packages consacrés à l'informatique scientifique, sujet que nous allons développer dans la section suivante.

Informatique scientifique

Le succès de Python est principalement dû au fait qu'il a été créé en tant que langage de programmation à usage général. Les capacités de calcul scientifique ont été ajoutées ultérieurement sous la forme de packages tiers. Cela présente l'avantage unique qu'un expert des données peut utiliser le même langage pour ses expériences et la recherche qu'un développeur Web, qui peut éventuellement créer une application prête à être mise en production en se basant sur le noyau de calcul. La possibilité de créer des applications scientifiques à partir d'un seul langage réduit les problèmes, le temps de mise en œuvre et les coûts. Des packages scientifiques comme NumPy, SciPy et pandas permettent de formuler des problèmes mathématiques de façon très concise. À titre d'exemple, jetons un coup d'œil à l'une des formules financières les plus célèbres, qui est utilisée pour calculer la variance d'un portefeuille selon la Théorie moderne du portefeuille :

$$\sigma^2 = w^T C w$$

La variance du portefeuille est indiquée par σ^2, tandis que `w` est le vecteur de pondération des actifs individuels et `C` la matrice de covariance du portefeuille. Si `w` et `C` sont des plages Excel, vous pouvez calculer la variance du portefeuille en VBA de la manière suivante :

```
variance = Application.MMult(Application.MMult(Application.Transpose(w), C), w)
```

Comparez cela à la notation presque mathématique en Python, en supposant que `w` et `C` soient des DataFrames pandas ou des tableaux NumPy (je présenterai en détail ces notions dans la Partie II) :

```
variance = w.T @ C @ w
```

Il ne s'agit pas seulement d'une question d'esthétique et de lisibilité : NumPy et pandas utilisent sous le capot du code Fortran et C compilé, ce qui améliore les performances lorsque vous travaillez avec de grandes matrices par rapport à VBA.

L'absence de prise en charge de l'informatique scientifique est une lacune évidente de VBA. Mais même si l'on considère les fonctionnalités de base du langage, VBA a pris du retard, comme je vais le montrer dans la section suivante.

Caractéristiques des langages modernes

Depuis Excel 97, le langage VBA n'a pas connu de changements majeurs en termes de fonctionnalités. Cela ne signifie pas cependant que VBA n'est plus maintenu par Microsoft qui livre des mises à jour à chaque nouvelle version d'Excel afin de pouvoir programmer les nouvelles fonctionnalités. Par exemple, Excel 2016 a ajouté la prise en charge de l'automatisation de Power Query. Pour autant, un langage qui a cessé d'évoluer il y a plus de vingt ans ne bénéficie pas des concepts modernes qui ont été introduits dans les principaux langages de programmation au fil des ans. Par exemple, la gestion des erreurs en VBA paraît vraiment vieillotte. Si vous souhaitez gérer une erreur correctement en VBA, voilà à quoi cela ressemble :

```
Sub AfficheInverse(nombre As Variant)
    ' Erreur si nombre = 0 ou est une chaîne de caractères
    On Error GoTo ErrorHandler
        resultat = 1 / nombre
    On Error GoTo 0
    Debug.Print "Il n'y a pas eu d'erreur !"
Finally :
    ' S'exécute qu'une erreur se produise ou non
    If resultat = "" Then
        resultat = "N/A"
    End If
    Debug.Print "L'inverse est : " & resultat
    Exit Sub
ErrorHandler :
    ' S'exécute uniquement en cas d'erreur
    Debug.Print "Il y a eu une erreur : " & Err.Description
    Resume Finally
End Sub
```

La gestion des erreurs en VBA implique l'utilisation d'*étiquettes* telles que `Finally` et `ErrorHandler` dans cet exemple. On demande au code d'accéder à ces étiquettes *via* les instructions `GoTo` ou `Resume`. Très tôt, il a été reconnu que les étiquettes étaient responsables de ce que de nombreux programmeurs appellent du *code spaghetti* : une façon élégante de dire que le flux du code est difficile à suivre et donc difficile à maintenir. C'est la raison pour laquelle à peu près tous les langages modernes ont introduit le mécanisme `try/catch` (qui se nomme en Python `try/except`) que je détaillerai dans le Chapitre 11. Si vous êtes un développeur VBA chevronné, vous apprécierez peut-être le fait que Python prenne en charge l'héritage de classe, une fonctionnalité de programmation orientée objet qui manque en VBA.

Outre les fonctionnalités des langages modernes, il existe une autre exigence importante : la compatibilité multiplateforme.

Compatibilité multiplateforme

Même si vous développez votre code sur un ordinateur local qui s'exécute sous Windows ou macOS, il est très probable que vous souhaitiez à un moment donné exécuter votre programme sur un serveur ou dans le cloud. Les serveurs permettent à votre code d'être exécuté selon une planification précise et accessible partout où vous le souhaitez, en bénéficiant de la puissance de calcul dont vous avez besoin. En fait, je vais vous montrer dans le chapitre suivant comment exécuter du code Python sur un serveur en vous présentant les notebooks Jupyter. La grande majorité des serveurs fonctionnent sous Linux, car il s'agit d'un système d'exploitation stable, sécurisé et rentable. Et comme les programmes Python s'exécutent sans changement sur tous les principaux systèmes d'exploitation, cela éliminera une grande partie des problèmes lorsque vous voudrez passer de votre machine locale à une configuration de production.

En revanche, même si Excel VBA s'exécute sous Windows et macOS, il est facile d'introduire des fonctionnalités qui ne s'exécutent que sous Windows. Dans la documentation officielle de VBA ou sur les forums, vous verrez souvent du code comme celui-ci :

```
Set fso = CreateObject("Scripting.FileSystemObject")
```

Chaque fois que vous avez un appel à `CreateObject` ou qu'il faut accéder au menu Outils > Références dans l'éditeur VBA pour ajouter une référence, vous avez presque toujours affaire à du code qui ne s'exécutera que sous Windows. Les *contrôles ActiveX* sont un autre problème important auquel vous devez faire attention si vous souhaitez que vos fichiers Excel fonctionnent à la fois sous Windows et macOS. Les contrôles ActiveX sont des éléments tels que des boutons et des listes déroulantes que vous pouvez placer sur vos feuilles de calcul, mais ils fonctionnent uniquement sous Windows. Veillez à les éviter si vous souhaitez que votre classeur s'exécute également sous macOS.

Conclusion

Dans ce chapitre, nous avons fait la connaissance de Python et Excel, deux technologies très populaires qui existent depuis plusieurs décennies, ce qui constitue une longue période par rapport à de nombreuses technologies que nous employons aujourd'hui. La Baleine de Londres a servi d'exemple de ce qu'il ne faut pas faire et qui peut coûter très cher lorsque vous n'utilisez pas Excel correctement avec des classeurs critiques. C'est ce qui nous a incités à nous pencher sur un ensemble minimal de bonnes pratiques en matière de programmation : règle de séparation des problèmes, principe DRY, utilisation de tests automatisés et contrôle de version. Nous avons ensuite examiné Power Query et Power Pivot, l'approche de Microsoft pour traiter les données plus grandes que votre feuille de calcul. J'estime cependant qu'il ne s'agit pas de la bonne solution, car ces outils vous enferment dans le monde Microsoft et vous empêchent de profiter de la flexibilité et de la puissance des solutions modernes basées sur le cloud.

Python est livré avec des fonctionnalités qui ont fait leurs preuves et qui font défaut à Excel : la bibliothèque standard, le gestionnaire de packages, les bibliothèques pour le calcul scientifique et la compatibilité multiplateforme. En apprenant à combiner Excel avec Python, vous pouvez bénéficier du meilleur des deux mondes ; vous allez gagner du temps grâce à l'automatisation, commettre moins d'erreurs car il est plus facile de suivre les bonnes pratiques de programmation, et vous serez en mesure de faire évoluer votre application en dehors d'Excel si vous en avez besoin.

Maintenant que vous savez pourquoi Python est un compagnon si puissant pour Excel, il est temps de configurer votre environnement de développement afin de pouvoir écrire vos premières lignes de code Python !

CHAPITRE 2

Environnement de développement

Vous avez probablement hâte d'apprendre les bases de Python, mais avant d'en arriver là, vous devez d'abord configurer votre ordinateur. Pour écrire du code VBA ou des requêtes, il suffit de lancer Excel et d'ouvrir l'éditeur VBA ou Power Query, mais pour écrire du code Python, cela nécessite un peu plus de travail.

Nous allons commencer ce chapitre en installant la distribution Python Anaconda. Outre l'installation de Python, Anaconda nous donnera également accès à Anaconda Prompt et aux notebooks Jupyter, deux outils essentiels que nous utiliserons tout au long de ce livre. *Anaconda Prompt* est une invite de commande spéciale (Windows) ou un terminal (macOS) qui permet d'exécuter des scripts Python et d'autres outils de ligne de commande que nous aborderons dans ce livre. Les *notebooks Jupyter* permettent de travailler avec des données, du code et des graphiques de manière interactive, ce qui en fait un concurrent sérieux des classeurs Excel. Après avoir testé les notebooks Jupyter, nous installerons *Visual Studio Code* (VS Code), un éditeur de texte puissant. VS Code, qui fonctionne parfaitement pour l'écriture, l'exécution et le débogage de scripts Python, est fourni avec un terminal intégré. La Figure 2.1 résume ce qui est inclus dans Anaconda et VS Code.

Comme ce livre porte sur Excel, je me concentre sur Windows et macOS dans ce chapitre, mais tout ce qui est décrit dans les trois premières parties de ce livre fonctionne également sous Linux. Commençons par installer Anaconda !

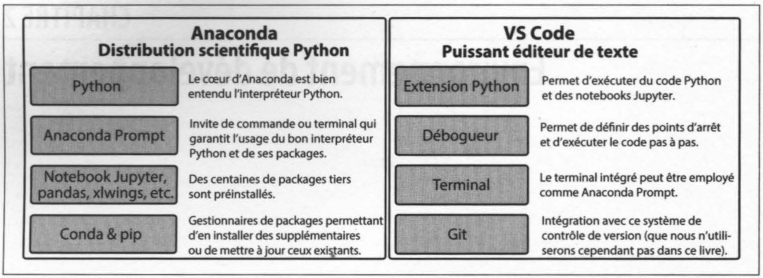


Figure 2.1 : Environnement de développement.

Distribution Python Anaconda

Anaconda, qui est sans doute la distribution Python la plus populaire pour la science des données, est livré avec des centaines de packages tiers préinstallés : cette distribution inclut les notebooks Jupyter et la plupart des autres packages que ce livre utilisera largement, notamment pandas, OpenPyXL et xlwings. L'édition individuelle d'Anaconda est gratuite pour un usage privé et garantit que tous les packages inclus sont compatibles les uns avec les autres. Elle s'installe dans un dossier unique et peut facilement être désinstallée. Après l'avoir installée, nous apprendrons quelques commandes de base d'Anaconda Prompt et exécuterons une session Python interactive. Nous aborderons ensuite les gestionnaires de packages Conda et pip, avant de conclure cette section avec les environnements Conda. Commençons par télécharger et installer Anaconda !

Installation

Accédez à la page d'accueil d'Anaconda[1] et téléchargez la dernière version du programme d'installation d'Anaconda (édition individuelle). Assurez-vous de télécharger le programme d'installation graphique 64 bits pour la version 3.x de Python[2]. Une fois téléchargé, faites un double-clic sur le programme d'installation afin de démarrer le processus d'installation et acceptez toutes les valeurs par défaut. Pour obtenir des instructions d'installation plus détaillées, consultez la documentation officielle[3].

1 https://www.anaconda.com/products/individual

2 Les systèmes 32 bits n'existent que sous Windows et ils deviennent de plus en plus rares. Pour trouver facilement quelle version de Windows vous avez, ouvrez l'Explorateur de fichiers et allez sur le lecteur C:\. Si vous pouvez voir les dossiers Program Files et Program Files (x86), vous êtes sur une version 64 bits de Windows. Si vous ne pouvez voir que le dossier Program Files, vous êtes sur un système 32 bits.

3 https://docs.anaconda.com/anaconda/install/



Bien que les instructions de ce livre partent du principe que vous avez installé l'édition individuelle d'Anaconda, le code fonctionnera également avec toute autre installation Python. Dans ce cas, toutefois, vous devrez installer les dépendances requises en suivant les instructions incluses dans le fichier *requirements.txt* des fichiers électroniques de ce livre.

Quand Anaconda est installé, nous pouvons commencer à utiliser Anaconda Prompt.

Anaconda Prompt

Anaconda Prompt n'est en fait qu'une invite de commande sous Windows et un terminal sous macOS qui ont été configurés pour fonctionner avec le bon interpréteur Python et des packages tiers. Anaconda Prompt est l'outil le plus basique qui soit pour exécuter du code Python, et nous en ferons un usage intensif dans ce livre pour exécuter des scripts Python et toutes sortes d'outils de ligne de commande proposés par différents packages.



Si vous ne vous servez pas de la distribution Python Anaconda, vous devrez employer l'invite de commande sous Windows et le terminal sous macOS chaque fois que je vous demanderai d'utiliser Anaconda Prompt.

Si vous n'avez jamais utilisé d'invite de commande sous Windows ou de terminal avec macOS, ne vous inquiétez pas : il vous suffit de connaître quelques commandes qui vous permettront déjà de faire beaucoup de choses. Une fois que vous vous y serez habitué, l'utilisation d'Anaconda Prompt est souvent plus rapide et plus pratique que de cliquer sur les différents menus de l'interface utilisateur graphique. C'est parti !

Windows

Cliquez sur le bouton du menu Démarrer et commencez à saisir **`Anaconda Prompt`**. Dans les entrées qui apparaissent, choisissez Anaconda Prompt, et non pas Anaconda Prompt Powershell. Sélectionnez-le au clavier, en vous déplaçant avec les flèches, et appuyez sur Entrée ou utilisez votre souris pour cliquer dessus. Si vous préférez l'ouvrir *via* le menu Démarrer, vous le trouverez sous Anaconda3. C'est une bonne idée d'épingler Anaconda Prompt à votre barre des tâches Windows, car vous l'utiliserez régulièrement tout au long de ce livre. La ligne d'entrée d'Anaconda Prompt commencera par `(base)` :

```
(base) C:\Utilisateurs\felix>
```

macOS

Sur Mac, vous ne trouverez pas d'application appelée Anaconda Prompt. Quand j'évoquerai Anaconda Prompt, je ferai en fait référence au terminal qui a été configuré par le programme d'installation d'Anaconda pour activer automatiquement un environnement Conda (je détaillerai les environnements Conda dans un instant) : appuyez sur la combinaison Commande + Espace ou ouvrez le Launchpad, puis saisissez **`Terminal`** et appuyez

sur Entrée. Vous pouvez également ouvrir le Finder et accéder à *Applications* > *Utilitaires*, où vous trouverez l'application Terminal sur laquelle vous pouvez faire un double-clic. Une fois que le terminal apparaît, il devrait ressembler à une ligne d'entrée commençant par `(base)` :

```
(base) felix@MacBook-Pro ~ %
```

Si vous utilisez une ancienne version de macOS, cela ressemble plutôt à ceci :

```
(base) MacBook-Pro :~ felix$
```

Contrairement à l'invite de commandes de Windows, le terminal sous macOS n'affiche pas le chemin d'accès complet du répertoire actif. Au lieu de cela, le tilde représente le répertoire de base, qui est généralement */Utilisateurs/<nom_utilisateur>*. Pour voir le chemin d'accès complet de votre répertoire actif, saisissez **`pwd`** suivi de la touche Entrée. `pwd` est l'abréviation de *print working directory* (qui signifie afficher le répertoire de travail).

Si la ligne d'entrée de votre Terminal ne commence pas par `(base)` après l'installation d'Anaconda, c'est en général parce que le Terminal était en cours d'exécution pendant l'installation d'Anaconda ; il vous suffit alors de redémarrer le Terminal pour régler le problème. Notez que si vous cliquez sur la croix rouge en haut à gauche de la fenêtre du Terminal, vous ne fermez pas le programme, mais vous ne faites que le masquer. Pour quitter réellement l'application, faites un clic droit sur le Terminal dans le Dock et sélectionnez Quitter ou appuyez sur Commande + Q quand le Terminal est la fenêtre active. Lorsque vous le redémarrez et que le Terminal affiche `(base)` au début d'une nouvelle ligne, vous êtes alors prêt. C'est une bonne idée de garder le terminal dans le Dock, car vous l'utiliserez régulièrement tout au long de ce livre.

Lorsque Anaconda Prompt est opérationnel, essayez les commandes listées dans le Tableau 2.1 (les commandes sont décrites après le tableau).

Tableau 2.1 : Commandes d'Anaconda Prompt.

Commande	Windows	macOS
Liste les fichiers du répertoire actif	`dir`	`ls -la`
Change de répertoire (relatif)	`cd chemin\répertoire`	`cd chemin/répertoire`
Change de répertoire (absolu)	`cd C:\chemin\répertoire`	`cd /chemin/répertoire`
Passe au lecteur D	`D:`	(n'existe pas)
Passe au répertoire parent	`cd ..`	`cd ..`
Fait défiler les commandes précédentes	↑ (flèche vers le haut)	↑ (flèche vers le haut)

Lister les fichiers du répertoire actif

- Sous Windows, saisissez **`dir`** (abréviation de directory, qui signifie répertoire) puis appuyez sur Entrée. Cela affiche le contenu du répertoire dans lequel vous vous trouvez actuellement.
- Sous macOS, saisissez **`ls -la`** suivi de la touche Entrée. `ls` est l'abréviation de list directory contents (liste le contenu du répertoire) et `-la` affiche la sortie sous la forme d'une liste longue incluant tous les fichiers, y compris les fichiers cachés.

Changer de répertoire

Saisissez **`cd Down`** et appuyez sur la touche Tab. `cd` est l'abréviation de *change directory* (changer de répertoire). Si vous êtes dans votre dossier personnel, Anaconda Prompt devrait très probablement être en mesure de compléter automatiquement votre saisie en `cd Downloads`. Si vous êtes dans un dossier différent ou si vous n'avez pas de dossier appelé *Downloads*, commencez simplement à saisir le début de l'un des noms de répertoires que vous avez listés avec la commande précédente (`dir` ou `ls -la`) avant d'appuyer sur la touche Tab pour la complétion automatique. Appuyez ensuite sur Entrée pour vous déplacer dans le répertoire dont le nom a été complété automatiquement. Si vous êtes sous Windows et si vous devez changer de lecteur, vous devez d'abord saisir le nom du lecteur avant de pouvoir changer de répertoire :

```
C:\Users\felix> D:
D:\> cd data
D:\data>
```

Notez que si vous commencez votre chemin d'accès par un répertoire ou un nom de fichier qui se trouve dans votre répertoire actuel, vous utilisez un *chemin relatif*, par exemple, `cd Downloads`. Si vous souhaitez sortir de votre répertoire actuel, vous pouvez saisir un chemin *absolu*, par exemple, `cd C:\Utilisateurs` sous Windows ou `cd /Utilisateurs` sous macOS (attention au slash au début).

Passer au répertoire parent

Pour accéder à votre répertoire parent, c'est-à-dire à un niveau supérieur dans la hiérarchie de vos répertoires, saisissez **`cd ..`** suivi de la touche Entrée (assurez-vous qu'il y a un espace entre `cd` et les deux points). Vous pouvez combiner cela avec un nom de répertoire ; par exemple, si vous souhaitez monter d'un niveau, puis vous déplacer dans le répertoire *Desktop*, saisissez **`cd ..\Desktop`**. Sous macOS, remplacez la barre oblique inverse (antislash) par une barre oblique (slash).

Faire défiler les commandes précédentes

Utilisez la touche flèche vers le haut pour faire défiler les commandes précédentes. Cela vous permettra d'économiser la frappe de nombreuses touches si vous avez besoin d'exécuter les mêmes commandes de manière répétée. Si vous remontez trop loin en arrière dans

l'historique des commandes, utilisez la touche flèche vers le bas pour faire défiler dans l'autre sens.

Extensions de fichiers

Malheureusement, Windows et macOS masquent par défaut les extensions de fichiers dans l'Explorateur Windows ou le Finder. Cela ne vous facilite pas la tâche quand vous travaillez avec des scripts Python et Anaconda Prompt, car vous devez employer leurs extensions quand vous faites référence à des fichiers. Lorsque vous travaillez avec Excel, l'affichage des extensions de fichiers vous aide également à faire la différence entre les fichiers *xlsx* (extension par défaut), les fichiers *xlsm* (prenant en charge les macros) ou les autres formats de fichiers Excel. Voici comment rendre les extensions de fichiers visibles :

Windows
: Ouvrez l'Explorateur de fichiers et cliquez sur l'onglet Affichage. Sous le groupe Afficher/Masquer, activez la case à cocher Extensions de noms de fichiers.

macOS
: Ouvrez le Finder et accédez à Préférences en appuyant sur Commande + , (Commande + virgule). Sous l'onglet Avancé, cochez la case en regard de Afficher toutes les extensions de fichiers.

Vous pouvez maintenant lancer Anaconda Prompt et exécuter des commandes dans le répertoire souhaité. Vous l'utiliserez immédiatement dans la section suivante, où je vais vous montrer comment démarrer une session Python interactive.

Python REPL : une session Python interactive

Vous pouvez démarrer une session Python interactive en exécutant la commande `python` dans Anaconda Prompt :

```
(base) C:\Users\felix>python
Python 3.8.5 (default, Sep 3 2020, 21 :29 :08) [MSC v.1916 64 bit (AMD64)]: : Anaconda, Inc. on
   win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le texte qui est affiché dans un terminal sous macOS différera légèrement, mais il fonctionne de la même manière. Ce livre est basé sur Python 3.8 : si vous souhaitez utiliser une version plus récente de Python, assurez-vous de consulter la page d'accueil du site Web du livre[4] pour obtenir des instructions.

4 https://xlwings.org/book



À l'avenir, je commencerai des lignes de code avec la mention `(base)>` pour indiquer que les commandes sont saisies dans Anaconda Prompt. Par exemple, pour lancer un interpréteur Python interactif, je vais écrire :

```
(base)> python
```

qui sous Windows ressemblera à ceci :

```
(base) C:\Users\felix> python
```

et sous macOS ressemblera à cela (rappelez-vous que sur Mac, le terminal est votre propre application Anaconda Prompt) :

```
(base) felix@MacBook-Pro ~ % python
```

Commençons par faire quelques tests ! Notez que `>>>` dans une session interactive signifie que Python attend votre entrée si bien que vous n'avez pas besoin de saisir ces caractères. Saisissez chaque ligne qui commence par `>>>` et validez votre commande en appuyant sur la touche Entrée :

```
>>> 3 + 4
7
>>> "python " * 3
'python python python '
```

Cette session Python interactive est également appelée Python *REPL*, qui est l'abréviation de *read-eval-print loop* (boucle de lecture, évaluation et affichage) : Python lit votre saisie, l'évalue et affiche le résultat instantanément en attendant votre prochaine entrée. Vous vous souvenez du Zen de Python que j'ai mentionné dans le chapitre précédent ? Vous pouvez maintenant lire la version complète afin d'avoir un aperçu des principes directeurs de Python[5]. Exécutez simplement cette ligne en appuyant sur Entrée après l'avoir saisie :

```
>>> import this
```

Pour quitter votre session Python, saisissez **`quit()`** suivi de la touche Entrée. Vous pouvez également appuyer sur Ctrl + Z sous Windows, puis appuyez sur la touche Entrée. Sous macOS, appuyez simplement sur Ctrl + D (il n'y a pas besoin d'appuyer sur Entrée).

Après avoir quitté le REPL Python, c'est le bon moment pour tester Conda et pip, les gestionnaires de packages qui sont livrés avec Anaconda.

Gestionnaires de packages : Conda et pip

Dans le chapitre précédent, j'ai déjà dit quelques mots sur pip, le gestionnaire de packages de Python ; pip gère le téléchargement, l'installation, la mise à jour et la désinstallation des packages Python ainsi que leurs dépendances et sous-dépendances. Alors qu'Anaconda fonctionne avec

5 Une traduction en français de ces principes est disponible à https://fr.wikipedia.org/wiki/Zen_de_Python.

pip, il dispose aussi d'un gestionnaire de packages alternatif appelé Conda. L'un des avantages de Conda est qu'il peut installer d'autres éléments que des simples packages Python, notamment des versions supplémentaires de l'interpréteur Python. En résumé, les packages ajoutent à votre installation Python des fonctionnalités supplémentaires qui ne sont pas couvertes par la bibliothèque standard. pandas, que je présenterai en détail au Chapitre 5, est un exemple d'un tel package. Comme il est préinstallé dans l'installation Python d'Anaconda, vous n'avez pas besoin de l'installer manuellement.



Avec Anaconda, vous devez installer tout ce que vous pouvez *via* Conda et n'utiliser pip que pour installer les packages que Conda ne peut pas trouver. Vous noterez également que Conda peut remplacer les fichiers qui ont été précédemment installés avec pip.

Le Tableau 2.2 offre une vue d'ensemble des commandes que vous utiliserez le plus souvent. Ces commandes doivent être saisies dans Anaconda Prompt et vous permettront d'installer, de mettre à jour et de désinstaller vos packages tiers.

Tableau 2.2 : Commandes Conda et pip.

Action	Conda	pip
Liste tous les packages installés	`conda list`	`pip freeze`
Installe la dernière version du package	`conda install package`	`pip install package`
Installe une version de package spécifique	`conda install package=1.0.0`	`pip install package==1.0.0`
Met à jour un package	`conda update package`	`pip install --upgrade package`
Désinstalle un package	`conda remove package`	`pip uninstall package`

Par exemple, pour voir quels packages sont déjà disponibles dans votre distribution Anaconda, saisissez :

```
(base)> conda list
```

Chaque fois que ce livre nécessite un package qui n'est pas inclus dans l'installation d'Anaconda, je le soulignerai explicitement et vous expliquerai comment l'installer. Cependant, il est préférable d'installer les packages manquants maintenant afin que vous n'ayez pas à le faire plus tard. Installons d'abord plotly et xlutils, des packages qui sont disponibles *via* Conda :

```
(base)> conda install plotly xlutils
```

Après avoir exécuté cette commande, Conda vous montrera ce qu'il va faire et vous demandera de confirmer l'installation en saisissant `y` et en appuyant sur Entrée. Une fois cela fait[6], vous pouvez installer pyxlsb et pytrends avec pip, car ces packages ne sont pas disponibles *via* Conda :

```
(base)> pip install pyxlsb pytrends
```

6 Si sous Windows, vous obtenez le message d'erreur « EnvironmentNotWritableError : The current user does not have permissions to the target environment. », quittez Anaconda Prompt et exécutez à nouveau l'application en tant qu'administrateur (clic droit sur le nom de l'application puis Exécuter en tant qu'administrateur dans le menu contextuel).

Contrairement à Conda, pip installera les packages immédiatement lorsque vous appuyez sur Entrée sans avoir besoin de confirmation.



De nombreux packages Python sont souvent mis à jour et introduisent parfois des modifications qui ne sont pas rétrocompatibles. Cela provoquera probablement des erreurs dans certains des exemples de ce livre, mais je tenterai de suivre ces changements et de publier des correctifs sur le site Web dédié à ce livre. Vous pouvez également créer un environnement Conda qui utilise les mêmes versions des packages que j'ai employés lors de l'écriture de ce livre. Je vais présenter les environnements Conda dans la section suivante, et vous trouverez des instructions détaillées sur la façon de créer un environnement Conda avec des packages spécifiques dans l'Annexe A.

Vous savez maintenant comment utiliser Anaconda Prompt pour démarrer un interpréteur Python et installer des packages supplémentaires. Dans la section suivante, je vais vous expliquer ce que signifie le terme `(base)` qui se trouve au début d'Anaconda Prompt.

Environnements Conda

Vous vous demandez peut-être pourquoi Anaconda Prompt affiche la mention `(base)` au début de chaque ligne d'entrée. Il s'agit du nom de l'*environnement Conda* actif. Un environnement Conda est un « monde Python » distinct avec une version spécifique de Python et un ensemble de packages installés avec des versions spécifiques. Pourquoi est-ce nécessaire ? Lorsque vous commencez à travailler sur différents projets en parallèle, il est probable qu'ils auront des exigences différentes : un projet peut utiliser Python 3.8 avec pandas 0.25.0, tandis qu'un autre projet peut utiliser Python 3.9 avec pandas 1.0.0. Le code écrit pour pandas 0.25.0 nécessitera souvent des modifications pour s'exécuter avec pandas 1.0.0, si bien que vous ne pouvez pas simplement mettre à niveau vos versions Python et pandas sans apporter des modifications à votre code. L'utilisation d'un environnement Conda pour chaque projet permet de s'assurer que chaque projet s'exécute avec les bonnes dépendances. Alors que les environnements Conda sont spécifiques à la distribution Anaconda, le concept existe avec chaque installation Python sous le nom d'*environnement virtuel*. Les environnements Conda sont plus puissants car ils facilitent la gestion des différentes versions de Python lui-même, et non pas seulement des packages.

Dans le cadre de ce livre, vous n'aurez pas à modifier votre environnement Conda, car nous utiliserons toujours l'environnement par défaut `(base)`. Toutefois, lorsque vous commencerez à créer des projets réels, il est recommandé d'utiliser un environnement Conda ou un environnement virtuel pour chaque projet afin d'éviter tout conflit potentiel entre leurs dépendances. Tout ce que vous devez savoir sur la gestion de plusieurs environnements Conda est expliqué dans l'Annexe A. Vous y trouverez également des instructions sur la création d'un environnement Conda avec les versions exactes des packages que j'ai utilisés pour écrire ce livre. Cela vous permettra d'exécuter les exemples de ce livre tels quels pendant de nombreuses années. Vous pouvez

également consulter la page d'accueil du site Web du livre pour voir si des modifications sont nécessaires pour les versions plus récentes de Python et des packages.

Après avoir résolu le mystère des environnements Conda, il est temps de présenter le prochain outil, que nous utiliserons intensément dans ce livre : les notebooks Jupyter.

Notebooks Jupyter

Dans la section précédente, je vous ai montré comment démarrer une session Python interactive à partir d'Anaconda Prompt. Ceci est utile si vous souhaitez tester quelque chose de simple. Cependant, pour la majeure partie de votre travail, vous apprécierez un environnement plus facile à utiliser. Par exemple, revenir aux commandes précédentes et afficher des graphiques est difficile avec un REPL Python s'exécutant dans Anaconda Prompt. Heureusement, Anaconda n'est pas livré qu'avec un interpréteur Python, et il inclut également les *notebooks Jupyter*, qui sont devenus l'un des moyens les plus populaires d'exécuter du code Python dans un contexte de science des données. Les notebooks Jupyter permettent de raconter une histoire en combinant du code Python exécutable avec du texte, des images et des graphiques mis en forme dans un notebook interactif qui s'exécute dans un navigateur. Ils sont adaptés aux débutants et donc particulièrement utiles pour les premières étapes de votre découverte de Python. Pour autant, ils sont également extrêmement populaires pour l'enseignement, le prototypage et la recherche, car ils facilitent la recherche reproductible.

Les notebooks Jupyter sont devenus un concurrent sérieux d'Excel, car ils couvrent à peu près les mêmes fonctionnalités qu'un classeur : vous pouvez rapidement préparer, analyser et visualiser des données. La différence avec Excel est que tout cela se produit en écrivant du code Python au lieu de cliquer dans Excel avec votre souris. Les notebooks Jupyter ont aussi un autre avantage, car ils ne mélangent pas les données et la logique métier : un notebook Jupyter contient votre code et vos graphiques, alors que vous importez généralement des données à partir d'un fichier CSV externe ou d'une base de données. Le fait d'avoir du code Python visible dans votre notebook permet de voir facilement ce qui se passe, ce qui n'est pas le cas dans Excel, où les formules sont cachées derrière la valeur d'une cellule. Les notebooks Jupyter sont également faciles à exécuter localement et sur un serveur distant. Les serveurs ont généralement plus de puissance que votre ordinateur local et peuvent exécuter votre code sans que vous ayez besoin d'être devant votre ordinateur, ce qui est difficile à faire avec Excel.

Dans cette section, je vais vous montrer les bases de l'exécution et de la navigation dans un notebook Jupyter : nous allons en apprendre davantage sur les cellules des notebooks et voir quelle est la différence entre le mode édition et le mode commande. Nous allons ensuite voir pourquoi l'ordre d'exécution des cellules est important avant de conclure cette section en apprenant à arrêter correctement les notebooks.

Exécution de notebooks Jupyter

Dans Anaconda Prompt, accédez au répertoire où vous avez sauvegardé le code de ce livre, puis lancez un serveur notebook Jupyter :

```
(base)> cd C:\Utilisateurs\nom d'utilisateur\python-for-excel
(base)> jupyter notebook
```

Cela ouvre automatiquement votre navigateur et affiche le tableau de bord Jupyter avec les fichiers du répertoire à partir duquel vous avez exécuté la commande. En haut à droite du tableau de bord Jupyter, cliquez sur New, puis sélectionnez Python 3 dans la liste déroulante (Figure 2.2).

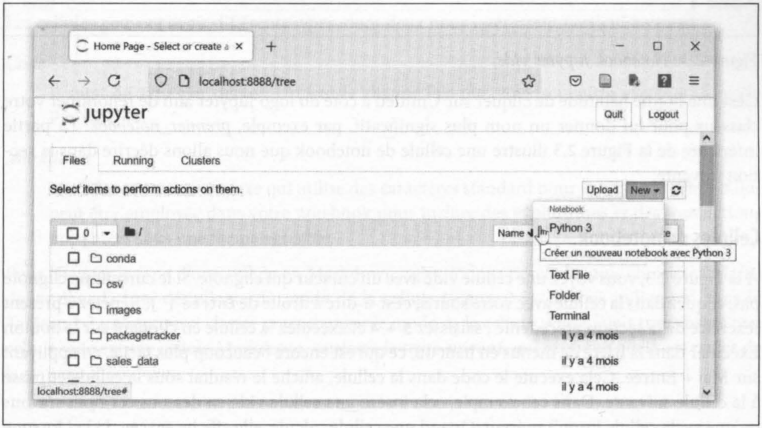


Figure 2.2 : Tableau de bord Jupyter.

Cela ouvre un nouvel onglet de navigateur avec votre premier notebook Jupyter vide, comme cela est illustré à la Figure 2.3.

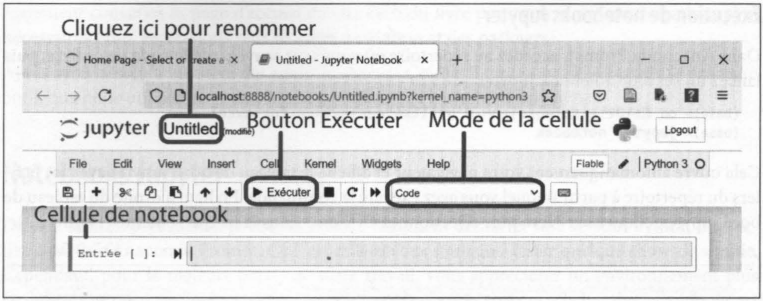


Figure 2.3 : Notebook Jupyter vide.

C'est une bonne habitude de cliquer sur Untitled à côté du logo Jupyter afin de renommer votre classeur pour lui donner un nom plus significatif, par exemple, *premier_notebook*. La partie inférieure de la Figure 2.3 illustre une cellule de notebook que nous allons décrire dans la section suivante.

Cellules de notebook

À la Figure 2.3, vous voyez une cellule vide avec un curseur qui clignote. Si le curseur ne clignote pas, cliquez dans la cellule avec votre souris, c'est-à-dire à droite de `Entrée [ ]`. Répétez à présent l'exercice de la section précédente : saisissez **`3 + 4`** et exécutez la cellule en cliquant sur le bouton Exécuter dans la barre de menus en haut ou, ce qui est encore beaucoup plus facile, en appuyant sur Maj + Entrée. Cela exécute le code dans la cellule, affiche le résultat sous la cellule et passe à la cellule suivante. Dans cet exemple, cela insère une cellule vide en dessous car nous n'avons qu'une seule cellule jusqu'à présent. Quand une cellule calcule, elle affiche `Entrée [*]` et lorsque le calcul est terminé, l'astérisque se transforme en un nombre, par exemple, `Entrée [1]`. Sous la cellule, vous avez la sortie correspondante qui comporte le même numéro : `Out[1]`. Chaque fois que vous exécutez une cellule, le compteur augmente d'une unité, ce qui vous aide à voir dans quel ordre les cellules ont été exécutées. À l'avenir, je montrerai les exemples de code dans ce format-là ; par exemple, le code REPL précédent ressemble à ceci :

```
Entrée [1]: 3 + 4
Out[1]: 7
```

Cette notation vous permet de tester facilement le code, par exemple en saisissant `3 + 4` dans une cellule de notebook. Lorsque vous l'exécutez en appuyant sur Maj + Entrée, vous obtenez ce que j'affiche comme sortie sous `Out[1]`. Vous remarquerez que la cellule d'entrée formate les chaînes, les nombres, *etc.* avec des couleurs différentes afin de faciliter la saisie et la lecture du code. C'est ce qu'on appelle la *coloration syntaxique*.



Si la dernière ligne d'une cellule renvoie une valeur, elle est automatiquement affichée par le notebook Jupyter sous `Out[ ]`. Toutefois, lorsque vous utilisez la fonction `print` ou lorsque vous obtenez une exception, elle est affichée directement sous la cellule `Entrée` sans l'étiquette `Out[ ]`. Les exemples de code de ce livre sont mis en forme pour refléter ce comportement.

Les cellules peuvent avoir différents types, dont deux nous intéressent particulièrement :

Code

Il s'agit du type par défaut. Utilisez-le chaque fois que vous souhaitez exécuter du code Python.

Markdown

Markdown est une syntaxe qui utilise des caractères standard pour la mise en forme et qui peut être employée dans votre notebook pour inclure des explications et des instructions présentées sous une forme agréable.

Pour changer le type d'une cellule en Markdown, sélectionnez la cellule, puis choisissez Markdown dans la liste déroulante du mode de la cellule (Figure 2.3). Je vous indiquerai dans le Tableau 2.3 un raccourci clavier pour changer le mode de la cellule. Après avoir transformé une cellule vide en une cellule Markdown, saisissez le texte suivant, qui explique quelques règles de la syntaxe Markdown :

```
# Il s'agit d'un titre de premier niveau

## Il s'agit d'un titre de deuxième niveau

Vous pouvez formater votre texte en *italique* ou en **gras**.

* Voici une puce
* Voici une autre puce
```

Après avoir appuyé sur Maj + Entrée, le texte sera affiché en HTML avec une belle mise en forme. À ce stade, votre notebook doit ressembler à la copie d'écran illustrée à la Figure 2.4. Les cellules Markdown permettent également d'inclure des images, des vidéos ou des formules[7].

7 Pour de plus amples informations sur le sujet, consultez la documentation des notebooks Jupyter qui est accessible à : https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Working%20With%20Markdown%20Cells.html#Markdown-Cells

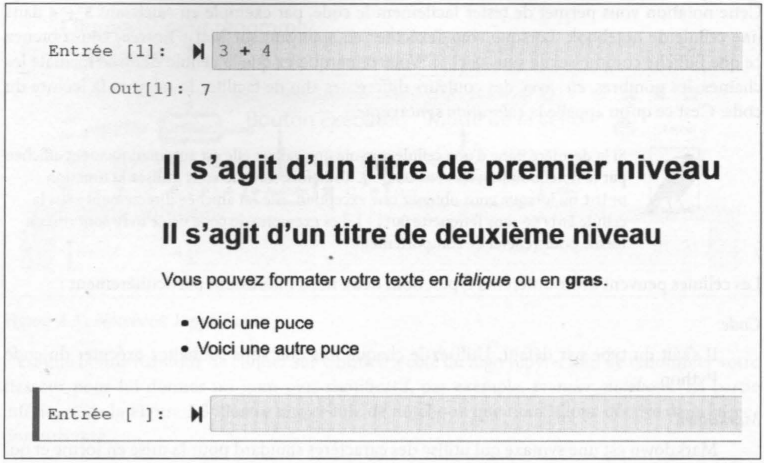


Figure 2.4 : Notebook après l'exécution d'une cellule de code et d'une cellule Markdown.

Maintenant que vous connaissez les types de cellules Code et Markdown, il est temps d'apprendre un moyen plus simple de naviguer entre les cellules : la section suivante présente les modes édition et commande ainsi que quelques raccourcis clavier.

Mode édition et mode commande

Lorsque vous travaillez avec des cellules d'un notebook Jupyter, vous êtes soit en *mode édition*, soit en *mode commande* :

Mode édition

Le fait de cliquer sur une cellule démarre le mode édition : la bordure autour de la cellule sélectionnée devient verte et le curseur de la cellule clignote. Au lieu de cliquer dans une cellule, vous pouvez également appuyer sur Entrée lorsque la cellule est sélectionnée.

Mode commande

Pour passer en mode commande, appuyez sur la touche Echap ; la bordure autour de la cellule sélectionnée devient bleue et il n'y aura pas de curseur qui clignote. Les raccourcis clavier les plus importants que vous pouvez utiliser en mode commande sont listés dans le Tableau 2.3.

Tableau 2.3 : Raccourcis clavier (mode commande).

Raccourci	Action
Maj + Entrée	Exécute la cellule (fonctionne également en mode édition).
↑ (flèche vers le haut)	Déplace le sélecteur de cellule vers le haut.
↓ (flèche vers le bas)	Déplace le sélecteur de cellule vers le bas.
b	Insère une nouvelle cellule *sous* la cellule active.
a	Insère une nouvelle cellule *au-dessus* de la cellule active.
dd	Supprime la cellule active (saisir deux fois la lettre d).
m	Change le type de cellule en Markdown.
y	Change le type de cellule en Code.

Connaître ces raccourcis clavier vous permettra de travailler efficacement avec les notebooks sans avoir à basculer entre le clavier et la souris tout le temps. Dans la section suivante, je vais vous montrer un problème courant dont vous devez être conscient lorsque vous utilisez des notebooks Jupyter : il est très important d'exécuter les cellules dans l'ordre.

L'ordre d'exécution des cellules est important

Les notebooks sont parfaits pour démarrer l'apprentissage de Python car ils sont simples et conviviaux, mais ils peuvent aussi facilement vous désorienter si vous n'exécutez pas les cellules séquentiellement. Supposons que vous ayez les cellules suivantes qui sont exécutées de haut en bas :

```
Entrée [2]: a = 1
Entrée [3]: a
Out[3]: 1
Entrée [4]: a = 2
```

La cellule `Out[3]` affiche la valeur `1` comme prévu. Toutefois, si vous revenez en arrière et exécutez à nouveau `Entrée [3]`, vous vous retrouverez dans cette situation :

```
Entrée [2]: a = 1
Entrée [5]: a
Out[5]: 2
Entrée [4]: a = 2
```

`Out[5]` affiche maintenant la valeur `2`, ce qui n'est probablement pas ce à quoi vous vous attendez lorsque vous lisez le notebook de haut en bas, surtout si la cellule `Entrée [4]` est plus éloignée, ce qui vous obligerait à faire défiler le contenu du notebook vers le bas. Pour éviter de tels problèmes, je vous recommande de ne pas vous contenter de réexécuter une seule cellule, mais d'exécuter aussi toutes ses cellules précédentes. Les notebooks Jupyter vous offrent un moyen facile de réaliser en choisissant la commande Run all above dans le menu Cell. Après cette mise en garde, voyons comment arrêter un notebook Jupyter correctement.

Notebooks Jupyter dans le cloud

Les notebooks Jupyter sont devenus si populaires qu'ils sont proposés en tant que solution hébergée par divers fournisseurs de cloud. Je présente ici trois services qui sont tous gratuits. L'avantage de ces services est qu'ils s'exécutent instantanément et à partir de n'importe quel endroit où vous pouvez accéder à un navigateur, sans avoir besoin d'installer quoi que ce soit en local. Vous pouvez, par exemple, exécuter les exemples des trois premières parties de cet ouvrage sur une tablette (comme la quatrième partie nécessite une installation locale d'Excel, cette solution ne fonctionnera pas).

Binder
: Binder est un service fourni par Project Jupyter, l'organisation à l'origine de la création des notebooks Jupyter. Binder est destiné à essayer les notebooks Jupyter à partir de dépôts Git publics : vous ne stockez rien sur Binder lui-même et vous n'avez donc pas besoin de vous inscrire ou de vous connecter pour l'utiliser.

Notebooks Kaggle
: Kaggle est une plateforme pour la science des données, qui héberge des concours, si bien que vous avez facilement accès à une énorme collection de jeux de données. Kaggle fait partie de Google depuis 2017.

Google Colab
: Google Colab (abréviation de Colaboratory) est la plateforme de Google pour les notebooks. Malheureusement, la majorité des raccourcis clavier des notebooks Jupyter ne fonctionnent pas, mais vous pouvez accéder aux fichiers stockés sur votre Google Drive, y compris Google Sheets.

Le moyen le plus simple d'exécuter les notebooks Jupyter des exemples de ce livre à partir du cloud consiste à pointer un navigateur vers leur URL Binder[8]. Vous travaillerez sur une copie des exemples, si bien que vous ne devez pas hésiter à faire des modifications, car vous ne risquez pas d'endommager quoi que ce soit.

Arrêt des notebooks Jupyter

Chaque notebook s'exécute dans un *noyau Jupyter* distinct. Un noyau est le « moteur » qui exécute le code Python que vous saisissez dans une cellule de notebook. Chaque noyau utilise les ressources de votre système d'exploitation sous forme de CPU et de RAM. Par conséquent, lorsque vous fermez un notebook, vous devez également arrêter son noyau afin que les ressources puissent être réutilisées par d'autres tâches, ce qui empêchera votre système de ralentir. Le moyen le plus simple d'y parvenir consiste à fermer un notebook *via* la commande File > Close and Halt.

8 https://mybinder.org/v2/gh/fzumstein/python-for-excel/1st-edition

Si vous fermez simplement l'onglet du navigateur, le noyau ne sera pas arrêté automatiquement. Vous pouvez également fermer les notebooks en cours d'exécution à partir de l'onglet Running.

Pour arrêter l'ensemble du serveur Jupyter, cliquez sur le bouton Quit en haut à droite du tableau de bord Jupyter. Si vous avez déjà fermé votre navigateur, vous pouvez saisir Ctrl + C deux fois dans Anaconda Prompt où le serveur de notebook est en cours d'exécution ou fermer complètement Anaconda Prompt.

Maintenant que nous savons comment travailler avec les notebooks Jupyter, passons à autre chose et apprenons à écrire et à exécuter des scripts Python standard. Pour ce faire, nous allons utiliser Visual Studio Code, un éditeur de texte puissant avec une excellente prise en charge de Python.

Visual Studio Code

Dans cette section, nous allons installer et configurer *Visual Studio Code* (VS Code), un éditeur de texte gratuit et open source de Microsoft. Après avoir présenté ses composants les plus importants, nous allons écrire un premier script Python et l'exécuter de différentes manières. Pour commencer, cependant, je vais expliquer quand nous utiliserons des notebooks Jupyter plutôt que d'exécuter des scripts Python et pourquoi j'ai choisi VS Code pour ce livre.

Bien que les notebooks Jupyter soient parfaits pour les flux de travail interactifs tels que la recherche, l'enseignement et l'expérimentation, ils sont moins adaptés si vous souhaitez écrire des scripts Python destinés à un environnement de production qui n'a pas besoin des capacités de visualisation des notebooks. En outre, les projets plus complexes qui impliquent de nombreux fichiers et plusieurs développeurs sont difficiles à gérer avec les notebooks Jupyter. Dans ce cas, il est préférable d'utiliser un éditeur de texte approprié pour écrire et exécuter des fichiers Python classiques. En théorie, vous pouvez utiliser à peu près n'importe quel éditeur de texte (même le Bloc-notes fonctionnerait), mais en réalité, il est préférable d'employer un éditeur qui « comprend » Python, c'est-à-dire un éditeur de texte qui prend en charge au moins les fonctionnalités suivantes :

Coloration syntaxique

L'éditeur colore les mots différemment selon qu'ils représentent une fonction, une chaîne, un nombre, *etc.* Cela facilite grandement la lecture et la compréhension du code.

Saisie semi-automatique

La saisie semi-automatique ou technologie *IntelliSense*, comme Microsoft l'appelle, suggère automatiquement des éléments de texte afin de diminuer la saisie, ce qui entraîne moins d'erreurs.

Et assez rapidement, d'autres besoins se feront ressentir et vous souhaiterez bénéficier des fonctionnalités suivantes depuis l'éditeur :

Exécution du code

Faire des allers-retours entre l'éditeur de texte et Anaconda Prompt (c'est-à-dire une invite de commande ou un terminal) pour exécuter votre code peut s'avérer fastidieux.

Débogueur

Un débogueur permet d'exécuter le code ligne par ligne pour voir ce qui se passe.

Contrôle de version

Si vous utilisez Git pour contrôler la version de vos fichiers, il est logique de gérer les éléments liés à Git directement dans l'éditeur afin de ne pas avoir à passer d'une application à l'autre.

Il existe un large éventail d'outils qui peuvent vous aider dans ce domaine, et comme toujours, chaque développeur a des besoins et des préférences différents. Certains peuvent en effet vouloir utiliser un éditeur de texte sans fioritures avec une invite de commande externe. D'autres peuvent préférer un *environnement de développement intégré* (IDE) : les IDE tentent de regrouper tout ce dont vous avez besoin dans un seul outil, ce qui peut les transformer en usine à gaz.

J'ai choisi VS Code pour ce livre, car il est rapidement devenu l'un des éditeurs de code les plus populaires parmi les développeurs après sa sortie initiale en 2015 : dans l'enquête effectuée par StackOverflow[9] en 2019, il est apparu comme l'environnement de développement le plus populaire. Qu'est-ce qui fait de VS Code un outil aussi populaire ? En substance, c'est le bon mélange entre un éditeur de texte dépouillé et un IDE complet : VS Code est un mini IDE qui est livré avec tout ce dont vous avez besoin pour programmer, mais pas plus :

Multiplateforme

VS Code tourne sous Windows, macOS, et Linux. Il existe également des versions hébergées dans le cloud comme GitHub Codespaces[10].

Outils intégrés

VS Code est fourni avec un débogueur, la prise en charge du contrôle de version Git et dispose d'un terminal intégré que vous pouvez utiliser comme Anaconda Prompt.

Extensions

D'autres fonctionnalités, par exemple la prise en charge de Python, peuvent être ajoutées *via* des extensions qui sont installées en un seul clic.

9 https://insights.stackoverflow.com/survey/2019
10 https://github.com/features/codespaces

Léger

Selon votre système d'exploitation, le programme d'installation de VS Code a une taille comprise entre 50 et 100 Mo.



Ne confondez pas Visual Studio Code avec l'IDE Visual Studio ! Bien que vous puissiez utiliser Visual Studio pour le développement Python (il est fourni avec PTVS, les outils Python pour Visual Studio), il s'agit d'une installation très lourde qui est traditionnellement utilisée pour fonctionner avec des langages .NET tels que C#.

Pour savoir si vous êtes d'accord avec tout le bien que je pense de VS Code, il n'y a pas de meilleur moyen que de l'installer et de le tester vous-même.

Installation et configuration

Téléchargez le programme d'installation à partir de la page d'accueil de VS Code[11]. Pour les dernières instructions relatives à l'installation, veuillez toujours vous référer à la documentation officielle.

Windows

Faites un double-clic sur le programme d'installation et acceptez toutes les valeurs par défaut. Ouvrez ensuite VS Code *via* le menu Démarrer de Windows, où vous le trouverez sous la dénomination Visual Studio Code.

macOS

Faites un double-clic sur le fichier ZIP pour décompresser l'application. Faites ensuite glisser *Visual Studio Code.app* dans le dossier *Applications* : vous pouvez à présent le démarrer à partir du Launchpad. Si l'application ne démarre pas, allez dans Préférences Système > Sécurité et confidentialité > Général et choisissez Ouvrir quand même.

Lorsque vous ouvrez VS Code pour la première fois, il ressemble à la copie d'écran illustrée à la Figure 2.5. Notez que je suis passé du thème sombre par défaut à un thème clair pour rendre les captures d'écran plus faciles à lire.

11 https://code.visualstudio.com/

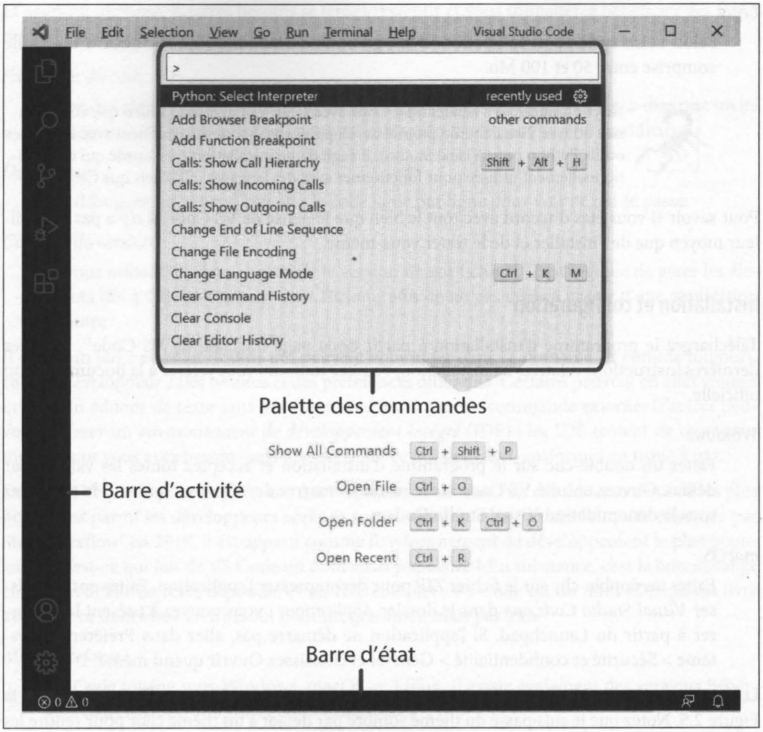


Figure 2.5 : Visual Studio Code.

Barre d'activité

Sur le côté gauche, vous voyez la barre d'activité avec les icônes suivantes de haut en bas :

- explorateur ;
- recherche ;
- contrôle de code source ;
- exécution ;
- extensions.

Barre d'état

En bas de l'éditeur, vous avez la barre d'état. Une fois la configuration terminée et quand vous éditerez un fichier Python, vous verrez le nom de l'interpréteur Python s'y afficher.

Palette des commandes

Vous pouvez afficher la palette des commandes en appuyant sur la touche F1 ou avec le raccourci clavier Ctrl + Maj + P (Windows) ou Commande + Maj + P (macOS). Si vous avez des doutes sur une fonctionnalité, votre premier réflexe devra être d'afficher la palette des commandes, car elle procure un accès facile à presque tout ce que vous pouvez faire avec VS Code. Par exemple, si vous recherchez des raccourcis clavier, saisissez **`shortcuts`**, et sélectionnez l'entrée Help : Keyboard Shortcuts Reference et appuyez sur Entrée.

VS Code est un excellent éditeur de texte prêt à l'emploi, mais pour le faire fonctionner correctement avec Python, il y a quelques éléments supplémentaires à configurer : cliquez sur l'icône Extensions dans la barre d'activité et recherchez Python. Installez l'extension Python officielle qui indique que Microsoft en est l'auteur. L'installation prendra un moment et quand elle sera terminée, vous devrez peut-être cliquer sur le bouton Reload Required (Rechargement nécessaire) pour terminer la configuration (vous pouvez également redémarrer VS Code). Finalisez la configuration en fonction de votre plateforme :

Windows

Ouvrez la palette des commandes et saisissez **`default profile`**. Sélectionnez l'entrée Terminal: Select Default Profile et appuyez sur Entrée. Dans le menu déroulant, sélectionnez Command Prompt et confirmez votre choix en appuyant sur Entrée. Ce paramètre est nécessaire, sinon VS Code ne peut pas activer correctement les environnements Conda.

macOS

Ouvrez la palette des commandes et saisissez **`shell command`**. Sélectionnez l'entrée Shell Command: Install 'code' command in PATH et appuyez sur Entrée. Cela est nécessaire pour que vous puissiez démarrer VS Code facilement à partir d'Anaconda Prompt (c'est-à-dire le terminal).

Maintenant que VS Code est installé et configuré, utilisons-le pour écrire et exécuter notre premier script Python !

Exécution d'un script Python

Bien que vous puissiez ouvrir VS Code *via* le menu Démarrer sous Windows ou le Launchpad sous macOS, il est souvent plus rapide d'ouvrir VS Code à partir d'Anaconda Prompt, où vous pouvez le lancer *via* la commande `code`. Par conséquent, ouvrez une nouvelle instance d'Anaconda Prompt et, à l'aide de la commande `cd`, positionnez-vous dans le répertoire où vous souhaitez travailler, puis demandez à VS Code d'ouvrir le répertoire actif (représenté par le point) :

```
(base)> cd C:\Utilisateurs\nom d'utilisateur\python-for-excel
(base)> code .
```

En démarrant VS Code de cette façon, l'Explorateur de la barre d'activité affiche automatiquement le contenu du répertoire dans lequel vous vous trouviez lorsque vous avez exécuté la commande `code`.

Vous pouvez également ouvrir un répertoire *via* File > Open Folder (sur macOS : File > Open), mais cela peut entraîner des erreurs d'autorisation sur macOS lorsque nous commencerons à utiliser xlwings dans la Partie IV. Lorsque vous pointez sur la liste des fichiers dans l'Explorateur de la barre d'activité, le bouton New File s'affiche, comme cela est illustré à la Figure 2.6. Cliquez sur ce bouton pour créer un nouveau fichier et nommez-le *hello_world.py*, puis appuyez sur Entrée. Une fois qu'il s'ouvre dans l'éditeur, écrivez la ligne de code suivante :

```
print("hello world!")
```

Est-ce que vous vous souvenez que les notebooks Jupyter affichent automatiquement la valeur de retour de la dernière ligne ? Lorsque vous exécutez un script Python traditionnel, vous devez indiquer explicitement à Python ce qu'il faut afficher, et c'est pourquoi vous devez utiliser la fonction `print` dans cet exemple. Dans la barre d'état, vous devriez maintenant voir votre version de Python, par exemple, « Python 3.8.5 64-bit (conda) ». Si vous cliquez dessus, la palette des commandes s'ouvrira et vous permettra de sélectionner un interpréteur Python différent si vous en avez plusieurs (notamment les environnements Conda). Votre configuration doit maintenant ressembler à celle illustrée à la Figure 2.6.

Avant de pouvoir exécuter le script, assurez-vous de l'enregistrer en appuyant sur Ctrl + S sous Windows ou Commande + S sous macOS. Avec les notebooks Jupyter, il suffirait de sélectionner une cellule et d'appuyer sur Maj + Entrée pour exécuter cette cellule. Avec VS Code, vous pouvez exécuter votre code à partir d'Anaconda Prompt ou en cliquant sur le bouton Exécuter. L'exécution de code Python à partir d'Anaconda Prompt est la façon dont vous exécuterez probablement les scripts qui se trouvent sur un serveur, et il est donc important de savoir comment cela fonctionne.

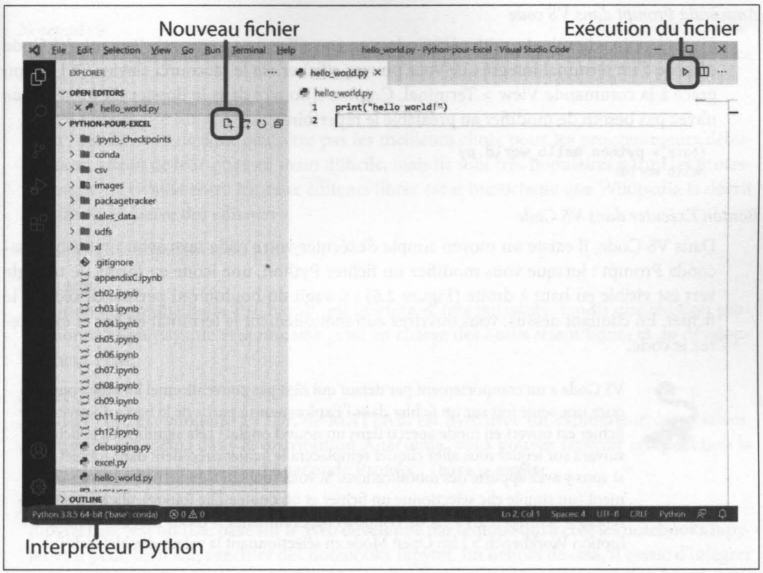


Figure 2.6 : VS Code avec le fichier hello_world.py ouvert.

Anaconda Prompt

Ouvrez Anaconda Prompt, et avec la commande `cd`, déplacez-vous dans le répertoire qui contient le script, puis exécutez-le script de la manière suivante :

```
(base)> cd C:\Utilisateurs\nom d'utilisateur\python-for-excel
(base)> python hello_world.py
hello world!
```

La dernière ligne est la sortie qui est affichée par le script. Notez que si vous n'êtes pas dans le même répertoire que votre fichier Python, vous devez utiliser le chemin d'accès complet à votre fichier Python :

```
(base)> python C:\Utilisateurs\nom d'utilisateur\python-for-excel\hello_world.py
hello world!
```



Afin de gérer les longs chemins de fichiers, il est pratique de faire glisser et de déposer le fichier sur Anaconda Prompt. Cela écrira le chemin d'accès complet, quel que soit l'emplacement du curseur.

Anaconda Prompt dans VS code

Vous n'avez pas besoin de quitter VS Code pour travailler avec Anaconda Prompt : VS Code dispose d'un terminal intégré que vous pouvez afficher *via* le raccourci clavier Ctrl + ù ou grâce à la commande View > Terminal. Comme il s'ouvre dans le dossier du projet, vous n'avez pas besoin de modifier au préalable le répertoire :

```
(base)> python hello_world.py
hello world!
```

Bouton Exécuter dans VS Code

Dans VS Code, il existe un moyen simple d'exécuter votre code sans avoir à utiliser Anaconda Prompt : lorsque vous modifiez un fichier Python, une icône en forme de triangle vert est visible en haut à droite (Figure 2.6) ; il s'agit du bouton qui permet d'exécuter le fichier. En cliquant dessus, vous ouvrirez automatiquement le terminal en bas et exécuterez le code.



VS Code a un comportement par défaut qui n'est pas conventionnel lorsque vous cliquez une seule fois sur un fichier dans l'Explorateur (à partir de la barre d'activité) : le fichier est ouvert en mode aperçu dans un nouvel onglet ; cela signifie que le fichier suivant sur lequel vous allez cliquer remplacera le fichier précédent dans l'onglet, sauf si vous y avez apporté des modifications. Si vous souhaitez désactiver ce comportement (un simple clic sélectionne un fichier et un double-clic l'ouvre), allez dans Preferences > Settings (Ctrl + , sous Windows ou Commande + , sous macOS) et modifiez l'option Workbench > List: Open Mode en sélectionnant la valeur doubleClick[12].

À ce stade, vous savez comment créer, modifier et exécuter des scripts Python dans VS Code. VS Code peut cependant en faire beaucoup plus et j'explique dans l'Annexe B comment utiliser le débogueur et exécuter des notebooks Jupyter.

Autres éditeurs de textes et IDE

Les outils sont quelque chose de très personnel, et ce n'est pas parce que ce livre est basé sur les notebooks Jupyter et VS Code que vous devez vous interdire de jeter un coup d'œil à d'autres produits.

Parmi les éditeurs de texte populaires, on peut citer :

Sublime Text
Sublime[13] est un éditeur de texte commercial rapide.

12 *NdT* : Comme il y a de nombreux paramètres dans VS Code, il est plus efficace, afin de restreindre leur liste, d'effectuer une sélection dans la liste globale des paramètres en saisissant dans la zone de recherche située en haut de la fenêtre les termes « workbench open mode ».

13 https://www.sublimetext.com/

Notepad++

Notepad++[14] est gratuit et existe depuis très longtemps, mais il fonctionne uniquement sous Windows.

Vim ou Emacs

Vim[15] ou Emacs[16] ne sont peut-être pas les meilleurs choix pour les programmeurs débutants en raison de leur prise en main difficile, mais ils sont très populaires parmi les professionnels. La rivalité entre les deux éditeurs libres est si importante que Wikipédia la décrit comme la « guerre des éditeurs ».

Parmi les IDE populaires, on peut citer :

PyCharm

L'édition communautaire PyCharm[17] est gratuite et très puissante, tandis que l'édition professionnelle est payante et ajoute une prise en charge des outils scientifiques et du développement Web.

Spyder

Spyder[18], qui est similaire à l'IDE de MATLAB, est livré avec un explorateur de variables. Comme il est inclus dans la distribution Anaconda, vous pouvez l'essayer en exécutant la commande suivante à partir d'Anaconda Prompt : `(base)> spyder`.

JupyterLab

JupyterLab[19] est un IDE basé sur le Web développé par l'équipe qui a créé les notebooks Jupyter et il peut, bien sûr, exécuter des notebooks Jupyter. En dehors de cela, il essaie d'intégrer en un seul outil tout ce dont vous avez besoin pour vos activités en science des données.

Wing Python IDE

Wing Python IDE[20] est un IDE qui existe depuis longtemps. Il y a des versions simplifiées gratuites et une version commerciale appelée Wing Pro.

Komodo IDE

Komodo IDE[21], qui est un IDE commercial développé par ActiveState, prend en charge de nombreux autres langages en dehors de Python.

PyDev

PyDev[22] est un IDE Python basé sur l'IDE populaire Eclipse.

14 https://notepad-plus-plus.org/
15 https://www.vim.org/
16 https://www.gnu.org/software/emacs/
17 https://www.jetbrains.com/pycharm/
18 https://www.spyder-ide.org/
19 https://jupyter.org/
20 https://wingware.com/
21 https://www.activestate.com/products/komodo-ide/
22 https://www.pydev.org/

Conclusion

Dans ce chapitre, je vous ai montré comment installer et utiliser les outils avec lesquels nous allons travailler : Anaconda Prompt, les notebooks Jupyter et VS Code. Nous avons également exécuté un tout petit bout de code Python dans un REPL Python, dans un notebook Jupyter et en tant que script dans VS Code.

Je vous recommande de vous familiariser avec Anaconda Prompt, car cette application vous rendra très efficace une fois que vous vous y serez habitué. La possibilité de travailler avec des notebooks Jupyter dans le cloud est également très confortable, car elle permet d'exécuter les exemples de code des trois premières parties de ce livre à partir d'un navigateur.

Grâce à votre environnement de développement fonctionnel, vous êtes maintenant prêt à aborder le chapitre suivant, où vous apprendrez suffisamment de Python pour pouvoir suivre le reste de cet ouvrage.

CHAPITRE 3

Prise en main de Python

Avec Anaconda installé et les notebooks Jupyter opérationnels, vous avez tout ce qu'il vous faut pour commencer à utiliser Python. Bien que ce chapitre n'aille pas beaucoup plus loin que les bases, il couvre quand même de nombreuses notions. Si vous êtes un codeur débutant, il se peut que vous ayez beaucoup de choses à assimiler. Cependant, la plupart des concepts seront plus clairs une fois que vous les utiliserez dans les chapitres suivants, sous la forme d'exemples pratiques, et il n'y a donc pas lieu de s'inquiéter si vous ne comprenez pas tout, tout de suite. Chaque fois que Python et VBA diffèrent de manière significative, je le soulignerai afin de m'assurer que vous pouvez passer de VBA à Python en douceur et que vous êtes conscient des pièges les plus flagrants. Si vous n'avez jamais codé en VBA auparavant, n'hésitez pas à ignorer ces parties.

Je vais commencer ce chapitre avec les types de données de base de Python, tels que les entiers et les chaînes. Après cela, je présenterai l'indexation et le découpage, un concept de base en Python qui donne accès à des éléments spécifiques d'une séquence. Viennent ensuite les structures de données telles que les listes et les dictionnaires qui peuvent contenir plusieurs objets. Je continuerai avec l'instruction `if` et les boucles `for` et `while` avant d'introduire les concepts de fonctions et de modules qui permettent d'organiser et de structurer le code. Pour conclure ce chapitre, je vous montrerai comment formater correctement votre code Python. Comme vous l'avez probablement deviné maintenant, ce chapitre est très technique ; dans ces conditions, l'exécution des exemples de code dans un notebook Jupyter est une bonne idée afin de rendre votre apprentissage plus interactif et plus ludique. Saisissez les exemples vous-même ou exécutez-les à l'aide des notebooks qui accompagnent ce livre.

Types de données

Python, comme tous les autres langages de programmation, traite les nombres, le texte, les booléens, *etc.* différemment en leur attribuant un *type de données* différent. Les types de données que nous utiliserons le plus souvent sont les entiers, les flottants, les booléens et les chaînes. Dans

cette section, je vais les présenter l'un après l'autre en proposant quelques exemples. Cependant, afin de pouvoir comprendre les types de données, je dois d'abord expliquer ce qu'est un objet.

Objets

En Python, *tout* est objet, y compris les nombres, les chaînes, les fonctions et tout ce que nous rencontrerons dans ce chapitre. Les objets peuvent rendre les choses complexes faciles et intuitives en donnant accès à un ensemble de variables et de fonctions. Avant toute chose, permettez-moi donc de dire quelques mots sur les variables et les fonctions.

Variables

En Python, une *variable* est un nom que vous attribuez à un objet à l'aide du signe égal. Dans la première ligne de l'exemple suivant[1], le nom `a` est assigné à l'objet `3` :

```
In [1]: a = 3
        b = 4
        a + b
Out[1]: 7
```

Cela fonctionne de la même manière pour tous les objets, ce qui est plus simple qu'en VBA, où l'on utilise le signe égal pour les types de données tels que les nombres et les chaînes et l'instruction `Set` pour les objets tels que les classeurs ou les feuilles de calcul. En Python, on modifie le type d'une variable simplement en l'assignant à un nouvel objet. C'est ce qu'on appelle le *typage dynamique* :

```
In [2]: a = 3
        print(a)
        a = "trois"
        print(a)
3
trois
```

Contrairement à VBA, Python fait la différence entre les majuscules et les minuscules, si bien que `a` et `A` sont deux variables différentes. Les noms de variables doivent suivre certaines règles :

- Ils doivent commencer par une lettre ou une barre de soulignement.
- Ils doivent être composés de lettres, de chiffres et de barres de soulignement.

Après cette courte introduction aux variables, voyons comment nous pouvons effectuer des appels de fonction.

1 À partir de maintenant, dans les exemples de code extraits des notebooks Jupyter, nous conservons la terminologie anglaise « In » à la place de la traduction française « Entrée ».

Fonctions

Je présenterai les fonctions de manière plus détaillée plus loin dans ce chapitre. Pour l'instant, vous devez simplement savoir comment appeler des fonctions intégrées telles que `print` que nous avons utilisée dans l'exemple de code précédent. Pour appeler une fonction, vous ajoutez des parenthèses au nom de la fonction et fournissez les arguments entre parenthèses, ce qui est à peu près équivalent à la notation mathématique :

```
nom_de_fonction(argument1, argument2, …)
```

Voyons maintenant comment les variables et les fonctions opèrent dans le contexte des objets.

Attributs et méthodes

Dans le contexte des objets, les variables sont appelées *attributs* et les fonctions sont appelées *méthodes* : les attributs donnent accès aux données d'un objet, et les méthodes permettent d'effectuer une action. Pour accéder aux attributs et aux méthodes, on utilise la notation pointée de la manière suivante : `mon_object.attribut` et `mon_object.méthode()`.

Rendons cela un peu plus concret : si vous écrivez un jeu de course automobile, vous utiliserez probablement un objet qui représente une voiture. L'objet `voiture` pourrait avoir un attribut vitesse qui permet d'obtenir la vitesse actuelle *via* `voiture.vitesse`, et vous pourriez être en mesure d'accélérer la voiture en appelant la méthode permettant d'accélérer, `voiture.accelerer(10)`, ce qui augmenterait la vitesse de dix kilomètres par heure.

Le type d'un objet, et par conséquent son comportement, est défini par une *classe*. Ainsi, l'exemple précédent vous obligerait à écrire une classe `Voiture`. Le processus d'obtention d'un objet `voiture` à partir d'une classe `Voiture` est appelé *instanciation*, et on instancie un objet en appelant la classe de la même manière que l'on appelle une fonction : `voiture = Voiture()`. Nous n'écrirons pas nos propres classes dans ce livre, mais si vous êtes intéressé par la façon dont cela fonctionne, jetez un œil à l'Annexe C.

Nous utiliserons une première méthode d'objet dans la section suivante pour créer une chaîne de caractères en majuscules, et nous reviendrons sur les objets et les classes lorsque nous étudierons les objets `datetime` vers la fin de ce chapitre. Nous allons à présent évoquer les objets qui ont un type de données numérique.

Types numériques

Les types de données `int` et `float` représentent respectivement des *entiers* et des *nombres à virgule flottante*. Pour connaître le type de données d'un objet, utilisez la fonction intégrée type :

```
In [3]: type(4)
Out[3]: int
In [4]: type(4.4)
Out[4]: float
```

Si vous souhaitez qu'un nombre soit un `float` au lieu d'un `int`, il suffit d'utiliser le point décimal à la fin ou le constructeur `float` :

```
In [5]: type(4.)
Out[5]: float
In [6]: float(4)
Out[6]: 4.0
```

Le dernier exemple peut également être inversé : à l'aide du constructeur `int`, vous pouvez transformer un `float` en `int`. Si la partie fractionnaire n'est pas nulle, elle sera tronquée :

```
In [7]: int(4.9)
Out[7]: 4
```



Vous devrez peut-être convertir un `float` en `int` lorsque vous lisez un nombre à partir d'une cellule Excel afin de le passer en tant qu'argument à une fonction Python qui attend un entier. Cela est dû au fait que les nombres dans les cellules Excel sont toujours stockés sous forme de nombre à virgule flottante en arrière-plan, même si Excel affiche ce qui ressemble à un entier.

Python a quelques autres types numériques que je n'utiliserai pas ou dont je ne discuterai pas dans ce livre : il y a les types de données `decimal`, `fraction` et `complex`. Si les inexactitudes des nombres à virgule flottante sont un problème (voir l'encadré), utilisez le type `decimal` pour obtenir des résultats exacts. Ces cas sont cependant très rares. En règle générale, si Excel convient pour les calculs, utilisez des nombres à virgule flottante.

Inexactitudes des nombres à virgule flottante

Par défaut, Excel affiche souvent des nombres arrondis : saisissez `= 1.125-1.1` dans une cellule, et vous verrez `0.025`. Bien que cela puisse être ce à quoi vous vous attendez, ce n'est pas ce qu'Excel stocke en interne. Modifiez le format d'affichage pour afficher au moins 16 décimales, et il passera à `0.0249999999999999`. C'est l'effet de l'imprécision des nombres à virgule flottante : les ordinateurs vivent dans un monde binaire, ce qui signifie qu'ils ne calculent qu'avec des zéros et des un. Certaines fractions décimales comme `0.1` ne peuvent pas être stockées sous la forme d'un nombre binaire fini à virgule flottante, ce qui explique le résultat de la soustraction. En Python, vous constaterez le même effet, mais Python ne cache pas les décimales :

```
In [8]: 1.125 - 1.1
Out[8]: 0.02499999999999991
```

Opérateurs mathématiques

Le calcul avec des nombres nécessite l'utilisation d'opérateurs mathématiques tels que le signe plus ou moins. À l'exception de l'opérateur qui permet d'élever à la puissance, il ne devrait pas y avoir de surprise si vous avez l'habitude d'Excel :

```
In [9]: 3 + 4  # Somme
Out[9]: 7
In [10]: 3 - 4  # Soustraction
Out[10]: -1
In [11]: 3 / 4  # Division
Out[11]: 0.75
In [12]: 3 * 4  # Multiplication
Out[12]: 12
In [13]: 3**4  # Opérateur pour élever à la puissance (Excel utilise 3^4)
Out[13]: 81
In [14]: 3 * (3 + 4)  # Utilisation de parenthèses
Out[14]: 21
```

Commentaires

Dans les exemples précédents, j'ai décrit le fonctionnement de certains exemples en utilisant des *commentaires* (par exemple, `# Somme`). Les commentaires aident d'autres personnes (et vous-même quelques semaines après avoir écrit le code) à comprendre ce qui se passe dans votre programme. Il est recommandé de ne commenter que les choses qui ne sont pas évidentes à la lecture du code : en cas de doute, il est préférable de ne pas avoir de commentaire plutôt qu'un commentaire obsolète qui contredise le code. Tout ce qui commence par un signe dièse est considéré comme un commentaire en Python et le texte qui suit # est ignoré lorsque vous exécutez le code :

```
In [15]: # Voici un exemple que nous avons déjà vu.
         # Chaque ligne de commentaire commence par le signe #
         3 + 4
Out[15]: 7
In [16]: 3 + 4  # Voici un commentaire en ligne
Out[16]: 7
```

La plupart des éditeurs ont un raccourci clavier pour commenter et supprimer les marques de commentaire. Dans les notebooks Jupyter et VS Code, il s'agit de Ctrl + / (Windows) ou Commande + / (macOS). Notez que les cellules Markdown dans les notebooks Jupyter n'acceptent pas les commentaires : si vous commencez une ligne avec un #, Markdown l'interprétera comme un titre de premier niveau.

Booléens

Les types booléens en Python sont `True` ou `False`, exactement comme en VBA. Les opérateurs booléens `and`, `or` et `not`, cependant, sont tous en minuscules, tandis que VBA les affiche avec l'initiale en majuscule. Les expressions booléennes fonctionnent de la même manière que dans Excel, à l'exception des opérateurs d'égalité et d'inégalité :

```
In [17]: 3 == 4  # Égalité (Excel utilise 3 = 4)
Out[17]: False
In [18]: 3 != 4  # Inégalité (Excel utilise 3 <> 4)
Out[18]: True
In [19]: 3 < 4  # Inférieur à. Utilisez > pour supérieur à.
Out[19]: True
In [20]: 3 <= 4  # Inférieur ou égal à. Utilisez >= pour supérieur ou égal à.
Out[20]: True
In [21]: # Vous pouvez chaîner les expressions logiques
         # En VBA, on aurait : 10 < 12 And 12 < 17
         # Dans une formule Excel, on aurait : =ET(10<12;12<17)
         10 < 12 < 17
Out[21]: True
In [22]: not True  # opérateur "not"
Out[22]: False
In [23]: False and True  # opérateur "and"
Out[23]: False
In [24]: False or True  # opérateur "or"
Out[24]: True
```

Chaque objet Python a la valeur `True` ou `False`. La majorité des objets sont égaux à `True`, mais il y en a qui sont évalués à `False`, notamment `None` (voir l'encadré), à `0` ou à des types de données vides, comme une chaîne vide (les chaînes sont étudiées à la section suivante).

None

`None` est une constante intégrée qui représente « l'absence de valeur » selon la documentation officielle. Par exemple, si une fonction ne retourne aucune valeur, elle retourne `None`. C'est également un bon choix pour représenter des cellules vides dans Excel comme nous le verrons dans la Partie III et la Partie IV.

Pour vérifier si un objet a la valeur `True` ou `False`, utilisez le constructeur `bool` :

```
In [25]: bool(2)
Out[25]: True
In [26]: bool(0)
Out[26]: False
In [27]: bool("du texte")  # On va aborder les chaînes dans un instant
Out[27]: True
```

```
In [28]: bool("")
Out[28]: False
In [29]: bool(None)
Out[29]: False
```

Il nous reste un autre type de données de base à aborder : les données textuelles, que l'on désigne sous le nom de *chaînes*.

Chaînes

Si vous avez déjà travaillé en VBA avec des chaînes[2] faisant plus d'une ligne et contenant des variables et des guillemets, vous avez probablement éprouvé quelques difficultés. Heureusement, c'est un domaine où Python est particulièrement doué. Les chaînes peuvent être exprimées à l'aide de guillemets doubles (") ou de guillemets simples ('), la seule condition étant que vous devez commencer et terminer la chaîne avec le même type de guillemets. Vous pouvez utiliser le signe + pour concaténer des chaînes ou * pour répéter des chaînes. Puisque je vous ai déjà montré un exemple de répétition lors du test du REPL Python dans le chapitre précédent, voici un exemple utilisant le signe plus :

```
In [30]: "Une chaîne avec des guillemets doubles. " + 'Une chaîne avec des guillemets simples.'
Out[30]: 'Une chaîne avec des guillemets doubles. Une chaîne avec des guillemets simples.'
```

Selon ce que vous souhaitez écrire, l'utilisation de guillemets simples ou doubles peut vous aider à afficher facilement des guillemets littéraux sans avoir besoin d'employer un caractère d'échappement. Si vous avez quand même besoin d'échapper un caractère, il faut le faire précéder d'une barre oblique inverse :

```
In [31]: print("N'attendez pas ! " + 'Apprenez à "parler" Python.')
Out[31]: N'attendez pas ! Apprenez à "parler" Python.
In [32]: print("Il est facile d'\"échapper\" des caractères avec un \\ en en-tête.")
Out[32]: Il est facile d'"échapper" des caractères avec un \ en en-tête.
```

Lorsque l'on mélange des chaînes avec des variables, on travaille généralement avec des *f-strings*, abréviation de *formatted string literal*, c'est-à-dire littéral de chaîne mis en forme. Il suffit de mettre un `f` devant votre chaîne et d'utiliser des variables entre accolades :

```
In [33]: # Notez comment Python permet facilement d'assigner
         # plusieurs valeurs à des variables sur une même ligne
         premier_adjectif, second_adjectif = "libre", "open source"
         f"Python est {premier_adjectif} et {second_adjectif}."
Out[33]: 'Python est libre et open source.'
```

Comme je l'ai mentionné au début de cette section, les chaînes sont des objets comme tout le reste, et elles offrent quelques méthodes (c'est-à-dire des fonctions) pour effectuer des actions

2 En VBA, comme en Python, on préfère souvent employer le terme anglais *string* (au pluriel *strings*) pour désigner les chaînes de caractères. En informatique, quand on parle de chaîne, il s'agit d'une abréviation pour chaîne de caractères.

sur les chaînes. Par exemple, voici comment inverser la casse de lettres en majuscules et en minuscules :

```
In [34]: "PYTHON".lower()
Out[34]: 'python'
In [35]: "python".upper()
Out[35]: 'PYTHON'
```

Obtenir de l'aide

Comment savoir quels attributs possèdent certains objets comme les chaînes de caractères et quels arguments acceptent leurs méthodes ? La réponse dépend un peu de l'outil que vous utilisez : avec les notebooks Jupyter, appuyez sur la touche Tab après avoir saisi le point qui suit un objet, par exemple `"python".<Tab>`. Cela fera apparaître une liste déroulante avec tous les attributs et méthodes que propose cet objet. Si votre curseur se trouve à l'intérieur d'une méthode, par exemple entre les parenthèses de `"python".upper()`, appuyez sur Maj + Tab pour obtenir la description de cette fonction. VS Code affiche automatiquement ces informations sous forme d'info-bulle. Si vous exécutez un REPL Python dans Anaconda Prompt, utilisez `dir("python")` pour obtenir les attributs disponibles et `help("python".upper)` pour afficher la description de la méthode `upper`. À part cela, c'est toujours une bonne idée de vous reporter à la documentation en ligne de Python[3]. Si vous recherchez la documentation de packages tiers comme pandas, il est préférable de la rechercher sur PyPI[4], l'index des packages de Python, où vous trouverez les liens vers la page d'accueil et la documentation des packages en question.

Lorsque vous travaillez avec des chaînes, il est courant d'avoir à sélectionner des parties d'une chaîne : par exemple, vous pouvez vouloir extraire la partie USD de la notation du taux de change EURUSD. La section suivante vous montre le puissant mécanisme d'indexation et de découpage de Python qui permet de faire exactement cela.

Indexation et découpage

L'indexation et le découpage[5] donnent accès à des éléments spécifiques d'une séquence. Étant donné que les chaînes sont des séquences de caractères, nous pouvons les utiliser pour apprendre ce mécanisme. Dans la section suivante, nous étudierons d'autres séquences telles que les listes et les tuples qui prennent également en charge l'indexation et le découpage.

3 https://docs.python.org/fr/3/

4 https://pypi.org/

5 *NdT* : De nombreuses sources en français conservent le terme anglais slicing, qui signifie littéralement découpage en tranches, pour désigner cette opération.

Indexation

La Figure 3.1 présente le concept d'indexation. Python se base sur le zéro, ce qui signifie que le premier élément d'une séquence est désigné par l'indice 0. Les indices négatifs à partir de -1 permettent de faire référence aux éléments à partir de la fin de la séquence.

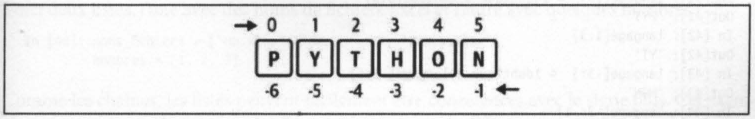


Figure 3.1 : Indices à partir du début et de la fin d'une séquence.



Si vous programmez en VBA, l'indexation est une source d'erreur fréquente. VBA utilise des indices débutant à un pour la plupart des collections comme les feuilles de calcul (`Feuil(1)`) mais utilise des indices à partir de zéro pour les tableaux (`MyArray(0)`), bien que cette valeur par défaut puisse être modifiée. Il y a également une autre différence, car VBA emploie des parenthèses pour l'indexation tandis que Python utilise des crochets.

La syntaxe de l'indexation est la suivante :

```
séquence[indice]
```

En conséquence, on accède à des éléments spécifiques d'une chaîne de la manière suivante :

```
In [36]: langage = "PYTHON"
In [37]: langage[0]
Out[37]: 'P'
In [38]: langage[1]
Out[38]: 'Y'
In [39]: langage[-1]
Out[39]: 'N'
In [40]: langage[-2]
Out[40]: 'O'
```

Quand vous voudrez extraire plusieurs caractères, c'est à ce moment-là que vous utiliserez le découpage.

Découpage

Quand on souhaite obtenir plusieurs éléments d'une séquence, on emploie la syntaxe de *découpage*, qui fonctionne de la manière suivante :

```
séquence[début:arrêt:pas]
```

Python utilise des intervalles semi-ouverts : l'indice de début est inclus alors que l'indice d'arrêt ne l'est pas. Si vous omettez les arguments `début` ou `arrêt`, Python inclut respectivement toute la séquence, à partir du début ou de la fin. L'argument `pas` détermine la direction et la taille du pas : par exemple, 2 retournera un élément sur deux en partant de la gauche et -3 retournera un élément sur trois en partant de la droite. Par défaut, la taille du pas est fixée à un :

```
In [41]: langage[:3]  # Identique à langage[0:3]
Out[41]: 'PYT'
In [42]: langage[1:3]
Out[42]: 'YT'
In [43]: langage[-3:]  # Identique à langage[-3:6]
Out[43]: 'HON'
In [44]: langage[-3:-1]
Out[44]: 'HO'
In [45]: langage[::2]  # Un élément sur deux en partant de la gauche
Out[45]: 'PTO'
In [46]: langage[-1:-4:-1]  # Un pas négatif démarre en partant de la droite
Out[46]: 'NOH'
```

Jusqu'à présent, nous n'avons examiné qu'une seule opération d'indexation ou de découpage à la fois, mais Python permet également d'*enchaîner* plusieurs opérations. Par exemple, si vous souhaitez obtenir le deuxième caractère parmi les trois derniers, vous pouvez procéder de la manière suivante :

```
In [47]: langage[-3:][1]
Out[47]: 'O'
```

Cette opération est identique à `langage[-2]`, si bien que dans ce cas il ne serait pas très logique d'utiliser le chaînage, mais cela aura plus de sens lorsque nous utiliserons l'indexation et le découpage avec des listes, structure de données que je vais présenter dans la section suivante.

Structures de données

Python offre des structures de données puissantes qui facilitent l'utilisation d'une collection d'objets. Dans cette section, je vais présenter les listes, les dictionnaires, les tuples et les ensembles. Bien que chacune de ces structures de données ait des caractéristiques légèrement différentes, elles sont toutes capables de stocker plusieurs objets. En VBA, vous avez peut-être déjà utilisé des collections ou des tableaux pour stocker plusieurs valeurs. VBA offre même une structure de données appelée dictionnaire qui fonctionne conceptuellement de la même manière que les dictionnaires de Python. Les dictionnaires ne sont toutefois disponibles que dans la version Windows d'Excel. Commençons par les listes, qui sont la structure de données que vous utiliserez probablement le plus souvent.

Listes

Les *listes* peuvent contenir plusieurs objets ayant des types de données différents. Elles sont si polyvalentes que vous les utiliserez tout le temps. On crée une liste de la manière suivante :

```
[élément1, élément2, ...]
```

Voici deux listes, l'une avec des noms de fichiers Excel et l'autre avec quelques nombres :

```
In [48]: noms_fichiers = ["un.xlsx", "deux.xlsx", "trois.xlsx"]
         nombres = [1, 2, 3]
```

Comme les chaînes, les listes peuvent facilement être concaténées avec le signe plus. Cet exemple vous montre également que les listes peuvent contenir différents types d'objets :

```
In [49]: noms_fichiers + nombres
Out[49]: ['un.xlsx', 'deux.xlsx', 'trois.xlsx', 1, 2, 3]
```

Dans la mesure où les listes sont des objets, comme pratiquement tout en Python, elles peuvent également avoir d'autres listes comme éléments, que l'on appellera *listes imbriquées :*

```
In [50]: liste_imbriquee = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Si vous réorganisez ces données pour qu'elles s'étendent sur plusieurs lignes, vous vous rendrez compte qu'il s'agit d'une très belle représentation d'une matrice ou d'une plage de cellules d'une feuille de calcul. Notez que les crochets permettent implicitement de décomposer les lignes de code (voir l'encadré). Grâce à l'indexation et au découpage, on obtient les éléments que l'on souhaite :

```
In [51]: cells = [[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]
In [52]: cells[1]  # Deuxième ligne
Out[52]: [4, 5, 6]
In [53]: cells[1][1:]  # Deuxième ligne, deuxième et troisième colonnes
Out[53]: [5, 6]
```

Vous pouvez modifier des éléments d'une liste :

```
In [56]: utilisateurs = ["Linda", "Brian"]
In [57]: utilisateurs.append("Jennifer")  # Ajoute à la fin
         utilisateurs
Out[57]: ['Linda', 'Brian', 'Jennifer']
In [58]: utilisateurs.insert(0, "Kim")  # Insère "Kim" à l'indice 0
         utilisateurs
Out[58]: ['Kim', 'Linda', 'Brian', 'Jennifer']
```

Continuation de ligne

Il arrive parfois qu'une ligne de code soit si longue que l'on soit obligé de la diviser en deux lignes, voire plus, pour que le code reste lisible. Techniquement, vous pouvez utiliser des parenthèses ou une barre oblique inverse pour diviser la ligne :

```
In [54]: a = (1 + 2
              + 3)
In [55]: a = 1 + 2 \
           + 3
```

Cependant, le guide de style de Python préfère utiliser si possible des *sauts de ligne implicites* : chaque fois que vous utilisez une expression qui contient des parenthèses, des crochets ou des accolades, utilisez-les pour introduire un saut de ligne sans avoir à introduire de caractère supplémentaire. J'évoquerai de manière plus détaillée le guide de style de Python à la fin de ce chapitre.

Pour supprimer un élément, utilisez `pop` ou `del`. Alors que `pop` est une méthode, `del` est implémentée en tant qu'instruction en Python :

```
In [59]: utilisateurs.pop()  # Supprime et renvoie le dernier élément par défaut
Out[59]: 'Jennifer'
In [60]: utilisateurs
Out[60]: ['Kim', 'Linda', 'Brian']
In [61]: del utilisateurs[0]  # del supprime un élément dont l'indice est donné
```

Voici d'autres choses utiles que vous pouvez faire avec les listes :

```
In [62]: len(utilisateurs)  # longueur de la liste
Out[62]: 2
In [63]: "Linda" in utilisateurs  # Vérifie si utilisateurs contient "Linda"
Out[63]: True
In [64]: print(sorted(utilisateurs))  # Retourne une nouvelle liste triée
         print(utilisateurs)  # La liste originale n'est pas modifiée
Out[64]: ['Brian', 'Linda']
['Linda', 'Brian']
In [65]: utilisateurs.sort()  # Trie la liste originale
utilisateurs
Out[65]: ['Brian', 'Linda']
```

Notez que vous pouvez également utiliser `len` et `in` avec des chaînes :

```
In [66]: len("Python")
Out[66]: 6
In [67]: "libre" in "Python est libre et open source."
Out[67]: True
```

Pour accéder aux éléments d'une liste, on fait référence à leur position ou leur indice, ce qui n'est pas toujours pratique. Les dictionnaires (voir la section suivante) permettent d'accéder à des éléments *via* une clé, qui est souvent un nom.

Dictionnaires

Les *dictionnaires* associent des clés à des valeurs. On rencontre des combinaisons clé-valeur tout le temps. Voici le moyen le plus simple de créer un dictionnaire :

```
{clé1: valeur1, clé2: valeur2, ...}
```

Alors que les listes permettent d'accéder aux éléments par leur indice, c'est-à-dire par leur position, on accède aux éléments d'un dictionnaire grâce à leur clé. Comme pour les indices, les clés sont désignées entre crochets. Les exemples de code suivants utilisent une paire de devises (clé) qui est associée à un taux de change (valeur) :

```
In [68]: taux_de_change = {"EURUSD": 1.1152,
                           "GBPUSD": 1.2454,
                           "AUDUSD": 0.6161}
In [69]: taux_de_change["EURUSD"]  # Accède au taux de change EURUSD
Out[69]: 1.1152
```

Les exemples suivants montrent comment modifier des valeurs existantes et ajouter de nouvelles paires clé-valeur :

```
In [70]: taux_de_change["EURUSD"] = 1.2  # Modifie une valeur existante
         taux_de_change
Out[70]: {'EURUSD': 1.2, 'GBPUSD': 1.2454, 'AUDUSD': 0.6161}
In [71]: taux_de_change["CADUSD"] = 0.714  # Ajoute une nouvelle paire clé-valeur
         taux_de_change
Out[71]: {'EURUSD': 1.2, 'GBPUSD': 1.2454, 'AUDUSD': 0.6161, 'CADUSD': 0.714}
```

Le moyen le plus simple de fusionner deux dictionnaires (voire plusieurs) est de les *combiner* dans un nouveau dictionnaire. On combine un dictionnaire à l'aide de deux astérisques en en-tête. Si le deuxième dictionnaire contient des clés du premier dictionnaire, les valeurs du premier seront remplacées. Vous pouvez voir un exemple de cela en examinant le taux de change GBPUSD :

```
In [72]: {**taux_de_change, **{"SGDUSD": 0.7004, "GBPUSD": 1.2222}}
Out[72]: {'EURUSD': 1.2,
          'GBPUSD': 1.2222,
          'AUDUSD': 0.6161,
          'CADUSD': 0.714,
          'SGDUSD': 0.7004}
```

Python 3.9 a introduit le caractère de barre verticale (appelé en anglais *pipe*) en tant qu'opérateur de fusion dédié pour les dictionnaires, ce qui permet de simplifier l'expression précédente :

```
taux_de_change | {"SGDUSD" : 0,7004, "GBPUSD" : 1,2222}
```

De nombreux objets peuvent servir de clés et voici un exemple avec des entiers :

```
In [73]: devises = {1: "EUR", 2: "USD", 3: "AUD"}
In [74]: devises[1]
Out[74]: 'EUR'
```

En utilisant la méthode `get`, les dictionnaires permettent d'utiliser une valeur par défaut au cas où la clé n'existe pas :

```
In [75]: # devises[100] provoquerait une erreur. À la place de 100,
         # vous pouvez employer n'importe quelle autre clé inexistante.
         devises.get(100, "N/A")
Out[75]: 'N/A'
```

Les dictionnaires peuvent souvent être utilisés lorsque vous employez une instruction `Case` en VBA. L'exemple précédent pourrait être écrit comme ceci en VBA :

```
Select Case x
Case 1
    Debug.Print "EUR"
Case 2
    Debug.Print "USD"
Case 3
    Debug.Print "AUD"
Case Else
    Debug.Print "N/A"
End Select
```

Maintenant que vous savez comment travailler avec les dictionnaires, passons à la structure de données suivante, les tuples, qui sont similaires aux listes, mais qui ont cependant une différence importante, comme nous le verrons dans la section suivante.

Tuples

Les *tuples*[6] sont similaires aux listes, à la différence qu'ils sont *immuables* : une fois créés, leurs éléments ne peuvent pas être modifiés. Bien que l'on puisse souvent utiliser des tuples et des listes de manière interchangeable, les tuples sont à privilégier pour une collection dont les éléments ne sont jamais modifiés dans un programme. Les tuples sont créés en séparant les valeurs par des virgules :

```
mon_tuple = élément1, élément2, ...
```

L'utilisation de parenthèses facilite souvent la lecture :

```
In [76]: devises = ("EUR", "GBP", "AUD")
```

6 La traduction de tuple en français est *n-uplet*, mais la communauté Python francophone emploie majoritairement le terme anglais.

Les tuples permettent d'accéder aux éléments de la même manière que les listes, mais on ne peut pas modifier leurs éléments. Au lieu de cela, la concaténation de tuples créera un nouveau tuple en arrière-plan, puis associera votre variable à ce nouveau tuple :

```
In [77]: devises[0]  # Accès au premier élément
Out[77]: 'EUR'
In [78]: # La concaténation de tuples retourne un nouveau tuple.
         devises + ("SGD",)
Out[78]: ('EUR', 'GBP', 'AUD', 'SGD')
```

J'explique la différence entre les objets *mutables* et *immuables* en détail dans l'Annexe C, mais pour l'instant, jetons un coup d'œil à la dernière structure de données de cette section : les ensembles.

Ensembles

Les *ensembles* (en anglais *sets*) sont des collections qui n'ont pas d'éléments en double. Bien que vous puissiez les utiliser pour les opérations de la théorie des ensembles, en pratique, ils vous aident souvent à obtenir les valeurs uniques d'une liste ou d'un tuple. On crée des ensembles à l'aide d'accolades :

```
{élément1, élément2, ...}
```

Pour obtenir les objets uniques d'une liste ou d'un tuple, utilisez le constructeur `set` de la manière suivante :

```
In [79]: set(["USD", "USD", "SGD", "EUR", "USD", "EUR"])
Out[79]: {'EUR', 'SGD', 'USD'}
```

En dehors de cela, vous pouvez appliquer des opérations de la théorie des ensembles comme l'intersection et l'union :

```
In [80]: portfolio1 = {"USD", "EUR", "SGD", "CHF"}
         portfolio2 = {"EUR", "SGD", "CAD"}
In [81]: # Identique à portfolio2.union(portfolio1)
         portfolio1.union(portfolio2)
Out[81]: {'CAD', 'CHF', 'EUR', 'SGD', 'USD'}
In [82]: # Identique à portfolio2.intersection(portfolio1)
         portfolio1.intersection(portfolio2)
Out[82]: {'EUR', 'SGD'}
```

Pour un aperçu complet des opérations sur les ensembles, reportez-vous à la documentation officielle[7]. Avant de continuer, révisons rapidement à l'aide du Tableau 3.1 les quatre structures de données que nous venons d'étudier. Ce tableau montre un exemple pour chaque structure de données avec la notation que j'ai utilisée dans les paragraphes précédents, c'est-à-dire des *littéraux*. En outre, je liste également leurs constructeurs qui offrent une alternative à l'utilisation

7 https://docs.python.org/fr/3/library/stdtypes.html

des littéraux et qui sont souvent utilisés pour convertir une structure de données en une autre. Par exemple, pour convertir un tuple en liste, procédez de la manière suivante :

```
In [83]: devises = "USD", "EUR", "CHF"
         devises
Out[83]: ('USD', 'EUR', 'CHF')
In [84]: list(devises)
Out[84]: ['USD', 'EUR', 'CHF']
```

Tableau 3.1 : Structures de données.

Structure de données	Littéraux	Constructeur
Liste	`[1, 2, 3]`	`list((1, 2, 3))`
Dictionnaire	`{"a": 1, "b": 2}`	`dict(a=1, b=2)`
Tuple	`(1, 2, 3)`	`tuple([1, 2, 3])`
Ensemble	`{1, 2, 3}`	`set((1, 2, 3))`

À ce stade, vous connaissez tous les types de données importants, y compris les types de données de base tels que les flottants et les chaînes, et les structures de données comme les listes et les dictionnaires. Dans la section suivante, nous abordons le contrôle du flux des instructions dans un programme.

Structures de contrôle

Cette section présente l'instruction `if` ainsi que les boucles `for` et `while`. L'instruction `if` permet d'exécuter certaines lignes de code uniquement si une condition est remplie, et les boucles `for` et `while` exécuteront un bloc de code à plusieurs reprises. À la fin de la section, je présenterai également les compréhensions de listes, qui sont un moyen de construire des listes qui peuvent servir d'alternative aux boucles `for`. Je commencerai cette section avec la définition des blocs de code, pour laquelle je dois également introduire l'une des particularités les plus importantes de Python : les espaces significatifs.

Blocs de code et instruction pass

Un *bloc de code* définit une section dans votre code source qui est utilisée pour quelque chose de spécial. Par exemple, on utilise un bloc de code pour définir les lignes sur lesquelles votre programme effectue une boucle ou réalise la définition d'une fonction. En Python, on définit des blocs de code en les mettant en retrait, et non pas en utilisant des mots-clés (comme en VBA) ou des accolades, comme dans la plupart des autres langages. C'est ce qu'on appelle des *espaces significatifs*. La communauté Python s'est mise d'accord sur le fait que cette mise en retrait (que l'on nomme indentation) soit constituée de quatre espaces ; généralement, on ne saisit pas ces

quatre espaces, mais on appuie sur la touche Tab, les notebooks Jupyter et VS Code se chargeant de convertir automatiquement la touche Tab en espaces. Laissez-moi vous montrer comment les blocs de code sont formellement définis en prenant l'exemple de l'instruction `if` :

```
if condition:
    pass # Ne fait rien
```

La ligne précédant le bloc de code se termine toujours par un signe deux-points. Étant donné que la fin du bloc de code est atteinte lorsque la ligne de code n'est plus mise en retrait, vous devez utiliser l'instruction `pass` si vous souhaitez créer un bloc de code factice qui ne fait rien. En VBA, cela correspondrait à ce qui suit :

```
If condition Then
    ' Ne fait rien
End If
```

L'instruction if et les expressions conditionnelles

Pour présenter l'instruction `if`, permettez-moi de reproduire l'exemple présenté dans la section « Lisibilité et maintenabilité » du Chapitre 1, mais cette fois-ci en Python :

```
In [85]: i = 20
         if i < 5:
             print("i est inférieur à 5")
         elif i <= 10:
             print("i est compris entre 5 et 10")
         else:
             print("i est supérieur à 10")
i est supérieur à 10
```

Si vous faites la même chose que nous au Chapitre 1, c'est-à-dire mettre en retrait les instructions `elif` et `else`, vous obtiendrez une erreur de syntaxe. Python ne permet pas d'indenter votre code différemment par rapport à la logique. Si l'on fait une comparaison avec VBA, les mots-clés sont en minuscules et au lieu des `ElseIf` de VBA, Python utilise `elif`. Les instructions `if` sont un moyen facile de savoir si un programmeur débute en Python ou s'il a déjà adopté un style *pythonique* : en Python, une instruction `if` simple ne nécessite aucune parenthèse et pour tester si une valeur est égale à `True`, vous n'avez pas besoin de le faire explicitement. Cet exemple éclairera mon propos :

```
In [86]: is_important = True
         if is_important:
             print("Ceci est important.")
         else:
             print("Ceci n'est pas important.")
Ceci est important.
```

La même chose fonctionne quand vous voulez vérifier si une séquence telle qu'une liste est vide ou non :

```
In [87]: valeurs = []
         if valeurs:
             print(f"Voici les valeurs suivantes : {valeurs}")
         else:
             print("Aucune valeur n'a été fournie.")
Aucune valeur n'a été fournie.
```

Les programmeurs venant d'autres langages écrivent souvent à la place quelque chose comme `if (is_important == True)` ou `if len(valeurs) > 0`.

Les *expressions conditionnelles*, également appelées *opérateurs ternaires*, permettent d'utiliser un style plus compact pour les instructions `if/else` simples :

```
In [88]: is_important = False
         print("important") if is_important else print("pas important")
pas important
```

Boucles for et while

Si vous avez besoin de faire quelque chose à plusieurs reprises comme afficher la valeur de dix variables différentes, vous vous rendrez un grand service en ne faisant pas dix fois un copier-coller de l'instruction d'affichage. Au lieu de cela, utilisez une boucle `for` pour effectuer cette tâche. Les boucles `for` parcourent chaque élément d'une séquence comme une liste, un tuple ou une chaîne (rappelez-vous que les chaînes sont des séquences de caractères, d'où leur nom de chaînes de caractères). À titre d'exemple introductif, nous allons créer une boucle `for` qui prend chaque élément de la liste `devises`, l'affecte à la variable `devise` et l'affiche, l'un après l'autre, jusqu'à ce qu'il n'y ait plus d'éléments dans la liste :

```
In [89]: devises = ["USD", "HKD", "AUD"]

         for devise in devises:
             print(devise)
USD
HKD
AUD
```

À titre d'information, l'instruction `For Each` de VBA est proche du fonctionnement de la boucle `for` de Python. L'exemple précédent pourrait être écrit de la manière suivante en VBA :

```
Dim devises As Variant
Dim devise As Variant 'currency est un mot réservé en VBA

devises = Array("USD", "HKD", "AUD")
For Each devise In devises
    Debug.Print devise
Next
```

En Python, si vous avez besoin d'une variable compteur dans une boucle `for`, les fonctions natives `range` (qui signifie plage en anglais) ou `enumerate` peuvent vous aider. Examinons d'abord

`range`, qui fournit une séquence de nombres : on l'appelle soit en fournissant un argument `stop` unique, soit avec les arguments `start` et `stop`, et un argument `step` facultatif. Comme avec le découpage, `start` est inclusif, `stop` est exclusif et `step` détermine la taille du pas, 1 étant la valeur par défaut :

```
range(stop)
range(start, stop, step)
```

`range` réalise une évaluation paresseuse (*lazy*), ce qui signifie que si vous ne le demandez pas explicitement, vous ne verrez pas la séquence générée :

```
In [90]: range(5)
Out[90]: range(0, 5)
```

La conversion de la plage en une liste résout ce problème :

```
In [91]: list(range(5))  # argument stop
Out[91]: [0, 1, 2, 3, 4]
In [92]: list(range(2, 5, 2))  # avec les arguments start, stop, step
Out[92]: [2, 4]
```

La plupart du temps, il n'est cependant pas nécessaire d'encapsuler la plage dans une liste :

```
In [93]: for i in range(3):
             print(i)
0
1
2
```

Si vous avez besoin d'une variable compteur lors d'une boucle sur une séquence, utilisez la fonction `enumerate` qui retourne une séquence de tuples (`indice`, `élément`). Par défaut, l'indice commence à zéro et s'incrémente d'une unité. Vous pouvez utiliser `enumerate` dans une boucle comme celle-ci :

```
In [94]: for i, devise in enumerate(devises):
             print(i, devise)
0 USD
1 HKD
2 AUD
```

La boucle sur les tuples et les ensembles fonctionne de la même manière qu'avec les listes. Lorsque vous exécutez une boucle sur un dictionnaire, Python effectue une boucle sur les clés :

```
In [95]: taux_de_change = {"EURUSD": 1.1152,
                           "GBPUSD": 1.2454,
                           "AUDUSD": 0.6161}
         for paire_devises in taux_de_change:
             print(paire_devises)
EURUSD
GBPUSD
AUDUSD
```

En utilisant la méthode `items`, on obtient la clé et la valeur en même temps que le tuple :

```
In [96]: for paire_devises, taux_de_change in taux_de_change.items():
             print(paire_devises, taux_de_change)
EURUSD 1.1152
GBPUSD 1.2454
AUDUSD 0.6161
```

Pour quitter une boucle, utilisez l'instruction `break` :

```
In [97]: for i in range(15):
             if i == 2:
                 break
             else:
                 print(i)
0
1
```

On saute le reste d'une boucle avec l'instruction `continue`, ce qui signifie que l'exécution se poursuit avec une nouvelle boucle et l'élément suivant :

```
In [98]: for i in range(4):
             if i == 2:
                 continue
             else:
                 print(i)
0
1
3
```

Lorsque vous comparez les boucles `for` en VBA avec celles de Python, il existe une différence subtile : en VBA, la variable compteur augmente au-delà de la limite supérieure après avoir terminé la boucle :

```
For i = 1 To 3
    Debug.Print i
Next i
Debug.Print i
```

Cela affiche :

```
1
2
3
4
```

En Python, le code se comporte comme vous vous y attendez probablement :

```
In [99]: for i in range(1, 4):
             print(i)
         print(i)
1
2
3
3
```

Au lieu de parcourir une séquence, vous pouvez également utiliser une *boucle while* pour exécuter une boucle quand une certaine condition est vraie :

```
In [100]: n = 0
          while n <= 2:
              print(n)
              n += 1
0
1
2
```

Affectation augmentée

J'ai utilisé la notation d'*affectation augmentée* dans le dernier exemple : `n += 1`. C'est la même chose que si vous écriviez `n = n + 1`. Cette notation fonctionne également avec tous les autres opérateurs mathématiques que j'ai présentés précédemment ; par exemple, avec le signe moins, vous pouvez écrire `n -= 1`.

Très souvent, vous devrez collecter certains éléments dans une liste pour un traitement ultérieur. Dans ce cas, Python offre une alternative à l'écriture de boucles : les compréhensions de listes, de dictionnaires et d'ensembles.

Compréhensions de listes, de dictionnaires et d'ensembles

Les compréhensions de listes, de dictionnaires et d'ensembles sont techniquement un moyen de créer des structures de données, mais elles remplacent souvent une boucle `for`, raison pour laquelle je les présente ici. Supposons que dans la liste suivante de paires de devises, vous souhaitiez sélectionner les devises où la devise USD est cotée comme deuxième devise. Vous pouvez écrire la boucle `for` suivante :

```
In [101]: paires_devises = ["USDJPY", "USDGBP", "USDCHF",
                            "USDCAD", "AUDUSD", "NZDUSD"]
          In [102]: devise_usd = []
          for paire in paires_devises:
              if paire[3:] == "USD":
                  devise_usd.append(paire[:3])
          devise_usd
Out[102]: ['AUD', 'NZD']
```

Cela est souvent plus facile à écrire avec une *compréhension de liste* qui est une façon concise de créer une liste. Vous pouvez découvrir sa syntaxe à partir de cet exemple, qui fait la même chose que la boucle `for` précédente :

```
In [103]: [paire[:3] for paire in paires_devises if paire[3:] == "USD"]
Out[103]: ['AUD', 'NZD']
```

Si vous n'avez aucune condition à satisfaire, laissez simplement la partie `if` de côté. Par exemple, pour inverser toutes les paires de devises afin que la première devise arrive en second et *vice versa*, on peut procéder de la manière suivante :

```
In [104]: [paire[3:] + paire[:3] for paire in paires_devises]
Out[104]: ['JPYUSD', 'GBPUSD', 'CHFUSD', 'CADUSD', 'USDAUD', 'USDNZD']
```

Avec les dictionnaires, il existe des compréhensions de dictionnaires :

```
In [105]: taux_de_change = {"EURUSD": 1.1152,
                            "GBPUSD": 1.2454,
                            "AUDUSD": 0.6161}
          {k: v * 100 for (k, v) in taux_de_change.items()}
Out[105]: {'EURUSD': 111.52, 'GBPUSD': 124.54, 'AUDUSD': 61.61}
```

Et avec les ensembles, il y a aussi des compréhensions d'ensembles :

```
In [106]: {s + "USD" for s in ["EUR", "GBP", "EUR", "HKD", "HKD"]}
Out[106]: {'EURUSD', 'GBPUSD', 'HKDUSD'}
```

À ce stade, vous êtes déjà en mesure d'écrire des scripts simples, car vous connaissez la plupart des blocs de base de Python. Dans la section suivante, vous apprendrez à organiser votre code pour qu'il reste maintenable lorsque vos scripts commencent à prendre de l'ampleur.

Organisation du code

Dans cette section, nous allons examiner la manière d'intégrer le code dans une structure maintenable : je vais commencer par présenter les fonctions avec tous les détails dont vous aurez généralement besoin avant de vous montrer comment diviser votre code en différents modules Python. Les connaissances sur les modules nous permettront d'achever cette section en examinant le module `datetime` qui fait partie de la bibliothèque standard.

Fonctions

Même si vous n'utilisez Python que pour des scripts simples, vous allez quand même écrire des fonctions régulièrement, car elles sont l'une des constructions les plus importantes de chaque langage de programmation et elles permettent de réutiliser les mêmes lignes de code à partir de n'importe quel endroit de votre programme. Avant de voir comment appeler une fonction, nous allons apprendre à les définir.

Définition des fonctions

Pour écrire votre propre fonction en Python, vous devez utiliser le mot-clé `def`, qui est l'abréviation de *définition*. Contrairement à VBA, Python ne fait pas la différence entre une fonction

et une procédure `Sub`. En Python, l'équivalent d'une procédure `Sub` est simplement une fonction qui ne retourne rien. Les fonctions en Python suivent la syntaxe des blocs de code, c'est-à-dire qu'il faut terminer la première ligne par un signe deux-points et mettre en retrait le corps de la fonction :

```
def nom_fonction(argument_requis, argument_facultatif=valeur_par_défaut, ...):
    return valeur_1, valeur_2, ...
```

Arguments requis

Les arguments requis n'ont pas de valeur par défaut. Quand il y a plusieurs arguments, ils sont séparés par des virgules.

Arguments facultatifs

On rend un argument facultatif en lui fournissant une valeur par défaut. `None` est souvent utilisé pour rendre un argument facultatif s'il n'existe aucune valeur par défaut significative.

Valeur de retour

L'instruction `return` définit la valeur retournée par la fonction. Si vous omettez la valeur de retour, la fonction renvoie automatiquement `None`. Python permet de renvoyer plusieurs valeurs séparées par des virgules.

Afin de tester une fonction, nous allons en définir une capable de convertir une température exprimée en degrés Fahrenheit ou Kelvin en degrés Celsius :

```
In [107]: def convert_to_celsius(degres, source="fahrenheit"):
              if source.lower() == "fahrenheit":
                  return (degres-32) * (5/9)
              elif source.lower() == "kelvin":
                  return degres - 273.15
              else:
                  return f"Impossible de convertir à partir de {source}"
```

J'utilise la méthode de chaîne `lower`, qui transforme les chaînes en minuscules. Cela permet d'accepter la chaîne `source` avec n'importe quelle casse, ce qui garantit le fonctionnement de la comparaison. Une fois la fonction `convert_to_celsius` définie, voyons comment nous pouvons l'appeler.

Appel de fonctions

Comme je l'ai brièvement mentionné au début de ce chapitre, on appelle une fonction en ajoutant au nom de la fonction des arguments encadrés par des parenthèses :

```
Valeur_1, valeur_2, ... = nom_fonction(argument_positionnel, nom_argument=valeur, ...)
```

Arguments positionnels

Si vous fournissez une valeur en tant qu'argument positionnel, les valeurs sont associées aux arguments en fonction de leur position dans la définition de la fonction.

Arguments nommés

En fournissant l'argument sous la forme `nom_argument=valeur`, vous fournissez un argument nommé. Cela présente l'avantage de pouvoir fournir les arguments dans n'importe quel ordre. C'est également plus explicite pour le lecteur, ce qui facilite la compréhension du code. Par exemple, si la fonction est définie sous la forme `f(a, b)`, vous pouvez appeler la fonction de la manière suivante : `f(b=1, a=2)`. Ce concept existe également en VBA, où vous pouvez utiliser des arguments nommés en appelant une fonction comme ceci : `f(b:=1, a:=1)`.

Testons à présent la fonction `convert_to_celsius` pour voir comment tout cela fonctionne en pratique :

```
In [108]: convert_to_celsius(100, "fahrenheit")  # Arguments positionnels
Out[108]: 37.77777777777778
In [109]: convert_to_celsius(50)  # Utilise la source par défaut (fahrenheit)
Out[109]: 10.0
In [110]: convert_to_celsius(source="kelvin", degres=0)  # Arguments nommés
Out[110]: -273.15
```

Maintenant que vous savez comment définir et appeler des fonctions, voyons comment les organiser à l'aide de modules.

Modules et instruction import

Lorsque vous écrivez du code pour des projets plus importants, à un moment donné, vous devrez le diviser en différents fichiers pour pouvoir l'intégrer dans une structure maintenable. Comme nous l'avons déjà vu dans le chapitre précédent, les fichiers Python ont une extension *.py* et on appelle généralement le fichier principal un *script*. Si vous souhaitez que votre script principal accède à des fonctionnalités à partir d'autres fichiers, vous devez d'abord *importer* ces fonctionnalités. Dans ce contexte, les fichiers sources Python sont appelés *modules*. Pour avoir une meilleure idée de la façon dont cela fonctionne et comprendre les différentes options d'importation, jetez un œil au fichier *temperature.py* dans les exemples de code de ce livre en l'ouvrant avec VS Code (voir l'Exemple 3.1). Si vous avez besoin d'un rappel sur la façon d'ouvrir des fichiers dans VS Code, reportez-vous au Chapitre 2.

Exemple 3.1 : temperature.py.

```
TEMPERATURE_SCALES = ("fahrenheit", "kelvin", "celsius")

def convert_to_celsius(degres, source="fahrenheit"):
    if source.lower() == "fahrenheit":
```

```
        return (degres-32) * (5/9)
    elif source.lower() == "kelvin":
        return degres - 273.15
    else:
        return f"Impossible de convertir à partir de {source}"

print("Voici le module temperature.")
```

Pour pouvoir importer le module `temperature` à partir de votre notebook Jupyter, vous devez disposer du notebook Jupyter et du module `temperature` dans le même répertoire, comme c'est le cas dans les exemples de code. Pour importer, on emploie uniquement le nom du module, sans son extension *.py*. Après avoir exécuté l'instruction `import`, vous aurez accès à tous les objets de ce module Python *via* la notation pointée. Par exemple, utilisez `temperature.convert_to_celsius()` pour effectuer une conversion :

```
In [111]: import temperature
Voici le module temperature.
In [112]: temperature.TEMPERATURE_SCALES
Out[112]: ('fahrenheit', 'kelvin', 'celsius')
In [113]: temperature.convert_to_celsius(120, "fahrenheit")
Out[113]: 48.88888888888889
```

Notez que j'ai utilisé des lettres majuscules pour `TEMPERATURE_SCALES` afin d'exprimer qu'il s'agit d'une constante (j'en dirai un peu plus à ce sujet à la fin de ce chapitre). Lorsque vous exécutez la cellule avec le code `import temperature`, Python exécute le fichier *temperature.py* en lisant le fichier de haut en bas. Vous pouvez facilement voir que cela se produit puisque l'importation du module déclenche la fonction d'affichage en bas de *temperature.py*.



Si vous réexécutez la cellule `import temperature`, vous remarquerez qu'elle n'affiche plus rien. En effet, les modules Python ne sont importés qu'une seule fois par session. Si vous modifiez le code d'un module que vous importez, vous devez redémarrer votre interpréteur Python pour prendre en compte toutes les modifications, c'est-à-dire que dans un notebook Jupyter, vous devrez cliquer sur Kernel > Restart.

En réalité, on n'affiche généralement rien dans les modules. Grâce à cette fonction d'affichage, j'ai simplement voulu vous montrer l'effet de l'importation d'un module plusieurs fois. Le plus souvent, on place des fonctions et des classes dans les modules (pour plus d'informations sur les classes, reportez-vous à l'Annexe C). Si vous ne souhaitez pas saisir `temperature` chaque fois que vous utilisez un objet du module `temperature`, modifiez l'instruction `import` de la manière suivante :

```
In [114]: import temperature as tp
In [115]: tp.TEMPERATURE_SCALES
Out[115]: ('fahrenheit', 'kelvin', 'celsius')
```

L'attribution de l'alias court `tp` à votre module peut le rendre plus facile à utiliser tout en permettant de toujours savoir clairement d'où vient cet objet. De nombreux packages tiers suggèrent

une convention spécifique quand on utilise un alias. Par exemple, pandas utilise `import pandas as pd`. Il existe une autre option pour importer des objets à partir d'un autre module :

```
In [116]: from temperature import TEMPERATURE_SCALES, convert_to_celsius
In [117]: TEMPERATURE_SCALES
Out[117]: ('fahrenheit', 'kelvin', 'celsius')
```



Lorsque vous importerez le module `temperature`, vous verrez que Python crée un dossier appelé __pycache__ avec des fichiers ayant l'extension .pyc. Il s'agit de fichiers compilés en bytecode que l'interpréteur Python crée lorsque vous importez un module. Pour nos besoins, nous pouvons simplement ignorer ce dossier, car il s'agit d'un détail technique de la façon dont Python exécute votre code.

Lorsque vous utilisez la syntaxe `from x import y`, vous importez uniquement des objets spécifiques. Ce faisant, vous les importez directement dans *l'espace de noms* de votre script principal : autrement dit, si vous n'examinez pas les instructions `import`, vous ne pourrez pas dire si les objets importés ont été définis dans votre script Python actuel ou votre notebook Jupyter ou s'ils proviennent d'un autre module. Cela pourrait provoquer des conflits : si votre script principal a une fonction appelée `convert_to_celsius`, il remplacerait celle que vous importez à partir du module `temperature`. Toutefois, si vous utilisez l'une des deux méthodes précédentes, votre fonction locale et celle du module importé peuvent cohabiter en tant que `convert_to_celsius` et `temperature.convert_to_celsius`.



Une source courante d'erreurs consiste à nommer votre fichier Python de la même manière qu'un package ou un module Python existant. Si vous créez un fichier pour tester certaines fonctionnalités pandas, n'appelez pas ce fichier pandas.py, car cela peut provoquer des conflits.

Maintenant que vous savez comment fonctionne le mécanisme d'importation, utilisons-le tout de suite pour importer le module `datetime`. Cela vous permettra également d'approfondir vos connaissances sur les objets et les classes.

Classe datetime

L'utilisation de la date et de l'heure est une opération courante dans Excel, mais elle comporte des limites : par exemple, le format de cellule d'Excel pour l'heure ne prend pas en charge des unités plus petites que les millisecondes et les fuseaux horaires ne sont pas du tout pris en charge. Dans Excel, la date et l'heure sont stockées sous la forme d'un simple nombre en virgule flottante appelé *numéro de série de date*. La cellule Excel est ensuite mise en forme pour l'afficher sous la forme d'une date ou d'une heure (voire les deux). Par exemple, le 1er janvier 1900 a le numéro de série de date 1, ce qui signifie qu'il s'agit également de la date la plus ancienne avec laquelle vous pouvez travailler dans Excel. Le temps est traduit dans la partie décimale du nombre en virgule flottante, par exemple, 01/01/1900 10:10:00 est représenté par 1.42361111111.

En Python, pour travailler avec la date et l'heure, on importe le module `datetime`, qui fait partie de la bibliothèque standard. Le module `datetime` contient une classe portant le même nom qui permet de créer des objets `datetime`. Comme le fait d'avoir le même nom pour le module et la classe peut prêter à confusion, j'utiliserai la convention d'importation suivante tout au long de ce livre : `import datetime as dt`. Cela facilite la différenciation entre le module (`dt`) et la classe (`datetime`).

Jusqu'à présent, nous avons la plupart du temps employé des *littéraux* pour créer des objets tels que des listes ou des dictionnaires. Les littéraux font référence à la syntaxe qui permet à Python de reconnaître un type d'objet spécifique : dans le cas d'une liste, il s'agirait, par exemple, de [1, 2, 3]. Cependant, la plupart des objets doivent être créés en appelant leur classe : ce processus est appelé *instanciation*, et les objets sont donc appelés *instances de classe*. L'appel d'une classe fonctionne de la même manière que l'appel d'une fonction, c'est-à-dire que l'on ajoute des parenthèses au nom de la classe et on fournit les arguments de la même manière que nous l'avons fait avec les fonctions. Pour instancier un objet `datetime`, vous devez appeler la classe de la manière suivante :

```
import datetime as dt
dt.datetime(year, month, day, hour, minute, second, microsecond, timezone)
```

Passons en revue quelques exemples pour voir comment on travaille avec les objets `datetime` en Python. Pour les besoins de cette introduction, nous allons ignorer les fuseaux horaires et nous manipulerons des objets `datetime` naïfs de fuseau horaire :

```
In [118]: # Importe le module datetime avec l'alias "dt"
          import datetime as dt
In [119]: # Instancie un objet datetime appelé "timestamp"
          timestamp = dt.datetime(2020, 1, 31, 14, 30)
          timestamp
Out[119]: datetime.datetime(2020, 1, 31, 14, 30)
In [120]: # Les objets datetime offrent des attributs,
          # par exemple pour obtenir le numéro du jour
          timestamp.day
Out[120]: 31
In [121]: # La différence entre deux objets datetime renvoie un objet timedelta
          timestamp - dt.datetime(2020, 1, 14, 12, 0)
Out[121]: datetime.timedelta(days=17, seconds=9000)
In [122]: # Par conséquent, on peut aussi travailler avec des objets timedelta
          timestamp + dt.timedelta(days=1, hours=4, minutes=11)
Out[122]: datetime.datetime(2020, 2, 1, 18, 41)
```

Pour mettre en *forme* des objets `datetime` et les convertir en chaînes, utilisez la méthode `strftime` ; pour analyser une chaîne et la convertir en objet `datetime`, utilisez la fonction `strptime` (vous trouverez des informations sur les codes de format acceptés dans la documentation sur `datetime`[8]) :

8 https://docs.python.org/fr/3/library/datetime.html#strftime-and-strptime-behavior

```
In [123]: # Pour formater un objet datetime spécifiquement
          # on peut aussi utiliser une f-string: f"{timestamp:%d/%m/%Y %H:%M}"
          timestamp.strftime("%d/%m/%Y %H:%M")
Out[123]: '31/01/2020 14:30'
In [124]: # Analyse une chaine pour la convertir en un objet datetime
          dt.datetime.strptime("12.1.2020", "%d.%m.%Y")
Out[124]: datetime.datetime(2020, 1, 12, 0, 0)
```

Après cette brève introduction au module `datetime`, passons à la dernière rubrique de ce chapitre, qui traite de la mise en forme correcte de votre code.

PEP 8 : Guide de style du code Python

Vous vous demandez peut-être pourquoi j'utilise parfois des noms de variables avec des barres de soulignement ou des majuscules. Cette section expliquera mes choix de formatage en vous présentant le guide de style officiel de Python. Python utilise ce que l'on appelle les PEP (*Python Enhancement Proposals*, c'est-à-dire Propositions d'amélioration de Python) pour discuter de l'introduction de nouvelles fonctionnalités dans le langage. L'une de ces propositions, le Guide de style pour le code Python, est généralement désignée par son numéro : PEP 8. PEP 8 est un ensemble de recommandations qui définit le style des programmes de la communauté Python ; si tous ceux qui travaillent sur le même code suivent le même guide de style, le code devient alors beaucoup plus lisible. Cela est particulièrement important dans le monde de l'open source où de nombreux programmeurs travaillent sur le même projet, souvent sans se connaître. L'exemple 3.2 montre un fichier Python court qui illustre les conventions les plus importantes.

Exemple 3.2 : pep8_sample.py.

```
"""Ce script illustre quelques règles PEP 8. ❶
"""

import datetime as dt ❷


TEMPERATURE_SCALES = ("fahrenheit", "kelvin",
                      "celsius")❸
❹

class TemperatureConverter: ❺
    pass  # Ne fait rien pour le moment ❻


def convert_to_celsius(degrees, source="fahrenheit"): ❼
    """Cette fonction convertit des degrés Fahrenheit ou Kelvin
    en degrés Celsius. ❽
    """
    if source.lower() == "fahrenheit": ❾
        return (degrees-32) * (5/9) ❿
```

```
    elif source.lower() == "kelvin":
        return degrees - 273.15
    else:
        return f"Impossible de convertir à partir de {source}"


celsius = convert_to_celsius(44, source="fahrenheit") ⓫
non_celsius_scales = TEMPERATURE_SCALES[:-1] ⓬

print("Heure actuelle : " + dt.datetime.now().isoformat())
print(f"La température en Celsius est : {celsius}")
```

❶ Explique ce que fait le script ou le module avec une *docstring* au sommet du code. Une docstring est un type spécial de chaîne, entouré de guillemets triples. En plus de servir de texte pour documenter votre code, une docstring, qui facilite également l'écriture de chaînes sur plusieurs lignes, est utile si votre texte contient beaucoup de guillemets doubles ou de guillemets simples, car vous n'aurez pas besoin de les « échapper ». Les docstrings sont aussi utiles pour écrire des requêtes SQL multilignes, comme nous le verrons au Chapitre 11.

❷ Toutes les importations se trouvent en haut du fichier, une importation par ligne. On liste d'abord les importations de la bibliothèque standard, puis celles des packages tiers et enfin celles de ses propres modules. Cet exemple utilise uniquement la bibliothèque standard.

❸ On utilise des majuscules avec des barres de soulignement pour les constantes. Employez une longueur de ligne maximale de 79 caractères. Si possible, employez des parenthèses, des crochets ou des accolades pour les sauts de ligne implicites.

❹ On sépare du reste du code les classes et les fonctions avec deux lignes vides.

❺ Bien que de nombreuses classes comme `datetime` soient toutes en minuscules, vos propres classes doivent utiliser la convention qui emploie des initiales en majuscules, comme dans `ConvertirTemperature`. Pour plus d'informations sur les classes, reportez-vous à l'Annexe C.

❻ Les commentaires en ligne doivent être séparés du code par au moins deux espaces. Les blocs de code doivent être indentés avec quatre espaces.

❼ Les fonctions et les arguments de fonction doivent utiliser des noms en minuscules avec des barres de soulignement s'ils améliorent la lisibilité. N'utilisez pas d'espaces entre le nom de l'argument et sa valeur par défaut.

❽ La docstring d'une fonction doit également répertorier et expliquer les arguments de la fonction. Je ne l'ai pas fait ici pour que cet exemple reste court, mais vous trouverez des docstrings complètes dans le fichier *excel.py* qui est inclus dans les exemples de code de ce livre et que nous étudierons au Chapitre 8.

❾ N'utilisez pas d'espaces autour du signe deux-points.

⑩ Utilisez des espaces de chaque côté des opérateurs mathématiques. Si des opérateurs avec des priorités différentes sont utilisés, vous pouvez envisager d'ajouter des espaces seulement de chaque côté de ceux ayant la priorité la plus faible. Étant donné que dans cet exemple la multiplication a la priorité la plus basse, j'ai ajouté des espaces autour de l'étoile.

⑪ Utilisez des noms en minuscules pour les variables. Utilisez des barres de soulignement si elles améliorent la lisibilité. Lorsque vous affectez une valeur à une variable, utilisez des espaces autour du signe égal. Toutefois, lorsque vous appelez une fonction, n'utilisez pas d'espaces autour du signe égal employé avec des arguments nommés.

⑫ Avec l'indexation et le découpage, n'utilisez pas d'espaces autour des crochets.

Il s'agit ici d'un résumé simplifié des règles de PEP 8, et il est donc conseillé d'examiner le texte original de PEP 8[9] une fois que vous aurez acquis plus d'expérience en Python. PEP 8 indique clairement qu'il s'agit d'une recommandation et que vos propres guides de style restent prioritaires. Après tout, la cohérence est le facteur le plus important. Si vous êtes intéressé par d'autres directives publiques, vous voudrez peut-être consulter le guide de style de Google pour Python, qui est assez proche de PEP 8. En pratique, la plupart des programmeurs Python adhèrent vaguement à PEP 8, et ignorer la longueur de ligne maximale de 79 caractères est probablement le péché le plus courant.

Comme il peut être difficile de mettre en forme votre code correctement lors de son écriture, vous pouvez faire vérifier votre style automatiquement et la section suivante vous montre comment cela fonctionne avec VS Code.

PEP 8 et VS Code

Lorsque vous travaillez avec VS Code, il existe un moyen simple de vous assurer que votre code respecte les règles de PEP 8 : il suffit d'employer un *linter*. Un linter vérifie les erreurs de syntaxe et de style dans votre code source. Lancez la palette de commandes (Ctrl + Maj + P sous Windows ou Commande + Maj + P sous macOS) et recherchez « Python: Enable Linting ». Parmi les options populaires, on peut citer *flake8*, un package qui est préinstallé avec Anaconda. S'il est activé, VS Code souligne les problèmes à l'aide de lignes ondulées chaque fois que vous enregistrez votre fichier. Si vous survolez une ligne ondulée, vous obtenez une explication dans une info-bulle. Vous pourrez désactiver le linter en recherchant à nouveau « Python: Enable Linting » dans la palette de commandes et en choisissant « Disable Linting ». Si vous préférez, vous pouvez également exécuter `flake8` dans Anaconda Prompt pour afficher un rapport (la commande n'affiche quelque chose que s'il y a une violation des règles de PEP 8, si bien que l'exécution de cette commande sur *pep8_sample.py* n'affichera rien à moins que vous n'ayez pas respecté une des règles) :

9 https://www.python.org/dev/peps/pep-0008/

```
(base)> cd C:\Utilisateurs\nom d'utilisateur\python-for-excel
(base)> flake8 pep8_sample.py
```

Python a récemment poussé l'analyse du code statique un peu plus loin en ajoutant la prise en charge des *indicateurs de type*.

Indicateurs de type

En VBA, on voit souvent du code qui préfixe chaque variable avec une abréviation indiquant le type de données, comme `strNomEmploye` ou `wbNomClasseur`. Bien que personne ne vous empêche de le faire en Python, ce n'est pas une pratique courante. Vous ne trouverez pas non plus d'équivalent à l'instruction `Option Explicit` ou `Dim` de VBA pour déclarer le type d'une variable. Au lieu de cela, Python 3.5 a introduit une fonctionnalité appelée *indicateur de type*. Les indicateurs de type sont également appelés *annotations de type* et permettent de déclarer le type de données d'une variable. Ils sont totalement facultatifs et n'ont aucun effet sur la façon dont le code est exécuté par l'interpréteur Python (il existe toutefois des packages tiers tels que pydantic[10] qui peuvent appliquer des indicateurs de type au moment de l'exécution). L'objectif principal des indicateurs de type est de permettre aux éditeurs de texte tels que VS Code d'intercepter plus d'erreurs avant d'exécuter le code, mais ils peuvent également améliorer l'autocomplétion de VS Code et d'autres éditeurs. Le vérificateur de type le plus populaire pour le code employant des annotations de type est mypy, que VS Code propose comme linter. Pour avoir une idée du fonctionnement des annotations de type dans Python, voici un court exemple sans indicateurs de type :

```
x = 1
def hello(nom) :
    return f"Bonjour {nom} !"
```

Et le même code avec des indicateurs de type :

```
x : int = 1
def hello(nom: str) -> str:
    return f"Bonjour {nom} !"
```

Comme les indicateurs de type ont généralement plus de sens dans les bases de code plus importantes, je ne les utiliserai pas dans le reste de ce livre.

Conclusion

Ce chapitre était une introduction complète à Python. Nous avons étudié les principaux éléments constitutifs du langage, notamment les structures de données, les fonctions et les modules. Nous avons également abordé certaines des particularités de Python, telles que les espaces

10 https://pydantic-docs.helpmanual.io/

significatifs et les directives de formatage du code, plus connues sous le nom de PEP 8. Pour poursuivre la lecture de ce livre, vous n'aurez pas besoin de connaître tous les détails : en tant que débutant, il suffit de connaître les listes et les dictionnaires, l'indexation et le découpage, ainsi que la façon de travailler avec les fonctions, les modules, les boucles `for` et les instructions `if` ; toutes ces notions vous permettront déjà de faire beaucoup de choses.

Par rapport à VBA, je trouve Python plus cohérent et plus puissant, tout en étant plus facile à apprendre. Si vous êtes un fan inconditionnel de VBA et que ce chapitre ne vous a pas encore convaincu, je suis à peu près sûr que la partie suivante saura vous persuader de la justesse de mon propos : je vous y présenterai les calculs basés sur les tableaux avant d'entamer notre voyage vers l'analyse de données avec la bibliothèque pandas. Nous commencerons cependant la Partie II par l'apprentissage des notions de base de NumPy.

PARTIE II

Introduction à pandas

CHAPITRE 4

Bases de NumPy

Comme vous vous en souvenez peut-être, nous avons écrit au Chapitre 1 que NumPy est le package de base pour le calcul scientifique en Python et qu'il fournit un support pour les calculs basés sur les tableaux et l'algèbre linéaire. NumPy étant l'épine dorsale de pandas, je vais présenter ses bases dans ce chapitre : après avoir expliqué ce qu'est un tableau NumPy, nous allons nous pencher sur la vectorisation et la diffusion, deux concepts importants qui permettent d'écrire du code mathématique concis et que vous retrouverez dans pandas. Après cela, nous allons voir pourquoi NumPy offre des fonctions spéciales appelées fonctions universelles avant de conclure ce chapitre en apprenant à lire et à définir des valeurs à partir d'un tableau et en expliquant la différence entre une vue et une copie d'un tableau NumPy. Même si nous n'allons pas beaucoup utiliser NumPy directement dans ce livre, connaître ses bases facilitera l'apprentissage de pandas dans le prochain chapitre.

Prise en main de NumPy

Dans cette section, nous allons en apprendre davantage sur les tableaux NumPy unidimensionnels et bidimensionnels et sur les termes techniques *vectorisation, diffusion* et *fonction universelle.*

Tableau NumPy

Pour effectuer des calculs basés sur des tableaux avec des listes imbriquées, comme ceux que nous avons rencontrés dans le chapitre précédent, vous devrez écrire une sorte de boucle. Par exemple, pour ajouter un nombre à chaque élément d'une liste imbriquée, vous pouvez employer la compréhension de listes imbriquées suivante :

```
In [1]: matrice = [[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]]
In [2]: [[i + 1 for i in row] for row in matrice]
Out[2]: [[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

Cela n'est pas très lisible et, plus important encore, si vous travaillez avec de grands tableaux, la boucle qui parcourt chaque élément devient très lente. En fonction de votre exemple et de la taille des tableaux, un calcul employant des tableaux NumPy à la place de listes Python peut être beaucoup plus rapide, le gain pouvant atteindre un facteur 100. NumPy réalise ces performances en utilisant du code qui a été écrit en C ou en Fortran, deux langages de programmation compilés qui sont beaucoup plus rapides que Python. Un tableau NumPy est un tableau à n dimensions de *données homogènes*, c'est-à-dire que tous les éléments du tableau doivent être du même type de données. Le plus souvent, vous avez affaire à des tableaux unidimensionnels et bidimensionnels de nombres à virgule flottante tels qu'ils sont illustrés à la Figure 4.1.

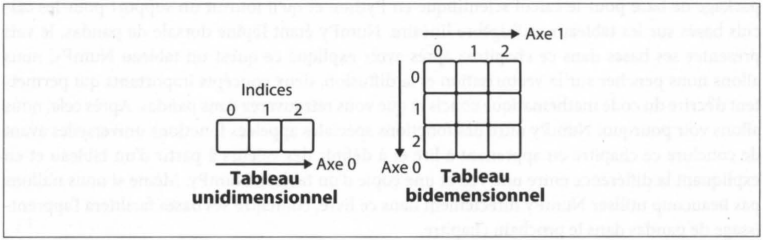


Figure 4.1 : Tableaux NumPy unidimensionnel et bidimensionnel.

Créons un tableau unidimensionnel et un tableau bidimensionnel avec lesquels nous allons travailler tout au long de ce chapitre :

```
In [3]: # D'abord, on importe NumPy
        import numpy as np
In [4]: # On crée un tableau unidimensionnel avec une simple liste
        array1 = np.array([10, 100, 1000.])
In [5]: # On crée un tableau bidimensionnel avec une liste imbriquée
        array2 = np.array([[1., 2., 3.],
                           [4., 5., 6.]])
```



Il est important de noter la différence entre un tableau unidimensionnel et un tableau bidimensionnel : un tableau unidimensionnel n'a qu'un seul axe et n'a donc pas d'orientation explicite de colonne ou de ligne. Bien que les tableaux en VBA se comportent de la même manière, vous devrez peut-être vous y habituer si vous avez l'expérience d'un langage comme MATLAB, où les tableaux unidimensionnels ont toujours une orientation de colonne ou de ligne.

Même si `array1` se compose d'entiers à l'exception du dernier élément (qui est un float), l'homogénéité des tableaux NumPy force le type de données du tableau à être `float64`, qui est capable de prendre en charge tous les éléments. Pour en savoir plus sur le type de données d'un tableau, accédez à son attribut `dtype` :

```
In [6]: array1.dtype
Out[6]: dtype('float64')
```

Étant donné que `dtype` renvoie `float64` au lieu de `float` que nous avons rencontré dans le chapitre précédent, vous avez peut-être deviné que NumPy utilise ses propres types de données numériques, qui sont plus granulaires que les types de données de Python. Ce n'est généralement pas un problème, car la plupart du temps, la conversion entre les différents types de données en Python et NumPy est réalisée automatiquement. Si jamais vous avez besoin de convertir explicitement un type de données NumPy en l'un des types de données de base de Python, utilisez simplement le constructeur correspondant (j'en dirai plus dans un moment sur l'accès aux éléments d'un tableau) :

```
In [7]: float(array1[0])
Out[7]: 10.0
```

Pour obtenir la liste complète des types de données de NumPy, consultez la documentation de NumPy[1]. Avec les tableaux NumPy, vous pouvez écrire du code simple pour effectuer des calculs basés sur des tableaux, comme nous le verrons plus loin.

Vectorisation et diffusion

Si vous générez la somme d'un scalaire et d'un tableau NumPy, NumPy effectuera une opération par élément, ce qui signifie que vous n'avez pas à parcourir les éléments vous-même. La communauté NumPy appelle cette fonctionnalité *vectorisation*. Cela permet d'écrire un code concis, très proche de la notation mathématique :

```
In [8]: array2 + 1
Out[8]: array([[2., 3., 4.],
               [5., 6., 7.]])
```



Un scalaire désigne un type de données Python de base comme un float ou un string. Il s'agit de les différencier des structures de données ayant plusieurs éléments comme des listes, des dictionnaires ou des tableaux NumPy unidimensionnels et bidimensionnels.

Le même principe s'applique lorsque vous travaillez avec deux tableaux : NumPy effectue l'opération au niveau des éléments :

```
In [9]: array2 * array2
Out[9]: array([[ 1.,  4.,  9.],
               [16., 25., 36.]])
```

1 https://numpy.org/devdocs/user/basics.types.html

Si vous utilisez dans une opération arithmétique deux tableaux de formes différentes, NumPy agrandit automatiquement – si c'est possible – le tableau plus petit afin que leurs formes deviennent compatibles. C'est ce que l'on appelle la *diffusion* (*broadcasting*) :

```
In [10]: array2 * array1
Out[10]: array([[  10.,  200., 3000.],
                [  40.,  500., 6000.]])
```

Pour effectuer des multiplications matricielles ou des produits scalaires, utilisez l'opérateur @.[2]

```
In [11]: array2 @ array2.T  # array2.T est un raccourci pour array2.transpose()
Out[11]: array([[14., 32.],
                [32., 77.]])
```

Ne vous laissez pas intimider par les termes que j'ai introduits dans cette section, comme scalaire, vectorisation ou diffusion ! Si vous avez déjà travaillé avec des matrices dans Excel, tout cela devrait vous paraître très naturel, comme cela est illustré à la Figure 4.2. La capture d'écran est extraite du fichier *array_calculations.xlsx*, que vous trouverez dans le répertoire *xl* des exemples de code.

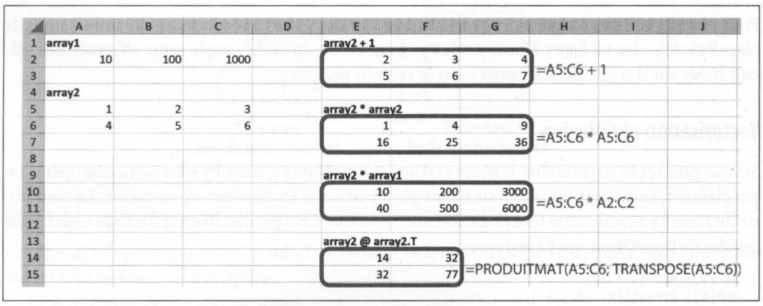


Figure 4.2 : Calculs fondés sur des matrices dans Excel.

Vous savez maintenant que les tableaux effectuent des opérations arithmétiques au niveau des éléments, mais comment pouvez-vous appliquer une fonction sur chaque élément d'un tableau ? C'est là où les fonctions universelles entrent en scène.

2 Si votre dernier cours d'algèbre linéaire remonte à un certain temps, vous pouvez ignorer cet exemple, car la multiplication matricielle n'est pas un sujet abordé dans ce livre.

Fonctions universelles (ufunc)

Les *fonctions universelles* (ufunc) s'appliquent sur chaque élément d'un tableau NumPy. Par exemple, si vous utilisez la fonction racine carrée standard de Python à partir du module `math` sur un tableau NumPy, vous obtiendrez une erreur :

```
In [12]: import math
In [13]: math.sqrt(array2)  # Provoque une erreur
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-5c37e8f41094> in <module>
----> 1 math.sqrt(array2)  # Provoque une erreur

TypeError: only size-1 arrays can be converted to Python scalars
```

Vous pouvez, bien sûr, écrire une boucle imbriquée pour obtenir la racine carrée de chaque élément, puis générer à nouveau un tableau NumPy à partir du résultat :

```
In [14]: np.array([[math.sqrt(i) for i in row] for row in array2])
Out[14]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

Cela fonctionnera dans les cas où NumPy n'offre pas de fonction universelle et où le tableau est suffisamment petit. Cependant, si NumPy a une ufunc, utilisez-la, car cela sera beaucoup plus rapide avec de grands tableaux, et, en outre, plus facile à saisir et à lire :

```
In [15]: np.sqrt(array2)
Out[15]: array([[1.        , 1.41421356, 1.73205081],
                [2.        , 2.23606798, 2.44948974]])
```

De plus, certaines fonctions universelles de NumPy, comme `sum`, sont disponibles en tant que méthodes de tableau : si vous souhaitez la somme de chaque colonne, procédez de la manière suivante :

```
In [16]: array2.sum(axis=0)  # Retourne un tableau unidimensionnel
Out[16]: array([5., 7., 9.])
```

L'argument `axis=0` fait référence à l'axe le long des lignes tandis que `axis=1` fait référence à l'axe le long des colonnes, comme cela est illustré à la Figure 4.1. Si l'on omet l'argument `axis`, on additionne l'ensemble du tableau :

```
In [17]: array2.sum()
Out[17]: 21.0
```

Vous rencontrerez d'autres fonctions universelles NumPy tout au long de ce livre, car elles peuvent être utilisées avec des DataFrames pandas.

Jusqu'à présent, nous avons toujours travaillé avec la totalité d'un tableau, mais dans la section suivante, je vais vous montrer comment manipuler des parties d'un tableau.

Création et manipulation de tableaux

Je commencerai cette section en lisant et en définissant des éléments spécifiques d'un tableau avant d'introduire quelques constructeurs de tableaux utiles, dont un pour créer des nombres pseudo-aléatoires que vous pouvez employer pour une simulation de Monte-Carlo. Je terminerai cette section en expliquant la différence entre une vue et une copie d'un tableau.

Lecture et définition d'éléments de tableau

Dans le chapitre précédent, je vous ai montré comment indexer et découper des listes pour accéder à des éléments spécifiques. Lorsque vous travaillez avec des listes imbriquées comme `matrice` (voir le premier exemple de ce chapitre), vous pouvez utiliser l'*indexation chaînée* : `matrice[0][0]` permet d'obtenir le premier élément de la première ligne. Toutefois, avec les tableaux NumPy, on fournit les arguments `index` et `slice` pour les deux dimensions dans une seule paire de crochets :

```
tableau_numpy[sélection_ligne, sélection_colonne]
```

Pour les tableaux unidimensionnels, la syntaxe est simplifiée en `tableau_numpy[sélection]`. Lorsque vous sélectionnez un seul élément, vous récupérerez un scalaire ; dans le cas contraire, vous récupérerez un tableau unidimensionnel ou bidimensionnel. N'oubliez pas que la notation de découpage utilise un indice de début (inclus) et un indice de fin (exclu) avec un caractère deux-points entre les deux, comme dans `début:fin`. En laissant de côté l'indice de début et de fin, vous vous retrouvez avec un deux-points, qui représente donc toutes les lignes ou toutes les colonnes d'un tableau à deux dimensions. J'ai illustré quelques exemples à la Figure 4.3, mais vous voudrez peut-être aussi revenir à la Figure 4.1, car les indices et les axes y sont étiquetés. Rappelez-vous qu'en découpant une colonne ou une ligne d'un tableau à deux dimensions, vous vous retrouvez avec un tableau unidimensionnel, et non pas avec une colonne ou un vecteur de ligne à deux dimensions.

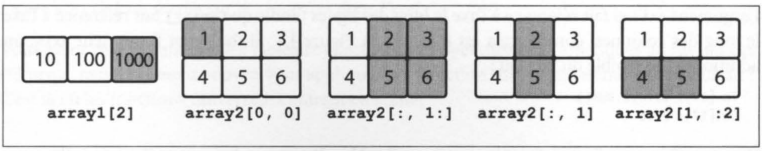


Figure 4.3 : Sélection d'éléments d'un tableau NumPy.

Vous pouvez tester les exemples illustrés à la Figure 4.3 en exécutant le code suivant :

```
In [18]: array1[2]  # Retourne un scalaire
Out[18]: 1000.0
In [19]: array2[0, 0]  # Retourne un scalaire
Out[19]: 1.0
In [20]: array2[:, 1:]  # Retourne un tableau bidimensionnel
Out[20]: array([[2., 3.],
                [5., 6.]])
In [21]: array2[:, 1]  # Retourne un tableau unidimensionnel
Out[21]: array([2., 5.])
In [22]: array2[1, :2]  # Retourne un tableau unidimensionnel
Out[22]: array([4., 5.])
```

Jusqu'à présent, j'ai créé des exemples de tableaux manuellement, c'est-à-dire en fournissant des nombres dans une liste. Cependant, NumPy offre également quelques fonctions utiles pour construire des tableaux.

Constructeurs de tableau utiles

NumPy offre également la possibilité de construire des tableaux qui faciliteront la création de DataFrames pandas, comme nous le verrons au Chapitre 5. Pour créer facilement des tableaux, on peut employer la fonction `arange`. Abréviation de *array range*, qui signifie plage de tableau, cette fonction est similaire à la fonction native `range` que nous avons rencontrée dans le chapitre précédent, à la différence que `arange` retourne un tableau NumPy. En la combinant avec `reshape`, cela permet de générer rapidement un tableau avec les dimensions souhaitées :

```
In [23]: np.arange(2 * 5).reshape(2, 5)  # 2 lignes, 5 colonnes
Out[23]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])
```

On a également souvent besoin, par exemple pour les simulations de Monte-Carlo, de générer des tableaux de nombres pseudo-aléatoires normalement distribués. NumPy facilite cette tâche :

```
In [24]: np.random.randn(2, 3)  # 2 lignes, 3 colonnes
Out[24]: array([[ 0.67323219,  1.19314728,  0.26022859],
                [-0.49209012, -0.79183758, -1.66199108]])
```

Parmi les constructeurs utiles à découvrir, on peut citer `np.ones` et `np.zeros` pour créer respectivement des tableaux avec des un et des zéros, et `np.eye` pour créer une matrice d'identité. Nous rencontrerons à nouveau certains de ces constructeurs dans le chapitre suivant, mais pour l'instant, nous allons aborder la différence entre une vue et une copie d'un tableau NumPy.

Vue ou copie ?

Les tableaux NumPy retournent des *vues* lorsque vous les découpez. Cela signifie que vous travaillez avec un sous-ensemble du tableau d'origine sans copier les données. La définition d'une valeur sur une vue modifie par conséquent le tableau d'origine :

```
In [25]: array2
Out[25]: array([[1., 2., 3.],
                [4., 5., 6.]])
In [26]: subset = array2[:, :2]
         subset
Out[26]: array([[1., 2.],
                [4., 5.]])
In [27]: subset[0, 0] = 1000
In [28]: subset
Out[28]: array([[1000.,    2.],
                [   4.,    5.]])
In [29]: array2
Out[29]: array([[1000.,    2.,    3.],
                [   4.,    5.,    6.]])
```

Si ce n'est pas ce que vous voulez, vous devrez modifier `In [26]` de la manière suivante :

```
subset = array2[:, :2].copy()
```

En travaillant sur une copie, vous ne modifierez pas le tableau d'origine.

Conclusion

Dans ce chapitre, je vous ai montré comment travailler avec les tableaux NumPy et ce qui se cache derrière des expressions telles que vectorisation et diffusion. Si vous faites abstraction de ces termes techniques, la manipulation des tableaux devrait être assez intuitive étant donné qu'ils suivent de très près la notation mathématique. Bien que NumPy soit une bibliothèque incroyablement puissante, il y a deux problèmes principaux lorsque vous souhaitez l'utiliser pour l'analyse de données :

- Les éléments d'un tableau NumPy doivent être du même type de données. Par exemple, cela signifie que vous ne pouvez effectuer aucune des opérations arithmétiques que nous avons employées dans ce chapitre lorsque votre tableau contient un mélange de texte et de nombres. Dès qu'il y a du texte, le tableau aura le type de données `object`, ce qui n'autorisera pas les opérations mathématiques.
- L'utilisation de tableaux NumPy pour l'analyse des données est complexe, car il est difficile de savoir à quoi chaque colonne ou chaque ligne correspond, puisque l'on sélectionne généralement des colonnes grâce à leur position, par exemple `array2[:, 1]`.

pandas, que nous découvrirons au prochain chapitre, a résolu ces problèmes en fournissant des structures de données plus intelligentes, qui s'appuient sur des tableaux NumPy.

CHAPITRE 5

Analyse de données avec pandas

Ce chapitre étudie pandas, la *bibliothèque d'analyse de données Python* ou, comme j'aime le dire, la feuille de calcul basée sur Python avec des super-pouvoirs. Cette bibliothèque est si puissante que certaines des entreprises avec lesquelles j'ai travaillé ont réussi à se débarrasser complètement d'Excel en le remplaçant par une combinaison de notebooks Jupyter et de pandas. En tant que lecteur de ce livre, cependant, je suppose que vous allez conserver Excel, auquel cas pandas servira d'interface pour analyser des données stockées dans des feuilles de calcul ou bien sur d'autres supports. pandas rend plus faciles, plus rapides et moins sujettes aux erreurs les tâches qui sont particulièrement pénibles dans Excel. Parmi ces tâches, on peut citer l'obtention de grands jeux de données à partir de sources externes et l'utilisation de statistiques, de séries chronologiques et de graphiques interactifs. Les fonctionnalités les plus puissantes de pandas sont la vectorisation et l'alignement des données. Comme nous l'avons déjà vu dans le chapitre précédent avec les tableaux NumPy, la vectorisation permet d'écrire du code concis basé sur un tableau, tandis que l'alignement des données permet de s'assurer qu'il n'y a pas d'incompatibilité de données lorsque l'on travaille avec plusieurs jeux de données.

Ce chapitre couvre l'ensemble du parcours d'analyse des données : il commence par le nettoyage et la préparation des données avant de vous montrer comment donner un sens à des jeux de données plus volumineux grâce à l'agrégation, les statistiques descriptives et la représentation sous une forme visuelle. À la fin du chapitre, nous verrons comment nous pouvons importer et exporter des données avec pandas. Mais tout d'abord, commençons par une introduction aux principales structures de données de pandas : les DataFrames et les séries.

DataFrames et séries

Les objets DataFrame et série (en anglais *Series*) sont les structures de données de base dans pandas. Dans cette section, je les présente en mettant l'accent sur les principaux composants d'un DataFrame : index, colonnes et données. Un *DataFrame* est similaire à un tableau NumPy

à deux dimensions, mais il est fourni avec des étiquettes de colonnes et de lignes et chaque colonne peut contenir des types de données différents. En extrayant une seule colonne ou une seule ligne d'un DataFrame, on obtient une série unidimensionnelle. Là encore, une *série* est similaire à un tableau NumPy unidimensionnel avec des étiquettes. Lorsque vous regardez la structure d'un DataFrame (voir la Figure 5.1), il ne faut pas beaucoup d'imagination pour voir que les DataFrames sont vos feuilles de calcul basées sur Python.

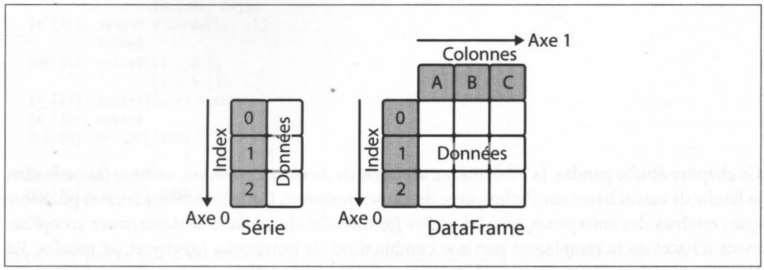


Figure 5.1 : Une série et un DataFrame pandas.

Pour vous montrer à quel point il est facile de passer d'une feuille de calcul à un DataFrame, considérez le tableau Excel suivant, illustré à la Figure 5.2, qui liste les participants d'un cours en ligne avec leur note. Vous trouverez le fichier correspondant *course_participants.xlsx* dans le dossier *xl* des exemples de code.

	A	B	C	D	E	F
1	user_id	name	age	country	score	continent
2	1001	Mark	55	Italy	4,5	Europe
3	1000	John	33	USA	6,7	America
4	1002	Tim	41	USA	3,9	America
5	1003	Jenny	12	Germany	9	Europe

Figure 5.2 : Fichier Excel course_participants.xlsx.

Pour que ce tableau Excel soit disponible en Python, commencez par importer pandas, puis utilisez sa fonction `read_excel`, qui renvoie un DataFrame :

```
In [1]: import pandas as pd
In [2]: pd.read_excel("xl/course_participants.xlsx")
```

Out[2]:

	user_id	name	age	country	score	continent
0	1001	Mark	55	Italy	4.5	Europe
1	1000	John	33	USA	6.7	America
2	1002	Tim	41	USA	3.9	America
3	1003	Jenny	12	Germany	9.0	Europe



Si vous exécutez `pd.read_excel` avec Python 3.9 ou une version supérieure, assurez-vous d'utiliser au moins pandas 1.2, sinon vous obtiendrez une erreur lors de la lecture des fichiers xlsx.

Si vous exécutez ce code dans un notebook Jupyter, le DataFrame sera correctement mis en forme et apparaîtra comme un tableau HTML, ce qui le rapproche encore plus de l'apparence d'un tableau Excel. L'intégralité du Chapitre 7 est consacrée à la lecture et à l'écriture des fichiers Excel avec pandas, et cet exemple n'est qu'une introduction pour vous montrer que les feuilles de calcul et les DataFrames sont, en effet, très similaires. Recréons maintenant ce DataFrame à partir de zéro, sans le lire à partir du fichier Excel : on peut créer un DataFrame en fournissant les données sous forme de liste imbriquée, ainsi que les valeurs des colonnes et des index :

```
In [3]: data=[["Mark", 55, "Italy", 4.5, "Europe"],
               ["John", 33, "USA", 6.7, "America"],
               ["Tim", 41, "USA", 3.9, "America"],
               ["Jenny", 12, "Germany", 9.0, "Europe"]]
        df = pd.DataFrame(data=data,
                          columns=["name", "age", "country",
                                   "score", "continent"],
                          index=[1001, 1000, 1002, 1003])
        df
```

Out[3]:

	name	age	country	score	continent
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

En appelant la méthode `info`, vous obtiendrez des informations de base, notamment le nombre de points de données et les types de données de chaque colonne :

```
In [4]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 1001 to 1003
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   name       4 non-null      object
 1   age        4 non-null      int64
 2   country    4 non-null      object
```

```
 3   score      4 non-null      float64
 4   continent  4 non-null      object
dtypes: float64(1), int64(1), object(3)
memory usage: 192.0+ bytes
```

Si vous êtes uniquement intéressé par le type de données de vos colonnes, exécutez `df.dtypes` à la place. Les colonnes avec des chaînes ou des types de données mixtes auront le type de données `object`.[1] Examinons maintenant de plus près les index et les colonnes d'un DataFrame.

Index

Les étiquettes de lignes d'un DataFrame sont appelées *index*. Si vous n'avez pas d'index significatif, laissez-le de côté lors de la construction d'un DataFrame. pandas crée alors automatiquement un index composé d'entiers et commençant à partir de zéro. Nous avons déjà vu cela dans le tout premier exemple lorsque nous avons lu un DataFrame à partir d'un fichier Excel. Les index permettent à pandas de rechercher des données plus rapidement et ils s'avèrent essentiels pour de nombreuses opérations courantes, par exemple la combinaison de deux DataFrames. On accède à l'objet index de la manière suivante :

```
In [5]: df.index
Out[5]: Int64Index([1001, 1000, 1002, 1003], dtype='int64')
```

Si cela a du sens, donnez un nom à l'index. Si l'on suit l'exemple du tableau dans Excel, on peut lui attribuer le nom `user_id` :

```
In [6]: df.index.name = "user_id"
        df
Out[6]:
```

user_id	name	age	country	score	continent
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

Contrairement à la clé primaire d'une base de données, l'index d'un DataFrame peut avoir des doublons, mais la recherche de valeurs peut être plus lente dans ce cas. Pour transformer un index en colonne normale, utilisez `reset_index`, et pour définir un nouvel index, `set_index`. Si vous ne souhaitez pas perdre l'index existant lors de la définition d'un nouvel index, veillez à le réinitialiser au préalable :

```
In [7]: # "reset_index" transforme l'index en une colonne, et remplace
        # l'index par l'index par défaut. Cela correspond au DataFrame
        # du début que nous avons chargé à partir d'Excel.
        df.reset_index()
```

1 pandas 1.0.0 a introduit un type de données `string` dédié pour faciliter certaines opérations et qu'elles soient plus cohérentes avec le texte. Comme ce type est encore expérimental, je ne l'utiliserai pas dans ce livre.

```
Out[7]:
```

	user_id	name	age	country	score	continent
0	1001	Mark	55	Italy	4.5	Europe
1	1000	John	33	USA	6.7	America
2	1002	Tim	41	USA	3.9	America
3	1003	Jenny	12	Germany	9.0	Europe

```
In [8]: # "reset_index" transforme "user_id" en une colonne normale
        # "set_index" transforme la colonne "name" en index
        df.reset_index().set_index("name")
Out[8]:
```

name	user_id	age	country	score	continent
Mark	1001	55	Italy	4.5	Europe
John	1000	33	USA	6.7	America
Tim	1002	41	USA	3.9	America
Jenny	1003	12	Germany	9.0	Europe

En faisant `df.reset_index().set_index("name")`, vous utilisez le *chaînage de méthode* : puisque `reset_index()` retourne un DataFrame, vous pouvez appeler directement une autre méthode de DataFrame sans avoir à écrire d'abord le résultat intermédiaire.



Chaque fois que vous appelez une méthode sur un DataFrame sous la forme `df.nom_méthode()`,vous récupérez une copie du DataFrame avec l'application de cette méthode, ce qui conserve intact le DataFrame d'origine. Nous venons de faire cela en appelant `df.reset_index()`. Si vous souhaitez modifier le DataFrame d'origine, vous devez affecter la valeur de retour à la variable d'origine de la manière suivante :

```
df = df.reset_index()
```

Puisque nous ne le faisons pas, cela signifie que notre variable `df` contient toujours ses données d'origine. Les exemples suivants appellent également des méthodes de DataFrame, c'est-à-dire qu'elles ne modifient pas le DataFrame d'origine.

Pour modifier l'index, utilisez la méthode `reindex` :

```
In [9]: df.reindex([999, 1000, 1001, 1004])
Out[9]:
```

user_id	name	age	country	score	continent
999	NaN	NaN	NaN	NaN	NaN
1000	John	33.0	USA	6.7	America
1001	Mark	55.0	Italy	4.5	Europe
1004	NaN	NaN	NaN	NaN	NaN

Il s'agit d'un premier exemple où l'on voit à l'œuvre l'alignement des données : `reindex` va prendre en compte toutes les lignes qui correspondent au nouvel index et va introduire des lignes avec des valeurs manquantes (`NaN`) là où il n'existe aucune information. Les éléments d'index que vous ne spécifiez pas seront supprimés. Je présenterai `NaN` en détail un peu plus loin dans ce chapitre. Enfin, pour trier un index, utilisez la méthode `sort_index` :

```
In [10]: df.sort_index()
Out[10]:
```

user_id	name	age	country	score	continent
1000	John	33	USA	6.7	America
1001	Mark	55	Italy	4.5	Europe
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

Si, au contraire, vous souhaitez trier les lignes sur une ou plusieurs colonnes, utilisez `sort_values` :

```
In [11]: df.sort_values(["continent", "age"])
Out[11]:
```

user_id	name	age	country	score	continent
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe
1001	Mark	55	Italy	4.5	Europe

L'exemple précédent montre comment trier d'abord sur la colonne `continent`, puis sur la colonne `age`. Si vous souhaitez trier sur une seule colonne, vous pouvez également fournir le nom de la colonne sous forme de chaîne :

```
df.sort_values("continent")
```

Nous venons d'étudier les bases du fonctionnement des index et nous allons à présent focaliser notre attention sur son équivalent horizontal, les colonnes de DataFrame.

Colonnes

Pour obtenir des informations sur les colonnes d'un DataFrame, exécutez le code suivant :

```
In [12]: df.columns
Out[12]: Index(['name', 'age', 'country', 'score', 'continent'], dtype='object')
```

Si vous ne fournissez aucun nom de colonne lors de la construction d'un DataFrame, pandas numérote les colonnes avec des entiers, en commençant à zéro. Avec les colonnes, cependant, cela est rarement souhaitable, car les colonnes représentent des variables et elles sont par conséquent faciles à nommer. On attribue un nom aux en-têtes de colonnes de la même manière que nous l'avons fait avec l'index :

```
In [13]: df.columns.name = "properties"
         df
Out[13]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

Si vous n'aimez pas les noms des colonnes, renommez-les :

```
In [14]: df.rename(columns={"name": "First Name", "age": "Age"})
Out[14]:
```

properties	First Name	Age	country	score	continent
user_id					
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

Si vous souhaitez supprimer des colonnes, utilisez la syntaxe suivante (l'exemple montre comment supprimer des colonnes et des index en même temps) :

```
In [15]: df.drop(columns=["name", "country"],
                 index=[1000, 1003])
Out[15]:
```

properties	age	score	continent
user_id			
1001	55	4.5	Europe
1002	41	3.9	America

Les colonnes et l'index d'un DataFrame sont tous deux représentés par un objet `Index`, si bien que vous pouvez transformer vos colonnes en lignes et *vice versa* en transposant votre DataFrame :

```
In [16]: df.T  # Raccourci pour df.transpose()
```

```
Out[16]:
```

user_id	1001	1000	1002	1003
properties				
name	Mark	John	Tim	Jenny
age	55	33	41	12
country	Italy	USA	USA	Germany
score	4.5	6.7	3.9	9
continent	Europe	America	America	Europe

Il convient de rappeler ici que notre DataFrame `df` est toujours inchangé, car nous n'avons jamais réaffecté le DataFrame retourné à partir des appels de méthode sur la variable `df` d'origine. Si vous souhaitez réorganiser les colonnes d'un DataFrame, vous pouvez utiliser la méthode `reindex` que nous avons utilisée avec l'index, mais la sélection des colonnes dans l'ordre souhaité est souvent plus intuitive :

```
In [17]: df.loc[:, ["continent", "country", "name", "age", "score"]]
Out[17]:
```

properties	continent	country	name	age	score
user_id					
1001	Europe	Italy	Mark	55	4.5
1000	America	USA	John	33	6.7
1002	America	USA	Tim	41	3.9
1003	Europe	Germany	Jenny	12	9.0

Ce dernier exemple nécessite des explications et tout ce qui concerne `loc` et le fonctionnement de la sélection des données est exposé dans la section suivante.

Manipulation des données

Les données du monde réel sont rarement servies sur un plateau d'argent, et avant de pouvoir les analyser, vous devez donc les nettoyer et les convertir sous une forme plus exploitable. Nous allons commencer cette section en examinant la manière de sélectionner des données à partir d'un DataFrame, de les modifier et de traiter les données manquantes et dupliquées. Nous effectuerons ensuite quelques calculs avec des DataFrames et nous verrons comment travailler avec les données textuelles. Pour conclure cette section, nous découvrirons quand pandas retourne une vue à la place d'une copie des données. Un certain nombre de concepts de cette section sont liés à ce que nous avons déjà vu avec les tableaux NumPy dans le dernier chapitre.

Sélection des données

Commençons par accéder aux données en nous basant sur leur étiquette et leur position avant d'examiner d'autres méthodes, notamment l'indexation booléenne et la sélection de données à l'aide d'un MultiIndex.

Sélection par étiquette

La façon la plus courante d'accéder aux données d'un DataFrame consiste à faire référence à ses étiquettes. Utilisez l'attribut `loc` (abréviation anglaise de *location*), qui signifie emplacement, pour spécifier les lignes et les colonnes que vous souhaitez récupérer :

```
df.loc[sélection_lignes, sélection_colonnes]
```

`loc` prend en charge la notation de découpage (*slicing*) et accepte donc un signe deux-points pour sélectionner toutes les lignes ou toutes les colonnes. En outre, vous pouvez fournir des listes avec des étiquettes ainsi qu'un nom de colonne ou de ligne unique. Examinez le Tableau 5.1 pour voir quelques exemples de la façon dont vous pouvez sélectionner différents sous-ensembles de notre exemple de DataFrame `df`.

Tableau 5.1 : Sélection de données par spécification de leur étiquette.

Sélection	Type de données de retour	Exemple
Valeur unique	Scalaire	`df.loc[1000, "country"]`
Une colonne (1d)	Série	`df.loc[:, "country"]`
Une colonne (2d)	DataFrame	`df.loc[:, ["country"]]`
Plusieurs colonnes	DataFrame	`df.loc[:, ["country", "age"]]`
Plage de colonnes	DataFrame	`df.loc[:, "name":"country"]`
Une ligne (1d)	Série	`df.loc[1000, :]`
Une ligne (2d)	DataFrame	`df.loc[[1000], :]`
Plusieurs lignes	DataFrame	`df.loc[[1003, 1000], :]`
Plage de lignes	DataFrame	`df.loc[1000:1002, :]`



L'utilisation de la notation de découpage avec des étiquettes est incohérente par rapport au fonctionnement de Python et de pandas, car l'étiquette de fin est incluse dans la sélection.

En appliquant les informations du Tableau 5.1, utilisons `loc` pour sélectionner des scalaires, des séries et des DataFrames :

```
In [18]: # L'emploi de scalaires à la fois pour la sélection
         # de lignes et de colonnes renvoie un scalaire
         df.loc[1001, "name"]
```

```
Out[18]: 'Mark'
In [19]: # L'emploi d'un scalaire pour la sélection d'une ligne
         # ou d'une colonne renvoie une série
         df.loc[[1001, 1002], "age"]
Out[19]: user_id
         1001    55
         1002    41
         Name: age, dtype: int64
In [20]: # La sélection de plusieurs lignes et colonnes renvoie un DataFrame
         df.loc[:1002, ["name", "country"]]
Out[20]:
```

properties	name	country
user_id		
1001	Mark	Italy
1000	John	USA
1002	Tim	USA

Il est important que vous compreniez la différence entre un DataFrame avec une ou plusieurs colonnes et une série : même avec une seule colonne, les DataFrames sont bidimensionnels, tandis que les séries sont unidimensionnelles. Un DataFrame et une série ont tous deux un index, mais seul le DataFrame a des en-têtes de colonnes. Lorsque vous sélectionnez une colonne en tant que série, l'en-tête de colonne devient le nom de la série. De nombreuses fonctions ou méthodes fonctionneront à la fois sur une série et un DataFrame, mais lorsque vous effectuez des calculs arithmétiques, le comportement diffère : avec les DataFrames, pandas aligne les données en fonction des en-têtes de colonnes (nous approfondirons cette question un peu plus loin dans ce chapitre).



Étant donné que la sélection des colonnes est une opération très courante, pandas offre un raccourci. Au lieu de :

```
df.loc[:, sélection_colonnes]
```

vous pouvez écrire :

```
df[sélection_colonnes]
```

Par exemple, `df["country"]` retourne une série à partir de notre exemple de DataFrame et `df[["name", "country"]]` retourne un DataFrame avec deux colonnes.

Sélection par position

La sélection d'un sous-ensemble d'un DataFrame en indiquant sa position correspond à ce que nous avons fait au début de ce chapitre avec les tableaux NumPy. Avec les DataFrames, cependant, vous devez utiliser l'attribut `iloc`, qui est l'abréviation de *integer location* (*localisation* par la fourniture d'un *entier* qui représente le numéro de ligne ou de colonne) :

```
df.iloc[sélection_lignes, sélection_colonnes]
```

Lorsque vous utilisez des découpages, vous traitez les intervalles semi-ouverts standard. Le Tableau 5.2 fournit les mêmes exemples que ceux du Tableau 5.1.

Tableau 5.2 : Sélection de données par spécification de leur position.

Sélection	Type de données de retour	Exemple
Valeur unique	Scalaire	`df.iloc[1, 2]`
Une colonne (1d)	Série	`df.iloc[:, 2]`
Une colonne (2d)	DataFrame	`df.iloc[:, [2]]`
Plusieurs colonnes	DataFrame	`df.iloc[:, [2, 1]]`
Plage de colonnes	DataFrame	`df.iloc[:, :3]`
Une ligne (1d)	Série	`df.iloc[1, :]`
Une ligne (2d)	DataFrame	`df.iloc[[1], :]`
Plusieurs lignes	DataFrame	`df.iloc[[3, 1], :]`
Plage de lignes	DataFrame	`df.iloc[1:3, :]`

Voici comment vous pouvez utiliser `iloc` (j'ai réemployé les mêmes exemples qu'avec `loc`) :

```
In [21]: df.iloc[0, 0]  # Retourne un scalaire
Out[21]: 'Mark'
In [22]: df.iloc[[0, 2], 1]  # Retourne un scalaire
Out[22]: user_id
         1001    55
         1002    41
         Name: age, dtype: int64
In [23]: df.iloc[:3, [0, 2]]  # Retourne un DataFrame
Out[23]:
```

	properties	name	country
user_id			
1001		Mark	Italy
1000		John	USA
1002		Tim	USA

La sélection de données par étiquette ou par position n'est pas le seul moyen d'accéder à un sous-ensemble d'un DataFrame, et on peut aussi employer l'indexation booléenne.

Sélection par indexation booléenne

L'indexation booléenne fait référence à la sélection de sous-ensembles d'un DataFrame à l'aide d'une série ou d'un DataFrame dont les données se composent uniquement de valeurs `True` ou `False`. Les séries booléennes sont utilisées pour sélectionner des colonnes et des lignes spécifiques d'un DataFrame, tandis que les DataFrames booléens sont utilisés pour sélectionner des valeurs spécifiques d'un DataFrame entier. Le plus souvent, vous utiliserez l'indexation

booléenne pour filtrer les lignes d'un DataFrame, et vous pouvez vous la représenter comme la fonctionnalité de filtrage automatique d'Excel. Par exemple, voici comment filtrer votre DataFrame afin qu'il n'affiche que les personnes qui vivent aux États-Unis et qui ont plus de 40 ans :

```
In [24]: tf = (df["age"] > 40) & (df["country"] == "USA")
         tf  # Cette série ne contient que des valeurs True/False
Out[24]: user_id
         1001    False
         1000    False
         1002     True
         1003    False
         dtype: bool
In [25]: df.loc[tf, :]
Out[25]:
```

user_id	properties	name	age	country	score	continent
1002		Tim	41	USA	3.9	America

Il y a deux choses que je dois expliquer ici. Tout d'abord, en raison de limitations techniques, vous ne pouvez pas utiliser les opérateurs booléens de Python que nous avons étudiés au Chapitre 3 avec des DataFrames. À la place, vous devez utiliser les symboles listés dans le Tableau 5.3.

Tableau 5.3 : Opérateurs booléens.

Types de données Python de base	
and	&
or	|
not	~

Deuxièmement, si vous avez plusieurs conditions, assurez-vous de placer chaque expression booléenne entre parenthèses, afin que la priorité des opérateurs ne soit pas une gêne : par exemple, & a une priorité d'opérateur supérieure à ==. Par conséquent, sans parenthèses, l'expression de l'exemple précédent serait interprétée de la manière suivante :

```
df["age"] > (40 &df["country"]) == "USA"
```

Si vous souhaitez filtrer l'index, vous pouvez vous y référer sous le nom de `df.index` :

```
In [26]: df.loc[df.index > 1001, :]
Out[26]:
```

user_id	properties	name	age	country	score	continent
1002		Tim	41	USA	3.9	America
1003		Jenny	12	Germany	9.0	Europe

Là où vous utiliseriez l'opérateur `in` avec des structures de données Python de base, comme des listes, employez `isin` avec une série. Voici comment filtrer votre DataFrame pour les participants en provenance d'Italie et d'Allemagne :

```
In [27]: df.loc[df["country"].isin(["Italy", "Germany"]), :]
Out[27]:
```

user_id	properties	name	age	country	score	continent
1001		Mark	55	Italy	4.5	Europe
1003		Jenny	12	Germany	9.0	Europe

Alors que vous utilisez `loc` pour fournir une série booléenne, les DataFrames offrent une syntaxe spéciale sans `loc` pour sélectionner des valeurs d'un DataFrame complet de booléens :

```
df[boolean_df]
```

Ceci est particulièrement utile si vous avez des DataFrames qui se composent uniquement de nombres. Fournir un DataFrame de booléens retourne le DataFrame avec `NaN` partout où le DataFrame booléen est égal à `False`. Encore une fois, une discussion plus détaillée sur `NaN` suivra sous peu. Commençons par créer un nouvel exemple de DataFrame appelé `rainfall` (précipitations de pluie) qui se compose uniquement de nombres :

```
In [28]: # Il peut s'agir de cumuls de pluie annuels en millimètres
         rainfall = pd.DataFrame(data={"City 1": [300.1, 100.2],
                                       "City 2": [400.3, 300.4],
                                       "City 3": [1000.5, 1100.6]})
         rainfall
Out[28]:
```

	City 1	City 2	City 3
0	300.1	400.3	1000.5
1	100.2	300.4	1100.6

```
In [29]: rainfall < 400
Out[29]:
```

	City 1	City 2	City 3
0	True	False	False
1	True	True	False

```
In [30]: rainfall[rainfall < 400]
Out[30]:
```

	City 1	City 2	City 3
0	300.1	NaN	NaN
1	100.2	300.4	NaN

Notez que dans cet exemple, j'ai utilisé un dictionnaire pour construire un nouveau DataFrame, ce qui est souvent pratique si les données existent déjà sous cette forme. L'exploitation de valeurs booléennes de cette manière-là est souvent utilisée pour filtrer des valeurs spécifiques telles que des valeurs aberrantes.

Pour conclure cette partie consacrée à la sélection des données, je vais introduire un type spécial d'index appelé MultiIndex.

Sélection à l'aide d'un MultiIndex

Un *MultiIndex* est un index avec plusieurs niveaux. Il permet de regrouper hiérarchiquement vos données et donne un accès facile aux sous-ensembles. Par exemple, si vous définissez l'index de notre exemple de DataFrame `df` sur une combinaison de `continent` et de `country`, vous pouvez facilement sélectionner toutes les lignes égales à un certain continent :

```
In [31]: # Un MultiIndex doit être trié
         df_multi = df.reset_index().set_index(["continent", "country"])
         df_multi = df_multi.sort_index()
         df_multi
Out[31]:
```

		properties	user_id	name	age	score
continent	country					
America	USA		1000	John	33	6.7
	USA		1002	Tim	41	3.9
Europe	Germany		1003	Jenny	12	9.0
	Italy		1001	Mark	55	4.5

```
In [32]: df_multi.loc["Europe", :]
Out[32]:
```

	properties	user_id	name	age	score
country					
Germany		1003	Jenny	12	9.0
Italy		1001	Mark	55	4.5

Notez que pandas embellit la sortie d'un MultiIndex en ne répétant pas le niveau d'index le plus à gauche (les continents) pour chaque ligne. Au lieu de cela, il n'affiche le continent que lorsque ce dernier change. La sélection sur plusieurs niveaux d'index s'effectue en fournissant un tuple :

```
In [33]: df_multi.loc[("Europe", "Italy"), :]
Out[33]:
```

		properties	user_id	name	age	score
continent	country					
Europe	Italy		1001	Mark	55	4.5

Si vous souhaitez réinitialiser de manière sélective une partie d'un MultiIndex, fournissez le niveau en tant qu'argument. Zéro représente la première colonne à partir de la gauche :

```
In [34]: df_multi.reset_index(level=0)
```

```
Out[34]:
```

properties	continent	user_id	name	age	score
country					
USA	America	1000	John	33	6.7
USA	America	1002	Tim	41	3.9
Germany	Europe	1003	Jenny	12	9.0
Italy	Europe	1001	Mark	55	4.5

Nous n'allons pas créer manuellement un MultiIndex dans ce livre, mais il existe certaines opérations comme `groupby`, qui provoquent la création par pandas d'un DataFrame avec un MultiIndex, et il est donc bon de savoir de quoi il s'agit. Nous évoquerons `groupby` plus loin dans ce chapitre.

Maintenant que vous connaissez différentes façons de *sélectionner* des données, il est temps d'apprendre à les *modifier*.

Modification des données

Le moyen le plus simple de modifier les données d'un DataFrame consiste à affecter des valeurs à certains éléments à l'aide des attributs `loc` ou `iloc`. Nous commencerons par étudier ce point avant de nous tourner vers d'autres façons de manipuler les DataFrames existants, comme le replacement des valeurs et l'ajout de nouvelles colonnes.

Modification des données par étiquette ou par position

Comme indiqué précédemment dans ce chapitre, lorsque vous appelez des méthodes de DataFrame telles que `df.reset_index()`, la méthode sera toujours appliquée à une copie, ce qui ne modifie pas le DataFrame d'origine. Toutefois, l'affectation de valeurs *via* les attributs `loc` et `iloc` modifie le DataFrame original. Puisque je veux laisser notre DataFrame `df` intact, je travaille ici avec une copie que j'appelle `df2`. Si vous souhaitez modifier une valeur unique, procédez de la manière suivante :

```
In [35]: # On copie d'abord le DataFrame pour laisser intact l'original
         df2 = df.copy()
In [36]: df2.loc[1000, "name"] = "JOHN"
         df2
Out[36]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.5	Europe
1000	JOHN	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

Vous pouvez également modifier plusieurs valeurs en même temps. Afin de modifier les notes des utilisateurs ayant l'ID 1000 et 1001, vous pouvez employer une liste :

```
In [37]: df2.loc[[1000, 1001], "score"] = [3, 4]
         df2
Out[37]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.0	Europe
1000	JOHN	33	USA	3.0	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe

La modification des données par position avec `iloc` fonctionne de la même manière. Passons maintenant à la façon dont on modifie les données à l'aide de l'indexation booléenne.

Modification des données par indexation booléenne

L'indexation booléenne, que nous avons utilisée pour filtrer les lignes, peut également être employée pour assigner des valeurs dans un DataFrame. Imaginez que vous ayez besoin d'anonymiser tous les noms des personnes de moins de 20 ans ou demeurant aux États-Unis :

```
In [38]: tf = (df2["age"] < 20) | (df2["country"] == "USA")
         df2.loc[tf, "name"] = "xxx"
         df2
Out[38]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.0	Europe
1000	xxx	33	USA	3.0	America
1002	xxx	41	USA	3.9	America
1003	xxx	12	Germany	9.0	Europe

Parfois, vous avez un jeu de données où vous devez remplacer systématiquement certaines valeurs, sans tenir compte de la colonne où elles se trouvent. Dans ce cas, utilisez à nouveau la syntaxe spéciale et fournissez à l'ensemble du DataFrame des booléens de la manière suivante (l'exemple utilise à nouveau le DataFrame `rainfall`) :

```
In [39]: # On copie d'abord le DataFrame pour laisser intact l'original
         rainfall2 = rainfall.copy()
         rainfall2
Out[39]:
```

	City 1	City 2	City 3
0	300.1	400.3	1000.5
1	100.2	300.4	1100.6

```
In [40]: # Modifie les valeurs à 0 quand elles sont inférieures à 400
         rainfall2[rainfall2 < 400] = 0
         rainfall2
Out[40]:
```

	City 1	City 2	City 3
0	0.0	400.3	1000.5
1	0.0	0.0	1100.6

Si vous voulez juste remplacer une valeur par une autre, il existe un moyen plus simple de le faire, comme nous allons le voir immédiatement.

Modification des données en remplaçant leurs valeurs

Si vous souhaitez remplacer une certaine valeur sur l'ensemble de votre DataFrame ou des colonnes sélectionnées, utilisez la méthode `replace` :

```
In [41]: df2.replace("USA", "U.S.")
Out[41]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.0	Europe
1000	xxx	33	U.S.	3.0	America
1002	xxx	41	U.S.	3.9	America
1003	xxx	12	Germany	9.0	Europe

Si, au lieu de cela, vous souhaitez uniquement agir sur la colonne `country`, vous pouvez utiliser cette syntaxe à la place :

```
df2.replace({"country": {"USA": "U.S."}})
```

Dans cet exemple, comme `USA` n'apparaît que dans la colonne `country`, on obtient le même résultat que précédemment. Pour conclure cette section, voyons comment ajouter des colonnes supplémentaires à un DataFrame.

Modification des données en ajoutant une nouvelle colonne

Pour ajouter une nouvelle colonne à un DataFrame, affectez des valeurs à un nouveau nom de colonne. Par exemple, vous pouvez ajouter une nouvelle colonne à un DataFrame à l'aide d'un scalaire ou d'une liste :

```
In [42]: df2.loc[:, "discount"] = 0
         df2.loc[:, "price"] = [49.9, 49.9, 99.9, 99.9]
         df2
```

Out[42]:

properties	name	age	country	score	continent	discount	price
user_id							
1001	Mark	55	Italy	4.0	Europe	0	49.9
1000	xxx	33	USA	3.0	America	0	49.9
1002	xxx	41	USA	3.9	America	0	99.9
1003	xxx	12	Germany	9.0	Europe	0	99.9

L'ajout d'une nouvelle colonne implique souvent des calculs vectorisés :

```
In [43]: df2 = df.copy()  # On commence par copier le DataFrame
         df2.loc[:, "birth year"] = 2021 - df2["age"]
         df2
```

Out[43]:

properties	name	age	country	score	continent	birth year
user_id						
1001	Mark	55	Italy	4.5	Europe	1966
1000	John	33	USA	6.7	America	1988
1002	Tim	41	USA	3.9	America	1980
1003	Jenny	12	Germany	9.0	Europe	2009

Je vous en dirai plus sur le calcul avec des DataFrames dans un instant, mais avant d'y arriver, vous vous rappelez que j'ai déjà utilisé `NaN` plusieurs fois. La section suivante vous donnera enfin des explications au sujet des données manquantes.

Données manquantes

Les données manquantes peuvent constituer un problème, car elles peuvent biaiser les résultats de votre analyse de données, diminuant ainsi la robustesse de vos conclusions. Néanmoins, il est très courant d'avoir des lacunes dans les jeux de données que vous devrez gérer. Dans Excel, vous devez généralement faire face à des cellules vides ou à des erreurs `#N/A`, mais pandas utilise `np.nan` de NumPy pour les données manquantes, affichées sous la forme `NaN` : il s'agit de l'abréviation de *Not-a-Number* (pas un nombre), appellation standard pour indiquer qu'une valeur qui est censée être un nombre à virgule flottante n'est en fait pas un nombre. Pour les valeurs d'horodatages, on utilise à la place `pd.NaT`, et pour le texte, pandas emploie `None`. En utilisant `None` ou `np.nan`, vous pouvez insérer des valeurs manquantes dans un DataFrame :

```
In [44]: df2 = df.copy() # On commence par copier le DataFrame
         df2.loc[1000, "score"] = None
         df2.loc[1003, :] = None
         df2
```

```
Out[44]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55.0	Italy	4.5	Europe
1000	John	33.0	USA	NaN	America
1002	Tim	41.0	USA	3.9	America
1003	None	NaN	None	NaN	None

Pour nettoyer un DataFrame, on souhaite souvent supprimer les lignes avec des données manquantes et c'est aussi simple que cela :

```
In [45]: df2.dropna()
Out[45]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55.0	Italy	4.5	Europe
1002	Tim	41.0	USA	3.9	America

Toutefois, si vous souhaitez uniquement supprimer les lignes où *toutes* les valeurs sont manquantes, utilisez le paramètre `how` :

```
In [46]: df2.dropna(how="all")
Out[46]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55.0	Italy	4.5	Europe
1000	John	33.0	USA	NaN	America
1002	Tim	41.0	USA	3.9	America

Pour obtenir une série ou un DataFrame booléen selon qu'il contient une valeur manquante, utilisez `isna` :

```
In [47]: df2.isna()
Out[47]:
```

properties	name	age	country	score	continent
user_id					
1001	False	False	False	False	False
1000	False	False	False	True	False
1002	False	False	False	False	False
1003	True	True	True	True	True

Pour remplir les valeurs manquantes, utilisez `fillna` ; par exemple, pour remplacer `NaN` dans la colonne `score` par sa moyenne (je vais présenter les statistiques descriptives comme la moyenne (`mean`) dans un instant) :

```
In [48]: df2.fillna({"score": df2["score"].mean()})
Out[48]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55.0	Italy	4.5	Europe
1000	John	33.0	USA	4.2	America
1002	Tim	41.0	USA	3.9	America
1003	None	NaN	None	4.2	None

Les données manquantes ne sont pas le seul cas de figure nous obligeant à nettoyer notre jeu de données et il en va de même pour les données en double.

Données en double

Comme les données manquantes, les doublons ont un impact négatif sur la fiabilité de votre analyse. Pour vous débarrasser des lignes en double, utilisez la méthode `drop_duplicates`. Si vous le souhaitez, vous pouvez fournir un sous-ensemble des colonnes en tant qu'argument :

```
In [49]: df.drop_duplicates(["country", "continent"])
Out[49]:
```

properties	name	age	country	score	continent
user_id					
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1003	Jenny	12	Germany	9.0	Europe

Par défaut, cela conserve la première occurrence. Pour savoir si une certaine colonne contient des doublons ou pour obtenir ses valeurs uniques, utilisez les deux commandes suivantes (utilisez `df.index` au lieu de `df["country"]` si vous souhaitez exécuter ceci sur l'index) :

```
In [50]: df["country"].is_unique
Out[50]: False
In [51]: df["country"].unique()
Out[51]: array(['Italy', 'USA', 'Germany'], dtype=object)
```

Et enfin, pour savoir quelles lignes sont des doublons, utilisez la méthode `duplicated`, qui retourne une série booléenne : par défaut, elle utilise le paramètre `keep="first"`, qui conserve la première occurrence et ne marque que les doublons avec `True`. En définissant le paramètre `keep=False`, `True` est retourné pour toutes les lignes, y compris la première occurrence, ce qui facilite l'identification des doublons dans un DataFrame. Dans l'exemple suivant, nous recherchons les doublons de la colonne `country`, mais en réalité, on regarde souvent l'index ou des lignes entières. Dans ce cas, vous devrez utiliser `df.index.duplicated()` ou `df.duplicated()` à la place :

```
In [52]: # Par défaut, seuls les doublons renvoient True,
         # à l'exception de la première occurrence
         df["country"].duplicated()
Out[52]: user_id
         1001    False
         1000    False
         1002     True
         1003    False
         Name: country, dtype: bool
In [53]: # Pour avoir toutes les lignes où "country" est en double,
         # utilisez keep=False
         df.loc[df["country"].duplicated(keep=False), :]
Out[53]:
```

user_id	properties	name	age	country	score	continent
1000		John	33	USA	6.7	America
1002		Tim	41	USA	3.9	America

Une fois que vous avez nettoyé vos DataFrames en supprimant les données manquantes et en double, vous souhaiterez souvent effectuer certaines opérations arithmétiques.

Opérations arithmétiques

Comme les tableaux NumPy, les DataFrames et les séries utilisent la vectorisation. Par exemple, pour ajouter un nombre à chaque valeur du DataFrame `rainfall`, procédez simplement de la manière suivante :

```
In [54]: rainfall
Out[54]:
```

	City 1	City 2	City 3
0	300.1	400.3	1000.5
1	100.2	300.4	1100.6

```
In [55]: rainfall + 100
Out[55]:
```

	City 1	City 2	City 3
0	400.1	500.3	1100.5
1	200.2	400.4	1200.6

Cependant, la véritable puissance de pandas réside dans son mécanisme d'*alignement des données* automatique : lorsque vous utilisez des opérateurs arithmétiques avec plusieurs DataFrames, pandas les aligne automatiquement en fonction de leurs index de colonne et de ligne. Créons un deuxième DataFrame partageant certaines étiquettes de lignes et de colonnes avec le premier. Nous allons ensuite les additionner :

```
In [56]: more_rainfall = pd.DataFrame(data=[[100, 200], [300, 400]],
                                      index=[1, 2],
                                      columns=["City 1", "City 4"])
         more_rainfall
Out[56]:
```

	City 1	City 4
1	100	200
2	300	400

```
In [57]: rainfall + more_rainfall
Out[57]:
```

	City 1	City 2	City 3	City 4
0	NaN	NaN	NaN	NaN
1	200.2	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN

L'index et les colonnes du DataFrame résultant sont l'union des index et des colonnes des deux DataFrames : les champs qui ont une valeur dans les deux DataFrames affichent la somme, tandis que le reste du DataFrame affiche `NaN`. Vous devrez peut-être vous y habituer si vous venez d'Excel, où les cellules vides sont automatiquement transformées en zéros lorsque vous les utilisez dans des opérations arithmétiques. Pour obtenir le même comportement que dans Excel, utilisez la méthode `add` avec le paramètre `fill_value` pour remplacer les valeurs `NaN` par des zéros :

```
In [58]: rainfall.add(more_rainfall, fill_value=0)
Out[58]:
```

	City 1	City 2	City 3	City 4
0	300.1	400.3	1000.5	NaN
1	200.2	300.4	1100.6	200.0
2	300.0	NaN	NaN	400.0

Cela fonctionne de la même manière pour les autres opérateurs arithmétiques listés dans le Tableau 5.4.

Tableau 5.4 : Opérateurs arithmétiques.

Opérateur	Méthode
*	mul
+	add
-	sub
/	div
**	pow

Lorsque vous avez un DataFrame et une série dans votre calcul, par défaut, la série est diffusée en prenant en compte l'index :

```
In [59]: # Série extraite d'une ligne
         rainfall.loc[1, :]
Out[59]: City 1     100.2
         City 2     300.4
         City 3    1100.6
         Name: 1, dtype: float64
In [60]: rainfall + rainfall.loc[1, :]
Out[60]:
```

	City 1	City 2	City 3
0	400.3	700.7	2101.1
1	200.4	600.8	2201.2

Par conséquent, pour ajouter une série en prenant en compte une colonne, vous devez utiliser la méthode `add` avec un argument `axis` explicite :

```
In [61]: # Série extraite d'une colonne
         rainfall.loc[:, "City 2"]
Out[61]: 0    400.3
         1    300.4
         Name: City 2, dtype: float64
In [62]: rainfall.add(rainfall.loc[:, "City 2"], axis=0)
Out[62]:
```

	City 1	City 2	City 3
0	700.4	800.6	1400.8
1	400.6	600.8	1401.0

Bien que cette section traite des DataFrames avec des nombres et de leur comportement lors d'opérations arithmétiques, la section suivante présente les possibilités de manipuler du texte dans les DataFrames.

Utilisation des colonnes de texte

Comme nous l'avons vu au début de ce chapitre, les colonnes avec du texte ou des types de données mixtes ont le type de données `object`. Pour effectuer des opérations sur des colonnes avec des chaînes de caractères, utilisez l'attribut `str` qui donne accès aux méthodes de chaînes de caractères de Python. Nous avons déjà rencontré quelques-unes de ces méthodes au Chapitre 3, mais il n'est peut-être pas inutile de jeter un coup d'œil à la documentation Python consacrée aux méthodes de chaînes de caractères disponibles[2]. Par exemple, pour supprimer les espaces au début et à la fin d'une chaîne, utilisez la méthode `strip` ; pour mettre toutes les initiales en majuscules, il existe la méthode `capitalize`. En les enchaînant, vous allez nettoyer les colonnes de texte incorrectes qui sont souvent le résultat de la saisie manuelle des données :

2 https://docs.python.org/fr/3.8/library/stdtypes.html#string-methods

```
In [63]: # On crée un nouveau DataFrame
         users = pd.DataFrame(data=[" mArk ", "JOHN  ", "Tim", " jenny"],
                              columns=["name"])
         users
Out[63]:
```

	name
0	mArk
1	JOHN
2	Tim
3	jenny

```
In [64]: users_cleaned = users.loc[:, "name"].str.strip().str.capitalize()
         users_cleaned
Out[64]: 0     Mark
         1     John
         2      Tim
         3    Jenny
         Name: name, dtype: object
```

Voici le code pour rechercher tous les noms qui commencent par un « J » :

```
In [65]: users_cleaned.str.startswith("J")
Out[65]: 0    False
         1     True
         2    False
         3     True
         Name: name, dtype: bool
```

Les méthodes de chaînes de caractères sont faciles à utiliser, mais parfois vous devrez peut-être manipuler un DataFrame d'une manière peu intuitive. Dans ce cas, il est préférable de créer sa propre fonction et de l'appliquer au DataFrame.

Application d'une fonction

Les DataFrames offrent la méthode `applymap`, qui applique une fonction à chaque élément individuel, ce qui est utile s'il n'y a pas de fonction universelle NumPy disponible. Par exemple, il n'y a pas d'ufunc pour la mise en forme de chaînes de caractères, si bien que nous pouvons formater chaque élément d'un DataFrame de la manière suivante :

```
In [66]: rainfall
Out[66]:
```

	City 1	City 2	City 3
0	300.1	400.3	1000.5
1	100.2	300.4	1100.6

```
In [67]: def format_string(x):
             return f"{x:,.2f}"
```

```
In [68]: # Notez que l'on passe la fonction sans l'appeler,
         # c'est-à-dire, format_string et non pas format_string()
         rainfall.applymap(format_string)
Out[68]:
```

	City 1	City 2	City 3
0	300.10	400.30	1,000.50
1	100.20	300.40	1,100.60

Expliquons ce code : la chaîne de formatage (f-string) suivante retourne x sous forme de chaîne : `f"{x}"`. Pour ajouter une mise en forme, ajoutez un signe deux-points à la variable suivi de la chaîne de mise en forme `,.2f`. La virgule est le séparateur des milliers et `.2f` signifie que l'on utilise deux chiffres après la virgule qui est en fait un point décimal. Pour obtenir plus de détails sur la façon de mettre en forme des chaînes, reportez-vous au Mini-langage de spécification de format, qui fait partie de la documentation de Python[3].

Pour ce genre de cas d'utilisation, les *expressions lambda* (voir encadré) sont largement utilisées, car elles permettent d'écrire la même chose sur une seule ligne, sans avoir à définir une fonction distincte. Avec une expression lambda, nous pouvons réécrire l'exemple précédent de la manière suivante :

```
In [69]: rainfall.applymap(lambda x: f"{x:,.2f}")
Out[69]:
```

	City 1	City 2	City 3
0	300.10	400.30	1,000.50
1	100.20	300.40	1,100.60

Expressions lambda

Python permet de définir une fonction sur une seule ligne avec des *expressions lambda*. Les expressions lambda sont des fonctions anonymes, ce qui signifie qu'il s'agit d'une fonction sans nom. Considérez cette fonction :

```
def nom_fonction(arg1, arg2, ...):
    return valeur_de_retour
```

Cette fonction peut être réécrite en tant qu'expression lambda comme ceci :

```
lambda arg1, arg2, ...: valeur_de_retour
```

En substance, vous remplacez `def` par `lambda`, vous laissez de côté le mot-clé `return` et le nom de la fonction, et vous mettez tout sur une seule ligne. Comme nous l'avons vu avec la méthode `applymap`, cela peut être très pratique dans ce cas, car nous n'avons pas besoin de définir une fonction pour une chose qui n'est utilisée qu'une seule fois.

3 https://docs.python.org/fr/3/library/string.html#format-specification-mini-language

Nous avons à présent étudié toutes les méthodes importantes de manipulation de données, mais avant de passer à autre chose, il est essentiel de comprendre quand pandas utilise une vue d'un DataFrame et quand il utilise une copie.

Vue ou copie ?

Vous vous souvenez peut-être qu'au chapitre précédent, nous avons remarqué que le découpage des tableaux NumPy renvoie une vue. Avec les DataFrames, c'est malheureusement plus compliqué : il n'est pas toujours facile de déterminer si `loc` et `iloc` retournent des vues ou des copies, ce qui crée beaucoup de confusion. Comme il y a une grande différence entre la modification d'une vue ou la copie d'un DataFrame, pandas déclenche régulièrement l'avertissement suivant lorsqu'il pense que vous modifiez les données de manière involontaire : `SettingWithCopyWarning`. Pour contourner cet avertissement plutôt énigmatique, voici quelques conseils :

- Modifiez les valeurs sur le DataFrame d'origine, et non pas sur un DataFrame qui a été découpé à partir d'un autre DataFrame.
- Si vous souhaitez disposer d'un DataFrame indépendant après le découpage, effectuez une copie explicite :

```
selection = df.loc[:, ["country", "continent"]].copy()
```

Bien que les choses soient compliquées avec `loc` et `iloc`, il convient de rappeler que *toutes* les méthodes de DataFrame telles que `df.dropna()` ou `df.sort_values("nom_colonne")` retournent *toujours* une copie.

Jusqu'à présent, nous avons principalement travaillé avec un seul DataFrame à la fois, mais la section suivante montre différentes façons de combiner plusieurs DataFrames en un seul, tâche très courante pour laquelle pandas offre des outils puissants.

Combinaison de DataFrames

La combinaison de différents jeux de données dans Excel peut être une tâche fastidieuse et implique généralement de nombreuses formules `RECHERCHEV`. Heureusement, la combinaison de DataFrames est l'une des fonctionnalités phares de pandas où ses capacités d'alignement des données vous faciliteront la vie, réduisant ainsi considérablement la possibilité d'introduire des erreurs. La combinaison et la fusion de DataFrames peuvent être effectuées de différentes manières ; cette section examine uniquement les cas les plus courants utilisant `concat`, `join` et `merge`. Bien qu'elles se recouvrent, chaque fonction permet d'effectuer une tâche spécifique très simplement. Je commencerai par la fonction `concat`, puis expliquerai les différentes options de `join`, et conclurai en évoquant `merge`, la fonction la plus générique des trois.

Concaténation

Pour simplement coller plusieurs DataFrames ensemble, la fonction `concat` est votre meilleur ami. Comme vous pouvez le constater d'après le nom de la fonction, ce processus s'appelle techniquement une *concaténation*. Par défaut, `concat` colle les DataFrames en se basant sur les lignes et aligne automatiquement les colonnes. Dans l'exemple suivant, je crée un autre DataFrame, `more_users`, et je l'attache au bas de notre exemple de DataFrame `df` :

```
In [70]: data=[[15, "France", 4.1, "Becky"],
               [44, "Canada", 6.1, "Leanne"]]
         more_users = pd.DataFrame(data=data,
                                   columns=["age", "country", "score", "name"],
                                   index=[1000, 1011])
         more_users
Out[70]:
```

	age	country	score	name
1000	15	France	4.1	Becky
1011	44	Canada	6.1	Leanne

```
In [71]: pd.concat([df, more_users], axis=0)
Out[71]:
```

	name	age	country	score	continent
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America
1002	Tim	41	USA	3.9	America
1003	Jenny	12	Germany	9.0	Europe
1000	Becky	15	France	4.1	NaN
1011	Leanne	44	Canada	6.1	NaN

Vous noterez que vous avez à présent des éléments d'index en double, car `concat` colle les données sur l'axe indiqué (les lignes) et aligne uniquement les données sur l'autre axe (les colonnes), faisant ainsi correspondre automatiquement les noms de colonnes, même s'ils ne sont pas dans le même ordre dans les deux DataFrames. Si vous souhaitez coller deux DataFrames ensemble en vous basant sur les colonnes, définissez `axis` à `1` :

```
In [72]: data=[[3, 4],
               [5, 6]]
         more_categories = pd.DataFrame(data=data,
                                        columns=["quizzes", "logins"],
                                        index=[1000, 2000])
         more_categories
Out[72]:
```

	quizzes	logins
1000	3	4
2000	5	6

```
In [73]: pd.concat([df, more_categories], axis=1)
Out[73]:
```

	name	age	country	score	continent	quizzes	logins
1000	John	33.0	USA	6.7	America	3.0	4.0
1001	Mark	55.0	Italy	4.5	Europe	NaN	NaN
1002	Tim	41.0	USA	3.9	America	NaN	NaN
1003	Jenny	12.0	Germany	9.0	Europe	NaN	NaN
2000	NaN	NaN	NaN	NaN	NaN	5.0	6.0

La fonction `concat` possède une caractéristique spéciale qui est très utile : elle accepte plus de deux DataFrames. Nous l'utiliserons dans le chapitre suivant pour créer un seul DataFrame à partir de plusieurs fichiers CSV :

```
pd.concat([df1, df2, df3, ...])
```

En revanche, la jointure et la fusion ne fonctionnent qu'avec deux DataFrames, comme nous le verrons plus tard.

Jointure et fusion

Lorsque vous faites une *jointure* de deux DataFrames, vous combinez les colonnes de chaque DataFrame dans un nouveau DataFrame, tout en décidant de ce qui se passe au niveau des lignes, en vous appuyant sur la théorie des ensembles. Si vous avez déjà travaillé avec des bases de données relationnelles, il s'agit du même concept que la clause `JOIN` dans les requêtes SQL. La Figure 5.3 montre comment les quatre types de jointures (à savoir jointures interne, gauche, droite et externe) fonctionnent, en prenant l'exemple de deux DataFrames, `df1` et `df2`.

Avec `join`, pandas utilise les index des deux DataFrames pour aligner les lignes. Une *jointure interne* (*inner*) retourne un DataFrame avec uniquement les lignes où les index se recoupent. Une *jointure gauche* (*left*) prend toutes les lignes du DataFrame de gauche `df1` et fait correspondre les lignes du DataFrame de droite `df2` en se basant sur l'index. Là où `df2` n'a pas de ligne correspondante, pandas remplira les valeurs avec `NaN`. La jointure gauche correspond à la fonction `RECHERCHEV` dans Excel. La *jointure droite* (*right*) prend toutes les lignes de la table de droite `df2` et les fait correspondre aux lignes de `df1` en se basant sur l'index. Et enfin, la *jointure externe*, dont le nom exact est *jointure externe complète* (*full outer*), prend l'union des index des deux DataFrames et fait correspondre les valeurs où elle le peut. Le Tableau 5.5 est l'équivalent de la Figure 5.3, mais sous une forme textuelle.

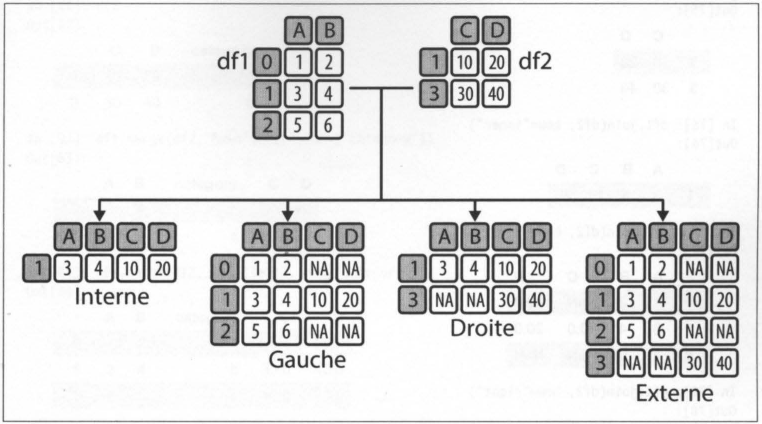


Figure 5.3 : Types de jointures.

Tableau 5.5 : Types de jointures.

Type	Description
Interne	Uniquement les lignes dont l'index existe dans les deux DataFrames.
Gauche	Toutes les lignes du DataFrame gauche, les lignes correspondantes du DataFrame droit.
Droite	Toutes les lignes du DataFrame droit, les lignes correspondantes du DataFrame gauche.
Externe	L'union des index des lignes des deux DataFrames.

Voyons comment cela fonctionne dans la pratique, en donnant vie aux exemples de la Figure 5.3 :

```
In [74]: df1 = pd.DataFrame(data=[[1, 2], [3, 4], [5, 6]],
                            columns=["A", "B"])
         df1
Out[74]:
```

	A	B
0	1	2
1	3	4
2	5	6

```
In [75]: df2 = pd.DataFrame(data=[[10, 20], [30, 40]],
                            columns=["C", "D"], index=[1, 3])
         df2
```

```
Out[75]:
```

	C	D
1	10	20
3	30	40

```
In [76]: df1.join(df2, how="inner")
Out[76]:
```

	A	B	C	D
1	3	4	10	20

```
In [77]: df1.join(df2, how="left")
Out[77]:
```

	A	B	C	D
0	1	2	NaN	NaN
1	3	4	10.0	20.0
2	5	6	NaN	NaN

```
In [78]: df1.join(df2, how="right")
Out[78]:
```

	A	B	C	D
1	3.0	4.0	10	20
3	NaN	NaN	30	40

```
In [79]: df1.join(df2, how="outer")
Out[79]:
```

	A	B	C	D
0	1.0	2.0	NaN	NaN
1	3.0	4.0	10.0	20.0
2	5.0	6.0	NaN	NaN
3	NaN	NaN	30.0	40.0

Si vous souhaitez faire une jointure sur une ou plusieurs colonnes de DataFrame au lieu de vous baser sur l'index, utilisez `merge` à la place de `join`. `merge` accepte l'argument `on` pour fournir une ou plusieurs colonnes comme *condition de jointure* : ces colonnes, qui doivent exister dans les deux DataFrames, sont utilisées pour faire correspondre les lignes :

```
In [80]: # Ajoute une colonne "category" aux deux DataFrames
         df1["category"] = ["a", "b", "c"]
         df2["category"] = ["c", "b"]
In [81]: df1
Out[81]:
```

	A	B	category
0	1	2	a
1	3	4	b
2	5	6	c

```
In [82]: df2
Out[82]:
```

	C	D	category
1	10	20	c
3	30	40	b

```
In [83]: df1.merge(df2, how="inner", on=["category"])
Out[83]:
```

	A	B	category	C	D
0	3	4	b	30	40
1	5	6	c	10	20

```
In [84]: df1.merge(df2, how="left", on=["category"])
Out[84]:
```

	A	B	category	C	D
0	1	2	a	NaN	NaN
1	3	4	b	30.0	40.0
2	5	6	c	10.0	20.0

Étant donné que `join` et `merge` acceptent un certain nombre d'arguments facultatifs pour s'adapter à des scénarios plus complexes, je vous invite à consulter la documentation officielle[4] pour en savoir plus à leur sujet.

Vous savez maintenant comment manipuler un ou plusieurs DataFrames, ce qui nous amène à l'étape suivante de notre parcours d'analyse de données : donner un sens aux données.

Statistiques descriptives et agrégation des données

Afin de donner un sens aux grands jeux de données, on peut calculer une statistique descriptive, comme la somme ou la moyenne sur l'ensemble du jeu de données ou sur des sous-ensembles significatifs. Cette section commence par examiner comment cela fonctionne avec pandas, avant de présenter deux façons d'agréger des données en sous-ensembles : la méthode `groupby` et la fonction `pivot_table`.

Statistiques descriptives

Les *statistiques descriptives* permettent de synthétiser des jeux de données à l'aide de mesures quantitatives. Par exemple, le nombre de points de données est une statistique descriptive simple. Les moyennes telles que la moyenne arithmétique, la médiane ou le mode sont d'autres exemples populaires. Les DataFrames et les séries permettent d'accéder facilement aux statistiques descriptives grâce à des méthodes telles que `sum`, `mean` et `count`, pour n'en citer que quelques-

4 https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

unes. Vous en rencontrerez beaucoup tout au long de ce livre, et la liste complète est disponible dans la documentation de pandas[5]. Par défaut, ces méthodes renvoient une série basée sur le paramètre `axis=0`, ce qui signifie que vous obtenez la statistique des colonnes :

```
In [85]: rainfall
Out[85]:
```

	City 1	City 2	City 3
0	300.1	400.3	1000.5
1	100.2	300.4	1100.6

```
In [86]: rainfall.mean()
Out[86]: City 1     200.15
         City 2     350.35
         City 3    1050.55
         dtype: float64
```

Si vous souhaitez la statistique par ligne, modifiez la valeur de l'argument `axis` :

```
In [87]: rainfall.mean(axis=1)
Out[87]: 0    566.966667
         1    500.400000
         dtype: float64
```

Par défaut, les valeurs manquantes ne sont pas incluses dans les statistiques descriptives telles que `sum` ou `mean`. Ceci est conforme à la façon dont Excel traite les cellules vides, si bien que l'utilisation de la formule `MOYENNE` d'Excel sur une plage avec des cellules vides vous donnera le même résultat que la méthode `mean` appliquée sur une série avec les mêmes nombres et valeurs `NaN` à la place des cellules vides.

L'obtention d'une statistique sur toutes les lignes d'un DataFrame n'est parfois pas suffisante et vous avez besoin d'informations plus granulaires, par exemple la moyenne par catégorie.

Regroupement

En utilisant à nouveau notre exemple de DataFrame `df`, voyons comment calculer les notes moyennes par continent. Pour ce faire, on regroupe d'abord les lignes par continent, puis on applique la méthode `mean`, qui calculera la moyenne *par groupe*. Toutes les colonnes non numériques sont automatiquement exclues :

```
In [88]: df.groupby(["continent"]).mean()
Out[88]:
```

	properties	age	score
continent			
America		37.0	5.30
Europe		33.5	6.75

5 https://pandas.pydata.org/pandas-docs/stable/reference/frame.html#computations-descriptive-stats

Si vous incluez plusieurs colonnes, le DataFrame résultant aura un index hiérarchique, le MultiIndex que nous avons étudié précédemment :

```
In [89]: df.groupby(["continent", "country"]).mean()
Out[89]:
```

		properties	age	score
continent	country			
America	USA		37	5.3
Europe	Germany		12	9.0
	Italy		55	4.5

Outre la moyenne, vous pouvez utiliser la plupart des statistiques descriptives que pandas offre, et si vous souhaitez utiliser votre propre fonction, utilisez la méthode `agg`. Par exemple, voici comment on obtient la différence entre la valeur maximale et la valeur minimale par groupe :

```
In [90]: df.groupby(["continent"]).agg(lambda x: x.max() - x.min())
Out[90]:
```

	properties	age	score
continent			
America		8	2.8
Europe		43	4.5

Une façon populaire d'obtenir des statistiques par groupe dans Excel consiste à utiliser des tableaux croisés dynamiques. Ils introduisent une deuxième dimension et sont parfaits pour examiner vos données sous différents angles. pandas a également une fonctionnalité de tableau croisé dynamique.

Tableaux croisés dynamiques et fusion

Si vous utilisez des tableaux croisés dynamiques dans Excel, vous n'aurez aucun mal à appliquer la fonction `pivot_table` de pandas, car elle fonctionne en grande partie de la même manière. Les données du DataFrame suivant sont organisées de la même manière que les enregistrements stockés dans une base de données ; chaque ligne affiche une transaction de vente pour un fruit spécifique dans une région :

```
In [91]: data = [["Oranges", "North", 12.30],
                 ["Apples", "South", 10.55],
                 ["Oranges", "South", 22.00],
                 ["Bananas", "South", 5.90],
                 ["Bananas", "North", 31.30],
                 ["Oranges", "North", 13.10]]
         sales = pd.DataFrame(data=data,
                              columns=["Fruit", "Region", "Revenue"])
         sales
```

Out[91]:

	Fruit	Region	Revenue
0	Oranges	North	12.30
1	Apples	South	10.55
2	Oranges	South	22.00
3	Bananas	South	5.90
4	Bananas	North	31.30
5	Oranges	North	13.10

Pour créer un tableau croisé dynamique, on fournit le DataFrame comme premier argument à la fonction `pivot_table`. Les paramètres `index` et `column` définissent respectivement les lignes du DataFrame et les étiquettes des colonnes du tableau croisé dynamique. Le paramètre `values` sera agrégé dans la partie données du DataFrame résultant à l'aide de `aggfunc`, une fonction qui peut être fournie sous forme de chaîne ou de fonction universelle NumPy. Enfin, les paramètres `margins` correspondent au `Total général` dans Excel, c'est-à-dire que si vous omettez les paramètres `margins` et `margins_name`, la colonne et la ligne Total ne seront pas affichées :

```
In [92]: pivot = pd.pivot_table(sales,
                                index="Fruit", columns="Region",
                                values="Revenue", aggfunc="sum",
                                margins=True, margins_name="Total")
         pivot
```

Out[92]:

Region	North	South	Total
Fruit			
Apples	NaN	10.55	10.55
Bananas	31.3	5.90	37.20
Oranges	25.4	22.00	47.40
Total	56.7	38.45	95.15

En résumé, faire un tableau croisé dynamique de vos données signifie prendre les valeurs uniques d'une colonne (`Region` dans notre exemple) et les transformer en en-têtes de colonnes du tableau croisé dynamique, agrégeant ainsi les valeurs d'une autre colonne. Cela facilite la lecture d'informations récapitulatives sur les dimensions qui nous intéressent. Dans notre tableau croisé dynamique, vous voyez instantanément qu'il n'y a pas de pommes vendues dans la région nord et que dans la région sud, la plupart des revenus proviennent des oranges. Si vous souhaitez faire l'inverse et transformer les en-têtes de colonnes en valeurs d'une seule colonne, utilisez `melt`. En ce sens, `melt` est l'opposé de la fonction `pivot_table` :

```
In [93]: pd.melt(pivot.iloc[:-1,:-1].reset_index(),
                 id_vars="Fruit",
                 value_vars=["North", "South"], value_name="Revenue")
```

```
Out[93]:
```

	Fruit	Region	Revenue
0	Apples	North	NaN
1	Bananas	North	31.30
2	Oranges	North	25.40
3	Apples	South	10.55
4	Bananas	South	5.90
5	Oranges	South	22.00

Dans cet exemple, je fournis notre tableau croisé dynamique comme entrée, mais j'utilise `iloc` pour me débarrasser de la ligne et de la colonne du total. J'ai également réinitialisé l'index afin que toutes les informations soient disponibles sous forme de colonnes normales. Je fournis ensuite `id_vars` pour indiquer les identificateurs et `value_vars` pour définir les colonnes que je veux intégrer dans mon « tableau croisé dynamique inversé ». La fusion (*melt*) peut être utile si vous souhaitez préparer les données afin qu'elles puissent être stockées dans une base de données qui les attend dans ce format.

Travailler avec des statistiques agrégées aide à comprendre les données, mais personne n'apprécie de lire une page pleine de chiffres. Pour rendre les informations facilement compréhensibles, rien ne fonctionne mieux que de créer des représentations visuelles. Alors qu'Excel utilise le terme *graphique*, pandas emploie généralement le mot *tracé*. J'utiliserai ces termes de manière interchangeable dans ce livre.

Réalisation de tracés

Le traçage permet de visualiser les résultats d'une analyse de données et pourrait bien être l'étape la plus importante de l'ensemble du processus. Pour le traçage, nous allons utiliser deux bibliothèques : nous commençons par étudier Matplotlib, la bibliothèque de traçage par défaut de pandas, avant de nous concentrer sur Plotly, une bibliothèque de traçage moderne qui offre une expérience plus interactive avec les notebooks Jupyter.

Matplotlib

Matplotlib est un package de traçage qui existe depuis longtemps et qui est inclus dans la distribution Anaconda. Avec lui, vous pouvez générer des tracés dans de nombreux formats, y compris des graphiques vectoriels pour une impression de haute qualité. Lorsque vous appelez la méthode `plot` d'un DataFrame, pandas produit un tracé Matplotlib par défaut.

Pour utiliser Matplotlib dans un notebook Jupyter, vous devez d'abord exécuter l'une des deux commandes magiques (voir le prochain encadré « Commandes magiques ») : `%matplotlib inline`

ou `%matplotlib notebook`. Ces commandes configurent le notebook afin que les tracés puissent être affichés dans le notebook lui-même. La dernière commande ajoute un peu plus d'interactivité, ce qui permet de modifier la taille ou le facteur de zoom du graphique. Commençons par créer un premier graphique avec pandas et Matplotlib (voir Figure 5.4) :

```
In [94]: import numpy as np
         %matplotlib inline
         # Ou %matplotlib notebook
In [95]: data = pd.DataFrame(data=np.random.rand(4, 4) * 100000,
                              index=["Q1", "Q2", "Q3", "Q4"],
                              columns=["East", "West", "North", "South"])
         data.index.name = "Quarters"
         data.columns.name = "Region"
         data
Out[95]:
```

Region	East	West	North	South
Quarters				
Q1	10968.156300	70753.504707	23604.531596	68935.641277
Q2	64399.016120	15874.191343	3948.962783	7634.708082
Q3	26131.596714	45317.686077	54447.419106	55249.844238
Q4	447.414150	88214.636039	47982.791670	7187.010450

```
In [96]: data.plot() # Raccourci for data.plot.line()
Out[96]: <AxesSubplot:xlabel='Quarters'>
```

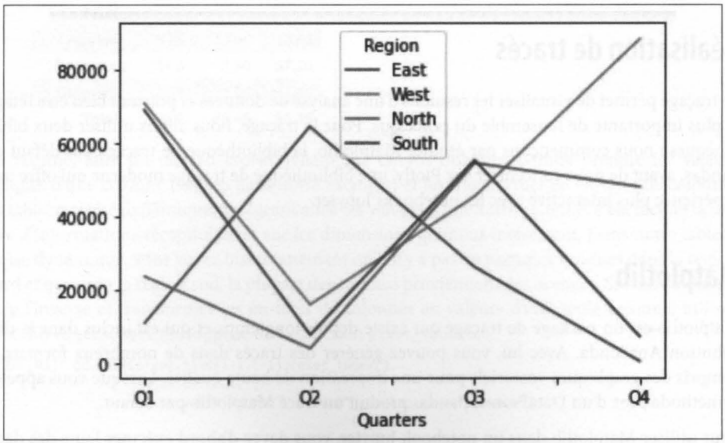


Figure 5.4 : Graphique tracé avec Matplotlib.

Vous noterez que dans cet exemple, j'ai utilisé un tableau NumPy pour construire un DataFrame pandas. Fournir des tableaux NumPy permet de tirer parti des constructeurs de NumPy que nous avons étudiés dans le chapitre précédent ; ici, nous utilisons NumPy pour générer un DataFrame pandas basé sur des nombres pseudo-aléatoires. Par conséquent, lorsque vous exécuterez l'exemple sur votre ordinateur, vous obtiendrez des valeurs différentes.

Commandes magiques

La commande `%matplotlib inline` que nous avons utilisée pour que Matplotlib fonctionne correctement avec les notebooks Jupyter est une *commande magique*. Les commandes magiques sont un ensemble de commandes simples qui modifient le comportement d'une cellule de notebook Jupyter ou qui rendent les tâches fastidieuses si faciles que l'on dirait presque de la magie. Vous écrivez des commandes magiques dans des cellules comme s'il s'agissait de code Python, mais elles commencent par `%%` ou `%`. Les commandes qui affectent la totalité de la cellule commencent par `%%` et les commandes qui n'affectent qu'une seule ligne dans une cellule commencent par `%`.

Nous verrons plus de commandes magiques dans les chapitres suivants, mais si vous souhaitez lister toutes les commandes magiques actuellement disponibles, exécutez `%lsmagic`, et pour en obtenir une description détaillée, exécutez `%magic`.

Même si vous utilisez la commande magique `%matplotlib notebook`, vous vous rendrez probablement compte que Matplotlib a été conçu à l'origine pour les tracés statiques plutôt que pour une expérience interactive sur une page Web. C'est pour cette raison que nous allons par la suite utiliser Plotly, une bibliothèque de traçage conçue pour le Web.

Plotly

Plotly est une bibliothèque basée sur JavaScript, qui peut, depuis la version 4.8.0, être utilisée comme un backend de traçage pour pandas, en assurant une grande interactivité : vous pouvez facilement zoomer, cliquer sur la légende pour sélectionner ou désélectionner une catégorie et obtenir des info-bulles avec plus d'informations sur le point de données que vous survolez. Plotly n'est pas incluse dans l'installation d'Anaconda, et si vous ne l'avez pas encore installée, faites-le maintenant en exécutant la commande suivante :

```
(base)> conda install plotly
```

Une fois que vous exécutez la cellule suivante, le backend de traçage de l'ensemble du notebook sera défini à Plotly et, si vous réexécutez la cellule précédente, vous obtiendrez alors un graphique Plotly. Pour tracer un graphique Plotly, vous n'avez pas besoin d'exécuter une commande magique, et il vous suffit de définir Plotly comme backend avant de pouvoir tracer la Figure 5.5 et la Figure 5.6 :

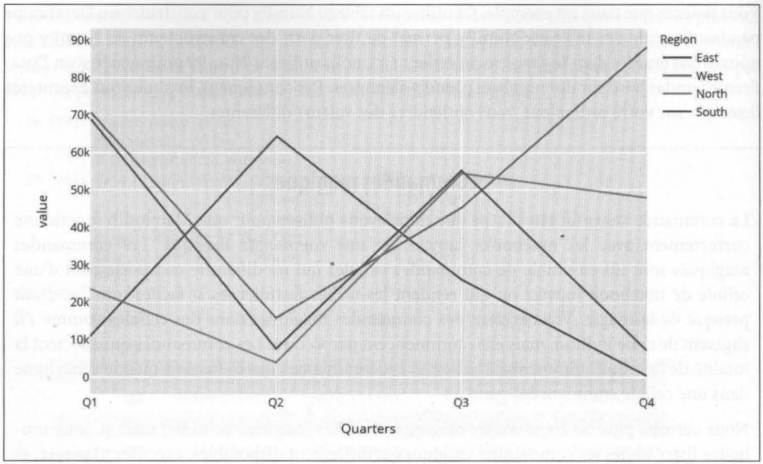


Figure 5.5 : Graphique en courbes avec Plotly.

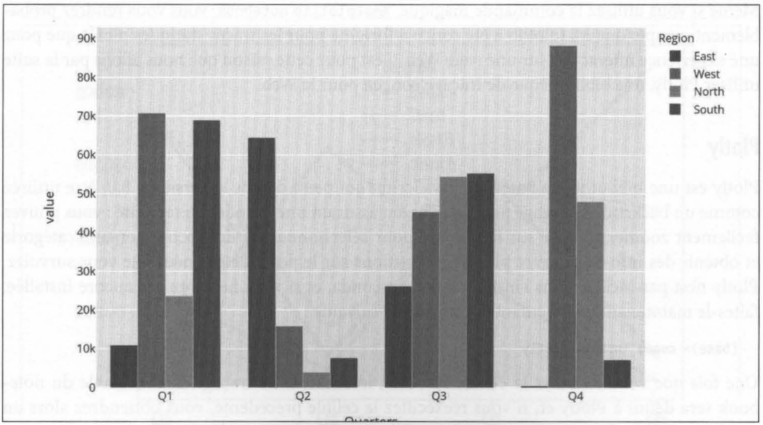


Figure 5.6 : Graphique à barres avec Plotly.

```
In [97]: # Définit le backend de traçage à Plotly
         pd.options.plotting.backend = "plotly"
In [98]: data.plot()

In [99]: # Affiche les mêmes données dans un graphique à barres
         data.plot.bar(barmode="group")
```



Si vous utilisez Plotly comme backend de traçage, vous devrez vérifier les arguments acceptés par les méthodes de traçage directement dans la documentation de Plotly. Par exemple, vous pouvez consulter l'argument `barmode=group` dans la documentation sur les graphiques à barres de Plotly[6].

pandas et les bibliothèques de traçage sous-jacentes offrent une multitude de types de graphiques et d'options pour formater les graphiques de la manière souhaitée. Il est également possible d'organiser plusieurs graphiques en une série de sous-tracés. Le Tableau 5.6 propose un aperçu des types de tracés disponibles.

Tableau 5.6 : Types de graphiques pandas.

Type	Description
`line`	Graphique en courbes, par défaut lors de l'exécution de `df.plot()`
`bar`	Graphique à barres verticales
`barh`	Graphique à barres horizontales
`hist`	Histogramme
`box`	Graphique en boîte et moustache
`kde`	Diagramme de densité, peut également être utilisé *via* `density`
`area`	Graphique en secteurs
`scatter`	Nuage de points
`hexbin`	Graphique hexbin
`pie`	Camembert

En plus de cela, pandas propose des outils et des techniques de traçage de niveau supérieur constitués de plusieurs composants individuels. Pour plus d'informations, consultez la documentation de pandas[7].

Nous allons créer plus de tracés dans le chapitre suivant afin d'analyser des séries chronologiques, mais avant d'y arriver, concluons ce chapitre en apprenant à importer et à exporter des données avec pandas.

6 https://plotly.com/python/bar-charts/
7 https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

Autres bibliothèques de traçage

Dans le domaine scientifique, le paysage de la représentation graphique en Python est très actif, et outre Matplotlib et Plotly, il existe de nombreuses autres bibliothèques de grande qualité. En fonction de votre contexte d'utilisation, l'une des bibliothèques suivantes pourrait se révéler un meilleur choix :

Seaborn
Seaborn[8] est construite à partir de Matplotlib. Elle améliore le style par défaut et ajoute des tracés supplémentaires, comme des cartes de chaleur, qui simplifient souvent votre travail : vous pouvez créer des tracés statistiques avancés avec seulement quelques lignes de code.

Bokeh
Bokeh[9] est semblable à Plotly dans la technologie et les fonctionnalités : elle est basée sur JavaScript et fonctionne donc également très bien pour les graphiques interactifs dans les notebooks Jupyter. Bokeh est incluse dans Anaconda.

Altair
Altair[10] est une bibliothèque de visualisations statistiques basée sur le projet Vega. Altair est également basée sur JavaScript et offre une certaine interactivité, comme le zoom.

HoloViews
HoloViews[11] est un autre package basé sur JavaScript qui se concentre sur la facilité d'analyse et de visualisation des données. Avec quelques lignes de code, vous pouvez obtenir des graphiques statistiques complexes.

Importation et exportation de DataFrames

Jusqu'à présent, nous avons construit des DataFrames à partir de zéro, à l'aide de listes imbriquées, de dictionnaires ou de tableaux NumPy. Ces techniques sont importantes à connaître, mais en règle générale, les données sont déjà disponibles et il vous suffit de les transformer en DataFrames. Pour ce faire, pandas propose diverses fonctions de lecture. Mais même si vous avez besoin d'accéder à un système propriétaire pour lequel pandas n'offre pas de lecteur intégré, vous avez souvent un package Python pour vous connecter à ce système, et une fois que vous avez les données, il est assez facile de les transformer en DataFrames. Dans Excel, l'importation de données est une tâche que vous gérez généralement avec Power Query.

8 https://seaborn.pydata.org/
9 https://docs.bokeh.org/en/latest/
10 https://altair-viz.github.io/
11 https://holoviews.org/

Après avoir analysé et modifié votre jeu de données, vous pouvez renvoyer les résultats dans une base de données, les exporter vers un fichier CSV ou les présenter dans un classeur Excel. Pour exporter des DataFrames, utilisez l'une des méthodes d'exportation proposées par pandas. Le Tableau 5.7 propose un aperçu des méthodes d'importation et d'exportation les plus courantes.

Tableau 5.7 : Importation et exportation de DataFrames.

Format de données/système	Importation : fonction pandas (pd)	Exportation : méthode DataFrame (df)
Fichiers CSV	`pd.read_csv`	`df.to_csv`
JSON	`pd.read_json`	`df.to_json`
HTML	`pd.read_html`	`df.to_html`
Presse-papiers	`pd.read_clipboard`	`df.to_clipboard`
Fichiers Excel	`pd.read_excel`	`df.to_excel`
Bases de données SQL	`pd.read_sql`	`df.to_sql`

Nous étudierons `pd.read_sql` et `pd.to_sql` au Chapitre 11 où nous les utiliserons dans le cadre d'une étude de cas. Et puisque je vais consacrer l'intégralité du Chapitre 7 à la lecture et à l'écriture de fichiers Excel avec pandas, je vais me concentrer sur l'importation et l'exportation de fichiers CSV dans cette section. Commençons par exporter un DataFrame existant.

Exportation de fichiers CSV

Si vous avez besoin de transmettre un DataFrame à un collègue qui n'utilise pas Python ou pandas, il est préférable de le lui envoyer sous la forme d'un fichier CSV : à peu près tous les programmes savent importer ce type de fichier. Pour exporter notre exemple de DataFrame `df` vers un fichier CSV, utilisez la méthode `to_csv` :

```
In [100]: df.to_csv("course_participants.csv")
```

Si vous souhaitez stocker le fichier dans un répertoire différent, fournissez le chemin d'accès complet sous la forme d'une chaîne brute, par exemple, `r"C:\path\to\desired\location\msft.csv"`.



Dans les chaînes de caractères, la barre oblique inverse est utilisée pour échapper certains caractères. C'est la raison pour laquelle pour travailler avec des chemins d'accès de fichier sous Windows, vous devez soit utiliser des antislashs doubles (`C:\\path\\to\\file.csv`) ou préfixer la chaîne avec un `r` pour la transformer en une chaîne brute qui interprète les caractères littéralement. Ce n'est pas un problème sur macOS ou Linux, car ces systèmes utilisent des barres obliques dans les chemins d'accès.

En fournissant uniquement le nom de fichier comme je le fais, cela produira le fichier *course_participants.csv* dans le même répertoire que le notebook avec le contenu suivant :

```
user_id,name,age,country,score,continent
1001,Mark,55,Italy,4.5,Europe
1000,John,33,USA,6.7,America
1002,Tim,41,USA,3.9,America
1003,Jenny,12,Germany,9.0,Europe
```

Maintenant que vous savez comment utiliser la méthode `df.to_csv`, voyons comment fonctionne l'importation d'un fichier CSV.

Importation de fichiers CSV

L'importation d'un fichier CSV en local est aussi simple que de fournir son chemin d'accès à la fonction `read_csv`. *MSFT.csv* est un fichier CSV que j'ai téléchargé à partir de Yahoo! Finance et qui contient l'historique des cours quotidiens des actions de Microsoft (le fichier se trouve dans les exemples de code, dans le dossier *csv*) :

```
In [101]: msft = pd.read_csv("csv/MSFT.csv")
```

Souvent, vous devrez fournir, outre le nom du fichier, quelques paramètres supplémentaires à `read_csv`. Par exemple, `sep` permet de dire à pandas quel séparateur ou délimiteur le fichier CSV utilise au cas où il ne s'agit pas de la virgule, qui est le séparateur par défaut. Nous utiliserons quelques paramètres supplémentaires dans le chapitre suivant, mais pour un aperçu complet, consultez la documentation de pandas[12].

Maintenant que nous avons affaire à des DataFrames plus importants comportant plusieurs milliers de lignes, la première chose à faire est généralement d'exécuter la méthode `info` pour obtenir un résumé du DataFrame. Ensuite, vous souhaiterez peut-être jeter un coup d'œil aux premières et dernières lignes du DataFrame à l'aide des méthodes `head` et `tail`. Ces méthodes retournent cinq lignes par défaut, mais cela peut être modifié en fournissant le nombre souhaité de lignes en tant qu'argument. Vous pouvez également exécuter la méthode `describe` pour obtenir des statistiques de base :

```
In [102]: msft.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8622 entries, 0 to 8621
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       8622 non-null   object
 1   Open       8622 non-null   float64
 2   High       8622 non-null   float64
 3   Low        8622 non-null   float64
 4   Close      8622 non-null   float64
 5   Adj Close  8622 non-null   float64
```

12 https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

```
 6   Volume      8622 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 471.6+ KB
In [103]: # Je ne sélectionne que quelques colonnes
          # par souci de lisibilité
          # Vous pouvez aussi exécuter : msft.head()
          msft.loc[:, ["Date", "Adj Close", "Volume"]].head()
Out[103]:
```

	Date	Adj Close	Volume
0	1986-03-13	0.062205	1031788800
1	1986-03-14	0.064427	308160000
2	1986-03-17	0.065537	133171200
3	1986-03-18	0.063871	67766400
4	1986-03-19	0.062760	47894400

```
In [104]: msft.loc[:, ["Date", "Adj Close", "Volume"]].tail(2)
Out[104]:
```

	Date	Adj Close	Volume
8620	2020-05-26	181.570007	36073600
8621	2020-05-27	181.809998	39492600

```
In [105]: msft.loc[:, ["Adj Close", "Volume"]].describe()
Out[105]:
```

	Adj Close	Volume
count	8622.000000	8.622000e+03
mean	24.921952	6.030722e+07
std	31.838096	3.877805e+07
min	0.057762	2.304000e+06
25%	2.247503	3.651632e+07
50%	18.454313	5.350380e+07
75%	25.699224	7.397560e+07
max	187.663330	1.031789e+09

`Adj Close` est l'abréviation de *ajusted close price* (*prix de clôture ajusté*), c'est-à-dire que le prix de l'action est corrigé pour les opérations de sociétés telles que les fractionnements d'actions. `Volume` est le nombre d'actions qui ont été négociées. J'ai synthétisé les différentes méthodes d'exploration de DataFrames que nous avons vues dans ce chapitre dans le Tableau 5.8.

Tableau 5.8 : Méthodes et attributs d'exploration de DataFrames.

DataFrame (df), méthode/attribut	Description
`df.info()`	Fournit le nombre de points de données, le type d'index, dtype et l'utilisation de la mémoire.
`df.describe()`	Fournit des statistiques de base, notamment le comptage, la moyenne, l'écart type, le min, le max et les percentiles.

Tableau 5.8 : Méthodes et attributs d'exploration de DataFrames (suite).

DataFrame (df), méthode/attribut	Description
`df.head(n=5)`	Retourne les *n* premières lignes du DataFrame.
`df.tail(n=5)`	Retourne les *n* dernières lignes du DataFrame.
`df.dtypes`	Retourne le dtype de chaque colonne.

La fonction `read_csv` accepte également une URL à la place d'un fichier CSV en local. Voici comment lire le fichier CSV directement à partir du référentiel associé :

```
In [106]: # Le saut de ligne dans l'URL sert à faire tenir sur la page
          url = ("https://raw.githubusercontent.com/fzumstein/"
                 "python-for-excel/1st-edition/csv/MSFT.csv")
          msft = pd.read_csv(url)
In [107]: msft.loc[:, ["Date", "Adj Close", "Volume"]].head(2)
Out[107]:
```

	Date	Adj Close	Volume
0	1986-03-13	0.062205	1031788800
1	1986-03-14	0.064427	308160000

Nous allons continuer avec ce jeu de données et la fonction `read_csv` dans le chapitre suivant consacré aux séries chronologiques, où nous transformerons la colonne Date en `datetimeIndex`.

Conclusion

Ce chapitre a apporté son lot de nouveaux concepts et d'outils pour analyser les ensembles de données avec pandas. Nous avons appris à charger des fichiers CSV, à gérer des données manquantes ou en double et à utiliser des statistiques descriptives. Nous avons également vu à quel point il est facile de transformer vos DataFrames en graphiques interactifs. Bien qu'il faille un certain temps pour tout assimiler, il ne vous faudra probablement pas longtemps avant de comprendre l'immense pouvoir que vous gagnez en ajoutant pandas à votre panoplie d'outils. En cours de route, nous avons comparé pandas aux fonctionnalités Excel suivantes :

- *Fonctionnalité de filtrage automatique :* reportez-vous à la section « Sélection par indexation booléenne ».
- *Formule RECHERCHEV* : reportez-vous à la section « Jointure et fusion ».
- *Tableau croisé dynamique :* reportez-vous à la section « Tableaux croisés dynamiques et fusion ».
- *Power Query :* il s'agit d'une combinaison des sections « Importations et exportation de DataFrames », « Manipulation des données » et « Combinaison de DataFrames ».

Le chapitre suivant traite de l'analyse des séries chronologiques, fonctionnalité qui a conduit à une large adoption de pandas par le secteur financier. Voyons pourquoi cet aspect de pandas a pris un tel avantage sur Excel…

CHAPITRE 6

Analyse de séries chronologiques avec pandas

Une *série chronologique* est une série de points de données sur un axe temporel qui joue un rôle central dans de nombreux scénarios différents : alors que les traders utilisent l'historique du cours des actions pour calculer les mesures de risque, les prévisionnistes météo se basent sur des séries chronologiques générées par des capteurs mesurant la température, l'humidité et la pression atmosphérique. En outre, les services chargés du marketing digital s'appuient sur des séries chronologiques générées par des pages Web, par exemple la source et le nombre de pages vues par heure, qu'ils utiliseront pour tirer des conclusions sur leurs campagnes marketing.

L'analyse de séries chronologiques est l'un des principaux atouts qui ont poussé les scientifiques et les analystes à chercher une alternative à Excel. Voici quelques-unes des raisons qui ont motivé ce changement :

- **Grands ensembles de données :** les séries chronologiques peuvent rapidement dépasser la limite d'Excel qui se situe autour du million de lignes par feuille de calcul. Par exemple, si vous travaillez avec les cours intrajournaliers des actions au niveau des données de tick, vous avez souvent affaire à des centaines de milliers d'enregistrements, par action et par jour !
- **Date et heure :** comme nous l'avons vu dans le Chapitre 3, Excel présente des limites en ce qui concerne la gestion de la date et de l'heure, qui constituent l'épine dorsale des séries chronologiques. Parmi ces limites, on peut citer l'absence de prise en charge des fuseaux horaires et un format des nombres limité aux millisecondes. pandas prend en charge les fuseaux horaires et utilise le type de données `datetime64[ns]` de NumPy, qui offre une précision allant jusqu'aux nanosecondes.
- **Fonctionnalités manquantes :** il manque aussi à Excel certains outils de base pour pouvoir travailler décemment avec des séries chronologiques. Par exemple, si vous souhaitez transformer une série chronologique quotidienne en une série chronologique mensuelle, il n'existe aucun moyen facile de réaliser cela, bien qu'il s'agisse d'une tâche très courante.

Les DataFrames permettent de travailler avec différents index basés sur le temps : `DatetimeIndex` est le plus courant et représente un index avec des horodatages. D'autres types d'index, tels que `PeriodIndex`, sont basés sur des intervalles de temps tels que des heures ou des mois. Dans ce chapitre, cependant, nous examinerons uniquement `DatetimeIndex`.

DatetimeIndex

Dans cette section, nous allons apprendre à construire un `DatetimeIndex`, comment filtrer un tel index sur une plage de temps spécifique et comment utiliser des fuseaux horaires.

Création d'un DatetimeIndex

Pour construire un `DatetimeIndex`, pandas propose la fonction `date_range` qui accepte une date de début, une fréquence et le nombre de périodes ou la date de fin :

```
In [1]: # On commence par importer les packages utilisés dans ce chapitre
        # et par définir le backend de traçage à Plotly
        import pandas as pd
        import numpy as np
        pd.options.plotting.backend = "plotly"
In [2]: # On crée un DatetimeIndex basé sur une date de départ,
        # un nombre de périodes et une fréquence ("D" = journalier).
        daily_index = pd.date_range("2020-02-28", periods=4, freq="D")
        daily_index
Out[2]: DatetimeIndex(['2020-02-28', '2020-02-29', '2020-03-01', '2020-03-02'],
         dtype='datetime64[ns]', freq='D')
In [3]: # On crée un DatetimeIndex basé sur une date de début et de fin
        # La fréquence est hebdomadaire et a lieu le dimanche ("W-SUN").
        weekly_index = pd.date_range("2020-01-01", "2020-01-31", freq="W-SUN")
        weekly_index
Out[3]: DatetimeIndex(['2020-01-05', '2020-01-12', '2020-01-19', '2020-01-26'],
         dtype='datetime64[ns]', freq='W-SUN')
In [4]: # On crée un DataFrame basé sur weekly_index. Il peut s'agir
        # du nombre de visiteurs d'un musée qui n'ouvre que le dimanche
        pd.DataFrame(data=[21, 15, 33, 34],
                     columns=["visitors"], index=weekly_index)
Out[4]:
```

	visitors
2020-01-05	21
2020-01-12	15
2020-01-19	33
2020-01-26	34

Revenons à présent à la série chronologique du cours des actions Microsoft que nous avons évoquée dans le chapitre précédent. Lorsque vous regardez de plus près les types de données des

colonnes, vous remarquerez que la colonne `Date` a le type `Object`, ce qui signifie que pandas a interprété les horodatages comme des chaînes de caractères :

```
In [5]: msft = pd.read_csv("csv/MSFT.csv")
In [6]: msft.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8622 entries, 0 to 8621
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       8622 non-null   object
 1   Open       8622 non-null   float64
 2   High       8622 non-null   float64
 3   Low        8622 non-null   float64
 4   Close      8622 non-null   float64
 5   Adj Close  8622 non-null   float64
 6   Volume     8622 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 471.6+ KB
```

Il existe deux façons de résoudre ce problème et de transformer la colonne en un type de données `datetime`. La première consiste à exécuter la fonction `to_datetime` sur cette colonne. Veillez à affecter la colonne transformée au DataFrame d'origine si vous souhaitez la modifier à la source :

```
In [7]: msft.loc[:, "Date"] = pd.to_datetime(msft["Date"])
In [8]: msft.dtypes
Out[8]: Date          datetime64[ns]
        Open                 float64
        High                 float64
        Low                  float64
        Close                float64
        Adj Close            float64
        Volume                 int64
        dtype: object
```

L'autre possibilité consiste à indiquer à `read_csv` les colonnes qui contiennent des horodatages à l'aide de l'argument `parse_dates`. `parse_dates` attend une liste de noms de colonnes ou d'index. En outre, il est presque toujours souhaitable de transformer les horodatages en index du DataFrame car cela permet de filtrer facilement les données, comme nous le verrons dans un instant. Pour vous épargner un appel supplémentaire à `set_index`, fournissez la colonne que vous souhaitez utiliser comme index, *via* l'argument `index_col`, sous la forme d'un nom de colonne ou d'index :

```
In [9]: msft = pd.read_csv("csv/MSFT.csv",
                           index_col="Date", parse_dates=["Date"])
In [10]: msft.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 8622 entries, 1986-03-13 to 2020-05-27
Data columns (total 6 columns):
```

```
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Open       8622 non-null   float64
 1   High       8622 non-null   float64
 2   Low        8622 non-null   float64
 3   Close      8622 non-null   float64
 4   Adj Close  8622 non-null   float64
 5   Volume     8622 non-null   int64
dtypes: float64(5), int64(1)
memory usage: 471.5 KB
```

Comme `info` le révèle, vous avez maintenant affaire à un DataFrame qui a un `DatetimeIndex`. Si vous devez modifier un autre type de données (supposons que vous vouliez que `Volume` soit un `float` au lieu d'un `int`), vous avez à nouveau deux options : soit fournir `dtype={"Volume": float}` comme argument à la fonction `read_csv`, soit appliquer la méthode `astype` de la manière suivante :

```
In [11]: msft.loc[:, "Volume"] = msft["Volume"].astype("float")
         msft["Volume"].dtype
Out[11]: dtype('float64')
```

Avec les séries chronologiques, il est toujours judicieux de s'assurer que l'index est trié correctement avant de commencer votre analyse :

```
In [12]: msft = msft.sort_index()
```

Enfin, si vous avez besoin d'accéder uniquement à des parties d'un `DatetimeIndex`, comme la partie date sans l'heure, accédez à l'attribut `date` de la manière suivante :

```
In [13]: msft.index.date
Out[13]: array([datetime.date(1986, 3, 13), datetime.date(1986, 3, 14),
                datetime.date(1986, 3, 17), ..., datetime.date(2020, 5, 22),
                datetime.date(2020, 5, 26), datetime.date(2020, 5, 27)],
               dtype=object)
```

À la place de `date`, vous pouvez également utiliser des parties d'une date comme l'année (`year`), le mois (`month`), le jour (`day`), *etc.* Pour accéder à la même fonctionnalité sur une colonne normale avec le type de données `datetime`, vous devez utiliser l'attribut `dt`, par exemple, `df["nom_colonne"].dt.date`.

Avec un `DatetimeIndex` trié, voyons comment nous pouvons filtrer le DataFrame sur certaines périodes.

Filtrage d'un DatetimeIndex

Si votre DataFrame a un `DatetimeIndex`, il existe un moyen simple de sélectionner des lignes à partir d'une période spécifique en utilisant `loc` avec une chaîne au format `AAAA-MM-JJ HH:MM:SS`. pandas transformera cette chaîne en une limite d'intervalle afin qu'elle couvre toute la période.

Par exemple, pour sélectionner toutes les lignes de 2019, fournissez l'année sous la forme d'une *chaîne*, et non pas d'un nombre :

```
In [14]: msft.loc["2019", "Adj Close"]
Out[14]: Date
         2019-01-02     99.099190
         2019-01-03     95.453529
         2019-01-04     99.893005
         2019-01-07    100.020401
         2019-01-08    100.745613
                           ...
         2019-12-24    156.515396
         2019-12-26    157.798309
         2019-12-27    158.086731
         2019-12-30    156.724243
         2019-12-31    156.833633
         Name: Adj Close, Length: 252, dtype: float64
```

Allons plus loin et faisons un graphique à partir des données entre juin 2019 et mai 2020 (voir la Figure 6.1) :

```
In [15]: msft.loc["2019-06":"2020-05", "Adj Close"].plot()
```

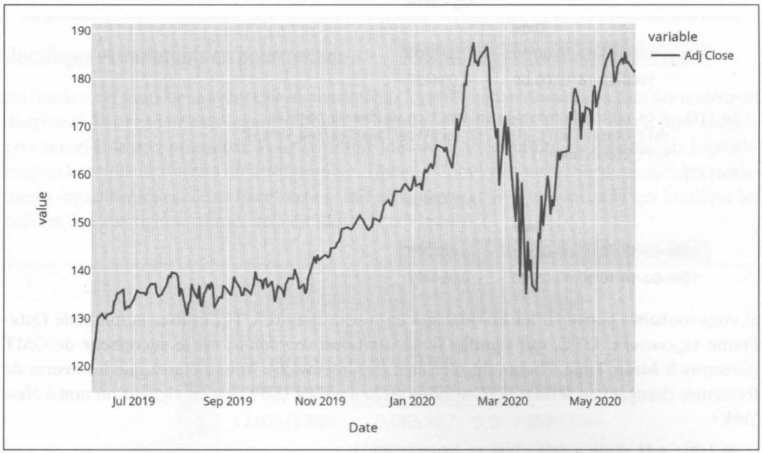


Figure 6.1 : Prix de clôture ajusté pour l'action Microsoft.

Pointez la courbe du graphique pour lire les valeurs sous forme d'info-bulle et effectuez un zoom en dessinant un rectangle avec votre souris. Faites un double-clic sur le graphique pour revenir à la vue par défaut.

Nous utiliserons le prix de clôture ajusté dans la section suivante pour apprendre à gérer les fuseaux horaires.

Utilisation des fuseaux horaires

Microsoft est cotée au Nasdaq, qui est situé à New York, où les marchés ferment à 16 heures. Pour ajouter ces informations supplémentaires à l'index du DataFrame, ajoutez d'abord l'heure de fermeture à la date *via* `DateOffset`, puis attachez le fuseau horaire correct aux horodatages *via* `tz_localize`. Étant donné que l'heure de fermeture ne s'applique qu'au prix de clôture, créons un nouveau DataFrame qui contient cette information :

```
In [16]: # On ajoute l'heure de clôture à la date
         msft_close = msft.loc[:, ["Adj Close"]].copy()
         msft_close.index = msft_close.index + pd.DateOffset(hours=16)
         msft_close.head(2)
Out[16]:
```

Date	Adj Close
1986-03-13 16:00:00	0.062205
1986-03-14 16:00:00	0.064427

```
In [17]: # On crée l'horodatage en fonction du fuseau horaire
         msft_close = msft_close.tz_localize("America/New_York")
         msft_close.head(2)
Out[17]:
```

Date	Adj Close
1986-03-13 16:00:00-05:00	0.062205
1986-03-14 16:00:00-05:00	0.064427

Si vous souhaitez convertir les horodatages en fuseau horaire UTC, utilisez la méthode DataFrame `tz_convert`. UTC, qui signifie *Temps universel coordonné*, est le successeur de GMT (Greenwich Mean Time, l'heure du méridien de Greenwich). Vous noterez que les heures de fermeture changent au format UTC selon que l'heure d'été (DST) est en vigueur ou non à New York :

```
In [18]: msft_close = msft_close.tz_convert("UTC")
         msft_close.loc["2020-01-02", "Adj Close"]  # 21:00 sans heure d'été
Out[18]: Date
         2020-01-02 21:00:00+00:00    159.737595
         Name: Adj Close, dtype: float64
In [19]: msft_close.loc["2020-05-01", "Adj Close"]  # 20:00 avec heure d'été
```

```
Out[19]: Date
         2020-05-01 20:00:00+00:00    174.085175
         Name: Adj Close, dtype: float64
```

La préparation de séries chronologiques comme celle-ci vous permettra de comparer les prix de clôture des Bourses sur différents fuseaux horaires, même si les informations horaires sont manquantes ou indiquées dans le fuseau horaire local.

Maintenant que vous savez ce qu'est un `DatetimeIndex`, nous allons tester dans la section suivante quelques manipulations courantes sur des séries chronologiques en calculant et en comparant les performances des actions.

Manipulations courantes sur les séries chronologiques

Dans cette section, je vais vous montrer comment effectuer des tâches courantes d'analyse de séries chronologiques telles que le calcul du rendement des actions, le traçage des performances de divers titres et la visualisation de la corrélation de leurs rendements dans une carte de chaleur (*heatmap*). Nous verrons également comment modifier la fréquence des séries chronologiques et comment calculer des statistiques glissantes.

Décalages et variations en pourcentage

En finance, on suppose souvent que les *rendements logarithmiques* des actions sont normalement distribués. Par rendements logarithmiques, j'entends le logarithme naturel du rapport entre le prix actuel et le prix précédent. Pour avoir une idée de la distribution des rendements logarithmiques quotidiens, traçons un histogramme. Toutefois, nous devons d'abord calculer les rendements logarithmiques. Dans Excel, cela se fait généralement avec une formule qui implique les cellules de deux lignes, comme cela est illustré à la Figure 6.2.

	A	B	C
1	Date	Adj Close	
2	13/03/1986	0,062205	
3	14/03/1986	0,064427	=LN(B3/B2)
4	17/03/1986	0,065537	0,01708207

Figure 6.2 : Calcul des rendements logarithmiques dans Excel.



Excel utilise la fonction `LN` pour désigner le logarithme népérien et LOG pour le logarithme en base 10. En revanche, le module mathématique de Python et de NumPy emploie `log` pour le logarithme népérien et `log10` pour le logarithme en base 10.

Avec pandas, plutôt que d'avoir une formule accédant à deux lignes différentes, on utilise la méthode `shift` pour décaler les valeurs d'une ligne vers le bas. Cela permet d'opérer sur une seule ligne afin que vos calculs puissent utiliser la vectorisation. `shift` accepte un entier positif ou négatif qui déplace la série chronologique du nombre de lignes respectivement vers le bas ou vers le haut. Voyons d'abord comment fonctionne la méthode `shift` :

```
In [20]: msft_close.head()
Out[20]:
```

Date	Adj Close
1986-03-13 21:00:00+00:00	0.062205
1986-03-14 21:00:00+00:00	0.064427
1986-03-17 21:00:00+00:00	0.065537
1986-03-18 21:00:00+00:00	0.063871
1986-03-19 21:00:00+00:00	0.062760

```
In [21]: msft_close.shift(1).head()
Out[21]:
```

Date	Adj Close
1986-03-13 21:00:00+00:00	NaN
1986-03-14 21:00:00+00:00	0.062205
1986-03-17 21:00:00+00:00	0.064427
1986-03-18 21:00:00+00:00	0.065537
1986-03-19 21:00:00+00:00	0.063871

Vous êtes maintenant en mesure d'écrire une formule vectorielle unique qui est facile à lire et à comprendre. Pour obtenir le logarithme népérien, utilisez la fonction universelle `log` de NumPy, qui est appliquée à chaque élément. Ensuite, nous pouvons tracer un histogramme (voir la Figure 6.3) :

```
In [22]: returns = np.log(msft_close / msft_close.shift(1))
         returns = returns.rename(columns={"Adj Close": "returns"})
         returns.head()
```

Out[22]:

Date	returns
1986-03-13 21:00:00+00:00	NaN
1986-03-14 21:00:00+00:00	0.035097
1986-03-17 21:00:00+00:00	0.017082
1986-03-18 21:00:00+00:00	-0.025749
1986-03-19 21:00:00+00:00	-0.017547

```
In [23]: # Trace un histogramme avec les rendements logarithmiques journaliers
         returns.plot.hist()
```

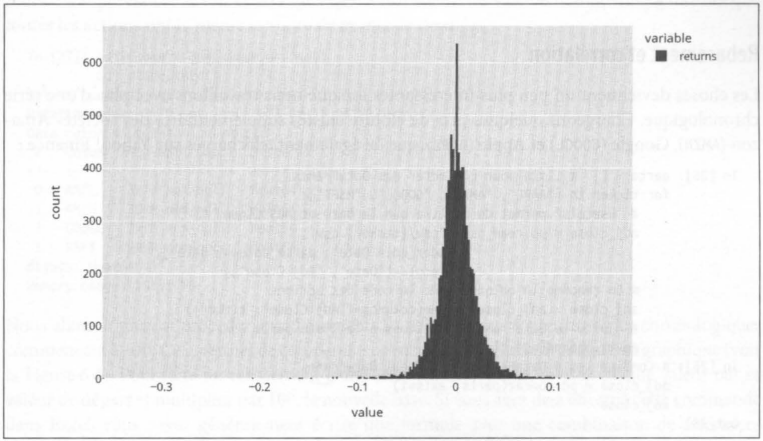


Figure 6.3 : Histogramme des rendements logarithmiques journaliers.

Pour obtenir des *rendements simples* à la place, utilisez la méthode intégrée `pct_change` de pandas. Par défaut, elle calcule la variation en pourcentage par rapport à la ligne précédente, ce qui est également la définition des rendements simples :

```
In [24]: simple_rets = msft_close.pct_change()
         simple_rets = simple_rets.rename(columns={"Adj Close": "simple rets"})
         simple_rets.head()
```

Out[24]:

Date	simple rets
1986-03-13 21:00:00+00:00	NaN
1986-03-14 21:00:00+00:00	0.035721
1986-03-17 21:00:00+00:00	0.017229
1986-03-18 21:00:00+00:00	-0.025421
1986-03-19 21:00:00+00:00	-0.017394

Jusqu'à présent, nous n'avons examiné que l'action Microsoft. Dans la section suivante, nous allons charger plus de séries chronologiques afin de pouvoir examiner d'autres méthodes de DataFrame qui nécessitent plusieurs séries chronologiques.

Rebasement et corrélation

Les choses deviennent un peu plus intéressantes lorsque nous travaillons avec plus d'une série chronologique. Chargeons quelques prix de clôture ajustés supplémentaires des sociétés Amazon (`AMZN`), Google (`GOOGL`) et Apple (`AAPL`), que j'ai également téléchargés sur Yahoo! Finance :

```
In [25]: parts = []  # Liste pour collecter des DataFrames
         for ticker in ["AAPL", "AMZN", "GOOGL", "MSFT"]:
             # "usecols" permet de ne lire que la Date et Adj Close
             adj_close = pd.read_csv(f"csv/{ticker}.csv",
                                     index_col="Date", parse_dates=["Date"],
                                     usecols=["Date", "Adj Close"])
             # On renomme la colonne avec le code des actions
             adj_close = adj_close.rename(columns={"Adj Close": ticker})
             # Ajoute le DataFrame des actions à la liste parts
             parts.append(adj_close)
In [26]: # Combine les 4 DataFrames en un seul DataFrame
         adj_close = pd.concat(parts, axis=1)
         adj_close
```

Out[26]:

Date	AAPL	AMZN	GOOGL	MSFT
1980-12-12	0.405683	NaN	NaN	NaN
1980-12-15	0.384517	NaN	NaN	NaN
1980-12-16	0.356296	NaN	NaN	NaN
1980-12-17	0.365115	NaN	NaN	NaN
1980-12-18	0.375698	NaN	NaN	NaN
...	...	...	...	...
2020-05-22	318.890015	2436.879883	1413.239990	183.509995
2020-05-26	316.730011	2421.860107	1421.369995	181.570007

2020-05-27	318.109985	2410.389893	1420.280029	181.809998
2020-05-28	318.250000	2401.100098	1418.239990	NaN
2020-05-29	317.940002	2442.370117	1433.520020	NaN

```
9950 rows × 4 columns
```

Avez-vous remarqué la puissance de `concat` ? pandas a automatiquement aligné les séries chronologiques individuelles sur les dates. C'est la raison pour laquelle vous obtenez des valeurs `NaN` pour les actions qui ne remontent pas aussi loin que celles d'Apple. Et puisque `MSFT` a des valeurs `NaN` aux dates les plus récentes, vous avez peut-être deviné que j'ai téléchargé *MSFT.csv* deux jours avant les autres. L'alignement des séries chronologiques par date est une opération courante qui est très lourde à réaliser avec Excel et elle est par conséquent souvent source d'erreurs. La suppression de toutes les lignes qui contiennent des valeurs manquantes garantira que toutes les actions ont le même nombre de points de données :

```
In [27]: adj_close = adj_close.dropna()
         adj_close.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3970 entries, 2004-08-19 to 2020-05-27
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   AAPL    3970 non-null   float64
 1   AMZN    3970 non-null   float64
 2   GOOGL   3970 non-null   float64
 3   MSFT    3970 non-null   float64
dtypes: float64(4)
memory usage: 155.1 KB
```

Nous allons à présent procéder au rebasement des prix afin que toutes les séries chronologiques commencent à 100. Cela permet de comparer leurs performances relatives dans un graphique (voir la Figure 6.4). Pour faire un rebasement d'une série chronologique, divisez chaque valeur par sa valeur de départ et multipliez par 100, la nouvelle base. Si vous avez déjà effectué cette commande dans Excel, vous devez généralement écrire une formule avec une combinaison de références absolues et relatives, puis copier la formule pour chaque ligne et chaque série chronologique. Avec pandas, grâce à la vectorisation et à la diffusion, vous avez affaire à une seule formule :

```
In [28]: # On utilise un échantillon compris entre juin 2019 et mai 2020
         adj_close_sample = adj_close.loc["2019-06":"2020-05", :]
         rebased_prices = adj_close_sample / adj_close_sample.iloc[0, :] * 100
         rebased_prices.head(2)
Out[28]:
```

Date	AAPL	AMZN	GOOGL	MSFT
2019-06-03	100.000000	100.000000	100.00000	100.000000
2019-06-04	103.658406	102.178197	101.51626	102.770372

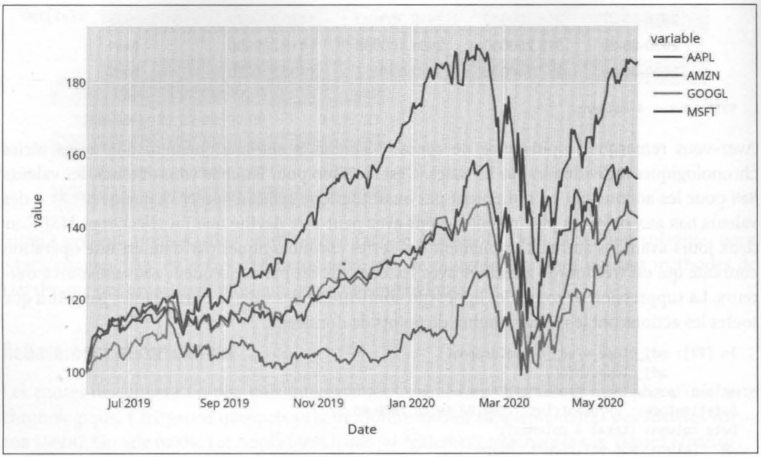


Figure 6.4 : Séries chronologiques rebasées.

Pour voir à quel point les rendements des différentes actions sont indépendants, jetez un coup d'œil à leurs corrélations à l'aide de la méthode `corr`. Malheureusement, pandas ne fournit pas de type de graphique intégré pour visualiser la matrice de corrélation sous la forme d'une carte de chaleur, et nous devons donc utiliser Plotly directement *via* son interface `plotly.express` (voir la Figure 6.5) :

```
In [30]: # Corrélation des rendements logarithmiques journaliers
         returns = np.log(adj_close / adj_close.shift(1))
         returns.corr()
```

Out[30]:

	AAPL	AMZN	GOOGL	MSFT
AAPL	1.000000	0.424910	0.503497	0.486065
AMZN	0.424910	1.000000	0.486690	0.485725
GOOGL	0.503497	0.486690	1.000000	0.525645
MSFT	0.486065	0.485725	0.525645	1.000000

```
In [31]: import plotly.express as px
In [32]: fig = px.imshow(returns.corr(),
                         x=adj_close.columns,
                         y=adj_close.columns,
                         color_continuous_scale=list(
                             reversed(px.colors.sequential.RdBu)),
                         zmin=-1, zmax=1)
         fig.show()
```

Si vous voulez comprendre en détail comment `imshow` fonctionne, consultez la documentation de l'API Plotly Express[1].

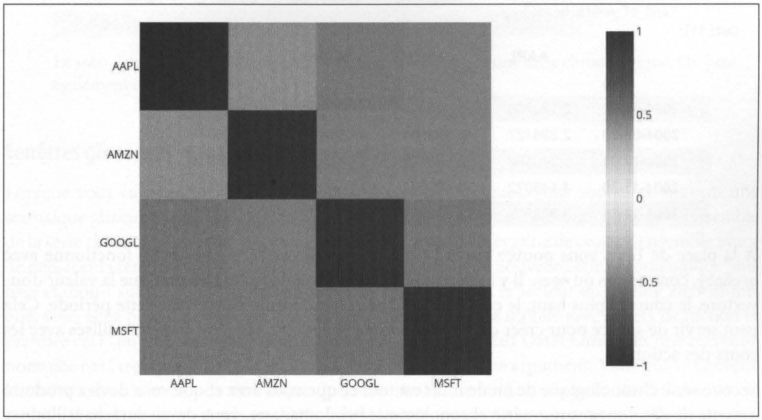


Figure 6.5 : Carte de chaleur de corrélation.

À ce stade, nous avons déjà appris pas mal de choses sur les séries chronologiques, y compris comment les combiner et les nettoyer, ainsi que calculer les rendements et les corrélations. Mais que se passe-t-il si vous décidez que les rendements quotidiens ne constituent pas une bonne base pour votre analyse et que vous préfériez des rendements mensuels ? Nous allons voir dans la section suivante comment modifier la fréquence des données de série chronologique.

Rééchantillonnage

Parmi les tâches courantes avec les séries chronologiques, il y a le *suréchantillonnage* et le *sous-échantillonnage*. Le suréchantillonnage signifie que la série chronologique est convertie en une série avec une fréquence plus élevée, et le sous-échantillonnage signifie qu'elle est convertie en une série avec une fréquence plus basse. Sur les fiches d'informations financières, on voit souvent les performances mensuelles ou trimestrielles, par exemple. Pour transformer la série chronologique quotidienne en une série mensuelle, utilisez la méthode `resample` qui accepte une chaîne de fréquence comme `M` (pour *end-of-calendar-month*, fin du mois civil) ou `BM` (pour *end-of-business-month*, fin du mois d'activité). Vous trouverez une liste de toutes les chaînes de fréquence dans la documentation de pandas[2]. Fonctionnant de la même manière que `groupby`,

1 https://plotly.com/python-api-reference/generated/plotly.express.html
2 https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#dateoffset-objects

on enchaîne ensuite une méthode qui définit la façon dont on rééchantillonne. J'utilise `last` pour toujours prendre la dernière observation de ce mois :

```
In [33]: end_of_month = adj_close.resample("M").last()
         end_of_month.head()
Out[33]:
```

Date	AAPL	AMZN	GOOGL	MSFT
2004-08-31	2.132708	38.139999	51.236237	17.673630
2004-09-30	2.396127	40.860001	64.864868	17.900215
2004-10-31	3.240182	34.130001	95.415413	18.107374
2004-11-30	4.146072	39.680000	91.081078	19.344421
2004-12-31	3.982207	44.290001	96.491493	19.279480

À la place de `last`, vous pouvez choisir n'importe quelle autre méthode qui fonctionne avec `groupby`, comme `sum` ou `mean`. Il y a aussi `ohlc`, qui retourne de façon très pratique la valeur d'ouverture, le cours le plus haut, le cours le plus bas et la valeur de clôture sur cette période. Cela peut servir de source pour créer des graphiques en chandelier qui sont souvent utilisés avec les cours des actions.

Si cette série chronologique de fin de mois est tout ce que vous avez et que vous deviez produire à partir de ces données une série chronologique hebdomadaire, vous devez suréchantillonner votre série chronologique. En utilisant `asfreq`, vous dites à pandas de ne pas appliquer de transformation et vous verrez donc la plupart des valeurs afficher `NaN`. Si vous préférez *remplir* avec la dernière valeur connue, utilisez la méthode `ffill` :

```
In [34]: end_of_month.resample("D").asfreq().head()  # Pas de transformation
Out[34]:
```

Date	AAPL	AMZN	GOOGL	MSFT
2004-08-31	2.132708	38.139999	51.236237	17.67363
2004-09-01	NaN	NaN	NaN	NaN
2004-09-02	NaN	NaN	NaN	NaN
2004-09-03	NaN	NaN	NaN	NaN
2004-09-04	NaN	NaN	NaN	NaN

```
In [35]: end_of_month.resample("W-FRI").ffill().head()  # Remplissage
Out[35]:
```

Date	AAPL	AMZN	GOOGL	MSFT
2004-09-03	2.132708	38.139999	51.236237	17.673630
2004-09-10	2.132708	38.139999	51.236237	17.673630
2004-09-17	2.132708	38.139999	51.236237	17.673630

Date	AAPL	AMZN	GOOGL	MSFT
2004-09-24	2.132708	38.139999	51.236237	17.673630
2004-10-01	2.396127	40.860001	64.864868	17.900215

Le sous-échantillonnage des données est un moyen de lisser une série chronologique. On peut également calculer les statistiques sur une fenêtre glissante.

Fenêtres glissantes

Lorsque vous calculez des statistiques de séries chronologiques, vous souhaitez souvent une statistique glissante telle que la *moyenne mobile*. La moyenne mobile examine un sous-ensemble de la série chronologique (par exemple, 25 jours) et prend la moyenne de ce sous-ensemble avant de déplacer la fenêtre d'un jour vers l'avant. Il en résultera une nouvelle série chronologique plus lisse et moins sujette aux valeurs aberrantes. Si vous êtes dans le secteur du trading algorithmique, vous allez peut-être regarder l'intersection de la moyenne mobile avec le prix de l'action et prendre ceci (ou une variante) comme un signal de trading. Les DataFrames ont une méthode nommée `rolling`, qui accepte le nombre d'observations comme argument. Vous pouvez ensuite la chaîner avec la méthode statistique que vous souhaitez utiliser : dans le cas de la moyenne mobile, il s'agit de la méthode `mean`. En examinant la Figure 6.6, vous pouvez facilement comparer la série chronologique d'origine avec la moyenne mobile lissée :

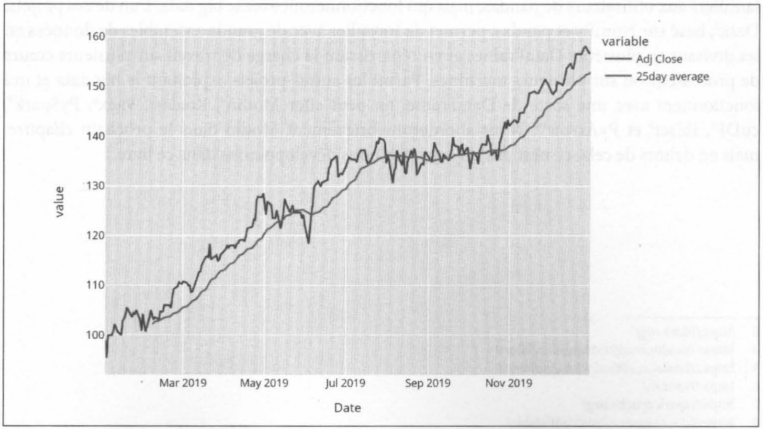


Figure 6.6 : Graphique de la moyenne mobile.

```
In [36]: # Trace la moyenne mobile de MSFT avec des données à partir de 2019
         msft19 = msft.loc["2019", ["Adj Close"]].copy()
         # Ajoute une moyenne mobile de 25 jours
         # dans une nouvelle colonne du DataFrame
         msft19.loc[:, "25day average"] = msft19["Adj Close"].rolling(25).mean()
         msft19.plot()
```

Au lieu de `mean`, vous pouvez utiliser de nombreuses autres mesures statistiques, notamment `count`, `sum`, `median`, `min`, `max`, `std` (écart type) ou `var` (variance).

À ce stade, nous avons vu les fonctionnalités les plus importantes de pandas. Cependant, il est tout aussi important de comprendre les limites de pandas, même si vous êtes encore loin de les avoir atteintes.

Limites de pandas

Lorsque vos DataFrames commencent à prendre du volume, il est souhaitable de connaître la limite supérieure de ce qu'un DataFrame peut contenir. Contrairement à Excel, où vous avez une limite stricte d'environ un million de lignes et 12 000 colonnes par feuille de calcul, pandas n'a qu'une limite souple : toutes les données doivent tenir dans la mémoire disponible de votre ordinateur. Si ce n'est pas le cas, il est facile de remédier à ce problème : chargez uniquement les colonnes de votre jeu de données dont vous avez besoin ou supprimez les résultats intermédiaires pour libérer de la mémoire. Si cela ne suffit pas, il y a plusieurs projets qui sembleront familiers aux utilisateurs de pandas, mais qui fonctionneront avec le big data. L'un de ces projets, Dask[3], basé sur NumPy et pandas, permet de travailler avec de grands ensembles de données en les divisant en plusieurs DataFrames et en répartissant la charge de travail sur plusieurs cœurs de processeur ou sur plusieurs machines. Parmi les autres projets exploitant le big data et qui fonctionnent avec une sorte de DataFrame, on peut citer Modin[4], Koalas[5], Vaex[6], PySpark[7], cuDF[8], Ibiset[9] et PyArrow[10]. Nous aborderons brièvement Modin dans le prochain chapitre, mais en dehors de cela, ce n'est pas un sujet que nous développerons dans ce livre.

3 https://dask.org/
4 https://modin.readthedocs.io/en/latest/
5 https://koalas.readthedocs.io/en/latest/
6 https://vaex.io/
7 https://spark.apache.org/
8 https://docs.rapids.ai/api/cudf/stable/
9 https://ibis-project.org/
10 https://arrow.apache.org/docs/python/

Conclusion

Selon moi, l'analyse des séries chronologiques est le domaine où Excel a le plus pris de retard, si bien qu'après avoir lu ce chapitre, vous comprenez probablement pourquoi pandas recueille un si grand succès dans le secteur de la finance, une industrie qui repose fortement sur les séries chronologiques. Nous avons vu à quel point il est facile de travailler avec des fuseaux horaires, de rééchantillonner des séries chronologiques ou de produire des matrices de corrélation, des fonctionnalités qui ne sont pas prises en charge dans Excel ou qui nécessitent des solutions de contournement fastidieuses.

Cependant, savoir comment utiliser pandas ne signifie pas que vous devez vous débarrasser d'Excel, car les deux univers peuvent très bien cohabiter : les DataFrames de pandas sont un excellent moyen de transférer des données d'un monde à l'autre, comme nous le verrons dans la partie suivante, où nous apprendrons à lire et à écrire des fichiers Excel en contournant entièrement cette application. Ceci est très utile, car cela signifie que vous pouvez manipuler des fichiers Excel avec Python sur tous les systèmes d'exploitation que Python prend en charge, y compris Linux. Pour débuter cette exploration, le chapitre suivant vous montrera comment pandas peut être utilisé pour automatiser des processus manuels fastidieux tels que l'agrégation de fichiers Excel dans des rapports de synthèse.

PARTIE III

Lecture et écriture de fichiers Excel sans Excel

CHAPITRE 7

Manipulation de fichiers Excel avec pandas

Après six chapitres de contenus nourris sur les outils, Python et pandas, nous allons nous accorder une pause et entamer ce chapitre par une étude de cas pratique qui permet de mettre à profit les compétences que vous venez d'acquérir : avec seulement dix lignes de code pandas, vous allez consolider des dizaines de fichiers Excel dans un rapport prêt à être envoyé à votre manager. Après cette étude de cas, je vous présenterai de manière plus approfondie les outils proposés par pandas pour travailler avec des fichiers Excel : la fonction `read_excel` et la classe `ExcelFile` pour la lecture, ainsi que la méthode `to_excel` et la classe `ExcelWriter` pour l'écriture des fichiers. pandas ne s'appuie pas sur l'application Excel pour lire et écrire des fichiers Excel, ce qui signifie que tous les exemples de code de ce chapitre peuvent s'exécuter dans n'importe quel environnement où Python est disponible, notamment Linux.

Étude de cas : génération d'un rapport Excel

Cette étude de cas s'inspire de quelques projets concrets de production de rapports auxquels j'ai participé au cours des dernières années. Même si les projets se sont déroulés dans des secteurs complètement différents, notamment les télécommunications, le marketing digital et la finance, ils étaient toujours très similaires : on part généralement d'un répertoire contenant des fichiers Excel qui doivent être traités afin de produire un rapport Excel, souvent sur une base mensuelle, hebdomadaire ou quotidienne. Dans les exemples de code de ce livre, dans le répertoire *sales_data*, vous trouverez des fichiers Excel stockant des transactions de ventes fictives d'un opérateur de télécommunications vendant différents types d'abonnement (colonne `plan` avec les valeurs Bronze, Silver ou Gold) dans des magasins à travers les États-Unis. Pour chaque mois, il existe deux fichiers, l'un dans le sous-dossier *new* pour les nouveaux contrats et l'autre dans le sous-dossier *existing* pour les clients existants. Comme les rapports proviennent de différents systèmes, ils sont fournis dans différents formats : les nouveaux clients sont stockés sous la forme de fichiers *xlsx*, tandis que les clients existants sont enregistrés dans l'ancien format *xls*.

Chacun des fichiers comporte jusqu'à 10 000 transactions, et notre objectif est de produire un rapport Excel qui affiche les ventes totales par magasin et par mois. Pour commencer, jetons un coup d'œil au fichier *January.xlsx* du sous-dossier *new* qui est illustré à la Figure 7.1.

	A	B	C	D	E	F	G
1	transaction_id	store	status	transaction_date	plan	contract_type	amount
2	abfbdd6d	Chicago	ACTIVE	01/01/2019	Silver	NEW	14,25
3	136a9997	San Francisco	ACTIVE	01/01/2019	Gold	NEW	19,35
4	c6688f32	San Francisco	ACTIVE	01/01/2019	Bronze	NEW	12,2
5	6ef349c1	Chicago	ACTIVE	01/01/2019	Gold	NEW	19,35
6	22066f29	San Francisco	ACTIVE	01/01/2019	Silver	NEW	14,25

Figure 7.1 : Premières lignes du fichier Excel January.xlsx.

Les fichiers Excel du sous-dossier *existing* sont pratiquement identiques, sauf qu'il leur manque la colonne `status` et qu'ils sont stockés dans l'ancien format *xls*. Dans un premier temps, nous allons lire les nouvelles transactions du fichier *January.xlsx* avec la fonction `read_excel` de pandas :

```
In [1]: import pandas as pd
In [2]: df = pd.read_excel("sales_data/new/January.xlsx")
        df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9493 entries, 0 to 9492
Data columns (total 7 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   transaction_id    9493 non-null   object
 1   store             9493 non-null   object
 2   status            9493 non-null   object
 3   transaction_date  9493 non-null   datetime64[ns]
 4   plan              9493 non-null   object
 5   contract_type     9493 non-null   object
 6   amount            9493 non-null   float64
dtypes: datetime64[ns](1), float64(1), object(5)
memory usage: 519.3+ KB
```



Il s'agit du même avertissement que dans le Chapitre 5 : si vous exécutez `pd.read_excel` avec Python 3.9 ou une version supérieure, assurez-vous d'utiliser au moins pandas 1.2, sinon vous obtiendrez une erreur lors de la lecture des fichiers xlsx.

Comme vous pouvez le voir, pandas a correctement reconnu les types de données de toutes les colonnes, y compris le format de date de `transaction_date`. Cela nous permet d'exploiter les données sans travail préparatoire supplémentaire. Comme cet exemple est délibérément simple, nous pouvons passer à la création d'un script court nommé *sales_report_pandas.py* dont le code

est listé dans l'Exemple 7.1. Ce script lit tous les fichiers Excel des deux répertoires, agrège les données et enregistre le tableau récapitulatif dans un nouveau fichier Excel. Utilisez VS Code pour saisir vous-même le script ou ouvrez-le à partir des exemples de code. Pour un rappel sur la création ou l'ouverture de fichiers dans VS Code, reportez-vous au Chapitre 2. Si vous saisissez vous-même le code, veillez à l'enregistrer au même niveau hiérarchique que le dossier *sales_data*, ce qui vous permettra d'exécuter le script sans avoir à modifier les chemins d'accès aux fichiers.

Exemple 7.1 : sales_report_pandas.py.

```
from pathlib import Path

import pandas as pd


# Répertoire de ce fichier
this_dir = Path(__file__).resolve().parent ❶

# Lecture des fichiers Excel à partir des sous-dossiers de sales_data
parts = []
for path in (this_dir / "sales_data").rglob("*.xls*"): ❷
    print(f'Reading {path.name}')
    part = pd.read_excel(path, index_col="transaction_id")
    parts.append(part)

# Combine les DataFrames de chaque fichier en un seul DataFrame
# pandas aligne proprement les colonnes
df = pd.concat(parts)

# Convertit chaque magasin en une colonne et
# additionne toutes les transactions par date
pivot = pd.pivot_table(df,
                       index="transaction_date", columns="store",
                       values="amount", aggfunc="sum")

# On réduit l'échantillon à la fin du mois
# et on assigne un nom d'index
summary = pivot.resample("M").sum()
summary.index.name = "Month"

# On écrit le rapport de synthèse dans un fichier Excel
summary.to_excel(this_dir / "sales_report_pandas.xlsx")
```

❶ Jusqu'à présent, j'utilisais des chaînes pour spécifier les chemins d'accès aux fichiers. En utilisant à la place la classe `Path` du module `pathlib` de la bibliothèque standard, vous avez accès à un ensemble puissant d'outils : les objets `path` permettent de construire facilement des chemins en concaténant des parties individuelles avec des slashs, comme cela se fait quatre lignes plus bas avec `this_dir / "sales_data"`. Ces chemins fonctionnent sur plusieurs plateformes et permettent d'appliquer des filtres tels que `rglob`, dont le fonctionnement est expliqué au point suivant. `__file__` résout le chemin d'accès du fichier du code source lorsque

vous l'exécutez, l'emploi de `parent` donnant le nom du répertoire de ce fichier. La méthode `resolve` que nous utilisons avant d'appeler `parent` transforme le chemin d'accès en un chemin d'accès absolu. Si vous exécutez ce code à partir d'un notebook Jupyter, vous devez remplacer cette ligne par `this_dir = Path(".").resolve()`, avec le point représentant le répertoire actif. Dans la plupart des cas, les fonctions et les classes qui acceptent un chemin d'accès sous la forme d'une chaîne acceptent également un objet `path`.

❷ Le moyen le plus simple de lire tous les fichiers Excel de manière récursive à partir d'un répertoire consiste à utiliser la méthode `rglob` de l'objet `path`. `glob` est l'abréviation de *globbing*, qui fait référence à l'expansion d'un chemin d'accès à l'aide de caractères génériques. Le caractère générique ? représente exactement un caractère, tandis que * représente n'importe quel nombre de caractères (y compris zéro, c'est-à-dire une chaîne vide). Le *r* dans `rglob` signifie *récursif*, c'est-à-dire que la recherche des fichiers correspondants s'effectue dans tous les sous-répertoires (par conséquent, `glob` ignorerait les sous-répertoires). L'utilisation de `*.xls*` comme expression avec `rglob` permet de s'assurer que les anciens et les nouveaux fichiers Excel sont trouvés, car ils correspondent à la fois à `.xls` et `.xlsx`. Il est cependant conseillé d'améliorer l'expression sous la forme `[!~$]*.xls*`, car cela ignore les fichiers Excel temporaires (leur nom de fichier commence par ~$). Pour plus d'informations sur l'utilisation du *globbing* en Python, consultez la documentation Python[1].

Exécutez le script, par exemple en cliquant sur le bouton Exécuter le script Python dans un terminal, en haut à droite de VS Code. L'exécution du script prend un certain temps, et une fois terminée, le classeur Excel *sales_report_pandas.xlsx* apparaît dans le même répertoire que le script. Le contenu de la feuille Sheet1 doit ressembler à ce qui est illustré à la Figure 7.2. C'est un résultat plutôt impressionnant pour seulement une dizaine de lignes de code, même si vous devez ajuster la largeur de la première colonne pour pouvoir voir les dates !

Pour des cas simples comme celui-ci, pandas offre une solution vraiment facile pour l'exploitation de fichiers Excel. Cependant, nous pouvons faire beaucoup mieux, et on pourrait améliorer les choses avec un titre, une mise en forme plus adéquate (notamment la largeur des colonnes et le nombre de décimales) ainsi qu'un graphique. C'est exactement ce que nous ferons dans le prochain chapitre en utilisant directement les bibliothèques d'écriture que pandas emploie sous le capot. Cependant, avant d'en arriver là, examinons plus en détail la manière dont nous pouvons lire et écrire des fichiers Excel avec pandas.

1 https://docs.python.org/3/library/glob.html

	A	B	C	D	E	F	G
1	Month	Boston	Chicago	Las Vegas	New York	an Francis	ashington DC
2	########	21784,1	51187,7	23012,75	49872,85	58629,85	14057,6
3	########	21454,9	52330,85	25493,1	46669,85	55218,65	15235,4
4	########	20043	48897,25	23451,1	41572,25	52712,95	14177,05
5	########	18791,05	47396,35	22710,15	41714,3	49324,65	13339,15
6	########	18036,75	45117,05	21526,55	40610,4	47759,6	13147,1
7	########	21556,25	49460,45	21985,05	47265,65	53462,4	14284,3
8	########	19853	47993,8	23444,3	40408,3	50181,6	14161,5
9	########	22332,9	50838,9	24927,65	45396,85	55336,35	16127,05
10	########	19924,5	49096,25	24410,7	42830,6	49931,45	14994,4
11	########	16550,95	42543,8	22827,5	34090,05	44311,65	12846,7
12	########	21312,9	52011,6	24860,25	46959,85	55056,45	14057,6
13	########	19722,6	49355,1	24535,75	42364,35	50933,45	14702,15

Figure 7.2 : Fichier Excel sales_report_pandas.xlsx (en l'état, sans ajustement de la largeur des colonnes).

Lecture et écriture de fichiers Excel avec pandas

L'étude de cas utilise `read_excel` et `to_excel` avec leurs arguments par défaut afin que cet exemple reste simple. Dans cette section, je vais vous montrer les arguments et les options les plus couramment utilisés lors de la lecture et de l'écriture de fichiers Excel avec pandas. Nous allons commencer par la fonction `read_excel` et la classe `ExcelFile` avant d'examiner la méthode `to_excel` et la classe `ExcelWriter`. En cours de route, je vais également présenter l'instruction `with` de Python.

Fonction read_excel et classe ExcelFile

L'étude de cas a utilisé des classeurs Excel où les données se trouvent commodément dans la cellule A1 de la première feuille. Dans la réalité, vos fichiers Excel ne sont probablement pas aussi bien organisés. Dans ce cas, pandas propose des paramètres pour affiner le processus de lecture. Pour les prochains exemples, nous allons utiliser le fichier *stores.xlsx* que vous trouverez dans le dossier *xl* des exemples de code. La première feuille est illustrée à la Figure 7.3.

	A	B	C	D	E	F
1						
2		Store	Employees	Manager	Since	Flagship
3		New York	10	Sarah	20/07/2018	FAUX
4		San Francisco	12	Neriah	02/11/2019	MISSING
5		Chicago	4	Katelin	31/01/2020	
6		Boston	5	Georgiana	01/04/2017	VRAI
7		Washington DC	3	Evan		FAUX
8		Las Vegas	11	Paul	06/01/2020	FAUX
9						

Figure 7.3 : Première feuille de calcul du fichier stores.xlsx.

En utilisant les paramètres `sheet_name`, `skiprowset` et `usecols`, nous pouvons indiquer à pandas la plage de cellules que nous voulons lire. Comme d'habitude, il est judicieux d'examiner les types de données du DataFrame retourné en exécutant la méthode `info` :

```
In [3]: df = pd.read_excel("xl/stores.xlsx",
                           sheet_name="2019", skiprows=1, usecols="B:F")
        df
Out[3]:
```

	Store	Employees	Manager	Since	Flagship
0	New York	10	Sarah	2018-07-20	False
1	San Francisco	12	Neriah	2019-11-02	MISSING
2	Chicago	4	Katelin	2020-01-31	NaN
3	Boston	5	Georgiana	2017-04-01	True
4	Washington DC	3	Evan	NaT	False
5	Las Vegas	11	Paul	2020-01-06	False

```
In [4]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Store      6 non-null      object
 1   Employees  6 non-null      int64
 2   Manager    6 non-null      object
 3   Since      5 non-null      datetime64[ns]
 4   Flagship   5 non-null      object
dtypes: datetime64[ns](1), int64(1), object(3)
memory usage: 368.0+ bytes
```

Tout semble correct, à l'exception de la colonne `Flagship` (qui indique s'il s'agit d'un magasin principal), dont le type de données doit être `bool` plutôt que `object`. Pour résoudre ce problème, nous pouvons fournir une fonction de conversion qui traite les cellules erronées dans cette colonne (au lieu d'écrire la fonction `fix_missing`, nous aurions également pu fournir une expression lambda) :

```
In [5]: def fix_missing(x):
            return False if x in ["", "MISSING"] else x
In [6]: df = pd.read_excel("xl/stores.xlsx",
                           sheet_name="2019", skiprows=1, usecols="B:F",
                           converters={"Flagship": fix_missing})
        df
Out[6]:
```

	Store	Employees	Manager	Since	Flagship
0	New York	10	Sarah	2018-07-20	False
1	San Francisco	12	Neriah	2019-11-02	False
2	Chicago	4	Katelin	2020-01-31	False
3	Boston	5	Georgiana	2017-04-01	True
4	Washington DC	3	Evan	NaT	False
5	Las Vegas	11	Paul	2020-01-06	False

```
In [7]: # La colonne Flagship a maintenant le type de données "bool"
        df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Store      6 non-null      object
 1   Employees  6 non-null      int64
 2   Manager    6 non-null      object
 3   Since      5 non-null      datetime64[ns]
 4   Flagship   6 non-null      bool
dtypes: bool(1), datetime64[ns](1), int64(1), object(2)
memory usage: 326.0+ bytes
```

La fonction `read_excel` accepte également une liste de noms de feuilles. Dans ce cas, elle retourne un dictionnaire avec le DataFrame comme valeur et le nom de la feuille comme clé. Pour lire toutes les feuilles, vous devez fournir `sheet_name=None`. Notez également la légère variation de la façon dont j'emploie `usecols` en fournissant les noms de colonnes de la table :

```
In [8]: sheets = pd.read_excel("xl/stores.xlsx", sheet_name=["2019", "2020"],
                               skiprows=1, usecols=["Store", "Employees"])
        sheets["2019"].head(2)
```

Out[8]:

	Store	Employees
0	New York	10
1	San Francisco	12

Si le fichier source n'a pas d'en-têtes de colonne, définissez `header` à `None` et fournissez-les grâce à des noms. Notez que `sheet_name` accepte également les index de feuilles :

```
In [9]: df = pd.read_excel("xl/stores.xlsx", sheet_name=0,
                           skiprows=2, skipfooter=3,
                           usecols="B:C,F", header=None,
                           names=["Branch", "Employee_Count", "Is_Flagship"])
        df
```

Out[9]:

	Branch	Employee_Count	Is_Flagship
0	New York	10	False
1	San Francisco	12	MISSING
2	Chicago	4	NaN

Pour gérer les valeurs `NaN`, utilisez une combinaison de `na_values` et `keep_default_na`. L'exemple suivant indique à pandas d'interpréter uniquement les cellules contenant le mot `MISSING` et rien d'autre comme `NaN` :

```
In [10]: df = pd.read_excel("xl/stores.xlsx", sheet_name="2019",
                            skiprows=1, usecols="B,C,F", skipfooter=2,
                            na_values="MISSING", keep_default_na=False)
         df
```

Out[10]:

	Store	Employees	Flagship
0	New York	10	False
1	San Francisco	12	NaN
2	Chicago	4	
3	Boston	5	True

pandas offre une autre façon de lire des fichiers Excel en utilisant la classe `ExcelFile`. Cela fait surtout une différence si vous souhaitez lire plusieurs feuilles à partir d'un fichier enregistré dans l'ancien format *xls* : dans ce cas, l'utilisation d'`ExcelFile` sera plus rapide car elle empêche pandas de lire la totalité du fichier plusieurs fois. `ExcelFile` peut être utilisée comme gestionnaire de contexte (voir l'encadré) afin que le fichier soit correctement fermé.

Gestionnaires de contexte et instruction with

Commençons par dire que l'instruction `with` en Python n'a rien à voir avec l'instruction `With` en VBA où elle est utilisée pour exécuter une série d'instructions sur le même objet, tandis qu'en Python, elle est employée pour gérer des ressources telles que des fichiers ou des connexions de base de données. Si vous souhaitez charger les dernières données de vente afin de pouvoir les analyser, vous devrez peut-être ouvrir un fichier ou établir une connexion à une base de données. Une fois que vous avez terminé de lire les données, il est recommandé de fermer le fichier ou la connexion dès que possible. Sinon, vous pouvez rencontrer des situations où vous ne pourrez pas ouvrir un autre fichier ou établir une autre connexion à la base de données, les handlers de fichiers et les connexions de base de données étant des ressources limitées. L'ouverture et la fermeture manuelles d'un fichier texte fonctionnent de la manière suivante (`w` signifie que le fichier est ouvert en mode écriture, c'est-à-dire qu'il remplace le fichier s'il existe déjà) :

```
In [11]: f = open("output.txt", "w")
         f.write("Du texte")
         f.close()
```

L'exécution de ce code crée un fichier appelé *output.txt* dans le même répertoire que le notebook à partir duquel vous l'exécutez et écrit dans ce fichier « Du texte ». Pour *lire* un fichier, vous devez utiliser `r` au lieu de `w`, et pour *ajouter* du contenu à la fin du fichier, utiliser `a`. Étant donné que les fichiers peuvent également être manipulés en dehors de votre programme, une telle opération peut échouer. Vous pouvez gérer cela en utilisant le mécanisme try/except que je présenterai au Chapitre 11. Cependant, comme il s'agit d'une opération très courante, Python fournit l'instruction `with` pour faciliter les choses :

```
In [12]: with open("output.txt", "w") as f:
             f.write("Du texte")
```

Lorsque l'exécution de code quitte le corps de l'instruction `with`, le fichier est automatiquement fermé, qu'une exception se produise ou non. Cela garantit que les ressources sont correctement gérées et que le nettoyage s'effectue automatiquement. Les objets qui prennent en charge l'instruction `with` sont appelés *gestionnaires de contexte* ; cela inclut notamment les objets `ExcelFile` et `ExcelWriter` étudiés dans ce chapitre, ainsi que les objets de connexion de base de données que nous aborderons au Chapitre 11.

Voyons à présent comment fonctionne la classe ExcelFile :

```
In [13]: with pd.ExcelFile("xl/stores.xls") as f:
             df1 = pd.read_excel(f, "2019", skiprows=1, usecols="B:F", nrows=2)
             df2 = pd.read_excel(f, "2020", skiprows=1, usecols="B:F", nrows=2)

         df1
```

```
Out[13]:
```

	Store	Employees	Manager	Since	Flagship
0	New York	10	Sarah	2018-07-20	False
1	San Francisco	12	Neriah	2019-11-02	MISSING

`ExcelFile` donne également accès aux noms de toutes les feuilles :

```
In [14]: stores = pd.ExcelFile("xl/stores.xlsx")
         stores.sheet_names
Out[14]: ['2019', '2020', '2019-2020']
```

Enfin, pandas permet de lire des fichiers Excel à partir d'une URL, comme nous l'avons fait avec des fichiers CSV au Chapitre 5. Nous pouvons ainsi lire le fichier directement à partir du référentiel de code de l'ouvrage :

```
In [15]: url = ("https://raw.githubusercontent.com/fzumstein/"
                "python-for-excel/1st-edition/xl/stores.xlsx")
         pd.read_excel(url, skiprows=1, usecols="B:E", nrows=2)
Out[15]:
```

	Store	Employees	Manager	Since
0	New York	10	Sarah	2018-07-20
1	San Francisco	12	Neriah	2019-11-02



Si vous utilisez pandas avec une version inférieure à 1.3, la lecture des fichiers xlsb nécessite de spécifier explicitement le nom du moteur dans la fonction `read_excel` ou la classe `ExcelFile` :

```
pd.read_excel("xl/stores.xlsb", engine="pyxlsb ")
```

Cela nécessite l'installation du package pyxlsb, car il ne fait pas partie d'Anaconda (nous y reviendrons dans le chapitre suivant et nous évoquerons aussi les autres moteurs).

À titre de résumé, le Tableau 7.1 affiche les paramètres `read_excel` les plus couramment utilisés. Vous trouverez la liste complète dans la documentation officielle[2].

Tableau 7.1 : Paramètres les plus courants de read_excel.

Paramètre	Description
`sheet_name`	Au lieu de fournir un nom de feuille, vous pouvez également fournir l'index de la feuille (à partir de zéro), par exemple, `sheet_name= 0`. Si vous définissez `sheet_name=None`, pandas lit l'ensemble du classeur et renvoie un dictionnaire sous la forme `{"sheetname": df}`. Pour lire une sélection de feuilles, fournissez une liste avec des noms de feuilles ou des index.
`skiprows`	Permet d'ignorer le nombre de lignes indiqué.

2 https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html#pandas.read_excel

Paramètre	Description
`usecols`	Si le fichier Excel comprend des noms d'en-têtes de colonnes, fournissez-les dans une liste pour sélectionner les colonnes, par exemple, `["Store", "Employees"]`. Vous pouvez aussi indiquer une liste d'index de colonnes, par exemple, `[1, 2]`, ou une chaîne (mais pas de liste !) de noms de colonnes Excel, y compris des plages, par exemple, `"B:D, G"`. Vous pouvez également fournir une fonction : par exemple, pour inclure uniquement les colonnes qui commencent par `Manager`, utilisez : `usecols=lambda x: x.startswith(«Manager»)`.
`nrows`	Nombre de lignes que vous souhaitez lire.
`index_col`	Indique quelle colonne doit être l'index, accepte un nom de colonne ou un index, par exemple, `index_col=0`. Si vous fournissez une liste avec plusieurs colonnes, un index hiérarchique est créé.
`header`	Si vous définissez `header=None`, les en-têtes par défaut sont assignés, sauf si vous fournissez les noms souhaités dans le paramètre `names`. Si vous fournissez une liste d'index, des en-têtes de colonnes hiérarchiques sont créés.
`names`	Fournissez les noms souhaités de vos colonnes sous forme de liste.
`na_values`	pandas interprète par défaut à `NaN` (j'ai présenté `NaN` au Chapitre 5) les valeurs des cellules suivantes : cellules vides, `#NA`, `NA`, `null`, `#N/A`, `N/A`, `NaN`, `n/a`, `-NaN`, `1.#IND`, `nan`, `#N/A N/A`, `-1.#QNAN`, `-nan`, `NULL`, `-1.#IND`, `<NA>`, `1.#QNAN`. Si vous souhaitez ajouter une ou plusieurs valeurs à cette liste, fournissez-les dans `na_values`.
`keep_default_na`	Si vous souhaitez ignorer les valeurs par défaut que pandas interprète comme `NaN`, définissez `keep_default_na = False`.
`convert_float`	Excel stocke tous les nombres en interne sous forme de nombres à virgule flottante et, par défaut, pandas transforme les nombres sans décimales significatives en entiers. Si vous souhaitez modifier ce comportement, définissez `convert_float=False` (votre programme pourra s'exécuter un peu plus rapidement).
`converters`	Permet de fournir une fonction par colonne pour convertir ses valeurs. Par exemple, pour convertir le texte en majuscules dans une colonne, utilisez `converters={"column_name": lambda x: x.upper()}`.

Voilà tout ce qu'il faut savoir sur la lecture de fichiers Excel avec pandas, et nous allons à présent apprendre à écrire des fichiers dans la section suivante.

Méthode to_excel et classe ExcelWriter

Le moyen le plus simple d'écrire un fichier Excel avec pandas consiste à utiliser la méthode `to_excel` d'un DataFrame, qui permet de spécifier dans quelle cellule de la feuille vous souhaitez écrire le DataFrame. Vous pouvez également décider d'inclure ou non les en-têtes de colonnes et l'index du DataFrame et choisir la manière de traiter les types de données tels que `np.nan` et `np.inf` qui n'ont pas de représentation équivalente dans Excel. Commençons par créer un DataFrame avec différents types de données et utilisons sa méthode `to_excel` :

```
In [16]: import numpy as np
         import datetime as dt
In [17]: data=[[dt.datetime(2020,1,1, 10, 13), 2.222, 1, True],
               [dt.datetime(2020,1,2), np.nan, 2, False],
               [dt.datetime(2020,1,2), np.inf, 3, True]]
         df = pd.DataFrame(data=data,
                           columns=["Dates", "Floats", "Integers", "Booleans"])
         df.index.name="index"
         df
```

```
Out[17]:
```

index	Dates	Floats	Integers	Booleans
0	2020-01-01 10:13:00	2.222	1	True
1	2020-01-02 00:00:00	NaN	2	False
2	2020-01-02 00:00:00	inf	3	True

```
In [18]: df.to_excel("written_with_pandas.xlsx", sheet_name="Output",
                     startrow=1, startcol=1, index=True, header=True,
                     na_rep="<NA>", inf_rep="<INF>")
```

L'exécution de la commande `to_excel` crée le fichier Excel illustré à la Figure 7.4 (vous devrez élargir la colonne C pour voir les dates correctement) :

	A	B	C	D	E	F
1						
2		index	Dates	Floats	Integers	Booleans
3		0	2020-01-01 10:13:00	2,222	1	VRAI
4		1	2020-01-02 00:00:00	<NA>	2	FAUX
5		2	2020-01-02 00:00:00	<INF>	3	VRAI

Figure 7.4 : Fichier Excel written_with_pandas.xlsx.

Si vous souhaitez écrire plusieurs DataFrames dans les mêmes feuilles ou dans des feuilles différentes, vous devez utiliser la classe `ExcelWriter`. L'exemple suivant écrit le même DataFrame à deux emplacements différents sur la feuille Sheet1 et une fois sur la feuille Sheet2 :

```
In [19]: with pd.ExcelWriter("written_with_pandas2.xlsx") as writer:
             df.to_excel(writer, sheet_name="Sheet1", startrow=1, startcol=1)
             df.to_excel(writer, sheet_name="Sheet1", startrow=10, startcol=1)
             df.to_excel(writer, sheet_name="Sheet2")
```

Étant donné que nous utilisons la classe `ExcelWriter` comme gestionnaire de contexte, le fichier est automatiquement écrit sur le disque lorsqu'il quitte le gestionnaire de contexte, c'est-à-dire après la dernière indentation du code. Dans le cas contraire, vous devez appeler `writer.save()` explicitement. Pour obtenir une synthèse des paramètres de `to_excel` les plus couramment utilisés, consultez le Tableau 7.2. Vous trouverez la liste complète de ces paramètres dans la documentation officielle[3].

3 https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html#pandas.read_excel

Tableau 7.2 : Paramètres les plus courants de to_excel.

Paramètre	Description
`sheet_name`	Nom de la feuille dans laquelle écrire.
`startrow` et `startcol`	`startrow` est la première ligne où le DataFrame est écrit et `startcol` est la première colonne. On utilise l'indexation à partir de zéro, si bien que lorsque vous souhaitez écrire votre DataFrame dans la cellule B3, il faut employer `startrow = 2` et `startcol = 1`.
`index` et `header`	Si vous souhaitez masquer l'index ou l'en-tête, définissez-les respectivement à `index=False` et `header=False`.
`na_rep` et `inf_rep`	Par défaut, `np.nan` est converti en cellule vide, tandis que `np.inf`, la représentation de l'infini de NumPy, est converti dans la chaîne `inf`. En fournissant des valeurs, vous pouvez modifier ce comportement.
`freeze_panes`	Fige les deux premières lignes et colonnes en fournissant un tuple : par exemple `(2, 1)` fige les deux premières lignes et la première colonne.

Comme vous pouvez le voir, la lecture et l'écriture de fichiers Excel simples avec pandas fonctionnent bien. Il y a cependant quelques limites que nous allons détailler.

Limites lors de l'utilisation de pandas avec des fichiers Excel

L'utilisation de pandas pour lire et écrire des fichiers Excel fonctionne très bien pour les cas simples, mais il y a des limites :

- Lorsque vous écrivez des DataFrames dans des fichiers, vous ne pouvez pas inclure de titre ou de graphique.
- Il n'existe aucun moyen de modifier le format par défaut de l'en-tête et de l'index dans Excel.
- Lors de la lecture de fichiers, pandas transforme automatiquement les cellules comportant des erreurs comme `#REF!` ou `#NUM!` en `NaN`, ce qui rend impossible la recherche d'erreurs spécifiques dans vos feuilles de calcul.
- L'exploitation de fichiers Excel volumineux peut nécessiter des paramètres supplémentaires qui sont plus faciles à contrôler en utilisant directement les packages de lecture et d'écriture, comme nous le verrons dans le chapitre suivant.

Conclusion

Ce qui est bien avec pandas, c'est qu'il offre une interface cohérente pour exploiter tous les formats de fichiers Excel pris en charge, qu'il s'agisse de fichiers *xls*, *xlsx*, *xlsm* ou *xlsb*. Cela nous a permis de lire facilement un répertoire de fichiers Excel, d'agréger des données et de transférer la synthèse dans un rapport Excel, en seulement dix lignes de code.

pandas, cependant, ne fait pas l'essentiel du travail tout seul : sous le capot, il sélectionne un package de lecture ou d'écriture pour assurer les tâches. Dans le chapitre suivant, je vais vous montrer les packages de lecture et d'écriture employés par pandas et comment les utiliser directement ou en combinaison avec pandas. Cela nous permettra de contourner les limites que nous avons relevées dans la section précédente.

CHAPITRE 8

Manipulation de fichiers Excel avec des packages de lecture et d'écriture

Ce chapitre présente les packages OpenPyXL, XlsxWriter, pyxlsb, xlrd et xlwt qui permettent de lire et d'écrire des fichiers Excel et qui sont utilisés par pandas sous le capot lorsque vous appelez les fonctions `read_excel` ou `to_excel`. L'utilisation directe des packages de lecture et d'écriture permet de créer des rapports Excel plus complexes et d'optimiser le processus de lecture. De plus, si jamais vous travaillez sur un projet où il suffit de lire et d'écrire des fichiers Excel sans avoir besoin du reste des fonctionnalités de pandas, l'installation complète de NumPy et de pandas est probablement superflue. Nous débuterons ce chapitre en apprenant à choisir quel package utiliser et en étudiant le fonctionnement de leur syntaxe, avant d'examiner quelques fonctionnalités avancées, notamment l'exploitation de gros fichiers Excel et la manière de combiner pandas avec les packages de lecture et d'écriture afin d'améliorer le style des DataFrames. Pour conclure, nous reprendrons l'étude de cas du début du chapitre précédent et nous améliorerons le rapport Excel en mettant en forme le tableau et en ajoutant un graphique. Comme pour le chapitre précédent, celui-ci ne nécessite pas d'installation d'Excel, ce qui signifie que tous les exemples de code s'exécutent sous Windows, macOS et Linux.

Packages de lecture et d'écriture

Le nombre de packages de lecture et d'écriture peut sembler impressionnant, car nous allons en examiner pas moins de six dans cette section, presque tous les types de fichiers Excel nécessitant un package différent. Le fait que chaque package utilise une syntaxe différente, qui s'écarte souvent de manière substantielle du modèle objet d'Excel d'origine, ne facilite pas les choses (j'en dirai plus sur le modèle objet d'Excel dans le prochain chapitre). Cela signifie que vous devrez probablement consulter la documentation de nombreuses commandes, même si vous êtes un développeur VBA chevronné. Cette section commence par une vue d'ensemble des cas où vous

avez besoin de tel ou tel package avant d'introduire un module d'assistance qui facilite un peu l'utilisation de ces packages. Après cela, je vous présenterai chacun des packages sous la forme d'un livre de recettes, où vous pouvez consulter le fonctionnement des commandes les plus couramment utilisées.

Savoir quel package utiliser

Cette section présente six packages pour lire, écrire et modifier des fichiers Excel :

- OpenPyXL[1]
- XlsxWriter[2]
- pyxlsb[3]
- xlrd[4]
- xlwt[5]
- xlutils[6]

Pour comprendre quel package peut faire quoi, consultez le Tableau 8.1. Par exemple, pour lire le format de fichier *xlsx*, vous devez utiliser le package OpenPyXL :

Tableau 8.1 : Quand utiliser quel package.

Format de fichier Excel	Lecture	Écriture	Modification
xlsx	OpenPyXL	OpenPyXL, XlsxWriter	OpenPyXL
xlsm	OpenPyXL	OpenPyXL, XlsxWriter	OpenPyXL
xltx, xltm	OpenPyXL	OpenPyXL	OpenPyXL
xlsb	pyxlsb	-	-
xls, xlt	xlrd	xlwt	xlutils

Si vous souhaitez écrire des fichiers *xlsx* ou *xlsm*, vous devez choisir entre OpenPyXL et XlsxWriter. Les deux packages couvrent des fonctionnalités similaires, mais chaque package a quelques fonctionnalités uniques que l'autre n'a pas. Comme les deux bibliothèques sont encore en cours de développement, leur fonctionnement évolue au fil des modifications. Voici un résumé de quelques-unes de leurs différences :

- OpenPyXL peut lire, écrire et modifier tandis que XlsxWriter peut uniquement écrire.
- OpenPyXL facilite la production de fichiers Excel contenant des macros VBA.

1 https://openpyxl.readthedocs.io/en/stable/
2 https://xlsxwriter.readthedocs.io/
3 https://github.com/willtrnr/pyxlsb
4 https://xlrd.readthedocs.io/en/latest/
5 https://xlwt.readthedocs.io/en/latest/
6 https://xlutils.readthedocs.io/en/latest/

- XlsxWriter a une meilleure documentation.
- XlsxWriter a tendance à être plus rapide qu'OpenPyXL, mais en fonction de la taille du classeur, les différences peuvent être insignifiantes.



Si vous vous demandez où se situe xlwings dans le Tableau 8.1, alors la réponse est nulle part ou partout, en fonction de votre scénario d'utilisation : contrairement à tous les packages étudiés dans ce chapitre, xlwings dépend de l'application Excel, qui n'est souvent pas disponible, notamment si vous devez exécuter vos scripts sous Linux. Si, en revanche, vous pouvez exécuter vos scripts sous Windows ou macOS où vous disposez d'Excel, xlwings peut en effet être utilisé comme alternative à tous les packages de ce chapitre. Étant donné que la dépendance par rapport à Excel est une différence fondamentale entre xlwings et tous les autres packages Excel, je présenterai xlwings dans le chapitre suivant, qui entame la Partie IV de ce livre.

pandas utilise le package d'écriture qu'il peut trouver et si vous avez installé à la fois OpenPyXL et XlsxWriter, pandas emploiera XlsxWriter par défaut. Si vous souhaitez choisir le package que pandas doit utiliser, spécifiez le paramètre `engine` dans les fonctions `read_excel` ou `to_excel` ou les classes `ExcelFile` et `ExcelWriter`. Le paramètre `engine` est le nom du package en minuscules, et pour écrire un fichier avec OpenPyXL à la place de XlsxWriter, vous devez donc exécuter la ligne suivante :

```
df.to_excel("nom_fichier.xlsx", engine="openpyxl")
```

Une fois que vous savez de quel package vous avez besoin, un deuxième défi vous attend : la plupart de ces packages vous obligent à écrire quelques lignes de code pour lire ou écrire une plage de cellules, et chaque package utilise une syntaxe différente. Pour vous faciliter la tâche, j'ai créé un module d'assistance que je présente dans la section suivante.

Module excel.py

J'ai créé le module `excel.py` pour vous faciliter la vie lors de l'utilisation des packages de lecture et d'écriture, car il prend en charge les problèmes suivants :

Permutation entre les packages

Devoir changer de package de lecture ou d'écriture est un scénario relativement courant. Par exemple, les fichiers Excel ont tendance à prendre du poids au fil du temps, ce que de nombreux utilisateurs combattent en passant du format de fichier *xlsx* à *xlsb*, car cela peut réduire considérablement la taille du fichier. Dans ce cas, vous devrez passer d'OpenPyXL à pyxlsb. Cela vous oblige à réécrire votre code OpenPyXL pour refléter la syntaxe de pyxlsb.

Conversion de type de données

Ceci est lié au point précédent : lorsque vous changez de package, vous n'avez pas seulement à ajuster la syntaxe de votre code, mais vous devez également faire attention aux différents types de données que ces packages renvoient pour un même contenu. Par exemple, OpenPyXL renvoie `None` pour les cellules vides, tandis que xlrd renvoie une chaîne vide.

Parcours de cellules

Les packages de lecture et d'écriture sont des packages de *bas niveau* : cela signifie qu'ils manquent de fonctions pratiques qui permettent de gérer facilement les tâches courantes. Par exemple, la plupart des packages nécessitent que vous fassiez une boucle pour parcourir chaque cellule d'une plage que vous allez lire ou modifier.

Vous trouverez le module `excel.py` dans les exemples de code et nous l'utiliserons dans les prochaines sections, mais, en guise d'aperçu, voici déjà la syntaxe pour lire et écrire des valeurs :

```
import excel
values = excel.read(sheet_object, first_cell="A1", last_cell=None)
excel.write(sheet_object, values, first_cell="A1")
```

La fonction `read` accepte un objet `sheet` à partir de l'un des packages suivants : xlrd, OpenPyXL ou pyxlsb. Elle accepte également les arguments facultatifs `first_cell` et `last_cell` qui peuvent être fournis en notation A1 ou sous la forme de tuples représentant un numéro de ligne et un numéro de colonne, ces numéros commençant à un : (1, 1). La valeur par défaut du paramètre `first_cell` est A1 tandis que la valeur par défaut de `last_cell` est le coin inférieur droit de la plage utilisée. Par conséquent, si vous fournissez uniquement l'objet `sheet`, la feuille entière sera lue. La fonction `write` fonctionne de la même manière : elle attend un objet `sheet` à partir de xlwt, OpenPyXL ou XlsxWriter avec les valeurs sous forme de liste imbriquée et un paramètre `first_cell` facultatif, qui indique le coin supérieur gauche de l'emplacement où la liste imbriquée sera écrite. Le module `excel.py` harmonise également la conversion des types de données, comme cela est renseigné dans le Tableau 8.2.

Tableau 8.2 : Conversion de type de données.

Représentation Excel	Type de données Python
Cellule vide	`None`
Cellule avec un format de date	`datetime.datetime` (sauf pour pyxlsb)
Cellule avec une valeur booléenne	`bool`
Cellule avec une erreur	`str` (le message d'erreur)
Chaîne	`str`
Nombre à virgule flottante	`float` ou `int`

Armés du module `excel.py`, nous sommes maintenant prêts à étudier les packages : les quatre sections suivantes concernent OpenPyXL, XlsxWriter, pyxlsb et xlrd/xlwt/xlutils. Elles se

présentent sous la forme d'un livre de recettes, ce qui vous permet d'être rapidement opérationnel avec chaque package. Au lieu de lire intégralement toutes les sections, je vous recommande de choisir le package dont vous avez besoin en vous aidant du Tableau 8.1, puis de passer directement à la section correspondante.



Nous utiliserons l'instruction `with` à diverses occasions dans ce chapitre. Si vous avez besoin d'un rappel, reportez-vous à l'encadré « Gestionnaires de contexte et instruction with » du Chapitre 7.

OpenPyXL

OpenPyXL est le seul package de cette section qui peut lire et écrire des fichiers Excel. Vous pouvez même l'utiliser pour modifier des fichiers Excel, mais seulement s'ils sont simples.

Lecture avec OpenPyXL

L'exemple de code suivant montre comment effectuer des tâches courantes lorsque vous utilisez OpenPyXL pour lire des fichiers Excel. Pour lire les valeurs de cellules, vous devez ouvrir le classeur avec `data_only=True`. La valeur par défaut est `False`, ce qui a pour effet de renvoyer les formules des cellules et non pas les données :

```
In [1]: import pandas as pd
        import openpyxl
        import excel
        import datetime as dt
In [2]: # Ouvre le classeur pour lire les valeurs des cellules
        # Le fichier est automatiquement fermé après le chargement des données
        book = openpyxl.load_workbook("xl/stores.xlsx", data_only=True)
In [3]: # Lit l'objet worksheet par son nom ou son indice (à partir de 0)
        sheet = book["2019"]
        sheet = book.worksheets[0]
In [4]: # Récupère la liste de tous les noms des feuilles
        book.sheetnames
Out[4]: ['2019', '2020', '2019-2020']
In [5]: # Boucle sur les objets sheet
        # À la place de "name", openpyxl emploie "title".
        for i in book.worksheets:
            print(i.title)
2019
2020
2019-2020
In [6]: # Lit les dimensions,
        # c'est-à-dire la plage utilisée de la feuille
        sheet.max_row, sheet.max_column
Out[6]: (8, 6)
In [7]: # Lit la valeur d'une seule cellule
        # on utilise la notation "A1" puis les n° des cellules (à partir de 1)
        sheet["B6"].value
```

```
        sheet.cell(row=6, column=2).value
Out[7]: 'Boston'
In [8]: # Lit une plage de valeurs de cellules avec notre module excel
        data = excel.read(book["2019"], (2, 2), (8, 6))
        data[:2]  # Affiche les deux premières lignes
Out[8]: [['Store', 'Employees', 'Manager', 'Since', 'Flagship'],
         ['New York', 10, 'Sarah', datetime.datetime(2018, 7, 20, 0, 0), False]]
```

Écriture avec OpenPyXL

OpenPyXL génère le fichier Excel en mémoire et enregistre le fichier une fois que vous appelez la méthode save. Le code suivant produit le fichier illustré à la Figure 8.1 :

```
In [9]: import openpyxl
        from openpyxl.drawing.image import Image
        from openpyxl.chart import BarChart, Reference
        from openpyxl.styles import Font, colors
        from openpyxl.styles.borders import Border, Side
        from openpyxl.styles.alignment import Alignment
        from openpyxl.styles.fills import PatternFill
        import excel
In [10]: # Instancie un classeur
         book = openpyxl.Workbook()

         # Lit la première feuille et lui attribue un nom
         sheet = book.active
         sheet.title = "Sheet1"

         # Écrit des cellules en notation A1
         # puis avec les numéros des cellules (à partir de 1)
         sheet["A1"].value = "Hello 1"
         sheet.cell(row=2, column=1, value="Hello 2")

         # Mise en forme : couleur de fond, alignement, bordure et police
         font_format = Font(color="FF0000", bold=True)
         thin = Side(border_style="thin", color="FF0000")
         sheet["A3"].value = "Hello 3"
         sheet["A3"].font = font_format
         sheet["A3"].border = Border(top=thin, left=thin,
                                     right=thin, bottom=thin)
         sheet["A3"].alignment = Alignment(horizontal="center")
         sheet["A3"].fill = PatternFill(fgColor="FFFF00", fill_type="solid")

         # Mise en forme des nombres (emploi des chaînes de format d'Excel)
         sheet["A4"].value = 3.3333
         sheet["A4"].number_format = "0.00"

         # Mise en forme des dates (emploi des chaînes de format d'Excel)
         sheet["A5"].value = dt.date(2016, 10, 13)
         sheet["A5"].number_format = "mm/dd/yy"
```

```
# Formule : vous devez utiliser le nom anglais de la fonction
# avec des virgules pour les délimiteurs
sheet["A6"].value = "=SUM(A4, 2)"

# Image
sheet.add_image(Image("images/python.png"), "C1")

# Liste à deux dimensions (on emploie notre module excel)
data = [[None, "North", "South"],
["Last Year", 2, 5],
["This Year", 3, 6]]
excel.write(sheet, data, "A10")

# Graphique
chart = BarChart()
chart.type = "col"
chart.title = "Sales Per Region"
chart.x_axis.title = "Regions"
chart.y_axis.title = "Sales"
chart_data = Reference(sheet, min_row=11, min_col=1,
                       max_row=12, max_col=3)
chart_categories = Reference(sheet, min_row=10, min_col=2,
                             max_row=10, max_col=3)
# from_rows interprète les données de la même manière
# que si vous ajoutez un graphique manuellement dans Excel
chart.add_data(chart_data, titles_from_data=True, from_rows=True)
chart.set_categories(chart_categories)
sheet.add_chart(chart, "A15")

# Enregistrement du classeur et création du fichier sur le disque
book.save("openpyxl.xlsx")
```

Si vous souhaitez écrire un fichier Excel qui soit un modèle (*.xltx*), vous devez définir l'attribut `template` à `True` avant de l'enregistrer :

```
In [11]: book = openpyxl.Workbook()
         sheet = book.active
         sheet["A1"].value = "This is a template"
         book.template = True
         book.save("template.xltx")
```

Comme vous pouvez le voir dans le code, OpenPyXL définit les couleurs en fournissant une chaîne sous la forme FF0000. Cette chaîne est composée de trois valeurs hexadécimales (`FF`, `00` et `00`) qui correspondent aux valeurs rouge/vert/bleu de la couleur souhaitée. Hex est l'abréviation d'*hexadécimal* où les nombres sont représentés en utilisant la base de seize au lieu de la base de dix qui est employée par notre système décimal standard.



Pour trouver la valeur souhaitée en hexadécimal d'une couleur dans Excel, cliquez sur la liste déroulante de l'outil permettant de modifier la couleur de remplissage des cellules (symbolisé par un pot de peinture), puis sélectionnez Autres couleurs. Sélectionnez ensuite votre couleur et lisez sa valeur hexadécimale dans l'onglet Personnalisées.

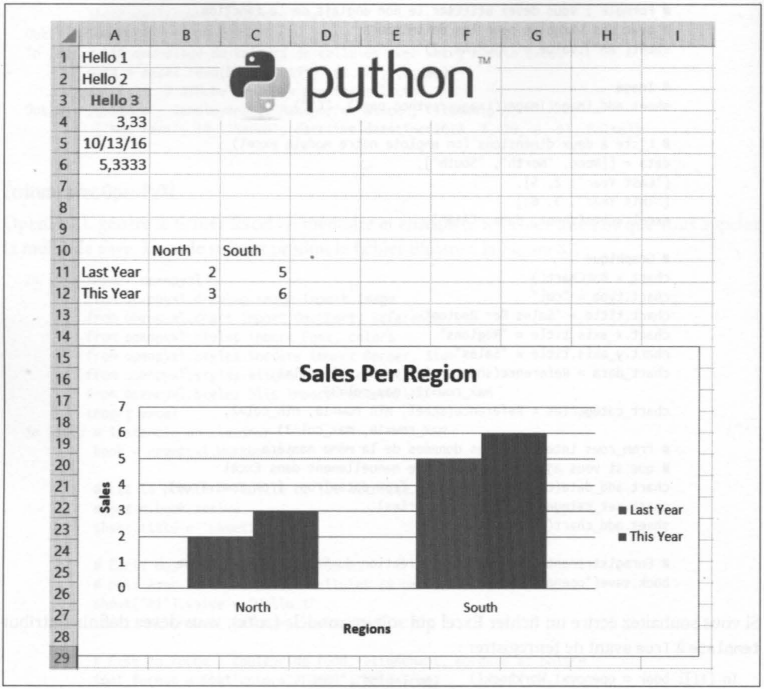


Figure 8.1 : Fichier écrit par OpenPyXL (openpyxl.xlsx).

Édition avec OpenPyXL

Il n'y a pas de package lecture/écriture qui puisse vraiment modifier les fichiers Excel : en réalité, OpenPyXL lit le fichier et l'interprète, puis réécrit le fichier à partir de zéro, en incluant toutes les modifications que vous apportez entre les deux opérations. Cela constitue un mécanisme très puissant pour les fichiers Excel simples qui contiennent principalement des cellules formatées avec des données et des formules, mais le principe est limité lorsque vous avez des graphiques et d'autres contenus plus avancés dans votre feuille de calcul, car OpenPyXL les modifie ou les supprime totalement. Par exemple, à compter de la version 3.0.5, OpenPyXL renomme les graphiques et supprime leur titre. Voici un exemple d'édition simple :

```
In [12]: # Lit le fichier stores.xlsx, et modifie une cellule, puis
         # l'enregistre sous un nouveau nom et un nouvel emplacement
         book = openpyxl.load_workbook("xl/stores.xlsx")
         book["2019"]["A1"].value = "modified"
         book.save("stores_edited.xlsx")
```

Si vous souhaitez écrire un fichier *xlsm* (contenant des macros), OpenPyXL doit travailler sur un fichier existant que vous devez charger avec le paramètre `keep_vba` défini à `True` :

```
In [13]: book = openpyxl.load_workbook("xl/macro.xlsm", keep_vba=True)
         book["Sheet1"]["A1"].value = "Click the button!"
         book.save("macro_openpyxl.xlsm")
```

Le bouton dans le fichier d'exemple appelle une macro qui affiche une boîte de message. OpenPyXL possède beaucoup d'autres fonctionnalités que je ne suis pas en mesure de couvrir dans cette section, et c'est donc une bonne idée de consulter la documentation officielle. Nous verrons également plus de fonctionnalités à la fin de ce chapitre lorsque nous reprendrons l'étude de cas du chapitre précédent.

XlsxWriter

Comme son nom l'indique, XlsxWriter ne peut écrire que des fichiers Excel. Le code suivant produit le même classeur que celui que nous avons généré précédemment avec OpenPyXL, et qui est illustré à la Figure 8.1. Vous noterez que XlsxWriter utilise des indices de cellule à partir de zéro, tandis qu'OpenPyXL emploie des indices de cellule à partir de un ; veillez à en tenir compte si vous passez d'un package à l'autre :

```
In [14]: import datetime as dt
         import xlsxwriter
         import excel
In [15]: # Instancie un classeur
         book = xlsxwriter.Workbook("xlxswriter.xlsx")

         # Ajoute une feuille et lui attribue un nom
         sheet = book.add_worksheet("Sheet1")

         # Écrit des cellules en employant la notation A1
         # puis des indices de cellules (à partir de 0)
         sheet.write("A1", "Hello 1")
         sheet.write(1, 0, "Hello 2")

         # Mise en forme : couleur de remplissage, alignement, bordure et police
         formatting = book.add_format({"font_color": "#FF0000",
                                       "bg_color": "#FFFF00",
                                       "bold": True, "align": "center",
                                       "border": 1, "border_color": "#FF0000"})
         sheet.write("A3", "Hello 3", formatting)
```

```
# Mise en forme des nombres (emploi des chaînes de format d'Excel)
number_format = book.add_format({"num_format": "0.00"})
sheet.write("A4", 3.3333, number_format)

# Mise en forme des dates (emploi des chaînes de format d'Excel)
date_format = book.add_format({"num_format": "mm/dd/yy"})
sheet.write("A5", dt.date(2016, 10, 13), date_format)

# Formule : vous devez utiliser le nom anglais de la fonction
# avec des virgules pour les délimiteurs
sheet.write("A6", "=SUM(A4, 2)")

# Image
sheet.insert_image(0, 2, "images/python.png")

# Liste à deux dimensions (on utilise notre module excel)
data = [[None, "North", "South"],
        ["Last Year", 2, 5],
        ["This Year", 3, 6]]
excel.write(sheet, data, "A10")

# Graphique : cf. le fichier "sales_report_xlsxwriter.py" dans les
# exemples de code pour voir comment exploiter les indices
# des cellules à la place de leurs adresses
chart = book.add_chart({"type": "column"})
chart.set_title({"name": "Sales per Region"})
chart.add_series({"name": "=Sheet1!A11",
                  "categories": "=Sheet1!B10:C10",
                  "values": "=Sheet1!B11:C11"})
chart.add_series({"name": "=Sheet1!A12",
                  "categories": "=Sheet1!B10:C10",
                  "values": "=Sheet1!B12:C12"})
chart.set_x_axis({"name": "Regions"})
chart.set_y_axis({"name": "Sales"})
sheet.insert_chart("A15", chart)

# Fermeture du classeur et création du fichier sur le disque
book.close()
```

Par rapport à OpenPyXL, XlsxWriter doit adopter une approche plus compliquée pour écrire des fichiers *xlsm*, car il s'agit purement d'un package d'écriture. Tout d'abord, vous devez extraire le code de la macro du fichier Excel à partir d'Anaconda Prompt (l'exemple utilise le fichier *macro.xlsm*, que vous trouverez dans le dossier *xl* des exemples de code) :

Windows

Positionnez-vous dans le répertoire *xl*, puis recherchez le chemin d'accès à *vba_extract.py*, un script qui est fourni avec XlsxWriter :

```
(base)> cd C:\Users\nom_utilisateur\python-for-excel\xl
(base)> where vba_extract.py
C:\Utilisateurs\nom_utilisateur\Anaconda3\Scripts\vba_extract.py
```

Utilisez ensuite ce chemin d'accès dans la commande suivante :

```
(base)> python C:\...\Anaconda3\Scripts\vba_extract.py macro.xlsm
```

macOS

Sur macOS, la commande est disponible en tant que script exécutable et peut être lancée de la manière suivante :

```
(base)> cd /Users/nom_utilisateur/python-for-excel/xl
(base)> vba_extract.py macro.xlsm
```

Cela enregistre le fichier *vbaProject.bin* dans le répertoire où vous exécutez la commande. J'ai également inclus le fichier extrait dans le dossier *xl* des exemples de code. Nous allons l'utiliser dans l'exemple suivant pour écrire un classeur avec un bouton de macro :

```
In [16]: book = xlsxwriter.Workbook("macro_xlxswriter.xlsm")
         sheet = book.add_worksheet("Sheet1")
         sheet.write("A1", "Click the button!")
         book.add_vba_project("xl/vbaProject.bin")
         sheet.insert_button("A3", {"macro": "Hello", "caption": "Button 1",
                                    "width": 130, "height": 35})
         book.close()
```

pyxlsb

Par rapport aux autres bibliothèques de lectures, pyxlsb offre moins de fonctionnalités, mais c'est la seule option lorsqu'il s'agit de lire des fichiers Excel au format binaire *xlsb*. pyxlsb ne fait pas partie d'Anaconda, et vous devez donc l'installer si vous ne l'avez pas déjà fait. Il n'est actuellement pas disponible *via* Conda non plus, si bien que vous devez utiliser pip pour l'installer :

```
(base)> pip install pyxlsb
```

On lit les feuilles et les valeurs des cellules de la manière suivante :

```
In [17]: import pyxlsb
         import excel
In [18]: # Boucle sur les feuilles. Avec pyxlsb, les objets workbook et sheet
         # peuvent être utilisés en tant que gestionnaires de contexte.
         # book.sheets renvoie une liste des noms de feuilles,
         # mais pas des objets.
         # Pour récupérer un objet sheet, utilisez get_sheet() à la place.
         with pyxlsb.open_workbook("xl/stores.xlsb") as book:
             for sheet_name in book.sheets:
                 with book.get_sheet(sheet_name) as sheet:
                     dim = sheet.dimension
                     print(f"Sheet '{sheet_name}' has "
                           f"{dim.h} rows and {dim.w} cols")
Sheet '2019' has 7 rows and 5 cols
Sheet '2020' has 7 rows and 5 cols
Sheet '2019-2020' has 20 rows and 5 cols
```

```
In [19]: # Lit les valeurs d'une plage de cellules avec notre module excel
         # Au lieu de "2019", vous pouvez aussi utiliser son indice
         # (à partir de 1)
         with pyxlsb.open_workbook("xl/stores.xlsb") as book:
             with book.get_sheet("2019") as sheet:
                 data = excel.read(sheet, "B2")
         data[:2]  # Affiche les deux premières lignes
Out[19]: [['Store', 'Employees', 'Manager', 'Since', 'Flagship'],
          ['New York', 10.0, 'Sarah', 43301.0, False]]
```

pyxlsb n'offre actuellement aucun moyen de reconnaître les cellules comportant des dates, et vous devez donc convertir manuellement les valeurs des cellules stockées sous forme de date en objets `datetime` de la manière suivante :

```
In [20]: from pyxlsb import convert_date
         convert_date(data[1][3])
Out[20]: datetime.datetime(2018, 7, 20, 0, 0)
```

N'oubliez pas que lorsque vous lisez le format de fichier *xlsb* avec une version de pandas inférieure à 1.3, vous devez spécifier explicitement le paramètre `engine` :

```
In [21]: df = pd.read_excel("xl/stores.xlsb", engine="pyxlsb")
```

xlrd, xlwt et xlutils

La combinaison de xlrd, xlwt et xlutils offre à peu près les mêmes fonctionnalités pour l'ancien format *xls* qu'OpenPyXL pour le format *xlsx* : xlrd lit, xlwt écrit et xlutils édite les fichiers *xls*. Ces packages ne sont plus activement développés, mais ils trouveront probablement leur utilité tant qu'il y aura encore des fichiers *xls*. xlutils ne fait pas partie d'Anaconda, et vous devez donc l'installer si vous ne l'avez pas déjà fait :

```
(base)> conda install xlutils
```

Lecture avec xlrd

L'exemple de code suivant montre comment lire les valeurs à partir d'un classeur Excel avec xlrd :

```
In [22]: import xlrd
         import xlwt
         from xlwt.Utils import cell_to_rowcol2
         import xlutils
         import excel
In [23]: # Ouvre le classeur pour lire les valeurs des cellules.
         # Le fichier est fermé automatiquement après le chargement des données.
         book = xlrd.open_workbook("xl/stores.xls")
In [24]: # Récupère la liste de tous les noms des feuilles
         book.sheet_names()
Out[24]: ['2019', '2020', '2019-2020']
In [25]: # Boucle sur les objets sheet
```

```
        for sheet in book.sheets():
            print(sheet.name)
2019
2020
2019-2020
In [26]: # Récupère un objet sheet par son nom ou son indice (à partir de 0)
         sheet = book.sheet_by_index(0)
         sheet = book.sheet_by_name("2019")
In [27]: # Dimensions
         sheet.nrows, sheet.ncols
Out[27]: (8, 6)
In [28]: # Lit la valeur d'une seule cellule
         # en notation "A1" puis avec les indices des cellules (à partir de 0).
         # Le symbole "*" sépare le tuple renvoyé par cell_to_rowcol2
         # en deux arguments individuels.
         sheet.cell(*cell_to_rowcol2("B3")).value
         sheet.cell(2, 1).value
Out[28]: 'New York'
In [29]: # Lit une plage de valeurs de cellules avec notre module excel
         data = excel.read(sheet, "B2")
         data[:2]  # Affiche les deux premières lignes
Out[29]: [['Store', 'Employees', 'Manager', 'Since', 'Flagship'],
          ['New York', 10.0, 'Sarah', datetime.datetime(2018, 7, 20, 0, 0), False]]
```



Contrairement à OpenPyXL et pyxlsb, xlrd renvoie les dimensions des cellules avec une valeur, au lieu de la plage utilisée d'une feuille lors de l'emploi de `sheet.nrows` et `sheet.ncols`. La plage utilisée qu'Excel renvoie contient souvent des lignes et des colonnes vides en bas de la plage et sur sa bordure droite. Cela peut, par exemple, se produire lorsque vous supprimez le contenu de lignes (en appuyant sur la touche Suppr), plutôt que de supprimer les lignes elles-mêmes (en faisant un clic droit et en sélectionnant Supprimer).

Écriture avec xlwt

Le code suivant reproduit ce que nous avons fait précédemment avec OpenPyXL et XlsxWriter et qui est illustré à la Figure 8.1. xlwt, toutefois, ne peut pas produire de graphiques et prend uniquement en charge le format `bmp` pour les images :

```
In [30]: import xlwt
         from xlwt.Utils import cell_to_rowcol2
         import datetime as dt
         import excel
In [31]: # Instancie un classeur
         book = xlwt.Workbook()

         # Ajoute une feuille et lui attribue un nom
         sheet = book.add_sheet("Sheet1")

         # Écrit des cellules en notation A1
         # puis avec des indices de cellules (à partir de 0)
```

```
sheet.write(*cell_to_rowcol2("A1"), "Hello 1")
sheet.write(r=1, c=0, label="Hello 2")

# Mise en forme : couleur de remplissage, alignement, bordure et police
formatting = xlwt.easyxf("font: bold on, color red;"
                         "align: horiz center;"
                         "borders: top_color red, bottom_color red,"
                                 "right_color red, left_color red,"
                                 "left thin, right thin,"
                                 "top thin, bottom thin;"
                         "pattern: pattern solid, fore_color yellow;")
sheet.write(r=2, c=0, label="Hello 3", style=formatting)

# Mise en forme des nombres (emploi des chaînes de format d'Excel)
number_format = xlwt.easyxf(num_format_str="0.00")
sheet.write(3, 0, 3.3333, number_format)

# Mise en forme des dates (emploi des chaînes de format d'Excel)
date_format = xlwt.easyxf(num_format_str="mm/dd/yyyy")
sheet.write(4, 0, dt.datetime(2012, 2, 3), date_format)

# Formule : vous devez utiliser le nom anglais de la fonction
# avec des virgules pour les délimiteurs
sheet.write(5, 0, xlwt.Formula("SUM(A4, 2)"))

# Liste à deux dimensions (on utilise notre module excel)
data = [[None, "North", "South"],
        ["Last Year", 2, 5],
        ["This Year", 3, 6]]
excel.write(sheet, data, "A10")

# Image (seul le format bmp est autorisé)
sheet.insert_bitmap("images/python.bmp", 0, 2)

# Cela écrit le fichier sur le disque
book.save("xlwt.xls")
```

Édition avec xlutils

xlutils agit comme un pont entre xlrd et xlwt. Vous devez comprendre qu'il ne s'agit pas d'une véritable opération d'édition : la feuille de calcul est lue, y compris la mise en forme *via* xlrd (en définissant `formatting_info = True`), puis écrite à nouveau par xlwt, y compris les modifications qui ont été apportées entre les deux opérations :

```
In [32]: import xlutils.copy
In [33]: book = xlrd.open_workbook("xl/stores.xls", formatting_info=True)
         book = xlutils.copy.copy(book)
         book.get_sheet(0).write(0, 0, "changed!")
         book.save("stores_edited.xls")
```

À ce stade, vous savez lire et écrire un classeur Excel dans un format spécifique. La section suivante traite de quelques fonctionnalités avancées, notamment l'exploitation de fichiers Excel volumineux et l'utilisation conjointe de pandas et des packages de lecture et d'écriture.

Fonctionnalités avancées pour la lecture et l'écriture de fichiers Excel

Si vos fichiers sont plus volumineux et plus complexes que les simples fichiers Excel que nous avons utilisés jusqu'à présent dans les exemples, les options par défaut ne seront peut-être plus suffisantes. Par conséquent, nous allons voir comment travailler avec des fichiers plus importants. Ensuite, nous apprendrons à utiliser pandas avec les packages de lecture et d'écriture, ce qui vous offrira la possibilité d'appliquer les styles que vous souhaitez à vos DataFrames. Pour conclure cette section, nous utiliserons tout ce que nous avons appris dans ce chapitre pour apporter une touche plus professionnelle au rapport Excel de l'étude de cas du chapitre précédent.

Exploitation de fichiers Excel volumineux

L'utilisation de fichiers volumineux peut entraîner deux problèmes : le processus de lecture et d'écriture peut être lent ou votre ordinateur peut manquer de mémoire. En général, le problème de mémoire est le plus préoccupant, car il provoque le plantage de votre programme. Le moment à partir duquel un fichier Excel est considéré comme *volumineux* dépend toujours des ressources disponibles sur votre système et de votre définition de la *lenteur*… Cette section présente les techniques d'optimisation offertes par les différents packages, qui vous permettront de travailler avec des fichiers Excel dépassant les limites. Je commencerai par examiner les options pour les bibliothèques d'écriture, et traiterai ensuite les options pour les bibliothèques de lecture. À la fin de cette section, je vous montrerai comment lire les feuilles d'un classeur en parallèle afin de réduire le temps de traitement.

Écriture avec OpenPyXL

Lorsque vous écrivez des fichiers volumineux avec OpenPyXL, assurez-vous que le package lxml est installé, car cela accélère le processus d'écriture. Ce package étant inclus dans Anaconda, vous n'avez donc en principe rien à faire. Il y a cependant une option critique, le paramètre `write_only=True`, qui permet de s'assurer que la consommation de mémoire reste faible. Toutefois, cela vous oblige à écrire ligne par ligne à l'aide de la méthode `append` et ne vous permet plus d'écrire des cellules individuelles :

```
In [34]: book = openpyxl.Workbook(write_only=True)
         # Avec write_only=True, book.active ne fonctionne plus
         sheet = book.create_sheet()
         # Cela produit une feuille de 1000 x 200 cellules
         for row in range(1000):
             sheet.append(list(range(200)))
         book.save("openpyxl_optimized.xlsx")
```

Écriture avec XlsxWriter

XlsxWriter a une option similaire à celle d'OpenPyXL, appelée `constant_memory`. Cela vous oblige également à écrire des lignes séquentiellement. On active l'option en fournissant un dictionnaire nommé `options` de la manière suivante :

```
In [35]: book = xlsxwriter.Workbook("xlsxwriter_optimized.xlsx",
                                    options={"constant_memory": True})
         sheet = book.add_worksheet()
         # Cela produit une feuille de 1000 x 200 cellules
         for row in range(1000):
             sheet.write_row(row , 0, list(range(200)))
         book.close()
```

Lecture avec xlrd

Lors de la lecture de fichiers volumineux dans l'ancien format *xls*, xlrd permet de charger des feuilles à la demande, de la manière suivante :

```
In [36]: with xlrd.open_workbook("xl/stores.xls", on_demand=True) as book:
             sheet = book.sheet_by_index(0)  # Charge la première feuille
```

Si vous n'utilisez pas le classeur comme gestionnaire de contexte comme nous le faisons ici, vous devrez appeler `book.release_resources()` manuellement afin de fermer correctement le classeur. Pour utiliser xlrd dans ce mode avec pandas, utilisez-le comme ceci :

```
In [37]: with xlrd.open_workbook("xl/stores.xls", on_demand=True) as book:
             with pd.ExcelFile(book, engine="xlrd") as f:
                 df = pd.read_excel(f, sheet_name=0)
```

Lecture avec OpenPyXL

Pour préserver la mémoire lors de la lecture de gros fichiers Excel avec OpenPyXL, vous devez charger le classeur avec l'option `read_only=True`. Étant donné qu'OpenPyXL ne prend pas en charge l'instruction `with`, vous devez vous assurer de fermer le fichier lorsque vous avez terminé. Si votre fichier contient des liens vers des classeurs externes, vous pouvez également utiliser `keep_links=False` pour accélérer le processus. `keep_links` s'assure que les références aux classeurs externes sont conservées, ce qui peut ralentir inutilement le processus si vous êtes uniquement intéressé par la lecture des valeurs d'un classeur :

```
In [38]: book = openpyxl.load_workbook("xl/big.xlsx",
                                       data_only=True, read_only=True,
                                       keep_links=False)
         # On accomplit ici les opérations de lecture souhaitées
         book.close()  # Nécessaire avec read_only=True
```

Lecture de feuilles en parallèle

Lorsque vous utilisez la fonction `read_excel` de pandas pour lire plusieurs feuilles d'un grand classeur, vous constaterez que cela prend beaucoup de temps (nous prendrons un exemple concret dans un instant). La raison en est que pandas lit les feuilles séquentiellement, c'est-à-dire les unes après les autres. Pour accélérer les choses, vous pouvez lire les feuilles en parallèle. Bien qu'il n'existe aucun moyen facile de paralléliser l'écriture de classeurs en raison de la façon dont les fichiers sont structurés en interne, la lecture de plusieurs feuilles en parallèle est assez simple. Cependant, comme la parallélisation est un sujet avancé, je ne l'ai pas abordée dans l'introduction à Python et je n'entrerai pas plus ici dans les détails.

En Python, si vous souhaitez tirer parti des différents cœurs du processeur dont dispose chaque ordinateur moderne, vous allez utiliser le package de multitraitement qui fait partie de la bibliothèque standard. Cela engendrera plusieurs interpréteurs Python (généralement un par cœur de processeur), qui fonctionnent sur une tâche en parallèle. Au lieu de traiter une feuille après l'autre, vous avez un interpréteur Python qui traite la première feuille, tandis qu'en même temps un deuxième interpréteur Python traite la deuxième feuille, *etc.* Cependant, chaque interpréteur Python supplémentaire met un certain temps à démarrer et utilise de la mémoire supplémentaire, de telle sorte que si vous avez de petits fichiers, ils s'exécuteront probablement plus lentement lorsque vous parallélisez le processus de lecture, au lieu d'accélérer les traitements. Dans le cas d'un fichier volumineux comportant plusieurs grandes feuilles, le multitraitement peut accélérer considérablement le processus, mais cela suppose que votre système dispose de la mémoire requise pour gérer la charge de travail. Si vous exécutez le notebook Jupyter sur Binder (voir le Chapitre 2), vous n'aurez pas assez de mémoire et, par conséquent, la version parallélisée s'exécutera plus lentement. Dans les exemples de code, vous trouverez le fichier *parallel_pandas.py*, qui est une implémentation simple pour lire les feuilles en parallèle, OpenPyXL étant le moteur employé. Cela est simple à utiliser, et vous n'aurez donc pas besoin de savoir quoi que ce soit sur le multitraitement :

```
import parallel_pandas
parallel_pandas.read_excel(filename, sheet_name=None)
```

Par défaut, toutes les feuilles sont lues, mais vous pouvez fournir une liste des noms de feuilles que vous souhaitez traiter. Comme avec pandas, la fonction renvoie un dictionnaire sous la forme suivante : `{"nom_feuille": df}`, c'est-à-dire que les clés sont les noms des feuilles et les valeurs sont les DataFrames.

Commande magique %%time

Dans les exemples suivants, je vais utiliserla commande magique`%%time` J'ai évoqué les commandes magiques au Chapitre 5 en relation avec Matplotlib. `%%time` est une commande magique de cellule qui peut être très utile pour un réglage simple des performances, car elle permet de comparer facilement le temps d'exécution de deux cellules avec des extraits de code différents. Le *wall time* représente le temps écoulé entre le début et la fin du programme, c'est-à-dire la cellule. Si vous êtes sous macOS ou Linux, vous n'obtiendrez pas seulement l'indication du wall time, mais également une ligne supplémentaire pour le temps CPU sous la forme suivante :

```
CPU times: user 49.4 s, sys: 108 ms, total: 49.5 s
```

Le temps CPU mesure le temps passé sur le CPU, qui peut être inférieur au wall time (si le programme doit attendre que le CPU soit disponible) ou supérieur (si le programme s'exécute sur plusieurs cœurs de CPU en parallèle). Pour mesurer le temps plus précisément, utilisez `%%timeit` au lieu de `%%time`, qui exécute la cellule plusieurs fois et prend la moyenne de toutes les exécutions. `%%time` et `%%timeit` sont des commandes magiques au niveau des cellules, c'est-à-dire qu'elles doivent être sur la première ligne de la cellule et qu'elles mesurent le temps d'exécution de la totalité de la cellule. Si, au lieu de cela, vous souhaitez mesurer une seule ligne, commencez cette ligne par `%time` ou `%timeit`.

Voyons à quel point la version parallélisée lit plus rapidement le fichier *big.xlsx* que vous trouverez dans le dossier *xl* des exemples de code :

```
In [39]: %%time
         data = pd.read_excel("xl/big.xlsx",
                              sheet_name=None, engine="openpyxl")
Wall time: 49.5 s
In [40]: %%time
         import parallel_pandas
         data = parallel_pandas.read_excel("xl/big.xlsx", sheet_name=None)
Wall time: 12.1 s
```

Pour lire le DataFrame qui représente Sheet1, vous devez écrire `data["Sheet1"]` dans les deux cas. En regardant le wall time des deux exemples, vous verrez que la version parallélisée est beaucoup plus rapide que `pd.read_excel` avec ce classeur particulier exécuté sur mon ordinateur portable doté de 6 cœurs de processeur. Si vous souhaitez encore accélérer les choses, parallélisez directement OpenPyXL : vous trouverez également une implémentation de cela dans les exemples de code (*parallel_openpyxl.py*), ainsi qu'une implémentation pour xlrd afin de lire l'ancien format *xls* en parallèle (*parallel_xlrd.py*). En exploitant directement les packages sous-jacents sans passer par pandas, cela vous permettra d'ignorer la transformation en DataFrame

ou d'appliquer uniquement les étapes de nettoyage dont vous avez besoin, ce qui vous aidera probablement à accélérer les choses, si c'est votre préoccupation principale.

Lecture d'une feuille en parallèle avec Modin

Si vous ne lisez qu'une seule feuille énorme, cela vaut la peine de s'intéresser à Modin, un projet qui remplace pandas. Modin parallélise le processus de lecture d'une seule feuille et offre des améliorations impressionnantes en matière de vitesse. Comme il nécessite une version spécifique de pandas, son installation pourrait rétrograder la version de pandas fournie avec Anaconda. Si vous souhaitez le tester, je vous recommande de créer un environnement Conda distinct afin de vous assurer que vous ne perturbez pas votre environnement de base. Pour obtenir des instructions plus détaillées sur la création d'un environnement Conda, reportez-vous à l'Annexe A :

```
(base)> conda create --name modin python=3.8 -y
(base)> conda activate modin
(modin)> conda install -c conda-forge modin -y
```

Sur ma machine, quand j'utilise le fichier *big.xlsx*, l'exécution du code suivant a pris environ cinq secondes alors qu'il en a fallu environ douze avec pandas :

```
import modin.pandas
data = modin.pandas.read_excel("xl/big.xlsx", sheet_name=0, engine="openpyxl")
```

Maintenant que vous savez comment gérer les fichiers volumineux, voyons comment nous pouvons utiliser pandas et les packages de bas niveau conjointement afin d'améliorer le formatage par défaut lors de l'écriture de DataFrames dans des fichiers Excel.

Mise en forme des DataFrames dans Excel

Pour formater les DataFrames dans Excel comme nous le souhaitons, nous pouvons écrire du code qui utilise pandas avec OpenPyXL ou XlsxWriter. Nous allons d'abord utiliser cette combinaison pour ajouter un titre au DataFrame exporté. Nous allons ensuite mettre en forme l'en-tête et l'index d'un DataFrame avant de conclure cette section en mettant en forme la partie données d'un DataFrame. Combiner pandas avec OpenPyXL pour la lecture peut également se révéler utile :

```
In [41]: with pd.ExcelFile("xl/stores.xlsx", engine="openpyxl") as xlfile:
             # On lit un DataFrame
             df = pd.read_excel(xlfile, sheet_name="2020")

             # On lit l'objet workbook de OpenPyXL
             book = xlfile.book
```

```
# À partir d'ici, c'est du code OpenPyXL
sheet = book["2019"]
value = sheet["B3"].value  # Lit une valeur unique
```

Lors de l'écriture de classeurs, cela fonctionne de manière analogue, ce qui nous permet d'ajouter facilement un titre à notre rapport issu d'un DataFrame :

```
In [42]: with pd.ExcelWriter("pandas_and_openpyxl.xlsx",
                              engine="openpyxl") as writer:
             df = pd.DataFrame({"col1": [1, 2, 3, 4], "col2": [5, 6, 7, 8]})
             # Écrit un DataFrame
             df.to_excel(writer, "Sheet1", startrow=4, startcol=2)

             # Lit les objets workbook et sheet de OpenPyXL
             book = writer.book
             sheet = writer.sheets["Sheet1"]

             # À partir d'ici, c'est du code OpenPyXL
             sheet["A1"].value = "This is a Title"  # Écrit une valeur unique
```

Ces exemples utilisent OpenPyXL, mais cela fonctionne conceptuellement de la même manière avec les autres packages. Continuons maintenant à découvrir comment nous pouvons mettre en forme l'index et l'en-tête d'un DataFrame.

Mise en forme de l'index et des en-têtes d'un DataFrame

Le moyen le plus simple d'obtenir un contrôle complet sur la mise en forme de l'index et des en-têtes de colonnes consiste simplement à les écrire vous-même. L'exemple suivant montre comment procéder avec OpenPyXL et XlsxWriter, respectivement. Vous pouvez voir la sortie illustrée à la Figure 8.2. Commençons par créer un DataFrame :

```
In [43]: df = pd.DataFrame({"col1": [1, -2], "col2": [-3, 4]},
                           index=["row1", "row2"])
         df.index.name = "ix"
         df
Out[43]:
```

ix	col1	col2
row1	1	-3
row2	-2	4

Pour mettre en forme l'index et les en-têtes avec OpenPyXL, procédez de la manière suivante :

```
In [44]: from openpyxl.styles import PatternFill
In [45]: with pd.ExcelWriter("formatting_openpyxl.xlsx",
                              engine="openpyxl") as writer:
             # Écrit le df avec la mise en forme par défaut en A1
             df.to_excel(writer, startrow=0, startcol=0)
```

```
        # Écrit le df avec une mise en forme personnalisée
        # de l'index et l'en-tête en A6
        startrow, startcol = 0, 5
        # 1. Écrit la partie données du DataFrame
        df.to_excel(writer, header=False, index=False,
                    startrow=startrow + 1, startcol=startcol + 1)
        # Lit l'objet sheet et crée un objet style
        sheet = writer.sheets["Sheet1"]
        style = PatternFill(fgColor="D9D9D9", fill_type="solid")

        # 2. Écrit les en-têtes de colonnes stylés
        for i, col in enumerate(df.columns):
            sheet.cell(row=startrow + 1, column=i + startcol + 2,
                       value=col).fill = style

        # 3. Écrit l'index stylé
        index = [df.index.name if df.index.name else None] + list(df.index)
        for i, row in enumerate(index):
            sheet.cell(row=i + startrow + 1, column=startcol + 1,
                       value=row).fill = style
```

Si vous voulez à la place mettre en forme l'index et les en-têtes avec XlsxWriter, vous devez ajuster légèrement le code :

```
In [46]: # Formatage index/en-tête avec XlsxWriter
         with pd.ExcelWriter("formatting_xlsxwriter.xlsx",
                             engine="xlsxwriter") as writer:
             # Écrit le df avec la mise en forme par défaut en A1
             df.to_excel(writer, startrow=0, startcol=0)

             # Écrit le df avec une mise en forme personnalisée
             # de l'index et l'en-tête en A6
             startrow, startcol = 0, 5
             # 1. Écrit la partie données du DataFrame
             df.to_excel(writer, header=False, index=False,
                         startrow=startrow + 1, startcol=startcol + 1)
             # Lit les objets book et sheet et crée un objet style
             book = writer.book
             sheet = writer.sheets["Sheet1"]
             style = book.add_format({"bg_color": "#D9D9D9"})

             # 2. Écrit les en-têtes de colonnes stylés
             for i, col in enumerate(df.columns):
                 sheet.write(startrow, startcol + i + 1, col, style)

             # 3. Écrit l'index stylé
             index = [df.index.name if df.index.name else None] + list(df.index)
             for i, row in enumerate(index):
                 sheet.write(startrow + i, startcol, row, style)
```

Voyons à présent comment nous pouvons styler la partie données.

	A	B	C	D	E	F	G	H
1	ix	col1	col2			ix	col1	col2
2	row1	1	-3			row1	1	-3
3	row2	-2	4			row2	-2	4

Figure 8.2 : DataFrame avec le format par défaut (à gauche) et un format personnalisé (à droite).

Mise en forme de la partie données d'un DataFrame

Les possibilités dont vous disposez pour formater la partie données d'un DataFrame dépendent du package que vous utilisez : si vous utilisez la méthode `to_excel` de pandas, OpenPyXL peut appliquer un format à chaque cellule, tandis que XlsxWriter ne peut appliquer des formats que sur une ligne ou une colonne. Par exemple, pour définir le format numérique des cellules avec trois décimales et aligner le contenu au centre comme cela est illustré à la Figure 8.3, procédez de la manière suivante avec OpenPyXL :

```
In [47]: from openpyxl.styles import Alignment
In [48]: with pd.ExcelWriter("data_format_openpyxl.xlsx",
                             engine="openpyxl") as writer:
             # Écrit le DataFrame
             df.to_excel(writer)

             # Lit les objets book et sheet
             book = writer.book
             sheet = writer.sheets["Sheet1"]

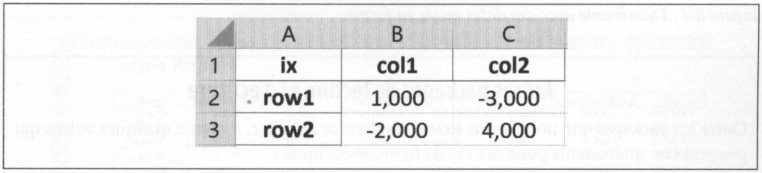
             # Mise en forme des cellules individuelles
             nrows, ncols = df.shape
             for row in range(nrows):
                 for col in range(ncols):
                     # +1 pour prendre en compte l'en-tête/index
                     # +1 comme OpenPyXL démarre à partir de 1
                     cell = sheet.cell(row=row + 2,
                                       column=col + 2)
                     cell.number_format = "0.000"
                     cell.alignment = Alignment(horizontal="center")
```

Pour XlsxWriter, modifiez le code de la manière suivante :

```
In [49]: with pd.ExcelWriter("data_format_xlsxwriter.xlsx",
                             engine="xlsxwriter") as writer:
             # Écrit le DataFrame
             df.to_excel(writer)

             # Lit les objets book et sheet
             book = writer.book
             sheet = writer.sheets["Sheet1"]
```

```
# Formatage des colonnes
# Mise en forme des cellules individuelles impossible
number_format = book.add_format({"num_format": "0.000",
                                 "align": "center"})
sheet.set_column(first_col=1, last_col=2,
                 cell_format=number_format)
```



	A	B	C
1	ix	col1	col2
2	row1	1,000	-3,000
3	row2	-2,000	4,000

Figure 8.3 : DataFrame avec des données mises en forme.

pandas offre une alternative avec la prise en charge *expérimentale* de la propriété `style` de DataFrames (expérimentale signifie que la syntaxe peut changer à tout moment). Étant donné que les styles ont été introduits pour formater les DataFrames au format HTML, ils utilisent la syntaxe CSS (*feuilles de style en cascade* utilisées pour définir le style des éléments HTML). Pour appliquer le même format que dans l'exemple précédent (trois décimales et alignement au centre), vous devez appliquer une fonction à chaque élément d'un objet `Styler` *via* `applymap`. On obtient un objet `Styler` *via* l'attribut `df.style` :

```
In [50]: df.style.applymap(lambda x: "number-format: 0.000;"
                                      "text-align: center")\
                 .to_excel("styled.xlsx")
```

Le résultat de ce code est le même que celui qui est illustré à la Figure 8.3. Pour plus d'informations sur cette nouvelle approche de stylage des DataFrames, reportez-vous à sa documentation[7].

Si vous ne voulez pas employer l'attribut `style`, pandas offre la prise en charge du formatage des objets `date et datetime` (voir la Figure 8.4) :

```
In [51]: df = pd.DataFrame({"Date": [dt.date(2020, 1, 1)],
                            "Datetime": [dt.datetime(2020, 1, 1, 10)]})
         with pd.ExcelWriter("date.xlsx",
                             date_format="yyyy-mm-dd",
                             datetime_format="yyyy-mm-dd hh:mm:ss") as writer:
             df.to_excel(writer)
```

7 https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html

	A	B	C
1		Date	Datetime
2	0	2020-01-01	2020-01-01 10:00:00

Figure 8.4 : DataFrame avec des dates mises en forme.

Autres packages de lecture et d'écriture

Outre les packages que nous avons examinés dans ce chapitre, il y en a quelques autres qui peuvent être intéressants pour des cas de figure spécifiques :

pyexcel
pyexcel[8] offre une syntaxe harmonisée entre différents packages Excel et d'autres formats de fichiers, notamment les fichiers CSV et les fichiers OpenOffice.

PyExcelerate
L'objectif de PyExcelerate[9] est d'écrire des fichiers Excel de la manière la plus rapide possible.

pylightxl
pylightxl[10] peut lire des fichiers *xlsx* et *xlsm* ainsi qu'écrire des fichiers *xlsx*.

styleframe
styleframe[11] encapsule pandas et OpenPyXL pour produire des fichiers Excel avec des DataFrames mis en forme de manière professionnelle.

oleotools
oletools[12] n'est pas un package de lecture ou d'écriture classique, mais il peut être utilisé pour analyser des documents Microsoft Office, par exemple, pour l'analyse des logiciels malveillants. Il offre un moyen pratique d'extraire le code VBA des classeurs Excel.

Maintenant que vous savez comment formater les DataFrames dans Excel, il est temps de revenir sur l'étude de cas du chapitre précédent et de voir si nous pouvons enjoliver le rapport en appliquant les connaissances étudiées dans ce chapitre.

8 http://www.pyexcel.org/
9 https://github.com/kz26/PyExcelerate
10 https://pylightxl.readthedocs.io/en/latest/
11 https://styleframe.readthedocs.io/en/latest/
12 https://github.com/decalage2/oletools

Étude de cas (revisitée) : création d'un rapport Excel

Nous souhaitons améliorer le rapport Excel du chapitre précédent afin de le rendre visuellement plus attrayant. Le but de l'exercice est de repartir du fichier *sales_report_pandas.py* (présent dans les exemples de code) et de tenter de le transformer en un rapport tel que celui qui est illustré à la Figure 8.5.

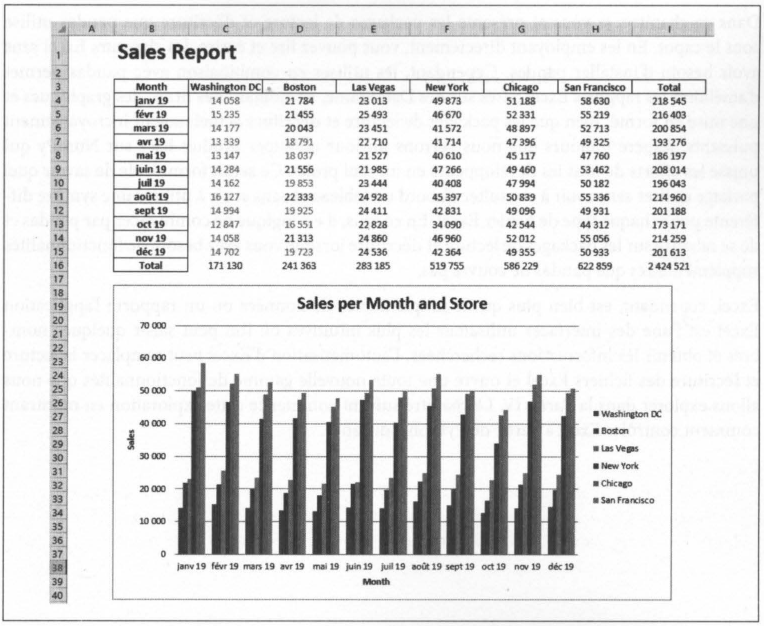


Figure 8.5 : Rapport des ventes revisité à partir de la version créée par sales_report_openpyxl.py.

Les chiffres en rouge sont des chiffres de ventes inférieurs à 20 000. Je n'ai pas abordé tous les aspects de la mise en forme dans ce chapitre (notamment la mise en forme conditionnelle), et vous devrez donc utiliser la documentation du package avec lequel vous choisissez de travailler. Pour évaluer votre solution, j'ai inclus deux versions du script qui produisent ce rapport dans les exemples de code. La première version est basée sur OpenPyXL (*sales_report_openpyxl.py*) et l'autre sur XlsxWriter (*sales_report_xlsxwriter.py*). Si vous visualisez les scripts côte à côte, vous pourrez comparer les packages et prendre plus facilement une décision sur celui que vous

souhaitez choisir pour votre prochaine tâche d'écriture de fichier Excel. Nous reviendrons à cette étude de cas une fois de plus dans le chapitre suivant, où nous nous appuierons sur Excel pour travailler avec des modèles de rapport.

Conclusion

Dans ce chapitre, je vous ai présenté les packages de lecture et d'écriture que pandas utilise sous le capot. En les employant directement, vous pouvez lire et écrire des classeurs Excel sans avoir besoin d'installer pandas. Cependant, les utiliser en combinaison avec pandas permet d'améliorer les rapports Excel basés sûr des DataFrame, en ajoutant des titres, des graphiques et une mise en forme. Bien que les packages de lecture et d'écriture actuels soient incroyablement puissants, j'espère toujours que nous verrons un jour un super produit basé sur NumPy qui unisse les efforts de tous les développeurs en un seul projet. Ce serait formidable de savoir quel package utiliser sans avoir à consulter d'abord un tableau et sans avoir à utiliser une syntaxe différente pour chaque type de fichier Excel. En ce sens, il est logique de commencer par pandas et de se rabattre sur les packages de lecture et d'écriture lorsque vous avez besoin de fonctionnalités supplémentaires que pandas ne couvre pas.

Excel, cependant, est bien plus qu'un simple fichier de données ou un rapport : l'application Excel est l'une des interfaces utilisateur les plus intuitives où l'on peut saisir quelques nombres et obtenir les informations recherchées. L'automatisation d'Excel peut remplacer la lecture et l'écriture des fichiers Excel et ouvre une toute nouvelle gamme de fonctionnalités que nous allons explorer dans la Partie IV. Le chapitre suivant commence cette exploration en montrant comment contrôler Excel à partir de Python à distance.

PARTIE IV

Programmer Excel avec xlwings

CHAPITRE 9
Automatisation d'Excel

Jusqu'à présent, nous avons appris à remplacer les tâches courantes d'Excel par pandas (Partie II) et à utiliser des fichiers Excel à la fois comme source de données et comme format de fichier pour vos rapports (Partie III). Ce chapitre inaugure la Partie IV, où nous abandonnons la manipulation des *fichiers* Excel avec les packages de lecture et d'écriture pour commencer à automatiser l'*application* Excel avec xlwings.

xlwings s'emploie principalement pour créer des applications interactives où les feuilles de calcul Excel servent d'interface utilisateur, ce qui permet d'appeler du code Python en cliquant sur un bouton ou d'appeler une fonction définie par l'utilisateur, fonctionnalité qui n'est pas couverte par les packages de lecture et d'écriture. Cela ne signifie pas pour autant que xlwings ne peut pas être utilisé pour lire et écrire des fichiers, à condition vous soyez sous macOS ou Windows et que l'application Excel soit installée. xlwings a un avantage dans ce domaine : il permet d'éditer vraiment les fichiers Excel, dans tous les formats, sans modifier, voire perdre le contenu ou le formatage existant. Il y a un autre avantage : vous pouvez lire les valeurs des cellules à partir d'un classeur Excel sans avoir besoin de l'enregistrer au préalable. Cependant, il peut également être parfaitement judicieux d'utiliser un package de lecture et d'écriture conjointement avec xlwings, comme nous le verrons lorsque nous reprendrons une fois de plus l'étude de cas de génération d'un rapport du Chapitre 7.

Je commencerai ce chapitre en vous présentant le modèle objet d'Excel ainsi que xlwings : nous allons d'abord apprendre les bases, comme la connexion à un classeur ou la lecture et l'écriture de valeurs de cellules avant de creuser un peu plus profondément pour comprendre comment les convertisseurs et les options permettent de travailler avec les DataFrames pandas et les tableaux NumPy. Nous examinerons également comment interagir avec les graphiques, les images et les noms définis avant de passer à la dernière section, qui explique comment xlwings fonctionne sous le capot : cela vous apportera les connaissances nécessaires pour rendre vos scripts performants et pallier les fonctionnalités manquantes. À partir de ce chapitre, vous devrez exécuter

les exemples de code sous Windows ou macOS, car ils dépendent de l'installation sur votre ordinateur de Microsoft Excel[1].

Prise en main de xlwings

L'un des objectifs de xlwings est de remplacer VBA, ce qui permet d'interagir avec Excel à partir de Python (sous Windows et macOS). Comme l'agencement en lignes et en colonnes d'Excel constitue une disposition parfaite pour afficher les structures de données de Python telles que les listes imbriquées, les tableaux NumPy et les DataFrames pandas, l'une des principales fonctionnalités de xlwings est de faciliter au maximum leur lecture et leur écriture. Je commencerai cette section en présentant Excel en tant qu'outil de visualisation de données, ce qui est utile lorsque vous interagissez avec des DataFrames dans un notebook Jupyter. J'expliquerai ensuite le modèle objet d'Excel avant de l'explorer de manière interactive avec xlwings. Pour conclure cette section, je vous montrerai comment appeler le code VBA que vous pouvez encore avoir dans vos anciens classeurs. Puisque xlwings fait partie d'Anaconda, nous n'avons pas besoin de l'installer manuellement.

Utilisation d'Excel pour visualiser des données

Vous avez probablement remarqué dans les chapitres précédents que, par défaut, les notebooks Jupyter masquent la majorité des données des DataFrames volumineux et n'affichent que les premières et dernières lignes ainsi que les premières et dernières colonnes. Pour mieux appréhender vos données, vous pouvez aussi les représenter sous une forme graphique, ce qui permet de repérer les valeurs aberrantes ou d'autres irrégularités. Cependant, il est parfois très utile de pouvoir faire défiler une table comportant des données. Après avoir lu le Chapitre 7, vous savez comment utiliser la méthode `to_excel` sur vos DataFrames. Bien que cette méthode fonctionne, elle peut être un peu fastidieuse, car vous devez donner un nom au fichier Excel, le trouver dans le système de fichiers, l'ouvrir et, après avoir apporté des modifications à votre DataFrame, vous devez fermer le fichier Excel et exécuter à nouveau l'ensemble du processus. Une meilleure approche consiste à exécuter `df.to_clipboard()`, qui copie le DataFrame `df` dans le presse-papiers ; cela vous permet de le coller dans Excel, mais il existe un moyen encore plus simple : l'emploi de la fonction `view` fournie avec xlwings :

```
In [1]: # On importe d'abord les packages utilisés dans ce chapitre
        import datetime as dt
        import xlwings as xw
        import pandas as pd
        import numpy as np
```

1 Sous Windows, la configuration minimale requise est Excel 2007, et sous macOS, Excel 2016. Vous pouvez également installer la version de bureau d'Excel, qui fait partie de votre abonnement à Microsoft 365. Consultez votre abonnement pour savoir comment procéder.

```
In [2]: # On crée un DataFrame basé sur des nombres aléatoires avec
        # suffisamment de lignes dont on ne montre que le début et la fin
        df = pd.DataFrame(data=np.random.randn(100, 5),
                          columns=[f"Trial {i}" for i in range(1, 6)])
        df
Out[2]:
```

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
0	-0.604918	-1.164014	1.005481	0.142161	-0.115246
1	-1.002379	-0.354194	-0.220871	-1.146373	-1.211933
2	-1.579143	0.796907	-0.162507	-1.178965	-0.295403
3	0.421516	-0.003893	1.670008	0.501002	-0.374748
4	0.777520	0.645801	-0.425558	-2.134099	1.990117
...	...	...	...	...	...
95	-2.226782	-0.474091	2.032531	2.018016	1.419685
96	0.150706	-1.163307	-0.310742	0.637623	-1.006158
97	1.673484	-0.892898	0.123930	-0.637346	-2.150886
98	-1.585635	-0.548795	-0.411500	1.742112	0.833226
99	-0.729816	-1.176273	-2.208679	-0.713579	-1.817730

```
100 rows × 5 columns
In [3]: # Affiche le DataFrame dans Excel
        xw.view(df)
```

La fonction `view` accepte tous les objets Python courants, y compris les nombres, les chaînes, les listes, les dictionnaires, les tuples, les tableaux NumPy et les DataFrames pandas. Par défaut, elle ouvre un nouveau classeur et colle l'objet dans la cellule A1 de la première feuille ; elle ajuste même la largeur des colonnes à l'aide de la fonctionnalité d'ajustement automatique d'Excel. Au lieu d'ouvrir un nouveau classeur à chaque fois, vous pouvez également réutiliser le même classeur en fournissant à la fonction `view` un objet xlwings `sheet` comme deuxième argument : `xw.view(df, ma_feuille)`. Je vous expliquerai bientôt comment accéder à l'objet `sheet` et comment il s'intègre dans le modèle objet d'Excel[2].



Sous macOS, assurez-vous d'exécuter les notebooks Jupyter et VS Code à partir d'Anaconda Prompt (c'est-à-dire via le Terminal) comme cela est indiqué dans le Chapitre 2. Cela garantit que vous serez accueilli par deux fenêtres contextuelles lorsque vous utilisez xlwings pour la première fois : la première affiche « Terminal demande l'autorisation de contrôler System Events » et la seconde « Terminal demande l'autorisation de contrôler Microsoft Excel ». Vous devez accepter en cliquant sur le bouton OK des deux fenêtres pour permettre à Python d'automatiser Excel. En théorie, ces fenêtres devraient être déclenchées par n'importe quelle application à partir de laquelle vous exécutez du code xlwings, mais en pratique, ce n'est

2 Notez que xlwings 0.22.0 a introduit la fonction `xw.load`, qui est similaire à `xw.view`, mais qui fonctionne dans la direction opposée : cette fonction permet de charger facilement une plage Excel dans un notebook Jupyter sous la forme d'un DataFrame pandas (cf. https://docs.xlwings.org/en/stable/jupyternotebooks.html).

souvent pas le cas, si bien qu'exécuter les commandes à partir du Terminal vous évitera des ennuis. En outre, vous devrez ouvrir les préférences d'Excel et décocher la case « Afficher la bibliothèque de classeurs à l'ouverture d'Excel » dans la catégorie Général. Cela ouvre Excel directement sur un classeur vide au lieu d'ouvrir d'abord la galerie, ce qui serait gênant en cas d'ouverture d'une nouvelle instance d'Excel via xlwings.

Modèle objet d'Excel

Lorsque vous travaillez avec Excel par programmation, vous interagissez avec ses composants comme les classeurs ou les feuilles de calcul. Ces composants sont organisés dans le *modèle objet d'Excel*, une structure hiérarchique qui représente l'interface utilisateur graphique d'Excel (voir la Figure 9.1). Microsoft utilise en grande partie le même modèle objet avec tous les langages de programmation officiellement pris en charge, qu'il s'agisse de VBA, Office Scripts (l'interface JavaScript pour Excel sur le Web) ou C#. Contrairement aux packages de lecture et d'écriture étudiés au Chapitre 8, xlwings suit de très près le modèle objet d'Excel, mais avec une bouffée d'air frais : par exemple, xlwings utilise les noms `app` au lieu de `application` et `book` au lieu de `workbook` (classeur) :

- Un objet `app` contient la collection `books`.
- Un objet `book` contient la collection `sheets`.
- Un objet `sheet` (feuille) donne accès à des objets `range` (plage) et à des collections telles que `charts`.
- Un objet `range` contient une ou plusieurs cellules contiguës qui constituent ses éléments.

Les zones en pointillé sont des *collections* et contiennent un ou plusieurs objets du même type. Un objet `app` correspond à une instance d'Excel, c'est-à-dire une application Excel qui s'exécute en tant que processus distinct. Les utilisateurs expérimentés utilisent parfois plusieurs instances d'Excel en parallèle pour ouvrir le même classeur deux fois, par exemple, afin de réaliser des calculs dans un classeur avec différentes entrées en parallèle. Avec les versions les plus récentes d'Excel, Microsoft a rendu un peu plus compliqué l'ouverture manuelle de plusieurs instances d'Excel : il faut démarrer Excel, puis cliquer avec le bouton droit de la souris sur son icône dans la barre des tâches Windows. Dans le menu qui apparaît, il faut cliquer avec le bouton gauche de la souris sur l'entrée Excel tout en maintenant la touche Alt enfoncée (assurez-vous de maintenir la touche Alt enfoncée jusqu'à ce que vous relâchiez le bouton de la souris) ; une fenêtre contextuelle vous demandera alors si vous souhaitez démarrer une nouvelle instance d'Excel. Sous macOS, il n'existe aucun moyen manuel de lancer plus d'une instance du même programme, mais vous pouvez lancer plusieurs instances d'Excel par programmation *via* xlwings, comme nous le verrons plus tard. Pour résumer, une instance d'Excel est un environnement *sandboxé* (bac à sable), ce qui signifie qu'une instance ne peut pas communiquer avec une autre instance[3].

3 Pour obtenir des informations complémentaires sur les instances d'Excel distinctes, consultez l'article suivant : https://answers.microsoft.com/en-us/msoffice/forum/msoffice_excel-mso_win10-mso_2016/what-are-excel-instances-and-why-is-this-important/20c39a6f-0857-4033-b713-18bf72e91d8b

L'objet `sheet` donne accès à des collections telles que les graphiques, les images et les noms définis, sujets que nous aborderons dans la deuxième section de ce chapitre.

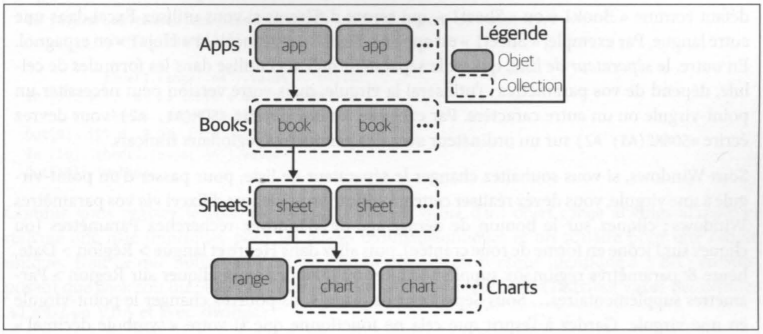


Figure 9.1 : Modèle objet d'Excel tel qu'il est implémenté par xlwings (extrait).

Pour vous familiariser avec le modèle objet d'Excel, comme d'habitude, il est préférable de le tester de manière interactive. Commençons par la classe `Book` qui permet de créer de nouveaux classeurs et de se connecter à des classeurs existants ; le Tableau 9.1 propose une vue d'ensemble.

Tableau 9.1 : Exploitation de classeurs Excel.

Commande	Description
`xw.Book()`	Renvoie un objet `book` représentant un nouveau classeur Excel dans l'instance d'Excel active. S'il n'y a pas d'instance active, Excel est démarré.
`xw.Book("Book1")`	Renvoie un objet `book` représentant un classeur non enregistré portant le nom Book1 (nom sans extension de fichier).
`xw.Book("Book1.xlsx")`	Renvoie un objet `book` représentant un classeur précédemment enregistré sous le nom *Book1.xlsx* (nom avec extension de fichier). Le fichier doit être ouvert ou dans le répertoire de travail en cours.
`xw.Book(r"C:\chemin\Book1.xlsx")`	Renvoie un objet `book` d'un classeur précédemment enregistré (chemin d'accès complet au fichier). Le fichier peut être ouvert ou fermé. Le paramètre `r` en en-tête transforme la chaîne en chaîne brute afin que les barres obliques inverses (\) du chemin soient interprétées littéralement sous Windows (j'ai évoqué les chaînes brutes au Chapitre 5). Sous macOS, le paramètre `r` n'est pas requis car les chemins d'accès aux fichiers utilisent des barres obliques à la place des barres obliques inverses.
`xw.books.active`	Renvoie un objet `book` représentant le classeur actif dans l'instance d'Excel active.

Paramètres linguistiques et régionaux

Ce livre est basé sur la version américaine d'Excel. Je me référerai parfois à des noms par défaut comme « Book1 » ou « Sheet1 », qui seront différents si vous utilisez Excel dans une autre langue. Par exemple, « Sheet1 » est appelé « Feuil1 » en français et « Hoja1 » en espagnol. En outre, le *séparateur de liste*, qui est le séparateur qu'Excel utilise dans les formules de cellule, dépend de vos paramètres : j'utiliserai la virgule, mais votre version peut nécessiter un point-virgule ou un autre caractère. Par exemple, au lieu d'écrire `=SUM(A1, A2)`, vous devrez écrire `=SOMME(A1; A2)` sur un ordinateur avec des paramètres régionaux français.

Sous Windows, si vous souhaitez changer le séparateur de liste, pour passer d'un point-virgule à une virgule, vous devez réaliser cette modification en dehors d'Excel *via* vos paramètres Windows : cliquez sur le bouton de démarrage de Windows, recherchez Paramètres (ou cliquez sur l'icône en forme de roue crantée), puis allez dans Heure et langue > Région > Date, heure & paramètres régionaux supplémentaires où il faudra enfin cliquer sur Région > Paramètres supplémentaires… Sous Séparateur de listes, vous pourrez changer le point-virgule en une virgule. Gardez à l'esprit que cela ne fonctionne que si votre « symbole décimal » (dans le même menu) n'est pas aussi une virgule. Pour remplacer les séparateurs de décimale et de milliers à l'échelle du système (mais pas le séparateur de liste), dans Excel, accédez à Options > Options avancées, où vous trouverez les paramètres sous Options d'édition.

Sous macOS, cela fonctionne de la même manière, mais vous ne pouvez pas modifier directement le séparateur de liste : dans les Préférences Système de macOS (et non d'Excel), sélectionnez Langue et région, puis définissez une région spécifique globalement (sous l'onglet Général) ou spécifiquement pour Excel (sous l'onglet Apps).

Voyons comment nous pouvons parcourir la hiérarchie du modèle objet de l'objet `book` jusqu'à l'objet `range` :

```
In [4]: # Crée un nouveau classeur vide et affiche son nom. C'est
        # le classeur que nous allons utiliser pour exécuter
        # la plupart des exemples de code de ce chapitre.
        book = xw.Book()
        book.name
Out[4]: 'Classeur2'
In [5]: # Accès à la collection des feuilles de calcul
        book.sheets
Out[5]: Sheets([<Sheet [Classeur2]Feuil1>])
In [6]: # Lit un objet sheet par son indice ou son nom. Vous devez modifier
        # le nom "Feuil1" si votre feuille porte un nom différent
        sheet1 = book.sheets[0]
        sheet1 = book.sheets["Feuil1"]
In [7]: sheet1.range("A1")
Out[7]: <Range [Classeur2]Feuil1!$A$1>
```

Avec l'objet `range`, nous sommes arrivés au bas de la hiérarchie. La chaîne qui est affichée entre crochets donne des informations utiles sur cet objet, mais pour faire quelque chose, on utilise généralement l'objet avec un attribut, comme dans l'exemple suivant :

```
In [8]: # Les tâches les plus courantes sont l'écriture de valeurs...
        sheet1.range("A1").value = [[1, 2],
                                    [3, 4]]
        sheet1.range("A4").value = "Hello!"
In [9]: # ... et la lecture de valeurs
        sheet1.range("A1:B2").value
Out[9]: [[1.0, 2.0], [3.0, 4.0]]
In [10]: sheet1.range("A4").value
Out[10]: 'Hello!'
```

Comme vous pouvez le voir, par défaut, l'attribut `value` d'un objet `range` xlwings accepte et renvoie une liste imbriquée pour les plages bidimensionnelles et un scalaire pour une cellule unique. Tout ce que nous avons utilisé jusqu'à présent est presque identique à VBA : en supposant que `book` soit un objet de classeur, voici comment on accède à l'attribut `value` des cellules A1 à B2 en VBA et avec xlwings :

```
book.Sheets(1).Range("A1:B2").Value # VBA
book.sheets[0].range("A1:B2").value # xlwings
```

Les différences sont les suivantes :

- *Attributs*. Python utilise des lettres minuscules, éventuellement avec des barres de soulignement comme le suggère PEP 8, le guide de style de Python que j'ai présenté au Chapitre 3.
- *Indices*. Python utilise des crochets et des indices à partir de zéro pour accéder à un élément de la collection `sheets`.

Le Tableau 9.3 donne une vue d'ensemble des chaînes acceptées par un objet `range` xlwings.

Tableau 9.3 : Chaînes pour définir une plage en notation A1.

Référence	Description
`"A1"`	Une seule cellule
`"A1:B2"`	Cellules de A1 à B2
`"A:A"`	Colonne A
`"A:B"`	Colonnes A à B
`"1:1"`	Ligne 1
`"1:2"`	Lignes 1 à 2

L'indexation et le découpage fonctionnent avec des objets `range` xlwings : examinez l'adresse entre les signes < et > (la représentation de l'objet affiché) pour voir quelle plage de cellules vous obtenez :

```
In [11]: # Indexation
sheet1.range("A1:B2")[0, 0]
Out[11]: <Range [Classeur2]Feuil1!$A$1>
In [12]: # Découpage
sheet1.range("A1:B2")[:, 1]
Out[12]: <Range [Classeur2]Feuil1!$B$1:$B$2>
```

L'indexation correspond à l'utilisation de la propriété `Cells` en VBA :

```
book.Sheets(1).Range("A1:B2").Cells(1, 1) # VBA
book.sheets[0].range("A1:B2")[0, 0] # xlwings
```

Au lieu d'utiliser explicitement `range` comme attribut de l'objet `sheet`, vous pouvez également obtenir un objet `range` en indexant et en découpant l'objet `sheet`. L'utilisation de cette technique avec la notation A1 économise de la saisie, et le fait d'employer des indices entiers donne à la feuille Excel l'impression d'être un tableau NumPy :

```
In [13]: # Une seule cellule : notation A1
         sheet1["A1"]
Out[13]: <Range [Classeur2]Feuil1!$A$1>
In [14]: # Plusieurs cellules : notation A1
         sheet1["A1:B2"]
Out[14]: <Range [Classeur2]Feuil1!$A$1:$B$2>
In [15]: # Une seule cellule : indexation
         sheet1[0, 0]
Out[15]: <Range [Classeur2]Feuil1!$A$1>
In [16]: # Plusieurs cellules : découpage
         sheet1[:2, :2]
Out[16]: <Range [Classeur2]Feuil1!$A$1:$B$2>
```

Parfois, cependant, il peut être plus intuitif de définir une plage en se référant à la cellule supérieure gauche et inférieure droite d'une plage. Les exemples suivants font référence aux plages de cellules D10 et D10:F11, respectivement, ce qui permet de comprendre la différence entre l'indexation et le découpage d'un objet `sheet` et l'utilisation d'un objet `range` :

```
In [17]: # D10 avec indexation de sheet
         sheet1[9, 3]
Out[17]: <Range [Classeur2]Feuil1!$D$10>
In [18]: # D10 via objet range
         sheet1.range((10, 4))
Out[18]: <Range [Classeur2]Feuil1!$D$10>
In [19]: # D10:F11 avec découpage de sheet
         sheet1[9:11, 3:6]
Out[19]: <Range [Classeur2]Feuil1!$D$10:$F$11>
In [20]: # D10:F11 via objet range
         sheet1.range((10, 4), (11, 6))
Out[20]: <Range [Classeur2]Feuil1!$D$10:$F$11>
```

La définition d'objets `range` avec des tuples est très similaire au fonctionnement de la propriété `Cells` en VBA, comme le montre la comparaison suivante : cela suppose à nouveau que `book` soit un objet `workbook` VBA ou un objet `book` xlwings. Examinons d'abord la version VBA :

```
With book.Sheets(1)
    myrange = .Range(.Cells(10, 4), .Cells(11, 6))
End With
```

Ceci est équivalent à l'expression xlwings suivante :

```
myrange = book.sheets[0].range((10, 4), (11, 6))
```



En tant que package Python, xlwings utilise systématiquement des indices à partir de zéro chaque fois que vous accédez à des éléments via la syntaxe d'indexation ou de découpage de Python, c'est-à-dire en employant des crochets. Toutefois, les objets `range` xlwings utilisent les indices de lignes et de colonnes d'Excel qui commencent à 1. Avoir les mêmes indices de lignes et de colonnes que l'interface utilisateur d'Excel peut se révéler très utile. Si vous préférez cependant n'utiliser que des indices à partir de zéro, utilisez simplement la syntaxe `sheet[sélection_ligne, sélection_colonne]`.

L'exemple suivant montre comment passer un objet `range` (`sheet1["A1"]`) jusqu'à l'objet `app`. N'oubliez pas que l'objet `app` représente une instance d'Excel (la sortie représente l'ID de processus d'Excel et sera donc différente sur votre machine) :

```
In [21]: sheet1["A1"].sheet.book.app
Out[21]: <Excel App 4916>
```

Arrivé au sommet du modèle objet d'Excel, c'est le bon moment pour voir comment vous pouvez travailler avec plusieurs instances d'Excel. Vous devez utiliser l'objet `app` explicitement si vous souhaitez ouvrir le même classeur dans plusieurs instances d'Excel ou si vous souhaitez spécifiquement distribuer vos classeurs sur différentes instances pour des raisons de performances. Il est également courant de travailler avec un objet `app` pour ouvrir un classeur dans une instance d'Excel masquée : cela permet d'exécuter un script xlwings en arrière-plan sans empêcher d'effectuer d'autres tâches dans Excel pendant ce temps :

```
In [22]: # On crée un objet app à partir d'un classeur ouvert
         # et on crée une autre instance invisible
         visible_app = sheet1.book.app
         invisible_app = xw.App(visible=False)
In [23]: # Liste les noms des classeurs ouverts dans chaque instance
         # en utilisant une compréhension de liste
         [book.name for book in visible_app.books]
Out[23]: ['Classeur1', 'Classeur2']
In [24]: [book.name for book in invisible_app.books]
Out[24]: ['Classeur3']
In [25]: # La propriété key d'une app représente le numéro de processus (PID)
         xw.apps.keys()
Out[25]: [5288, 4916]
In [26]: # On peut aussi y accéder par l'attribut pid
         xw.apps.active.pid
Out[26]: 5288
In [27]: # On exploite le classeur dans l'instance invisible d'Excel
         invisible_book = invisible_app.books[0]
         invisible_book.sheets[0]["A1"].value = "Created by an invisible app."
```

```
In [28]: # On enregistre le classeur Excel dans le répertoire xl
         invisible_book.save("xl/invisible.xlsx")
In [29]: # Quitte l'instance invisible d'Excel
         invisible_app.quit()
```



Si vous exécutez la commande `save` sous macOS, vous obtiendrez dans Excel une fenêtre contextuelle qui vous demande d'octroyer l'accès aux fichiers et vous devrez confirmer en cliquant sur le bouton Sélectionner avant de cliquer sur le bouton Accorder l'accès. Sous macOS, Excel est en mode sandbox (bac à sable), ce qui signifie que votre programme ne peut accéder aux fichiers et dossiers en dehors de l'application Excel que si vous validez la boîte de dialogue. Une fois que vous avez validé, Excel se souviendra des emplacements et ne vous embêtera plus lorsque vous exécuterez le script la fois suivante.

Si vous avez le même classeur ouvert dans deux instances d'Excel ou si vous souhaitez spécifier dans quelle instance d'Excel vous désirez ouvrir un classeur, vous ne pouvez plus utiliser `xw.Book`. Au lieu de cela, vous devez utiliser la collection `books` comme cela est indiqué dans le Tableau 9.3. Notez que `myapp` représente un objet `app` xlwings. Si vous remplacez `myapp.books` par `xw.books`, xlwings utilisera l'objet `app` actif.

Tableau 9.3 : Exploitation de la collection books.

Commande	Description
`myapp.books.add()`	Crée un classeur Excel dans l'instance d'Excel à laquelle `myapp` fait référence et renvoie l'objet `book` correspondant.
`myapp.books.open(r"C:\chemin\Book.xlsx")`	Renvoie l'objet `book` s'il est déjà ouvert, sinon il est ouvert dans l'instance d'Excel à laquelle `myapp` fait référence. N'oubliez pas que le `r` en en-tête transforme le chemin d'accès au fichier en une chaîne brute pour interpréter les barres obliques inverses littéralement.
`myapp.books["Book1.xlsx"]`	Renvoie l'objet `book` s'il est ouvert. Cela déclenchera une exception `KeyError` s'il n'est pas encore ouvert. Assurez-vous d'utiliser le nom et non pas le chemin d'accès complet. Utilisez-le si vous avez besoin de savoir si un classeur est déjà ouvert dans Excel.

Avant d'approfondir la façon dont xlwings peut *remplacer* vos macros VBA, voyons comment xlwings peut *interagir* avec votre code VBA existant : cela peut être utile si vous avez beaucoup de code et que vous n'avez pas le temps de tout migrer vers Python.

Exécution du code VBA

Si vous avez des projets Excel avec beaucoup de code VBA, cela peut représenter un travail important pour tout migrer vers Python. Dans ce cas, vous pouvez utiliser Python pour exécuter vos macros VBA. L'exemple suivant utilise le fichier *vba.xlsm* que vous trouverez dans le dossier *xl* des exemples de code. Il contient le code suivant dans le Module1 :

```
Function MySum(x As Double, y As Double) As Double
    MySum = x + y
End Function
Sub ShowMsgBox(msg As String)
    MsgBox msg
End Sub
```

Pour appeler ces fonctions *via* Python, vous devez d'abord instancier un objet `macro` xlwings que vous appellerez par la suite, ce qui donne l'impression qu'il s'agit d'une fonction Python native :

```
In [30]: vba_book = xw.Book("xl/vba.xlsm")
In [31]: # Instancie un objet macro avec la fonction VBA
         mysum = vba_book.macro("Module1.MySum")
         # Appelle une fonction VBA
         mysum(5, 4)
Out[31]: 9.0
In [32]: # Fonctionne de la même manière avec une procédure Sub VBA
         show_msgbox = vba_book.macro("Module1.ShowMsgBox")
         show_msgbox("Hello xlwings!")
In [33]: # Ferme à nouveau le classeur (assurez-vous de fermer d'abord MessageBox)
         vba_book.close()
```



Si vous stockez la fonction VBA `MySum` dans le module de classeur `ThisWorkbook` ou dans un module de feuille (par exemple, Feuil1), vous devez l'appeler via `ThisWorkbook.MySum` ou `Feuil1.MySum`. Cependant, vous ne pourrez pas accéder à la valeur de retour de la fonction à partir de Python, si bien qu'il est préférable de stocker les fonctions VBA dans un module de code VBA standard que l'on insère en cliquant avec le bouton droit sur le dossier Modules de l'éditeur VBA.

Maintenant que vous savez comment interagir avec le code VBA existant, nous pouvons poursuivre notre exploration de xlwings en examinant comment l'utiliser avec des DataFrames, des tableaux NumPy et des collections telles que des graphiques, des images et des noms définis.

Convertisseurs, options et collections

Dans les exemples de code qui introduisaient ce chapitre, nous avons déjà lu et écrit une chaîne et une liste imbriquée à l'aide de l'attribut `value` d'un objet `range` xlwings. Je commencerai cette section en vous montrant comment cela fonctionne avec les DataFrames pandas avant d'examiner de plus près la méthode `options` qui permet d'influencer la façon dont xlwings lit et écrit les valeurs. Nous poursuivrons avec les graphiques, les images et les noms définis, qui sont les collections auxquelles on accède habituellement à partir d'un objet `sheet`. Grâce à ces nouvelles connaissances sur xlwings, nous nous pencherons à nouveau sur l'étude de cas du Chapitre 7.

Exploitation des DataFrames

L'écriture d'un DataFrame dans Excel n'est pas différente de l'écriture d'un scalaire ou d'une liste imbriquée dans Excel : il suffit d'affecter le DataFrame à la cellule supérieure gauche d'une plage Excel :

```
In [34]: data=[["Mark", 55, "Italy", 4.5, "Europe"],
               ["John", 33, "USA", 6.7, "America"]]
         df = pd.DataFrame(data=data,
                           columns=["name", "age", "country",
                                    "score", "continent"],
                           index=[1001, 1000])
         df.index.name = "user_id"
         df
Out[34]:
```

user_id	name	age	country	score	continent
1001	Mark	55	Italy	4.5	Europe
1000	John	33	USA	6.7	America

```
In [35]: sheet1["A6"].value = df
```

Si, toutefois, vous souhaitez supprimer les en-têtes des colonnes ou des index, utilisez la méthode `options` de la manière suivante :

```
In [36]: sheet1["B10"].options(header=False, index=False).value = df
```

La lecture de plages Excel sous la forme de DataFrames nécessite que vous fournissiez la classe `DataFrame` comme paramètre `convert` à la méthode `options`. Par défaut, vos données présentent à la fois un en-tête et un index, mais vous pouvez à nouveau utiliser les paramètres `index` et `header` pour modifier cela. Au lieu d'utiliser le convertisseur, vous pouvez également lire d'abord les valeurs sous la forme d'une liste imbriquée, puis construire manuellement votre DataFrame, mais l'utilisation du convertisseur facilite un peu la gestion de l'index et de l'en-tête.



Dans l'exemple de code suivant, je présente la méthode `expand` qui facilite la lecture d'un bloc contigu de cellules, offrant la même plage que si vous faisiez Maj + Ctrl + Flèche bas + Flèche droite dans Excel, sauf que la méthode `expand` saute par-dessus une cellule vide dans le coin supérieur gauche.

```
In [37]: df2 = sheet1["A6"].expand().options(pd.DataFrame).value
         df2
Out[37]:
```

user_id	name	age	country	score	continent
1001.0	Mark	55.0	Italy	4.5	Europe
1000.0	John	33.0	USA	6.7	America

```
In [38]: # Si vous voulez que l'index soit un entier,
```

```
        # vous pouvez changer son type de données
        df2.index = df2.index.astype(int)
        df2
Out[38]:
```

user_id	name	age	country	score	continent
1001	Mark	55.0	Italy	4.5	Europe
1000	John	33.0	USA	6.7	America

```
In [39]: # En définissant index=False, on place toutes les valeurs Excel dans
         # la partie données du DataFrame et on utilise l'index par défaut
         sheet1["A6"].expand().options(pd.DataFrame, index=False).value
Out[39]:
```

	user_id	name	age	country	score	continent
0	1001.0	Mark	55.0	Italy	4.5	Europe
1	1000.0	John	33.0	USA	6.7	America

Après un premier exemple de lecture et d'écriture de DataFrames illustrant le fonctionnement des convertisseurs et des options, nous allons voir la façon dont ils sont formellement définis et comment les utiliser avec d'autres structures de données.

Convertisseurs et options

Comme nous venons de le voir, la méthode `options` de l'objet `range` xlwings permet d'influencer la façon dont les valeurs sont lues et écrites. Autrement dit, la méthode `options` n'est évaluée que lorsque vous appelez l'attribut `value` sur un objet `range`. La syntaxe est la suivante (`myrange` est un objet `range` xlwings) :

```
myrange.options(convert=None, option1=value1, option2=value2, ...).value
```

Le Tableau 9.4 affiche les convertisseurs intégrés, c'est-à-dire les valeurs acceptées par l'argument `convert`. Ils sont appelés *intégrés* car xlwings offre un moyen d'écrire vos propres convertisseurs, ce qui est utile si vous devez appliquer à plusieurs reprises des transformations supplémentaires avant d'écrire ou après avoir lu des valeurs (pour voir comment cela fonctionne, reportez-vous à la documentation de xlwings[4]).

Tableau 9.4 : Convertisseurs intégrés.

Convertisseur	Description
`dict`	Dictionnaires simples sans imbrication, c'est-à-dire sous la forme {clé1: valeur1, clé2: valeur2, ...}
`np.array`	Tableaux NumPy, nécessite `import numpy as np`
`pd.Series`	Series pandas, nécessite `import pandas as pd`
`pd.DataFrame`	DataFrame pandas, nécessite `import pandas as pd`

4 https://docs.xlwings.org/en/stable/converters.html#custom-converter

Nous avons déjà utilisé les options `index` et `header` avec l'exemple de DataFrame, mais d'autres options sont disponibles, comme cela est indiqué dans le Tableau 9.5.

Tableau 9.5 : Options intégrées.

Option	Description
`empty`	Par défaut, les cellules vides sont interprétées comme `None`. Vous pouvez modifier ce comportement en fournissant une valeur à `empty`.
`date`	Accepte une fonction appliquée aux valeurs des cellules au format date.
`number`	Accepte une fonction appliquée aux nombres.
`ndim`	Nombre de dimensions : lors de la lecture, utilisez `ndim` pour forcer les valeurs d'une plage à se présenter dans un nombre de dimensions défini. La valeur doit être égale à `None`, `1`, ou `2`. Peut être utilisé lors de la lecture de valeurs sous la forme de listes ou de tableaux NumPy.
`transpose`	Transpose les valeurs, c'est-à-dire transforme les colonnes en lignes ou *vice versa*.
`index`	À utiliser avec les DataFrames et les Series pandas : lors de la lecture, utilisez cette option pour définir si la plage Excel contient un index. La valeur peut être égale à `True`/`False` ou à un entier. L'entier définit le nombre de colonnes à transformer en un `MultiIndex`. Par exemple, la valeur `2` utilisera les deux colonnes les plus à gauche comme index. Lors de l'écriture, vous pouvez décider si vous souhaitez écrire un index en définissant l'option `index` à `True` ou `False`.
`header`	Fonctionne de la même manière que l'option `index`, mais elle est appliquée aux en-têtes de colonnes.

Regardons de plus près `ndim` : par défaut, lorsque vous lisez une seule cellule à partir d'Excel, vous obtenez un scalaire (par exemple, un float ou une chaîne) ; lorsque vous lisez une colonne ou une ligne, vous obtenez une liste simple ; et enfin, lorsque vous lisez une plage bidimensionnelle, vous obtenez une liste imbriquée (c'est-à-dire bidimensionnelle). Ceci est non seulement cohérent, mais c'est aussi équivalent à la façon dont le découpage fonctionne avec les tableaux NumPy, comme nous l'avons vu dans le Chapitre 4. Le cas unidimensionnel est un cas particulier : parfois, une colonne peut simplement être un cas limite de ce qui serait autrement une plage bidimensionnelle. Dans ce cas, il est logique de forcer une plage à toujours se présenter sous la forme d'une liste bidimensionnelle en utilisant `ndim=2` :

```
In [40]: # Plage horizontale (une dimension)
         sheet1["A1:B1"].value
Out[40]: [1.0, 2.0]
In [41]: # Plage verticale (une dimension)
         sheet1["A1:A2"].value
Out[41]: [1.0, 3.0]
In [42]: # Plage horizontale (deux dimensions)
         sheet1["A1:B1"].options(ndim=2).value
Out[42]: [[1.0, 2.0]]
In [43]: # Plage verticale (deux dimensions)
         sheet1["A1:A2"].options(ndim=2).value
Out[43]: [[1.0], [3.0]]
In [44]: # L'emploi du convertisseur de tableau NumPy se comporte de la même
         # manière : la plage verticale génère un tableau à une dimension
         sheet1["A1:A2"].options(np.array).value
Out[44]: array([1., 3.])
```

```
In [45]: # On préserve l'orientation de la colonne
         sheet1["A1:A2"].options(np.array, ndim=2).value
Out[45]: array([[1.],
                [3.]])
In [46]: # Si vous devez écrire une liste verticalement,
         # l'option "transpose" se révèle pratique
         sheet1["D1"].options(transpose=True).value = [100, 200]
```

Utilisez `ndim=1` pour forcer la lecture de la valeur d'une seule cellule sous la forme d'une liste au lieu d'un scalaire. Vous n'aurez pas besoin de `ndim` avec pandas, car un DataFrame est toujours bidimensionnel et une Series est toujours unidimensionnelle. Voici un autre exemple montrant le fonctionnement des options `empty`, `date` et `number` :

```
In [47]: # Écrit un échantillon de date
         sheet1["A13"].value = [dt.datetime(2020, 1, 1), None, 1.0]
In [48]: # On la lit avec les options par défaut
         sheet1["A13:C13"].value
Out[48]: [datetime.datetime(2020, 1, 1, 0, 0), None, 1.0]
In [49]: # On lit la date en utilisant d'autres options
         sheet1["A13:C13"].options(empty="NA",
                                   dates=dt.date,
                                   numbers=int).value
Out[49]: [datetime.date(2020, 1, 1), 'NA', 1]
```

Jusqu'à présent, nous avons travaillé avec les objets `book`, `sheet` et `range`. Nous allons maintenant voir comment gérer des collections telles que les graphiques auxquels on accède à partir de l'objet `sheet`.

Graphiques, images et noms définis

Dans cette section, je vais vous montrer comment travailler avec trois collections auxquelles on accède *via* les objets `sheet` ou `book` : les graphiques, les images et les noms définis[5]. xlwings ne prend en charge que les fonctionnalités graphiques les plus élémentaires, mais comme vous pouvez travailler avec des modèles, il ne vous manquera sans doute pas grand-chose. Et pour compenser, xlwings permet d'intégrer des tracés Matplotlib sous forme d'images (si vous vous souvenez du Chapitre 5, vous savez que Matplotlib est le backend de traçage par défaut proposé par pandas). Commençons par créer un premier graphique Excel !

Graphiques Excel

Pour ajouter un nouveau graphique, utilisez la méthode `add` de la collection `charts`, puis définissez le type de graphique et les données source :

5 Il existe aussi une autre collection populaire : `tables`. Pour l'utiliser, vous aurez besoin au minimum de la version xlwings 0.21.0 (voir la documentation à https://docs.xlwings.org/en/stable/api.html#xlwings.main.Tables)

```
In [50]: sheet1["A15"].value = [[None, "North", "South"],
                                ["Last Year", 2, 5],
                                ["This Year", 3, 6]]
In [51]: chart = sheet1.charts.add(top=sheet1["A19"].top,
                                   left=sheet1["A19"].left)
         chart.chart_type = "column_clustered"
         chart.set_source_data(sheet1["A15"].expand())
```

Cela produira le graphique illustré sur le côté gauche de la Figure 9.2. Pour connaître les types de graphiques disponibles, consultez la documentation de xlwings[6]. Si vous préférez les tracés de pandas plutôt que les graphiques Excel, ou si vous souhaitez utiliser un type de graphique qui n'est pas disponible dans Excel, xlwings a tout ce qu'il vous faut !

Images : tracés Matplotlib

Lorsque vous utilisez le backend par défaut de pandas pour générer des graphiques, vous créez un tracé Matplotlib. Pour transférer un tel tracé dans Excel, vous devez d'abord mettre la main sur son objet `figure`, que vous fournissez comme argument à `pictures.add`, ce qui convertira le tracé en image et l'enverra à Excel :

```
In [52]: # Lit les données du graphique sous la forme d'un DataFrame
         df = sheet1["A15"].expand().options(pd.DataFrame).value
         df
Out[52]:
```

	North	South
Last Year	2.0	5.0
This Year	3.0	6.0

```
In [53]: # Active Matplotlib avec une commande magique
         # et passe au style "seaborn"
         %matplotlib inline
         import matplotlib.pyplot as plt
         plt.style.use("seaborn")
In [54]: # La méthode plot de pandas retourne un objet "axis" à partir
         # duquel vous pouvez obtenir la figure. "T" transpose le
         # DataFrame pour donner l'orientation souhaitée au graphique.
         ax = df.T.plot.bar()
         fig = ax.get_figure()
In [55]: # Envoie le graphique à Excel
         plot = sheet1.pictures.add(fig, name="SalesPlot",
                                    top=sheet1["H19"].top,
                                    left=sheet1["H19"].left)
         # On met le graphique à l'échelle 70%
         plot.width, plot.height = plot.width * 0.7, plot.height * 0.7
```

6 https://docs.xlwings.org/en/stable/api.html#xlwings.Chart.chart_type

Pour mettre à jour l'image avec un nouveau tracé, utilisez simplement la méthode `update` avec un autre objet `figure` : techniquement, cela remplacera l'image dans Excel mais conservera toutes les propriétés telles que l'emplacement, la taille et le nom :

```
In [56]: ax = (df + 1).T.plot.bar()
         plot = plot.update(ax.get_figure())
```

La Figure 9.2 illustre la comparaison entre le graphique Excel et le tracé Matplotlib après l'appel à `update`.

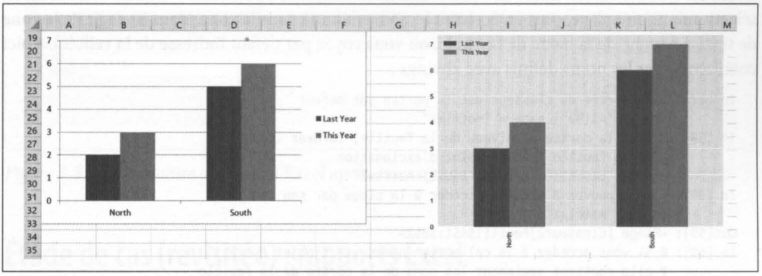


Figure 9.2 : Un graphique Excel (à gauche) et un tracé Matplotlib (à droite).



Lorsque vous travaillez avec des images, assurez-vous que Pillow[7], la bibliothèque de référence de Python pour les images, est installée : cela permettra de s'assurer que les images sont transmises à Excel dans la bonne taille et les bonnes proportions. Pillow fait partie d'Anaconda, et si vous utilisez une distribution différente, vous devrez donc l'installer en exécutant `conda install pillow` ou `pip install pillow`. Notez que `pictures.add` accepte également un chemin d'accès vers une image stockée sur disque à la place d'une figure Matplotlib.

Les graphiques et les images sont des collections accessibles *via* un objet `sheet`. Les *noms définis* (la collection que nous allons examiner ensuite) sont accessibles à partir des objets `sheet` ou `book`. Voyons quelle différence cela fait !

Noms définis

Dans Excel, on crée un *nom défini* en affectant un nom à une plage, une formule ou une constante[8]. L'attribution d'un nom à une plage est probablement le cas le plus courant et permet d'obtenir ce que l'on appelle une *plage nommée*. Avec une plage nommée, vous pouvez faire

7 https://pillow.readthedocs.io/en/stable/

8 Les noms définis avec des formules sont également utilisés pour les fonctions lambda, une nouvelle façon d'écrire des fonctions définies par l'utilisateur sans employer VBA ni JavaScript ; Microsoft a annoncé cette nouvelle fonctionnalité pour les abonnés Microsoft 365 en décembre 2020.

référence à la plage Excel dans les formules et le code en utilisant un nom descriptif plutôt qu'une adresse abstraite sous la forme A1:B2. En utilisant des plages nommées avec xlwings, vous rendez votre code plus flexible et plus robuste : la lecture et l'écriture de valeurs dans des plages nommées donnent la possibilité de restructurer votre classeur sans avoir à modifier votre code Python : un nom reste attaché à la cellule, même si vous la déplacez en insérant une nouvelle ligne, par exemple. Les noms définis peuvent être définis à l'échelle globale du classeur ou à l'échelle d'une feuille. L'avantage d'un nom ayant une portée à l'échelle de la feuille est que vous pouvez copier la feuille sans rencontrer de conflits avec des plages nommées en double. Dans Excel, on ajoute manuellement des noms définis en accédant à Formules > Définir le nom ou en sélectionnant une plage, puis en écrivant le nom souhaité dans la zone Nom (il s'agit de la zone de texte à gauche de la barre de formule, où vous voyez par défaut l'adresse de la cellule). Voici comment gérer les noms définis avec xlwings :

```
In [57]: # La portée du classeur est la portée par défaut
         sheet1["A1:B2"].name = "matrix1"
In [58]: # Pour la portée au niveau de la feuille, suffixez le nom
         # de la feuille avec un point d'exclamation
         sheet1["B10:E11"].name = "Feuil1!matrix2"
In [59]: # Vous pouvez à présent accéder à la plage par son nom
         sheet1["matrix1"]
Out[59]: <Range [Classeur2]Feuil1!$A$1:$B$2>
In [60]: # Si vous accédez à la collection names via l'objet "sheet1",
         # elle contient seulement les noms de la portée de la feuille
         sheet1.names
Out[60]: [<Name 'Feuil1!matrix2': =Feuil1!$B$10:$E$11>]
In [61]: # Si vous accédez à la collection names via l'objet "book",
         # elle contient tous les noms (portée classeur et feuille)
         book.names
Out[61]: [<Name 'matrix1': =Feuil1!$A$1:$B$2>, <Name 'Feuil1!matrix2': =Feuil1!$B$10:$E$11>]
In [62]: # Les noms ont des méthodes et des attributs.
         # Vous pouvez, par exemple, lire la plage d'un objet.
         book.names["matrix1"].refers_to_range
Out[62]: <Range [Classeur2]Feuil1!$A$1:$B$2>
In [63]: # Si vous voulez assigner un nom à une constante
         # ou à une formule, utilisez la méthode "add"
         book.names.add("EURUSD", "=1,1151")
Out[63]: <Name 'EURUSD': =1.1151>
```

Jetez un coup d'œil aux noms définis générés dans Excel en ouvrant le Gestionnaire de noms *via* Formules > Gestionnaire de noms (voir la Figure 9.3). Notez qu'Excel sous macOS n'a pas de gestionnaire de noms, et il faut alors accéder à Formules > Définir le nom, où vous verrez les noms existants.

À ce stade, vous savez comment utiliser les composants les plus couramment exploités dans un classeur Excel. Cela signifie que nous pouvons nous repencher sur l'étude de cas du Chapitre 7 où nous générons un rapport. Nous allons voir les bénéfices que nous pouvons retirer de l'emploi de xlwings.

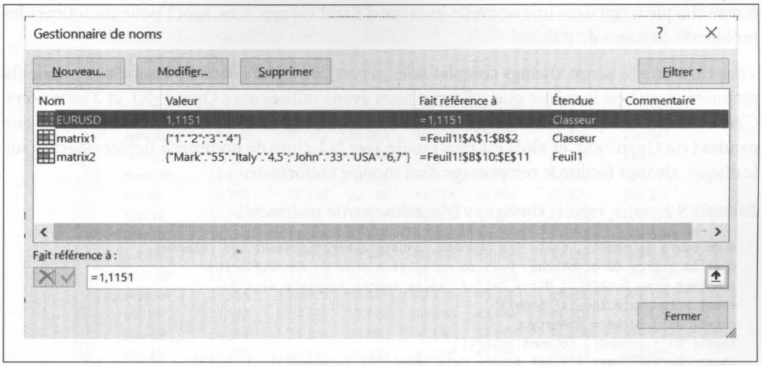


Figure 9.3 : Gestionnaire de noms d'Excel après avoir ajouté quelques noms définis via xlwings.

Étude de cas (revisitée) : rapports Excel

Le fait de pouvoir véritablement éditer des fichiers Excel *via* xlwings permet de travailler avec des fichiers de modèles qui seront conservés à 100 %, quelle que soit leur complexité ou leur format de stockage ; par exemple, vous pouvez facilement modifier un fichier *xlsb,* un cas qui n'est actuellement pris en charge par aucun des packages d'écriture que nous avons étudiés dans le chapitre précédent. Lorsque vous examinez le fichier *sales_report_openpxyl.py* dans les exemples de code, vous constatez qu'après avoir préparé le DataFrame `summary`, nous avons dû écrire près de quarante lignes de code pour créer un graphique et styler un DataFrame avec OpenPyXL. Avec xlwings, on obtient la même chose en seulement six lignes de code, comme cela est indiqué dans l'Exemple 9.1. Être capable de gérer la mise en forme dans le modèle Excel permet d'économiser beaucoup de temps. Cela a cependant un prix : xlwings nécessite qu'Excel soit installé pour s'exécuter ; cela ne pose généralement pas de problème si la création de ces rapports sur votre propre ordinateur n'est pas très fréquente, mais la situation peut être moins idéale si vous essayez de créer des rapports sur un serveur dans le cadre d'une application Web.

Premièrement, vous devez vous assurer que votre licence Microsoft Office permet l'installation sur un serveur et, deuxièmement, vous devez être conscient qu'Excel n'a pas été conçu pour une automatisation sur un serveur, ce qui signifie que vous pouvez rencontrer des problèmes de stabilité, surtout si vous devez générer de nombreux rapports en peu de temps. Cela étant dit, j'ai plusieurs clients qui ont réalisé avec succès ce type d'opération ; par conséquent, si vous ne pouvez pas utiliser un package d'écriture pour une raison quelconque, l'exécution de xlwings sur un serveur est une possibilité qui mérite d'être explorée. Assurez-vous simplement d'ex-

écuter chaque script dans une nouvelle instance d'Excel *via* `app = xw.App()` pour contourner les problèmes courants de stabilité.

Vous trouverez le script xlwings complet *sales_report_xlwings.py* dans les exemples de code (la première moitié est la même que celle que nous avons utilisée avec OpenPyXL et XlsxWriter). C'est aussi un exemple parfait de combinaison d'un package de lecteur avec xlwings : alors que pandas (*via* OpenPyXL et xlrd) est plus rapide avec la lecture de nombreux fichiers stockés sur le disque, xlwings facilite le remplissage d'un modèle préformaté.

Exemple 9.1 : sales_report_xlwings.py (deuxième partie seulement).

```
# On ouvre le modèle, colle les données, ajuste automatiquement les colonnes
# et la source du graphique. Puis on enregistre sous un nom différent.
template = xw.Book(this_dir / "xl" / "sales_report_template.xlsx")
sheet = template.sheets["Sheet1"]
sheet["B3"].value = summary
sheet["B3"].expand().columns.autofit()
sheet.charts["Chart 1"].set_source_data(sheet["B3"].expand()[:-1, :-1])
template.save(this_dir / "sales_report_xlwings.xlsx")
```

Lorsque vous exécutez ce script pour la toute première fois sous macOS (par exemple en l'ouvrant dans VS Code et en cliquant sur le bouton Exécuter), vous devrez à nouveau valider une fenêtre contextuelle pour accorder l'accès au système de fichiers, situation que nous avons déjà rencontrée plus tôt dans ce chapitre.

Avec des modèles Excel formatés, vous pouvez créer des rapports professionnels très rapidement. Vous avez également accès à des méthodes telles que `autofit`, fonctionnalité qui n'est pas disponible avec les packages d'écriture car elle repose sur des calculs effectués par l'application Excel : cela permet de définir correctement la largeur et la hauteur des cellules en fonction de leur contenu. La Figure 9.4 illustre la partie supérieure du rapport des ventes généré par xlwings avec un en-tête de tableau personnalisé ainsi que les colonnes où la méthode `autofit` a été appliquée.

Lorsque vous allez commencer à utiliser xlwings de façon intensive et que vous ne vous contenterez plus de remplir simplement quelques cellules dans un modèle, il sera souhaitable d'en savoir un peu plus sur son fonctionnement interne, et la section suivante étudie la manière dont xlwings fonctionne sous le capot.

Approfondissement de xlwings

Cette section montre comment améliorer les performances du code xlwings et pallier les fonctionnalités manquantes. Pour comprendre tout cela, nous devons cependant dire quelques mots sur la façon dont xlwings communique avec Excel.

	A	B	C	D	E	F	G	H	I
1		**Sales Report**							
2									
3		**Month**	**Washington DC**	**Boston**	**Las Vegas**	**New York**	**Chicago**	**San Francisco**	**Total**
4		**janv 19**	14 058	21 784	23 013	49 873	51 188	58 630	218 545
5		**févr 19**	15 235	21 455	25 493	46 670	52 331	55 219	216 403
6		**mars 19**	14 177	20 043	23 451	41 572	48 897	52 713	200 854
7		**avr 19**	13 339	18 791	22 710	41 714	47 396	49 325	193 276
8		**mai 19**	13 147	18 037	21 527	40 610	45 117	47 760	186 197
9		**juin 19**	14 284	21 556	21 985	47 266	49 460	53 462	208 014
10		**juil 19**	14 162	19 853	23 444	40 408	47 994	50 182	196 043
11		**août 19**	16 127	22 333	24 928	45 397	50 839	55 336	214 960
12		**sept 19**	14 994	19 925	24 411	42 831	49 096	49 931	201 188
13		**oct 19**	12 847	16 551	22 828	34 090	42 544	44 312	173 171
14		**nov 19**	14 058	21 313	24 860	46 960	52 012	55 056	214 259
15		**déc 19**	14 702	19 723	24 536	42 364	49 355	50 933	201 613
16		**Total**	171 130	241 363	283 185	519 755	586 229	622 859	2 424 521

Figure 9.4 : Tableau du rapport des ventes basé sur un modèle préformaté.

Sur quoi repose xlwings ?

xlwings dépend d'autres packages Python pour communiquer avec le mécanisme d'automatisation du système d'exploitation :

- *Windows*. Sous Windows, xlwings s'appuie sur la technologie COM, abréviation de Component Object Model. COM est une norme qui permet à deux processus de communiquer entre eux, dans notre cas Excel et Python. xlwings utilise le package Python pywin32[9] pour gérer les appels COM.
- *macOS*. Sous macOS, xlwings s'appuie sur AppleScript. AppleScript est le langage de script d'Apple pour automatiser les applications scriptables dont, heureusement pour nous, Excel fait partie. Pour exécuter des commandes AppleScript, xlwings utilise le package Python appscript[10].

Windows : comment empêcher les processus zombies

Lorsque vous exploitez xlwings sous Windows, vous remarquerez parfois qu'Excel semble être complètement fermé, mais lorsque vous ouvrez le Gestionnaire des tâches (cliquez avec le bouton droit sur la barre des tâches, puis sélectionnez Gestionnaire des tâches), vous verrez

9 https://github.com/mhammond/pywin32
10 https://github.com/hhas/appscript

Microsoft Excel sous Processus en arrière-plan dans l'onglet Processus. Si vous ne voyez aucun onglet, cliquez sur Plus de détails. Vous pouvez également accéder à l'onglet Détails, où vous verrez Excel listé sous la forme EXCEL.EXE. Pour mettre fin à un processus zombie, cliquez avec le bouton droit sur la ligne correspondante et sélectionnez Fin de tâche pour forcer Excel à se fermer.

Comme ces processus ne sont pas morts et pas correctement terminés, on les appelle souvent *processus zombies*. Le fait de les laisser actifs consomme des ressources et peut entraîner un comportement indésirable : par exemple, des fichiers peuvent être bloqués ou des compléments peuvent ne pas être correctement chargés lorsque vous ouvrez une nouvelle instance d'Excel. Excel ne parvient parfois pas à se fermer correctement parce que les processus ne peuvent être arrêtés qu'une fois qu'il n'y a plus de références COM, par exemple sous la forme d'un objet `app` xlwings. Le plus souvent, vous vous retrouvez avec un processus zombie Excel après avoir arrêté l'interpréteur Python, car cela l'empêche de nettoyer correctement les références COM. Prenons un exemple saisi dans Anaconda Prompt :

```
(base)> python
>>> import xlwings as xw
>>> app = xw.App()
```

Une fois la nouvelle instance d'Excel en cours d'exécution, quittez-la à nouveau *via* l'interface utilisateur d'Excel : pendant la fermeture d'Excel, le processus Excel dans le Gestionnaire des tâches continuera à s'exécuter. Si vous arrêtez correctement la session Python en exécutant `quit()` ou en utilisant le raccourci Ctrl + Z, le processus Excel sera finalement arrêté. Si, cependant, vous quittez Anaconda Prompt en cliquant sur le « x » en haut à droite de la fenêtre, vous remarquerez que le processus demeure en tant que processus zombie. La même chose se produit si vous arrêtez Anaconda Prompt avant de fermer Excel ou si vous l'arrêtez pendant qu'il exécute un serveur Jupyter et que vous conservez un objet `app` xlwings dans l'une des cellules du notebook Jupyter. Pour minimiser les risques de se retrouver avec des processus zombies Excel, voici quelques suggestions :

- Exécutez `app.quit()` à partir de Python au lieu de fermer Excel manuellement. Cela permet de s'assurer que les références sont nettoyées correctement.
- N'arrêtez pas les sessions Python interactives lorsque vous travaillez avec xlwings, par exemple, si vous exécutez un REPL Python dans Anaconda Prompt ; arrêtez correctement l'interpréteur Python en exécutant `quit()` ou en utilisant le raccourci Ctrl + Z. Lorsque vous travaillez avec des notebooks Jupyter, arrêtez le serveur en cliquant sur le bouton Quit dans l'interface Web.
- Avec les sessions Python interactives, il vaut mieux éviter d'utiliser directement l'objet `app`, par exemple en utilisant `xw.Book()` au lieu de `myapp.books.add()`. Cela devrait mettre fin correctement à Excel, même si le processus Python est arrêté.

Maintenant que vous avez une idée de la technologie sous-jacente de xlwings, voyons comment nous pouvons accélérer les scripts.

Amélioration des performances

Pour que vos scripts xlwings restent performants, il existe quelques stratégies : la plus importante est de réduire au minimum les appels interapplications. L'utilisation de valeurs brutes peut aussi être une possibilité, mais vous pouvez également définir les bonnes propriétés de l'objet `app`. Passons en revue ces options les unes après les autres.

Minimiser les appels interapplications

Il est essentiel de savoir que chaque appel interapplications de Python à Excel est « coûteux », c'est-à-dire lent. Par conséquent, ces appels doivent être réduits autant que possible. Le moyen le plus simple de le faire est de lire et d'écrire des plages Excel entières au lieu de parcourir des cellules individuelles dans une boucle. Dans l'exemple suivant, nous lisons et écrivons 150 cellules, d'abord en faisant une boucle sur chaque cellule, puis en traitant toute la plage en un seul appel :

```
In [64]: # Ajoute une nouvelle feuille et écrit 150 valeurs
         # pour avoir du grain à moudre
         sheet2 = book.sheets.add()
         sheet2["A1"].value = np.arange(150).reshape(30, 5)
In [65]: %%time
         # La boucle réalise 150 appels interapplications
         for cell in sheet2["A1:E30"]:
             cell.value += 1
Wall time: 909 ms
In [66]: %%time
         # Ici, il n'y a que deux appels interapplications
         values = sheet2["A1:E30"].options(np.array).value
         sheet2["A1"].value = values + 1
Wall time: 97.2 ms
```

Les résultats du temps d'exécution (*wall time*) sont encore plus extrêmes sous macOS, où la deuxième option est environ 50 fois plus rapide que la première sur ma machine.

Valeurs brutes

Le principal objectif de conception de xlwings a été de mettre l'accent sur la simplicité d'emploi plutôt que sur la vitesse. Cependant, si vous exploitez d'énormes plages de cellules, vous pouvez vous heurter à des situations où vous pouvez gagner du temps en sautant l'étape de nettoyage des données de xlwings : xlwings parcourt chaque valeur lorsque vous lisez et écrivez des données, par exemple, pour aligner les types de données entre Windows et macOS. En utilisant la chaîne `raw` comme convertisseur dans la méthode `options`, vous ignorez cette étape. Normalement, toutes les opérations devraient être plus rapides, mais la différence peut ne pas être significative,

à moins que vous n'écriviez de grands tableaux sous Windows. Toutefois, l'utilisation de valeurs brutes implique que vous ne pouvez plus travailler directement avec des DataFrames. Au lieu de cela, vous devez fournir vos valeurs sous forme de listes imbriquées ou de tuples. En outre, vous devrez fournir l'adresse complète de la plage dans laquelle vous écrivez, car l'indication de la cellule en haut à gauche ne suffit plus :

```
In [67]: # Avec des valeurs brutes, vous devez fournir la plage cible
         # complète, car sheet["A35"] ne fonctionne plus
         sheet1["A35:B36"].options("raw").value = [[1, 2], [3, 4]]
```

Propriétés de l'objet app

Selon le contenu de votre classeur, la modification des propriétés de vos objets `app` peut également aider à accélérer l'exécution du code. Habituellement, il est souhaitable de porter votre attention sur les propriétés suivantes (`myapp` est un objet `app` xlwings) :

- `myapp.screen_updating = False`
- `myapp.calculation = "manual"`
- `myapp.display_alerts = False`

À la fin du script, assurez-vous de remettre les attributs à leur état d'origine. Si vous utilisez Windows, vous pouvez également constater une légère amélioration des performances en exécutant votre script dans une instance d'Excel masquée *via* `xw.App(visible=False)`.

Maintenant que vous savez comment améliorer les performances, voyons comment étendre les fonctionnalités de xlwings.

Comment pallier les fonctionnalités manquantes

xlwings fournit une interface dans l'esprit de Python pour les commandes Excel les plus couramment utilisées et les fait fonctionner sous Windows et macOS. Il existe cependant de nombreuses méthodes et attributs du modèle objet d'Excel qui ne sont pas encore couverts nativement par xlwings, mais tout n'est pas perdu ! xlwings donne accès à l'objet pywin32 sous Windows et à l'objet appscript sous macOS en utilisant l'attribut `api` sur n'importe quel objet xlwings. De cette façon, vous avez accès à l'ensemble du modèle objet d'Excel, mais en contrepartie, vous perdez la compatibilité multiplateforme. Par exemple, supposons que vous souhaitiez effacer la mise en forme d'une cellule. Voici comment vous vous y prendrez :

- Vérifiez si la méthode est disponible sur l'objet `range` xlwings, par exemple en utilisant la touche Tab après avoir saisi un point à la fin d'un objet `range` dans un notebook Jupyter, en exécutant `dir(sheet["A1"])` ou en recherchant dans la documentation de l'API xlwings[11]. Dans VS Code, les méthodes disponibles doivent être affichées automatiquement dans une info-bulle.

11 https://docs.xlwings.org/en/stable/api.html

- Si la fonctionnalité souhaitée est manquante, utilisez l'attribut `api` pour obtenir l'objet sous-jacent : sous Windows, `sheet["A1"].api` vous donnera un objet pywin32 et sous macOS, vous obtiendrez un objet appscript.
- Consultez le modèle objet d'Excel dans la documentation d'Excel VBA[12]. Pour effacer le format d'une plage, il s'agit de la méthode Range.ClearFormats[13].
- Sous Windows, dans la plupart des cas, vous pouvez utiliser la méthode ou la propriété VBA directement avec votre objet `api`. S'il s'agit d'une méthode, assurez-vous d'ajouter des parenthèses dans Python : `sheet["A1"].api.ClearFormats()`. Si vous faites cela sous macOS, les choses sont plus compliquées car appscript utilise une syntaxe qui peut être difficile à deviner. Le plus simple est de consulter le guide du développeur qui fait partie du code source xlwings[14]. Effacer la mise en forme des cellules, cependant, est assez facile : il suffit d'appliquer les règles de syntaxe de Python sur le nom de la méthode en utilisant des caractères minuscules avec des barres de soulignement : `sheet["A1"].api.clear_formats()`.

Si vous devez vous assurer que `ClearFormats` fonctionne sur les deux plateformes, vous pouvez le faire de la manière suivante (`darwin`, qui est le cœur de macOS, est utilisé comme nom par `sys.platform`) :

```
import sys
if sys.platform.startswith("darwin"):
    sheet["A10"].api.clear_formats()
elif sys.platform.startswith("win"):
    sheet["A10"].api.ClearFormats()
```

Dans tous les cas, cela vaut la peine de poster une demande sur le dépôt GitHub de xlwings[15] pour que la fonctionnalité soit incluse dans une future version.

Conclusion

Ce chapitre vous a présenté le concept d'automatisation d'Excel : *via* xlwings, vous pouvez utiliser Python pour des tâches que vous feriez traditionnellement en VBA. Nous avons découvert le modèle objet d'Excel et comment xlwings permet d'interagir avec ses composants tels que les objets `sheet` et `range`. Forts de ces connaissances, nous avons repris notre étude de cas et avons utilisé xlwings pour remplir un modèle de rapport préformaté ; cela vous a montré qu'il est possible d'utiliser les packages de lecture et xlwings en parallèle. Nous avons également abordé les bibliothèques que xlwings utilise sous le capot pour comprendre comment nous pouvons améliorer les performances et pallier les fonctionnalités manquantes. Ce que je préfère dans

12 https://docs.microsoft.com/en-us/office/vba/api/overview/excel
13 https://docs.microsoft.com/en-us/office/vba/api/excel.range.clearformats
14 https://github.com/xlwings/xlwings/blob/master/DEVELOPER_GUIDE.md
15 https://github.com/xlwings/xlwings

xlwings, c'est qu'il fonctionne aussi bien sous macOS que sous Windows. C'est d'autant plus intéressant que Power Query sous macOS n'a pas encore toutes les fonctionnalités de la version Windows : vous devriez pouvoir remplacer facilement tout ce qui manque par une combinaison de pandas et de xlwings.

Maintenant que vous connaissez les bases de xlwings, vous êtes prêt pour le chapitre suivant où nous allons appeler des scripts xlwings à partir d'Excel, ce qui permettra de créer des outils Excel dont la puissance sera boostée par Python.

CHAPITRE 10

Outils Excel dopés par Python

Dans le dernier chapitre, nous avons appris à écrire des scripts Python pour automatiser Microsoft Excel. Cette solution est très puissante, mais elle nécessite que l'utilisateur se sente à l'aise avec Anaconda Prompt ou un éditeur comme VS Code pour exécuter les scripts. Ce n'est probablement pas le cas si vos outils sont utilisés par des utilisateurs professionnels qui ne sont pas informaticiens. Pour ces personnes, il est souhaitable de masquer la partie Python afin que l'outil Excel ressemble à nouveau à un classeur normal prenant en charge les macros. Ce chapitre traite de la manière de remplir cet objectif avec xlwings. Je commencerai par vous montrer le chemin le plus court pour exécuter du code Python à partir d'Excel avant d'examiner les défis du déploiement des outils xlwings, ce qui nous permettra également d'avoir un aperçu plus détaillé des paramètres de xlwings. Comme le chapitre précédent, celui-ci nécessite que Microsoft Excel soit installé sur votre ordinateur (Windows ou macOS).

Utilisation d'Excel comme frontend avec xlwings

Le *frontend* est la partie d'une application qu'un utilisateur voit et avec laquelle il interagit. Le frontend est également appelé *interface utilisateur graphique* (GUI en anglais, pour *graphical user interface*) ou tout simplement *interface utilisateur*. Lorsque je demande aux utilisateurs de xlwings pourquoi ils créent leurs outils avec Excel plutôt que de générer une application Web moderne, j'entends généralement la réponse suivante : « Excel est l'interface que nos utilisateurs connaissent ». S'appuyer sur les cellules d'une feuille de calcul permet aux utilisateurs de saisir rapidement et intuitivement des données, ce qui les rend souvent plus productifs que s'ils devaient utiliser une interface Web pas très bien finalisée. Je commencerai cette section en vous présentant le complément Excel xlwings et l'interface de ligne de commande de xlwings (que j'abrège par la suite en CLI pour *command line interface*) avant de créer notre premier projet *via* la commande `quickstart`. Je conclurai cette section en vous montrant deux façons d'appeler du code Python à partir d'Excel : en cliquant sur le bouton Run main du complément et en utilisant la fonction `RunPython` en VBA. Commençons par installer le complément Excel xlwings.

Complément Excel

Étant donné que xlwings est inclus dans la distribution Anaconda, nous avons pu, dans le chapitre précédent, exécuter des commandes xlwings en Python sans avoir à installer quoi que ce soit. Cependant, si vous souhaitez appeler des scripts Python à partir d'Excel, vous devez installer le complément Excel ou configurer le classeur en mode autonome. Je présenterai le mode autonome dans la section « Déploiement » de ce chapitre, et cette section montre comment travailler avec le complément. Pour l'installer, exécutez la commande suivante dans Anaconda prompt :

```
(base)> xlwings addin install
```

Vous devez maintenir la synchronisation entre la version du package Python et la version du complément chaque fois que vous mettez à jour xlwings. Par conséquent, vous devez toujours exécuter deux commandes lorsque vous mettez à jour xlwings : une commande pour le package Python et l'autre pour le complément Excel. Selon que vous utilisez le gestionnaire de packages Conda ou pip, voici comment mettre à jour votre installation xlwings :

Conda (à utiliser avec la distribution Python Anaconda)

```
(base)> conda update xlwings
(base)> xlwings addin install
```

pip (à utiliser avec n'importe quelle autre distribution Python)

```
(base)> pip install --upgrade xlwings
(base)> xlwings addin install
```



Malheureusement, le complément xlwings est parfois signalé comme un complément malveillant par les logiciels antivirus, surtout si vous utilisez une toute nouvelle version. Si cela se produit sur votre machine, accédez aux paramètres de votre logiciel antivirus, où vous devriez pouvoir indiquer que xlwings est une application sûre. Habituellement, il est également possible de signaler de tels faux positifs via la page d'accueil de l'antivirus.

Lorsque vous saisissez `xlwings` dans Anaconda Prompt, vous utilisez la CLI xlwings. Outre la simplification de l'installation du complément xlwings, la CLI offre quelques commandes supplémentaires que je présenterai chaque fois que nous en aurons besoin, mais vous pouvez toujours saisir xlwings dans Anaconda Prompt et appuyer sur Entrée pour afficher la liste des options disponibles. Examinons maintenant de plus près ce que fait la commande `xlwings addin install` :

Installation

L'installation proprement dite du complément se réalise en copiant *xlwings.xlam* à partir du répertoire du package Python dans le dossier *XLSTART* d'Excel, qui est un dossier spécial : Excel ouvre tous les fichiers qui se trouvent dans ce dossier chaque fois que vous démarrez Excel. Lorsque vous exécutez `xlwings addin status` dans Anaconda Prompt,

cela affiche la version du complément et l'emplacement du répertoire *XLSTART* sur votre système.

Configuration

Lorsque vous installez le complément la toute première fois, il est également configuré pour utiliser l'interpréteur Python ou l'environnement Conda à partir duquel vous exécutez la commande `install` : comme vous le voyez à la Figure 10.1, les valeurs de Conda Path et Conda Env sont renseignées automatiquement par la CLI xlwings[1]. Ces valeurs sont stockées dans un fichier nommé *xlwings.conf* dans le dossier *.xlwings* de votre répertoire personnel. Sous Windows, il s'agit généralement de *C:\Users\<nom_utilisateur>\.xlwings\xlwings.conf* et sous macOS de */Users/<nom_utilisateur>/.xlwings/xlwings.conf*. Sous macOS, les dossiers et les fichiers avec un point en en-tête sont masqués par défaut. Lorsque vous êtes dans le Finder, saisissez le raccourci clavier fn + Commande + Maj + . (point) pour afficher ou masquer les fichiers.

Après avoir exécuté la commande d'installation, vous devez redémarrer Excel pour voir l'onglet xlwings dans le ruban qui est illustré à la Figure 10.1.

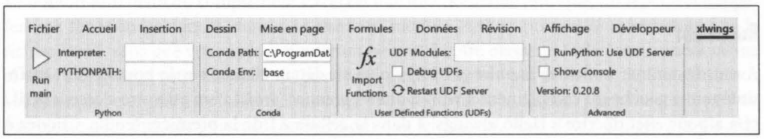


Figure 10.1 : Le complément xlwings affiché dans le ruban après l'exécution de la commande install.



Sous macOS, le ruban est un peu différent, car il manque les sections des fonctions définies par l'utilisateur et Conda : alors que les fonctions définies par l'utilisateur ne sont pas prises en charge sous macOS, les environnements Conda ne nécessitent pas de traitement spécial, car ils sont configurés en tant qu'interpréteur dans le groupe Python.

Maintenant que vous avez installé le complément xlwings, nous aurons besoin d'un classeur et de code Python pour le tester. Le moyen le plus rapide de réaliser cela est d'utiliser la commande `quickstart`.

1 Si vous êtes sous macOS ou si vous utilisez une distribution Python autre qu'Anaconda, le complément configure l'interpréteur au lieu des paramètres Conda.

Commande quickstart

Pour rendre la création de votre premier outil xlwings aussi facile que possible, la CLI xlwings propose la commande `quickstart`. Dans Anaconda Prompt, utilisez la commande `cd` pour vous positionner dans le répertoire où vous souhaitez créer votre premier projet (par exemple, `cd test`), puis exécutez la commande suivante pour créer un projet portant le nom `premier_projet` :

```
(base)> xlwings quickstart premier_projet
```

Le nom du projet doit être un nom de module Python valide : il peut contenir des caractères, des nombres et des barres de soulignement, mais pas d'espaces ou de tirets, et il ne doit pas commencer par un nombre. Je vous montrerai dans la section Fonction RunPython comment changer le nom du fichier Excel en s'affranchissant de ces règles. L'exécution de la commande `quickstart` crée un dossier appelé *premier_projet* dans votre répertoire actuel. Lorsque vous l'ouvrez dans l'Explorateur de fichiers sous Windows ou le Finder sous macOS, vous verrez deux fichiers : *premier_projet.xlsm* et *premier_projet.py*. Ouvrez les deux fichiers : le fichier Excel dans Excel et le fichier Python dans VS Code. Le moyen le plus simple d'exécuter le code Python à partir d'Excel consiste à utiliser le bouton Run main du complément.

Run main

Avant d'examiner le fichier *premier_projet.py* plus en détail, cliquez sur le bouton Run main, situé tout à gauche du complément xlwings, quand *premier_projet.xlsm* est votre classeur actif ; cela a pour effet d'écrire « Hello xlwings! » dans la cellule A1 de la première feuille. Cliquez à nouveau sur le bouton et le contenu de la cellule sera remplacé par « Bye xlwings! ». Félicitations, vous venez d'exécuter votre première fonction Python à partir d'Excel ! Reconnaissez que ce n'était pas beaucoup plus difficile que d'écrire une macro VBA ! Jetons maintenant un coup d'œil à *premier_projet.py* qui est reproduit dans l'Exemple 10.1 :

Exemple 10.1 : premier_projet.py.

```
import xlwings as xw


def main():
    wb = xw.Book.caller() ❶
    sheet = wb.sheets[0]
    if sheet["A1"].value == "Hello xlwings!":
        sheet["A1"].value = "Bye xlwings!"
    else:
        sheet["A1"].value = "Hello xlwings!"


@xw.func ❷
def hello(name):
    return f"Hello {name}!"
```

```
if __name__ == "__main__": ❸
    xw.Book("premier_projet.xlsm").set_mock_caller()
    main()
```

❶ `xw.Book.caller()` est un objet xlwings book qui fait référence au classeur Excel actif lorsque vous cliquez sur le bouton Run main. Dans notre cas, cela correspond à `xw.Book("premier_projet.xlsm")`. L'emploi de `xw.Book.caller()` permet de renommer et de déplacer votre fichier Excel sur le système de fichiers sans rompre la référence. Cela garantit également que vous manipulez le bon classeur si vous l'avez ouvert dans plusieurs instances d'Excel.

❷ Dans ce chapitre, nous allons ignorer la fonction `hello` car ce sera le sujet du Chapitre 12. Si vous exécutez la commande `quickstart` sous macOS, vous ne verrez de toute façon pas la fonction `hello`, car les fonctions définies par l'utilisateur ne sont prises en charge que sous Windows.

❸ J'expliquerai les trois dernières lignes lorsque nous aborderons le débogage dans le chapitre suivant. Pour les besoins de ce chapitre, vous pouvez ignorer, voire supprimer, tout ce qui se trouve sous la première fonction.

Le bouton Run main du complément Excel est une fonctionnalité pratique, car il permet d'appeler une fonction portant le nom `main` dans un module Python portant le même nom que le fichier Excel, sans que vous ayez à ajouter un bouton à votre classeur. Cela fonctionnera même si vous enregistrez votre classeur au format *xlsx* qui ne contient pas de macro. Si, toutefois, vous souhaitez appeler une ou plusieurs fonctions Python qui ne se nomment pas `main` et qui ne font pas partie d'un module portant le même nom que le classeur, vous devez utiliser à la place la fonction RunPython en VBA.

Fonction RunPython

Si vous avez besoin de plus de contrôle sur la façon dont votre code Python est appelé, utilisez la fonction VBA `RunPython`. Par conséquent, RunPython nécessite que votre classeur soit enregistré en tant que classeur prenant en charge les macros.



Vous devez cliquer sur Activer le contenu (Windows) ou Activer les macros (macOS) lorsque vous ouvrez un classeur prenant en charge les macros (extension xlsm) tel que celui généré par la commande `quickstart`. Sous Windows, lorsque vous travaillez avec des fichiers xlsm à partir des exemples de code, vous devez également cliquer sur Activer la modification, sinon Excel n'ouvrira pas correctement les fichiers téléchargés à partir d'Internet.

`RunPython` accepte une chaîne avec du code Python : le plus souvent, vous importez un module Python et exécutez l'une de ses fonctions. Lorsque vous ouvrez l'éditeur VBA *via* Alt + F11 (Windows) ou Option + F11 (macOS), vous verrez que la commande `quickstart` ajoute une macro appelée `SampleCall` dans un module VBA portant le nom « Module1 » (voir la Figure 10.2). Si

vous ne voyez pas SampleCall, faites un double-clic sur Module1 dans l'arborescence du projet VBA sur le côté gauche.

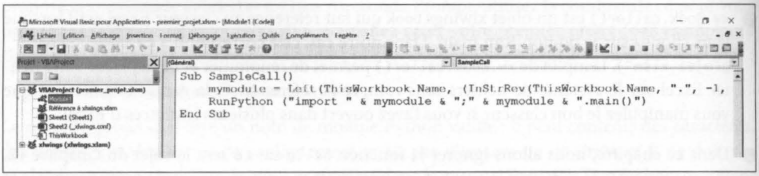


Figure 10.2 : L'éditeur VBA affichant Module1.

Le code semble un peu alambiqué, mais il s'agit uniquement de le faire fonctionner dynamiquement, quel que soit le nom du projet que vous choisissez lors de l'exécution de la commande `quickstart`. Comme notre module Python s'appelle `premier_projet`, vous pouvez remplacer le code par quelque chose d'équivalent qui soit plus facile à comprendre :

```
Sub SampleCall()
    RunPython "import premier_projet ; premier_projet.main()"
End Sub
```

Comme il n'est pas évident d'écrire des chaînes de caractères sur plusieurs lignes en VBA, nous utilisons un point-virgule que Python accepte à la place d'un saut de ligne. Il existe plusieurs façons d'exécuter ce code : par exemple, lorsque vous êtes dans l'éditeur VBA, placez votre curseur sur n'importe quelle ligne de la macro SampleCall et appuyez sur la touche F5. Cependant, on exécute en général le code à partir d'une feuille Excel et non pas à partir de l'éditeur VBA. Par conséquent, fermez l'éditeur VBA et revenez au classeur. En saisissant Alt + F8 (Windows) ou Option + F8 (macOS), le menu Macro apparaît : sélectionnez `SampleCall` et cliquez sur le bouton Exécuter. Pour rendre cela plus convivial, vous pouvez aussi ajouter un bouton à votre classeur Excel afin de le connecter à la macro `SampleCall` : tout d'abord, assurez-vous que l'onglet Développeur du ruban est affiché. Si ce n'est pas le cas, accédez à Fichiers > Options > Personnaliser le ruban et activez la case à cocher en regard de Développeur (sous macOS, vous trouverez cette option sous Excel > Préférences > Ruban et barre d'outils). Pour insérer un bouton, allez dans l'onglet Développeur et dans le groupe Contrôles, cliquez sur Insérer > Bouton (sous Contrôles de formulaire). Sous macOS, le bouton s'affichera sans avoir à aller au préalable dans le menu Insérer. Lorsque vous cliquez sur l'icône du bouton, votre curseur se transforme en une petite croix : utilisez le curseur pour dessiner un bouton sur votre feuille en maintenant le bouton gauche de la souris enfoncé et en dessinant une forme rectangulaire. Une fois que vous avez relâché le bouton de votre souris, le menu Affecter une macro s'affiche : sélectionnez `SampleCall` et cliquez sur OK. Cliquez sur le bouton que vous venez de créer (dans mon cas, il s'appelle « Bouton 1 »), et cela exécutera à nouveau la fonction `main` (voir la Figure 10.3).

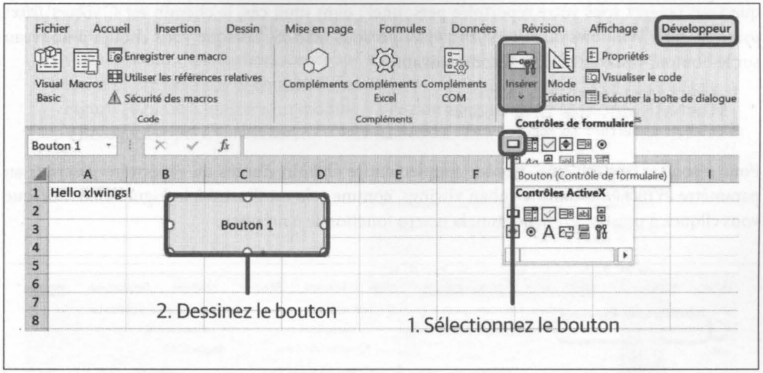


Figure 10.3 : Dessin d'un bouton sur une feuille de calcul.



Sous Windows, vous disposez de deux types de contrôles : les contrôles de formulaire et les contrôles ActiveX. Bien que vous puissiez utiliser les deux types de boutons pour connecter votre macro `SampleCall`, les contrôles ActiveX ne fonctionnent pas sous macOS. Dans le chapitre suivant, nous utiliserons des rectangles comme boutons pour les rendre un peu plus modernes.

Voyons maintenant comment nous pouvons modifier les noms par défaut attribués par la commande `quickstart` : revenez à votre fichier Python *premier_projet.py* et renommez-le *hello.py*. Renommez également votre fonction `main` en `hello_world`. Assurez-vous d'enregistrer le fichier, puis ouvrez à nouveau l'éditeur VBA *via* Alt + F11 (Windows) ou Option + F11 (macOS) et modifiez `SampleCall` pour refléter les modifications :

```
Sub SampleCall()
    RunPython "import hello; hello.hello_world()"
End Sub
```

De retour sur la feuille, cliquez sur le « Bouton 1 » pour vous assurer que tout fonctionne correctement. Enfin, vous pouvez également stocker le script Python et le fichier Excel dans deux répertoires différents. Pour que vous compreniez les implications d'une telle modification, je vais d'abord devoir dire un mot sur le *chemin de recherche des modules* de Python : si vous importez un module dans votre code, Python le recherche dans différents répertoires. Tout d'abord, Python vérifie s'il existe un module intégré portant ce nom, et s'il n'en trouve pas, il effectue la recherche dans le répertoire de travail en cours et dans les répertoires indiqués par ce que l'on appelle `PYTHONPATH`. xlwings ajoute automatiquement le répertoire du classeur à `PYTHONPATH` et permet d'ajouter des chemins supplémentaires *via* le complément. Pour tester cela, prenez le script Python qui s'appelle à présent *hello.py* et déplacez-le vers un dossier appelé *pyscripts*

que vous créerez sous votre répertoire personnel : dans mon cas, le chemin est *C:\Users\felix\pyscripts* sous Windows ou */Users/felix/pyscripts* sous macOS. Lorsque vous cliquez à nouveau sur le bouton, vous obtiendrez l'erreur suivante :

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'premier_projet'
```

Pour résoudre ce problème, ajoutez simplement le chemin d'accès du répertoire *pyscripts* au paramètre `PYTHONPATH` dans le ruban xlwings, comme cela est illustré à la Figure 10.4. Lorsque vous cliquez à présent sur le bouton, la macro fonctionne à nouveau.

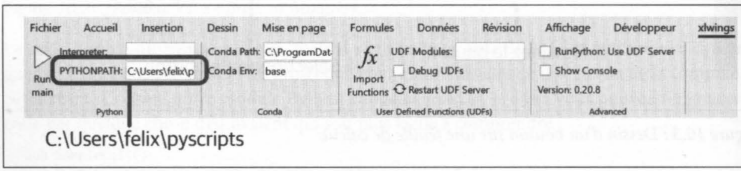


Figure 10.4 : Modification du paramètre PYTHONPATH.

Je n'ai pas encore abordé la question du nom du classeur Excel : une fois que votre appel de fonction `RunPython` utilise un nom de module explicite comme `premier_projet` au lieu du code qui a été ajouté par `quickstart`, vous êtes libre de renommer votre classeur Excel comme vous le souhaitez.

S'appuyer sur la commande `quickstart` est le moyen le plus simple de démarrer un nouveau projet xlwings. Toutefois, si vous disposez d'un classeur existant, vous préférerez peut-être le configurer manuellement.

RunPython sans commande quickstart

Si vous souhaitez utiliser la fonction `RunPython` avec un classeur existant qui n'a pas été créé par la commande `quickstart`, vous devez gérer manuellement des choses que la commande `quickstart` fait à votre place. Notez que les étapes suivantes ne sont requises que pour l'appel à `RunPython`, mais pas lorsque vous souhaitez utiliser le bouton Run main :

1. Tout d'abord, assurez-vous d'enregistrer votre classeur en tant que classeur prenant en charge les macros avec l'extension *xlsm* ou *xlsb*.

2. Ajouter un module VBA ; pour ce faire, ouvrez l'éditeur VBA *via* Alt + F11 (Windows) ou Option + F11 (macOS) et assurez-vous de sélectionner le projet VBAProject de votre classeur dans l'arborescence sur le côté gauche, puis cliquez dessus avec le bouton droit de la souris et choisissez Insertion > Module, comme cela est illustré à la Figure 10.5. Cela insérera un module VBA vide où vous pourrez écrire votre macro VBA avec l'appel à `RunPython`.

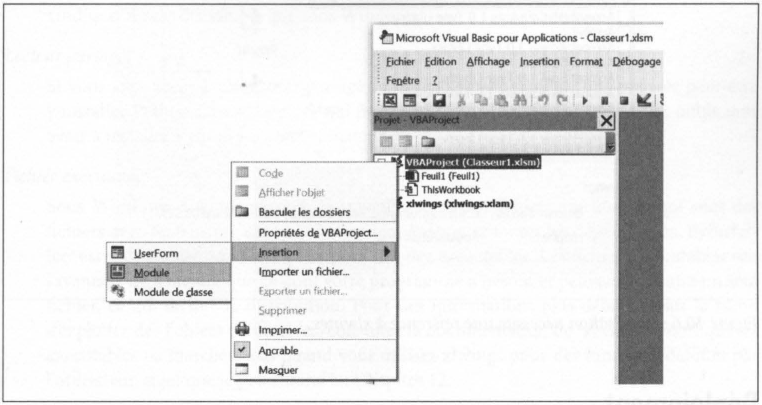


Figure 10.5 : Ajout d'un module VBA.

3. Ajoutez une référence à xlwings : `RunPython` est une fonction qui fait partie du complément xlwings. Pour l'utiliser, vous devez vous assurer que vous avez défini une référence à xlwings dans votre projet VBA. Encore une fois, commencez par sélectionner le bon classeur dans l'arborescence sur le côté gauche de l'éditeur VBA, puis accédez à Outils > Références et activez la case à cocher pour xlwings, comme cela est indiqué à la Figure 10.6.

Votre classeur est maintenant prêt à être utilisé à nouveau avec un appel à `RunPython`. Une fois que tout fonctionne sur votre machine, l'étape suivante consiste généralement à faire fonctionner la macro sur la machine de votre collègue ; nous allons passer en revue quelques options pour faciliter cette opération.

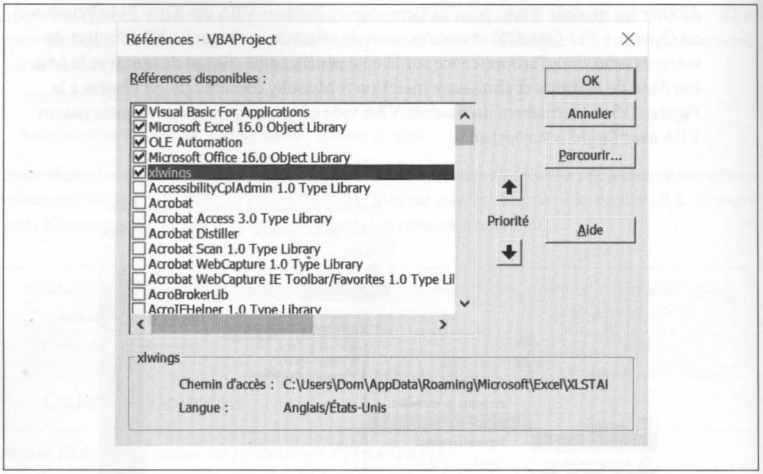


Figure 10.6 : RunPython nécessite une référence à xlwings.

Déploiement

Dans le développement de logiciels, le terme *déploiement* fait référence à la distribution et à l'installation de programmes afin que les utilisateurs finaux puissent les utiliser. Dans le cas des outils xlwings, il est utile de savoir quelles dépendances sont requises et quels paramètres peuvent faciliter le déploiement. Je commencerai par la dépendance la plus importante, Python, avant de me pencher sur les classeurs qui ont été configurés en mode autonome afin de se débarrasser du complément Excel xlwings. Je conclurai cette section en examinant en détail le fonctionnement de la configuration de xlwings.

Dépendance Python

Pour pouvoir exécuter les outils xlwings, vos utilisateurs finaux doivent disposer d'une installation de Python. Mais ce n'est pas parce qu'ils n'ont pas encore Python qu'il est impossible de faciliter le processus d'installation. Voici quelques options :

Anaconda ou WinPython

Demandez à vos utilisateurs de télécharger et d'installer la distribution Anaconda. Par mesure de sécurité, vous devez vous mettre d'accord sur une version spécifique d'Anaconda afin de vous assurer qu'ils utilisent les mêmes versions des packages que vous employez. C'est une bonne option si vous n'utilisez que des packages qui font partie d'Anaconda. WinPython[2] est une alternative intéressante à Anaconda, car il est distribué sous la licence open source du MIT et il est également livré avec xlwings préinstallé. Comme son nom l'indique, il n'est disponible que sous Windows.

Lecteur partagé

Si vous avez accès à un lecteur partagé raisonnablement rapide, vous pourrez peut-être y installer Python directement, ce qui permettra à tout le monde d'utiliser les outils sans avoir à installer Python sur son ordinateur.

Fichier exécutable

Sous Windows, xlwings permet de travailler avec des *fichiers exécutables*, qui sont des fichiers avec l'extension *.exe* qui contiennent Python et toutes les dépendances. PyInstaller[3] est un package populaire pour produire des exécutables. Les fichiers exécutables ont l'avantage de n'inclure que ce dont votre programme a besoin et peuvent produire un seul fichier, ce qui facilite la distribution. Pour des informations plus détaillées sur la façon d'exploiter des fichiers exécutables, consultez la documentation de [4]xlwings. Notez que les exécutables ne marchent pas quand vous utilisez xlwings pour des fonctions définies par l'utilisateur, sujet que je présenterai au Chapitre 12.

Bien que l'installation de Python soit une condition *sine qua non*, ce n'est pas le cas du complément xlwings.

Classeurs autonomes : comment se débarrasser du complément xlwings

Dans ce chapitre, nous nous sommes toujours appuyés sur le complément xlwings pour appeler le code Python en cliquant sur le bouton Run main ou en utilisant la fonction `RunPython`. Même si la CLI xlwings facilite l'installation du complément, cela peut toujours être un problème pour les utilisateurs goûtant peu la technique et qui ne sont pas à l'aise à l'idée d'utiliser Anaconda Prompt. En outre, étant donné que le complément xlwings et le package Python xlwings doivent avoir la même version, il y a un risque de conflit si vos utilisateurs ont déjà installé le complément xlwings, mais avec une version différente de celle dont votre outil a besoin. Il existe cependant une solution simple : xlwings ne nécessite pas que le complément Excel soit installé et peut être configuré en tant que *classeur autonome*. Dans ce cas, le code VBA du complément est stocké

2 https://winpython.github.io/
3 https://www.pyinstaller.org/
4 https://docs.xlwings.org/en/stable/deployment.html#runfrozenpython

directement dans votre classeur. Comme d'habitude, le moyen le plus simple de configurer le tout consiste à utiliser la commande `quickstart`, cette fois-ci avec l'indicateur `--standalone` :

```
(base)> xlwings quickstart second_projet --standalone
```

Lorsque vous ouvrez le classeur *second_projet.xlsm* généré dans Excel et appuyez sur Alt + F11 (Windows) ou Option + F11 (macOS), vous verrez le module `xlwings` et le module de classe `Dictionary` qui remplacent le complément. Plus important encore, un projet autonome ne doit plus avoir de référence à xlwings. Bien que cela soit configuré automatiquement avec l'emploi de l'indicateur `--standalone`, il est important que vous supprimiez la référence au cas où vous souhaiteriez convertir un classeur existant : accédez à Outils > Références dans votre éditeur VBA et désactivez la case à cocher `xlwings`.



Bien que cette section vous montre comment vous débarrasser de la dépendance du complément xlwings, vous pouvez parfois créer votre propre complément lors d'un déploiement. Cela a du sens si vous souhaitez utiliser les mêmes macros avec de nombreux classeurs différents. Vous trouverez des instructions sur la façon de créer votre propre complément personnalisé dans la documentation xlwings[5].

Après avoir abordé Python et le complément, examinons maintenant plus en profondeur le fonctionnement de la configuration de xlwings.

Hiérarchie de configuration

Comme nous l'avons mentionné au début de ce chapitre, le ruban stocke sa configuration dans le répertoire personnel de l'utilisateur, sous *.xlwings\xlwings.conf*. La configuration se compose de paramètres individuels, comme `PYTHONPATH` que nous avons déjà vu au début de ce chapitre. Les paramètres que vous définissez dans le complément peuvent être remplacés au niveau du répertoire et du classeur, xlwings recherchant les paramètres dans les emplacements suivants et dans l'ordre suivant :

Configuration du classeur

Tout d'abord, xlwings recherche une feuille appelée *xlwings.conf*. Il s'agit de la méthode recommandée pour configurer votre classeur pour le déploiement, car vous n'avez pas à gérer un fichier de configuration supplémentaire. Lorsque vous exécutez la commande `quickstart`, elle crée un exemple de configuration sur une feuille appelée « _xlwings.conf » : supprimez la barre de soulignement dans le nom pour l'activer. Si vous ne souhaitez pas l'utiliser, n'hésitez pas à supprimer la feuille.

Configuration du répertoire

Ensuite, xlwings recherche un fichier appelé *xlwings.conf* dans le même répertoire que votre classeur Excel.

5 https://docs.xlwings.org/en/stable/customaddin.html

Configuration de l'utilisateur

Enfin, xlwings recherche un fichier appelé *xlwings.conf* dans le dossier *.xlwings* du répertoire personnel de l'utilisateur. Normalement, vous ne devez pas modifier ce fichier directement, mais il est créé et modifié par le complément chaque fois que vous changez un paramètre.

Si xlwings ne trouve aucun paramètre dans ces trois emplacements, il revient aux valeurs par défaut.

Lorsque vous modifiez les paramètres *via* le complément Excel, ce dernier modifie automatiquement le fichier *xlwings.conf*. Si vous souhaitez modifier le fichier directement, recherchez le format exact et les paramètres disponibles en consultant la documentation xlwings[6], mais je vais vous indiquer les paramètres les plus utiles dans le contexte d'un déploiement.

Paramètres

Le paramètre le plus critique est certainement l'interpréteur Python : si votre outil Excel ne trouve pas le bon interpréteur Python, rien ne fonctionnera. Le paramètre `PYTHONPATH` permet de contrôler l'emplacement de vos fichiers source Python, et le paramètre Use UDF Server permet à l'interpréteur Python de s'exécuter entre les appels à Windows, ce qui peut améliorer considérablement les performances.

Interpréteur Python

xlwings s'appuie sur une installation locale de Python. Cependant, cela ne signifie pas nécessairement que le destinataire de votre outil xlwings doive jouer avec la configuration avant de pouvoir l'employer. Comme nous l'avons mentionné précédemment, vous pouvez indiquer à vos utilisateurs d'installer la distribution Anaconda avec les paramètres par défaut, ce qui l'installera dans le répertoire personnel de l'utilisateur. Si vous utilisez des *variables d'environnement* dans votre configuration, xlwings trouvera le chemin d'accès correct à l'interpréteur Python. Une variable d'environnement est une variable définie sur l'ordinateur de l'utilisateur qui permet aux programmes de collecter des informations spécifiques à cet environnement, comme le nom du dossier personnel de l'utilisateur actuel. Par exemple, sous Windows, vous pouvez définir `Conda Path` à `%USERPROFILE%\anaconda3` et sous macOS, `Interpreter_Mac` à `$HOME/opt/anaconda3/bin/python`. Ces chemins seront ensuite résolus dynamiquement en chemin d'installation par défaut d'Anaconda.

PYTHONPATH

Par défaut, xlwings recherche le fichier source Python dans le même répertoire que le fichier Excel. Cela peut ne pas être idéal lorsque vous diffusez votre outil à des utilisateurs qui ne sont pas familiers avec Python, car ils pourraient oublier de conserver les deux fichiers ensemble lors du déplacement du fichier Excel. Au lieu de cela, vous pouvez

6 https://docs.xlwings.org/en/stable/addin.html

placer vos fichiers source Python dans un dossier dédié (qui peut être situé sur un lecteur partagé) et ajouter ce dossier au paramètre `PYTHONPATH`. Vous pouvez également placer vos fichiers source dans un chemin qui fait déjà partie du chemin de recherche des modules Python. Vous pouvez réaliser cela en distribuant votre code source en tant que package Python (en l'installant, cela placera le package dans le répertoire des packages de Python, où Python trouvera votre code). Pour plus d'informations sur la création d'un package Python, consultez le tutoriel sur le packaging des projets Python[7].

RunPython : Use UDF Server (Windows uniquement)

Vous avez peut-être remarqué qu'un appel `RunPython` peut être relativement lent. En effet, xlwings démarre un interpréteur Python, exécute le code Python et arrête l'interpréteur. Cela n'est pas dramatique pendant le développement, car cela permet de s'assurer que tous les modules sont chargés à partir de zéro chaque fois que vous appelez la commande `RunPython`. Une fois que votre code est stable, cependant, vous pouvez activer la case à cocher « RunPython: Use UDF Server » qui n'est disponible que sous Windows. Cela utilisera le même serveur Python que les fonctions définies par l'utilisateur (le sujet du Chapitre 12) et maintiendra la session Python en cours d'exécution entre les appels, ce qui sera beaucoup plus rapide. Notez toutefois que vous devez cliquer sur le bouton Restart UDF Server du ruban après les modifications de code.

xlwings PRO

Bien que ce livre n'utilise que la version gratuite et open source de xlwings, il est également commercialisé un package professionnel afin de financer la maintenance et le développement continus du package open source. Parmi les fonctionnalités supplémentaires offertes par xlwings PRO, on peut citer :

- Le code Python peut être incorporé dans Excel, ce qui permet de se débarrasser des fichiers source externes.
- Le package de rapports permet de transformer vos classeurs en modèles avec des espaces réservés. Cela donne aux utilisateurs néophytes la faculté de modifier le modèle sans avoir à éditer le code Python.
- Les installateurs peuvent être générés facilement afin d'éliminer le casse-tête que constitue le déploiement : les utilisateurs finaux peuvent installer Python, y compris toutes les dépendances, en un seul clic, ce qui leur donne le sentiment d'exploiter des classeurs Excel normaux, sans avoir à configurer manuellement quoi que ce soit.

Pour plus de détails sur xlwings PRO et pour demander une version d'essai, consultez la page d'accueil de xlwings[8].

7 https://packaging.python.org/tutorials/packaging-projects/
8 https://www.xlwings.org/pro

Conclusion

Ce chapitre a commencé par vous montrer à quel point il est facile d'exécuter du code Python à partir d'Excel : une fois Anaconda installé, il vous suffit d'exécuter `xlwings addin install` suivi de `xlwings quickstart mon_projet`, et vous êtes prêt à cliquer sur le bouton Run main du complément xlwings ou à utiliser la fonction `RunPython` en VBA. La deuxième partie a introduit quelques paramètres qui facilitent le déploiement de votre outil xlwings auprès de vos utilisateurs finaux. Le fait que xlwings soit préinstallé avec Anaconda aide beaucoup les nouveaux utilisateurs à profiter de cet environnement.

Dans ce chapitre, nous nous sommes contentés du simple exemple Hello World pour apprendre comment tout fonctionne. Dans le chapitre suivant, nous allons repartir de ces bases pour créer une application professionnelle à part entière, le Python Package Tracker.

CHAPITRE 11

Python Package Tracker

Dans ce chapitre, nous allons créer une application métier typique qui télécharge des données à partir d'Internet et les stocke dans une base de données avant de les visualiser dans Excel. Cela vous aidera à comprendre quel rôle joue xlwings dans une telle application et vous permettra de voir à quel point il est facile de se connecter à des systèmes externes grâce à Python. Voulant construire un projet qui soit proche d'une application du monde réel, mais relativement simple à réaliser, j'ai créé *Python Package Tracker* (traqueur de packages Python), un outil Excel qui affiche le nombre de versions par an pour un package Python donné. Bien qu'il s'agisse d'une étude de cas, cet outil peut se révéler utile pour déterminer si un package Python est activement développé.

Après avoir fait connaissance avec l'application, nous passerons en revue quelques sujets que nous devons comprendre pour pouvoir analyser son code : nous verrons comment télécharger des données à partir d'Internet et interagir avec les bases de données avant d'en apprendre davantage sur la gestion des exceptions en Python, un concept important en matière de développement d'applications. Une fois que nous aurons terminé ces préliminaires, nous passerons en revue les composants de Python Package Tracker pour voir comment tout s'imbrique. Pour conclure ce chapitre, nous examinerons le fonctionnement du débogage du code xlwings. Comme les deux chapitres précédents, celui-ci nécessite que Microsoft Excel soit installé sur votre ordinateur. Commençons par tester Python Package Tracker !

Ce que nous allons construire

Dans les exemples de code, vous trouverez le dossier *packagetracker*. Il y a quelques fichiers dans ce dossier, mais pour l'instant, ouvrez simplement le fichier Excel *packagetracker.xlsm* et allez dans la feuille Database : nous devons d'abord alimenter la base de données pour avoir du grain à moudre. Comme cela est illustré à la Figure 11.1, saisissez un nom de package, par exemple

« xlwings », puis cliquez sur le bouton Add Package. Vous pouvez choisir n'importe quel nom de package existant dans l'index des packages Python[1] (PyPI).



Lorsque vous ajoutez le tout premier package sous macOS, vous devez donner votre autorisation afin que l'application puisse accéder au dossier packagetracker. Il s'agit de la même fenêtre que nous avons déjà rencontrée au Chapitre 9.

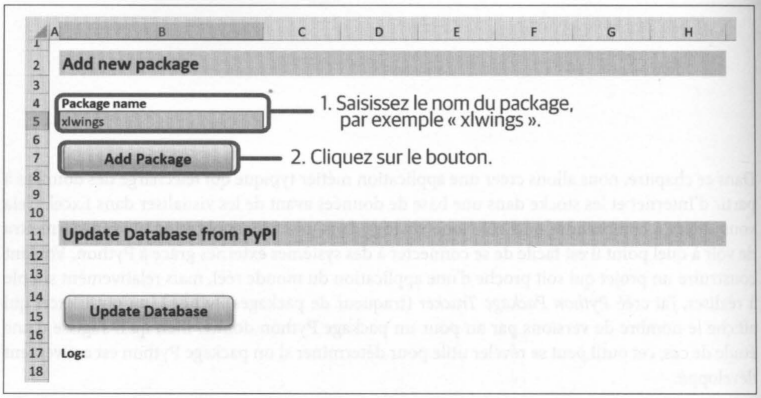


Figure 11.1 : Interface de Python Package Tracker (feuille Database).

Si tout fonctionne comme prévu, vous verrez le message « Added xlwings successfully » (xlwings ajouté avec succès) à droite de la cellule où vous avez saisi le nom du package. En outre, vous verrez un horodatage sous la section Update Database from PyPI indiquant la dernière mise à jour ainsi qu'une section Log (journal) où il est indiqué que xlwings a été téléchargé avec succès et qu'il est stocké dans la base de données. Afin d'avoir plus de données, ajoutons à présent le package pandas. Passez à présent à la feuille Tracker et sélectionnez xlwings dans la liste déroulante de la cellule B5 avant de cliquer sur le bouton Show History (afficher l'historique). Votre écran devrait maintenant ressembler au résultat illustré à la Figure 11.2 qui affiche la dernière version du package ainsi qu'un graphique avec le nombre de versions par année.

1 https://pypi.org/

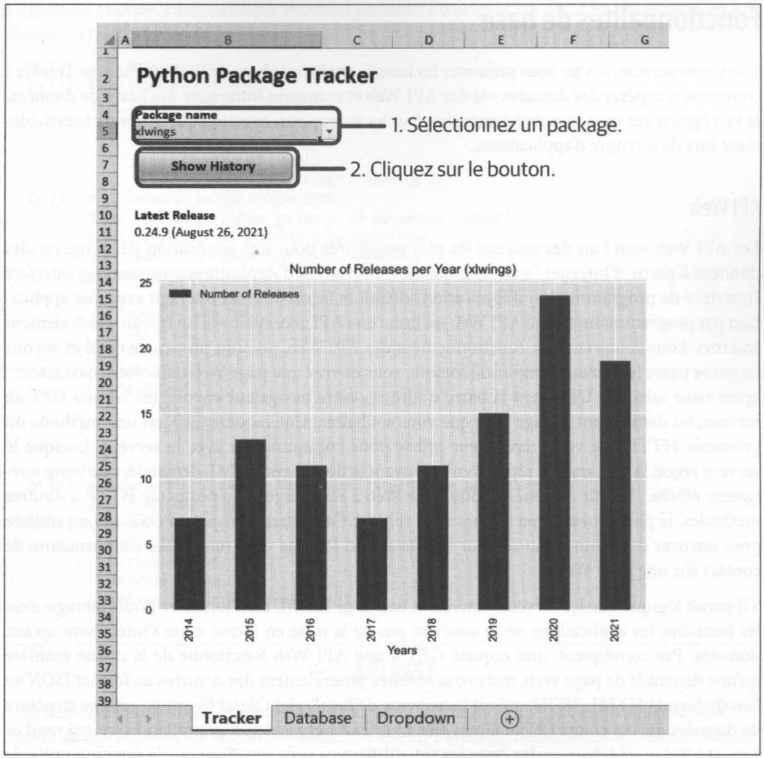


Figure 11.2 : Python Package Tracker (feuille Tracker).

Vous pouvez maintenant revenir à la feuille Database et ajouter d'autres packages. Chaque fois que vous souhaitez mettre à jour la base de données avec les dernières informations de PyPI, cliquez sur le bouton Update Database, ce qui synchronisera votre base de données avec les dernières données de PyPI.

Après avoir examiné le fonctionnement de Python Package Tracker du point de vue de l'utilisateur, nous allons maintenant présenter ses fonctionnalités de base.

Fonctionnalités de base

Dans cette section, je vais vous présenter les fonctionnalités de base de Python Package Tracker : comment récupérer des données *via* des API Web et comment interroger des bases de données. Je vais également vous montrer comment gérer les exceptions, question qui se posera inévitablement lors de l'écriture d'applications.

API Web

Les API Web sont l'un des moyens les plus populaires pour une application de récupérer des données à partir d'Internet : une API, qui est l'abréviation d'*Application programming interface* (interface de programmation d'application) définit la façon dont on interagit avec une application par programmation. Une API Web est donc une API accessible *via* un réseau, généralement Internet. Pour comprendre le fonctionnement des API Web, prenons un peu de recul et voyons ce qui se passe (en termes simplifiés) lorsque vous ouvrez une page Web dans votre navigateur : après avoir saisi une URL dans la barre d'adresse, votre navigateur envoie une *requête GET* au serveur, lui demandant la page Web que vous souhaitez. Une requête GET est une méthode du protocole HTTP que votre navigateur utilise pour communiquer avec le serveur. Lorsque le serveur reçoit la demande, il répond en renvoyant le document HTML demandé, que votre navigateur affiche : le tour est joué et votre page Web a été chargée ! Le protocole HTTP a d'autres méthodes, la plus courante, en dehors de la requête GET, étant la *requête POST*, qui est utilisée pour envoyer des données au serveur (par exemple, lorsque vous remplissez un formulaire de contact sur une page Web).

S'il paraît logique que les serveurs renvoient une page HTML bien formatée pour interagir avec les humains, les applications ne se soucient pas de la mise en forme et ne s'intéressent qu'aux données. Par conséquent, une requête GET à une API Web fonctionne de la même manière qu'une demande de page Web, mais vous obtenez généralement des données au format *JSON* au lieu du format HTML. JSON, qui est l'acronyme de *JavaScript Object Notation*, est une structure de données qui est comprise par à peu près tous les langages de programmation, ce qui rend ce format idéal pour échanger des données entre différents systèmes. Bien que la notation utilise la syntaxe de JavaScript, elle est très proche de la façon dont vous vous servez des dictionnaires et des listes (imbriqués) en Python. Voici la liste des différences entre JSON et Python :

- JSON n'accepte que les guillemets doubles pour les chaînes.
- JSON utilise `null` là où Python utilise `None`.
- JSON écrit `true` et `false` en minuscules alors qu'il y a une majuscule à l'initiale en Python.
- JSON n'accepte que les chaînes en tant que clés de dictionnaires, alors que les dictionnaires Python acceptent un large éventail d'objets en tant que clés.

Le module `json` de la bibliothèque standard permet de convertir un dictionnaire Python en une chaîne JSON et *vice versa* :

```
In [1]: import json
In [2]: # Dictionnaire Python...
        user_dict = {"name": "Jane Doe",
                     "age": 23,
                     "married": False,
                     "children": None,
                     "hobbies": ["hiking", "reading"]}
In [3]: # ...converti en une chaîne JSON
        # par json.dumps ("dump string"). Le paramètre "indent"
        # est facultatif mais améliore la lisibilité.
        user_json = json.dumps(user_dict, indent=4)
        print(user_json)
        {
            "name": "Jane Doe",
            "age": 23,
            "married": false,
            "children": null,
            "hobbies": [
                "hiking",
                "reading"
            ]
        }
In [4]: # Reconvertit la chaîne JSON en une structure de données Python
        json.loads(user_json)
Out[4]: {'name': 'Jane Doe',
         'age': 23,
         'married': False,
         'children': None,
         'hobbies': ['hiking', 'reading']}
```

API REST

Au lieu d'API Web, vous verrez souvent le terme *REST* ou API *RESTful*. REST, qui est l'abréviation de *representational state transfer*, définit une API Web qui respecte certaines contraintes. Le concept de base de REST est que vous accédez à des informations sous la forme de *ressources stateless* (sans état). *Stateless* signifie que chaque requête à une API REST est complètement indépendante de toute autre requête et doit toujours fournir l'ensemble complet des informations qu'elle demande. Vous noterez que le terme *API REST* est souvent utilisé à tort pour désigner tout type d'API Web, même si elle ne respecte pas les contraintes REST.

La consommation d'API Web est généralement très simple (nous verrons comment cela fonctionne avec Python dans un instant), et presque tous les services offrent ce type d'API. Si vous

souhaitez télécharger votre liste de lecture Spotify préférée, vous devez émettre la requête GET suivante (voir le Guide de référence de l'API Web Spotify[2]) :

```
GET https://api.spotify.com/v1/playlists/playlist_id
```

Ou, si vous souhaitez obtenir des informations sur vos derniers trajets Uber, vous devez exécuter la demande GET suivante (voir l'API REST Uber[3]) :

```
GET https://api.uber.com/v1.2/history
```

Pour utiliser ces API, cependant, vous devez être authentifié, ce qui signifie généralement que vous avez besoin d'un compte et d'un token que vous pouvez envoyer avec vos demandes. Pour Python Package Tracker, nous devrons récupérer des données de PyPI pour obtenir des informations sur les versions d'un package spécifique. Heureusement, l'API Web de PyPI ne nécessite aucune authentification, ce qui fait un sujet d'inquiétude en moins. Lorsque vous jetterez un coup d'œil à la documentation de l'API JSON de PyPI[4], vous verrez qu'il n'y a que deux *points de terminaison*, c'est-à-dire des fragments d'URL qui sont ajoutés à l'URL de base commune, *https://pypi.org/pypi* :

```
GET /project_name/json
GET /project_name/version/json
```

Le deuxième point de terminaison donne les mêmes informations que le premier, mais uniquement pour une version spécifique. Pour Python Package Tracker, le premier point de terminaison est tout ce dont nous avons besoin pour obtenir les détails sur les versions passées d'un package, alors voyons comment cela fonctionne. En Python, un moyen simple d'interagir avec une API Web consiste à utiliser le package Requests préinstallé avec Anaconda. Exécutez les commandes suivantes pour récupérer des données PyPI sur pandas :

```
In [5]: import requests
In [6]: response = requests.get("https://pypi.org/pypi/pandas/json")
response.status_code
Out[6]: 200
```

Chaque réponse fournit un code d'état HTTP : par exemple, `200` signifie OK et `404` signifie que la ressource est introuvable. Vous pouvez consulter la liste complète des codes d'état de réponse HTTP dans la documentation de Mozilla[5]. Vous connaissez sans doute le code d'état `404` car votre navigateur l'affiche parfois lorsque vous cliquez sur un lien mort ou saisissez une adresse qui n'existe pas. De même, vous obtiendrez également un code d'état `404` si vous exécutez une requête GET avec un nom de package qui n'existe pas sur PyPI. Pour examiner le contenu de la réponse, il est plus facile d'appeler la méthode `json` de l'objet `response`, qui transformera la chaîne JSON de la réponse en dictionnaire Python :

2 https://developer.spotify.com/documentation/web-api/reference/#endpoint-get-playlist
3 https://developer.uber.com/docs/riders/references/api/v1.2/history-get
4 https://warehouse.pypa.io/api-reference/json.html
5 https://developer.mozilla.org/fr/docs/Web/HTTP/Status

```
In [7]: response.json()
```

La réponse étant très longue, je n'imprime par conséquent qu'un court extrait pour vous permettre de comprendre la structure :

```
Out[7]: {
            'info': {
                'bugtrack_url': None,
                'license': 'BSD',
                'maintainer': 'The PyData Development Team',
                'maintainer_email': 'pydata@googlegroups.com',
                'name': 'pandas'
            },
            'releases': {
                '0.1': [
                    {
                        'filename': 'pandas-0.1.tar.gz',
                        'size': 238458,
                        'upload_time': '2009-12-25T23:58:31'
                    },
                    {
                        'filename': 'pandas-0.1.win32-py2.5.exe',
                        'size': 313639,
                        'upload_time': '2009-12-26T17:14:35'
                    }
                ]
            }
        }
```

Pour obtenir une liste avec toutes les versions et leurs dates, ce dont nous avons besoin pour Python Package Tracker, nous pouvons exécuter le code suivant pour parcourir le dictionnaire des versions :

```
In [8]: releases = []
        for version, files in response.json()['releases'].items():
            releases.append(f"{version}: {files[0]['upload_time']}")
        releases[:3]  # affiche les 3 premiers éléments de la liste
Out[8]: ['0.1: 2009-12-25T23:58:31',
         '0.10.0: 2012-12-17T16:52:06',
         '0.10.1: 2013-01-22T05:22:09']
```

Notez que nous choisissons arbitrairement l'horodatage de la version du package qui apparaît en premier dans la liste. Une version spécifique a souvent plusieurs packages pour prendre en compte différentes versions de Python et divers systèmes d'exploitation. Pour conclure sur ce sujet, vous vous souvenez peut-être (voir le Chapitre 5) que pandas dispose d'une méthode `read_json` pour renvoyer un DataFrame directement à partir d'une chaîne JSON. Ceci, cependant, ne nous serait d'aucune aide ici, car la réponse de PyPI n'est pas proposée dans un format qui puisse être directement transformé en un DataFrame.

Il ne s'agissait ici que d'une brève introduction aux API Web pour que vous compreniez leur emploi dans le code de Python Package Tracker, et nous allons maintenant voir comment nous

pouvons communiquer avec des bases de données, l'autre système externe que nous utilisons dans notre application.

Bases

Pour pouvoir utiliser les données de PyPI, même lorsque vous n'êtes pas connecté à Internet, vous devez les stocker après les avoir téléchargées. Alors que vous pouvez stocker vos réponses JSON sous forme de fichiers texte sur votre disque dur, une solution beaucoup plus confortable consiste à utiliser une base de données, car cela vous permet d'interroger facilement vos données. Python Package Tracker utilise SQLite[6], une *base de données relationnelle*. Les systèmes de bases de données relationnelles tirent leur nom du concept de *relation*, qui désigne une table d'une base de données (et non pas la relation entre les tables, ce qui est une erreur d'interprétation très répandue) : leur objectif principal est l'intégrité des données, qu'ils atteignent en répartissant les données entre différentes tables (un processus appelé normalisation) et en appliquant des contraintes pour éviter les données incohérentes et redondantes. Les bases de données relationnelles utilisent le langage SQL (Structured Query Language) pour effectuer des requêtes sur la base de données ; parmi les systèmes de bases de données relationnelles serveur les plus populaires, on peut citer SQL Server[7], Oracle[8], PostgreSQL[9] et MySQL[10]. En tant qu'utilisateur Excel, vous connaissez peut-être aussi la base de données Microsoft Access[11] qui est basée sur des fichiers.

SQLite, la base de données que nous allons utiliser, est une base de données qui s'appuie sur des fichiers, comme Microsoft Access. Cependant, contrairement à Microsoft Access, qui ne fonctionne que sous Windows, SQLite est opérationnelle sur toutes les plateformes prises en charge par Python. D'autre part, SQLite ne permet pas de créer une interface utilisateur comme Microsoft Access, mais Excel est très pratique dans ce domaine.

Jetons maintenant un coup d'œil à la structure de la base de données de Package Tracker avant de découvrir comment nous pouvons utiliser Python pour nous connecter à des bases de données et créer des requêtes SQL. Ensuite, pour conclure cette introduction sur les bases de données, nous examinerons les injections SQL, une vulnérabilité très courante dans les applications pilotées par des bases de données.

6 https://sqlite.org/index.html
7 https://www.microsoft.com/fr-fr/sql-server?rtc=1
8 https://www.oracle.com/fr/database/
9 https://www.postgresql.org/
10 https://www.mysql.com/
11 https://www.microsoft.com/fr-fr/microsoft-365/access/?rtc=1

Bases de données NoSQL

De nos jours, les bases de données relationnelles sont fortement concurrencées par les bases de données NoSQL qui stockent des données redondantes dans le but d'obtenir les avantages suivants :

Aucune jointure de table
Étant donné que les bases de données relationnelles répartissent leurs données entre plusieurs tables, vous devez souvent combiner les informations de plusieurs tables en effectuant une jointure, ce qui peut parfois être lent. Cela n'est pas nécessaire avec les bases de données NoSQL, ce qui peut améliorer les performances pour certains types de requêtes.

Aucune migration de base de données
Avec les systèmes de base de données relationnelle, chaque fois que vous apportez une modification à la structure d'une table, par exemple en ajoutant une nouvelle colonne à une table, vous devez exécuter une *migration* de la base de données. Une migration est un script qui transfère la base de données dans la nouvelle structure souhaitée. Cela rend le déploiement de nouvelles versions d'une application plus complexe et peut entraîner des temps d'arrêt, ce qui est plus facile à éviter avec les bases de données NoSQL.

Mise à l'échelle plus simple
Les bases de données NoSQL sont plus faciles à distribuer sur plusieurs serveurs car il n'y a pas de tables qui dépendent les unes des autres. Cela signifie qu'une application qui utilise une base de données NoSQL peut mieux évoluer lorsque votre nombre d'utilisateurs grimpe en flèche.

Les bases de données NoSQL se présentent sous de nombreuses formes : certaines bases de données sont de simples entrepôts clé-valeur, c'est-à-dire qu'elles fonctionnent de la même manière qu'un dictionnaire en Python (par exemple, Redis[12]) ; d'autres permettent le stockage de documents, souvent au format JSON (par exemple, MongoDB[13]). Certaines bases de données combinent même les mondes relationnel et NoSQL : PostgreSQL, qui se trouve être l'une des bases de données les plus populaires de la communauté Python, est traditionnellement une base de données relationnelle, mais elle permet également de stocker des données au format JSON, sans perdre la possibilité de les interroger *via* SQL.

12 https://redis.io/
13 https://www.mongodb.com/fr-fr

Base de données de Package Tracker

La base de données de Python Package Tracker ne pourrait pas être plus simple, car elle ne comporte que deux tables : la table `packages` stocke le nom du package et la table `package_versions` stocke sous la forme d'une chaîne de caractères le numéro de version et la date du téléchargement. On peut faire une jointure des deux tables sur le champ `package_id` : plutôt que de stocker les noms des packages (dans le champ `package_name`) à chaque ligne de la table `package_versions`, l'information a été normalisée dans la table `packages`. Cela élimine les données redondantes : les changements de nom, par exemple, ne doivent être effectués que dans un seul champ sur l'ensemble de la base de données. Pour avoir une meilleure idée de l'apparence de la base de données comportant les informations relatives aux packages xlwings et pandas, consultez le Tableau 11.1 et le Tableau 11.2.

Tableau 11.1 : Table packages.

package_id	package_name
1	xlwings
2	pandas

Tableau 11.2 : Table package_versions (trois premières lignes pour chaque package_id).

package_id	version_string	uploaded_at
1	0.1.0	2014-03-19 18:18:49.000000
1	0.1.1	2014-06-27 16:26:36.000000
1	0.2.0	2014-07-29 17:14:22.000000
...	...	...
2	0.1	2009-12-25 23:58:31.000000
2	0.2beta	2010-05-18 15:05:11.000000
2	0,2b1	2010-05-18 15:09:05.000000
...	...	...

La Figure 11.3 illustre un diagramme de base de données qui montre à nouveau le schéma des deux tables. Vous pouvez lire les noms des tables et des colonnes et obtenir des informations sur les clés primaires et étrangères :

Clé primaire

> Les bases de données relationnelles nécessitent que chaque table ait une *clé primaire*. Une clé primaire représente une ou plusieurs colonnes qui identifient de manière unique une ligne (une ligne est également appelée un *enregistrement*). Dans le cas de la table `packages`, la clé primaire est `package_id` et dans le cas de la table `package_versions`, la clé primaire est une clé dite composite, c'est-à-dire une combinaison des champs `package_id` et `version_string`.

Clé étrangère

La colonne `package_id` de la table `package_versions` est une clé étrangère pour la même colonne de la table `packages`, symbolisée par la ligne qui relie les deux tables : une clé étrangère est une contrainte qui, dans notre cas, garantit que chaque `package_id` de la table `package_versions` existe dans la table `packages`, ce qui garantit l'intégrité des données. Les branches à l'extrémité droite de la ligne de connexion symbolisent la nature de la relation : un `package` peut avoir plusieurs `package_versions` ; c'est ce que l'on appelle une *relation un-à-plusieurs*.

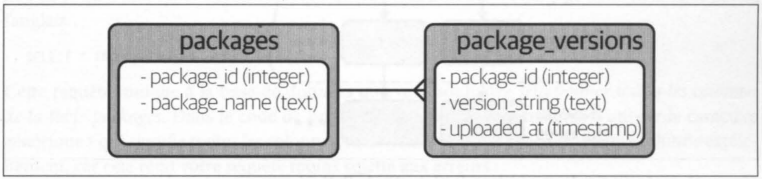


Figure 11.3 : Diagramme de base de données (les clés primaires sont en gras).

Pour consulter le contenu des tables d'une base de données et exécuter des requêtes SQL, vous pouvez installer une extension de VS Code, appelée SQLite (vous trouverez plus de détails dans la documentation de SQLite[14]) ou utiliser un logiciel de gestion de SQLite dédié (on en trouve en abondance). Cependant, nous utiliserons Python pour exécuter des requêtes SQL. Avant toute chose, voyons comment nous pouvons nous connecter à une base de données !

Connexions à des bases de données

Pour vous connecter à une base de données à partir de Python, vous avez besoin d'un pilote, c'est-à-dire d'un package Python qui sache comment communiquer avec la base de données que vous utilisez. Chaque base de données nécessite un pilote différent et chaque pilote utilise une syntaxe différente, mais heureusement, il existe un package puissant, nommé SQLAlchemy[15], qui fait abstraction de la plupart des différences entre les diverses bases de données et les pilotes. SQLAlchemy est principalement utilisé comme mappeur relationnel d'objet (ORM pour *object relational mapper*) qui traduit vos enregistrements de base de données en objets Python, un concept que de nombreux développeurs, mais pas tous, trouvent plus naturel à exploiter. Pour garder les choses simples, nous allons ignorer la fonctionnalité ORM et nous n'utiliserons SQLAlchemy que pour faciliter l'exécution de requêtes SQL brutes. SQLAlchemy est également employé en arrière-plan lorsque vous utilisez pandas pour lire et écrire des tables de base de données sous la forme de DataFrames. L'exécution d'une requête de base de données à partir de pandas implique trois niveaux de packages : pandas, SQLAlchemy et le pilote de base de

14 https://marketplace.visualstudio.com/items?itemName=alexcvzz.vscode-sqlite
15 https://www.sqlalchemy.org/

données (voir la Figure 11.4). Vous pouvez exécuter des requêtes de base de données à partir de chacun de ces trois niveaux.

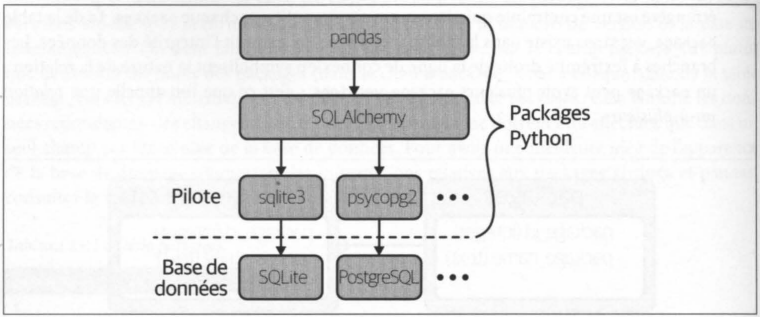


Figure 11.4 : Accès aux bases de données à partir de Python.

Le Tableau 11.3 montre quel pilote SQLAlchemy utilise par défaut (certaines bases de données peuvent être exploitées avec plusieurs pilotes). Il fournit également le format de la chaîne de connexion à la base de données : nous utiliserons la chaîne de connexion quand nous exécuterons des requêtes SQL réelles.

Tableau 11.3 : Pilotes par défaut et chaînes de connexion SQLAlchemy.

Base de données	Pilote par défaut	Chaîne de connexion
SQLite	`sqlite3`	`sqlite:///chemin`
PostgreSQL	`psycopg2`	`postgresql://nom_utilisateur:mot_de_passe@hôte:port/base_de_données`
MySQL	`mysql-python`	`mysql://nom_utilisateur:mot_de_passe@hôte:port/base_de_données`
Oracle	`cx_oracle`	`oracle://nom_utilisateur:mot_de_passe@hôte:port/base_de_données`
SQL Server	`pyodbc`	`mssql+pyodbc://nom_utilisateur:mot_de_passe@hôte:port/base_de_données`

À l'exception de SQLite, vous avez généralement besoin d'un mot de passe pour vous connecter à une base de données. Et puisque les chaînes de connexion sont des URL, vous devez utiliser la version codée en URL de vos mots de passe s'ils contiennent des caractères spéciaux. Voici comment afficher la version codée de votre mot de passe :

```
In [9]: import urllib.parse
In [10]: urllib.parse.quote_plus("pa$$word")
Out[10]: 'pa%24%24word'
```

Après avoir présenté pandas, SQLAlchemy et le pilote de base de données qui représentent les trois niveaux à partir desquels nous pouvons nous connecter aux bases de données, voyons comment ils se comparent dans la pratique en faisant quelques requêtes SQL.

Requêtes SQL

Même si vous êtes novice en SQL, vous ne devriez avoir aucun mal à comprendre les quelques requêtes SQL que j'utiliserai dans les exemples suivants et dans Python Package Tracker. SQL est un *langage déclaratif*, ce qui signifie que vous dites à la base de données *ce que vous voulez* au lieu de *ce qu'il faut faire*. Certaines requêtes sont très proches du langage naturel, si vous parlez l'anglais…

```
SELECT * FROM packages
```

Cette requête indique à la base de données que vous souhaitez *sélectionner toutes les colonnes de la table packages*. Dans le code de production, il n'est pas souhaitable d'utiliser le caractère générique * qui signifie toutes les colonnes, mais il faudra plutôt spécifier chaque colonne explicitement, car cela rend votre requête moins sujette aux erreurs :

```
SELECT package_id, package_name FROM packages
```



SQL est un langage basé sur la théorie des ensembles, ce qui signifie que vous opérez sur un ensemble de lignes plutôt que de parcourir en boucle des lignes individuelles. Ceci est très similaire à la façon dont vous travaillez avec les DataFrames pandas. La requête SQL :

```
SELECT package_id, package_name FROM packages
```

correspond à l'expression pandas suivante (en supposant que `packages` soit un DataFrame) :

```
packages.loc[:, ["package_id", "package_name"]]
```

Les exemples de code suivants utilisent le fichier *packagetracker.db* que vous trouverez dans le dossier *packagetracker* des exemples de code. Les exemples supposent que vous ayez déjà ajouté xlwings et pandas à la base de données *via* le frontend Excel de Python Package Tracker, comme nous l'avons fait au début de ce chapitre ; dans le cas contraire, vous n'obtiendrez que des résultats vides. En suivant de bas en haut le schéma de la Figure 11.4, nous allons d'abord faire notre requête SQL directement à partir du pilote, puis utiliser SQLAlchemy et enfin pandas :

```
In [11]: # On commence par les importations
         import sqlite3
         from sqlalchemy import create_engine
         import pandas as pd
In [12]: # La requête SQL sélectionne toutes les colonnes de la table packages
         sql = "SELECT * FROM packages"
```

```
In [13]: # Option 1: pilote (sqlite3 fait partie de la bibliothèque standard)
         # En utilisant la connexion comme gestionnaire de contexte,
         # on comite automatiquement la transaction ou
         # on l'annule en cas d'erreur.
         with sqlite3.connect("packagetracker/packagetracker.db") as con:
             cursor = con.cursor()  # We need a cursor to run SQL queries
             result = cursor.execute(sql).fetchall()
             # Renvoie tous les enregistrements
         result
Out[13]: [(1, 'xlwings'), (2, 'pandas')]
In [14]: # Option 2: SQLAlchemy
         # "create_engine" attend la chaîne de connexion de la BD.
         # Ici, on peut exécuter une requête comme une méthode
         # de l'objet connection.
         engine = create_engine("sqlite:///packagetracker/packagetracker.db")
         with engine.connect() as con:
             result = con.execute(sql).fetchall()
         result
Out[14]: [(1, 'xlwings'), (2, 'pandas')]
In [15]: # Option 3: pandas
         # En fournissant un nom de table à "read_sql", on lit la table complète.
         # pandas nécessite un moteur SQLAlchemy que l'on réutilise
         # à partir de l'exemple précédent.
         df = pd.read_sql("packages", engine, index_col="package_id")
         df
Out[15]:
```

package_id	package_name
1	xlwings
2	pandas

```
In [16]: # "read_sql" accepte aussi une requête SQL
         pd.read_sql(sql, engine, index_col="package_id")
Out[16]:
```

package_id	package_name
1	xlwings
2	pandas

```
In [17]: # La méthode de DataFrame "to_sql" écrit les DataFrames dans des tables
         # "if_exists" peut prendre les valeurs "fail", "append" ou "replace"
         # et définit le comportement si la table existe déjà
         df.to_sql("packages2", con=engine, if_exists="append")
In [18]: # La commande précédente a créé une nouvelle table "packages2"
         # et a inséré les enregistrements du DataFrame df
         # comme on peut le vérifier en relisant les valeurs
         pd.read_sql("packages2", engine, index_col="package_id")
```

```
Out[18]:
```

package_id	package_name
1	xlwings
2	pandas

```
In [19]: # On se débarrasse de la table en exécutant la commande
         # "drop table" via SQLAlchemy
         with engine.connect() as con:
             con.execute("DROP TABLE packages2")
```

L'emploi du pilote de base de données, de SQLAlchemy ou de pandas pour exécuter vos requêtes dépend en grande partie de vos préférences : personnellement, j'aime le contrôle précis que l'on obtient avec SQLAlchemy et j'apprécie de pouvoir utiliser la même syntaxe avec différentes bases de données. D'un autre côté, la fonction `read_sql` de pandas est pratique pour obtenir le résultat d'une requête sous la forme d'un DataFrame.



De manière assez surprenante, SQLite ne respecte pas les clés étrangères par défaut lors de l'exécution de requêtes. Toutefois, si vous utilisez SQLAlchemy, vous pouvez facilement appliquer des clés étrangères ; consultez pour ce faire la documentation de SQLAlchemy[16]. Cela fonctionnera également si vous exécutez les requêtes à partir de pandas. Vous trouverez le code correspondant en haut du module *database.py* dans le dossier *packagetracker* des exemples de code.

Maintenant que vous savez comment exécuter des requêtes SQL simples, achevons cette section en examinant les injections SQL, qui peuvent menacer la sécurité de votre application.

Injection SQL

Si vous ne protégez pas correctement vos requêtes SQL, un utilisateur malveillant peut exécuter du code SQL en injectant des instructions SQL dans les champs de saisie des données : par exemple, au lieu de sélectionner un nom de package comme xlwings dans la liste déroulante de Python Package Tracker, il peut envoyer une instruction SQL qui modifie votre requête. Cela peut dévoiler des informations sensibles ou effectuer des actions destructrices telles que la suppression d'une table. Comment empêcher cela ? Jetons d'abord un coup d'œil à la requête de base de données suivante, que le Package Tracker exécute lorsque vous sélectionnez xlwings et cliquez sur le bouton Show History[17] :

```
SELECT v.uploaded_at, v.version_string
FROM packages p
INNER JOIN package_versions v ON p.package_id = v.package_id
WHERE p.package_id = 1
```

16 https://docs.sqlalchemy.org/en/14/dialects/sqlite.html#foreign-key-support

17 En réalité, l'outil utilise `package_name` au lieu de `package_id` pour simplifier le code.

Cette requête effectue une jointure entre les deux tables et renvoie uniquement les lignes pour lesquelles `package_id` est égal à `1`. Pour vous aider à comprendre cette requête basée sur ce que nous avons appris au Chapitre 5, si `packages` et `package_versions` étaient des DataFrames pandas, vous pourriez écrire :

```
df = packages.merge(package_versions, how="inner", on="package_id")
df.loc[df["package_id"] == 1, ["uploaded_at", "version_string"]]
```

Il est évident que `package_id` doit être une variable (nous avons à présent un `1` codé en dur) afin de renvoyer les lignes correctes en fonction du package sélectionné. En puisant dans vos connaissances sur les chaînes de formatage (voir le Chapitre 3), vous pourriez être tenté de modifier la dernière ligne de la requête SQL de la manière suivante :

```
f"WHERE p.package_id = {package_id}"
```

Bien que cela fonctionne techniquement, vous ne devez jamais le faire car cela offre la possibilité d'une injection SQL : par exemple, quelqu'un pourrait envoyer `'1 OR TRUE'` à la place d'un entier représentant le numéro du package (le champ `package_id`). La requête qui en résulterait renverrait toutes les lignes de la table au lieu de celles pour lesquelles `package_id` est égal à `1`. Par conséquent, utilisez toujours la syntaxe que SQLAlchemy propose pour les espaces réservés (ils commencent par le caractère deux-points) :

```
In [20]: # On commence par importer la fonction text de SQLAlchemy
         from sqlalchemy.sql import text
In [21]: # ":package_id" est l'emplacement réservé
         sql = """
         SELECT v.uploaded_at, v.version_string
         FROM packages p
         INNER JOIN package_versions v ON p.package_id = v.package_id
         WHERE p.package_id = :package_id
         ORDER BY v.uploaded_at
         """
In [22]: # Via SQLAlchemy
         with engine.connect() as con:
             result = con.execute(text(sql), package_id=1).fetchall()
         result[:3]  # Print the first 3 records
Out[22]: [('2014-03-19 18:18:49.000000', '0.1.0'),
          ('2014-06-27 16:26:36.000000', '0.1.1'),
          ('2014-07-29 17:14:22.000000', '0.2.0')]
In [23]: # Via pandas
         pd.read_sql(text(sql), engine, parse_dates=["uploaded_at"],
                     params={"package_id": 1},
                     index_col=["uploaded_at"]).head(3)
Out[23]:
```

uploaded_at	version_string
2014-03-19 18:18:49	0.1.0
2014-06-27 16:26:36	0.1.1
2014-07-29 17:14:22	0.2.0

L'encapsulation de la requête SQL avec la fonction `text` de SQLAlchemy présente l'avantage de pouvoir utiliser la même syntaxe pour les espaces réservés dans différentes bases de données. Sinon, vous devez employer l'espace réservé utilisé par le pilote de base de données : `sqlite3` utilise `?` et `psycopg2` `%s`, par exemple.

Vous pourriez faire valoir que l'injection SQL n'est pas vraiment un problème lorsque vos utilisateurs ont un accès direct à Python et pourraient, de toute façon, exécuter du code sur la base de données. Mais si vous décidez un jour de transformer votre prototype xlwings en une application Web, cela deviendra alors un énorme problème, il est donc préférable de faire les choses correctement dès le début.

Il existe un autre sujet que nous avons survolé jusqu'à présent et qui est indispensable quand on veut développer des applications robustes : la gestion des exceptions. Voyons comment cela fonctionne !

Exceptions

J'ai mentionné la gestion des exceptions, dans le Chapitre 1, à titre d'exemple pour illustrer la manière dont VBA, avec son mécanisme *GoTo* a pris du retard. Dans cette section, je vais vous montrer comment Python utilise le mécanisme *try/except* pour gérer les erreurs dans les programmes. Chaque fois que quelque chose échappe à votre contrôle, des erreurs peuvent se produire et se produiront. Par exemple, le serveur de messagerie peut être en panne lorsque vous essayez d'envoyer un e-mail, ou un fichier que votre programme attend peut être manquant ; dans le cas de Python Package Tracker, il peut s'agir du fichier de base de données. Le traitement de la saisie de l'utilisateur est un autre domaine où vous devez vous préparer à la saisie d'informations incohérentes. Prenons un exemple : si la fonction suivante est appelée avec un zéro, vous obtiendrez une erreur `ZeroDivisionError` :

```
In [24]: def print_reciprocal(number):
             result = 1 / number
             print(f"L'inverse est : {result}")
In [25]: print_reciprocal(0)  # Provoque une erreur
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-30-095f19ebb9e9> in <module>
----> 1 print_reciprocal(0)  # This will raise an error

<ipython-input-29-b2efd687d0f8> in print_reciprocal(number)
      1 def print_reciprocal(number):
----> 2     result = 1 / number
      3     print(f"L'inverse est : {result}")

ZeroDivisionError: division by zero
```

Pour permettre à votre programme de réagir avec élégance à de telles erreurs, utilisez les instructions `try/except` (il s'agit de l'équivalent de l'exemple VBA du Chapitre 1) :

```
In [26]: def print_reciprocal(number):
             try:
                 result = 1 / number
             except Exception as e:
                 # "as e" rend l'objet Exception disponible comme variable "e"
                 # "repr" signifie "représentation affichable" d'un objet
                 # et fournit une chaîne avec le message d'erreur
                 print(f"Il y a eu une erreur : {repr(e)}")
                 result = "Non disponible"
             else:
                 print("Il n'y a pas eu d'erreur !")
             finally:
                 print(f"L'inverse est : {result}")
```

Chaque fois qu'une erreur se produit dans le bloc `try`, l'exécution du code passe au bloc `except` où vous pouvez gérer l'erreur : cela permet de fournir à l'utilisateur des commentaires utiles ou d'écrire l'erreur dans un journal des erreurs. La clause `else` ne s'exécute que si aucune erreur n'est déclenchée dans le bloc `try` et le bloc `finally` s'exécute toujours, qu'une erreur ait été déclenchée ou non. Souvent, vous vous en sortirez juste avec les blocs `try` et `except`. Voyons le résultat de la fonction quand on lui passe différents paramètres :

```
In [27]: print_reciprocal(10)
Il n'y a pas eu d'erreur !
L'inverse est : 0.1
In [28]: print_reciprocal("a")
Il y a eu une erreur : TypeError("unsupported operand type(s) for /: 'int' and 'str'")
L'inverse est : Non disponible
In [29]: print_reciprocal(0)
Il y a eu une erreur : ZeroDivisionError('division by zero')
L'inverse est : Non disponible
```

La façon dont j'ai utilisé l'instruction `except` a pour conséquence que toute exception qui se produit dans le bloc `try` entraînera la poursuite de l'exécution du code dans le bloc `except`. Habituellement, ce n'est pas ce qui est souhaitable. On veut vérifier une erreur aussi spécifique que possible et ne gérer que celles qui sont attendues. Dans le cas contraire, votre programme peut planter pour quelque chose de complètement inattendu, ce qui le rend difficile à déboguer. Pour résoudre ce problème, réécrivez la fonction de la manière suivante, en vérifiant explicitement les deux erreurs que nous attendons (je laisse de côté les instructions `else` et `finally`) :

```
In [30]: def print_reciprocal(number):
             try:
                 result = 1 / number
                 print(f"L'inverse est : {result}")
             except (TypeError, ZeroDivisionError):
                 print("Saisissez n'importe quel nombre, sauf 0.")
```

Exécutons à nouveau le code :

```
In [31]: print_reciprocal("a")
Saisissez n'importe quel nombre, sauf 0.
```

Si vous souhaitez gérer une erreur différemment, en fonction de l'exception, il faut gérer les exceptions séparément :

```
In [32]: def print_reciprocal(number):
             try:
                 result = 1 / number
                 print(f"L'inverse est : {result}")
             except TypeError:
                 print("Saisissez un nombre.")
             except ZeroDivisionError:
                 print("L'inverse de 0 n'est pas défini.")
In [33]: print_reciprocal("a")
Saisissez un nombre.
In [34]: print_reciprocal(0)
L'inverse de 0 n'est pas défini.
```

Maintenant que vous connaissez la gestion des erreurs, les API Web et les bases de données, vous êtes prêt à passer à la section suivante, où nous allons passer en revue chaque composant de Python Package Tracker.

Structure de l'application

Dans cette section, nous allons regarder dans les coulisses de Python Package Tracker pour comprendre comment tout fonctionne. Tout d'abord, nous allons parcourir le frontend de l'application, c'est-à-dire le fichier Excel, avant de regarder son backend, c'est-à-dire le code Python. Pour conclure cette section, nous verrons comment fonctionne le débogage d'un projet xlwings, une compétence utile avec des projets de la taille et de la complexité du Package Tracker.

Dans le répertoire *packagetracker* des exemples de code, vous trouverez quatre fichiers. Dans le Chapitre 1, j'ai évoqué le concept de *séparation des problèmes* et nous allons maintenant pouvoir appliquer cette notion en faisant correspondre ces quatre fichiers aux différentes couches de l'application, comme cela est indiqué dans le Tableau 11.4 :

Tableau 11.4 : Séparation des problèmes.

Couche	Fichier	Description
Couche de présentation	`packagetracker.xlsm`	Il s'agit du frontend et, en tant que tel, du seul fichier avec lequel l'utilisateur final interagit.
Couche métier	`packagetracker.py`	Ce module gère le téléchargement des données *via* l'API Web et effectue des calculs avec pandas.
Couche de données	`database.py`	Ce module gère toutes les requêtes de base de données.
Base de données	`packagetracker.db`	Il s'agit d'un fichier de base de données SQLite.

Dans ce contexte, il convient de mentionner que la couche de présentation, c'est-à-dire le fichier Excel, ne contient pas une seule formule de calcul dans les cellules, ce qui rend l'outil beaucoup plus facile à auditer et à contrôler.

Modèle-vue-contrôleur (MVC)

La séparation des problèmes peut se présenter sous différents aspects et la décomposition telle qu'elle est illustrée dans le Tableau 11.4 n'est qu'une possibilité. Il existe un autre modèle de conception populaire que vous pourrez croiser assez rapidement : MVC, qui est le sigle de modèle-vue-contrôleur. Dans le monde MVC, le cœur de l'application est le modèle où toutes les données et généralement la plupart de la logique métier sont gérées. Alors que la vue correspond à la couche de présentation, le contrôleur n'est qu'une couche mince qui se situe entre le modèle et la vue afin de s'assurer qu'ils sont toujours synchronisés. Pour simplifier mon propos, je n'utiliserai pas le modèle MVC dans ce livre.

Maintenant que vous savez à quoi sert chaque fichier, passons à autre chose et examinons de plus près comment le frontend Excel a été configuré.

Frontend

Lorsque vous créez une application Web, vous faites la différence entre le *frontend*, qui est la partie de l'application qui s'exécute dans votre navigateur, et le *backend*, qui est le code qui s'exécute sur le serveur. Nous pouvons appliquer la même terminologie avec les outils xlwings : le frontend est le fichier Excel et le backend est le code Python que vous appelez *via* `RunPython`. Si vous souhaitez créer le frontend à partir de zéro, commencez par exécuter la commande suivante dans Anaconda Prompt (assurez-vous de vous positionner d'abord dans le répertoire de votre choix à l'aide de la commande `cd`) :

```
(base)> xlwings quickstart packagetracker
```

Accédez au répertoire *packagetracker* et ouvrez *packagetracker.xlsm* dans Excel. Commencez par ajouter les trois onglets, Tracker, Database et Dropdown (voir la Figure 11.5).

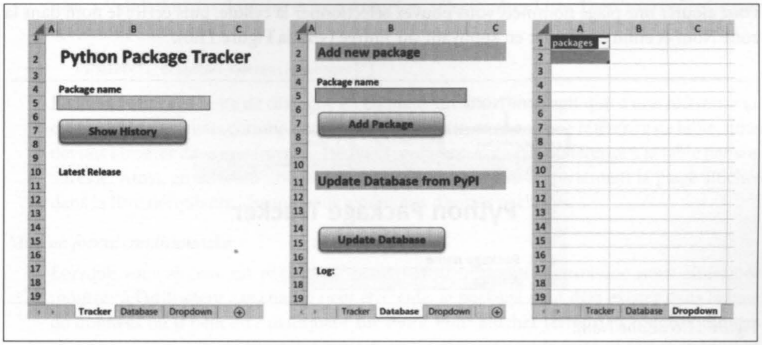


Figure 11.5 : Création de l'interface utilisateur.

La création des libellés et la mise en forme de l'interface utilisateur ne devraient pas vous poser de problème, mais je dois vous donner quelques détails supplémentaires sur les choses qui ne sont pas visibles :

Boutons

Pour que l'outil ressemble un peu moins à Windows 3.1, je n'ai pas utilisé les boutons standard que nous avons employés dans le chapitre précédent. Au lieu de cela, je suis allé dans le menu Insertion > Illustrations > Formes et j'ai choisi d'insérer un rectangle arrondi. Si vous préférez, vous pouvez également utiliser le bouton standard, mais à ce stade, n'affectez pas encore de macro.

Plages nommées

Pour faciliter la maintenance de l'outil, nous utiliserons des plages nommées plutôt que des adresses de cellule dans le code Python. Par conséquent, ajoutez les plages nommées qui sont listées dans le Tableau 11.5.

Tableau 11.5 : Plages nommées.

Feuille	Cellule	Nom
Tracker	B5	`package_selection`
Tracker	B11	`latest_release`
Database	B5	`new_package`
Database	B13	`updated_at`
Database	B18	`log`

Pour ajouter une plage nommée, vous pouvez sélectionner la cellule, puis écrire le nom dans la zone Nom et enfin confirmer en appuyant sur Entrée (voir la Figure 11.6).

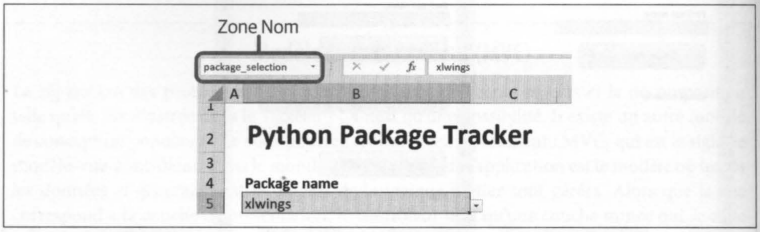


Figure 11.6 : Zone Nom.

Tables

Dans la feuille Dropdown, après avoir saisi « packages » dans la cellule A1, sélectionnez A1, puis accédez à Insertion > Tableau et assurez-vous d'activer la case à cocher « Mon tableau comporte des en-têtes ». Pour terminer, cliquez sur la table pour la sélectionner, accédez à l'onglet du ruban Conception de la table (Windows) ou Table (macOS) et changez le nom par défaut du tableau, `Tableau1`, en `dropdown_content`, comme cela est indiqué à la Figure 11.7.

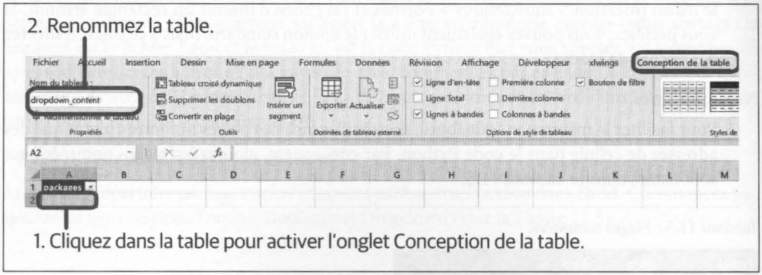


Figure 11.7 : Changement de nom d'une table Excel.

Validation des données

Nous utilisons la validation des données pour fournir la liste déroulante dans la cellule B5 de la feuille Tracker. Pour l'ajouter, sélectionnez la cellule B5, puis accédez à Don-

nées > Validation des données et sous Autoriser, sélectionnez Liste. Dans source, définissez la formule suivante :

```
=INDIRECT("dropdown_content[packages]")
```

Ensuite, validez la boîte de dialogue en cliquant sur OK. Il ne s'agit que d'une référence au corps de la table, mais comme Excel n'accepte pas directement une référence de table, nous devons l'insérer dans une formule `INDIRECT`, qui permet de faire référence à la table par son adresse. Ainsi, en utilisant une table, cela redimensionnera correctement la plage affichée dans la liste déroulante lorsque nous ajouterons d'autres packages.

Mise en forme conditionnelle

Lorsque vous ajoutez un package, il peut y avoir quelques erreurs que nous aimerions montrer à l'utilisateur : le champ peut être vide, le package peut déjà exister dans la base de données ou il peut être manquant sur PyPI. Pour afficher l'erreur en rouge et d'autres messages en noir, nous utiliserons une astuce simple basée sur la mise en forme conditionnelle : nous voulons une police rouge chaque fois que le message contient le mot « error ». Dans la feuille Database, sélectionnez la cellule C5, où nous allons écrire le message. Accédez ensuite à Accueil > Mise en forme conditionnelle > Règles de mise en surbrillance des cellules > Texte qui contient. Saisissez le mot « error » et sélectionnez Texte rouge dans la liste déroulante comme cela est indiqué à la Figure 11.8, puis cliquez sur OK. Appliquez le même format conditionnel à la cellule C5 de la feuille Tracker.

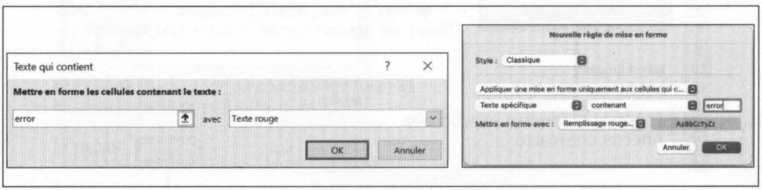


Figure 11.8 : Formatage conditionnel sous Windows (à gauche) et macOS (à droite).

Quadrillage

Sur les feuilles Tracker et Database, le quadrillage a été masqué en décochant la case Quadrillage sous Affichage > Afficher.

À ce stade, l'interface utilisateur est complète et devrait ressembler à celle qui est illustrée à la Figure 11.5. Nous devons maintenant ajouter les appels à `RunPython` dans l'éditeur VBA et les connecter avec les boutons. Cliquez sur Alt + F11 (Windows) ou Option + F11 (macOS) pour ouvrir l'éditeur VBA, puis, sous le VBAProject de *packagetracker.xlsm*, faites un double-clic sur Module1 sur le côté gauche de Modules afin de l'ouvrir. Supprimez le code `SampleCall` existant et remplacez-le par les macros suivantes :

```
Sub AddPackage()
    RunPython "import packagetracker; packagetracker.add_package()"
End Sub

Sub ShowHistory()
    RunPython "import packagetracker; packagetracker.show_history()"
End Sub

Sub UpdateDatabase()
    RunPython "import packagetracker; packagetracker.update_database()"
End Sub
```

Ensuite, cliquez avec le bouton droit sur chaque bouton, sélectionnez Affecter une macro et sélectionnez la macro qui correspond au bouton. La Figure 11.9 illustre le bouton Show History, mais cela fonctionne de la même manière pour les boutons Add Package et Update Database.

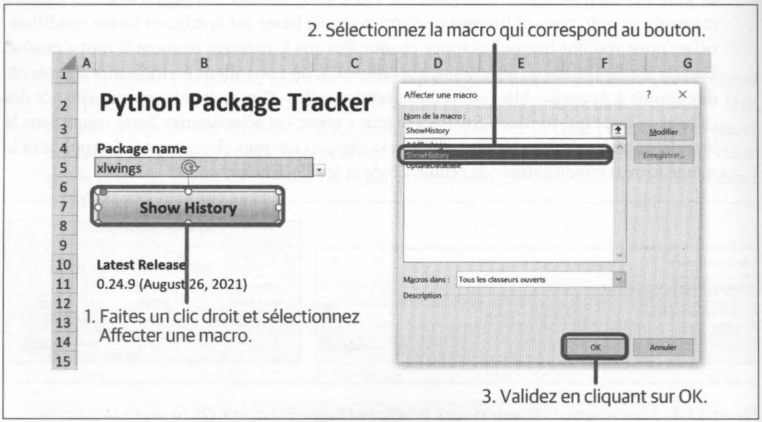


Figure 11.9 : Affectation de la macro ShowHistory au bouton Show History.

Le frontend est maintenant terminé et nous pouvons passer au backend Python.

Backend

Le code des deux fichiers Python *packagetracker.py* et *database.py* est trop long pour être imprimé, et vous devrez donc les ouvrir à partir des exemples de code dans VS Code. Je vais toutefois commenter quelques extraits de code dans cette section afin d'expliquer quelques concepts clés. Voyons ce qui se passe lorsque vous cliquez sur le bouton Add Package dans la feuille Database. La macro VBA suivante est affectée au bouton :

```
Sub AddPackage()
RunPython "import packagetracker; packagetracker.add_package()"
End Sub
```

Comme vous le voyez, la fonction `RunPython` appelle la fonction Python `add_package` du module `packagetracker` comme cela est indiqué dans l'Exemple 11.1.



L'application est aussi simple que possible afin de faciliter sa compréhension ; en ce sens, elle ne vérifie pas toutes les choses possibles qui peuvent mal tourner. Dans un environnement de production, il faudrait la rendre plus robuste : par exemple, il faudrait afficher un message d'erreur convivial si l'application ne trouve pas le fichier de base de données.

Exemple 11.1 : Fonction add_package dans packagetracker.py (sans commentaires).

```
def add_package():
    db_sheet = xw.Book.caller().sheets["Database"]
    package_name = db_sheet["new_package"].value
    feedback_cell = db_sheet["new_package"].offset(column_offset=1)

    feedback_cell.clear_contents()

    if not package_name:
        feedback_cell.value = "Error: Please provide a name!" ❶
        return
    if requests.get(f"{BASE_URL}/{package_name}/json",
                    timeout=6).status_code != 200: ❷
        feedback_cell.value = "Error: Package not found!"
        return

    error = database.store_package(package_name) ❸
    db_sheet["new_package"].clear_contents()

    if error:
        feedback_cell.value = f"Error: {error}"
    else:
        feedback_cell.value = f"Added {package_name} successfully."
        update_database() ❹
        refresh_dropdown() ❺
```

❶ Le mot « error » dans le message affiché en cas d'erreur déclenchera l'affichage dans une police rouge dans Excel *via* la mise en forme conditionnelle.

❷ Par défaut, Requests attend indéfiniment une réponse, ce qui peut conduire l'application à se bloquer au cas où PyPI ait un problème et réponde lentement. C'est la raison pour laquelle dans le code de production, vous devez toujours inclure un paramètre explicite `timeout`.

❸ La fonction `store_package` renvoie `None` si l'opération a réussi et une chaîne avec le message d'erreur dans le cas contraire.

❹ Pour garder les choses simples, toute la base de données est mise à jour. Dans un environnement de production, il ne faudrait ajouter que les enregistrements du nouveau package.

❺ Cela met à jour la table de la feuille Dropdown avec le contenu de la table `packages`. Avec la validation des données que nous avons configurée dans Excel, cela garantit que tous les packages apparaissent dans la liste déroulante de la feuille Tracker. Vous devez donner aux utilisateurs un moyen d'appeler cette fonction directement si vous autorisez le remplissage de la base de données en dehors de votre fichier Excel. C'est le cas dès que vous avez plusieurs utilisateurs qui exploitent la même base de données à partir de différents fichiers Excel.

Vous devriez pouvoir comprendre les autres fonctions du fichier *packagetracker.py* à l'aide des commentaires dans le code. Tournons maintenant notre attention vers le fichier *database.py*. Les premières lignes sont listées dans l'Exemple 11.2.

Exemple 11.2 : database.py (extrait avec les importations pertinentes).

```
from pathlib import Path

import sqlalchemy
import pandas as pd

...

# Nous voulons que le fichier de base de données soit à côté de ce fichier.
# Ici, on transforme le chemin en un chemin absolu.
this_dir = Path(__file__).resolve().parent ❶
db_path = this_dir / "packagetracker.db"

# Moteur de base de données
engine = sqlalchemy.create_engine(f"sqlite:///{db_path}")
```

❶ Si vous avez besoin d'un rappel de ce que fait cette ligne, jetez un coup d'œil au début du Chapitre 7, où je l'explique dans le code du rapport des ventes.

Bien que cet extrait de code concerne la mise en place du chemin d'accès du fichier de base de données, il montre également comment contourner une erreur courante lorsque vous travaillez avec n'importe quel type de fichier, qu'il s'agisse d'une image, d'un fichier CSV ou, comme dans ce cas, d'un fichier de base de données. Lorsque vous voulez rédiger rapidement un script Python, vous pouvez simplement utiliser un chemin relatif comme je l'ai fait dans la plupart des exemples de notebooks Jupyter :

```
engine = sqlalchemy.create_engine("sqlite:///packagetracker.db")
```

Cela fonctionne tant que votre fichier se trouve dans votre répertoire de travail. Cependant, lorsque vous exécutez ce code à partir d'Excel *via* `RunPython`, le répertoire de travail peut être différent, ce qui amènera Python à rechercher le fichier dans le mauvais dossier (vous obtiendrez alors une erreur `File not found`). Vous pouvez résoudre ce problème en fournissant un chemin

absolu ou en créant un chemin comme nous le faisons dans l'Exemple 11.2. Cela garantit que Python recherche le fichier dans le même répertoire que le fichier source, même si vous exécutez le code à partir d'Excel *via* `RunPython`.

Si vous souhaitez recréer Python Package Tracker à partir de zéro, vous devez créer la base de données manuellement : exécutez le fichier `database.py` en tant que script, par exemple en cliquant sur le bouton Exécuter le fichier dans VS Code. Cela créera le fichier de base de données *packagetracker.db* avec les deux tables. Le code qui crée la base de données se trouve tout en bas de *database.py* :

```
if __name__ == "__main__":
    create_db()
```

Alors que la dernière ligne appelle la fonction `create_db`, la signification de l'instruction `if` précédente est expliquée dans l'astuce suivante.



Vous verrez l'instruction `if __name__ == "__main__»` au bas de nombreux fichiers Python. Cela permet de s'assurer que ce code ne s'exécute que lorsque vous exécutez le fichier en tant que script, par exemple, à partir d'Anaconda Prompt en exécutant `python database.py` ou en cliquant sur le bouton Exécuter le fichier dans VS Code. Cependant, le code ne sera pas déclenché lorsque vous exécutez le fichier en l'important en tant que module, c'est-à-dire en effectuant un `import database` dans votre code. La raison en est que Python attribue le nom `__main__` au fichier si vous l'exécutez directement en tant que script, alors qu'il sera appelé par son nom de module (`database`) lorsque vous l'exécutez via l'instruction `import`. Étant donné que Python suit le nom de fichier dans une variable appelée `__name__`, l'instruction `if` n'est évaluée à `True` que lorsque vous l'exécutez en tant que script ; elle ne sera donc pas déclenchée lorsque vous l'importerez à partir du fichier packagetracker.py.

Le reste du module de base de données exécute des instructions SQL à la fois *via* SQLAlchemy et les méthodes pandas `to_sql` et `read_sql` afin que vous ayez une idée des deux approches.

Migration vers PostgreSQL

Si vous voulez remplacer SQLite par PostgreSQL, une base de données basée sur un serveur, il n'y a que quelques modifications à effectuer. Tout d'abord, vous devez exécuter `conda install psycopg2` (ou `pip install psycopg2-binary` si vous n'utilisez pas la distribution Anaconda) afin d'installer le pilote PostgreSQL. Ensuite, dans *database.py*, remplacez la chaîne de connexion dans la fonction `create_engine` par la version PostgreSQL comme cela est indiqué dans le Tableau 11.3. Enfin, pour créer les tables, vous devez remplacer le type de données `INTEGER` de `packages.package_id` par la notation spécifique à PostgreSQL qui est `SERIAL`. La création d'une clé primaire à incrémentation automatique est un exemple de différence entre les dialectes SQL.

Lorsque vous créez des outils aussi complexes que Python Package Tracker, vous rencontrerez probablement quelques problèmes en cours de route : par exemple, vous avez peut-être changé le nom d'une plage nommée dans Excel et oublié de modifier le code Python en conséquence. C'est le bon moment pour examiner le fonctionnement du débogage !

Débogage

Pour déboguer facilement vos scripts xlwings, exécutez vos fonctions directement à partir de VS Code, au lieu de les exécuter en cliquant sur un bouton dans Excel. Les lignes suivantes tout en bas du fichier *packagetracker.py* vous aideront à déboguer la fonction `add_package` (il s'agit du même code que vous trouverez également au bas d'un projet généré par `quickstart`) :

```
if __name__ == "__main__": ❶
    xw.Book("packagetracker.xlsm").set_mock_caller() ❷
    add_package()
```

❶ Nous venons de voir comment cette instruction `if` fonctionne lorsque nous avons examiné le code du fichier *database.py* ; reportez-vous par conséquent à l'astuce précédente.

❷ Comme ce code n'est exécuté que lorsque vous exécutez le fichier directement à partir de Python en tant que script, la commande `set_mock_caller()` ne sert qu'au débogage : lorsque vous exécutez le fichier dans VS Code ou à partir d'Anaconda Prompt, elle définit `xw.Book.caller()` à `xw.Book("packagetracker.xlsm")`. Si l'on fait cela, c'est uniquement pour pouvoir exécuter le script des deux côtés, Python et Excel, sans avoir à basculer l'objet `book` dans la fonction `add_package` entre `xw.Book("packagetracker.xlsm")` (lorsque vous l'appelez depuis VS Code) et `xw.Book.caller()` (lorsque vous l'appelez à partir d'Excel).

Ouvrez *packagetracker.py* dans VS Code et définissez un point d'arrêt sur n'importe quelle ligne de la fonction `add_package` en cliquant à gauche des numéros de ligne. Appuyez ensuite sur F5 et sélectionnez « Python File » dans la boîte de dialogue pour démarrer le débogueur et faire en sorte que votre code s'arrête sur le point d'arrêt. Assurez-vous d'appuyer sur la touche F5 au lieu d'utiliser le bouton Exécuter le fichier, car ce dernier ignore les points d'arrêt.



Sous Windows, lorsque vous exécutez le débogueur de VS Code pour la première fois avec du code qui utilise pandas, vous pouvez être accueilli par une erreur : « Exception has occurred: ImportError, Unable to import required dependencies: numpy » (Une exception s'est produite : ImportError, impossible d'importer les dépendances requises : numpy). Cela se produit car le débogueur est opérationnel avant que l'environnement Conda n'ait été activé correctement. Voici une solution de contournement : arrêtez le débogueur en cliquant sur l'icône d'arrêt, appuyez à nouveau sur F5 et cela fonctionnera la deuxième fois.

Si vous n'êtes pas familier avec le fonctionnement du débogueur dans VS Code, jetez un coup d'œil à l'Annexe B, à l'endroit où j'explique toutes les fonctionnalités et boutons pertinents. Nous

reviendrons également sur le sujet dans la section correspondante du chapitre suivant. Si vous souhaitez déboguer une autre fonction, arrêtez la session de débogage en cours, puis modifiez le nom de la fonction au bas de votre fichier. Par exemple, pour déboguer la fonction `show_history`, modifiez la dernière ligne de *packagetracker.py* de la manière suivante avant d'appuyer à nouveau sur F5 :

```
if __name__ == "__main__":
    xw.Book("packagetracker.xlsm").set_mock_caller()
    show_history()
```

Sous Windows, vous pouvez également activer la case à cocher Show Console dans le complément xlwings, ce qui affichera une invite de commandes pendant l'exécution de l'appel à `RunPython`[18]. Cela vous permet d'afficher des informations supplémentaires pour vous aider à déboguer le problème. Par exemple, vous pouvez afficher la valeur d'une variable pour l'inspecter dans l'invite de commandes. Cependant, une fois le code exécuté, l'invite de commandes sera fermée. Si vous devez la garder ouverte un peu plus longtemps, il existe une astuce simple : ajoutez `input()` comme dernière ligne à votre fonction. Ainsi, Python attendra la saisie de l'utilisateur au lieu de fermer immédiatement l'invite de commandes. Lorsque vous avez terminé d'inspecter la sortie, appuyez sur Entrée dans l'invite de commandes pour la fermer (et assurez-vous simplement de supprimer la ligne `input()` avant de décocher l'option Show Console).

Conclusion

Ce chapitre vous a montré qu'il est possible de créer des applications raisonnablement complexes avec un minimum d'effort. Selon moi, pouvoir tirer parti de puissants packages Python comme Requests ou SQLAlchemy fait toute la différence lorsque je fais la comparaison avec VBA, où communiquer avec des systèmes externes est beaucoup plus difficile. Si vous avez des cas d'utilisation similaires, je vous recommande vivement d'examiner de plus près Requests et SQLAlchemy : être capable de traiter efficacement des sources de données externes vous permettra d'abandonner définitivement la pratique du copier-coller.

Au lieu de cliquer sur des boutons, certains utilisateurs préfèrent créer leurs outils Excel à l'aide de formules de cellule. Le chapitre suivant montre comment xlwings permet d'écrire des fonctions définies par l'utilisateur en Python, ce qui permet de réutiliser la plupart des concepts xlwings que nous avons étudiés jusqu'à présent.

18 Au moment de la rédaction de ce livre, cette option n'est pas encore disponible sous macOS.

CHAPITRE 12
Fonctions définies par l'utilisateur (UDF)

Les trois chapitres précédents vous ont montré comment automatiser Excel avec un script Python et comment exécuter un tel script à partir d'Excel en un clic. Ce chapitre présente les fonctions définies par l'utilisateur (que l'on abrège en UDF car on emploie en général le sigle anglais qui signifie *user-defined function*) comme une alternative pour appeler du code Python à partir d'Excel avec xlwings. Les UDF sont des fonctions Python que l'on emploie dans les cellules Excel de la même manière que l'on utilise des fonctions intégrées telles que `SOMME` ou `MOYENNE`. Comme dans le chapitre précédent, nous allons commencer par la commande `quickstart` qui permet de tester une première UDF instantanément. Nous passerons ensuite à une étude de cas sur la récupération et le traitement des données de Google Trends, ce qui nous fera un prétexte pour travailler avec des UDF plus complexes : nous apprendrons notamment à travailler avec des DataFrames et des tracés pandas ainsi qu'à déboguer des UDF. Pour conclure ce chapitre, nous approfondirons quelques sujets avancés en mettant l'accent sur les performances. Malheureusement, xlwings ne prend pas en charge les UDF sous macOS, ce qui implique que les exemples de ce chapitre ne tournent que sous Windows[1].



Même si vous n'êtes pas sous Windows, vous voudrez peut-être quand même consulter l'étude de cas sur Google Trends, car il est possible de l'adapter facilement pour qu'elle fonctionne avec un appel `RunPython` sous macOS. Vous pouvez également produire un rapport en utilisant l'une des bibliothèques d'écriture que nous avons étudiées au Chapitre 8, et dont la plupart fonctionnent même sous Linux.

1 L'implémentation Windows utilise un serveur COM (j'ai brièvement présenté la technologie COM au Chapitre 9). Étant donné que COM n'existe pas sous macOS, il faudrait repartir de zéro pour que les UDF fonctionnent, ce qui représente un travail conséquent qui n'a pour l'instant pas pu être entrepris…

Prise en main des UDF

Cette section commence par énoncer les conditions préalables à l'écriture des UDF avant de pouvoir utiliser la commande `quickstart` afin d'exécuter notre première UDF. Pour suivre les exemples de ce chapitre, vous devez installer le complément xlwings et activer l'option Excel « Accès approuvé au modèle d'objet du projet VBA » :

Complément

> Je suppose que vous avez installé le complément xlwings comme cela est expliqué au Chapitre 10. Il ne s'agit pas cependant d'une exigence stricte : bien que cela facilite le développement, en particulier quand vous cliquez sur le bouton Import Functions, ce n'est pas nécessaire pour le déploiement et vous pouvez contourner le problème en configurant le classeur en mode autonome (pour plus de détails, reportez-vous au Chapitre 10).

Accès approuvé au modèle d'objet du projet VBA

> Pour pouvoir écrire vos premières UDF, vous devez modifier un paramètre dans Excel : accédez à Fichiers > Options > Centre de gestion de la confidentialité > Paramètres du Centre de gestion de la confidentialité > Paramètres des macros et cochez la case « Accès approuvé au modèle d'objet du projet VBA », comme cela est indiqué à la Figure 12.1. Cela permet à xlwings d'insérer automatiquement un module VBA dans votre classeur lorsque vous cliquez sur le bouton Import Functions du complément, comme nous le verrons sous peu. Étant donné que ce paramètre ne sert que pendant le processus d'importation, vous devez le considérer comme un choix relevant des compétences du développeur dont les utilisateurs finaux n'ont pas à se soucier.

Ces deux conditions préalables étant remplies, vous êtes prêt à exécuter votre première UDF !

Commande quickstart pour les UDF

Comme d'habitude, le moyen le plus simple de démarrer est d'utiliser la commande `quickstart`. Avant d'exécuter ce qui suit dans Anaconda Prompt, assurez-vous de vous positionner dans le répertoire de votre choix *via* la commande `cd` :

```
(base)> xlwings quickstart first_udf
```

Accédez au dossier *first_udf* dans l'Explorateur de fichiers et ouvrez *first_udf.xlsm* dans Excel et *first_udf.py* dans VS Code. Ensuite, sur le ruban du complément xlwings, cliquez sur le bouton Import Functions. Par défaut, il s'agit d'une action silencieuse, c'est-à-dire que vous ne verrez quelque chose qu'en cas d'erreur. Toutefois, si vous activez la case à cocher Show Console dans le complément Excel et cliquez à nouveau sur le bouton Import Functions, une invite de commandes s'ouvre et affiche les éléments suivants :

```
xlwings server running [...]
Imported functions from the following modules: first_udf
```

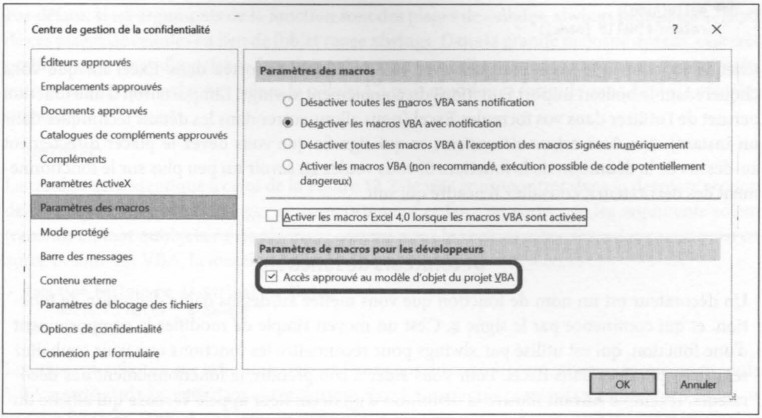


Figure 12.1 : Option Accès approuvé au modèle d'objet du projet VBA.

La première ligne affiche quelques détails supplémentaires que nous pouvons ignorer, la partie importante étant qu'une fois que cette ligne est affichée, cela signifie que Python est opérationnel. La deuxième ligne confirme que les fonctions du module `first_udf` ont été importées correctement. Maintenant, saisissez **`=hello("xlwings")`** dans la cellule A1 de la feuille active (Sheet1) de *first_udf.xlsm* et après avoir appuyé sur la touche Entrée, vous verrez le résultat de la formule (Figure 12.2).

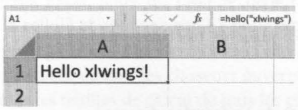


Figure 12.2 : Classeur first_udf.xlsm comportant votre première UDF.

Décomposons cela pour voir comment ça fonctionne : commencez par examiner la fonction `hello` dans *first_udf.py* (Exemple 12.1), qui est la partie du code `quickstart` que nous avons ignorée jusqu'à présent.

Exemple 12.1 : first_udf.py (extrait).

```
import xlwings as xw


@xw.func
```

```
def hello(name):
    return f"Hello {name}!"
```

Chaque fonction que vous marquez avec `@xw.func` sera importée dans Excel lorsque vous cliquerez sur le bouton Import Functions du complément xlwings. L'importation d'une fonction permet de l'utiliser dans vos formules Excel (nous allons entrer dans les détails techniques dans un instant). `@xw.func` est un *décorateur*, ce qui signifie que vous devez le placer directement au-dessus de la définition de la fonction. Si vous voulez en savoir un peu plus sur le fonctionnement des décorateurs, consultez l'encadré qui suit.

Décorateurs de fonctions

Un décorateur est un nom de fonction que vous mettez au-dessus d'une définition de fonction, et qui commence par le signe `@`. C'est un moyen simple de modifier le comportement d'une fonction, qui est utilisé par xlwings pour reconnaître les fonctions que vous souhaitez rendre disponibles dans Excel. Pour vous aider à comprendre le fonctionnement des décorateurs, l'exemple suivant illustre la définition d'un décorateur appelé `verbose` qui affiche du texte avant et après l'exécution de la fonction `print_hello`. Techniquement, le décorateur prend la fonction (`print_hello`) et la fournit comme argument `func` à la fonction `verbose`. La fonction interne appelée `wrapper` peut alors accomplir tout ce qui doit être fait ; dans cet exemple, elle affiche une valeur avant et après avoir appelé la fonction `print_hello`. Le nom de la fonction interne n'a pas d'importance :

```
In [1]: # Définition du décorateur de la fonction
        def verbose(func):
            def wrapper():
                print("Avant appel de fonction.")
                func()
                print("Après appel de fonction.")
            return wrapper
In [2]: # Utilisation du décorateur de fonction
        @verbose
        def print_hello():
            print("hello!")
In [3]: # Résultat de l'appel de la fonction décorée
        print_hello()
Avant appel de fonction.
hello!
Après appel de fonction.
```

À la fin de ce chapitre, vous trouverez dans le Tableau 12.1 un résumé de tous les décorateurs proposés par xlwings.

Par défaut, si les arguments de la fonction sont des plages de cellules, xlwings fournit les valeurs de ces plages de cellules au lieu de l'objet `range` xlwings. Dans la grande majorité des cas, c'est très pratique et permet d'appeler la fonction `hello` avec une cellule comme argument. Par exemple, vous pouvez écrire « xlwings » dans la cellule A2, puis modifier la formule dans A1 de la manière suivante :

```
=hello(A2)
```

Le résultat sera identique à celui de la Figure 12.2. Je vais vous montrer dans la dernière section de ce chapitre comment changer ce comportement et faire en sorte que les arguments soient transmis en tant qu'objets `range` xlwings ; comme nous le verrons alors, il y a des cas où vous en aurez besoin. En VBA, la fonction hello équivalente ressemblerait à ceci :

```
Function hello(name As String) As String
    hello = "Hello " & name & "!"
End Function
```

Lorsque vous cliquez sur le bouton Import Functions du complément, xlwings insère un module VBA appelé `xlwings_udfs` dans votre classeur Excel. Ce module contient une fonction VBA pour chaque fonction Python que vous importez : ces fonctions VBA qui servent d'intermédiaires se chargent d'exécuter la fonction correspondante Python. Bien que personne ne vous empêche de jeter un coup d'œil au module VBA `xlwings_udfs` en ouvrant l'éditeur VBA avec Alt + F11, vous pouvez vous en dispenser, car le code est généré automatiquement et toutes les modifications seront perdues si vous cliquez à nouveau sur le bouton Import Functions. Testons à présent notre fonction `hello` dans `first_udf.py` et remplaçons `Hello` dans la valeur de retour par `Bye` :

```
@xw.func
def hello(name):
    return f"Bye {name}!"
```

Pour recalculer la fonction dans Excel, faites un double-clic sur la cellule A1 pour modifier la formule (ou sélectionnez la cellule et appuyez sur F2 pour activer le mode d'édition), puis appuyez sur Entrée. Vous pouvez également saisir le raccourci clavier Ctrl + Alt + F9 qui force le recalcul de toutes les feuilles de calcul de tous les classeurs ouverts, y compris la formule `hello`. Notez que F9 (recalcul de toutes les feuilles de calcul de tous les classeurs ouverts) ou Maj + F9 (recalcul de la feuille de calcul active) ne recalculera pas l'UDF car Excel déclenche uniquement un recalcul des fonctions définies par l'utilisateur si une cellule dépendante a changé. Pour modifier ce comportement, vous pouvez rendre la fonction *volatile* en ajoutant l'argument correspondant au décorateur `func` :

```
@xw.func(volatile=True)
def hello(name):
    return f"Bye {name}!"
```

Les fonctions volatiles sont évaluées chaque fois qu'Excel effectue un nouveau calcul, que les dépendances de la fonction aient changé ou non. Quelques-unes des fonctions intégrées d'Excel sont volatiles comme `=ALEA()` ou `=MAINTENANT()` et l'emploi d'un grand nombre de ces fonc-

tions ralentira votre classeur, si bien qu'il est conseillé d'en user avec modération. Lorsque vous modifiez le nom ou les arguments d'une fonction ou le décorateur `func`, comme nous venons de le faire, vous devrez réimporter votre fonction en cliquant à nouveau sur le bouton Import Functions : cela redémarrera l'interpréteur Python avant d'importer la fonction mise à jour. Si vous annulez les modifications de la fonction, et repassez de `Bye` à `Hello`, il suffit d'utiliser les raccourcis clavier Maj + F9 ou F9 pour que la formule soit recalculée car la fonction est maintenant volatile.



Une erreur fréquente consiste à oublier d'enregistrer le fichier source Python après avoir apporté des modifications. Par conséquent, vérifiez toujours que le fichier Python est sauvegardé avant d'appuyer sur le bouton Import Functions ou de recalculer les UDF dans Excel.

Par défaut, xlwings importe des fonctions à partir d'un fichier Python situé dans le même répertoire et portant le même nom que le fichier Excel. Renommer et déplacer votre fichier source Python nécessite des modifications similaires à celles que nous avons vues au Chapitre 10, lorsque nous avons fait la même chose avec les appels à `RunPython` : faisons un test et renommons le fichier *first_udf.py* en *hello.py*. Pour informer xlwings de ce changement, ajoutez le nom du module, c'est-à-dire `hello` (sans l'extension *.py*) dans la zone UDF Modules du complément xlwings, comme cela est indiqué à la Figure 12.3.

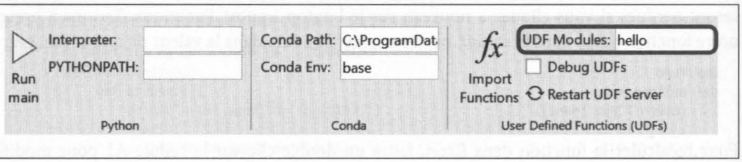


Figure 12.3 : Modification du paramètre UDF Modules.

Cliquez sur le bouton Import Functions pour réimporter la fonction. Ensuite, recalculez la formule dans Excel pour vous assurer que tout fonctionne correctement.



Si vous souhaitez importer des fonctions à partir de plusieurs modules, utilisez un point-virgule entre leurs noms dans le paramètre UDF Modules, par exemple, `hello;autre_module`.

Maintenant, déplacez *hello.py* sur le Bureau : cela vous oblige à ajouter le chemin du Bureau à `PYTHONPATH` dans le complément xlwings. Comme indiqué au Chapitre 10, vous pouvez utiliser des variables d'environnement pour y parvenir, c'est-à-dire que vous pouvez définir le paramètre `PYTHONPATH` dans le complément à `%USERPROFILE%\Desktop`. Si vous avez toujours le chemin d'accès au dossier *pyscripts* sur lequel nous avons travaillé dans le Chapitre 10, vous pouvez le supprimer ou le laissez tel quel, en séparant les chemins par un point-virgule. Après ces mod-

ifications, cliquez à nouveau sur le bouton Import Functions, puis recalculez la fonction dans Excel pour vérifier que tout fonctionne.



Dans ce chapitre, je fais toujours référence à la modification d'un paramètre dans le complément ; cependant, tout ce que nous avons étudié au Chapitre 10 et qui concerne la configuration et le déploiement peut également être appliqué à ce chapitre. Cela signifie qu'un paramètre peut aussi être modifié dans la feuille xlwings.conf ou dans un fichier de configuration situé dans le même répertoire que le fichier Excel. Et au lieu d'utiliser le complément xlwings, vous pouvez utiliser un classeur qui a été configuré en mode autonome. Avec les UDF, il est également logique de créer votre propre complément personnalisé, ce qui vous permet de partager vos UDF entre tous les classeurs, sans avoir à les importer dans chaque classeur. Pour plus d'informations sur la création de votre propre complément personnalisé, consultez la documentation de xlwings[2].

Si vous modifiez le code Python de votre UDF, xlwings récupère automatiquement les modifications chaque fois que vous enregistrez le fichier Python. Comme nous l'avons déjà dit, vous n'avez besoin de réimporter vos UDF que si vous modifiez quelque chose dans le nom, les arguments ou les décorateurs de la fonction. Si, toutefois, votre fichier source importe du code à partir d'autres modules et que vous modifiez quelque chose dans ces modules, le moyen le plus simple de laisser Excel récupérer toutes les modifications consiste à cliquer sur le bouton Restart UDF Server.

À ce stade, vous savez comment écrire une UDF simple en Python et comment l'utiliser dans Excel. L'étude de cas de la section suivante présente des UDF plus intéressantes qui utilisent des DataFrames pandas.

Étude de cas : Google Trends

Dans cette étude de cas, nous utiliserons les données de Google Trends pour apprendre à travailler avec les DataFrames pandas et les tableaux dynamiques, l'une des nouvelles fonctionnalités les plus prometteuses d'Excel que Microsoft a officiellement lancée en 2020. Nous allons ensuite créer une UDF qui se connecte directement à Google Trends ainsi qu'une fonction qui utilise la méthode `plot` d'un DataFrame. Pour conclure cette section, nous verrons comment le débogage fonctionne avec les UDF. Commençons par une brève introduction à Google Trends !

Introduction à Google Trends

Google Trends[3] (*trends* est un terme anglais signifiant « tendances ») est un service Google qui permet d'analyser la popularité des requêtes de recherche au fil du temps et de faire des

2 https://docs.xlwings.org/en/stable/customaddin.html
3 https://trends.google.com/trends/?geo=FR

comparaisons entre différents pays. La Figure 12.4 affiche Google Trends après avoir ajouté quelques langages de programmation populaires, et sélectionné les cinq dernières années à l'échelle mondiale. Chaque terme de recherche a été sélectionné avec le contexte Langage de programmation ou Langage qui apparaît dans une liste déroulante après avoir saisi le terme de recherche. Cela nous permet d'ignorer les recherches sur les serpents de la famille des Pythonidae et celles sur l'île de Java. Google indexe les données sur la période et l'emplacement sélectionnés, la valeur 100 représentant l'intérêt maximal de la recherche. Dans notre exemple, cela signifie que dans les cinq dernières années, à l'échelle mondiale, l'intérêt de recherche le plus élevé était Java en octobre 2016. Pour plus de détails sur Google Trends, consultez cet article publié sur le blog officiel :

https://medium.com/google-news-lab/what-is-google-trends-data-and-what-does-it-mean-b48f07342ee8

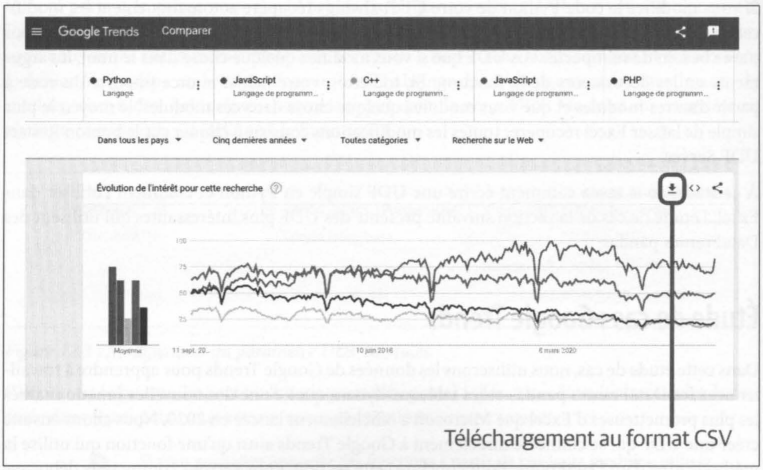


Figure 12.4 : Intérêt au fil du temps (source de données Google Trends).



Les chiffres de Google Trends sont basés sur des échantillons aléatoires, ce qui signifie que vous pouvez voir une image légèrement différente de celle illustrée à la Figure 12.4, même si vous utilisez le même emplacement, la même période et les mêmes termes de recherche que sur la capture d'écran.

J'ai cliqué sur le bouton de téléchargement qui est légendé sur la Figure 12.4 afin d'obtenir un fichier CSV à partir duquel j'ai copié les données dans le classeur Excel d'un projet généré avec

`quickstart`. Dans la section suivante, je vais vous montrer où trouver ce classeur que nous utiliserons pour analyser les données avec une UDF, directement à partir d'Excel.

Utilisation de DataFrames et de tableaux dynamiques

Si vous êtes arrivé jusqu'ici dans la lecture de cet ouvrage, vous ne devriez pas être surpris par le fait que les DataFrames pandas soient également les meilleurs amis des UDF. Pour voir comment DataFrames et UDF fonctionnent ensemble et pour en savoir plus sur les tableaux dynamiques, accédez au dossier *describe* du répertoire *udfs* des exemples de code et ouvrez *describe.xlsm* dans Excel et *describe.py* dans VS Code. Le fichier Excel contient des données de Google Trends et dans le fichier Python, vous trouverez une fonction simple pour commencer, qui est reproduite dans l'Exemple 12.2.

Exemple 12.2 : describe.py.

```
import xlwings as xw
import pandas as pd


@xw.func
@xw.arg("df", pd.DataFrame, index=True, header=True)
def describe(df):
    return df.describe()
```

Par rapport à la fonction `hello` du projet `quickstart`, vous remarquerez un deuxième décorateur :

```
@xw.arg("df", pd.DataFrame, index=True, header=True)
```

`arg`, qui est l'abréviation d'*argument*, permet d'appliquer les mêmes convertisseurs et options que ceux que j'ai employés lorsque j'ai présenté la syntaxe xlwings. En d'autres termes, le décorateur offre les mêmes fonctionnalités pour les UDF que la méthode options pour les objets `range` xlwings. Formellement, il s'agit de la syntaxe du décorateur `arg` :

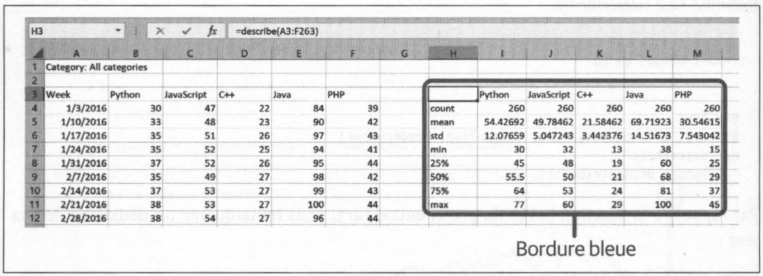
```
@xw.arg("argument_name", convert=None, option1=value1, option2=value2, ...)
```

Pour vous aider à faire le lien avec le Chapitre 9, l'équivalent de la fonction `describe` sous la forme d'un script ressemble à ceci (on suppose que *describe.xlsm* est ouvert dans Excel et que la fonction est appliquée à la plage A3:F263) :

```
import xlwings as xw
import pandas as pd


data_range = xw.Book("describe.xlsm").sheets[0]["A3:F263"]
df = data_range.options(pd.DataFrame, index=True, header=True).value
df.describe()
```

Les options `index` et `header` ne sont pas nécessaires car elles utilisent les arguments par défaut, mais je les ai incluses pour vous montrer comment elles sont appliquées avec les UDF. `describe.xlsm` étant le classeur actif, cliquez sur le bouton Import Functions, puis saisissez **`=describe(A3:F263)`** dans une cellule libre, par exemple H3. Ce qui se passe lorsque vous appuyez sur Entrée dépend de votre version d'Excel, et plus précisément, si votre version d'Excel est suffisamment récente pour prendre en charge les *tableaux dynamiques*. Si c'est le cas, vous verrez s'afficher le tableau illustré à la Figure 12.5, c'est-à-dire la sortie de la fonction `describe` dans les cellules H3:M11, entourée d'une fine bordure bleue. Vous ne pourrez voir la bordure bleue que si votre curseur se trouve dans le tableau, et elle est si légère que vous pourriez avoir des problèmes à la visualiser clairement si vous regardez la capture d'écran dans une version imprimée de ce livre. Nous verrons comment les tableaux dynamiques se comportent dans un instant et vous pourrez également en apprendre davantage à leur sujet dans l'encadré « Tableaux dynamiques » ci-dessous.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Category: All categories												
2													
3	Week	Python	JavaScript	C++	Java	PHP			Python	JavaScript	C++	Java	PHP
4	1/3/2016	30	47	22	84	39		count	260	260	260	260	260
5	1/10/2016	33	48	23	90	42		mean	54.42692	49.78462	21.58462	69.71923	30.54615
6	1/17/2016	35	51	26	97	43		std	12.07659	5.047243	3.442376	14.51673	7.543042
7	1/24/2016	35	52	25	94	41		min	30	32	13	38	15
8	1/31/2016	37	52	26	95	44		25%	45	48	19	60	25
9	2/7/2016	35	49	27	98	42		50%	55.5	50	21	68	29
10	2/14/2016	37	53	27	99	43		75%	64	53	24	81	37
11	2/21/2016	38	53	27	100	44		max	77	60	29	100	45
12	2/28/2016	38	54	27	96	44							

Figure 12 5 : Fonction `describe` avec des tableaux dynamiques.

Si, toutefois, vous utilisez une version d'Excel qui ne prend pas en charge les tableaux dynamiques, il ne se passera rien : par défaut, la formule ne renverra dans H3 que la cellule supérieure gauche, qui, dans notre exemple, est vide. Pour résoudre ce problème, utilisez ce que Microsoft appelle désormais des *tableaux CSE hérités* (également nommés formules matricielles). Les tableaux CSE doivent être validés en saisissant la combinaison de touches Ctrl + Maj + Entrée au lieu d'appuyer simplement sur Entrée, d'où leur nom (en anglais, la combinaison de touches correspond à Ctrl + Shift + Enter). Voyons leur fonctionnement en détail :

- Assurez-vous que H3 est une cellule vide en la sélectionnant et en appuyant sur la touche Suppr.
- Sélectionnez la plage de résultat en commençant par la cellule H3, puis étendez votre sélection jusqu'à la cellule M11.
- La plage H3:M11 étant sélectionnée, saisissez la formule `=describe(A3:F263)`, puis validez en appuyant sur Ctrl + Maj + Entrée.

Vous devriez maintenant voir presque la même image que celle qui est illustrée à la Figure 12.5, à ces différences près :

- Il n'y a pas de bordure bleue autour de la plage H3:M11.
- La formule est encadrée par des accolades pour indiquer qu'il s'agit d'un tableau CSE : `{=describe(A3:F263)}`.
- Alors que l'on supprime les tableaux dynamiques en sélectionnant la cellule supérieure gauche et en appuyant sur la touche Suppr, il faut sélectionner l'ensemble du tableau CSE pour pouvoir le supprimer.

Rendons maintenant notre fonction un peu plus utile en introduisant un paramètre facultatif appelé `selection` qui nous permettra de spécifier les colonnes que nous voulons inclure dans notre sortie. Si vous avez beaucoup de colonnes et que vous souhaitiez uniquement inclure un sous-ensemble dans la fonction `describe`, cela peut devenir une fonctionnalité utile. Modifiez la fonction de la manière suivante :

```
@xw.func
@xw.arg("df", pd.DataFrame) ❶
def describe(df, selection=None): ❷
    if selection is not None:
        return df.loc[:, selection].describe() ❸
    else:
        return df.describe()


        return df.describe()
```

❶ J'ai laissé de côté les arguments `index` et `header`, car ils utilisent les valeurs par défaut, mais n'hésitez pas à les laisser.

❷ Ajoutez le paramètre `selection` et rendez-le facultatif en affectant `None` comme valeur par défaut.

❸ Si le paramètre `selection` est fourni, on filtre les colonnes du DataFrame en fonction de sa valeur.

Une fois que vous avez modifié la fonction, assurez-vous de l'enregistrer, puis appuyez sur le bouton Import Functions du complément xlwings ; cela est nécessaire car nous avons ajouté un nouveau paramètre. Écrivez **`Selection`** dans la cellule A2 et **`TRUE`** dans les cellules B2:F2. Enfin, modifiez votre formule dans la cellule H3 selon la disponibilité des tableaux dynamiques dans votre version d'Excel :

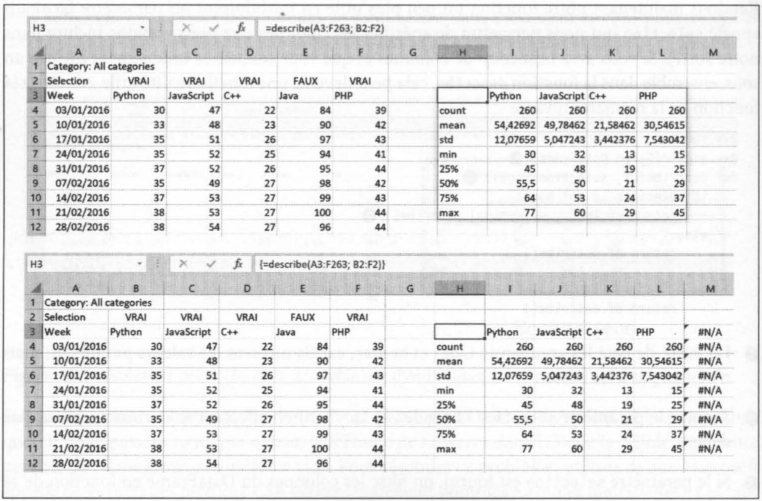
Avec des tableaux dynamiques

Sélectionnez H3, puis remplacez la formule par **`=describe(A3:F263; B2:F2)`** et appuyez sur Entrée.

Sans tableaux dynamiques

À partir de la cellule H3, sélectionnez H3:M11, puis appuyez sur F2 pour activer le mode d'édition de la cellule H3 et remplacez la formule par **`=describe(A3:F263; B2:F2)`**. Pour finaliser, appuyez sur Ctrl + Maj + Entrée.

Pour tester la fonction améliorée, changeons la valeur TRUE de Java dans la cellule E2 en FALSE et voyons ce qui se passe : avec les tableaux dynamiques, vous verrez le tableau se réduire comme par magie d'une colonne. Avec les tableaux CSE hérités, cependant, vous vous retrouverez avec une affreuse colonne pleine de valeurs `#N/A`, comme cela est illustré à la Figure 12.6.



H3 =describe(A3:F263; B2:F2)

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Category: All categories												
2	Selection	VRAI	VRAI	VRAI	FAUX	VRAI							
3	Week	Python	JavaScript	C++	Java	PHP			Python	JavaScript	C++	PHP	
4	03/01/2016	30	47	22	84	39		count	260	260	260	260	
5	10/01/2016	33	48	23	90	42		mean	54,42692	49,78462	21,58462	30,54615	
6	17/01/2016	35	51	26	97	43		std	12,07659	5,047243	3,442376	7,543042	
7	24/01/2016	35	52	25	94	41		min	30	32	13	15	
8	31/01/2016	37	52	26	95	44		25%	45	48	19	25	
9	07/02/2016	35	49	27	98	42		50%	55,5	50	21	29	
10	14/02/2016	37	53	27	99	43		75%	64	53	24	37	
11	21/02/2016	38	53	27	100	44		max	77	60	29	45	
12	28/02/2016	38	54	27	96	44							

H3 {=describe(A3:F263; B2:F2)}

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Category: All categories												
2	Selection	VRAI	VRAI	VRAI	FAUX	VRAI							
3	Week	Python	JavaScript	C++	Java	PHP			Python	JavaScript	C++	PHP	#N/A
4	03/01/2016	30	47	22	84	39		count	260	260	260	260	#N/A
5	10/01/2016	33	48	23	90	42		mean	54,42692	49,78462	21,58462	30,54615	#N/A
6	17/01/2016	35	51	26	97	43		std	12,07659	5,047243	3,442376	7,543042	#N/A
7	24/01/2016	35	52	25	94	41		min	30	32	13	15	#N/A
8	31/01/2016	37	52	26	95	44		25%	45	48	19	25	#N/A
9	07/02/2016	35	49	27	98	42		50%	55,5	50	21	29	#N/A
10	14/02/2016	37	53	27	99	43		75%	64	53	24	37	#N/A
11	21/02/2016	38	53	27	100	44		max	77	60	29	45	#N/A
12	28/02/2016	38	54	27	96	44							

Figure 12.6 : Tableaux dynamiques (en haut) et tableaux CSE (en bas) après exclusion d'une colonne.

Pour contourner ce problème, xlwings peut redimensionner les tableaux CSE en employant le décorateur de retour (`ret`). Ajoutez-le en modifiant la fonction de la manière suivante :

```
@xw.func
@xw.arg(«df», pd.DataFrame)
@xw.ret(expand=»table») ❶
def describe(df, selection=None):
    if selection is not None:
        return df.loc[:, selection].describe()
```

```
    else:
        return df.describe()
```

❶ En ajoutant le décorateur de retour avec l'option `expand="table"`, xlwings redimensionne le tableau CSE pour qu'il corresponde aux dimensions du DataFrame renvoyé.

Après avoir ajouté le décorateur de retour, enregistrez le fichier source Python, basculez vers Excel et appuyez sur Ctrl + Alt + F9 pour recalculer les formules : cela redimensionnera le tableau CSE et supprimera la colonne `#N/A`. Comme il s'agit d'une solution de contournement, je vous recommande fortement de faire tout ce qui est en votre pouvoir pour mettre la main sur une version d'Excel qui prenne en charge les tableaux dynamiques.



Assurez-vous de placer le décorateur `xw.func` au-dessus des décorateurs `xw.arg` et `xw.ret` ; notez que l'ordre des décorateurs `xw.arg` et `xw.ret` n'a pas d'importance.

Le *décorateur* de retour fonctionne conceptuellement de la même manière que le décorateur `arg`, à la seule différence que vous n'avez pas à spécifier le nom d'un argument. Formellement, sa syntaxe ressemble à ceci :

```
@xw.ret(convert=None, option1=value1, option2=value2, ...)
```

Vous n'avez généralement pas besoin de fournir un argument `convert` explicite car xlwings reconnaît automatiquement le type de la valeur renvoyée ; c'est le même comportement que nous avons vu au Chapitre 9 avec la méthode `options` lors de l'écriture de valeurs dans Excel.

Par exemple, si vous souhaitez supprimer l'index du DataFrame que vous renvoyez, utilisez ce décorateur :

```
@xw.ret(index=False)
```

Le téléchargement d'un fichier CSV et le copier-coller des valeurs dans un fichier Excel ont bien fonctionné pour cet exemple introductif avec un DataFrame, mais le copier-coller est un processus qui engendre tellement d'erreurs qu'il est souhaitable de s'en débarrasser chaque fois que c'est possible. Avec Google Trends, nous allons voir comment procéder.

Tableaux dynamiques

Après avoir vu comment les tableaux dynamiques fonctionnent dans le contexte de la fonction `describe`, je suis à peu près sûr que vous conviendrez qu'ils sont l'un des ajouts les plus fondamentaux et les plus intéressants que Microsoft a mis au point depuis longtemps. Cette nouvelle fonctionnalité a été officiellement présentée en 2020 aux abonnés Microsoft 365 qui utilisent la version la plus récente d'Excel. Pour voir si votre version est assez récente, vérifiez l'existence de la nouvelle fonction `UNIQUE` : commencez à saisir `=UNIQUE` dans une cellule et si Excel suggère le nom de la fonction, cela signifie que les tableaux dynamiques sont pris en charge. Si vous utilisez Excel avec une licence permanente plutôt que dans le cadre de l'abonnement Microsoft 365, vous êtes susceptible de bénéficier de cette fonctionnalité dans la version qui a été annoncée pour 2021 et qui s'appellera Office 2021. Voici quelques notes techniques sur le comportement des tableaux dynamiques :

- Si les tableaux dynamiques remplacent une cellule par une valeur, vous obtiendrez une erreur `#SPILL!`. Après avoir fait de la place pour le tableau dynamique en supprimant ou en déplaçant la cellule qui gêne, le tableau sera écrit. Notez que le décorateur de retour xlwings avec `expand="table"` est moins intelligent et écrasera les valeurs des cellules existantes sans avertissement.
- Vous pouvez faire référence à la plage d'un tableau dynamique à l'aide de la cellule en haut à gauche suivie d'un signe `#`. Par exemple, si votre tableau dynamique se trouve dans la plage A1:B2 et que vous souhaitiez faire la somme de toutes les cellules, écrivez `=SOMME(A1#)`.
- Si jamais vous souhaitez que vos tableaux se comportent à nouveau comme les tableaux CSE hérités, commencez votre formule par un signe `@`, par exemple, pour qu'une multiplication matricielle renvoie un tableau CSE hérité, utilisez `=@PRODUITMAT()`.

Récupération de données à partir de Google Trends

Les exemples précédents étaient tous très simples, se contentant d'intégrer une seule fonction pandas. Pour appréhender un cas plus réel, nous allons créer une UDF qui télécharge les données directement à partir de Google Trends, afin que vous n'ayez plus à aller en ligne et à télécharger un fichier CSV manuellement. Google Trends n'a pas d'API (interface de programmation d'applications) officielle, mais il existe un package Python appelé pytrends[4] qui comble cette lacune. Le fait qu'il ne s'agisse pas d'une API officielle signifie que Google peut la modifier quand il le souhaite, et il y a donc un risque que les exemples de cette section cessent de fonctionner à tout moment… Cependant, étant donné que pytrends existe depuis plus de cinq ans, il y a également de fortes chances qu'il soit mis à jour pour refléter les changements et qu'il soit à nouveau rapidement opérationnel. Dans tous les cas, c'est un bon exemple pour vous montrer qu'il existe un package Python pour à peu près n'importe quelle tâche, comme je vous l'avais affirmé dans le

4 https://github.com/GeneralMills/pytrends

Chapitre 1. Si vous étiez limité à l'utilisation de Power Query, vous auriez probablement besoin d'investir beaucoup plus de temps pour créer quelque chose qui fonctionne (je n'ai d'ailleurs pas été en mesure de trouver une solution prête à l'emploi et disponible gratuitement). Puisque pytrends ne fait pas partie d'Anaconda et n'a pas non plus de package officiel Conda, nous allons l'installer avec pip, si vous ne l'avez pas encore fait :

```
(base)> pip install pytrends
```

Pour reproduire exactement l'exemple de la version en ligne de Google Trends (voir la Figure 12.4), nous devrons trouver les identifiants corrects pour les termes de recherche avec le contexte « Langage de programmation ». Pour ce faire, pytrends peut afficher les différents contextes de recherche ou *types* que Google Trends suggère dans la liste déroulante. Dans l'exemple de code suivant, `mid` signifie *Machine ID*, qui est l'ID que nous recherchons :

```
In [4]: from pytrends.request import TrendReq
In [5]: # On commence par instancier un objet TrendRequest
        trend = TrendReq()
In [6]: # On peut afficher les suggestions comme elles apparaissent en ligne
        # dans la liste déroulante de Google Trends après la saisie de "Python"
        trend.suggestions("Python")
Out[6]: [{'mid': '/m/05z1_', 'title': 'Python', 'type': 'Programming language'},
         {'mid': '/m/05tb5', 'title': 'Python family', 'type': 'Snake'},
         {'mid': '/m/0cv6_m', 'title': 'Pythons', 'type': 'Snake'},
         {'mid': '/m/0l8ry', 'title': 'Python', 'type': 'Topic'},
         {'mid': '/m/0m5qz', 'title': 'Reticulated python', 'type': 'Snake'}]
```

En répétant cette opération pour les autres langages de programmation, cela nous permet de récupérer le bon `mid` pour chacun d'entre eux, et nous pouvons écrire l'UDF qui est reproduite dans l'Exemple 12.3. Vous trouverez le code source dans le répertoire *google_trends* du dossier *udfs* des exemples de code.

Exemple 12.3 : Fonction get_interest_over_time de google_trends.py (extrait avec les instructions d'importation pertinentes).

```
import pandas as pd
from pytrends.request import TrendReq
import xlwings as xw


@xw.func(call_in_wizard=False) ❶
@xw.arg("mids", doc="Machine IDs: A range of max 5 cells") ❷
@xw.arg("start_date", doc="A date-formatted cell")
@xw.arg("end_date", doc="A date-formatted cell")
def get_interest_over_time(mids, start_date, end_date):
    """Requête Google Trends - remplace le Machine ID (mid) des
    langages de programmation courants par un équivalent lisible
    dans la valeur de retour, par exemple au lieu de "/m/05z1_"
    on renvoie "Python".
    """ ❸
```

```
    # Vérifie et transforme les paramètres
    assert len(mids) <= 5, "Too many mids (max: 5)" ❹
    start_date = start_date.date().isoformat() ❺
    end_date = end_date.date().isoformat()

    # Fait la requête Google Trends et renvoie le DataFrame
    trend = TrendReq(timeout=10) ❻
    trend.build_payload(kw_list=mids,
                        timeframe=f"{start_date} {end_date}") ❼
    df = trend.interest_over_time() ❽

    # Remplace le mid de Google par un mot intelligible
    mids = {"/m/05z1_": "Python", "/m/02p97": "JavaScript",
            "/m/0jgqg": "C++", "/m/07sbkfb": "Java", "/m/060kv": "PHP"}
    df = df.rename(columns=mids) ❾

    # Supprime la colonne isPartial
    return df.drop(columns="isPartial") ❿
```

❶ Par défaut, Excel appelle la fonction lorsque vous l'ouvrez dans l'Assistant Fonction. Comme cela peut ralentir le processus, en particulier avec les requêtes d'API impliquées, nous désactivons cela.

❷ Vous pouvez ajouter une docstring pour l'argument de fonction, qui sera affichée dans l'Assistant Fonction lorsque vous modifiez l'argument en question (voir la Figure 12.8).

❸ La docstring de la fonction s'affiche dans l'Assistant Fonction (voir la Figure 12.8).

❹ L'instruction `assert` est un moyen facile de déclencher une erreur au cas où l'utilisateur fournirait trop de `mids`. Google Trends autorise un maximum de cinq `mids` par requête.

❺ pytrends attend les dates de début et de fin dans une chaîne au format `AAAA-MM-JJ AAAA-MM-JJ`. Comme nous fournissons les dates de début et de fin dans ces cellules au format date, elles sont transmises en tant qu'objets `datetime`. En appelant les méthodes `date` et `isoformat`, on les met en forme correctement.

❻ Nous instancions un objet `request` pytrends. En définissant le `timeout` à dix secondes, nous réduisons le risque de voir apparaître une erreur `requests.exceptions.ReadTimeout`, qui se produit parfois si Google Trends prend un peu trop de temps à répondre. Si cette erreur persiste, exécutez simplement la fonction à nouveau ou augmentez le timeout.

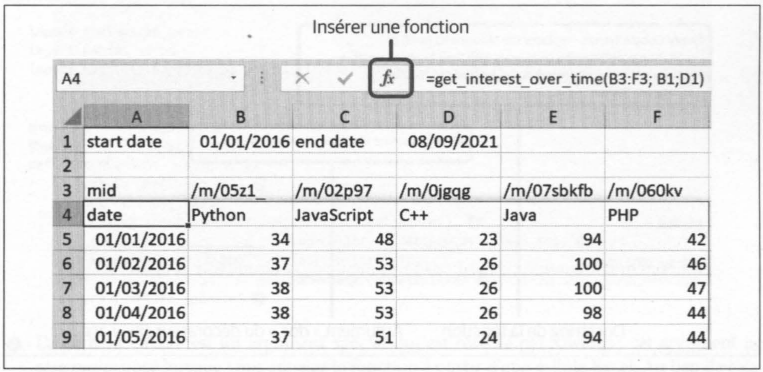
❼ Nous fournissons les arguments `kw_list` et `timeframe` à l'objet `request`.

❽ Nous effectuons la requête en appelant `interest_over_time`, qui renverra un DataFrame pandas.

❾ Nous renommons les `mids` avec leur équivalent intelligible.

⑩ La dernière colonne s'appelle `isPartial`. `True` indique que l'intervalle actuel (par exemple une semaine), est toujours en cours et ne dispose donc pas encore de toutes les données. Pour que les choses restent simples et être en phase avec la version en ligne, nous supprimons cette colonne lors du retour du DataFrame.

Ouvrez maintenant *google_trends.xlsm* à partir des exemples de code, cliquez sur Import Functions dans le complément xlwings, puis appelez la fonction `get_interest_over_time` à partir de la cellule A4 (Figure 12.7).



	A	B	C	D	E	F
1	start date	01/01/2016	end date	08/09/2021		
2						
3	mid	/m/05z1_	/m/02p97	/m/0jgqg	/m/07sbkfb	/m/060kv
4	date	Python	JavaScript	C++	Java	PHP
5	01/01/2016	34	48	23	94	42
6	01/02/2016	37	53	26	100	46
7	01/03/2016	38	53	26	100	47
8	01/04/2016	38	53	26	98	44
9	01/05/2016	37	51	24	94	44

Figure 12.7 : Classeur google_trends.xlsm.

Pour obtenir de l'aide sur les arguments de fonction, cliquez sur le bouton Insérer une fonction à gauche de la barre de formule, quand la cellule A4 est sélectionnée : cela ouvrira l'Assistant Fonction où vous trouverez vos UDF sous la catégorie xlwings. Après avoir sélectionné `get_interest_over_time`, vous verrez le nom des arguments de la fonction ainsi que la docstring qui décrit la fonction et qui est limitée à 256 caractères (Figure 12.8). Vous pouvez également commencer à saisir **`=get_interest_over_time(`** dans la cellule A4 (y compris la parenthèse ouvrante) avant d'appuyer sur le bouton Insérer une fonction, ce qui vous mènera directement à la boîte de dialogue illustrée à la Figure 12.8. Notez que les UDF renvoient les dates non formatées. Pour résoudre ce problème, cliquez avec le bouton droit sur la colonne contenant les dates, sélectionnez Format de cellule, puis sélectionnez le format de votre choix dans la catégorie Date.

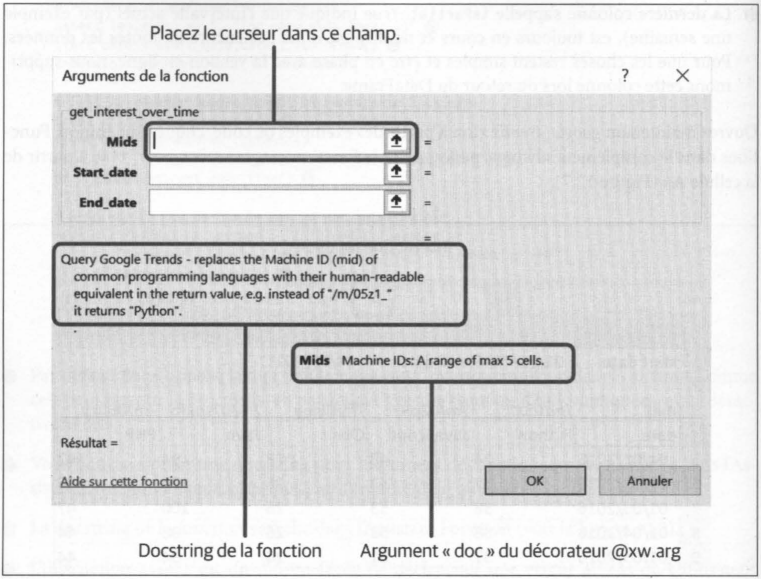


Figure 12.8 : Assistant Fonction.

Si vous regardez attentivement la Figure 12.7, vous pouvez déduire de la bordure bleue autour du tableau de résultats que j'utilise à nouveau des tableaux dynamiques. Comme la capture d'écran est recadrée en bas et que le tableau commence tout à gauche, vous ne voyez que les bordures supérieure et droite commençant à la cellule A4, qui peuvent elles-mêmes être difficiles à identifier sur la copie d'écran. Si votre version d'Excel ne prend pas en charge les tableaux dynamiques, utilisez la solution de contournement en ajoutant le décorateur de retour suivant à la fonction `get_interest_over_time` (sous les décorateurs existants) :

```
@xw.ret(expand="table")
```

Maintenant que vous savez comment travailler avec des UDF plus complexes, voyons comment nous pouvons utiliser des tracés avec des UDF !

Traçage avec des UDF

Au Chapitre 5, nous avons vu que l'appel de la méthode `plot` d'un DataFrame renvoie par défaut un tracé Matplotlib. Dans le Chapitre 9 et le Chapitre 11, nous avons déjà vu comment ajouter un graphique sous la forme d'une image. Lorsque vous travaillez avec des UDF, il existe un moyen simple de produire des tracés : examinez la deuxième fonction dans *google_trends.py*, qui est reproduite dans l'Exemple 12.4.

Exemple 12.4 : Fonction plot dans google_trends.py (extrait avec les instructions d'importation pertinentes).

```
import xlwings as xw
import pandas as pd
import matplotlib.pyplot as plt


@xw.func
@xw.arg("df", pd.DataFrame)
def plot(df, name, caller): ❶
    plt.style.use("seaborn") ❷
    if not df.empty: ❸
        caller.sheet.pictures.add(df.plot().get_figure(), ❹
                                  top=caller.offset(row_offset=1).top, ❺
                                  left=caller.left,
                                  name=name, update=True) ❻
    return f"<Plot: {name}>" ❼
```

❶ L'argument `caller` est un argument spécial qui est réservé par xlwings : cet argument ne sera pas exposé lorsque vous appelez la fonction à partir d'une cellule Excel. Au lieu de cela, `caller` sera fourni par xlwings en coulisses et il correspond à la cellule à partir de laquelle vous appelez la fonction (sous la forme d'un objet `range` xlwings). Le fait d'avoir l'objet `range` de la cellule appelante facilite le placement du tracé en utilisant les arguments `top` et `left` de `pictures.add`. L'argument `name` définit le nom de l'image dans Excel.

❷ Nous avons défini le style `seaborn` pour rendre le graphique visuellement plus attrayant.

❸ N'appelez la méthode `plot` que si le DataFrame n'est pas vide. L'appel de la méthode `plot` sur un DataFrame vide déclencherait une erreur.

❹ `get_figure()` renvoie l'objet `figure` Matplotlib à partir d'un tracé de DataFrame, ce qui est l'objet attendu par `pictures.add`.

❺ Les arguments `top` et `left` ne sont utilisés que lorsque vous insérez le tracé pour la première fois. Les arguments fournis placeront le tracé dans un endroit qui convient, une cellule en dessous de celle à partir de laquelle vous appelez cette fonction.

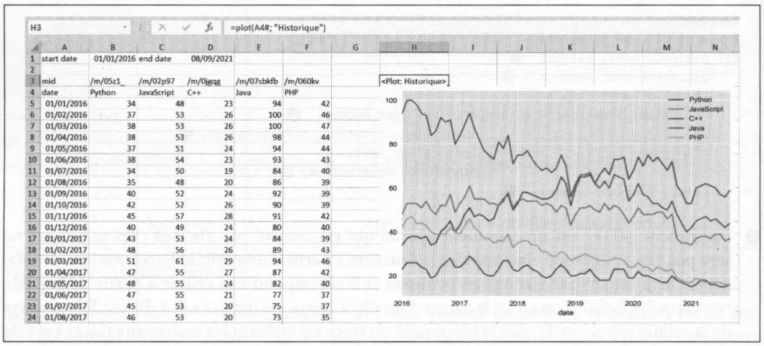
❻ L'argument `update=True` garantit que les appels de fonction répétés mettront à jour l'image existante avec le nom fourni dans Excel, sans modifier sa position ni sa taille. Sans cet argument, xlwings se plaindra qu'il existe déjà une image avec ce nom dans Excel.

❼ Bien que vous n'ayez pas vraiment besoin de retourner quoi que ce soit, cela vous facilitera grandement la vie si vous retournez une chaîne, car vous saurez où se trouve votre fonction de traçage dans la feuille.

Dans *google_trends.xlsm*, dans la cellule H3, appelez la fonction `plot` de la manière suivante :

```
=plot(A4:F263; "Historique")
```

Si votre version d'Excel prend en charge les tableaux dynamiques, utilisez `A4#` au lieu de `A4:F263` pour rendre la source dynamique comme cela est illustré à la Figure 12.9.



	A	B	C	D	E	F
1	start date	01/01/2016	end date	08/09/2021		
2						
3	mid	/m/05z1_	/m/02p97	/m/0jgqg	/m/07sbkfb	/m/060kv
4	date	Python	JavaScript	C++	Java	PHP
5	01/01/2016	34	48	23	94	42
6	01/02/2016	37	53	26	100	46
7	01/03/2016	38	53	26	100	47
8	01/04/2016	38	53	26	98	44
9	01/05/2016	37	51	24	94	44
10	01/06/2016	38	54	23	93	43
11	01/07/2016	34	50	19	84	40
12	01/08/2016	35	48	20	86	39
13	01/09/2016	40	52	24	92	39
14	01/10/2016	42	57	26	90	39
15	01/11/2016	45	57	28	91	42
16	01/12/2016	40	49	24	80	40
17	01/01/2017	43	53	24	84	39
18	01/02/2017	48	56	26	89	43
19	01/03/2017	51	61	29	94	46
20	01/04/2017	47	55	27	87	42
21	01/05/2017	48	55	24	82	40
22	01/06/2017	47	55	21	77	37
23	01/07/2017	45	53	20	75	37
24	01/08/2017	46	53	20	73	35

Figure 12.9 : Fonction plot en action.

Si vous n'arrivez pas très bien à comprendre le fonctionnement de la fonction `get_interest_over_time`, le plus simple est de déboguer le code de l'UDF.

Débogage des UDF

Un moyen simple de déboguer une UDF consiste à utiliser la fonction `print`. Si le paramètre Show Console est activé dans le complément xlwings, vous pouvez afficher la valeur d'une variable dans l'invite de commandes qui apparaît lorsque vous appelez votre UDF. Il est encore plus confortable d'utiliser le débogueur de VS Code, qui permet de faire une pause grâce aux points d'arrêt et de parcourir le code ligne à ligne. Pour utiliser le débogueur de VS Code (ou le débogueur de tout autre IDE), vous devez effectuer deux opérations :

1. Dans le complément Excel, activez la case à cocher Debug UDFs. Cela empêche Excel de démarrer automatiquement Python, ce qui signifie que vous devez le faire manuellement, comme cela est expliqué au point suivant.
2. Le moyen le plus simple d'exécuter manuellement le serveur UDF de Python consiste à ajouter les lignes suivantes tout en bas du fichier que vous voulez déboguer. J'ai déjà ajouté ces lignes au bas du fichier *google_trends.py* dans les exemples de code :

```
if __name__ == "__main__":
    xw.serve()
```

   Au Chapitre 11, nous avons vu que cette instruction `if` garantit que le code ne s'exécute que lorsque vous exécutez le fichier en tant que script (le code ne s'exécute pas lorsque vous l'importez en tant que module). La commande `serve` ayant été ajoutée, exécutez *google_trends.py* dans VS Code en mode débogage en appuyant sur F5 et en sélectionnant « Python File » ; assurez-vous de ne pas exécuter le fichier en cliquant sur le bouton Exécuter le script, car cela ignorerait les points d'arrêt.

Définissons un point d'arrêt sur la ligne 29 en cliquant à gauche du numéro de ligne. Si vous n'êtes pas familier avec l'utilisation du débogueur de VS Code, veuillez consulter l'Annexe B, où je le présente plus en détail. Lorsque vous recalculez maintenant la cellule A4, l'appel de fonction s'arrête sur le point d'arrêt et vous pouvez inspecter les variables. Il est toujours utile pendant le débogage d'exécuter `df.info()`. Activez l'onglet Console de débogage, saisissez **`df.info()`** dans l'invite en bas et validez en appuyant sur Entrée, comme cela est illustré à la Figure 12.10.



Il s'agit du même avertissement que celui que j'ai mentionné au Chapitre 11 : sous Windows, lorsque vous exécutez le débogueur de VS Code pour la première fois avec du code qui utilise pandas, vous pouvez être accueilli par l'erreur suivante : « Exception has occurred: ImportError, Unable to import required dependencies: numpy ». Cela se produit car le débogueur est opérationnel avant que l'environnement Conda n'ait été activé correctement. Pour contourner le problème, arrêtez le débogueur en cliquant sur l'icône d'arrêt et appuyez à nouveau sur F5 (la deuxième fois sera la bonne).

Si vous maintenez votre programme en pause pendant plus de quatre-vingt-dix secondes sur un point d'arrêt, Excel affiche une boîte de dialogue indiquant que « Microsoft Excel attend la fin de l'exécution d'une action OLE d'une autre application ». Cela ne devrait pas avoir d'autre impact sur votre expérience de débogage que d'avoir à cliquer sur le bouton OK de la boîte de dialogue pour la faire disparaître une fois que vous avez terminé le débogage. Pour terminer cette session de débogage, cliquez sur le bouton Arrêter de la barre d'outils de débogage de VS Code (voir la Figure 12.10) et assurez-vous de décocher la case Debug UDFs du complément xlwings. Si vous oubliez de décocher ce paramètre, vos fonctions renverront une erreur la prochaine fois que vous les recalculerez.

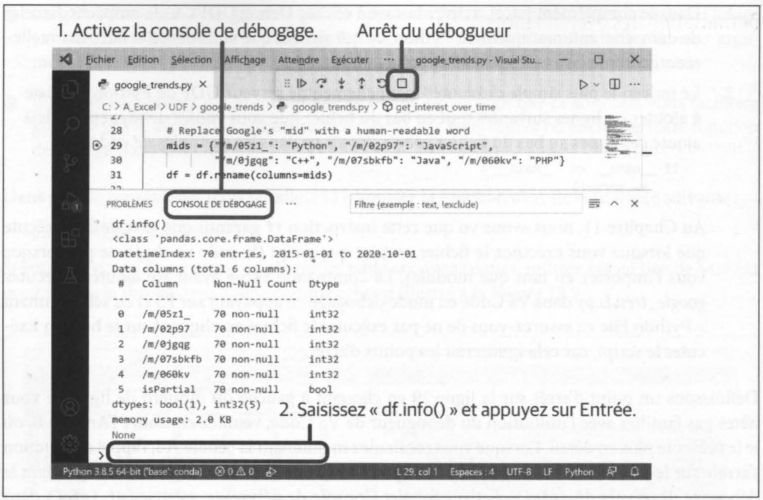


Figure 12.10 : Utilisation de la console de débogage pendant que le code est mis en pause sur un point d'arrêt.

En prenant l'exemple de l'étude de cas sur Google Trends, cette section vous a montré les fonctionnalités relatives aux UDF les plus couramment utilisées. La section suivante abordera quelques sujets avancés, notamment les performances des UDF et le décorateur `xw.sub`.

Sujets avancés sur les UDF

Si vous utilisez de nombreuses UDF dans un classeur, cela peut poser des problèmes de performances. Cette section commence par vous montrer les mêmes optimisations de performances de base que nous avons vues au Chapitre 9, mais appliquées aux UDF. La deuxième partie traite de la mise en cache, une technique supplémentaire d'optimisation des performances que nous pouvons utiliser avec les UDF. En cours de route, nous apprendrons également à faire en sorte que les arguments des fonctions soient transmis en tant qu'objets `range` xlwings plutôt que sous la forme de valeurs. À la fin de cette section, je vous présenterai le décorateur `xw.sub` que vous pouvez utiliser comme alternative à l'appel `RunPython` si vous travaillez exclusivement sous Windows.

Optimisation des performances de base

Cette partie examine deux techniques d'optimisation des performances : la minimisation des appels interapplications et l'emploi du convertisseur de valeurs brutes.

Minimiser les appels interapplications

Comme nous l'avons vu au Chapitre 9, les appels interapplications, c'est-à-dire les appels entre Excel et Python, sont relativement lents, si bien qu'il est préférable d'avoir le minimum d'UDF. Vous devez donc travailler avec des tableaux dynamiques chaque fois que vous le pouvez, et le fait d'avoir une version d'Excel prenant en charge cette fonctionnalité facilite assurément les choses. Lorsque vous travaillez avec des DataFrames pandas, il n'y a pas grand-chose qui puisse mal tourner, mais il existe certaines formules où vous pourriez ne pas penser à utiliser spontanément des tableaux dynamiques. Considérons l'exemple de la Figure 12.11 qui calcule les recettes totales comme étant la somme d'un tarif de base donné et d'un tarif variable déterminé par le nombre d'utilisateurs multiplié par le prix.

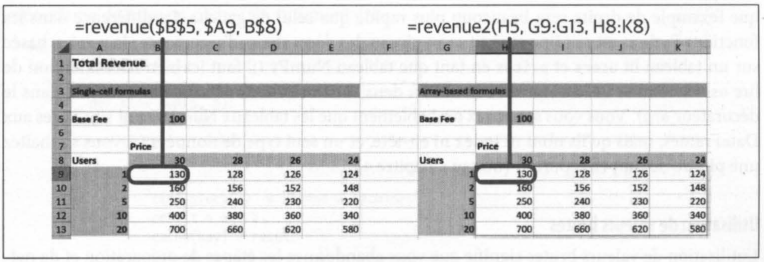


Figure 12.11 : Formules à cellule unique (à gauche) et formules basées sur un tableau (à droite).

Formules à cellule unique

Le tableau de gauche de la Figure 12.11 utilise la formule `=revenue($B$5; $A9;B$8)` dans la cellule B9. Cette formule est ensuite appliquée à toute la plage B9:E13. Cela signifie que vous avez 20 formules à cellule unique qui appellent la fonction `revenue`.

Formules basées sur des tableaux

Le tableau de droite de la Figure 12.11 utilise la formule `=revenue2(H5; G9:G13; H8:K8)`. Si votre version d'Excel ne prend pas en charge les tableaux dynamiques, vous devez ajouter le décorateur `xw.ret(expand="table")` à la fonction `revenue2` ou transformer le tableau en un tableau CSE hérité en sélectionnant H9:K13, en appuyant sur F2 pour modifier la formule et en validant avec Ctrl + Maj + Entrée. Contrairement à la formule à cellule unique, cette version n'appelle la fonction `revenue2` qu'une seule fois.

Vous pouvez voir le code Python des deux UDF dans l'Exemple 12.5, et vous trouverez le fichier source dans le dossier *revenues* du répertoire *udfs* des exemples de code.

Exemple 12.5 : revenues.py.

```
import numpy as np
import xlwings as xw


@xw.func
def revenue(base_fee, users, price):
    return base_fee + users * price


@xw.func
@xw.arg("users", np.array, ndim=2)
@xw.arg("price", np.array)
def revenue2(base_fee, users, price):
    return base_fee + users * price
```

Lorsque vous modifiez les frais de base dans la cellule B5 ou dans la cellule H5, vous verrez que l'exemple de droite sera beaucoup plus rapide que celui de gauche. La différence dans les fonctions Python est minime et se situe au niveau des décorateurs d'arguments : la version basée sur un tableau lit `users` et `prices` en tant que tableau NumPy (il faut ici bien faire attention de lire `users` comme un vecteur de colonnes à deux dimensions en définissant `ndim = 2` dans le décorateur `arg`). Vous vous souvenez probablement que les tableaux NumPy sont similaires aux DataFrames, mais qu'ils n'ont ni index ni en-tête, et un seul type de données (si vous souhaitez une piqûre de rappel, reportez-vous au Chapitre 4).

Utilisation de valeurs brutes

L'utilisation de valeurs brutes signifie que vous abandonnez les étapes de préparation et de nettoyage des données effectuées par xlwings, au-dessus de la couche pywin32, la dépendance de xlwings à Windows. Ceci, par exemple, signifie que vous ne pouvez plus travailler directement avec des DataFrames, car pywin32 ne les comprend pas, mais cela peut ne pas être un problème si vous travaillez avec des listes ou des tableaux NumPy. Pour employer des UDF avec des valeurs brutes, utilisez la chaîne `raw` comme argument `convert` dans le décorateur `arg` ou `ret`. C'est l'équivalent de l'utilisation du convertisseur `raw` *via* la méthode `options` d'un objet `range` xlwings, comme nous l'avons fait au Chapitre 9. Conformément à ce que nous avions remarqué dans ce chapitre, on obtient la plus grande accélération pendant les opérations d'écriture. Par exemple, appeler la fonction suivante sans le décorateur `ret` est environ trois fois plus lent sur mon ordinateur portable :

```
import numpy as np
import xlwings as xw
@xw.func
@xw.ret("raw")
```

```
def randn(i=1000, j=1000):
    """Retourne un tableau de dimensions (i, j) avec des nombres aléatoires
    normalement distribués et fournis par random.randn de NumPy
    """
    return np.random.randn(i, j)
```

Vous trouverez ce code dans le dossier *raw_values* du répertoire *udfs* des exemples de code. Lorsque vous travaillez avec des UDF, vous disposez d'une autre option simple pour améliorer les performances : empêcher les calculs répétés de fonctions lentes en mettant en cache leurs résultats.

Mise en cache

Lorsque vous appelez une fonction *déterministe*, c'est-à-dire une fonction qui, avec les mêmes entrées, renvoie toujours la même sortie, vous pouvez stocker le résultat dans un *cache* : ainsi les appels répétés de la fonction n'ont plus besoin d'attendre un calcul éventuellement lent, mais peuvent prendre le résultat dans le cache où il a déjà été précalculé. Un court exemple vous permettra de mieux comprendre ce concept. On peut programmer avec un dictionnaire un mécanisme de mise en cache très rudimentaire :

```
In [7]: import time
In [8]: cache = {}

        def slow_sum(a, b):
            key = (a, b)
            if key in cache:
                return cache[key]
            else:
                time.sleep(2)  # attend 2 secondes
                result = a + b
                cache[key] = result
                return result
```

Lorsque vous appelez cette fonction pour la première fois, la variable `cache` est vide. Le code exécutera donc la clause `else` avec la pause artificielle de deux secondes qui imite un calcul lent. Après avoir effectué le calcul, il ajoutera le résultat au dictionnaire `cache` avant de renvoyer le résultat. Lorsque vous appelez maintenant cette fonction une deuxième fois avec les mêmes arguments et au cours de la même session Python, elle trouvera le résultat dans le cache et le retournera immédiatement, sans avoir à effectuer à nouveau un calcul lent. La mise en cache d'un résultat en fonction de ses arguments est également appelée *mémoïsation*. Vous verrez donc la différence de temps lorsque vous appelez la fonction pour la première et la deuxième fois :

```
In [9]: %%time
        slow_sum(1, 2)
Wall time: 2.01 s
Out[9]: 3
In [10]: %%time
        slow_sum(1, 2)
```

```
Wall time: 0 ns
Out[10]: 3
```

Python a un décorateur intégré appelé `lru_cache` qui peut vous faciliter la vie et que vous importez à partir du module `functools` qui fait partie de la bibliothèque standard. lru est l'abréviation de *least recent used* (moins récemment utilisé), ce qui signifie que le cache contient un nombre maximum de résultats (par défaut 128) avant de se débarrasser des plus anciens. Nous pouvons l'utiliser avec notre exemple Google Trends de la section précédente. Tant que nous n'interrogeons que les valeurs de l'historique, nous pouvons mettre en cache le résultat en toute sécurité. Cela permettra non seulement d'accélérer les appels multiples, mais cela réduira également le nombre de requêtes que nous envoyons à Google, réduisant ainsi le risque que Google nous bloque, ce qui peut se produire si vous envoyez trop de requêtes en un temps réduit.

Voici les premières lignes de la fonction `get_interest_over_time` avec les modifications requises pour appliquer la mise en cache :

```
from functools import lru_cache ❶

import pandas as pd
from pytrends.request import TrendReq
import matplotlib.pyplot as plt
import xlwings as xw

@lru_cache ❷
@xw.func(call_in_wizard=False)
@xw.arg("mids", xw.Range, doc="Machine IDs: A range of max 5 cells") ❸
@xw.arg("start_date", doc="A date-formatted cell")
@xw.arg("end_date", doc="A date-formatted cell")
def get_interest_over_time(mids, start_date, end_date):
    """Query Google Trends - remplace le Machine ID (mid) des
    langages de programmation courants par un équivalent lisible
    dans la valeur de retour, par exemple au lieu de "/m/05z1_"
    il renvoie "Python".
    """
    mids = mids.value ❹
```

❶ Importe le décorateur `lru_cache`.

❷ Utilise le décorateur. Le décorateur doit être au-dessus du décorateur `xw.func`.

❸ Par défaut, `mids` est une liste. Cela crée un problème dans ce cas, car les fonctions avec des listes passées en tant qu'arguments ne peuvent pas être mises en cache. Le problème sous-jacent est que les listes sont des objets mutables qui ne peuvent pas être utilisés comme clés dans les dictionnaires (pour plus d'informations sur les objets mutables et immuables, consultez l'Annexe C). L'emploi du convertisseur `xw.Range` permet de récupérer `mids` en tant qu'objet `range` xlwings plutôt que sous forme de liste, ce qui résout notre problème.

❹ Pour que le reste du code fonctionne à nouveau, nous devons maintenant obtenir les valeurs *via* la propriété `value` de l'objet `range` xlwings.



Si vous utilisez une version de Python inférieure à 3.8, vous devrez utiliser le décorateur avec des parenthèses de la manière suivante : `@lru_cache()`. Si vous utilisez Python 3.9 ou une version ultérieure, remplacez `@lru_cache` par `@cache`, qui est identique à `@lru_cache(maxsize=None)`, ce qui signifie que le cache ne se débarrasse jamais des anciennes valeurs. Vous devez également importer le décorateur `cache` à partir de `functools`.

Le convertisseur `xw.Range` peut également être utile dans d'autres circonstances, par exemple, si vous devez dans votre UDF accéder aux formules plutôt qu'aux valeurs des cellules. Dans l'exemple précédent, vous pouvez écrire `mids.formula` pour accéder aux formules des cellules. Vous trouverez l'exemple complet dans le dossier *google_trends_cache* du répertoire *udfs* des exemples de code.

Maintenant que vous savez optimiser les performances des UDF, terminons cette section en présentant le décorateur `xw.sub`.

Décorateur sub

Au Chapitre 10, je vous ai montré comment accélérer l'appel `RunPython` en activant le paramètre Use UDF Server. Si vous travaillez exclusivement sous Windows, il existe une alternative à la combinaison `RunPython/Use UDF Server` sous la forme du décorateur `xw.sub`. Cela vous permet d'importer vos fonctions Python en tant que procédures Sub dans Excel, sans avoir à écrire manuellement des appels `RunPython`. Dans Excel, vous aurez besoin d'une procédure Sub pour pouvoir attacher votre fonction à un bouton (une fonction Excel, telle que vous l'obtenez lorsque vous utilisez le décorateur `xw.func`, ne fonctionnera pas). Pour tester cela, nous allons créer un nouveau projet `quickstart` appelé `importsub`. Comme d'habitude, assurez-vous de vous positionner d'abord dans le répertoire où vous souhaitez que le projet soit créé, à l'aide de la commande `cd` :

```
(base)> xlwings quickstart importsub
```

Dans l'Explorateur de fichiers, accédez au dossier *importsub* créé et ouvrez *importsub.xlsm* dans Excel et *importsub.py* dans VS Code, puis ajoutez le décorateur `@xw.sub` à la fonction `main`, comme cela est indiqué dans l'Exemple 12.6.

Exemple 12.6 : importsub.py (extrait).

```
import xlwings as xw


@xw.sub
def main():
```

```
wb = xw.Book.caller()
sheet = wb.sheets[0]
if sheet["A1"].value == "Hello xlwings!":
    sheet["A1"].value = "Bye xlwings!"
else:
    sheet["A1"].value = "Hello xlwings!"
```

Dans le complément xlwings, cliquez sur Import Functions avant d'appuyer sur Alt + F8 pour voir les macros disponibles : en plus de `SampleCall` qui utilise `RunPython`, vous verrez également une macro appelée `main`. Sélectionnez-la et cliquez sur le bouton Exécuter : vous verrez le message d'accueil familier dans la cellule A1. Vous pouvez maintenant affecter la macro `main` à un bouton comme nous l'avons fait dans le Chapitre 10. Bien que le décorateur `xw.sub` puisse vous faciliter la vie sous Windows, gardez à l'esprit qu'en l'utilisant, vous perdez la compatibilité multiplateforme. Avec `xw.sub`, nous avons achevé l'étude de tous les décorateurs xlwings que j'ai synthétisés dans le Tableau 12.1.

Tableau 12.1 : Décorateurs xlwings.

Décorateur	Description
`xw.func`	Placez ce décorateur au-dessus de toutes les fonctions que vous souhaitez importer dans Excel en tant que fonction.
`xw.sub`	Placez ce décorateur au-dessus de toutes les fonctions que vous souhaitez importer dans Excel en tant que procédure Sub.
`xw.arg`	Appliquez des convertisseurs et des options aux arguments ; par exemple, ajoutez une docstring *via* l'argument `doc` ou transmettez une plage en tant que DataFrame en fournissant `pd.DataFrame` comme premier argument (cela suppose que vous avez importé pandas).
`xw.ret`	Appliquez des convertisseurs et des options pour renvoyer des valeurs ; par exemple, supprimez l'index d'un DataFrame en fournissant `index=False`.

Pour plus de détails sur ces décorateurs, consultez la documentation xlwings[5].

Conclusion

Ce chapitre portait sur l'écriture de fonctions Python et leur importation dans Excel en tant qu'UDF, ce qui permet de les appeler *via* des formules. En parcourant l'étude de cas Google Trends, vous avez appris à modifier le comportement des arguments des fonctions et des valeurs de retour en utilisant respectivement les décorateurs `arg` et `ret`. La dernière partie vous a montré quelques astuces pour améliorer les performances et a présenté le décorateur `xw.sub`, que vous pouvez utiliser pour remplacer RunPython si vous travaillez exclusivement sous Windows. Il y a un avantage à écrire des UDF en Python : cela vous permet de remplacer des formules de cellules longues et complexes par du code Python qui sera plus facile à comprendre et à maintenir. Quand j'exploite des UDF, j'apprécie surtout d'utiliser des DataFrames pandas avec les nouveaux

5 https://docs.xlwings.org/en/stable/api.html#udf-decorators

tableaux dynamiques d'Excel, une combinaison qui facilite le travail avec le type de données que nous obtenons à partir de Google Trends, c'est-à-dire des DataFrames avec un nombre dynamique de lignes.

Et voilà, nous sommes arrivés à la fin de ce livre ! Je vous remercie beaucoup de l'intérêt que vous avez porté à mon interprétation d'un environnement moderne d'automatisation et d'analyse de données pour Excel. Mon idée était de vous présenter le monde de Python et ses puissants packages open source, ce qui vous permettra d'écrire du code Python pour votre prochain projet au lieu d'avoir à gérer les propres solutions d'Excel comme VBA ou Power Query. Vous trouverez ci-dessous une série d'exemples pratiques afin de faciliter votre démarrage. Après avoir lu ce livre, vous savez maintenant comment :

- Remplacer un classeur Excel par un notebook Jupyter et du code pandas.
- Traiter par lots les classeurs Excel en les lisant avec OpenPyXL, xlrd, pyxlsb ou xlwings, puis les consolider avec pandas.
- Produire des rapports Excel avec OpenPyXL, XlsxWriter, xlwt ou xlwings.
- Utiliser Excel comme frontend et le connecter à peu près à tout ce que vous voulez via xlwings, soit en cliquant sur un bouton, soit en écrivant une UDF.

Bientôt, cependant, vous voudrez dépasser le cadre de ce livre. Je vous invite à consulter la page d'accueil[6] de cet ouvrage de temps en temps pour vérifier la disponibilité de mises à jour et lire des documents supplémentaires. Dans cet esprit, voici quelques idées que vous pourriez explorer par vous-même :

- Planifiez l'exécution périodique d'un script Python à l'aide du Planificateur de tâches sous Windows ou d'une tâche cron sous macOS ou Linux. Vous pouvez, par exemple, créer un rapport Excel tous les vendredis en fonction des données provenant d'une API REST ou d'une base de données.
- Écrivez un script Python qui envoie des alertes par e-mail chaque fois que les valeurs de vos fichiers Excel remplissent une certaine condition. Cela peut être le cas lorsque le solde de votre compte, consolidé à partir de plusieurs classeurs, tombe en dessous d'une certaine valeur ou lorsqu'il affiche une valeur différente de ce qui est attendu en fonction de votre base de données.
- Écrivez du code qui trouve des erreurs dans les classeurs Excel : recherchez les erreurs de cellule comme `#REF!` ou `#VALEUR!` ou des erreurs logiques, par exemple pour vous assurer qu'une formule inclut toutes les cellules nécessaires. Si vous commencez à suivre vos classeurs critiques avec un système de contrôle de version professionnel comme Git, vous pouvez même exécuter ces tests automatiquement chaque fois que vous validez une nouvelle version.

6 https://www.xlwings.org/book

Je serai le plus heureux des hommes si ce livre vous incite à automatiser votre routine quotidienne ou hebdomadaire de téléchargement de données et de copier-coller dans Excel. L'automatisation ne se contente pas de vous faire gagner du temps, elle réduit également considérablement le risque de commettre des erreurs. Si vous avez des commentaires sur mon livre, n'hésitez pas, je vous en prie, à m'écrire. Vous pouvez me contacter en laissant un message sur GitHub (*https://github.com/fzumstein/python-for-excel*) ou sur Twitter à *@felixzumstein*.

ANNEXE A

Environnements Conda

Dans le Chapitre 2, j'ai présenté les environnements Conda en expliquant que la mention (base) au début d'Anaconda Prompt représente l'environnement Conda actuellement actif avec le nom base. Anaconda vous oblige à toujours travailler dans un environnement activé, mais l'activation se fait automatiquement pour l'environnement nommé base lorsque vous démarrez Anaconda Prompt sous Windows ou le Terminal sous macOS. Travailler avec des environnements Conda permet de séparer correctement les dépendances de vos projets : si vous souhaitez essayer une version plus récente d'un package comme pandas sans modifier votre environnement base, vous pouvez le faire dans un environnement Conda séparé. Dans la première partie de cette annexe, je vais vous guider à travers le processus de création d'un environnement Conda appelé xl38 où nous installerons tous les packages dans la version que j'ai utilisée pour écrire ce livre. Cela vous permettra d'exécuter les exemples de ce livre tels quels, même si certains packages ont publié de nouvelles versions avec des changements importants entre-temps. Dans la deuxième partie, je vous montrerai comment annuler l'activation automatique de l'environnement base si vous n'aimez pas ce comportement par défaut.

Création d'un nouvel environnement Conda

Exécutez la commande suivante dans Anaconda Prompt pour créer un nouvel environnement portant le nom xl38 et utilisant Python 3.8 :

```
(base)> conda create --name xl38 python=3.8
```

Lorsque vous appuyez sur Entrée, Conda affichera ce qu'il va installer dans le nouvel environnement et vous demandera une confirmation :

```
Proceed ([y]/n)?
```

Appuyez sur Entrée pour confirmer ou saisissez `n` si vous souhaitez annuler. Une fois l'installation terminée, activez votre nouvel environnement de la manière suivante :

```
(base)> conda activate xl38
(xl38)>
```

Le nom de l'environnement passe de `base` à `xl38` et vous pouvez désormais utiliser Conda ou pip pour installer des packages dans ce nouvel environnement sans affecter les autres environnements (pour rappel : n'utilisez pip que si le package n'est pas disponible *via* Conda). Installons à présent tous les packages de ce livre dans la version où je les ai utilisés. Tout d'abord, vérifiez que vous êtes dans l'environnement `xl38`, c'est-à-dire qu'Anaconda Prompt affiche `(xl38)`, puis installez les packages Conda comme suit (la commande suivante doit être saisie en une seule commande, les sauts de ligne ne figurant ici que pour rendre l'affichage plus lisible) :

```
(xl38)> conda install lxml=4.6.1 matplotlib=3.3.2 notebook=6.1.4 openpyxl=3.0.5
                      pandas=1.1.3 pillow=8.0.1 plotly=4.14.1 flake8=3.8.4
                      python-dateutil=2.8.1 requests=2.24.0 sqlalchemy=1.3.20
                      xlrd=1.2.0 xlsxwriter=1.3.7 xlutils=2.0.0 xlwings=0.20.8
                      xlwt=1.3.0
```

Confirmez l'installation en appuyant sur Entrée et finalisez l'environnement en installant les deux dépendances restantes avec pip :

```
(xl38)> pip install pyxlsb==1.0.7 pytrends==4.7.3
(xl38)>
```



Si vous souhaitez utiliser l'environnement `xl38` plutôt que l'environnement `base` pour parcourir les exemples de ce livre, assurez-vous de toujours activer votre environnement `xl38` en exécutant :

```
(base)> conda activate xl38
```

Autrement dit, partout où j'affiche Anaconda Prompt sous la forme `(base)>`, il faudra que votre invite affiche `(xl38)>` à la place.

Pour désactiver à nouveau l'environnement et revenir à l'environnement `base`, saisissez :

```
(xl38)> conda deactivate
(base)>
```

Si vous souhaitez supprimer complètement l'environnement, exécutez la commande suivante :

```
(base)> conda env remove --name xl38
```

Au lieu de suivre manuellement les étapes pour créer l'environnement `xl38`, vous pouvez également tirer parti du fichier d'environnement *xl38.yml* que j'ai inclus dans le dossier *conda* des exemples de code. L'exécution des commandes suivantes réalisera toute l'installation automatiquement :

```
(base)> cd C:\Users\nom_utilisateur\python-for-excel\conda
(base)> conda env create -f xl38.yml
(base)> conda activate xl38
(xl38)>
```

Par défaut, Anaconda active toujours l'environnement `base` lorsque vous ouvrez le Terminal sous macOS ou Anaconda Prompt sous Windows. Si vous n'aimez pas cela, vous pouvez annuler l'activation automatique.

Annulation de l'activation automatique

Si vous ne souhaitez pas que l'environnement `base` soit activé automatiquement chaque fois que vous lancez Anaconda Prompt, vous pouvez le désactiver : cela vous obligera cependant à saisir manuellement **conda activate base** dans une invite de commande (Windows) ou un terminal (macOS) avant de pouvoir utiliser Python.

Windows

Sous Windows, vous devrez utiliser l'invite de commande standard au lieu d'Anaconda Prompt. Les étapes suivantes activeront la commande `conda` dans une invite de commandes normale. Assurez-vous de remplacer le chemin de la première ligne par le chemin d'accès d'installation d'Anaconda sur votre système :

```
> cd C:\Users\nom_utilisateur\Anaconda3\condabin
> conda init cmd.exe
```

Votre invite de commande habituelle est maintenant configurée avec Conda, et vous pouvez donc activer l'environnement `base` de la manière suivante :

```
> conda activate base
(base)>
```

macOS

Sous macOS, exécutez simplement la commande suivante dans votre terminal pour annuler l'activation automatique :

```
(base)> conda config --set auto_activate_base false
```

Si jamais vous souhaitez revenir en arrière, exécutez à nouveau la même commande avec `true` au lieu de `false`. Les modifications entreront en vigueur après le redémarrage du Terminal. À l'avenir, vous devrez activer l'environnement `base` de la manière suivante avant de pouvoir utiliser à nouveau la commande `python` :

```
> conda activate base
(base)>
```

ANNEXE B

Fonctionnalités avancées de VS Code

Cette annexe vous montre comment le débogueur fonctionne dans VS Code et comment vous pouvez exécuter des notebooks Jupyter directement à partir de VS Code. Les sujets sont indépendants les uns des autres, et vous pouvez donc les lire dans n'importe quel ordre.

Débogueur

Si vous avez déjà utilisé le débogueur VBA dans Excel, j'ai une bonne nouvelle : le débogage avec VS Code est une expérience très similaire. Commençons par ouvrir le fichier *debugging.py* à partir de la racine des exemples de code dans VS Code. Cliquez dans la marge à gauche de la ligne numéro 4 pour qu'un point rouge apparaisse : il s'agit de votre point d'arrêt qui indique l'emplacement où l'exécution du code sera suspendue. Maintenant, appuyez sur F5 pour démarrer le débogage : la Palette de commandes apparaîtra avec une sélection de configurations de débogage. Choisissez « Python File » pour déboguer le fichier actif et exécuter le code jusqu'à ce qu'il atteigne le point d'arrêt. La ligne sera mise en surbrillance et l'exécution du code sera suspendue (Figure B.1). Pendant le débogage, la barre d'état devient orange.

Si la section Variables n'apparaît pas automatiquement sur la gauche, assurez-vous de cliquer sur le menu Exécuter et déboguer pour afficher les valeurs des variables. Vous pouvez également survoler une variable dans le code source et obtenir une info-bulle avec sa valeur. En haut, vous verrez la barre d'outils de débogage qui vous donne accès aux boutons suivants (de gauche à droite) : Continuer, Pas à pas principal, Pas à pas détaillé, Pas à pas sortant, Redémarrer et Arrêter. Lorsque vous survolez les boutons, vous verrez également les raccourcis clavier.

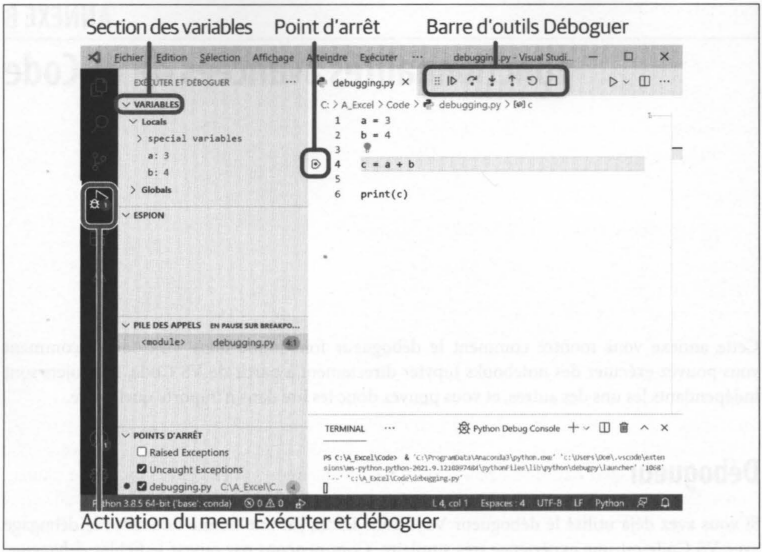


Figure B.1 : VS Code avec le débogueur arrêté sur un point d'arrêt.

Voyons ce que fait chacun de ces boutons :

Continuer

Continue d'exécuter le programme jusqu'à ce qu'il atteigne le point d'arrêt suivant ou la fin du programme. S'il atteint la fin du programme, le processus de débogage s'arrête.

Pas à pas principal

Le débogueur avance d'une ligne. Dans ce mode, le débogueur ne parcourt pas visuellement les lignes de code qui ne font pas partie de votre portée actuelle. Par exemple, il n'entrera pas dans le code d'une fonction que vous appelez, mais la fonction sera quand même appelée.

Pas à pas détaillé

Si vous avez du code qui appelle une fonction ou une classe, *etc.*, ce mode permet au débogueur d'entrer dans cette fonction ou cette classe. Si la fonction ou la classe se trouve dans un fichier différent, le débogueur ouvrira ce fichier automatiquement.

Pas à pas sortant

Si vous êtes entré dans une fonction en mode Pas à pas détaillé, le mode Pas à pas sortant fait revenir le débogueur au niveau supérieur suivant jusqu'à ce que vous reveniez au niveau le plus élevé à partir duquel vous avez appelé initialement le mode Pas à pas détaillé.

Redémarrer

Cela arrête le processus de débogage en cours et en démarre un nouveau depuis le début.

Arrêter

Cela arrête le processus de débogage en cours.

Maintenant que vous savez ce que fait chaque bouton, cliquez sur Pas à pas principal pour avancer d'une ligne et voir comment la variable c apparaît dans la section Variables, puis terminez cet exercice de débogage en cliquant sur Continuer.

Si vous enregistrez la configuration de débogage, la Palette de commandes ne s'affichera pas et ne vous posera pas de questions sur la configuration chaque fois que vous appuyez sur F5 : cliquez sur l'icône Exécuter et déboguer dans la barre d'activité, puis cliquez sur le lien « ouvrez un dossier ». Sélectionnez le dossier qui contient le fichier à déboguer (dans notre cas, il s'agit de la racine des exemples de code), puis rechargez le fichier *debugging.py* et cliquez sur l'icône Exécuter et déboguer dans la barre d'activité, ce qui affichera un lien « créer un fichier launch.json ». En cliquant sur ce lien, cela affiche à nouveau la Palette de commandes et lorsque vous sélectionnez « Python File », cela crée le fichier *launch.json* dans un sous-répertoire appelé *.vscode*. Lorsque vous appuyez maintenant sur F5, le débogueur démarre immédiatement. Si vous devez modifier la configuration ou si vous souhaitez récupérer la fenêtre contextuelle de la Palette de commandes, modifiez ou supprimez le fichier *launch.json* dans le répertoire *.vscode*.

Notebooks Jupyter dans VS Code

Au lieu d'exécuter vos notebooks Jupyter dans un navigateur Web, vous pouvez également les exécuter directement dans VS Code. En plus de cela, VS Code offre un explorateur de variables pratique ainsi que des options pour transformer le notebook en fichier Python standard sans perdre la fonctionnalité des cellules. Cela facilite l'utilisation du débogueur ou le copier-coller des cellules entre différents notebooks. Commençons par exécuter un notebook dans VS Code !

Exécution de notebooks Jupyter

Cliquez sur l'icône de l'Explorateur dans la barre d'activité et ouvrez *ch05.ipynb* à partir des exemples de code. Pour continuer, vous devrez indiquer que vous faites confiance à ce notebook en cliquant sur Confiance dans la notification qui apparaît. La disposition du notebook est un peu différente de celle du navigateur Web car elle s'adapte à l'interface de VS Code, mais sinon,

c'est la même expérience, y compris tous les raccourcis clavier. Exécutons les trois premières cellules *via* Maj + Entrée. Cela démarrera le serveur de notebook Jupyter s'il n'est pas déjà en cours d'exécution. Après avoir exécuté les cellules, cliquez sur le bouton Variables dans le menu en haut du notebook : cela ouvrira l'explorateur de variables, dans lequel vous pourrez voir les valeurs de toutes les variables qui existent actuellement (Figure B.2). Cela signifie que vous ne trouverez que les variables des cellules qui ont été exécutées.



Pour enregistrer les notebooks dans VS Code, vous devez appuyer sur Ctrl + S sous Windows ou Commande + S sous macOS. La commande Fichier > Enregistrer ne fonctionne pas.

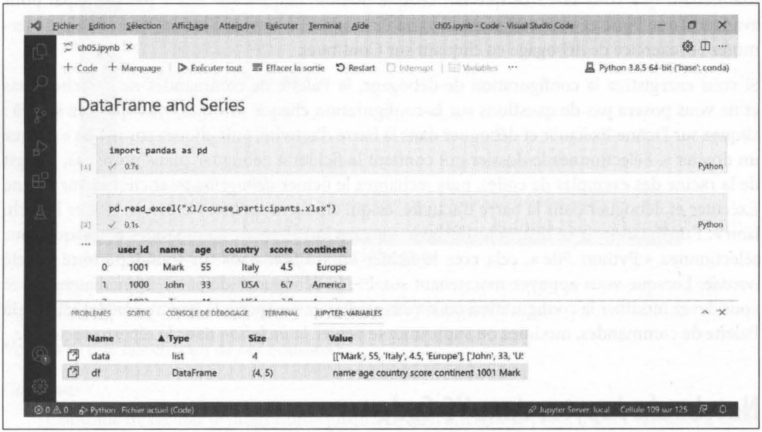
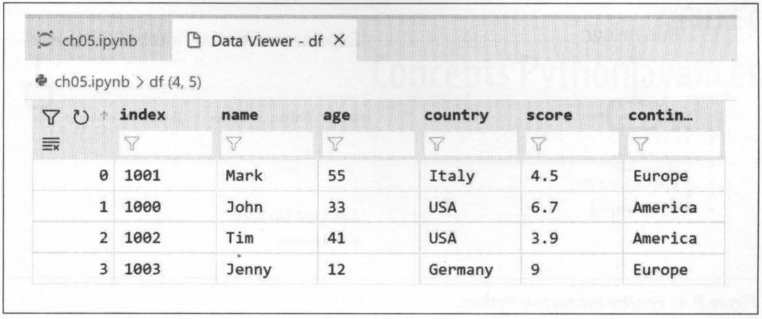


Figure B.2 : Explorateur de variables d'un notebook Jupyter.

Si vous utilisez des structures de données telles que des listes imbriquées, des tableaux NumPy ou des DataFrames, vous pouvez faire un double-clic sur le nom de la variable : cela ouvrira la visionneuse de données qui vous procurera une vue familière ressemblant à celle d'une feuille de calcul. La Figure B.3 illustre la visionneuse de données affichant la variable `df`.



	index	name	age	country	score	contin…
0	1001	Mark	55	Italy	4.5	Europe
1	1000	John	33	USA	6.7	America
2	1002	Tim	41	USA	3.9	America
3	1003	Jenny	12	Germany	9	Europe

Figure B.3 : Visionneuse de données d'un notebook Jupyter.

VS Code vous permet donc d'exécuter des fichiers de notebook Jupyter standard, mais il autorise également la transformation des notebooks en fichiers Python normaux, sans perte des cellules. Voyons comment cela fonctionne !

Scripts Python avec des cellules de code

Pour utiliser des cellules de notebook Jupyter dans des fichiers Python standard, VS Code utilise un commentaire spécial pour désigner les cellules : `# %%`. Pour convertir un notebook Jupyter existant, ouvrez-le et choisissez la commande Export qui est disponible à partir du bouton en forme de points de suspension dans le menu en haut du notebook (voir la Figure B.2). Cela vous permettra de sélectionner « Python Script » dans la Palette de commandes. Cependant, au lieu de convertir un fichier existant, créons un nouveau fichier que nous appellerons *cells.py* et qui aura le contenu suivant :

```
# %%
3 + 4
# %% [markdown]
# # Ceci est un titre
#
# Contenu markdown
```

Les cellules Markdown doivent commencer par `# %% [markdown]` et la cellule entière doit être marquée en tant que commentaire. Si vous souhaitez exécuter un tel fichier en tant que notebook, cliquez sur le lien « Run Below » qui apparaît lorsque vous survolez la première cellule. Cela ouvrira une fenêtre interactive Python, à droite, comme cela est illustré à la Figure B.4.

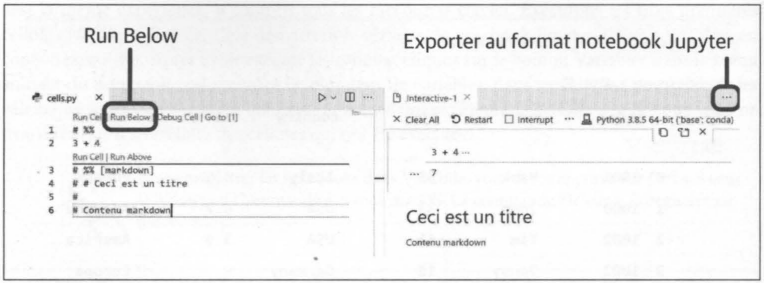


Figure B.4 : Fenêtre interactive Python.

La fenêtre interactive Python est affichée en tant que notebook. Pour exporter votre fichier au format *ipynb*, cliquez en haut de la fenêtre interactive Python sur l'icône en forme de points de suspension, et choisissez la commande Save (Export to Jupyter Notebook). La fenêtre interactive Python vous offre également une cellule en bas à partir de laquelle vous pouvez exécuter du code de manière interactive. L'emploi de fichiers Python standard à la place des notebooks Jupyter vous permet de vous servir du débogueur de VS Code et facilite l'utilisation du contrôle de version, car les cellules affichant les résultats, qui ajoutent généralement beaucoup d'informations inutiles entre les versions, sont ignorées.

ANNEXE C

Concepts Python avancés

Dans cette annexe, nous examinons de plus près les trois sujets suivants : classes et objets, objets datetime prenant en charge les fuseaux horaires, objets mutables ou immuables. Les sujets sont indépendants les uns des autres, et vous pouvez donc les lire dans n'importe quel ordre.

Classes et objets

Dans cette section, nous allons écrire notre propre classe pour mieux comprendre comment les classes et les objets sont liés. Les classes définissent de nouveaux types d'objets : une classe se comporte comme un moule que vous utilisez pour cuire un gâteau. Selon les ingrédients que vous utilisez, vous obtenez un gâteau différent, par exemple, un gâteau au chocolat ou un cheesecake. Le processus d'obtention d'un gâteau (l'objet) à partir du moule (la classe) est appelé *instanciation* ; c'est d'ailleurs la raison pour laquelle les objets sont également appelés *instances de classe*. Qu'il s'agisse de gâteau au chocolat ou de cheesecake, ils sont tous deux un *type* de gâteau : les classes permettent de définir de nouveaux types de données qui regroupent des données (*attributs*) et des fonctions (*méthodes*), ce qui vous aide à structurer et à organiser votre code. Permettez-moi maintenant de revenir à l'exemple de jeu de course automobile du Chapitre 3 pour définir notre propre classe :

```
In [1]: class Car:
            def __init__(self, color, speed=0):
                self.color = color
                self.speed = speed

            def accelerate(self, mph):
                self.speed += mph
```

Il s'agit d'une classe de voiture simple avec deux méthodes. Les méthodes sont des fonctions qui font partie d'une définition de classe. Cette classe possède une méthode normale appelée `accelerate`. Cette méthode modifie les données (`speed`) d'une instance de cette classe. Il y a

également une méthode spéciale, qui commence et se termine par des barres de soulignement doubles, appelée `__init__`. Cette méthode sera appelée automatiquement par Python lorsqu'un objet est *initialisé* afin de fournir des données initiales à l'objet. Le premier argument de chaque méthode, qui représente l'instance de la classe, est appelé `self` par convention. Cela deviendra plus clair lorsque vous verrez comment on utilise la classe `Car`. Tout d'abord, instancions deux voitures. On réalise cela de la même manière que l'on appelle une fonction : on appelle la classe en ajoutant des parenthèses et en fournissant les arguments de la méthode `__init__`. On ne fournit jamais rien à `self`, car Python s'en occupe. Dans cet exemple, `self` sera respectivement `car1` ou `car2` :

```
In [2]: # On instancie deux objets car
car1 = Car("red")
car2 = Car(color="blue")
```

Lorsque vous appelez une classe, vous appelez vraiment la fonction `__init__`, et c'est pourquoi tout ce qui concerne les arguments de fonction s'applique ici également : pour `car1`, nous fournissons l'argument en tant qu'argument positionnel, tandis que pour `car2`, nous utilisons un argument nommé. Après avoir instancié les deux objets de voiture de la classe `Car`, nous allons examiner leurs attributs et appeler leurs méthodes. Comme nous allons le voir, après l'accélération de `car1`, la vitesse de `car1` sera modifiée, mais elle reste inchangée pour `car2`, les deux objets étant indépendants l'un de l'autre :

```
In [3]: # Par défaut, un objet affiche son emplacement en mémoire
car1
Out[3]: <__main__.Car at 0x29b36c17190>
In [4]: # Les attributs donnent accès aux données d'un objet
car1.color
Out[4]: 'red'
In [5]: car1.speed
Out[5]: 0
In [6]: # On appelle la méthode accelerate sur car1
car1.accelerate(20)
In [7]: # L'attribut speed de car1 a été modifié
car1.speed
Out[7]: 20
In [8]: # L'attribut speed de car2 ne change pas
car2.speed
Out[8]: 0
```

Python permet également de modifier directement les attributs sans avoir à utiliser de méthodes :

```
In [9]: car1.color = "green"
In [10]: car1.color
Out[10]: 'green'
In [11]: car2.color  # pas de changement
Out[11]: 'blue'
```

Pour résumer : les classes définissent les attributs et les méthodes des objets. Les classes permettent de regrouper des fonctions qui ont un lien entre elles (méthodes) et des données (attributs) afin qu'elles soient facilement accessibles *via* la notation pointée : `mon_objet.attribut` ou `mon_objet.méthode()`.

Exploitation d'objets datetime prenant en charge les fuseaux horaires

Dans le Chapitre 3, nous avons brièvement examiné les objets `datetime` naïfs prenant en charge les fuseaux horaires. Quand le fuseau horaire est important, on travaille généralement dans le fuseau horaire UTC et on ne le transforme en fuseaux horaires locaux que lorsque l'on veut les afficher. UTC, qui signifie Temps Universel Coordonné, est le successeur du système GMT (heure du méridien de Greenwich). Lorsque vous travaillez avec Excel et Python, il est souvent souhaitable de transformer des horodatages naïfs, tels que ceux qui sont fournis par Excel, en objets `datetime` prenant en charge les fuseaux horaires. Pour la gestion des fuseaux horaires en Python, vous pouvez utiliser le package dateutil, qui ne fait pas partie de la bibliothèque standard, mais qui est préinstallé avec Anaconda. Les exemples suivants montrent quelques opérations courantes lors de l'utilisation d'objets `datetime` avec des fuseaux horaires :

```
In [12]: import datetime as dt
         from dateutil import tz
In [13]: # Objets datetime naïfs
         timestamp = dt.datetime(2020, 1, 31, 14, 30)
         timestamp.isoformat()
Out[13]: '2020-01-31T14:30:00'
In [14]: # Objets datetime prenant en compte les fuseaux horaires
         timestamp_eastern = dt.datetime(2020, 1, 31, 14, 30,
                                         tzinfo=tz.gettz("US/Eastern"))
         # L'affichage isoformat facilite la lecture
         # du décalage par rapport au format UTC
         timestamp_eastern.isoformat()
Out[14]: '2020-01-31T14:30:00-05:00'
In [15]: # Assigne un fuseau horaire à un objet datetime naïf
         timestamp_eastern = timestamp.replace(tzinfo=tz.gettz("US/Eastern"))
         timestamp_eastern.isoformat()
Out[15]: '2020-01-31T14:30:00-05:00'
In [16]: # Conversion d'un fuseau horaire en un autre
         # Comme le fuseau horaire UTC est très courant,
         # il y a une abréviation : tz.UTC
         timestamp_utc = timestamp_eastern.astimezone(tz.UTC)
         timestamp_utc.isoformat()
Out[16]: '2020-01-31T19:30:00+00:00'
In [17]: # On convertit en un fuseau naïf
         timestamp_eastern.replace(tzinfo=None)
Out[17]: datetime.datetime(2020, 1, 31, 14, 30)
In [18]: # Heure courante sans fuseau horaire
         dt.datetime.now()
```

```
Out[18]: datetime.datetime(2021, 9, 11, 17, 7, 51, 54855)
In [19]: # Heure courante dans le fuseau UTC
         dt.datetime.now(tz.UTC)
Out[19]: datetime.datetime(2021, 9, 11, 15, 8, 10, 904927, tzinfo=tzutc())
```

Fuseaux horaires avec Python 3.9

Python 3.9 a ajouté la prise en charge des fuseaux horaires à la bibliothèque standard sous la forme du module `timezone`. Utilisez-le pour remplacer les appels à `tz.gettz` de `dateutil` :

```
from zoneinfo import ZoneInfo
timestamp_eastern = dt.datetime(2020, 1, 31, 14, 30,
                                tzinfo=ZoneInfo("US/Eastern"))
```

Objets Python mutables ou immuables ?

En Python, les objets qui peuvent modifier leurs valeurs sont appelés mutables et ceux qui ne le peuvent pas sont appelés immuables. Le Tableau C.1 montre la répartition entre les différents types de données.

Tableau C.1 : Types de données mutables et immuables.

Mutabilité	Types de données
Mutable	Listes, dictionnaires, ensembles.
Immuable	Entiers, flottants, booléens, chaînes, datetime, tuples.

Il est important de connaître cette différence, car les objets mutables peuvent se comporter différemment de ce à quoi vous êtes habitué dans d'autres langages, notamment VBA. Jetez un coup d'œil à l'extrait de code VBA suivant :

```
Dim a As Variant, b As Variant
a = Array(1, 2, 3)
b = a
a(1) = 22
Debug.Print a(0) & ", " & a(1) & ", " & a(2)
Debug.Print b(0) & ", " & b(1) & ", " & b(2)
```

L'exécution de ce code affiche les résultats suivants :

```
1, 22, 3
1, 2, 3
```

Prenons maintenant le même exemple en Python, mais avec une liste :

```
In [20]: a = [1, 2, 3]
         b = a
         a[1] = 22
         print(a)
         print(b)
[1, 22, 3]
[1, 22, 3]
```

Que s'est-il passé ici ? En Python, les variables sont des noms que vous « attachez » à un objet. En écrivant `b = a`, vous attachez les deux noms au même objet, à savoir la liste `[1, 2, 3]`. Toutes les variables attachées à cet objet afficheront donc les modifications apportées à la liste. Cependant, cela ne se produit qu'avec des objets mutables : si vous remplacez la liste par un objet immuable comme un tuple, la modification de `a` ne changera pas `b`. Si vous voulez qu'un objet mutable comme `b` soit indépendant des modifications de `a`, vous devez copier explicitement la liste :

```
In [21]: a = [1, 2, 3]
         b = a.copy()
In [22]: a
Out[22]: [1, 2, 3]
In [23]: b
Out[23]: [1, 2, 3]
In [24]: a[1] = 22  # La modification de "a"...
In [25]: a
Out[25]: [1, 22, 3]
In [26]: b  # ...n'affecte pas "b"
Out[26]: [1, 2, 3]
```

En utilisant la méthode `copy` d'une liste, vous créez une copie *shallow* : vous obtiendrez une copie de la liste, mais si la liste contient des éléments mutables, ceux-ci seront toujours partagés. Si vous souhaitez copier tous les éléments de manière récursive, vous devez effectuer une copie *deep* à l'aide du module `copy` de la bibliothèque standard :

```
In [27]: import copy
         b = copy.deepcopy(a)
```

Voyons maintenant ce qui se passe lorsque vous utilisez des objets mutables comme arguments de fonction.

Appel de fonctions avec des objets mutables comme arguments

Si vous programmez en VBA, vous avez probablement l'habitude de marquer les arguments de fonction avec `ByRef` (passage par référence) ou `ByVal` (passage par valeur) : lorsque vous passez une variable à une fonction comme argument, la fonction aura la possibilité de la modifier (`ByRef`) ou exploitera une copie des valeurs (`ByVal`), ce qui laisse ainsi la variable d'origine intacte. `ByRef` étant la valeur par défaut, examinez le code de la fonction suivante :

```
Function increment(ByRef x As Integer) As Integer
    x = x + 1
    increment = x
End Function
```

Ensuite, appelez la fonction comme ceci :

```
Sub call_increment()
    Dim a As Integer
    a = 1
    Debug.Print increment(a)
    Debug.Print a
End Sub
```

L'exécution de ce code affiche les résultats suivants :

```
2
2
```

Toutefois, si vous remplacez `ByRef` dans la fonction `increment` par `ByVal`, le résultat suivant sera affiché :

```
2
1
```

Comment cela fonctionne-t-il en Python ? Lorsque vous passez des variables, vous transmettez des noms qui pointent vers des objets. Cela signifie que le comportement dépend du fait que l'objet soit mutable ou non. Utilisons d'abord un objet immuable :

```
In [28]: def increment(x):
             x = x + 1
             return x
In [29]: a = 1
         print(increment(a))
         print(a)
2
1
```

Répétons maintenant l'exemple avec un objet mutable :

```
In [30]: def increment(x):
             x[0] = x[0] + 1
             return x
In [31]: a = [1]
         print(increment(a))
         print(a)
[2]
[2]
```

Si l'objet est mutable et que vous ne souhaitiez pas modifier l'objet d'origine, vous devrez transmettre une copie de l'objet :

```
In [32]: a = [1]
         print(increment(a.copy()))
         print(a)
[2]
[2]
```

Fonctions avec des objets mutables comme arguments par défaut

Lorsque vous écrivez des fonctions, vous ne devez normalement pas utiliser d'objets mutables comme arguments par défaut. Cela est dû au fait que la valeur des arguments par défaut n'est évaluée qu'une seule fois dans le cadre de la définition de la fonction, mais pas à chaque fois que la fonction est appelée. Par conséquent, l'utilisation d'objets mutables comme arguments par défaut peut entraîner un comportement inattendu :

```
In [33]: # Ne faites pas cela :
         def add_one(x=[]):
             x.append(1)
             return x
In [34]: add_one()
Out[34]: [1]
In [35]: add_one()
Out[35]: [1, 1]
```

Si vous souhaitez utiliser une liste vide comme argument par défaut, procédez comme ceci :

```
In [36]: def add_one(x=None):
             if x is None:
                 x = []
             x.append(1)
             return x
In [37]: add_one()
Out[37]: [1]
In [38]: add_one()
Out[38]: [1]
```

Index

Symboles

@xw.ret 276

A

Access 242
add 109
Affectation augmentée 65
agg 119
aggfunc 120
Algèbre linéraire 82
Altair 126
Anaconda 19, 20, 229
  installation 20
Anaconda Prompt 19, 21
  commandes 22
  exécution d'un script Python 41
  macOS 21
and 50
Annotation de type 75
API 216
  REST 239
  Web 238
append 179
AppleScript 213
Application, division en couches 4
applymap 110
appscript 213, 217
arange 85
area 125
asfreq 144
autofit 212
Azure DevOps 8

B

Backend 254, 258
bar 125
barh 125
Base de données 242, 244
  connexion 245
  création manuelle 261
  NoSQL 243
Binder 34
Bitbucket 8
Bokeh 126
Boucle 62
  quitter 64
  valeur de la variable compteur 64
Bouton 224, 255
box 125
break 64

C

capitalize 109
cd 23
Chaîne, échappement de caractères 127
Chemin 153
  absolu 260
Classe 47
  instances 71
Classeur
  autonome 229
  configuration 230
Clé
  étrangère 245, 249
  primaire 244
Code
  continuation de ligne 56
  éditeurs 42
  exécution 39
  exemples xvi
  gestion des erreurs 251
  guide de style 72
  IDE 42
  lisibilité 11
  maintenance 11
  organisation 66
  sauts de ligne implicites 56
  système de contrôle 8
  vérification des types 75
Codespaces 36
Collection 196
Coloration syntaxique 35
columns 92
COM 213
Commentaire 49
Compatibilité 16
Complément, création 230, 271
complex 48
Compréhension, structures de données 65
concat 113, 141
Conda 26
  commandes 26
  environnement 27
Contrôle
  ActiveX 16, 225
  de code source 8
  de version 8, 36
Conversion
  dates 176
  types de données 168
convert 205
corr 142
Corrélation 140, 142
Couche
  de données 5, 253
  de présentation 5, 253
  métier 5, 253
Couleur, valeur hexadécimale 171

count 117
cuDF 146

D

Dask 146
DataFrame 87
  affichage des informations 89
  ajout d'une colonne 103
  ajout d'un graphique 121
  alignement des données 107
  alternatives 146
  analyse de séries chronologiques 132
  application d'une fonction 110
  colonnes 92
  combinaison 112
  concaténation 113
  copie 112
  CSS 187
  différence avec une série 96
  données
    en double 106
    manquantes 104
  étiquettes de lignes 90
  exportation 127
    au format CSV 127
  fenêtre glissante 145
  fuseau horaire UTC 136
  fusion 120
  importation 127
  importation de fichier CSV 128
  index 90
    doublons 90
  jointures 114
  méthodes d'exploration 129
  mise en forme 183
    de l'index 184
    des colonnes 184
    des données 186
  modification des données 101
    par étiquette 101
    par indexation booléenne 102
    par position 101
  moyenne mobile 145
  MultiIndex 95
  opérations arithmétiques 107
  regroupement 118
  remplacement des valeurs 103
  sélection
    avec un MultiIndex 100
    des données 95
    par étiquette 95
    par indexation booléenne 97
    par position 96
  statistiques 129
    descriptives 117
  style 187
  suppression des colonnes 93
  tableaux croisés dynamiques 119
  transposition 93
  types de jointures 114
  utilisation de colonnes de texte 109
  vectorisation 107
  vue 112
DateOffset 136
date_range 132
datetime 70, 133, 176
  mise en forme des objets 71
DatetimeIndex 132, 134
  ajout d'un graphique 135
  création 132
  filtrage 134
  fuseaux horaires 136
DAX xi
Débogage 262
débogueur 36
decimal 48
Décorateur 273, 276
Découpage 53, 200
def 66
Déploiement 228
describe 129
Développement, environnement 19
Dictionnaire, compréhension 66
Diffusion 82
dir 23
Données, validation 256
drop_duplicates 106
DRY 6
dtypes 90, 130
duplicated 106

E

Échappement 127
Eclipse 43
elif 61
else 61
ElseIf 61
Emacs 42
Ensemble 59
  compréhension 66
enumerate 63
Erreur, gestion 251
Excel
  affichage de la galerie à l'ouverture 196
  analyse de séries chronologiques 131
  automatisation 193
  classeur autonome 229
  classeurs 197
  comparaison des packages 166
  complément xlwings 220
  configuration
    du complément xlwings 221
    requise 194
  documentation du modèle objet 217
  environnement sandboxé 196
  fichiers volumineux 179
  fonctionnalités manquantes 216
  fonctions volatiles 269
  génération d'un rapport 151
  graphiques 207
  Horror Stories 4
  incidents 3
  langage de programmation 2
  London Whale 3
  mise en forme conditionnelle 257
  modèle objet 196, 197
  noms définis 209

packages
d'écriture des fichiers 165
de lecture des fichiers 165
paramètres linguistiques 198
plusieurs instances 201
recalcul des formules 270
tableaux
croisés dynamiques 119
CSE hérités 274
dynamiques 274
validation des données 256
version xv
serveur 211
visualisation des données 194
ExcelFile 155, 158
excel.py 167
ExcelWriter 162
except 251, 252
Exception 251
gestion séparée des erreurs 253
Expression
conditionnelle 62
lambda 111
eye 85

F

False 50
ffill 144
Fichier
chemin 153
CSV 127
extension 24
xls 166, 180
xlsb 166, 175, 176, 211
xlsm 166, 173, 174, 223
xlsx 166
xlt 166
xltm 166
xltx 166, 171
fillna 105
finally 252
float 47
Fonction
anonyme 111
appel 67
arguments
facultatifs 67
nommés 68
positionnels 68
requis 67
décorateurs 268
définition 66
importation 270
lambda 2
universelle 83
valeur de retour 67
volatile 269
for 62
For Each 62
fraction 48
Frontend 219, 254

G

Gestionnaire de contexte 159, 162
GET 238
Git 8, 9, 36
GitHub 8
GitLab 8
Globbing 154
Google Colab 34
Google Trends 271
téléchargement au format CSV 272
GoTo 16
Graphique 121, 207
bibliothèques de traçage 126
carte de chaleur 143
rendements logarithmiques 138
groupby 101

H

head 130
hexbin 125
hist 125
Histogramme 138
HoloViews 126
HTTP 238
code d'état 240

I

Ibiset 146
IDE 36, 42
if 61
iloc 96, 101
exemples 97
import 69
this 25
imshow 143
Indentation 12
Indexation 200
chaînée 84
Indicateur de type 75
INDIRECT 257
info 89, 128, 129, 156
Informatique scientifique 14
input 263
Instanciation 71
int 47
IntelliSense 35
Interface utilisateur 219, 255
Invite de commande 21
historique des commandes 23
isna 105
items 64

J

join 114
Jointure 114, 250
condition 116
types 114
json 239
JSON 238, 241
Jupyter 19
JupyterLab 43

K

Kaggle 34
kde 125
keep_vba 173
Koalas 146
Komodo IDE 43

L

lambda 111
line 125
Linter 74
Liste, compréhension 65
LN 138
loc 95, 101
  découpage 95
  exemples 95
log 138
LOG 138
Logarithme 138
Lotus 1-2-3 1
ls 23
lxml 179

M

Macro
  activation 223
  affectation à un bouton 224, 258
  exécution 224
  exécution du code 202
  extraction d'un fichier Excel 174
margins 120
Markdown 31
Matplotlib 121, 208
Matrice 82
mean 117, 118
melt 120
merge 116
Merge request 8
Microsoft 365 xv
Mise en forme conditionnelle 257
Modin 146, 183
Modularité 4
Module 68
  alias 69
  Dictionary 230
  excel.py 167
MongoDB 243
MultiIndex 95, 100
Multiplan 1
MVC 254
mypy 75
MySQL 242, 246

N

NaN 92, 104
NaT 104
ndim 206
None 50, 104
Normalisation 242
NoSQL 243
not 50
Notebook Jupyter 19, 28
  affichage des DataFrames 89
  arrêt 34
  cellules 30
  code 31
  coloration syntaxique 31
  dans le cloud 34
  exemples de code 34
  lancement du serveur 29
  Markdown 31
  mode commande 32
  mode édition 32
  ordre d'exécution des commandes 33
  raccourcis clavier 32
Notepad++ 42
NumFOCUS 14
NumPy 2, 14, 79
  constructeurs de tableaux 85
  constructeurs utiles 85
  copie d'un tableau 86
  création de tableaux 84
  diffusion 82
  données manquantes 104
  fonction universelle 83
  tableau 79
  vectorisation 81
  vue d'un tableau 86

O

Objet 71
  type 47
ohlc 144
oletools 188
ones 85
OpenPyXL 166, 169, 179, 180, 181, 183, 184
  écriture de fichier Excel 170
  édition de fichier Excel 172
  lecture de fichier Excel 169
Open source 14
Opérateur
  arithmétique 108
  booléen 50, 98
  mathématique 49
  ternaire 62
or 50
Oracle 242, 246
ORM 245

P

Package
  application de traqueur 235
  choix selon le format de fichier 166
  écriture de fichiers Excel 165
  gestion 25
  gestionnaire 13
  installation 26
  lecture de fichiers Excel 165
pandas 2, 14, 87
  alignement des données 107
  DataFrames 87
  écriture de fichiers Excel 161
  lecture de feuilles en parallèle 181
  lecture de fichiers Excel 152, 155
  limites 146
    avec des fichiers Excel 163
  manipulation de fichiers Excel 151
  Matplotlib 121
  Plotly 123
  séries 87
    chronologiques 131
  statistiques descriptives 118
  tracés 121
  types de graphiques 125

Parallélisation 181
  avec Modin 183
  estimation du gain de temps 182
parse_dates 133
pass 61
Path 153
pathlib 153
pct_change 139
PEP 72
PEP 8 72, 74
  violation des règles 74
PeriodIndex 132
pie 125
Pillow 209
pip 13, 26
  commandes 26
pivot_table 119
Plage nommée 209, 255
plot 121
plotly 26
Plotly 123, 142
  installation 123
Point de terminaison 240
POST 238
PostgreSQL 242, 243, 246, 261
Power BI 10
Power Pivot 10
Power Query xi, 10
Processus zombie 214
Programmation
  bonnes pratiques 4
  code spaghetti 16
  gestion des erreurs 15
  langages modernes 15
  multiplateforme 16
  principe DRY 6
  test 7
PTVS 37
Pull request 8
PyArrow 146
PyCharm 42
pydantic 75
PyDev 43
pyexcel 188
PyExcelerate 188
PyInstaller 229
pylightxl 188
PyPI 13, 52, 236
  API JSON 240
PySpark 146
Python
  aide en ligne 52
  attributs 47
  bibliothèque pour les images 209
  bibliothèque standard 13
  blocs de code 60
  chaînes de caractères 51
  chemin de recherche des modules 225
  commentaires 49
  créateur xi
  création d'un package 232
  date de sortie 1
  débogage 262
  découpage 53
  dictionnaires 57
  différence majuscules et minuscules 46
  distributions 19
  documentation 52
  ensembles 59
  environnement
    de développement 19
    virtuel 27
  exceptions 251
  exécution des scripts 39
  fonctions 47
  f-strings 51
  gestion
    des appels COM 213
    des dates 70
    des heures 70
  guide de style 56, 72
  indexation 53
  informatique scientifique 14
  listes 55
    imbriquées 55
  méthodes 47
    de chaînes 109
  mise en forme des chaînes 111
  modules 68
  objets 46
  opérateurs
    booléens 50
    mathématiques 49
  PEP 8 74
  REPL 24
  session interactive 24
  sponsoring 14
  structures
    de contrôle 60
    de données 54
  tuples 58
  types
    booléens 50
    de données 45, 75
    numériques 47
  variables 46
  versions xv, 27
  Zen de Python 12
Python Package Tracker 235
  base de données 244
  structure de l'application 253
PYTHONPATH 225, 231, 270
pytrends 26
pywin32 213, 216
pyxlsb 26, 160, 166, 175

R

range 62
  avec des tuples 200
  définition d'une plage 199
  indices à partir de 1 201
  options 205
  value 199
Range.ClearFormats 217
Rapport, génération automatique 151, 189, 211
raw 215
read_clipboard 127
read_csv 127, 128, 133
read_excel 127, 152, 155
  avec Python 3.9 152
  lecture de feuilles en parallèle 181
  paramètres 160

read_html 127
read_json 127, 241
read_sql 127
Rebasement 140, 141
RECHERCHEV 114
Redis 243
reindex 91, 94
Relation 242
Répertoire
  changement 23
  lecture des fichiers 154
  liste des fichiers 23
  parent 23
REPL 24, 25
Requests 259
Requête, SQL 247, 249
resample 143
reset_index 90, 101
reshape 85
resolve 154
return 67
rglob 154
rolling 145
Run main 222
RunPython 223, 226, 232, 257
  invite de commande 263
  référence à xlwings 227

S

SampleCall 223
Scalaire 81
scatter 125
SciPy 14
Script 68
Seaborn 126
SELECT 247
Séparation des problèmes 4, 253
Série 87
  chronologique 131
set_index 90
set_mock_caller 262
shift 138
sort_index 92
sort_values 92
Sous-échantillonnage 143
Spotify 240
Spyder 42
SQL 242
  injection 249
  problème de sécurité 250
  requêtes 247
SQLAlchemy 245, 249
  chaînes de connexion 246
  espaces réservés 250
  pilotes 246
SQLite 242, 245, 246, 249
  migration vers PostgreSQL 261
SQL Server 242, 246
StackOverflow 36
str 109
strftime 71
strip 109
strptime 71
styleframe 188
Sub 67
Sublime 42
sum 117
Suréchantillonnage 143

T

Tableau
  bidimensionnel 80
  création 84
  dynamique 274, 275
  lecture des éléments 84
  NumPy 79
  unidimensionnel 80
tail 130
Temps CPU 182
Test 7
  conditionnel 61
  unitaire 7
ThisWorkbook 203
to_clipboard 127, 194
to_csv 127
to_datetime 133
to_excel 127, 161
  paramètres 162
to_html 127
to_json 127
to_sql 127
Tracé 121
transpose 93, 206
True 50
try 251, 252
Typage dynamique 46
Type de données immuable 58
tz_convert 136
tz_localize 136

U

Uber 240
UDF 265
  commande quickstart 266
  Import Functions 266, 269, 270
  sous macOS 265
update 209

V

Vaex 146
Validation 256
van Rossum, Guido xi
VBA
  accès approuvé au modèle d'objet 266
  ajout de la référence à xlwings 227
  ajout d'un module 227, 257
  collections 196
  différences avec xlwings 199
  éditeur de code 223
  exécution du code 202
  insertion d'un bouton 224
  modèle objet d'Excel 196
  module de code 203
  remplacement par xlwings 194
  xlwings_udfs 269
VBAProject 257
vbaProject.bin 175
VCS 8
Vectorisation 81
Version, contrôle 8
view 194
Vim 42
VisiCalc 1

Visual Studio 37
VS Code 35
  débogage 262
  exécution d'un script Python 39
  extensions 36
  installation 37
  Interface utilisateur 37
  palette des commandes 38
  PEP 8 74
  respect du guide de style 74

W

Wall time 182, 215
while 65
Wing Python IDE 43
WinPython 229
with 159
writer.save() 162

X

xlrd 166, 176, 180
XlsxWriter 166, 173, 180, 185
xltrail 9
xlutils 26, 166, 178
xlwings 167, 193
  accès au modèle objet d'Excel 216
  ajout d'un graphique 207
  amélioration des performances 215
  api 216
  API 216
  app 196, 201
  appel entre applications 215
  book 196, 197
  classeur autonome 229
  CLI 219, 220
  collection
    books 202
    charts 207
    names 210
  commande quickstart 222
  configuration 230
    du complément 221
  convertisseurs intégrés 205
  décorateur
    de fonction 268
    de retour 277
  dépendance Python 228
  exemple d'application 235
  expand 204
  exploitation
    de fichiers exécutables 229
    des DataFrames 204
    sans complément 229
  figure 208
  fondations 213
  Import Functions 266, 269, 270
  installation du complément Excel 220
  macro 203
  mise à jour 220
  noms définis 210
  options intégrées 206
  paramètres 231
  processus zombies 213
  propriétés de l'objet app 216
  quickstart --standalone 230
  range 196, 198
  référence dans l'éditeur VBA 227
  remplacement de VBA 194
  Restart UDF Server 271
  Run main 222
  RunPython 223, 226, 232
  SampleCall 223
  sheet 196, 200
  Show Console 263
  valeurs brutes 215
  valeurs par défaut 231
  variables d'environnement 231
  version PRO 232
  view 195
  xlwings.conf 230
xlwings_udfs 269
xlwt 166, 177

Z

Zen de Python 12, 25
zeros 85

O'REILLY®

Python pour Excel

Alors même qu'Excel est devenu un logiciel aussi omniprésent qu'incontournable dans la plupart des secteurs professionnels, un nombre grandissant d'utilisateurs réclament à Microsoft d'adopter Python comme langage de script pour ce logiciel. Mais pourquoi cette combinaison est-elle autant souhaitée ? Dans ce guide pratique, Felix Zumstein, le créateur de xlwings, un package open source d'automatisation d'Excel avec Python, enseigne aux utilisateurs avancés d'Excel comment associer efficacement ces deux univers.

Certes, quelques nouveautés ont été intégrées à Excel ces dernières années, mais VBA, son langage d'automatisation, a cessé d'évoluer depuis un certain temps. De nombreux utilisateurs ont déjà adopté Python pour automatiser les tâches dans Excel. Grâce à cet ouvrage, vous allez pouvoir abandonner VBA au profit de Python.

- Utiliser Python sans forcément maîtriser la programmation
- Apprendre à exploiter des outils récents comme Jupyter et Visual Studio Code
- Récupérer, nettoyer et analyser des données et effectuer des opérations de calcul Excel classiques grâce à pandas
- Automatiser des tâches Excel fastidieuses comme la consolidation de classeurs ou la création de rapports
- Profiter de xlwings pour concevoir des outils Excel interactifs utilisant le moteur de calcul de Python
- Avec du code Python, connecter Excel à des bases de données et à des fichiers CSV pour télécharger des données
- Susbstituer Python à VBA, Power Query et Power Pivot

« Ce livre montre comment intégrer Python à Excel afin d'éviter tous les désagréments causés par l'utilisation de classeurs volumineux, la manipulation de milliers de formules et les bricolages incertains en VBA. *Python pour Excel* est assurément l'ouvrage le plus utile que j'ai lu sur Excel, et que je recommande sans hésiter à tout utilisateur avancé de ce tableur. »

Andreas F. Clenow
DG de ACIES Asset Management et auteur de best-sellers sur les investissements boursiers

Felix Zumstein est le créateur du package open source xlwings, qui permet d'automatiser Excel avec Python sous Windows et macOS. Il est également à l'origine de xltrail, un outil de contrôle de version des fichiers Excel, et a acquis à ce titre une véritable expertise d'Excel dans des domaines d'application variés.

MIXTE
Papier issu de sources responsables
FSC® C003309

editionsfirst.fr

FIRST INTERACTIVE

771132 XI-21 34 €

9 782412 073605