

O'REILLY®



Python pour la data science

LES MEILLEURS OUTILS POUR TRAVAILLER AVEC LES DONNÉES



Jake VanderPlas

Python pour la data science

Jake VanderPlas



Python pour la data science

Traduction française publiée et vendue avec l'autorisation de
O'Reilly Media, Inc. de *Python Data Science Handbook*
ISBN 9781491912058 © 2017 Jake VanderPlas.

© 2022 Éditions First, un département d'Édi8.
92, avenue de France
75013 Paris – France
Tél. : 01 44 16 09 00
Fax : 01 44 16 09 01

Courriel : firstinfo@editionsfirst.fr
Site Internet : www.editionsfirst.fr

ISBN : 9782412077689
Dépôt légal : avril 2022

Traduction de l'anglais : Gabriel Picarde
Mise en page : Pierre Brandeis

Tous droits réservés. Toute reproduction, même partielle,
du contenu, de la couverture ou des icônes, par quelque
procédé que ce soit (électronique, photocopie, bande

magnétique ou autre) est interdite sans autorisation par écrit d'Édi8.

Limites de responsabilité et de garantie. L'auteur et l'éditeur de cet ouvrage ont consacré tous leurs efforts à préparer ce livre. Édi8 et les auteurs déclinent toute responsabilité concernant la fiabilité ou l'exhaustivité du contenu de cet ouvrage. Ils n'assument pas de responsabilité pour ses qualités d'adaptation à quelque objectif que ce soit, et ne pourront être en aucun cas tenus responsables pour quelque perte, profit ou autre dommage commercial que ce soit, notamment mais pas exclusivement particulier, accessoire, conséquent, ou autre.

Marques déposées. Toutes les informations connues ont été communiquées sur les marques déposées pour les produits, services et sociétés mentionnés dans cet ouvrage. Édi8 décline toute responsabilité quant à l'exhaustivité et à l'interprétation des informations. Tous les autres noms de marques et de produits utilisés dans cet ouvrage sont des marques déposées ou des appellations commerciales de leur propriétaire respectif. Édi8 n'est lié à aucun produit ou vendeur mentionné dans ce livre. Le logo O'Reilly est une marque déposée de O'Reilly Media, Inc.

Ce livre numérique a été converti initialement au format EPUB par Isako www.isako.com à partir de l'édition papier du même ouvrage.

Introduction

Ce livre s'intéresse au domaine de la science des données (*data science*) en relation avec le langage de programmation Python. La première question qui se pose est celle-ci : mais qu'est-ce que cette science des données ? Définir cette activité est délicat alors même que le terme est déjà assez répandu. Certains ont rejeté le terme en le considérant superflu (après tout, qui dit science, dit données !) et d'autres pensent que ce n'est qu'un ronflant terme de cosmétique pour embellir un *curriculum vitae* et séduire les recruteurs angoissés à l'idée de rater la prochaine grosse tendance dans les emplois « techno ».

Je trouve que ces critiques oublient un point fondamental. Malgré le brouhaha autour de cette appellation, la science des données convient pour désigner un ensemble de compétences multidisciplinaires qui sont devenues indispensables dans plusieurs secteurs de l'industrie et du savoir. C'est cette attitude du croisement de plusieurs disciplines qui est la clé : je trouve que la plus efficace description en est donnée par le diagramme de Venn de la

datalogie imaginé par Drew Conway et publié sur son blog en septembre 2010 ([Figure 0.1](#)).



[Figure 0.1](#) : Diagramme de Venn de la datalogie selon Drew Conway.



(N.d.T) Un terme que nous proposons pour la science des données est *datalogie*.

Certains termes inscrits dans les intersections de la figure ne sont pas très rigoureux, mais l'essence de ce que je considère comme science des données est bien représentée : c'est d'abord un travail à la confluence de trois domaines de maîtrise : celui des statistiques et de la connaissance des modèles et de la synthèse des jeux de données (dont le volume ne cesse de croître) ; le domaine de l'informatique, en réclamant des capacités de conception et d'exploitation d'algorithmes pour stocker, traiter et visualiser les données ; enfin, une expertise du domaine d'application qui permet de

formuler les bonnes questions et de contextualiser les réponses (ce qui correspond à une formation classique).

Je vous suggère de ce fait de considérer la datalogie non comme un nouveau domaine de connaissances, mais comme un nouveau jeu de compétences dans le cadre de votre domaine d'expertise actuel : traitement des résultats d'élections, rentabilité d'actions en Bourse, optimisation des clics sur les annonces Web, identification de micro-organismes au microscope, découverte de nouvelles classes d'objets astronomiques, *etc.* Le but du livre est de vous doter des moyens de poser des questions et d'y répondre dans le secteur de connaissances choisi.

À qui s'adresse ce livre

Lors de mes exposés à l'université de Washington et lors des conférences et rencontres techniques, on me demande souvent comment il faudrait apprendre à programmer en Python. Ceux qui m'interrogent sont souvent des étudiants férus de technologie, des codeurs ou des chercheurs ; la plupart ont une bonne expérience de programmeur et d'utilisateur d'outils numériques et ne rêvent pas de savoir écrire en Python pour le plaisir, mais parce qu'ils ont des besoins de traitement de données. On trouve sur le Web des dizaines de tutoriels et de forums, mais je n'ai rien repéré qui réponde à l'attente de mes demandeurs. Et c'est pourquoi j'ai conçu ce livre.

Vous n'y trouverez pas un guide pour apprendre à programmer en Python ; je présuppose que le lecteur en sait un minimum en Python : définir une fonction, affecter une variable, appeler une méthode d'objet, écrire un accès à une liste, *etc.* Le but est de vous guider dans la prise en main d'une série de librairies consacrées aux traitements de datalogie (NumPy, Pandas, Matplotlib, Scikit-Learn, et autres) au moyen de la version interactive de Python nommée IPython.

Pourquoi Python ?

Python s'est affirmé en deux décennies comme un excellent outil pour les travaux d'informatique scientifique, en particulier l'analyse et la représentation de jeux de données volumineux. Les premiers utilisateurs du langage s'en sont étonnés, car ces domaines d'emploi n'étaient pas envisagés au départ.

L'intérêt de Python en datalogie est avant tout lié à l'existence d'un vaste écosystème technique dont les principales incarnations sont des librairies réputées :

- NumPy pour la manipulation de données tabulaires hétérogènes ;
- Pandas pour la manipulation de données hétérogènes et libellées ;
- SciPy pour les opérations de calcul numérique ;
- Matplotlib pour la production de visualisations graphiques de haute qualité ;
- Scikit-Learn pour l'apprentissage machine ;
- et d'autres outils qui seront présentés le moment voulu.

Pour profiter d'un bon confort dans vos séances pratiques, je vous conseille d'adopter l'outil à interface Web IPython et son concept de calepin partageable.

Si vous cherchez un ouvrage consacré à la programmation en Python, lisez *Programmer en Python* de Luciano Ramalho (First, 2019).

J'ai par ailleurs écrit un petit livre de moins de 100 pages intitulé *A Whirlwind Tour of the Python Language* en guise d'introduction au présent livre. (N.d.T : ce titre n'est pas disponible en français.)

Python 2 ou Python 3 ?

Ce livre fait définitivement le saut vers la syntaxe de Python 3, pour profiter des enrichissements qu'apporte cette version majeure de Python. Bien qu'apparu en 2008, Python 3 a mis du temps à se diffuser, notamment dans le monde scientifique et chez les concepteurs Web parce que les librairies n'ont pas toujours basculé vers la version 3 au plus vite. Cela dit, depuis 2014, tous les outils essentiels existent pour Python 3. La plupart des exemples du livre pourraient néanmoins fonctionner sous Python 2.

Organisation de ce livre

Ce livre réunit cinq grands chapitres ; chacun sauf le premier est dédié à une librairie qui constitue un composant incontournable d'un « atelier de datalogie pratique avec Python » :

IPython et Jupyter ([Chapitre 1](#))

Ces deux paquetages forment un environnement d'exécution complet pour réaliser des projets de datalogie avec Python.

NumPy ([Chapitre 2](#))

Cette librairie définit notamment la classe d'objets ndarray pour stocker et manipuler efficacement des tableaux de données denses.

Pandas ([Chapitre 3](#))

Cette librairie définit notamment la classe DataFrame dédiée aux données en colonnes et avec labels.

Matplotlib ([Chapitre 4](#))

Cette librairie offre des capacités de rendu visuel graphique des données (histogrammes, nuages de points, etc.).

Scikit-Learn ([Chapitre 5](#))

Cette librairie regroupe l'implémentation en Python de plus d'une douzaine d'algorithmes éprouvés d'apprentissage machine (régression, k-moyennes, bayésien, SVM, PCA, etc.).

Le monde de PyData ne se limite pas à ces paquetages, d'autant qu'il s'enrichit au quotidien. J'ai donc essayé d'indiquer le cas échéant les autres projets et paquetages qui apportent de nouvelles possibilités au domaine visé avec Python. Pour autant, les quatre librairies que je décris en détail sont fondamentales dans le secteur de la datalogie avec Python, et je pronostique qu'elles le resteront grâce à l'écosystème dont elles sont entourées.

Contenu additionnel

Le site de l'éditeur comporte une page dédiée au livre. Rendez-vous sur www.editionsfirst.com, puis cherchez le titre du livre. Une fois sur sa fiche, cliquez sur la rubrique *Contenu additionnel*. Vous y trouverez :

- Un fichier d'archive compressée avec plusieurs dizaines de fichiers calepins *.ipynb* (Notebook d'IPython). Chaque calepin contient tous les exemples de code Python d'une section d'un chapitre ainsi que les principaux sous-titres tirés du livre pour faciliter votre repérage.
- Un fichier PDF contenant une sélection des figures que l'impression du livre en noir et blanc ne permet pas d'apprécier à leur juste mesure. Vous pouvez ainsi les étudier en couleurs.

Exemples et évolution des librairies

Il reste possible qu'une évolution d'une librairie ait pour résultat qu'un des exemples ne fonctionne plus. Nous avons par exemple constaté (et adapté) que `factorplot()` a été remplacé par `catplot()`, que le paramètre `size` est devenu `height` dans `pairplot()`, que `normed` s'écrit dorénavant `density`, *etc.*).

Si vous rencontrez cette situation, et avant de remonter les manches pour chercher vous-même, allez vérifier sur le site de l'éditeur au cas où un errata y aurait été déposé.



(N.d.T) Nous avons retesté tous les exemples et vous proposons un fichier archive des calepins dans la page du livre sur le site www.editionsfirst.com (cherchez le titre du livre puis sur fiche, la rubrique *Contenu additionnel*).

Droits de réutilisation des exemples

Si vous devez, pour une raison ou une autre, vous resservir de parties de ce code pour votre propre usage, faites-le librement. En revanche, dans le cadre d'une publication, il vous est demandé de citer la source. C'est une question de respect du droit d'auteur en particulier, et du droit des auteurs en général. Voici un modèle de citation :

Titre, auteur, éditeur, ISBN.

Python pour la Data science par Jake VanderPlas (First),
9782412070048.

Version anglaise des errata et des exemples

La page suivante permet de se renseigner sur une éventuelle correction apportée au code :

<https://github.com/jakevdp/PythonDataScienceHandbook>

Préquis d'installation

Pour pratiquer les exercices, il faut installer Python et une série de librairies, ce qui est très facile. Suivez le guide.

Parmi les multiples façons possibles d'installer Python pour faire de la datalogie, je conseille la distribution Anaconda. Elle s'utilise de la même manière sous Windows, Linux ou macOS X. Il existe deux variantes d'Anaconda :

- Miniconda, qui réunit l'interpréteur Python et un outil sur ligne de commande nommé conda qui permet une gestion des paquetages multiplateformes, ressemblant en cela à apt ou yum, que les linuxiens connaissent sans doute.
- Anaconda, qui installe lui aussi Python et conda, mais aussi une série de paquetages dédiés à la datalogie. Cette installation complète va évidemment demander plusieurs gigaoctets d'espace libre et un peu de patience.

Si vous installez Miniconda, vous pouvez à tout moment y ajouter manuellement les paquetages qu'installe Anaconda. C'est pourquoi je vous suggère de commencer avec Miniconda.

Téléchargez et installez le paquetage de Miniconda en vérifiant que vous optez pour la version pour Python 3. Installez ensuite les paquetages primaires dont vos aurez besoin, ainsi :

```
[~]$ conda install numpy pandas scikit-learn  
matplotlib seaborn ipython-notebook
```

De temps à autre, je fais appel à une autre librairie plus spécialisée. Pour l'installer, il suffit de profiter de l'outil conda comme ceci :

```
conda install nompaquetage
```

Pour tous détails à propos de conda, notamment pour créer un environnement (fortement conseillé), voyez sa documentation en ligne.

Mise en pratique des exemples

La grande majorité des blocs de code sont exécutés en un instant dès que vous le demandez par le raccourci **Maj-Entrée** (ou **Retour**). Quelques-uns, notamment ceux qui supposent d'aller récupérer un fichier de données, peuvent réclamer un peu de patience. Ne concluez pas trop vite que le programme a planté. L'un des exemples demande même plusieurs minutes.

In[] et Out[]

Les numéros qui apparaissent dans chaque calepin Jupyter Notebook pour les sections de saisie :

In[99] :

et d'affichage des résultats :

Out[99] :

varient en fonction du nombre d'exécutions depuis la dernière ouverture de ce calepin. Ils ne vont donc pas coïncider nécessairement avec ceux du livre.

Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Met en valeur un terme fondamental et indique les noms de fichiers et les adresses Web.

Police à espacement fixe

Police utilisée pour les listings des programmes, ainsi que dans le corps du texte pour faire référence aux éléments de programme tels que les noms de variables ou de fonctions, les types de données, les variables d'environnement, les déclarations et les mots-clés.

Police à espacement fixe en gras

Dans les sessions interactives des calepins restituées dans le livre, apparaît en gras ce qui est à saisir par l'utilisateur.



Cet élément signifie une note générale. (N.d.T) indique une remarque spécifique à l'édition française.

Terminologie française

Plus encore que dans le domaine de la programmation informatique, la terminologie mérite un soin particulier en datalogie. Voici nos principales décisions.

Terme utilisé	Terme anglais	Commentaires si nécessaire
1 sur n	<i>one-hot</i>	Encodage binaire dont tous les bits sauf un sont forcés à 0. On rencontre aussi « 1 parmi n ».
aberrant	<i>outlier</i>	Une valeur anormalement éloignée des autres observations.
binning	<i>binning</i>	Répartition de valeurs uniquement numériques en plusieurs bacs ou baquets (<i>bins</i>).
baquet	<i>bucket</i>	Ou encore « godet » ou bien « compartiment » : réceptacle d'une distribution des données.
calepin (de Jupyter)	<i>notebook</i>	Le terme « notebook » sert d'abord à désigner un ordinateur portable. Le terme « calepin » permet de bien distinguer le format de fichier hybride qui combine des blocs de code source et des commentaires et graphiques.
datalogie	<i>data science</i>	L'expression « science des données » tente de remplacer « datascience ». « Datalogie » utilise le même mécanisme de construction que « technologie ».

datalogue	<i>data</i>	Personne produisant des informations essentielles (des discours, <i>logos</i>) par distillation probabiliste de données brutes.
enjambement	<i>stride</i>	Valeur entière supérieure à 1 qui détermine le nombre d'octets occupés par chaque élément de données dans un tableau et permet ainsi de progresser parmi les éléments.
erreur quadratique	<i>Mean Squared Error</i>	S'il est nécessaire, nous conservons le sigle anglais MSE.
moyenne	<i>Error</i>	
exétron	<i>thread</i>	Sous-processus d'exécution appelé également fil d'exécution.
grappe, groupe	<i>cluster</i>	Résultat de l'opération de partitionnement ou regroupement (<i>clustering</i>).
librairie	<i>library</i>	Terme à privilégier en remplacement de <i>bibliothèque</i> .
surajustement	<i>overfitting</i>	On utilise aussi « surapprentissage ».

CHAPITRE 1

IPython, plus loin que Python

Le langage Python dispose d'un vaste choix d'environnements et d'ateliers de développement. On me demande souvent lequel j'utilise personnellement. Certains sont étonnés de ma réponse : mon environnement préféré est IPython (<http://ipython.org/>) associé à un éditeur de texte, qui est soit Emacs, soit Atom (selon mon humeur). Le projet IPython (interactive Python) a démarré en 2001. Fernando Perez avait besoin d'un interpréteur Python plus puissant ; son projet est devenu entre-temps, selon ses propres termes, « un outil couvrant la totalité du cycle de vie de l'informatique de recherche ». Si on peut considérer Python comme le moteur de nos traitements de datalogie, IPython constitue son tableau de contrôle interactif.

Non seulement IPython est une interface interactive pour Python, mais il apporte plusieurs compléments syntaxiques au langage lui-même, et nous en découvrirons la majorité dans ce livre. De plus, IPython est intimement lié à un autre projet nommé Jupyter (<http://jupyter.org>). C'est une sorte d'éditeur fonctionnant dans un navigateur Web au moyen

duquel on peut développer, collaborer, partager et même publier des traitements et des résultats de datalogie. Le calepin d'IPython est en fait un cas particulier du principe de calepin de Jupyter, car il en existe également pour Julia, pour R et pour d'autres langages.



(N.d.T) : La version américaine de ce livre a entièrement été rédigée et mise en pages dans des calepins IPython combinant le texte et les lignes de code.

L'objectif d'IPython est d'utiliser le langage Python de manière efficace dans les travaux scientifiques et de datalogie. Commençons par découvrir les caractéristiques qui nous seront utiles dans ces activités, notamment les éléments de syntaxe qui viennent compléter l'offre standard de Python. Nous verrons ensuite certaines des commandes magiques qui peuvent abréger la création et l'utilisation du code en datalogie, et nous verrons pour finir celles qui sont très utiles à la mise au point de vos projets.

Interpréteur shell ou calepin ?

IPython peut être utilisé de deux façons que nous verrons dans ce chapitre : en mode texte, dans une fenêtre d'interpréteur shell, ou sous forme d'un calepin IPython dans un navigateur Web, comme nous venons de l'expliquer. L'essentiel du code de ce chapitre s'applique aux deux approches. Les exemples utiliseront le shell ou le calepin en fonction du sujet. Lorsqu'une description ne concerne que l'une des deux variantes, j'expliquerai pourquoi. Commençons par voir comment démarrer l'interpréteur shell IPython et le calepin IPython.

Démarrage du shell IPython

Ce chapitre, ainsi que les suivants, n'a pas été conçu pour une lecture passive. Je vous conseille fortement de pratiquer les exemples et d'expérimenter les outils tout en progressant dans la lecture. En mettant votre corps à contribution, vous mémoriserez beaucoup mieux les techniques que je propose. Commençons par démarrer l'interpréteur IPython en saisissant tout simplement `ipython` depuis la ligne de commande dans un terminal ou une fenêtre de commande. Si vous avez installé une

distribution complète telle qu'Anaconda ou EPD, vous trouverez sans doute une icône de démarrage.

Une fois que l'outil a démarré, vous devriez voir apparaître ce genre de message :

```
Python 3.8.8 (default, Apr 13 2021, 15:08:03)
[MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for
more information
IPython 7.22.0 -- An enhanced Interactive
Python. Type '?' for help.
```

In [1]:

Vous êtes ainsi prêt à poursuivre.

Démarrage du calepin Jupyter

Le calepin Jupyter est un éditeur à interface graphique hébergé par votre navigateur Web. Cette interface contrôle l'outil shell d'IPython et offre de riches possibilités d'affichage. Dans le calepin, vous pouvez exécuter des instructions Python et IPython, contrôler le format du texte, obtenir des visualisations statiques et dynamiques, ajouter des équations mathématiques, des widgets JavaScript, etc. Vous pouvez sauvegarder un calepin pour le transmettre à

d'autres personnes qui pourront voir les résultats et réexécuter le code inséré.

Le calepin IPython fonctionne dans un navigateur Web, mais il doit pouvoir se connecter à un processus Python en exécution. Pour démarrer ce processus qui correspond au noyau (kernel), vous lancez la commande suivante dans la fenêtre shell :

```
$ jupyter notebook
```

Vous demandez ainsi le démarrage d'un serveur Web local qui deviendra accessible au navigateur. Quelques lignes de messages sont affichées pour confirmer le bon démarrage :

```
$ jupyter notebook
[NotebookApp] Serving notebooks from local
directory: /Users/moi/...
[NotebookApp] 0 active kernels
[NotebookApp] The IPython Notebook is running
at: http://localhost:8888/
[NotebookApp] Use Control-C to stop server and
shut down all kernels...
```

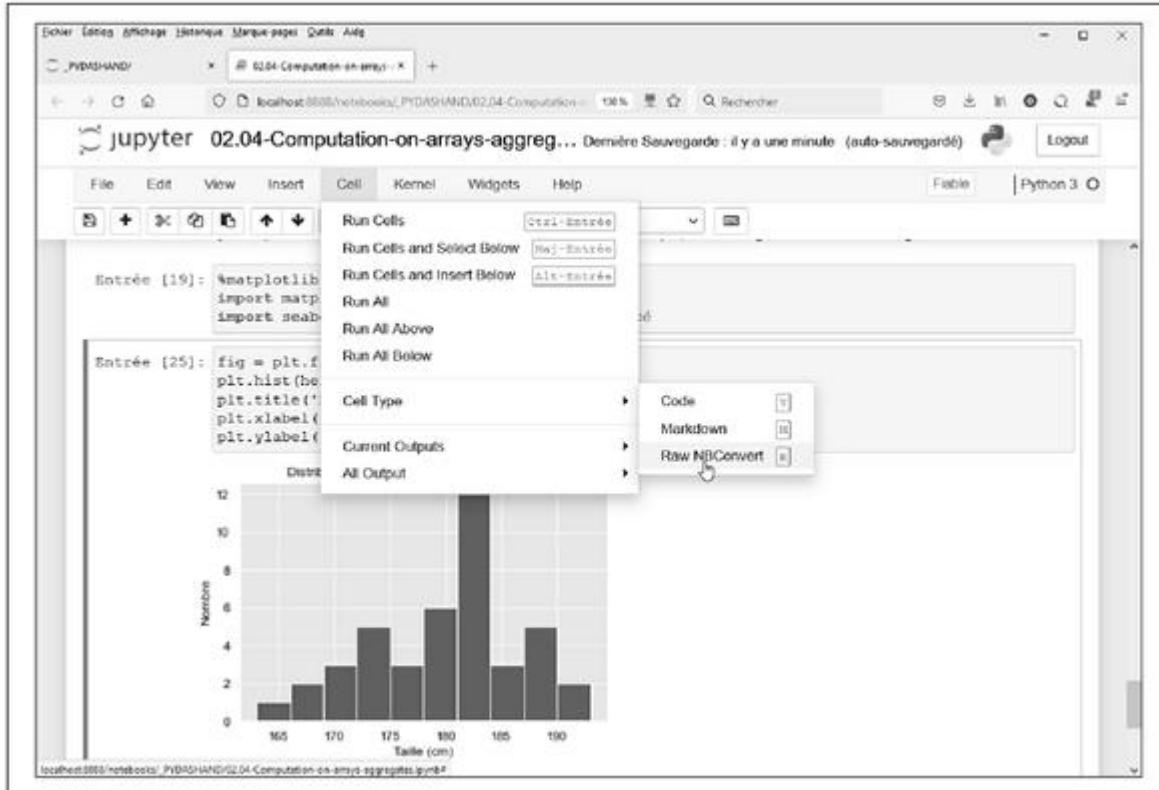


Figure 1.1 : Aperçu de l'atelier Jupyter.

La commande doit être immédiatement suivie du démarrage du navigateur Web et de l'affichage de la page à l'adresse URL locale mentionnée (qui peut varier selon le système). Si votre navigateur n'apparaît pas automatiquement, ouvrez une fenêtre puis demandez manuellement d'accéder à cette adresse (dans l'exemple, elle s'écrit <http://localhost/8888/>).

Aide et documentation d'IPython

S'il y a une section à ne pas négliger dans ce chapitre, c'est celle-ci. Les outils qui y sont présentés constituent les principales contributions d'IPython au travail quotidien.

Lorsqu'un connaisseur en informatique est sollicité par un membre de sa famille ou un ami pour l'aider à résoudre un problème d'ordinateur, il ne s'agit pas en général de connaître forcément la réponse, mais de connaître la technique qui permettra de trouver cette réponse rapidement. Il en va de même en datalogie : les ressources Web textuelles telles que les documentations en ligne, les fils de discussion et les forums de type Stack Overflow offrent des réponses, notamment lorsque le sujet fait partie de ceux pour lesquels vous avez déjà fait des recherches. Pour être efficace en datalogie, il s'agit moins de mémoriser les outils ou les commandes qu'il faut utiliser dans chacune des situations possibles, que d'apprendre comment trouver l'information qui manque, sur le Web ou ailleurs.

Une fonction vraiment pratique d'IPython avec Jupyter permet de réduire la distance entre vous et les documentations et recherches qui vont vous aider à progresser dans votre travail. Les recherches Web restent

utiles pour trouver des réponses aux questions complexes, mais vous disposez déjà d'une montagne d'informations au niveau local d'IPython. Voici quelques questions auxquelles IPython répond en quelques touches :

- Comment s'appelle cette fonction ? Quelles sont ses options ?
- À quoi ressemble le code source de cet objet Python ?
- Que contient ce paquetage que je viens d'importer ?
- Quels sont les attributs et méthodes de cet objet ?

Les outils Python pour accéder rapidement à ce genre d'information correspondent au caractère ? pour avancer dans la documentation, le double caractère ?? pour accéder au code source et la touche TAB pour l'aide à la saisie.

Accès à la documentation avec ?

Aussi bien le langage Python que son écosystème pour la datalogie n'oublient pas l'utilisateur. L'accès à la documentation a donc été soigné. Chaque objet Python est doté d'une référence vers une chaîne qui est la chaîne de documentation, *docstring*. Cette chaîne contient une description rapide des modalités d'usage de l'objet. Python propose la fonction intégrée `help()` pour accéder à cette

information. Voici par exemple comment accéder à la documentation de la fonction standard len() :

```
In [1]: help(len)
Help on built-in function len in module
builtins:
```

```
len(...)
    len(obj, /)
```

Return the number of items in a container.

Les informations peuvent s'afficher sous forme de texte en ligne ou dans une fenêtre volante, selon l'outil choisi (terminal ou calepin).

Obtenir de l'aide pour un objet étant une question fréquente, IPython propose d'ajouter le caractère ? en suffixe du nom pour accéder à la documentation :

```
In [2]: len?
Type:    builtin_function_or_method
String form: <built-in function len>
Namespace: Python builtin
Docstring:
len(object) -> integer
Return the number of items of a sequence or
mapping.
```

Cette notation est même valable pour les méthodes des objets :

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Type:    builtin_function_or_method
String form: <built-in method insert of list
object at 0x1024b8ea8>
Docstring: L.insert(index, object) -- insert
object before index
```

Et bien sûr pour les objets, afin d'accéder à la documentation du type :

```
In [5]: L?
Type:    list
String form: [1, 2, 3]
Length: 3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from
iterable's items
```

Notez que cette aide fonctionne également pour les fonctions et les objets que vous créez vous-même ! Commençons par définir une petite fonction elevcarr avec sa chaîne docstring :

```
In [6]:  
def elevcarr(a):  
    """Renvoie le carré de a."""  
    return a ** 2
```

Pour créer une docstring, il faut placer la chaîne littérale à partir de la première ligne du corps de fonction. Nous utilisons la notation basée sur le triple guillemet droit parce que c'est la convention Python pour les chaînes s'étendant sur plusieurs lignes.

Nous pouvons tester l'accès à cette documentation avec `elevcarr?` :

```
In [7]: elevcarr?  
Signature: elevcarr(a)  
Docstring: Renvoie le carré de a.  
File:      c:\users\sergent\<ipython-input-1-  
15b60f82f858>  
Type:      function
```

Cette possibilité devrait vous inviter à toujours prendre le temps d'ajouter la documentation en ligne en même temps que le code que vous écrivez.

Accès au code source avec ??

Lorsqu'un objet attise votre curiosité, n'hésitez pas à profiter de la bonne visibilité du langage Python pour

demandeur d'accéder à son code source. Dans IPython, le raccourci correspondant est le double signe ? :

```
In [8]: elevcarr??  
Signature: elevcarr(a)  
Source:  
def elevcarr(a):  
    """Renvoie le carré de a."""  
    return a ** 2  
File:      c:\users\sergent\<ipython-input-1-  
15b60f82f858>  
Type:      function
```

L'affichage du code source d'une fonction aussi simple permet immédiatement de tout savoir à son propos.

Vous remarquerez que l'opérateur ?? n'affiche pas le code source dans tous les cas. C'est en général lié au fait que l'objet désiré n'a pas été écrit en Python, mais en C ou dans un autre langage compilé. Dans ce cas, la demande donne le même résultat que l'opérateur simple ?. C'est le cas par exemple pour len() :

```
In [5]: len??  
Signature: len(obj, /)  
Docstring: Return the number of items in a  
container.  
Type:      builtin_function_or_method
```

Les deux opérateurs ? et ?? constituent donc d'excellents outils pour obtenir des informations à propos des éléments constituant le langage Python.

Exploration des modules avec Tab

IPython offre un autre mécanisme de soutien : la frappe de la touche de tabulation sert à terminer une saisie d'identifiant ou de mot-clé, mais également à explorer les descriptions des objets, des modules et des espaces de noms. Dans les exemples, nous écrirons <Tab> pour symboliser la frappe de cette touche.

Aide à la saisie des noms définis dans les objets

Un objet Python est doté d'attributs et de méthodes. Vous pouvez en obtenir la liste au moyen de la fonction standard de python dir(), mais il est plus simple d'utiliser le mécanisme basé sur la touche Tab. Vous pouvez par exemple saisir le nom de l'objet suivi du signe point et frapper Tab pour obtenir la liste de tous ses attributs et méthodes :

```
In [8]: L = [1,2,3]
```

```
In [9]: L.<Tab>
append() count() insert() reverse()
```

```
clear()    extend()   pop()     sort()
copy()     index()    remove()
```

Vous pouvez saisir plus de caractères pour restreindre le nombre de réponses proposées, avant de frapper Tab :

```
In [9]: L.c<Tab>
        clear() count()

        copy()
```

Lorsqu'il ne reste qu'un choix possible, il suffit de frapper à nouveau Tab pour faire saisir l'identifiant complet. La saisie suivante est automatiquement remplacée par elle :

```
In [9]: L.cou<Tab>
```

Le langage Python n'impose pas de distinction entre les attributs et méthodes publiques (externes) et ceux internes (à accès privé). En revanche, une convention demande de faire commencer les identifiants internes par un caractère souligné (*underscore*). Pour éviter un affichage trop long, les méthodes privées et spéciales ne sont pas proposées dans la liste, mais vous pouvez en prendre connaissance en saisissant ce caractère de soulignement :

```
In [9]: L._<Tab>
__add__          __eq__          __iadd__
__lt__
```

<code>__class__</code>	<code>__format__()</code>	<code>__imul__</code>
<code>__mul__</code>		
<code>__contains__</code>	<code>__ge__</code>	<code>__init__</code>
<code>__ne__</code>		
<code>__delattr__</code>	<code>__getattribute__</code>	
<code>__init_subclass__()</code>	<code>__new__()</code>	
<code>__delitem__</code>	<code>__getitem__()</code>	<code>__iter__</code>
<code>__reduce__()</code>		
<code>__dir__()</code>	<code>__gt__</code>	<code>__le__</code>
<code>__reduce_ex__()</code>		
<code>__doc__</code>	<code>__hash__</code>	<code>__len__</code>
<code>__repr__</code>		
<code>__reversed__()</code>	<code>__rmul__</code>	<code>__setattr__</code>
<code>__setitem__</code>		
<code>__sizeof__()</code>	<code>__str__</code>	
<code>__subclasshook__()</code>		

La plupart des méthodes sont à double caractère souligné (méthodes surnommées *dunder* ou double-souligné).

Aide à la saisie des importations avec Tab

L'aide à la saisie avec Tab sera également très utile pour importer des composants des paquetages. Voici comment afficher les noms de tous ceux du paquetage `itertools` qui commencent par `co` :

```
In [9]: from itertools import co<Tab>
          combinations()
          compress()
```

```
combinations_with_replacement() count()
```

Vous pouvez également vérifier quelles sont les imports possibles sur votre système, en fonction des scripts et des modules accessibles à votre session Python :

```
In [10]: import<Tab>
2to3-script                      anaconda-project-
script      appdirs
abc
Apple
adodbapi
anaconda_navigator          argh
afxres                         anaconda_project
argon2      >
aifc                            antigravity
argparse
alabaster
array
anaconda-navigator-script     AppData
asadmin-script
```

```
In [10]: import h<Tab>
h5py    heapq   html5lib
hashlib  hmac    http
heapdict html
```

Ceci n'est bien sûr qu'un extrait des plus de 400 paquetages et modules disponibles pour mon système.

Utilisation du caractère générique *

La touche Tab est pratique à partir du moment où vous connaissez le début du nom de l'élément recherché. En revanche, si vous ne connaissez que le milieu ou la fin du mot, il faut une autre solution. IPython propose le métacaractère *.

Voici comment dresser la liste de tous les objets dans l'espace de noms qui se termine par Warning :

```
In [10]: *Warning?  
BytesWarning  
DeprecationWarning  
FutureWarning  
ImportWarning  
PendingDeprecationWarning  
ResourceWarning  
RuntimeWarning  
SyntaxWarning  
UnicodeWarning  
UserWarning  
Warning
```

Sachez que le métacaractère * trouve toutes les chaînes, y compris la chaîne vide.

Supposons que nous cherchions une méthode de chaîne qui contient le mot find dans son nom. Voici comment nous pourrions la trouver :

```
In [11]: str.*find*?
```

```
str.find
```

```
str.rfind
```

Je trouve cette recherche par métacaractère très pratique pour localiser une commande lors de l'exploration d'un nouveau paquetage ou après l'avoir laissé de côté un certain temps.

Raccourcis clavier du shell IPython

Dès que vous passez un minimum de temps sur un ordinateur, vous prenez sans doute soin d'apprendre quelques raccourcis clavier. Les plus connus sont bien sûrs Ctrl-C et Ctrl-V pour copier/coller depuis le presse-papiers et ils sont standardisés. Les utilisateurs chevronnés profitent des possibilités des éditeurs de texte tels qu'Emacs, Vim et autres, qui offrent une vaste palette de fonctions disponibles par des combinaisons sophistiquées de frappes clavier.

L'interpréteur d'IPython ne va pas aussi loin, mais il propose un certain nombre de raccourcis pour aller plus vite dans la saisie des commandes. En réalité, ces raccourcis ne sont pas définis par IPython, qui profite d'une dépendance par rapport à la librairie GNU Readline. C'est pourquoi certains des raccourcis pourront différer de ce que nous montrons dans la suite, à cause de la configuration de votre système. Certains raccourcis fonctionnent aussi dans le calepin Web, mais intéressons-nous d'abord aux raccourcis accessibles dans l'interpréteur shell d'IPython.

Une fois que vous avez pris la peine de les apprendre, la plupart de ces raccourcis vous permettent d'émettre un

certain nombre de commandes sans déplacer les mains par rapport à la position de repos naturelle. Vous ne serez pas dépaysé si vous utilisez déjà l'éditeur Emacs ou connaissez un interpréteur de type Linux. Nous avons classé ces raccourcis en plusieurs catégories : navigation, saisie de texte, historique de commandes et divers.

Raccourcis de navigation

Les touches de flèche Gauche et Droite sont évidemment prévues pour se déplacer horizontalement dans une ligne de texte, mais vous pouvez profiter des raccourcis pour ne pas devoir repositionner les mains :

Raccourci	Action
Ctrl-a	Ramène le curseur en début de ligne.
Ctrl-e	Déplace le curseur en fin de ligne.
Ctrl-b (flèche Gauche)	Recule d'un caractère vers la gauche.
Ctrl-f (flèche Droite)	Avance d'un caractère vers la droite.

Raccourcis de saisie de texte

Tout le monde utilise la touche Retour arrière (*Backspace*) pour supprimer le caractère précédent, mais cela oblige à déplacer la main et ne supprime qu'un caractère à la fois. IPython offre plusieurs actions ayant un effet plus large. La

plus utilisée supprime toute une ligne de texte. Ceux qui utilisent déjà les raccourcis qui évitent la touche Retour arrière ont sans doute déjà mémorisé ces raccourcis d'édition.

Raccourci Action

Ret. Arr.	Supprime le caractère précédent.
Ctrl-d	Supprime le caractère suivant.
Ctrl-k	Coupe tout le texte entre le curseur et la fin de ligne.
Ctrl-u	Coupe tout le texte entre le début de ligne et le curseur.
Ctrl-y	Colle en place le texte coupé (Yank).
Ctrl-t	Intervertit les deux caractères précédents.

Raccourcis d'historique de commandes

La famille de raccourcis qui fait gagner beaucoup de temps est celle qui permet de rappeler une des anciennes commandes émises. Cet historique ne se limite pas à la session IPython en cours. Toutes les commandes saisies sont stockées dans une base SQL Lite dans le répertoire du profil IPython concerné. Vous rappelez les commandes en utilisant les deux touches flèche Haut et Bas, mais vous disposez d'autres options.

Raccourci	Action
Ctrl-p (flèche Haut)	Affiche la précédente commande émise.
Ctrl-n (flèche Bas)	Affiche la commande suivante émise (si pas sur la dernière).
Ctrl-r	Inverse le sens de la recherche dans l'historique.

La recherche inversée est particulièrement pratique. Vous vous souvenez que nous avons défini dans la section précédente une fonction pour calculer un carré. Passons en revue l'historique des commandes Python, en démarrant d'abord une nouvelle session d'interpréteur IPython (le numéro d'étape est à nouveau 1). Cherchons notre définition de fonction. Voici ce qui apparaît lorsque vous frappez le raccourci Ctrl-r :

```
In [1]:  
(reverse-i-search) '':
```

Dès que vous commencez à saisir des caractères, IPython va autoremplir avec la commande précédente convenant aux caractères déjà saisis :

```
In [1]:  
(reverse-i-search)'ele': elevcarre??
```

Vous pouvez continuer à taper des caractères pour limiter les choix ou chercher une autre commande en frappant Ctrl-r. Si vous suivez exactement l'exemple, le fait de frapper deux fois le raccourci donne ceci :

In [1]:

```
(reverse-i-search)'sqa': def elevcarre(a):  
def elevcarr(a):  
    """Renvoie le carré de a."""  
    return a ** 2
```

Dès que vous avez trouvé la commande à lancer, il suffit de valider par la touche Entrée. Nous utilisons alors la commande :

In [1]: def elevcarr(a):

```
    """Renvoie le carré de a."""  
    return a ** 2
```

In [2]: elevcarre(2)

Out[2]: 4

Vous pouvez chercher dans l'historique avec les deux raccourcis Ctrl-p et Ctrl-n ou les flèches Haut et Bas du clavier, mais uniquement pour trouver les caractères depuis le début de ligne. Par exemple, si vous saisissez def puis frappez Ctrl-P, vous verrez s'afficher la plus récente commande éventuelle de l'historique qui commence par def.

Raccourcis divers

Nous disposons enfin de quelques raccourcis à usage général :

Raccourci Action

Ctrl-l	Efface l'affichage.
Ctrl-c	Interrompt la commande en cours.
Ctrl-d	Quitte la session IPython.

Vous aurez sans doute besoin d'utiliser de temps à autre la commande Ctrl-c lorsque vous aurez lancé un traitement trop long ou aurez des soupçons sur sa bonne exécution.

Certains des raccourcis peuvent sembler un peu biscornus, mais vous les mémoriserez très vite si vous pratiquez. Une fois que vous les aurez bien intégrés, je parie que vous chercherez à disposer de la même vivacité dans d'autres applications.

Commandes magiques d'IPython

Dans les deux précédentes sections, nous avons vu comment IPython permettait d'exploiter le langage Python de façon plus efficace et interactive. Découvrons maintenant quelques enrichissements fonctionnels qu'apporte IPython à la syntaxe Python normale. Ces compléments sont appelés des commandes magiques et leur nom commence toujours par le caractère %.

Ces commandes magiques incarnent des fonctions de soutien très utiles dans un contexte d'analyse de données. Il en existe deux sortes : les magiques de ligne qui sont préfixées par un seul signe % et ne traitent qu'une ligne de saisie, et les magiques de cellule à préfixe % redoublé qui gèrent un bloc de lignes. Voyons quelques exemples brefs, car nous reviendrons plus en détail sur certaines commandes magiques dans la suite du chapitre.

Collage d'un bloc de code avec %paste et %cpaste

Quand vous éditez du code avec l'interpréteur IPython, il vous arrivera certainement de tomber sur un résultat incorrect après avoir collé un bloc de code, notamment à cause d'un problème d'indentation et de marqueur de suite

du bloc. C'est le cas lorsque vous voulez copier/coller un exemple depuis un site Web vers l'interpréteur. Partons de la fonction très simple suivante :

```
>>> def rienfaire(x):  
    return x
```

Le format du code se présente tel qu'on peut le voir dans l'interpréteur Python classique. Si vous copiez/collez directement ce bloc dans IPython, vous obtenez une erreur.

En effet, l'interpréteur ne sait pas quoi faire des caractères de suite de bloc. Si vous utilisez la commande magique %paste, le collage devient exploitable.

L'autre commande de collage, %cpaste, ouvre une invite multiligne interactive dans laquelle vous pouvez insérer plusieurs lignes de code à votre rythme. Vous sortez de ce mode de saisie avec deux tirets --.

Les commandes magiques permettent de profiter d'un certain nombre de fonctions qui sont difficiles, voire impossibles, à obtenir dans l'interpréteur Python standard.

Exécution de code externe avec %run

Dès que vous quitterez le nid douillet des petits exemples de découverte, vous allez certainement travailler tour à tour dans IPython pour la partie interactive et dans votre éditeur de texte pour rédiger et retoucher le code source. Au lieu d'ouvrir une nouvelle fenêtre pour l'exécution du code, vous pouvez tout à fait rester dans votre session IPython au moyen de la commande magique `%run`.

Supposons que nous ayons créé un fichier `monscript.py` stocké dans le répertoire courant de la session (par défaut votre répertoire principal) :

```
#-----
# file: monscript.py
def elevcarr(x):
    """ Eleve un nombre au carré"""
    return x ** 2

for N in range(1, 4):
    print(N, "Le carré est ", elevcarr(N))
```

Vous pouvez faire exécuter ce script depuis votre session IPython ainsi :

```
In [6]: %run monscript.py
```

```
1 Le carré est 1
2 Le carré est 4
3 Le carré est 9
```

Une fois que ce script est exécuté, la fonction qu'il a définie reste disponible dans votre session IPython :

```
In [7]: elevcarr(25)
```

```
Out[7]: 625
```

Les modalités d'exécution peuvent être contrôlées finement au moyen de quelques options décrites dans la documentation. Vous y accédez avec la commande magique `%run?`.

Chronométrage de l'exécution avec `%timeit`

Une autre fonction magique très utile est `%timeit`. Elle sert à chronométrer le temps d'exécution d'une instruction sur une ligne. Voici par exemple comment connaître le niveau de performances d'une liste par compréhension :

```
In [16]: %timeit L = [n ** 2 for n in
range(1000)]
```

```
317 µs ± 3.42 µs per loop (mean ± std. dev. of 7
runs, 1000 loops each)
```

L'intérêt de cette commande pour les instructions uniques est qu'elle réalise automatiquement plusieurs exécutions d'affilée afin d'en produire une moyenne. Dans le cas d'un bloc de plusieurs instructions, vous redoublez le préfixe %. Voici le chronométrage des deux lignes d'une boucle

for :

```
In [19]: %timeit
...: L = []
...: for n in range(1000):
...:     L.append(n ** 2)
...
373 µs ± 15.1 µs per loop (mean ± std. dev. of 7
runs, 1000 loops each)
```

Les deux derniers exemples une fois comparés montrent qu'une liste par compréhension est environ 10 % plus rapide que la boucle for équivalente. Nous verrons en fin de chapitre comment utiliser %timeit dans le cadre du profilage du code.

Aide des fonctions magiques ?, %magic et %lsmagic

Les commandes magiques d'IPython disposent, comme les fonctions normales de Python, d'une chaîne de documentation docstring à laquelle vous accédez de la même

façon. Voici par exemple comment accéder à la documentation de `%timeit` :

In [20]: `%timeit?`

Pour une description rapide des commandes magiques disponibles avec des exemples, vous pouvez saisir ceci :

In [21]: `%magic`

Pour la liste des fonctionnalités disponibles, saisissez ceci :

In [22]: `%lsmagic`

Vous pouvez ajouter vos propres définitions de commandes magiques. Nous n'en parlons pas ici, mais servez-vous des références fournies en fin de chapitre.

Historique d'entrées et de sorties

Nous avons parlé plus haut des raccourcis clavier permettant de rappeler les commandes précédentes, avec les touches Haut et Bas et les deux raccourcis Ctrl-p et Ctrl-n. Aussi bien dans l'interpréteur que dans le calepin, IPython permet d'accéder aux résultats d'une commande passée, et de récupérer une chaîne contenant des commandes.

Les objets In et Out d'IPython

Suite aux quelques exemples que nous venons de pratiquer, vous avez certainement compris la structure du dialogue d'IPython qui se base sur une ligne In[0] et une ligne Out[0]. Ce ne sont pas que des éléments de repérage visuel : la valeur numérique désigne une sorte de case mémoire dans laquelle est stocké l'élément d'entrée ou de sortie. Imaginons le début de session suivant (en gras ce qui est saisi) :

```
In [1]: import math
```

```
In [2]: math.sin(2)  
Out[2]: 0.9092974268256817
```

```
In [3]: math.cos(2)
Out[3]: -0.4161468365471424
```

D'abord, nous demandons l'import du paquetage math puis nous faisons calculer le sinus puis le cosinus de 2. Les éléments d'entrée et de sortie sont associés à des labels In et Out. (IPython crée de véritables variables Python qui portent le nom In et Out et qui sont automatiquement mises à jour selon vos actions) :

```
In [4]: print(In)
[ '', 'import math', 'math.sin(2)',
'math.cos(2)', 'print(In)']
```

```
In [5]: Out
Out[5]: {2: 0.9092974268256817, 3:
-0.4161468365471424}
```

L'objet In est une liste de toutes les commandes passées. Le premier élément de la liste est un conteneur, ce qui permet à l'écriture In[1] de faire référence à la première commande :

```
In [6]: print(In[1])
import math
```

En revanche, l'objet Out n'est pas une liste, mais un dictionnaire qui associe le nombre à la donnée en sortie, s'il y en a une :

```
In [7]: print(Out[2])
0.9092974268256817
```

Toutes les opérations ne produisent pas une sortie : import et print n'ont pas de partie Out. Cela peut vous étonner concernant la commande print, mais il faut rappeler qu'il s'agit d'une fonction qui renvoie la pseudo-valeur None (rien). Toutes les commandes qui renvoient cette pseudo-valeur n'impactent pas Out.



(N.d.T.) : Dans la suite du livre, nous aurons souvent besoin de faire débuter la première instruction des exemples non sur la même ligne que la mention In[x] : , mais sur la ligne suivante afin de ne pas perdre de précieuses positions de caractères en largeur. Cela ne suppose pas que vous deviez frapper Entrée pour sauter cette première ligne.

Il est intéressant d'interagir avec les résultats passés. Nous pouvons tout à fait vérifier la somme des carrés des sinus et cosinus précédents en réutilisant les valeurs trouvées plus haut :

```
In [8]: Out[2] ** 2 + Out[3] ** 2
Out[8]: 1.0
```

Comme prévu, le résultat vaut 1.0 (c'est une règle de base de la trigonométrie). Dans cet exemple, nous aurions pu nous passer d'un rappel des calculs antérieurs, mais lorsque ce

calcul est très gourmand en ressources, il est tout à fait intéressant de pouvoir réutiliser la valeur obtenue !

Raccourcis à préfixe souligné et sorties précédentes

L'interpréteur standard de Python ne permet de rappeler que la dernière donnée en sortie au moyen du symbole `_`. Cette convention est reconnue par IPython :

```
In [9]: print(_)
1.0
```

IPython va plus loin en permettant d'utiliser le double et le triple souligné pour accéder à l'avant-dernière et à l'antépénultième donnée de sortie (tout en ignorant les commandes n'ayant pas eu de sortie) :

```
In [10]: print(__)
-0.4161468365471424
In [11]: print(__)
0.9092974268256817
```

IPython ne remonte pas plus loin. D'ailleurs, il devient difficile de lire quatre caractères souligné (tirets bas) accolés les uns aux autres, et il devient plus facile d'indiquer le numéro de case de la sortie.

Il existe encore un raccourci pratique qui permet d'abréger l'écriture Out[X]. Il s'écrit _X, c'est-à-dire le caractère de soulignement suivi du numéro de la donnée :

```
In [12]: Out[2]  
Out[12]: 0.9092974268256817
```

```
In [13]: _2  
Out[13]: 0.9092974268256817
```

Suppression d'une donnée de sortie

Vous aurez parfois envie de supprimer une donnée de sortie pour libérer de la mémoire IPython. Ce sera particulièrement intéressant avec les commandes de tracé décrites dans le [Chapitre 4](#) ; de même, si vous ne voulez pas que le résultat d'une commande soit sauvegardé dans l'historique. La valeur sera éliminée lorsque plus rien n'y fera référence. La solution facile pour supprimer une donnée de sortie de commande consiste à ajouter un signe point-virgule en fin de ligne :

```
In [14]: math.sin(2) + math.cos(2);
```

Dans ce cas, le résultat est calculé de façon silencieuse et rien n'apparaît à l'écran, ni n'est sauvegardé dans le dictionnaire Out :

```
In [15]: 14 in Out  
Out[15]: False
```

Autres commandes magiques

Pour la liste des entrées précédentes, utilisez %history. Voici comment faire afficher les quatre premières entrées de la session :

```
In [16]: %history -n 1-4  
1: import math  
2: math.sin(2)  
3: math.cos(2)  
4: print(In)
```

Pour en savoir plus, vous pouvez saisir %history ? et en découvrir les options. Pour exécuter une partie de l'historique, vous utilisez %rerun. Enfin, %save permet d'enregistrer une sélection de commandes de l'historique dans un fichier.

IPython et les commandes shell

Lorsque vous fonctionnez de manière interactive avec l'interpréteur standard de Python, vous êtes vite gêné par l'obligation de basculer sans cesse entre plusieurs fenêtres pour accéder aux outils et à la ligne de commande. Avec Python, tout est regroupé, car vous pouvez exécuter directement des commandes système depuis le terminal IPython. La magie correspond au signe !. Tout ce qui est saisi après ce symbole sur une ligne n'est pas exécuté par le noyau Python, mais envoyé sur la ligne de commande système.

Nous supposons dans la suite que vous êtes dans un système de style Unix tel que Linux ou macOS. Certains des exemples ne vont pas fonctionner sous Windows, car son interpréteur n'est pas du même type. Ceci dit, Windows a depuis annoncé un interpréteur bash natif qu'il suffit d'installer. Si vous ne connaissez pas les commandes de l'interpréteur Unix, procurez-vous un tutoriel ou un livre à ce sujet. Internet regorge de ressources à ce sujet.

Une brève introduction au shell

Il est hors de question dans ce chapitre de présenter en détail l'utilisation de la ligne de commande du terminal ou

de l'interpréteur shell. Limitons-nous à une brève introduction, afin d'accueillir ceux qui n'ont jamais pratiqué ces outils. Ce que l'on appelle shell, l'interpréteur de commande, est un outil permettant d'interagir avec un ordinateur en dialoguant avec le clavier, donc en mode texte. Depuis le milieu des années 1980, Microsoft aussi bien qu'Apple ont proposé une interface utilisateur graphique, plus exactement, pilotée par des événements que sont notamment les clics de souris. De nos jours, quasiment tout le monde interagit avec son système d'exploitation en ouvrant des menus et en faisant des glisser/déposer.

Mais il existe des systèmes d'exploitation depuis les débuts de l'informatique, et au départ il n'y avait pas d'interface graphique. Pour contrôler une machine, il fallait saisir des commandes, et ces commandes devaient être saisies sur la ligne prévue pour analyser la saisie, la ligne d'invite (en anglais *prompt*). Le dialogue progresse donc ligne par ligne. L'utilisateur saisit une commande puis valide par la touche Entrée, l'ordinateur analyse la saisie et réalise le traitement demandé. Le dialogue en mode texte est celui qui est en vigueur dans les terminaux et interpréteurs qu'utilisent toujours intensivement de nos jours la plupart des datalogues.

Lorsque l'on n'a aucune pratique de ce genre d'outils, on peut se demander si ce n'est pas une perte de temps que

d'essayer de maîtriser cette façon archaïque de dialoguer avec l'ordinateur, alors que l'on peut cliquer et ouvrir des menus. L'utilisateur d'un interpréteur en mode texte y répond par une autre question : pourquoi perdre son temps à repérer visuellement des icônes et ouvrir des menus quand vous pouvez obtenir le même résultat bien plus vite en saisissant une commande ? Ce qui pourrait être considéré comme le nuage d'hermétisme protecteur de quelques programmeurs asociaux devient extrêmement précieux dès qu'il faut faire plus que des tâches élémentaires. L'interpréteur shell offre beaucoup plus de contrôle sur les opérations. Il faut admettre que la courbe d'apprentissage comporte quelques pentes qui peuvent intimider un utilisateur trop peu motivé.

Prenons en guise d'exemple une session de shell Linux/macOS dans laquelle l'utilisateur visite des répertoires et crée puis modifie des fichiers sur son système. Dans l'exemple, la mention osx : ~\$ en début de ligne correspond à l'invite qui est affichée par le système. Tout ce qui suit le signe \$ correspond à ce qui est saisi par l'utilisateur. Ce qui suit le signe dièse en fin de ligne sont des commentaires ajoutés dans ce livre et non saisis dans la session réelle :

```
osx:~ $ echo "hello world"          #
affichage, comme print de Python
hello world
```

```
osx:~ $ pwd # pwd =  
Print Working Directory  
/home/jake # chemin  
d'accès courant  
  
osx:~ $ ls # ls =  
lister contenu dossier  
notebooks projects  
  
osx:~ $ cd projects/ # cd =  
Changer Dossier (répertoire)  
osx:projects $ pwd  
/home/jake/projects  
  
osx:projects $ ls  
datasci_book mpld3 monprojet.txt  
  
osx:projects $ mkdir monprojet # créer  
(MaKe) dossier  
  
osx:projects $ cd monprojet/  
  
osx:monprojet $ mv ../../monprojet.txt ./ # mv =  
déplacer MoVe.  
# On  
déplace monprojet.txt du  
#  
dossier parent (../../) vers le  
#
```

```
dossier courant (./)
osx:monprojet $ ls
monprojet.txt
```

Toute cette séquence d'opérations représente une façon très compacte de réaliser des actions habituelles : navigation dans la structure arborescente, création d'un dossier/répertoire, déplacement d'un fichier, etc. Avec seulement cinq commandes (pwd, ls, cd, mkdir et cp), vous pouvez réaliser la plupart des opérations concernant les fichiers. Mais c'est lorsque vous voulez en faire plus que ces fonctions de base que l'interpréteur shell devient vraiment puissant.

Commandes shell dans IPython

Toutes les commandes disponibles dans votre système peuvent être émises depuis IPython en ajoutant en préfixe le caractère !. Quelques exemples :

```
In [1]: !ls
monprojet.txt
```

```
In [2]: !pwd
/home/jake/projects/monprojet
```

```
In [3]: !echo "Affichage par le shell"
Affichage par le shell
```

Transmettre une valeur de et vers le shell

Non seulement vous pouvez déclencher les commandes depuis IPython, mais vous pouvez interagir avec son espace de noms. Voici par exemple comment stocker la donnée résultant d'une commande de l'interpréteur dans une liste Python avec l'opérateur d'affectation :

```
In [4]: contenu = !ls
```

```
In [5]: print(contenu)
['monprojet.txt']
```

```
In [6]: dossier = !pwd
```

```
In [7]: print(dossier)
[/Users/jakevdp/notebooks/tmp/monprojet']
```

Précisons que le résultat n'est pas envoyé sous forme de liste, mais en tant que type de valeur renvoyée spécifique au shell et défini dans IPython :

```
In [8]: type(dossier)
IPython.utils.text.SList
```

Le résultat ressemble beaucoup à une liste Python, mais il offre quelques caractéristiques, notamment les méthodes

grep et fields ainsi que les propriétés s, n et p qui permettent respectivement de rechercher, de filtrer et d'afficher les résultats. (Servez-vous de l'aide interne d'IPython pour plus de détails.)

Vous pouvez également transmettre une variable de Python vers l'interpréteur en utilisant la syntaxe basée sur les accolades {nom_var} :

```
In [9]: message = "salut de Python"
```

```
In [10]: !echo {message}  
salut de Python
```

Vous citez le nom de la variable entre accolades. Le tout est remplacé par la valeur de cette variable au moment de l'appel à l'interpréteur shell.

Commandes magiques liées au shell

Les commandes shell ne sont pas toutes possibles dans IPython. Si vous faites des essais, vous verrez que vous ne pouvez pas changer de répertoire avec la commande ! cd :

```
In [11]: !pwd  
/home/jake/projects/monprojet
```

```
In [12]: !cd ..
```

```
In [13]: !pwd  
/home/jake/projects/monprojet
```

La raison est simple : dans un calepin, les commandes shell sont exécutées par un sous-processus interpréteur temporaire. Lorsque vous avez vraiment besoin de changer de répertoire courant, il faut utiliser la commande magique %cd :

```
In [14]: %cd ..  
/home/jake/projects
```

Vous pouvez même l'utiliser sans le préfixe :

```
In [15]: cd monprojet  
/home/jake/projects/monprojet
```

En effet, il s'agit d'une fonction automagic ; vous pouvez débrayer ce comportement au moyen de la fonction `%automagic`.

Vous disposez d'une douzaine de fonctions magiques de style shell en plus de `%cd` : `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm` et `%rmdir`. Toutes peuvent être utilisées sans le préfixe `%` si le mode automagic est actif. L'invite de command'IPython peut ainsi être considérée comme une invite d'interpréteur normale du système :

```
In [16]: mkdir tmp
```

```
In [17]: ls  
monprojet.txt tmp/
```

```
In [18]: cp monprojet.txt tmp/
```

```
In [19]: ls tmp  
monprojet.txt
```

```
In [20]: rm -r tmp
```

Cette possibilité d'accéder aux fonctions de l'interpréteur depuis la même fenêtre que celle de la session Python vous

épargne les incessants basculements entre la fenêtre d'interpréteur et celle d'édition de votre code Python.

Erreurs et débogage

Comme en programmation, il faut nécessairement progresser par essais et erreurs en datalogie. IPython offre tous les outils pour alléger ce travail. Découvrons quelques options permettant de contrôler la façon dont Python rend compte des exceptions. Nous verrons ensuite comment aider à la mise au point du code, le débogage.

Contrôle des exceptions avec %xmode

En général, lorsqu'un script Python échoue, il déclenche une exception. L'interpréteur laisse une trace de l'erreur dans le conteneur nommé *traceback* auquel vous avez accès depuis Python. La fonction magique `%xmode` permet de décider de la densité d'informations à afficher lors de la survenue d'une exception. Partons de l'extrait suivant :

```
In[1]: def fonc1(a, b):
         return a / b

def fonc2(x):
    a = x
    b = x - 1
    return fonc1(a, b)
```

```
In[2]: fonc2(1)
```

```
-----  
-----  
ZeroDivisionError Traceback (most recent call  
last)
```

```
<ipython-input-2-b2e110f6fc8f>; in <module>()  
----> 1 fonc2(1)
```

```
<ipython-input-1-d849e34d61fb> in fonc2(x)  
      5     a = x  
      6     b = x - 1  
----> 7     return fonc1(a, b)
```

```
<ipython-input-1-d849e34d61fb> in fonc1(a, b)  
      1 def fonc1(a, b):  
----> 2     return a / b  
      3  
      4 def fonc2(x):  
      5     a = x
```

```
ZeroDivisionError: division by zero
```

C'est l'appel à *fonc2* qui provoque l'erreur. La lecture de la trace d'exécution permet de le confirmer. Par défaut, cette trace indique plusieurs lignes pour donner un contexte autour de l'erreur. La fonction magique %xmode (mode d'exception) permet de choisir la quantité d'informations rapportées.

Un seul paramètre de mode doit être fourni à %xmode avec trois valeurs possibles : Plain, Context et Verbose. La valeur par défaut Context correspond à ce que nous venons de voir. Le mode Plain est plus compact :

```
In[3]: %xmode Plain
```

```
Exception reporting mode: Plain
```

```
In[4]: fonz2(1)
```

```
-----  
-----
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-4-b2e110f6fc8f>", line 1,  
in <module>  
    fonz2(1)
```

```
File "<ipython-input-1-d849e34d61fb>", line 7,  
in fonz2  
    return fonz1(a, b)
```

```
File "<ipython-input-1-d849e34d61fb>", line 2,  
in fonz1  
    return a / b
```

```
ZeroDivisionError: division by zero
```

Enfin, le mode bavard Verbose affiche notamment les paramètres de toutes les fonctions qui ont été appelées :

```
In[5]: %xmode Verbose
```

```
Exception reporting mode: Verbose
```

```
In[6]: fonc2(1)
```

```
-----  
-----
```

```
ZeroDivisionError Traceback (most recent call  
last)
```

```
<ipython-input-6-b2e110f6fc8f> in <module>()  
----> 1   fonc2(1)  
        global fonc2 = <function fonc2 at  
0x103729320>
```

```
<ipython-input-1-d849e34d61fb> in fonc2(x=1)  
      5   a = x  
      6   b = x - 1  
----> 7 return fonc1(a, b)  
        global fonc1 = <function fonc1 at  
0x1037294d0>  
          a = 1  
          b = 0
```

```
<ipython-input-1-d849e34d61fb> in fonc1(a=1,  
b=0)  
      1   def fonc1(a, b):  
----> 2     return a / b  
          a = 1  
          b = 0
```

```
3
4  def fonc2(x):
5    a = x
ZeroDivisionError: division by zero
```

Ce complément d'information étant très utile pour trouver la cause d'une exception, on peut se demander pourquoi on n'utilise pas en permanence le mode Verbose. Le souci est que cette trace d'exécution peut devenir très longue lorsque le code devient complexe. Dans certains cas, la compacité du mode par défaut est plus intéressante.

Débogage : si la trace d'exécution ne suffit pas

L'outil de débogage interactif de Python est en standard pdb. Il permet bien sûr de progresser dans l'exécution ligne par ligne pour trouver les erreurs les plus coriaces. La version proposée par IPython est ipdb ; elle apporte quelques améliorations.

Le lancement de l'un ou l'autre débogueur ne sera pas détaillé ici. Voyez la documentation en ligne.

Dans IPython, l'interface de débogage la plus pratique est sans doute celle liée à la commande magique %debug. Si vous le lancez juste après avoir déclenché une exception, vous verrez s'ouvrir une invite de débogage interactive à

l'endroit même de l'exception. L'invite de ipdb permet alors de parcourir l'état actuel de la pile d'appels, d'explorer les variables disponibles et même de lancer des commandes Python.

Revenons sur la dernière exception déclenchée et réalisons quelques tâches élémentaires. Nous affichons les valeurs de *a* et de *b*, puis sortons de la session de débogage par la commande quit :

```
In[7]: %debug
> <ipython-input-1-d849e34d61fb>(2)fonc1()
    1 def fonc1(a, b):
----> 2     return a / b
        3

ipdb> print(a)
1

ipdb> print(b)
0

ipdb> quit
```

Le débogueur interactif permet bien d'autres opérations. Il est possible de monter et descendre dans la pile d'appels, tout en explorant les valeurs des variables disponibles à chaque étape :

```
In[8]: %debug
> <ipython-input-1-d849e34d61fb>(2)fonc1()
    1 def fonc1(a, b):
----> 2     return a / b
3
ipdb> up
> <ipython-input-1-d849e34d61fb>(7)fonc2()
    5     a = x
    6     b = x - 1
----> 7     return fonc1(a, b)

ipdb> print(x)
1

ipdb> up
> <ipython-input-6-b2e110f6fc8f>(1)<module>()
----> 1 fonc2(1)

ipdb> down
> <ipython-input-1-d849e34d61fb>(7)fonc2()
    5     a = x
    6     b = x - 1
----> 7     return fonc1(a, b)

ipdb> quit
```

Il devient ainsi possible de trouver non seulement la cause de l'erreur, mais aussi l'appel de fonction en erreur.

Pour obtenir le démarrage automatique du débogueur en cas d'exception, vous activez ce mode avec la fonction magique %pib :

```
In[9]: %xmode Plain  
%pdb on  
fonc2(1)
```

```
Exception reporting mode: Plain  
Automatic pdb calling has been turned ON
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-9-569a67d2d312>", line 3,  
in <module>  
    fonc2(1)
```

```
File "<ipython-input-1-d849e34d61fb>", line 7,  
in fonc2  
    return fonc1(a, b)
```

```
File "<ipython-input-1-d849e34d61fb>", line 2,  
in fonc1  
    return a / b
```

```
ZeroDivisionError: division by zero
```

```
> <ipython-input-1-d849e34d61fb>(2)fonc1()  
    1 def fonc1(a, b):  
----> 2     return a / b
```

3

```
ipdb> print(b)  
0
```

```
ipdb> quit
```

Vous pouvez enfin demander l'exécution d'un script à l'entrée en mode interactif avec %run -d puis utilisez la commande next pour progresser interactivement le long des lignes de code.

Sélection de commandes de débogage

Voici une sélection des commandes de débogage les plus utiles (cette liste est loin d'être exhaustive) :

Commande	Description
list	Affiche la position actuelle dans le fichier.
h(elp)	Affiche une liste de commandes ou de l'aide sur une commande.
q(uit)	Quitte le débogueur et le programme.
c(ontinue)	Quitte le débogueur sans sortir du programme.
n(ext)	Va à la prochaine étape.
<Enter>	Répète la commande précédente.
print	Affiche les valeurs des variables.
s(tep)	Descend exécuter les instructions d'une fonction.
return	Abrège l'exécution de la fonction en cours.

Vous trouverez d'autres informations avec la commande help du débogueur. Voyez aussi la documentation en ligne de ipdb (<https://github.com/gotcha/ipdb>).

Profilage et chronométrage d'exécution

Lorsque vous développez du code et mettez en place des processus de traitement de données, il vous arrivera souvent de devoir choisir entre plusieurs approches de réalisation. Ceci dit, s'engager sur les détails lors des premières étapes peut s'avérer contre-productif. Comme l'avait stipulé le grand Donald Knuth : « Nous devons oublier les petites optimisations, quasiment 97 % du temps. Toute optimisation prématuée peut se transformer en fléau. ».

En revanche, une fois que votre code est opérationnel, vous avez tout intérêt à le passer en revue pour y chercher des opportunités d'optimisation. Vous voudrez par exemple chronométrier la durée d'exécution d'une commande ou bien étudier le comportement d'un processus complexe pour repérer l'endroit qui pourrait constituer le goulet d'étranglement. Avec IPython, vous disposez d'une panoplie d'outils pour le chronométrage et le profilage du code. Découvrons quelques commandes magiques de ce domaine :

`%time`

Chromomètre l'exécution d'une instruction.

```
%timeit
```

Chronomètre plusieurs exécutions de la même instruction pour augmenter la précision.

```
%prun
```

Exécute le code dans le profileur.

```
%lprun
```

Exécute le code dans le profileur ligne par ligne.

```
%memit
```

Mesure l'occupation mémoire d'une seule instruction.

```
%mprun
```

Exécute le code avec le profileur mémoire ligne par ligne.

Notez que les quatre dernières commandes ne sont pas fournies au départ avec IPython, mais il suffit d'installer les deux extensions `line_profiler` et `memory_profiler`, comme nous le verrons plus loin.

Chronométrage d'un bloc de code : %time et %timeit

Nous avons déjà vu les commandes magiques %timeit et %%timeit dans le début du chapitre. La variante %%timeit force l'exécution d'un bloc de code de façon répétée :

```
In[1]: %timeit sum(range(100))
1.39 µs ± 36.7 ns per loop (mean ± std. dev. of
7 runs, 1000000 loops each)
```

La commande tient compte de la durée d'exécution de chaque tour de boucle. Lorsque l'opération est très rapide, elle réalise d'office un grand nombre de répétitions. Pour un traitement plus lourd, elle en réalise moins :

```
In[2]: %%timeit
total = 0
for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
404 ms ± 2.58 ms per loop (mean ± std. dev. of 7
runs, 1 loop each)
```

Dans certains cas, il n'est pas idéal de répéter la même opération. Par exemple dans une opération de tri, dès le

premier tour, la liste est triée et le chronométrage ne donne pas une image fidèle :

```
In[3]: import random  
        L = [random.random() for i in  
range(100000)]  
        %timeit L.sort()
```

```
1.74 ms ± 151 µs per loop (mean ± std. dev. of 7  
runs, 1000 loops each)
```

Voilà pourquoi il est parfois préférable d'utiliser la fonction %time. Vous la privilégierez également pour les commandes de longue durée pour lesquelles de petits délais ajoutés par le système n'ont que peu d'impact sur les résultats. Comparons la durée de tri d'une liste non triée à celle d'une liste déjà triée :

```
In[4]: import random  
        L = [random.random() for i in  
range(100000)]  
        print("Tri d'une liste non triée :")  
        %time L.sort()
```

```
Tri d'une liste non triée  
CPU times: user 40.6 ms, sys: 896 µs, total:  
41.5 ms:  
Wall time: 29 ms
```

```
In[5]: print("Tri d'une liste déjà triée :")
        %time L.sort()

Tri d'une liste déjà triée :
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19
ms
Wall time: 8.24 ms
```

Bien sûr, la liste déjà triée est exécutée bien plus vite, mais comparez surtout la différence de durée entre %time et %timeit, même pour la liste déjà triée. Quelle est la cause de cette différence ? %timeit réalise des opérations de précaution pour éviter que des appels système viennent perturber le chronométrage. La commande suspend en particulier toute tentative de lancer le *recycleur mémoire* (ramasse-miettes) qui sert à supprimer les objets Python qui ne sont plus référencés. C'est pourquoi %timeit donne des résultats souvent beaucoup plus rapides que %time.

Pour les deux commandes, le fait de redoubler le préfixe % permet de chronométrer des scripts multilignes :

```
In[6]: %%time
        total = 0
        for i in range(1000):
            for j in range(1000):
                total += i * (-1) ** j
```

```
CPU times: user 504 ms, sys: 979 µs, total: 505
```

```
ms
Wall time: 505 ms
```

Pour d'autres détails, voyez l'aide d'IPython, par exemple avec `%time ?`.

Profilage d'un script entier avec `%prun`

Il est souvent appréciable de pouvoir chronométrer non pas une instruction isolée, mais tout un groupe. Python est doté d'un profileur (décrit dans la documentation) qu'IPython permet d'utiliser de façon plus efficace en utilisant la fonction magique `%prun`.

Commençons par définir une fonction qui effectue un calcul quelconque :

```
In[7]: def sommer_listes(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in
range(N)]
        total += sum(L)
    return total
```

Appelons cette fonction pour accéder aux résultats du profilage :

```
In[8]: %prun sommer_listes(1000000)
```

Si vous travaillez avec le calepin, le résultat est affiché dans la fenêtre du bas (*pager*). Voici son aspect :

```
14 function calls in 1.273 seconds
```

```
Ordered by: internal time
```

filename:lineno(function)	ncalls	tottime	percall	cumtime	percall
<ipython-input-7-cc7ed5d56840>:4(<listcomp>)	5	1.072	0.214	1.072	0.214
{built-in method builtins.sum}	5	0.132	0.026	0.132	0.026
<ipython-input-7-cc7ed5d56840>:1(sommer_listes)	1	0.052	0.052	1.257	1.257
<string>:1(<module>)	1	0.016	0.016	1.273	1.273
{built-in method builtins.exec}	1	0.000	0.000	1.273	1.273
{method 'disable' of '_lsprof.Profiler' ob}	1	0.000	0.000	0.000	0.000

Nous obtenons un tableau qui montre le temps d'exécution total de chaque appel de fonction. Dans cet exemple, la majeure partie du temps d'exécution concerne la ligne de la liste par compréhension, dans sommer_listes. Nous

obtenons ainsi un index pour orienter nos efforts et éventuellement améliorer les performances de l'algorithme.

Pour tout détail, utilisez l'aide d'IPython avec `%prun?`.

Profilage ligne par ligne avec `%lprun`

La commande `%prun` permet de travailler fonction par fonction, mais il est parfois plus pratique de profiler ligne par ligne. Cette possibilité n'est pas fournie en standard dans Python, ni dans IPython, mais vous pouvez installer le paquetage nommé `line_profiler`. Servez-vous de l'outil de gestion de paquetages pip pour ce faire :

```
$ pip install line_profiler
```

Il ne reste plus qu'à charger l'extension pour IPython :

```
In[9]: %load_ext line_profiler
```

La commande `%lprun` va profiler chaque ligne de chaque fonction. Nous devons donc lui dire quelle fonction nous intéresse :

```
In[10]: %lprun -f sommer_listes  
sommer_listes(5000)
```

Le résultat ressemble à ceci :

```
Timer unit: 1e-07 s
```

```
Total time: 0.0100692 s
File: <ipython-input-7-cc7ed5d56840>
Function: sommer_listes at line 1
```

Line #	Hits	Time	Per Hit	% Time
Line Contents				
=====	=====	=====	=====	=====
=====	=====	=====	=====	=====
1				
def sommer_listes(N):				
2	1	22.0	22.0	0.0
total = 0				
3	6	75.0	12.5	0.1
for i in range(5):				
4	5	97691.0	19538.2	97.0
L = [j ^ (j >> i) for j in range(N)]				
5	5	2899.0	579.8	2.9
total += sum(L)				
6	1	5.0	5.0	0.0
return total				

La première ligne rappelle l'unité temporelle en vigueur qui est ici la microseconde. On voit immédiatement où le programme passe l'essentiel de son temps. Cette

information va permettre de retoucher le script et d'espérer de meilleures performances.

Profilage de l'occupation mémoire avec %memit et %mprun

L'autre domaine d'informations que le profilage permet d'exploiter concerne l'occupation mémoire. Il faut installer une autre extension dans IPython, memory_profiler. Nous nous servons de l'outil pip :

```
$ pip install memory_profiler
```

Nous n'oublions pas de charger l'extension dans IPython :

```
In[11]: %load_ext memory_profiler
```

Cette extension offre deux commandes magiques : %memit donne l'équivalent de %timeit pour la mesure de l'espace mémoire et %mprun qui est l'équivalent pour la mémoire de %lprun. La première est très simple à utiliser :

```
In[12]: %memit sum_of_lists(1000000)
```

```
peak memory: 122.00 MiB, increment: 69.16 MiB
```

Nous voyons que la fonction occupe environ 120 Mo de mémoire.

L'autre commande, `%mprun`, permet de connaître l'occupation mémoire pour chaque ligne de code. Elle n'est hélas utilisable qu'avec des fonctions définies dans d'autres modules et pas directement celles du calepin. Nous allons donc créer avec la commande `%file` un module complémentaire portant le nom `mprun_demo.py`. Il contiendra notre fonction `sommer_listes()`. Nous ajouterons une instruction pour rendre les résultats plus lisibles :

```
In[13]: %%file mprun_demo.py
def sommer_listes(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in
range(N)]
        total += sum(L)
    del L           # Détruire la
référence à L
    return total
```

Nous importons bien sûr la nouvelle version de la fonction puis nous lançons le profileur mémoire :

```
In[14]: from mprun_demo import sommer_listes
        %mprun -f sommer_listes
sommer_listes(1000000)
```

Le résultat nous informe au sujet des besoins mémoire de la fonction :

Filename: ./mprun_demo.py

Line #	Mem usage	Increment	Line Contents
4	71.9 MiB	0.0 MiB	L = [j ^
(j >> i) for j in range(N)]			

Filename: ./mprun_demo.py

Line #	Mem usage	Increment	Line Contents
1	39.0 MiB	0.0 MiB	def
sommer_listes(N):			
2	39.0 MiB	0.0 MiB	total = 0
3	46.5 MiB	7.5 MiB	for i in
range(5):			
4	71.9 MiB	25.4 MiB	L = [j ^
(j >> i) for j in range(N)]			
5	71.9 MiB	0.0 MiB	total +=
sum(L)			
6	46.5 MiB	-25.4 MiB	del L #
Détruire ref a L			
7	39.1 MiB	-7.4 MiB	return total

Vous voyez que la colonne Increment permet de connaître l'occupation mémoire de chaque ligne. Vous voyez que la création de la liste réclame 25 Mo environ de mémoire, restitués quand nous supprimons la référence. Ces valeurs

ne tiennent pas compte de l'occupation mémoire de l'interpréteur IPython lui-même.

Pour tout détail, vous utilisez %memit? et %mprun?.

Autres ressources IPython

Après cette rapide présentation des principales fonctions d'IPython utiles en datalogie, n'hésitez pas à chercher sur le Web et chez votre librairie d'autres ressources au sujet de cet outil. Voici quelques pistes.

Ressources Web

Le site d'IPython

(<http://ipython.org>)

C'est ici que vous trouverez la documentation, les exemples, les tutoriels et d'autres ressources.

Le site de nbviewer

(<http://nbviewer.ipython.org/>)

Sur ce site, vous trouverez des calepins IPython déjà exécutés et sauvegardés tels quels. La page d'accueil met en vitrine quelques calepins, ce qui permet de se faire une bonne idée de ce que d'autres réussissent à faire avec IPython.

Une galerie de calepins IPython

(<http://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks/>)

Cette autre liste de calepins utilisable avec nbviewer montre la vaste étendue des analyses numériques possibles avec IPython. Il s'y trouve aussi bien de brefs exemples que des cours et présentations complètes au format calepin.

Tutoriels vidéo

Parmi les nombreux tutoriels vidéo concernant IPython, citons ceux produits lors des conférences PyCon, SciPy et PyData, notamment ceux de Fernando Perez et Brian Granger, deux des fondateurs et principaux mainteneurs d'IPython et de Jupyter.

Livres imprimés

Analyse de données avec Python (First Interactive, 2021)

Dans ce livre, Wes McKinney, créateur de la librairie pandas, a lui aussi prévu un chapitre sur l'outil IPython. Une partie de la description recoupe celle que nous venons de faire, mais le reste du livre regorge de conseils de premier choix.

Learning IPython for Interactive Computing and Data Visualization et IPython Interactive Computing and Visualization Cookbook

L'auteur Cyrille Rossant a écrit ces deux livres (Packt Publishing, 2015 et 2018) dont vous pouvez étudier les exemples même si vous ne parlez pas anglais :

<http://bit.ly/2eLCBB7>

<http://bit.ly/2fCEtNE>

Pour conclure, rappelons qu'il ne faut pas hésiter à utiliser la fonction d'accès à l'aide interne d'IPython fondée sur le signe ? en suffixe.

CHAPITRE 2

Introduction à NumPy

Ce chapitre ainsi que le suivant proposent une découverte des techniques qui permettent, en langage Python, de charger des données en mémoire, de les manipuler et de les sauvegarder. Le sujet est vaste, car les sources et formats de ces données sont très variés : documents textuels, images, clips audio, mesures numériques, et bien d'autres. Nous réduirons cette hétérogénéité apparente en considérant toutes les données comme des tableaux de nombres.

Une photographie numérique peut par exemple être vue comme un tableau de valeurs numériques en deux dimensions, représentant la luminosité de chaque point de la surface (chaque pixel). Un clip audio peut s'entendre comme un tableau d'intensités au cours du temps, à une dimension donc. Un fichier texte peut être traité sous forme numérique par exemple pour connaître la fréquence de certains mots ou couples de mots. Quelles que soient les données d'entrée, pour pouvoir les analyser, il faut d'abord les faire entrer dans des tableaux de valeurs numériques. (Nous verrons quelques exemples particuliers de ce

processus dans la section sur l'ingénierie des caractéristiques du [Chapitre 5.](#))

Il est donc crucial en datalogie de savoir stocker et manipuler efficacement les tableaux de valeurs numériques. Nous allons découvrir les outils spécialisés disponibles dans Python pour ce genre de traitement : nous verrons d'abord le paquetage NumPy, puis dans le [Chapitre 3](#) le paquetage Pandas.

NumPy (abréviation de *Numerical Python*) va être vu sous toutes les coutures. Il propose une interface pour stocker et traiter les contenus de tampons de données denses. Les tableaux NumPy sont proches du type liste standard de Python, mais NumPy est beaucoup plus efficace pour le stockage et la manipulation des données, d'autant plus quand les tableaux deviennent volumineux. Ces tableaux NumPy incarnent la cheville ouvrière de tout l'écosystème de datalogie Python. Le temps que vous consacrerez à maîtriser NumPy vous sera rendu, quel que soit le secteur de datalogie qui vous intéresse.

Si vous avez installé tout l'atelier Anaconda comme conseillé dans la préface, NumPy est déjà installé et prêt à fonctionner. Si vous êtes plutôt du genre à vouloir tout faire manuellement, rendez-vous sur le site Web de NumPy (<http://www.numpy.org/>) et suivez les instructions fournies pour l'installer. Vous pouvez ensuite demander

l'importation de NumPy et en profiter pour vérifier la version :

```
In[1]: import numpy  
        numpy.__version__  
Out[1]: '1.11.1'
```

Pour réaliser les exemples du chapitre, je conseille de disposer au moins de la version 1.8. Une convention dans le milieu des utilisateurs de NumPy consiste à importer NumPy en lui donnant l'alias abrégé np :

```
In[2]: import numpy as np
```

C'est de cette façon que nous allons systématiquement importer NumPy dans ce livre.

Rappel pour la documentation intégrée

Tout en lisant ce chapitre, n'oubliez pas qu'IPython permet de connaître rapidement le contenu d'un paquetage au moyen de sa fonction basée sur la touche Tabulation (symbolisée par <TAB> dans les exemples). Vous avez également accès à la documentation des différentes fonctions au moyen du caractère ?. Revoyez

la section qui décrit la documentation dans la préface pour d'autres détails.

Voici par exemple comment afficher tous les noms connus dans l'espace de noms de NumPy :

In [3]: `np.<TAB>`

Pour accéder à la documentation interne de NumPy, saisissez ceci :

In [4]: `np?`

Vous trouverez bien sûr une documentation plus complète ainsi que des exemples et des ressources sur le site Web de NumPy déjà cité.

2.1 : Les types de données Python

Pour qu'un traitement scientifique des données soit pertinent, il faut avoir compris comment ces données sont stockées et manipulées. Dans cette section, nous allons voir comment les tableaux de données sont gérés dans le langage Python, puis nous allons comparer à la façon dont cela est réalisé avec NumPy. Il est essentiel de comprendre ces différences pour bien apprécier la suite de ce livre.

Souvent, les gens adoptent le langage Python parce qu'il est simple à utiliser, et notamment parce que ses types des données sont dynamiques, déduits du contexte d'utilisation par le langage. Dans un langage compilé tel que le C ou le Java, le programmeur doit indiquer explicitement le type de chacune de ses variables. Dans Python, cela n'est plus nécessaire. Voici par exemple un petit bloc de code en langage C :

```
// Code C
int resultat = 0;

for(int i=0; i<100; i++){
    resultat += i;
}
```

Voici l'équivalent en Python :

```
# Code Python
resultat = 0

for i in range(100):
    resultat += i
```

Vous constatez qu'en langage C, nous indiquons le type de donnée avant le nom des deux variables, alors qu'en Python, nous citons directement leur nom sans type en préfixe, ce qui permet de stocker tour à tour des données de types différents dans la même variable :

```
# Python
x = 4
x = "quatre"
```

Ci-dessus, le contenu de `x` était au départ un entier ; la variable est ensuite recyclée en tant que conteneur pour une chaîne de caractères. En langage C, sous l'effet des options de compilation strictes, cela entraîne une erreur de compilation ou une conséquence plus gênante même :

```
// Code C
int x = 4;
x = "quatre"; // ECHEC DE COMPILEATION
```

C'est cette souplesse, cette versatilité, qui rend les langages à typage dynamique tels que Python séduisants, car plus simples d'emploi. Mais pour pouvoir faire de l'analyse de données efficace et pertinente en Python, il faut prendre la peine de comprendre comment fonctionne ce typage flou. Dans un langage à typage strict statique, une variable est un nom symbolique associé à une adresse mémoire, adresse à laquelle se trouve une valeur. Dans Python, les variables sont bien plus complexes. Elles englobent des informations décrivant le type. Voyons cela plus en détail.

Un entier Python est plus qu'un entier

Le langage Python standard est lui-même écrit en langage C. Chaque objet de Python est en réalité une structure du langage C rhabillée, qui contient en plus de la valeur, un certain nombre d'autres informations. Lorsque nous utilisons un entier Python, comme dans `x = 10000`, le nom `x` ne correspond pas qu'à une valeur numérique entière (*integer*). Il s'agit d'un pointeur vers une structure de données C qui contient plusieurs valeurs. En allant voir dans le code source de Python 3, nous constatons que la définition du type entier (`long`) offre l'aspect suivant, une fois les macros C interprétées :

```
struct _longobject {  
    long          ob_refcnt;  
    PyTypeObject *ob_type;  
    size_t        ob_size;  
    long          ob_digit[1];  
};
```

Un entier de Python 3 contient en fait quatre variables, quatre éléments :

- `ob_refcnt` est le compteur du nombre de références actives, ce qui permet à Python de gérer la libération et la réservation d'espace mémoire en coulisses.
- `ob_type` encode le type de la variable.
- `ob_size` indique la taille des données qui suivent.
- `ob_digit` contient la valeur numérique entière de la variable.

Le résultat est que l'opération de stockage d'une valeur entière en Python requiert plusieurs opérations en plus de ce que réclame le langage C, comme le montre la [Figure 2.1](#).

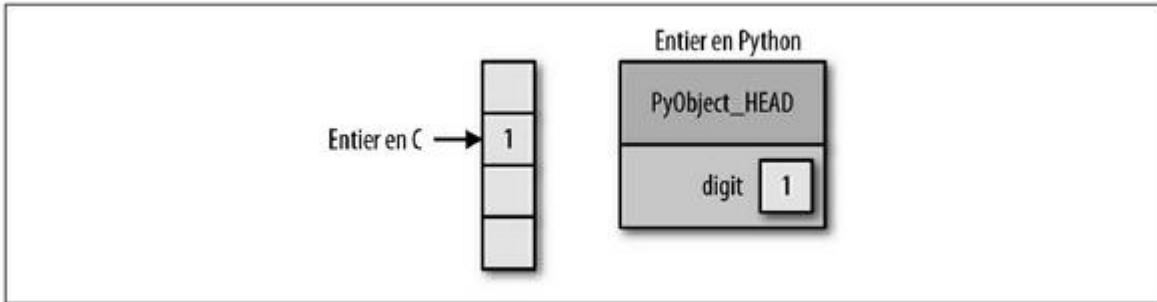


Figure 2.1 : Différence de stockage d'un entier en C et en Python.

Dans la figure, l'en-tête de la variable, PyObject_HEAD correspond à la partie de la structure des données qui réunit le compteur de références, le code du type et les autres éléments de gestion.

Vous constatez que la variable en C désigne directement un endroit en mémoire alors qu'en Python, le nom de variable correspond à un pointeur qui marque le début de la zone dans laquelle sont stockés les différents éléments de la variable Python. C'est grâce à cette structure que les variables Python peuvent changer de type de contenu de façon dynamique. Vous devinez que cette complexité des variables Python se paye, et ce sera d'autant plus sensible lorsqu'il s'agira de manipuler un grand nombre de ce genre d'objet dans les structures.

Une liste Python n'est pas qu'une simple liste

Voyons maintenant ce qui se passe lors de l'utilisation d'une structure de données Python contenant un nombre d'objets Python important. Le conteneur standard à plusieurs éléments modifiables de Python correspond au type list. Voici comment créer une liste de valeurs entières :

```
In[1]: L = list(range(10))  
L  
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In[2]: type(L[0])  
Out[2]: int
```

Voici comment créer une liste de chaînes de caractères :

```
In[3]: L2 = [str(c) for c in L]  
L2  
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7',  
'8', '9']
```



```
In[4]: type(L2[0])  
Out[4]: str
```

Le typage dynamique de Python nous autorise à créer des listes hétérogènes :

```
In[5]: L3 = [True, "2", 3.0, 4]
          [type(item) for item in L3]
Out[5]: [bool, str, float, int]
```

Cette souplesse n'est évidemment pas gratuite : chaque élément de la liste doit être doté de son information de type, de son compteur de références et des autres informations de gestion. Chaque élément est un vrai objet Python complet. Lorsque toutes les variables de la liste sont du même type, quasiment toutes les informations sont redondantes. Il est dans ce cas bien plus efficace de les stocker dans un tableau de type fixe. La [Figure 2.2](#) montre la différence entre un type liste dynamique et un type fixe dans le style NumPy.

Concrètement, le tableau correspond à un seul pointeur vers le début d'un bloc de données. La liste Python contient un pointeur vers un bloc de pointeurs, chacun de ces derniers pointant vers un objet Python complet, comme par exemple l'entier que nous venons de voir. La liste est souple : vous pouvez stocker n'importe quel type de données dans chacun des éléments de la liste. Dans un tableau à type unique de style NumPy, vous perdez cette souplesse, mais l'objet est beaucoup plus efficace en termes de stockage et de manipulation.

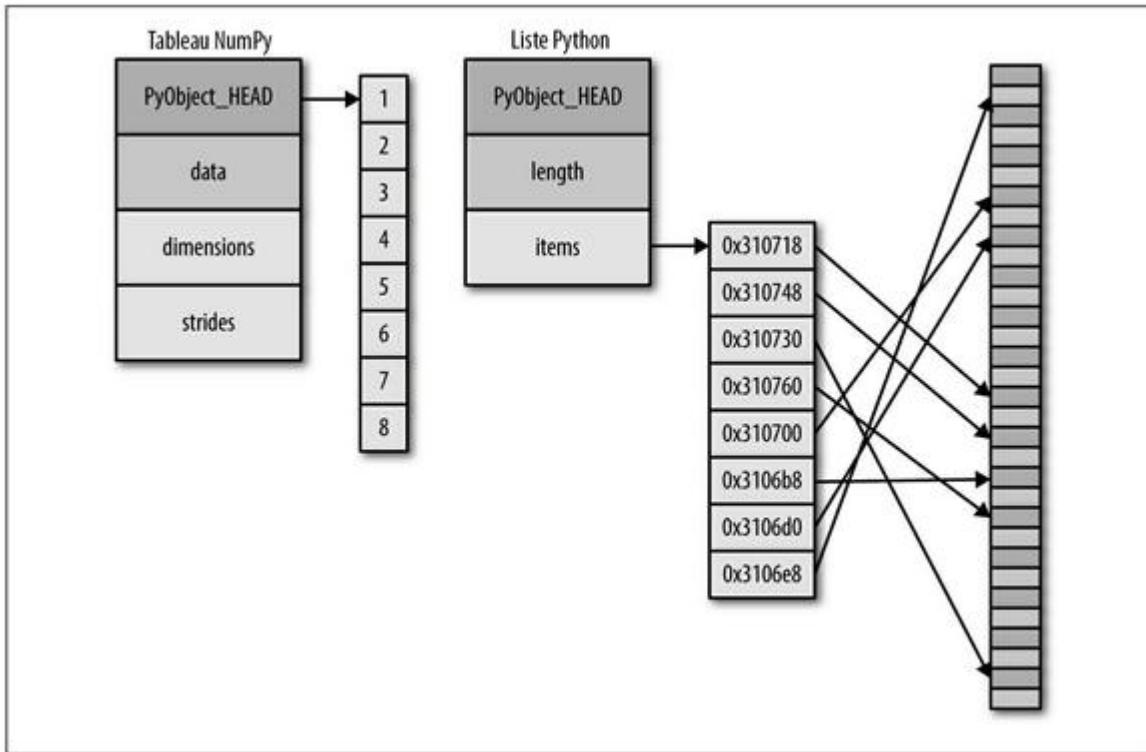


Figure 2.2. : Différences entre une liste C et une liste Python.

Tableaux Python à type fixe ou unique

Python propose heureusement plusieurs options lorsqu'il s'agit de stocker les données de façon efficace avec le même type, comme dans un tampon. Vous disposez tout d'abord depuis Python 3.3 du module standard `array` qui permet de créer un tableau de type unique :

```
In[6]: import array
L = list(range(10))
A = array.array('i', L)
```

A

```
Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Dans l'exemple, 'i' est un code de type qui correspond au type entier, *Integer*.

Le paquetage NumPy offre un objet tableau encore plus efficace, ndarray. Les tableaux (*array*) de Python permettent de stocker efficacement des données tabulaires, mais NumPy permet également des opérations efficaces sur les données. Nous verrons les différentes opérations possibles dans la suite du chapitre. Commençons par créer un tableau NumPy.

Nous demandons l'importation du module NumPy standard avec le nom d'alias np :

```
In[7]: import numpy as np
```

Création d'un tableau depuis une liste Python

Avec la fonction np.array, nous pouvons créer un tableau à partir d'une liste Python littérale :

```
In[8]: # Tableau d'entiers  
        np.array([1, 4, 2, 5, 3])  
Out[8]: array([1, 4, 2, 5, 3])
```

Rappelons que toutes les données doivent être du même type dans le tableau NumPy. Lorsqu'un autre type est détecté, NumPy tente d'effectuer son transtypage vers le type plus large. Dans l'exemple suivant, une valeur est de type flottant ; les trois autres valeurs sont de ce fait converties vers le même type flottant :

```
In[9]: np.array([3.14, 4, 2, 3])
Out[9]: array([ 3.14, 4. , 2. , 3. ])
```

Nous pouvons spécifier le type désiré pour le tableau résultant au moyen du mot-clé `dtype` :

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')
Out[10]: array([ 1., 2., 3., 4.], dtype=float32)
```

À la différence d'une liste Python, le tableau NumPy peut être déclaré avec plusieurs dimensions. Voici comment initialiser un tel tableau en spécifiant une liste de listes :

```
In[11]: # Listes imbriquées pour tableau
multidimensionnel
np.array([range(i, i + 3) for i in [2, 4,
6]])
Out[11]: array([[2, 3, 4],
```

La liste la plus interne est distribuée dans le sens des lignes du tableau résultant.

Création d'un tableau avec valeurs initiales

Il est en général plus efficace de créer vos tableaux au moyen d'une routine de remplissage disponible dans NumPy, notamment pour les grands tableaux. Voici plusieurs exemples :

```
In[2]: # Tableau de 10 entiers valant 0  
        np.zeros(10, dtype=int)
```

```
Out[2]: array([0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In[3]: # Tableau de 3 x 5 flottants avec la  
        valeur 1  
        np.ones((3, 5), dtype=float)
```

```
Out[3]: array([[ 1.,  1.,  1.,  1.,  1.],  
               [ 1.,  1.,  1.,  1.,  1.],  
               [ 1.,  1.,  1.,  1.,  1.]])
```

```
In[4]: # Tableau de 3 x 5 contenant 3.14  
        np.full((3, 5), 3.14)
```

```
Out[4]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],  
               [ 3.14,  3.14,  3.14,  3.14,  3.14],  
               [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```
In[5]: # Tableau avec séquence linéaire  
        # commençant à 0, jusqu'à 20, par pas de 2  
        # (équivaut à la fonction interne range())  
)
```

```
np.arange(0, 20, 2)
Out[5]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16,
18])

In[6]: # Tableau de cinq valeurs réparties entre
0 et 1
        np (0, 1, 5)
Out[6]: array([ 0. , 0.25, 0.5 , 0.75, 1. ])

In[7]: # Tableau de 3x3 aléatoire distribué de
façon
        # uniforme entre 0 et 1
        np.random.random((3, 3))
Out[7]: array([[ 0.99844933, 0.52183819,
0.22421193],
               [ 0.08007488, 0.45429293,
0.20941444],
               [ 0.14360941, 0.96910973, 0.946117
]])

In[8]: # Tableau de 3x3 aléatoires à
distribution normale
        # avec médiane 0 et écart-type 1
        np.random.normal(0, 1, (3, 3))
Out[8]: array([[ 1.51772646, 0.39614948,
-0.10634696],
               [ 0.25671348, 0.00732722,
0.37783601],
               [ 0.68446945, 0.15926039,
-0.70744073]])
```

```
In[9]: # Tableau de 3x3 entiers aléatoires entre  
0 et 10  
      np.random.randint(0, 10, (3, 3))  
Out[9]: array([[2, 3, 4],  
              [5, 7, 8],  
              [0, 5, 0]])  
  
In[10]: # Matrice d'identité de 3x3  
      np.eye(3)  
Out[10]: array([[ 1.,  0.,  0.],  
                 [ 0.,  1.,  0.],  
                 [ 0.,  0.,  1.]])  
  
In[11]: # Tableau non initialisé de trois  
entiers. Les valeurs  
        # seront celles présentes en mémoire à ce  
moment.  
      np.empty(3)  
Out[11]: array([ 1.,  1.,  1.])
```

Types de données NumPy standard

Chaque tableau NumPy ne peut contenir qu'un type de données. Il est essentiel de connaître ces différents types avec leurs contraintes. Du fait que NumPy est écrit en langage C, les types sembleront familiers à ceux qui connaissent cet autre langage, ou bien le Fortran ou un autre langage procédural.

Le [Tableau 2.1](#) présente les types de données standard de NumPy. Vous pouvez indiquer le type lorsque vous créez un tableau en indiquant le mot correspondant :

```
np.zeros(10, dtype='int16')
```

Vous pouvez également indiquer l'objet NumPy associé :

```
np.zeros(10, dtype=np.int16)
```

[Tableau 2.1](#) : Types de données NumPy standard.

Type	Description
bool_	Booléen (True ou False), un seul bit mais occupe un octet.
int_	Type entier par défaut (comme le long du C; int64 ou int32).
intc	Comme le int du C (int32 ou int64).
intp	Entier pour les index (comme ssize_t du C ; int32 ou int64).
int8	Octet, byte (-128 à 127).
int16	Entier sur 2 octets (-32768 à 32767).
int32	Entier sur 4 octets (-2147483648 à 2147483647).
int64	Entier sur 8 octets (-9223372036854775808 à 9223372036854775807).
uint8	Entier non signé sur 1 octet (0 à 255).
uint16	Entier non signé sur 2 octets (0 à 65535).
uint32	Entier non signé sur 4 octets (0 à 4294967295).

uint64	Entier non signé sur 8 octets (0 à 18446744073709551615).
float_	Abréviation de float64 .
float16	Flottant demi-précision: bit de signe, exposant sur 5 bits, mantisse sur 10 bits.
float32	Flottant simple précision: bit de signe, exposant sur 8 bits, mantisse sur 23 bits.
float64	Flottant double précision: bit de signe, exposant sur 11 bits, mantisse sur 52 bits.
complex_	Abréviation de complex128 .
complex64	Nombre complexe, deux flottants sur 32 bits
complex128	Nombre complexe, deux flottants sur 64 bits.

D'autres options concernant le type sont disponibles, et notamment le choix du *boutisme*, entre *big* et *little endian* (stockage du bit de poids fort en premier ou en dernier dans chaque octet). Voyez la documentation de NumPy pour les détails (<http://numpy.org/>). NumPy permet également d'utiliser des types de données composites ; nous les verrons dans la dernière section de ce chapitre.

2.2 : Fondamentaux des tableaux NumPy

Manipuler de gros volumes de données en Python revient en pratique souvent à manipuler des tableaux NumPy. Même un outil plus récemment apparu tel que Pandas (décrit dans le [Chapitre 3](#)) se fonde sur les tableaux NumPy. Nous allons découvrir plusieurs exemples d'utilisation de ces tableaux. Nous verrons comment accéder à des données, comment créer des sous-tableaux, comment distribuer un tableau, le reformer et en fusionner plusieurs. Le passage en revue de ces différentes opérations pourrait sembler un peu trop scolaire, mais ces opérations sont les blocs de construction de la plupart des exemples qui suivent dans ce livre. Vous devez donc apprendre à les maîtriser.

Voici les différents types de manipulations de tableaux que nous allons rencontrer :

Accès aux attributs d'un tableau

Pour connaître la taille, la forme, l'occupation mémoire et le type de données d'un tableau.

Indexation d'un tableau

Pour lire et écrire la valeur de chaque élément individuel.

Tranchage d'un tableau

Pour obtenir et créer de petits sous-tableaux à partir d'un grand.

Reformage d'un tableau

Pour changer la forme, la géométrie d'un tableau.

Compactage et éclatement d'un tableau

Pour combiner plusieurs tableaux en un seul et pour en répartir un en plusieurs.

Accès aux attributs d'un tableau NumPy

Voyons d'abord quelques attributs de tableau qui nous seront utiles. Pour les exemples, nous allons créer trois tableaux contenant des valeurs aléatoires : un à une dimension, un à deux dimensions et un à trois dimensions. Nous nous servons à cet effet du générateur de nombres pseudo-aléatoires de NumPy. Nous spécifions une graine de génération (*seed*) pour que la même série de valeurs aléatoires soit produite à chaque exécution du code :

```
In[1]: import numpy as np  
        np.random.seed(0) # Graine pour  
        reproductibilité  
        x1 = np.random.randint(10, size=6)
```

```
# Une dimension  
    x2 = np.random.randint(10, size=(3, 4))  
# Deux dimensions  
    x3 = np.random.randint(10, size=(3, 4, 5))  
# Trois dimensions
```

Le nombre de dimensions correspond à l'attribut `ndim`, la taille de chaque dimension à `shape` et la taille totale du tableau à `size` :

```
In[2]: print("x3 ndim: ", x3.ndim)  
        print("x3 shape:", x3.shape)  
        print("x3 size: ", x3.size)  
x3 ndim: 3  
x3 shape: (3, 4, 5)  
x3 size: 60
```

N'oublions pas l'indispensable attribut `dtype`, qui indique le type de données contenues dans le tableau, comme nous l'avons déjà vu lorsque nous avons décrit les types de données Python un peu plus haut :

```
In[3]: print( dtype: , x3.dtype)  
dtype: int64
```

Deux autres attributs qui nous intéressent sont `itemsize` qui contient la taille en octets de chaque élément du tableau et `nbytes` qui renvoie la taille totale occupée par le tableau en mémoire :

```
In[4]: print( itemsize: , x3.itemsize, bytes )
          print( nbytes: , x3.nbytes, bytes )
itemsize: 8 bytes
nbytes: 480 bytes
```

En général, on peut supposer que nbytes vaut itemsize multiplié par size.

Index de tableaux et accès à un élément

Si vous connaissez déjà les index de liste standard de Python, ceux de NumPy ne vous bouleverseront pas. Pour accéder au n-ième élément d'un tableau à une dimension, en commençant bien sûr par zéro, vous indiquez la valeur d'index entre crochets, comme pour une liste Python :

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
```

```
In[6]: x1[0]
Out[6]: 5
```

```
In[7]: x1[4]
Out[7]: 7
```

Pour obtenir un élément en partant de la fin du tableau, il suffit de fournir un index négatif :

```
In[8]: x1[-1]
```

```
Out[8]: 9
```

```
In[9]: x1[-2]
```

```
Out[9]: 7
```

Dans un tableau à plusieurs dimensions, vous indiquez des tuples d'index séparés par des virgules :

```
In[10]: x2
```

```
Out[10]: array([[3, 5, 2, 4],  
                 [7, 6, 8, 8],  
                 [1, 6, 7, 7]])
```

```
In[11]: x2[0, 0]
```

```
Out[11]: 3
```

```
In[12]: x2[2, 0]
```

```
Out[12]: 1
```

```
In[13]: x2[2, -1]
```

```
Out[13]: 7
```

Cette notation avec index permet bien sûr de modifier les valeurs désignées :

```
In[14]: x2[0, 0] = 12
```

```
x2
```

```
Out[14]: array([[12, 5, 2, 4],
```

```
[ 7, 6, 8, 8],  
[ 1, 6, 7, 7]])
```

Rappelons une fois de plus qu'un tableau NumPy ne contient qu'un type de valeurs. Si vous tentez d'insérer une valeur à virgule flottante dans un tableau d'entiers, la valeur sera tronquée sans vous en avertir. Méfiez-vous de ce comportement !

```
In[15]: x1[0] = 3.14159 # ceci va être tronqué  
!  
x1  
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Tranchage d'un tableau pour accéder à un sous-tableau

La notation avec une paire de crochets droits permet d'accéder à un élément, mais également d'accéder à un sous-tableau, ce qui correspond à la notation de *tranchage* (*slice*). Il suffit d'utiliser le signe (:). La syntaxe est la même que pour les listes Python standard ; voici comment accéder à une tranche d'un tableau x :

```
x[start:stop:step]
```

Lorsqu'un des trois paramètres n'est pas spécifié, il reçoit une valeur standard. Pour start, c'est 0 ; pour stop, c'est la

Sous-tableaux à une dimension

```
In[16]: x = np.arange(10)
```

```
    x
```

```
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[17]: x[:5] # Les 5 premiers éléments
```

```
Out[17]: array([0, 1, 2, 3, 4])
```

```
In[18]: x[5:] # Ceux après l'index 5
```

```
Out[18]: array([5, 6, 7, 8, 9])
```

```
In[19]: x[4:7] # Le sous-tableau du milieu
```

```
Out[19]: array([4, 5, 6])
```

```
In[20]: x[::-2] # Un sur deux
```

```
Out[20]: array([0, 2, 4, 6, 8])
```

```
In[21]: x[1::2] # Un sur deux à partir de  
l'index 1
```

```
Out[21]: array([1, 3, 5, 7, 9])
```

Une confusion est possible lorsque la valeur de step est négative. En effet, les valeurs par défaut de start et de stop

sont dans ce cas inversées. Cette astuce permet de renverser aisément le contenu d'un tableau :

```
In[22]: x[::-1] # Tous, à reculons  
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
In[23]: x[5::-2] # Idem à partir de l'index 5  
Out[23]: array([5, 3, 1])
```

Sous-tableaux à plusieurs dimensions

Le tranchage d'un tableau à plusieurs dimensions fonctionne de la même manière, les différentes tranches étant séparées par des virgules :

```
In[24]: x2  
Out[24]: array([[12, 5, 2, 4],  
                 [ 7, 6, 8, 8],  
                 [ 1, 6, 7, 7]])
```

```
In[25]: x2[:2, :3] # Deux lignes et trois colonnes
```

```
Out[25]: array([[12, 5, 2],  
                  [ 7, 6, 8]])
```

```
In[26]: x2[:3, ::2] # Toutes les lignes, une colonne sur deux
```

```
Out[26]: array([[12, 2],  
                  [ 7, 8],  
                  [ 1, 7]])
```

Vous pouvez également inverser une dimension de sous-tableau :

```
In[27]: x2[::-1, ::-1]
Out[27]: array([[7, 7, 6, 1],
                 [8, 8, 6, 7],
                 [4, 2, 5, 12]])
```

Accès aux lignes et aux colonnes d'un tableau

Vous aurez souvent besoin d'accéder à une seule ligne ou colonne. Pour y parvenir, il suffit de combiner index et tranchage en spécifiant une tranche vide au moyen d'un signe (:) isolé :

```
In[28]: print(x2[:, 0])      # 1re colonne de x2
[12 7 1]
```

```
In[29]: print(x2[0, :])      # 1re ligne de x2
[12 5 2 4]
```

Pour accéder à une ligne, vous pouvez omettre la tranche vide, ce qui permet une syntaxe plus compacte :

```
In[30]: print(x2[0])      # Équivaut à x2[0, :]
[12 5 2 4]
```

Un sous-tableau est une vue, pas une copie

Lorsque vous demandez une tranche à partir d'une liste Python, vous obtenez une copie des données correspondantes, ce qui n'est pas le cas dans un tableau NumPy. Une tranche NumPy est une vue sur les données et non une copie de ces données. Repartons du tableau à deux dimensions précédent :

```
In[31]: print(x2)
[[12, 5, 2, 4],
 [ 7, 6, 8, 8],
 [ 1, 6, 7, 7]]
```

Demandons l'extraction d'un sous-tableau de 2×2 :

```
In[32]: x2_sub = x2[:2, :2]
print(x2_sub)
[[12 5]
 [ 7 6]]
```

Si nous modifions ce sous-tableau, nous constatons que le tableau initial a changé lui aussi :

```
In[33]: x2_sub[0, 0] = 99
print(x2_sub)
[[99 5]
 [ 7 6]]
```

```
In[34]: print(x2)
```

```
[[99 5 2 4]
 [ 7 6 8 8]
 [ 1 6 7 7]]
```

Ce comportement standard est en général bienvenu : sur un énorme jeu de données, cela permet d'intervenir sur des portions du jeu sans devoir au préalable copier le tampon contenant les données.

Création d'une copie d'un sous-tableau

Les vues sont généralement appropriées au besoin, mais vous aurez dans certains cas besoin de faire réellement une copie des données pour un sous-tableau. Il suffit d'utiliser dans ce cas la méthode `copy()` :

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
         print(x2_sub_copy)
[[99 5]
 [ 7 6]]
```

Si nous modifions le sous-tableau, nous voyons que le tableau du départ n'est plus modifié :

```
In[36]: x2_sub_copy[0, 0] = 42
         print(x2_sub_copy)
[[42 5]
 [ 7 6]]
```

```
In[37]: print(x2)
```

```
[[99 5 2 4]
 [ 7 6 8 8]
 [ 1 6 7 7]]
```

Reformage d'un tableau (reshape)

Un autre type d'opération très utile concernant les tableaux est le *reformage (reshaping)*, notamment au moyen de la méthode `reshape()`. Voici par exemple comment repositionner les valeurs de 1 à 9 dans une grille de 3×3 :

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

La taille du tableau initial doit coïncider avec le tableau résultant. Cette méthode `reshape` utilise si possible une vue sans copie des données, mais ce n'est pas toujours le cas lorsque les tampons mémoire ne sont pas adjacents.

Une autre opération de reformage fréquente est la conversion d'un tableau à une dimension vers un tableau à deux dimensions, une matrice. On peut utiliser la même méthode `reshape` ou profiter du mot-clé `newaxis` dans une opération de tranchage :

```
In[39]: x = np.array([1, 2, 3])
        # Vecteur ligne avec reshape
        x.reshape((1, 3))
```

```
Out[39]: array([[1, 2, 3]])
```

```
In[40]: # Vecteur ligne avec newaxis
        x[np.newaxis, :]
```

```
Out[40]: array([[1, 2, 3]])
```

```
In[41]: # Vecteur colonne avec reshape
        x.reshape((3, 1))
```

```
Out[41]: array([[1],
                [2],
                [3]])
```

```
In[42]: # Vecteur colonne avec newaxis
        x[:, np.newaxis]
```

```
Out[42]: array([[1],
                [2],
                [3]])
```

Nous utiliserons souvent ce genre de transformation dans la suite du livre.

Concaténation et partage de tableau

Les opérations précédentes concernaient toutes un tableau unique, mais il est tout à fait possible de fusionner plusieurs

tableaux en un seul et de distribuer un tableau en plusieurs.

Concaténation de plusieurs tableaux

Concaténer, c'est fusionner au moins deux tableaux NumPy au moyen de trois routines : np.concatenate, np.vstack et np.hstack. La première attend en premier paramètre un tuple ou une liste de tableaux :

```
In[43]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])
Out[43]: array([1, 2, 3, 3, 2, 1])
```

Vous pouvez bien sûr concaténer plusieurs tableaux :

```
In[44]: z = [99, 99, 99]
         print(np.concatenate([x, y, z]))
[ 1 2 3 3 2 1 99 99 99]
```

La même fonction permet de traiter les tableaux à deux dimensions :

```
In[45]: grid = np.array([[1, 2, 3],
                      [4, 5, 6]])
```

```
In[46]: # Concaténation selon le premier axe
         np.concatenate([grid, grid])
Out[46]: array([[1, 2, 3],
                [4, 5, 6],
```

```
[1, 2, 3],  
[4, 5, 6]))
```

```
In[47]: # concaténation selon le second axe  
indéxé par zéro
```

```
    np.concatenate([grid, grid], axis=1)
```

```
Out[47]: array([[1, 2, 3],  
                 [4, 5, 6],  
                 [1, 2, 3],  
                 [4, 5, 6]])
```

Lorsque les tableaux à fusionner n'ont pas le même nombre de dimensions, il peut être plus lisible d'utiliser les fonctions spécialisées d'empilement vertical np.vstack ou horizontal np.hstack :

```
In[48]: x = np.array([1, 2, 3])  
        grid = np.array([[9, 8, 7],  
                        [6, 5, 4]])  
        # Empilement vertical des tableaux  
        np.vstack([x, grid])
```

```
Out[48]: array([[1, 2, 3],  
                  [9, 8, 7],  
                  [6, 5, 4]])
```

```
In[49]: # Empilement horizontal des tableaux  
y = np.array([[99],  
              [99]])  
np.hstack([grid, y])
```

```
Out[49]: array([[ 9,  8,  7, 99],
   [ 6,  5,  4, 99]]))
```

Vous disposez enfin de np.dstack pour empiler selon le troisième axe.

Partage d'un tableau

Le partage est l'opération inverse de la concaténation et correspond aux trois fonctions np.split, np.hsplit et np.vsplit. Il faut fournir en entrée de ce genre de fonction les index qui correspondent aux points de partage :

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

Nous constatons qu'en fournissant N points de partage, nous obtenons N + 1 sous-tableaux. Les deux fonctions np.hsplit et np.vsplit s'utilisent de façon similaire :

```
In[51]: grid = np.arange(16).reshape((4, 4))
         grid
Out[51]: array([[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11],
   [12, 13, 14, 15]])
```

```
In[52]: upper, lower = np.vsplit(grid, [2])
```

```
    print(upper)
    print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
In[53]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Ici aussi, la fonction np.dsplit permet de travailler selon le troisième axe.

2.3 : Les fonctions universelles ufuncs

Pour l'instant, nous n'avons fait que découvrir les mécanismes élémentaires de NumPy. Voyons maintenant l'une des grandes raisons de la popularité de NumPy dans le monde de la datalogie avec Python. La promesse de NumPy est en effet d'offrir une interface opérationnelle facile d'accès et souple pour effectuer des traitements optimisés sur des tableaux de données.

Les calculs NumPy sur les tableaux peuvent être très rapides ou bien très lents, tout dépendant du soin pris pour privilégier les opérations vectorisées, ce qui suppose en général d'utiliser les fonctions universelles de NumPy, appelées *ufuncs*. Nous allons voir les arguments qui poussent à adopter ces ufuncs, afin de réaliser des traitements répétitifs sur les éléments d'un tableau de façon très efficace. Nous passerons ensuite en revue la plupart des ufuncs d'usage général et d'arithmétique qui sont proposées dans le paquetage NumPy.

Les boucles ou l'éloge de la lenteur

L'implémentation standard de Python, qui est dans notre cas CPython, est très lente pour certains traitements. Les

causes en sont multiples : c'est un langage interprété et les types de données sont non définis d'avance, ce qui interdit de compiler les traitements en vue d'aboutir à du code machine exécutable aussi efficace que ce que permet le C ou le Fortran. Différentes initiatives ont tenté de combler cette faiblesse : c'est notamment le cas du projet PyPy (<http://pypy.org/>) qui est une compilation juste à temps de Python ; du projet Cython (<http://cython.org>) qui convertit le code Python en code C compilable ou encore du projet Numba (<http://numba.pydata.org/>) qui convertit des blocs de code Python en *octocode* LLVM performant (*bytecode*). Ces différentes propositions ont leurs points forts et leurs faiblesses, mais on peut affirmer qu'aucune ne peut rivaliser en termes de popularité avec le moteur CPython standard.

La relative lenteur de Python se remarque notamment dans les répétitions de petits traitements, par exemple lorsqu'il s'agit de répéter en boucle la même opération sur tous les éléments d'un tableau. Supposons que nous voulions faire calculer l'inverse de plusieurs valeurs. Voici l'approche qui semble la plus naturelle :

```
In[1]: import numpy as np
        np.random.seed(0)

def calculer_inverses(values):
    output = np.empty(len(values))
```

```
for i in range(len(values)):
    output[i] = 1.0 / values[i]
return output

values = np.random.randint(1, 10,
size=5)
calculer_inverses(values)

Out[1]: array([ 0.16666667,   1.,     0.25,     0.25,
 0.125])
```

Une personne écrivant en langage C ou en Java trouvera cette approche pertinente. Mais si nous mesurons le temps d'exécution avec un grand tableau d'entrée, force est de constater que l'opération est très lente ! Nous pouvons chronométrier au moyen de la fonction magique %timeit d'IPython (vue à la fin du [Chapitre 1](#)) :

```
In[2]: gros_tablo = np.random.randint(1, 100,
size=1000000)
%timeit calculer_inverses(gros_tablo)
1 loop, best of 3: 2.91 s per loop
```

Il faut plusieurs secondes pour calculer un million d'inverses et stocker les résultats ! De nos jours, même les téléphones offrent des performances de niveau gigaflop (un milliard d'opérations par seconde). Cette lenteur est donc inadmissible. Le problème n'est pas lié à la vitesse de réalisation de l'opération demandée, mais aux opérations de

gestion du contexte que sont la vérification dynamique du type de données et la distribution des fonctions que CPython doit faire pour chaque tour de boucle. Pour chaque calcul d'un inverse, Python regarde quel est le type effectif de l'objet pour faire une recherche dynamique de la fonction à utiliser. Si le code était compilé, le type serait connu avant l'exécution et le résultat pourrait être calculé bien plus rapidement, car la bonne fonction serait déjà connue.

Grands principes des ufuncs

Par bonheur, NumPy propose une interface très pratique pour accéder aux versions compilées à typage statique de nombreuses opérations. C'est cela que l'on nomme le traitement vectorisé. Pour en profiter, il suffit de demander le traitement approprié sur le tableau, chaque élément étant traité tour à tour. Dans cette approche vectorisée, la boucle de traitement est concrétisée dans la couche compilée qui soutient NumPy, et c'est ce qui permet une exécution plus rapide.

Comparons les résultats des deux appels suivants :

```
In[3]: print(calculer_inverses(values))
        print(1.0 / values)
[ 0.16666667 1.  0.25  0.25  0.125 ]
[ 0.16666667 1.  0.25  0.25  0.125 ]
```

Le chronométrage de ce grand tableau montre que la vitesse a quasiment été multipliée par mille :

```
In[4]: %timeit (1.0 / gros_tablo)
100 loops, best of 3: 4.6 ms per loop
```

Les opérations vectorisées de NumPy sont incarnées par des fonctions universelles appelées *ufuncs*. Leur seul objectif est d'exécuter rapidement la même opération sur des valeurs dans des tableaux NumPy. Ces fonctions sont très souples ; nous allons voir une opération entre un scalaire et un tableau, mais on peut également traiter deux tableaux :

```
In[5]: np.arange(5) / np.arange(1, 6)
Out[5]: array([ 0. ,  0.5 ,  0.66666667,  0.75
,  0.8 ])
```

Les ufuncs ne sont pas limitées aux tableaux à une dimension :

```
In[6]: x = np.arange(9).reshape((3, 3))
        2 ** x
Out[6]: array([[ 1,   2,    4],
               [ 8,  16,   32],
               [ 64, 128,  256]])
```

Les calculs vectorisés sont quasiment toujours plus efficaces que les mêmes calculs écrits sous forme de boucles Python

standard, et d'autant plus que le volume de données à traiter est important. Dès que vous rencontrez une boucle dans un script Python, voyez si elle ne peut pas être remplacée avantageusement par une expression vectorisée.

Découverte des ufuncs de NumPy

Il existe deux catégories d'ufuncs : les ufuncs unaires, qui n'attendent qu'une entrée, et les ufuncs binaires. Découvrons ces deux catégories tour à tour.

Arithmétique sur les tableaux

Les ufuncs de NumPy s'avèrent très faciles à utiliser parce qu'elles réutilisent les opérateurs arithmétiques standard de Python. Voici comment utiliser l'addition, la soustraction, la multiplication et la division :

```
In[7]: x = np.arange(4)
        print("x      =", x)
        print("x + 5 =", x + 5)
        print("x - 5 =", x - 5)
        print("x * 2 =", x * 2)
        print("x / 2 =", x / 2)
        print("x // 2 =", x // 2) # Division
tronquante
```

```
x      = [0 1 2 3]
x + 5 = [5 6 7 8]
```

```
x - 5 = [-5 -4 -3 -2]  
x * 2 = [0 2 4 6]  
x / 2 = [ 0. 0.5 1. 1.5]  
x // 2 = [0 0 1 1]
```

Vous disposez aussi d'une fonction pour la négation, d'une autre pour l'élévation à la puissance `**` et du modulo `%` :

```
In[8]: print("-x      = ", -x)  
       print("x ** 2 = ", x ** 2)  
       print("x % 2  = ", x % 2)  
  
-x      = [ 0 -1 -2 -3]  
x ** 2 = [0 1 4 9]  
x % 2  = [0 1 0 1]
```

Vous pouvez bien sûr combiner ces opérateurs ; l'ordre d'évaluation standard est en vigueur :

```
In[9]: -(0.5*x + 1) ** 2  
Out[9]: array([-1. , -2.25, -4. , -6.25])
```

Ces opérateurs arithmétiques réutilisent en interne les fonctions correspondantes de NumPy. L'opérateur `+` est une surcouche de la fonction `add()` :

```
In[10]: np.add(x, 2)  
Out[10]: array([2, 3, 4, 5])
```

Le tableau suivant dresse la liste des opérateurs arithmétiques disponibles dans NumPy.

Tableau 2.2 : Opérateurs arithmétiques de NumPy.

Opérateur	Équivalent ufunc	Description
+	np.add	Addition ($1 + 1 = 2$)
-	np.subtract	Soustraction ($3 - 2 = 1$)
-	np.negative	Négation unaire (-2)
*	np.multiply	Multiplication ($2 * 3 = 6$)
/	np.divide	Division ($3 / 2 = 1.5$)
//	np.floor_divide	Division tronquante ($3 // 2 = 1$)
**	np.power	Exponentiation ($2 ** 3 = 8$)
%	np.mod	Modulo/reste ($9 \% 4 = 1$)

Il existe enfin des opérateurs pour travailler sur les bits et en logique booléenne. Nous les verrons dans la section appropriée de ce même chapitre.

Valeur absolue

NumPy propose sa version de la fonction de calcul de la valeur absolue de Python que voici :

```
In[11]: x = np.array([-2, -1, 0, 1, 2])
          abs(x)
Out[11]: array([2, 1, 0, 1, 2])
```

Dans NumPy, l'ufunc correspondante s'écrit `np.absolute` que vous pouvez abréger en `np.abs` :

```
In[12]: np.absolute(x)
Out[12]: array([2, 1, 0, 1, 2])
```

```
In[13]: np.abs(x)
Out[13]: array([2, 1, 0, 1, 2])
```

Cette fonction est capable de gérer les données complexes, auquel cas la valeur absolue correspond à la magnitude :

```
In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0
+ 1j])
        np.abs(x)
Out[14]: array([ 5.,  5.,  2.,  1.])
```

Fonctions trigonométriques

Parmi les nombreuses ufuncs proposées, la famille de celles concernant la trigonométrie est essentielle pour le datologue. Commençons par définir un tableau d'angles :

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Nous pouvons appliquer quelques fonctions trigonométriques à ces valeurs :

```
In[16]: print("theta      = ", theta)
          print("sin(theta) = ", np.sin(theta))
          print("cos(theta) = ", np.cos(theta))
          print("tan(theta) = ", np.tan(theta))

theta      = [ 0. 1.57079633 3.14159265]
sin(theta) = [ 0.00000000e+00 1.00000000e+00
1.22464680e-16]
cos(theta) = [ 1.00000000e+00 6.12323400e-17
-1.00000000e+00]
tan(theta) = [ 0.00000000e+00 1.63312394e+16
-1.22464680e-16]
```

La précision des résultats est celle que permet le processeur concerné. Les valeurs flottantes qui devraient être exactement égales à zéro ne le sont pas toujours. Vous pouvez également utiliser les fonctions trigonométriques inverses :

```
In[17]: x = [-1, 0, 1]
          print("x      = ", x)
          print("arcsin(x) = ", np.arcsin(x))
          print("arccos(x) = ", np.arccos(x))
          print("arctan(x) = ", np.arctan(x))

x      = [-1, 0, 1]
arcsin(x) = [-1.57079633 0. 1.57079633]
arccos(x) = [ 3.14159265 1.57079633 0. ]
arctan(x) = [-0.78539816 0. 0.78539816]
```

Exposants et logarithmes

NumPy offre bien sûr de quoi réaliser des élévarions à la puissance :

```
In[18]: x = [1, 2, 3]
          print("x  =", x)
          print("e^x =", np.exp(x))
          print("2^x =", np.exp2(x))
          print("3^x =", np.power(3, x))
```

```
x    = [1, 2, 3]
e^x = [ 2.71828183 7.3890561 20.08553692]
2^x = [ 2. 4. 8.]
3^x = [ 3 9 27]
```

Nous disposons bien sûr des logarithmes, qui sont les inverses des exposants. Le logarithme naturel correspond à `np.log`, mais vous pouvez également travailler en base 2 ou en base 10, comme ceci :

```
In[19]: x = [1, 2, 4, 10]
          print("x      =", x)
          print("ln(x)   =", np.log(x))
          print("log2(x) =", np.log2(x))
          print("log10(x) =", np.log10(x))
x      = [1, 2, 4, 10]
ln(x)   = [ 0. 0.69314718 1.38629436 2.30258509]
log2(x) = [ 0. 1. 2. 3.32192809]
log10(x) = [ 0. 0.30103 0.60205999 1. ]
```

Vous disposez de quelques versions dédiées qui permettent de ne pas perdre en précision pour des valeurs d'entrées très faibles :

```
In[20]: x = [0, 0.001, 0.01, 0.1]
          print("exp(x) - 1 =", np.expm1(x))
          print("log(1 + x) =", np.log1p(x))

exp(x) - 1 = [ 0. 0.0010005 0.01005017
0.10517092]
log(1 + x) = [ 0. 0.0009995 0.00995033
0.09531018]
```

Lorsque la valeur de x est très petite, vous obtenez un résultat plus précis de cette façon qu'avec `np.log` ou `np.exp`.

Ufuncs spécialisées

NumPy offre également des ufuncs pour les calculs trigonométriques hyperboliques, pour l'arithmétique sur les bits, pour les comparaisons, les conversions entre radians et degrés, les arrondis et les restes, etc. Un parcours de la documentation NumPy permet de découvrir beaucoup de fonctions intéressantes.

Vous disposez d'un sous-module réunissant des ufuncs très spécifiques ; son nom est `scipy.special`. Dès que vous avez besoin d'appliquer une fonction mathématique spéciale à vos données, il est fort probable que vous la trouviez dans ce

module. La liste est longue, mais l'exemple suivant en utilise quelques exemples appropriés dans un contexte de statistiques :

```
In[21]: from scipy import special
```

```
In[22]: # fonction Gamma (factorielle généralisée) et autres
```

```
    x = [1, 5, 10]
    print("gamma(x)      =", special.gamma(x))
    print("ln|gamma(x)| =", special.gammaln(x))
    print("beta(x, 2)   =", special.beta(x,
2))
```

```
gamma(x)      = [ 1.00000000e+00 2.40000000e+01
3.62880000e+05]
```

```
ln|gamma(x)| = [ 0. 3.17805383 12.80182748]
```

```
beta(x, 2)   = [ 0.5 0.03333333 0.00909091]
```

```
In[23]: # Fonction d'erreur, intégrale du gaussien,
```

```
    # son complément et son inverse
    x = np.array([0, 0.3, 0.7, 1.0])
    print( erf(x)  = , special.erf(x))
    print( erfc(x) = , special.erfc(x))
    print( erfinv(x) = , special.erfinv(x))
```

```
erf(x)      = [ 0. 0.32862676 0.67780119
0.84270079]
```

```
erfc(x) = [ 1. 0.67137324 0.32219881  
0.15729921]  
erfinv(x) = [ 0. 0.27246271 0.73286908 inf]
```

Le nombre d'ufuncs disponibles dans NumPy et `scipy.special` est énorme, et nous vous renvoyons à la documentation en ligne de ces paquetages. Vous pouvez par exemple faire une recherche Web en indiquant « gamma function python ».

Caractéristiques avancées des ufuncs

Nombreux sont les utilisateurs de NumPy qui utilisent les ufuncs sans prendre la peine d'en découvrir toutes les caractéristiques. Ils ont bien tort.

Choix du conteneur des résultats

Lorsque vous préparez un calcul intensif, il est parfois pratique de pouvoir désigner le tableau dans lequel les résultats doivent être stockés. Vous évitez de provoquer la création d'un tableau temporaire en demandant de stocker directement les résultats à l'emplacement mémoire où vous voulez ensuite le trouver. Vous pouvez choisir cette destination pour toutes les ufuncs, au moyen du paramètre `out` :

```
In[24]: x = np.arange(5)
         y = np.empty(5)
         np.multiply(x, 10, out=y)
         print(y)
[ 0. 10. 20. 30. 40.]
```

Cet aiguillage est même possible avec une vue de tableau. Nous pouvons tout à fait écrire les résultats d'un calcul dans un élément sur deux du tableau désigné :

```
In[25]: y = np.zeros(10)
         np.power(2, x, out=y[::2])
         print(y)
[ 1. 0. 2. 0. 4. 0. 8. 0. 16. 0.]
```

Si nous avions écrit $y[::2] = 2^{**} x$, nous aurions provoqué la création d'un tableau temporaire pour les résultats de $2^{**} x$, ce qui aurait requis de copier dans un deuxième temps toutes ces valeurs dans le tableau y . Pour un petit calcul, la différence est négligeable, mais dans le cas d'un grand volume de données, l'économie d'espace mémoire permise par le paramètre `out` peut se révéler décisif.

Agrégats

Les ufuncs binaires permettent de produire quelques agrégats intéressants directement à partir de l'objet. Il devient aussi possible de réduire un tableau au moyen de la

méthode `reduce` d'une ufunc. Cette réduction applique l'opération demandée aux éléments du tableau jusqu'à n'obtenir qu'un résultat unique.

Voici par exemple comment appeler `reduce` sur l'ufunc `add` pour récupérer la somme de tous les éléments du tableau :

```
In[26]: x = np.arange(1, 6)
         np.add.reduce(x)
```

```
Out[26]: 15
```

En appelant `reduce` sur l'ufunc `multiply`, on obtient bien sûr le produit du contenu du tableau :

```
In[27]: np.multiply.reduce(x)
Out[27]: 120
```

Nous pouvons même conserver les résultats intermédiaires au moyen de `accumulate` :

```
In[28]: np.add.accumulate(x)
Out[28]: array([ 1, 3, 6, 10, 15])
```

```
In[29]: np.multiply.accumulate(x)
Out[29]: array([ 1, 2, 6, 24, 120])
```

Notez qu'il existe des fonctions NumPy dédiées au calcul des résultats pour ces cas particuliers (`np.sum`, `np.prod`,

`np.cumsum`, `np.cumprod`). Nous y reviendrons dans la section sur les agrégats un peu plus loin dans ce chapitre.

Produits externes

N'importe quelle ufunc est en mesure de calculer le résultat de toutes les paires d'entrées différentes au moyen de sa méthode `outer`. C'est ainsi que vous pouvez en une seule instruction créer une table de multiplication :

```
In[30]: x = np.arange(1, 6)
          np.multiply.outer(x, x)
```

```
Out[30]: array([[ 1,  2,  3,  4,  5],
                 [ 2,  4,  6,  8, 10],
                 [ 3,  6,  9, 12, 15],
                 [ 4,  8, 12, 16, 20],
                 [ 5, 10, 15, 20, 25]])
```

Deux autres ufuncs que nous découvrirons dans l'indexation fancy se nomment `ufunc.at` et

`ufunc.reduceat`.

Les ufuncs offrent un autre avantage qui est de savoir opérer entre des tableaux de tailles et de formes différentes, ce qui correspond à l'opération de diffusion (*broadcasting*). C'est un sujet suffisamment important pour que nous lui ayons dédié

une section complète un peu plus loin dans ce même chapitre.

Pour en savoir plus sur les ufuncs

Vous trouverez d'autres informations sur les fonctions universelles, et notamment la liste complète sur le site de NumPy (<http://www.numpy.org>) et celui de SciPy (<http://www.scipy.org>).

Rappelons que vous disposez d'informations directement dans IPython en important les paquetages puis en vous servant du mécanisme d'achèvement de saisie par la touche Tab et par la commande ?, comme vu dans le premier chapitre.

2.4 : Agrégats min, max et intermédiaire

Lorsque vous devez traiter un grand volume de données, il est souvent conseillé de commencer par demander des statistiques globales. Les deux mesures globales les plus demandées sont la moyenne et l'écart-type, puisqu'elles permettent d'avoir une idée des valeurs typiques du jeu. D'autres valeurs d'agrégats restent intéressantes, et notamment la somme, le produit, la médiane, le minimum et le maximum, et les différents quantiles.

NumPy propose des fonctions d'agrégation standard très performantes pour les tableaux. Découvrons-en quelques-unes.

Somme des valeurs d'un tableau

Prenons comme premier exemple la recherche de la somme de toutes les valeurs que contient un tableau. C'est possible en Python standard au moyen de la fonction sum :

```
In[1]: import numpy as np  
In[2]: L = np.random.random(100)  
       sum(L)  
Out[2]: 55.61209116604941
```

La syntaxe est quasiment la même que pour la fonction de même nom de NumPy, et le résultat est d'ailleurs le même dans les cas simples :

```
In[3]: np.sum(L)
```

```
Out[3]: 55.612091166049424
```

La version de NumPy est en revanche beaucoup plus rapide puisqu'elle est réalisée par du code compilé :

```
In[4]: gros_tablo = np.random.rand(1000000)
%timeit sum(gros_tablo)
%timeit np.sum(gros_tablo)
```

```
10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

Soyez cependant vigilant : la fonction `sum` et la fonction `np.sum` ne sont pas identiques et des confusions peuvent en résulter ! Les paramètres facultatifs n'ont pas la même signification et `np.sum` sait gérer des tableaux à plusieurs dimensions comme nous allons le voir.

Minimum et maximum

Python propose les fonctions `min` et `max` pour trouver les valeurs limites d'un tableau :

```
In[5]: min(gros_tablo), max(gros_tablo)
Out[5]: (1.1717128136634614e-06,
0.9999976784968716)
```

La syntaxe des fonctions correspondantes de NumPy est la même mais elles fonctionnent bien plus vite :

```
In[6]: np.min(gros_tablo), np.max(gros_tablo)
Out[6]: (1.1717128136634614e-06,
0.9999976784968716)
```

```
In[7]: %timeit min(gros_tablo)
        %timeit np.min(gros_tablo)
10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

Pour plusieurs fonctions d'agrégations de NumPy, vous pouvez utiliser une syntaxe basée sur l'appartenance des méthodes à l'objet tableau :

```
In[8]: print(gros_tablo.min(), gros_tablo.max(),
gros_tablo.sum())
1.17171281366e-06    0.999997678497
499911.628197
```

Arrangez-vous pour toujours utiliser la version NumPy des agrégats lorsque vous travaillez avec les tableaux NumPy.

Agrégats à plusieurs dimensions

Une action d'agrégation fréquemment demandée consiste à agréger selon une ligne ou une colonne. Partons d'un tableau à deux dimensions :

```
In[8]: M = np.random.random((3, 4))
print(M)
```

```
[[ 0.8967576  0.03783739  0.75952519  0.06682827]
 [ 0.8354065  0.99196818  0.19544769  0.43447084]
 [ 0.66859307  0.15038721  0.37911423  0.6687194 ]]
```

Par défaut, les fonctions d'agrégation de NumPy renvoient l'agrégat pour le tableau entier :

```
In[9]: M.sum()
```

```
Out[10]: 6.0850555667307118
```

Vous pouvez fournir un paramètre facultatif pour désigner l'axe selon lequel l'agrégat doit être calculé. Voici par exemple comment trouver la valeur minimale dans chacune des colonnes, en indiquant axis=0 :

```
In[10]: M.min(axis=0)
```

```
Out[11]: array([ 0.66859307,  0.03783739,
 0.19544769,  0.06682827])
```

Cette fonction renvoie bien quatre valeurs, puisqu'il y a quatre colonnes. La même chose peut être réalisée pour travailler par ligne :

```
In[11]: M.max(axis=1)
```

```
Out[12]: array([ 0.8967576 , 0.99196818,
 0.6687194 ])
```

La façon dont l'axe est indiqué peut dérouter ceux qui proviennent d'un autre langage. Ici, le mot-clé axis doit indiquer la dimension du tableau qui doit être réduite, repliée et non celle que l'on veut récupérer. En indiquant axis=0, on demande le repliement du premier axe, et dans le cas d'un tableau à deux dimensions, cela désigne les valeurs de chaque colonne. Ce sont les valeurs de chaque colonne qui seront agrégées.

Autres fonctions d'agrégation

Nous ne décrirons pas les nombreuses fonctions d'agrégation de NumPy. La plupart d'entre elles disposent d'une variante avec gestion des valeurs numériques manquantes, NaN. Ces valeurs sont codées par la valeur à virgule flottante standardisée IEEE nommée NaN (*Not a Number*). Nous revenons sur les données manquantes dans la section correspondante du [Chapitre 3](#). Certaines de ces

variantes NaN ne sont apparues que dans la version 1.8 de NumPy.

Le [Tableau 2.3](#) dresse la liste des fonctions d'agrégation de NumPy.

Tableau 2.3 : Fonctions d'agrégation de NumPy.

Fonction	Variante NaN	Description
np.sum	np.nansum	Calcule la somme des éléments.
np.prod	np.nanprod	Calcule le produit des éléments.
np.mean	np.nanmean	Calcule la moyenne des éléments.
np.std	np.nanstd	Calcule l'écart-type.
np.var	np.nanvar	Calcule la variance.
np.min	np.nanmin	Trouve la valeur minimale.
np.max	np.nanmax	Trouve la valeur maximale.
np.argmin	np.nanargmin	Trouve l'index de la valeur minimale.
np.argmax	np.nanargmax	Trouve l'index de la valeur maximale.
np.median	np.nanmedian	Calcule la médiane des éléments.
np.percentile	np.nanpercentile	Calcule les statistiques selon le rang.
np.any	N/A	Évalue si un élément vaut True.
np.all	N/A	Évalue si tous les éléments valent True.

Nous utiliserons ces différentes fonctions dans la suite du livre.

Un exemple : taille moyenne des présidents des USA

Les agrégats de NumPy permettent aisément de résumer une série de valeurs. Partons de la taille des présidents des USA, données qui sont disponibles dans le fichier *president_heights.csv*, une liste de labels et de valeurs séparées par des virgules :

```
In[13]: !head -4 data/president_heights.csv
```

```
order, name, height(cm)
1, George Washington, 189
2, John Adams, 170
3, Thomas Jefferson, 189
```

Nous allons avoir recours au paquetage Pandas que nous verrons plus en détail dans le chapitre suivant. Il va nous permettre de lire le contenu du fichier pour extraire les données (les tailles sont exprimées en centimètres) :

```
In[14]: import pandas as pd
        data =
pd.read_csv('data/president_heights.csv')
tailles = np.array(data['height(cm)'])
print(tailles)
```

```
[189 170 189 163 183 171 185 168 173 183 173 173]
```

```
175 178 183 193 178 173  
174 183 183 168 170 178 182 180 183 178 182 188  
175 179 183 193 182 183  
177 185 188 188 182 185]
```

Puisque nous obtenons un tableau, nous allons pouvoir lui faire dire quelques statistiques globales :

```
In[15]: print("Taille moyenne: ",  
tailles.mean())  
        print("Ecart-type:      ", tailles.std())  
        print("Taille minimale: ", tailles.min())  
        print("Taille maximale: ", tailles.max())
```

Taille moyenne: 179.738095238

Ecart-type: 6.93184344275

Taille minimale: 163

Taille maximale: 193

Vous constatez que l'opération d'agrégation a réduit à chaque fois le tableau en une seule valeur, ce qui permet d'en apprendre plus au sujet de la distribution de ces valeurs. Nous pouvons également calculer des quantiles :

```
In[16]: print("25ème percentile: ",  
np.percentile(tailles, 25))  
        print("Médiane:      ",  
np.median(tailles))  
        print("75ème percentile: ",  
np.percentile(tailles, 75))
```

```
25ème percentile: 174.25
Médiane:          182.0
75ème percentile: 183.0
```

Nous voyons que la taille médiane s'élève à 182 cm.

Il est bien sûr intéressant de demander une représentation graphique, ce que nous allons faire avec Matplotlib (qui sera décrit dans le [Chapitre 4](#)). Voici comment produire le diagramme montré en [Figure 2.3](#):

```
In[17]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()      # style
de tracé
```

```
In[18]: plt.hist(tailles)
        plt.title('Distribution des tailles des
présidents des USA')
        plt.xlabel('Taille (cm)')
        plt.ylabel('Numéro');
```

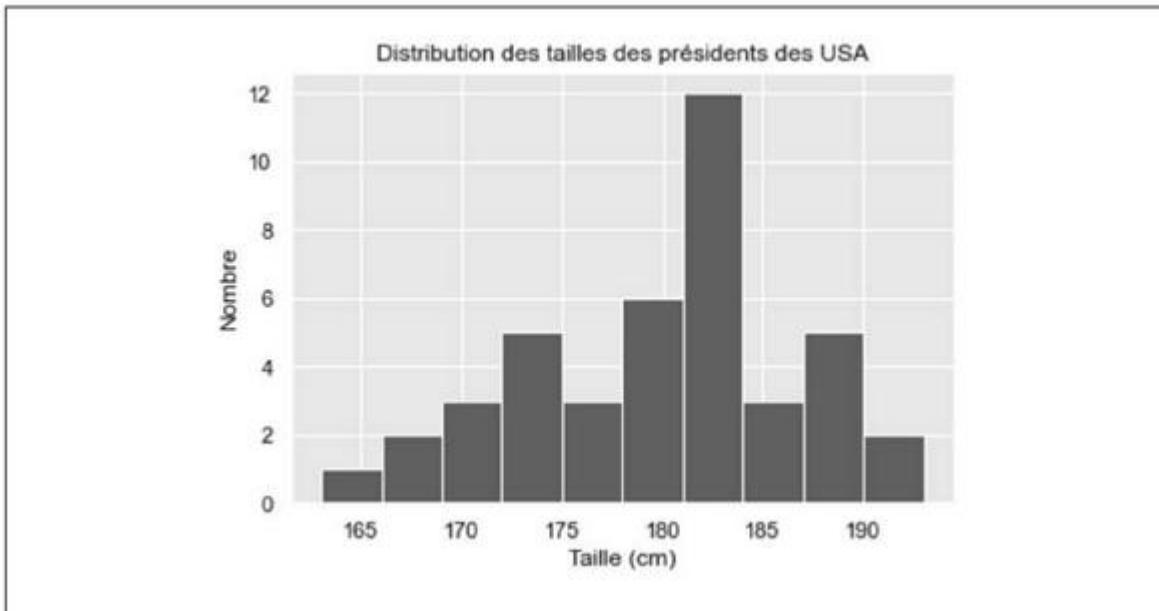


Figure 2.3 : Histogramme des tailles des présidents des USA.

Les agrégats constituent des briques de base de l'analyse de données exploratoire ; nous les retrouverons en détail dans les chapitres suivants.

2.5 : Calculs sur les tableaux : diffusion (broadcast)

Nous venons de voir comment les fonctions universelles de NumPy permettaient de vectoriser les opérations et donc d'éviter le recours aux lentes boucles Python. Une autre technique de vectorisation correspond au mécanisme de diffusion (*broadcasting*) de NumPy. Il s'agit d'un ensemble de règles pour appliquer des ufuncs binaires à des tableaux de tailles variées, par exemple pour des additions, des soustractions et des multiplications.

Présentation de la diffusion broadcasting

Les opérations binaires sont appliquées élément par élément, lorsque les tableaux ont la même taille :

```
In[1]: import numpy as np
```

```
In[2]: a = np.array([0, 1, 2])
        b = np.array([5, 5, 5])
        a + b
```

```
Out[2]: array([5, 6, 7])
```

Grâce au mécanisme de diffusion, les mêmes opérations peuvent être réalisées sur des tableaux de tailles différentes. On peut ainsi additionner une valeur scalaire (une sorte de tableau à dimension zéro) à un tableau :

```
In[3]: a + 5
```

```
Out[3]: array([5, 6, 7])
```

L'opération doit être vue comme s'il y avait duplication de la valeur unique 5 dans un pseudo tableau [5, 5, 5] avant de faire l'opération. L'avantage de NumPy est que la duplication n'est pas réalisée en réalité ; elle n'est ici qu'un outil pour faciliter la compréhension.

Cette extension s'applique également à des tableaux ayant plusieurs dimensions. Voici l'addition d'un tableau à une dimension à un autre à deux dimensions :

```
In[4]: M = np.ones((3, 3))  
M
```

```
Out[4]: array([[ 1.,  1.,  1.],  
               [ 1.,  1.,  1.],  
               [ 1.,  1.,  1.]])
```

```
In[5]: M + a
```

```
Out[5]: array([[ 1.,  2.,  3.],
```

```
[ 1., 2., 3.],  
[ 1., 2., 3.]])
```

Le tableau *a* est d'abord étendu, c'est-à-dire diffusé, selon la seconde dimension pour que sa forme soit adaptée à celle du tableau *M*.

Ces exemples sont faciles à comprendre, mais la situation peut devenir complexe lorsqu'il s'agit de diffuser dans deux tableaux à la fois. Voici un troisième exemple :

```
In[6]: a = np.arange(3)  
        b = np.arange(3)[:, np.newaxis]  
        print(a)  
        print(b)
```

```
[0 1 2]  
[[0]  
 [1]  
 [2]]
```

```
In[7]: a + b
```

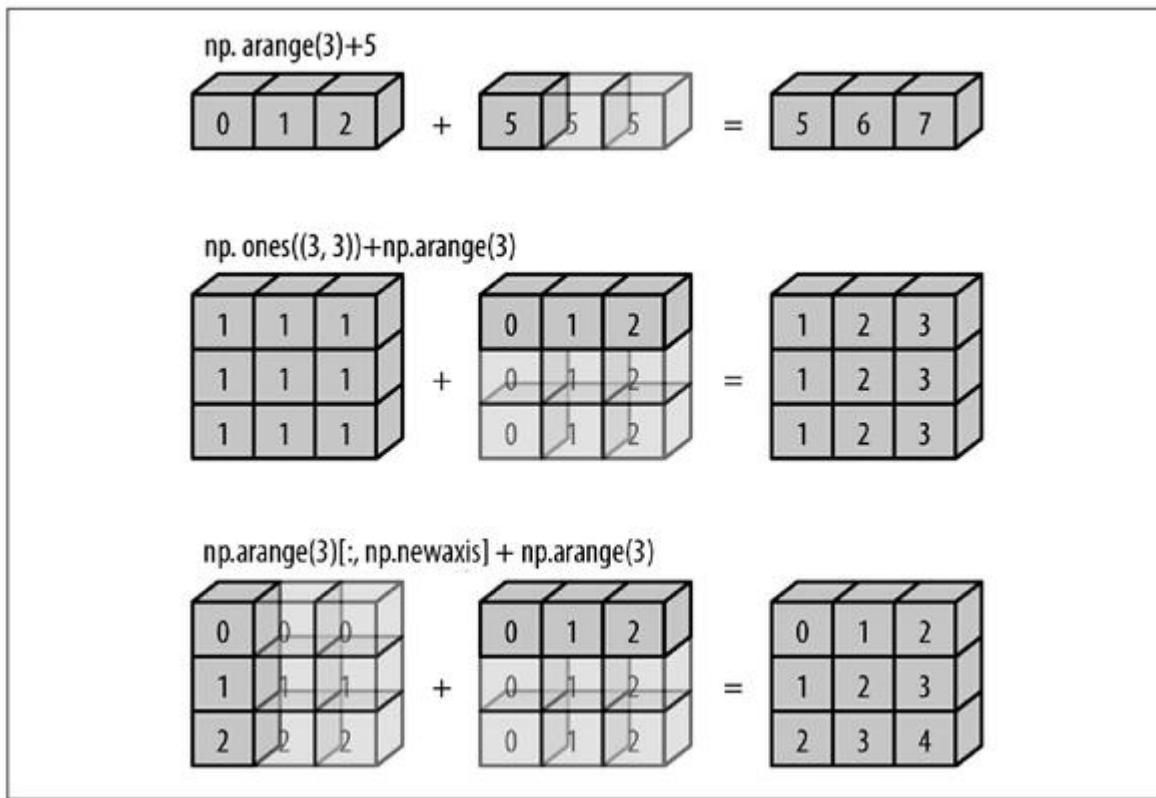
```
Out[7]: array([[0, 1, 2],  
               [1, 2, 3],  
               [2, 3, 4]])
```

Ici aussi, une valeur est diffusée pour coïncider avec la forme de l'autre tableau. Dans l'exemple, nous avons étendu *a* et *b* pour aboutir à une forme commune, ce qui produit en

résultat un tableau à deux dimensions ! La [Figure 2.4](#) montre la géométrie de l'opération (#1).



Vous trouverez le code pour réaliser ce dessin dans l'archive des exemples du livre.



[Figure 2.4](#) : Aspect visuel d'une opération de diffusion broadcasting NumPy.

Les cases en gris clair sont des valeurs produites par diffusion. Souvenez-vous qu'il n'y a pas d'occupation d'espace mémoire supplémentaire.

Règles de diffusion

La diffusion NumPy obéit à une série de règles très strictes pour contrôler les interactions se produisant entre les deux tableaux :

- Règle 1 : si le nombre de dimensions n'est pas le même dans les deux tableaux, le tableau qui a le moins de dimensions est rempli avec des 1 du côté gauche.
- Règle 2 : si les formes des deux tableaux ne coïncident dans aucune des dimensions, c'est le tableau dont la forme est égale à 1 dans cette dimension qui est étendu pour correspondre à l'autre.
- Règle 3 : si les tailles divergent dans toutes les dimensions et qu'aucune n'est égale à 1, cela déclenche une erreur.

Illustrons ces règles avec quelques exemples.

Exemple de diffusion 1

Nous voulons ajouter un tableau M ayant deux dimensions à un tableau a ayant une dimension :

```
In[8]: M = np.ones((2, 3))
         a = np.arange(3)
```

Nous voulons lancer une opération sur ces deux tableaux.
Voici leur forme :

```
M.shape = (2, 3)  
a.shape = (3, )
```

La règle 1 indique que le tableau *a* a moins de dimensions et doit être rempli à gauche avec des 1 :

```
M.shape -> (2, 3)  
a.shape -> (1, 3)
```

La règle 2 traite le fait que la première dimension diverge ; nous l'étendons :

```
M.shape -> (2, 3)  
a.shape -> (2, 3)
```

Les formes coïncident dorénavant et nous voyons que la forme finale vaudra (2, 3) :

In[9]: M + a

Out[9]: array([[1., 2., 3.],
 [1., 2., 3.]])

Exemple de diffusion 2

Voyons ce qu'il se passe lorsqu'il faut diffuser dans les deux tableaux :

```
In[10]: a = np.arange(3).reshape((3, 1))  
        b = np.arange(3)
```

Nous commençons par vérifier les formes des tableaux :

```
a.shape = (3, 1)  
b.shape = (3, )
```

D'après la règle 1, il faut remplir la forme de *b* avec des 1 :

```
a.shape -> (3, 1)  
b.shape -> (1, 3)
```

La règle 2 nous demande d'étendre ces nouvelles valeurs pour faire coïncider la dimension avec celle de l'autre tableau :

```
a.shape -> (3, 3)  
b.shape -> (3, 3)
```

Les formes sont alors devenues compatibles :

```
In[11]: a + b
```

```
Out[11]: array([[0, 1, 2],
```

```
[1, 2, 3],  
[2, 3, 4]])
```

Exemple de diffusion 3

Voyons enfin le cas dans lequel les deux tableaux ne sont pas compatibles :

```
In[12]: M = np.ones((3, 2))  
a = np.arange(3)
```

La situation est légèrement différente de celle du premier exemple : c'est la matrice M qui est transposée. Le calcul en est changé. Voici d'abord le format initial des tableaux :

```
M.shape = (3, 2)  
a.shape = (3, )
```

D'après la règle 1, il s'agit de compléter la forme de a avec des 1 :

```
M.shape -> (3, 2)  
a.shape -> (1, 3)
```

D'après la règle 2, la première dimension de a doit être étendue pour coïncider avec celle de M :

```
M.shape -> (3, 2)  
a.shape -> (3, 3)
```

Nous arrivons à la règle 3, mais les formes ne convergent pas. Autrement dit, les deux tableaux sont incompatibles, ce que prouve une tentative de lancer l'opération :

```
In[13]: M + a
```

```
ValueError Traceback (most recent call last)
<ipython-input-13-9e16e9f98da6> in <module>()
----> 1 M + a
ValueError: operands could not be broadcast
together with shapes (3, 2) (3, )
```

Méfiez-vous de la confusion possible : vous pourriez croire que vous réussirez à rendre a et M compatibles en remplaçant par exemple la forme de a avec des 1 du côté droit. Mais les règles de diffusion ne fonctionnent pas ainsi ! Cette possibilité peut parfois s'avérer utile, mais vous allez entraîner une ambiguïté. Si vous voulez vraiment remplir par la droite, vous pouvez le faire de façon explicite en reformant le tableau. Nous pouvons utiliser le mot-clé `np.newaxis` décrit dans les principes des tableaux NumPy un peu plus haut dans ce chapitre :

```
In[14]: a[:, np.newaxis].shape
Out[14]: (3, 1)
```

```
In[15]: M + a[:, np.newaxis]
Out[15]: array([[ 1.,  1.],
```

```
[ 2., 2.],  
[ 3., 3.]])
```

Ici, nous nous sommes intéressés à l'opérateur d'addition `+`, mais ces règles de diffusion s'appliquent à toutes les ufuncs binaires. Voici par exemple comment utiliser la fonction `logaddexp(a, b)` qui calcule le logarithme $\log(\exp(a) + \exp(b))$ de façon plus précise que dans l'approche naïve :

```
In[16]: np.logaddexp(M, a[:, np.newaxis])  
Out[16]: array([[ 1.31326169, 1.31326169],  
                 [ 1.69314718, 1.69314718],  
                 [ 2.31326169, 2.31326169]])
```

La diffusion en pratique

Puisque les opérations de diffusion forment un ingrédient de nombreux exemples de la suite du livre, découvrons par la pratique dans quels domaines ils peuvent se montrer utiles.

Centrage d'un tableau

Nous venons de voir que l'utilisation des fonctions universelles ufuncs permettait d'éviter l'écriture des boucles Python trop lentes. Le mécanisme de diffusion prolonge ces possibilités. Prenons l'exemple du centrage d'un tableau de données. Partons d'un tableau de dix observations, chacune regroupant trois valeurs. Si nous utilisons la convention

standard (elle sera rappelée dans le [Chapitre 5](#) dans la section concernant Scikit-Learn), nous stockons nos valeurs dans un tableau de 10 sur 3 :

```
In[17]: X = np.random.random((10, 3))
```

Pour obtenir la moyenne de chaque caractéristique, nous nous servons de la fonction d'agrégation mean appliquée à la première dimension :

```
In[18]: Xmean = X.mean(0)
```

Xmean

```
Out[18]: array([ 0.53514715, 0.66567217,  
 0.44385899])
```

Pour centrer le tableau X, nous soustrayons la moyenne, ce qui est une opération de diffusion :

```
In[19]: X_centered = X - Xmean
```

Nous vérifions que l'opération est correcte en demandant de calculer si le tableau centré a effectivement une moyenne proche de zéro :

```
In[20]: X_centered.mean(0)
```

```
Out[20]: array([ 2.22044605e-17, -7.77156117e-  
 17, -1.66533454e-17])
```

Dans les limites de précision offertes par la machine, la moyenne est bien égale à zéro.

Traçage d'une fonction à deux dimensions

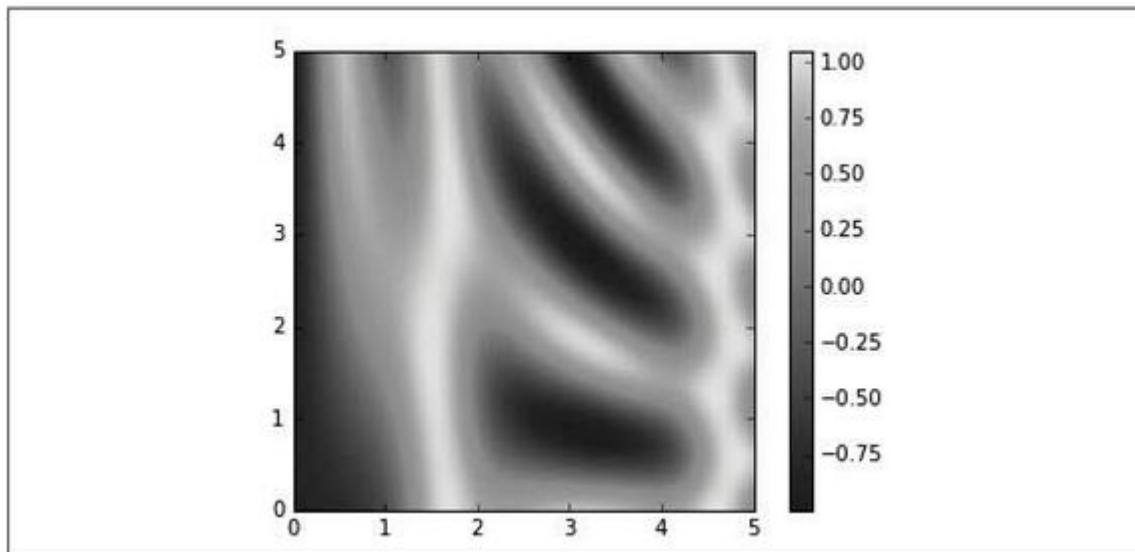
Le mécanisme de diffusion se montre très utile lorsqu'il s'agit d'afficher des graphiques à partir d'une fonction bidimensionnelle. Pour une fonction $z = f(x, y)$, nous utilisons la diffusion pour appliquer la fonction à travers la grille :

```
In[21]: # x et y ont 50 étapes de 0 à 5  
x = np.linspace(0, 5, 50)  
y = np.linspace(0, 5, 50)[:, np.newaxis]  
  
z = np.sin(x) ** 10 + np.cos(10 + y * x) *  
np.cos(x)
```

Nous profitons du paquetage de tracé graphique Matplotlib pour tracer une image du tableau à deux dimensions (nous verrons cet outil graphique dans le [Chapitre 4](#)) :

```
In[22]: %matplotlib inline  
import matplotlib.pyplot as plt  
  
In[23]: plt.imshow(z, origin='lower', extent=[0,  
5, 0, 5], cmap='viridis')  
plt.colorbar();
```

La [Figure 2.5](#) montre le résultat : une belle visualisation de la fonction bidimensionnelle.



[Figure 2.5](#) : Visualisation d'un tableau 2D.

2.6 : Comparaisons, masques et logique booléenne

Voyons dans cette section comment profiter des masques booléens pour étudier et manipuler les valeurs d'un tableau NumPy. L'opération de masquage permet d'extraire, de modifier, de dénombrer et, plus généralement, d'exploiter des valeurs en fonction de certains critères. On peut par exemple compter les valeurs supérieures à une valeur plancher ou éliminer toutes les valeurs aberrantes qui dépassent d'un certain plafond. Avec NumPy, c'est souvent le masquage booléen qui permet de réaliser ce genre d'opérations de la façon la plus efficace.

Un exemple : comptage des jours de pluie

Partons d'une série de données qui montre le volume de précipitations pour tous les jours d'une année dans une ville. Nous disposons parmi les exemples d'un fichier des statistiques de pluie de 2014 pour la ville de Seattle. Nous utilisons Pandas pour charger ces données (nous verrons ce paquetage dans le chapitre suivant) :

```
In[1]: import numpy as np
        import pandas as pd

        # extraction des précipitations avec
Pandas
        # vers un tableau NumPy
        precipit =
pd.read_csv('data/Seattle2014.csv')
['PRCP'].values
        pouces = precipit / 254      # 1/10mm ->
pouces
        pouces.shape

Out[1]: (365, )
```

Le tableau résultant contient 365 valeurs, donc du 1^{er} janvier au 31 décembre 2014.

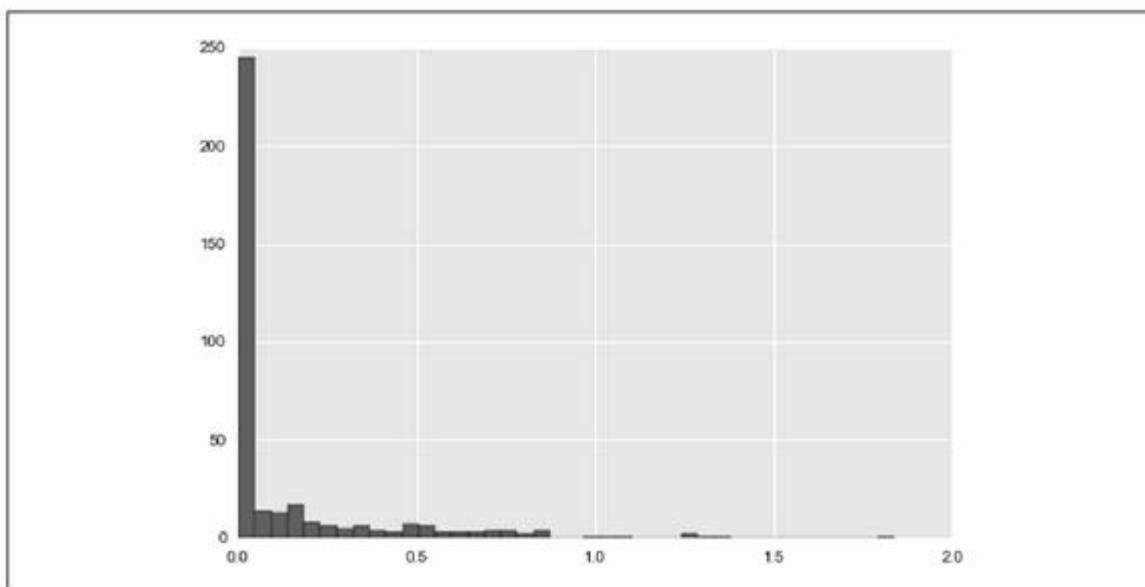
Voyons d'abord l'aspect graphique de la répartition des pluies au cours de l'année. La [Figure 2.6](#) a été générée avec l'outil Matplotlib que nous verrons dans le [Chapitre 4](#) :

```
In[2]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()
```

```
In[3]: plt.hist(pouces, 40);
```

Cet histogramme nous donne une première idée de ce que signifient ces données : malgré sa réputation, la ville de

Seattle a connu peu de jours de pluie en 2014. Mais ce diagramme ne nous montre pas les informations dont nous aimerais profiter : combien y a-t-il de jours de pluie dans une année ? Qu'elle a été la moyenne de pluie ces jours-là ? Combien y a-t-il eu de jours avec plus de 5 cm de précipitation ?



[Figure 2.6](#) : Histogramme des pluies à Seattle en 2014.

Plongée dans les données

Il reste bien sûr possible de répondre à ces questions à la main, en écrivant des boucles pour parcourir les données et en incrémentant un compteur à chaque rencontre d'une valeur satisfaisant à un critère. Cette approche est évidemment très peu efficace, puisqu'il faut d'abord écrire le code et lui laisser le temps de s'exécuter. Nous avons vu

dans le chapitre précédent (lorsque nous les avons présentées) que les fonctions universelles étaient largement préférables aux boucles de traitement pour réaliser des opérations arithmétiques rapides sur les tableaux. Nous pouvons utiliser d'autres ufuncs pour réaliser des *comparaisons* élément par élément puis traiter les résultats afin de répondre aux questions. Laissons notre exemple de côté pour l'instant pour découvrir les outils à usage général de NumPy afin d'appliquer des *masques* et répondre rapidement aux questions posées.

Les opérateurs de comparaison sous forme d'ufuncs

Nous avons vu les ufuncs dans le chapitre précédent, en décrivant celles correspondant aux opérateurs arithmétiques. Nous avons vu les opérateurs `+`, `-`, `*`, `/` et d'autres pour traiter les éléments d'un tableau. NumPy propose également des opérateurs de comparaison tels que `<` (inférieur à) et `>` (supérieur à) sous forme d'ufuncs. Ce que produit ce genre d'opérateurs est toujours un tableau de type booléen. Les six opérateurs de comparaison standard sont disponibles :

```
In[4]: x = np.array([1, 2, 3, 4, 5])
```

```
In[5]: x < 3 # Inférieur à  
Out[5]: array([ True, True, False, False,  
False], dtype=bool)
```

```
In[6]: x > 3 # Supérieur à  
Out[6]: array([False, False, False, True, True],  
dtype=bool)
```

```
In[7]: x <= 3  
Out[7]: array([ True, True, True, False, False],  
dtype=bool)
```

```
In[8]: x >= 3  
Out[8]: array([False, False, True, True, True],  
dtype=bool)
```

```
In[9]: x != 3 # Différent de  
Out[9]: array([ True, True, False, True, True],  
dtype=bool)
```

```
In[10]: x == 3 # Égal  
Out[10]: array([False, False, True, False,  
False], dtype=bool)
```

Vous pouvez bien sûr comparer les éléments de deux tableaux et utiliser des expressions composées :

```
In[11]: (2 * x) == (x ** 2)
```

```
Out[11]: array([False, True, False, False,  
False], dtype=bool)
```

Ces opérateurs de comparaison sont vraiment implémentés sous forme d'ufuncs dans NumPy. Lorsque vous écrivez `x < 3`, NumPy utilise `np.less(x, 3)`. Voici les six opérateurs avec l'ufunc correspondante :

Opérateur Ufunc correspondante	
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code><</code>	<code>np.less</code>
<code><=</code>	<code>np.less_equal</code>
<code>></code>	<code>np.greater</code>
<code>>=</code>	<code>np.greater_equal</code>

Ces opérateurs fonctionnent sur des tableaux de n'importe quels type et forme. Voici un exemple pour un tableau à deux dimensions :

```
In[12]: rng = np.random.RandomState(0)  
x = rng.randint(10, size=(3, 4))  
x
```

```
Out[12]: array([[5, 0, 3, 3],  
 [7, 9, 3, 5],  
 [2, 4, 7, 6]])
```

```
In[13]: x < 6
```

```
Out[13]: array([[ True,  True,  True,  True],  
                 [False, False,  True,  True],  
                 [ True,  True, False, False]],  
                dtype=bool)
```

Le résultat est toujours un tableau booléen. NumPy prévoit différentes tournures pour travailler avec ce genre de tableau.

Traitement d'un tableau booléen

Un certain nombre d'opérations sont possibles dans un tableau booléen. En guise d'exemple, nous allons repartir du tableau nommé `x` dans l'état dans lequel nous venons de le laisser, avec ses deux dimensions :

```
In[14]: print(x)
```

```
[[5 0 3 3]  
 [7 9 3 5]  
 [2 4 7 6]]
```

Comptage des entrées

Vous comptez le nombre d'entrées valant `True` dans un tableau booléen avec `np.count_nonzero` :

```
In[15]: # combien de valeurs inférieures à 6?  
np.count_nonzero(x < 6)
```

```
Out[15]: 8
```

Le tableau contient huit valeurs inférieures à 6. Le même résultat s'obtient avec np.sum, mais dans ce cas, False correspond à 0 et True à 1 :

```
In[16]: np.sum(x < 6)
```

```
Out[16]: 8
```

L'utilisation de sum() offre l'avantage, comme d'autres fonctions d'agrégation de NumPy, de pouvoir travailler le long des lignes ou des colonnes :

```
In[17]: # valeurs inférieures à 6 dans chaque  
ligne ?  
np.sum(x < 6, axis=1)
```

```
Out[17]: array([4, 2, 2])
```

Nous obtenons ainsi le nombre de valeurs inférieures à 6 dans chaque ligne de la matrice.

Pour savoir rapidement si au moins une ou si toutes les valeurs sont vraies, il suffit d'utiliser respectivement np.any() et np.all() :

```
In[18]: # des valeurs supérieures à 8 ?  
        np.any(x > 8)
```

```
Out[18]: True
```

```
In[19]: # des valeurs négatives ?  
        np.any(x < 0)
```

```
Out[19]: False
```

```
In[20]: # toutes inférieures à 10 ?  
        np.all(x < 10)
```

```
Out[20]: True
```

```
In[21]: # toutes égales à 6 ?  
        np.all(x == 6)
```

```
Out[21]: False
```

Ces deux fonctions peuvent être appliquées à un axe en particulier :

```
In[22]: # toutes valeurs de chaque ligne  
inférieures à 8 ?  
        np.all(x < 8, axis=1)
```

```
Out[22]: array([ True, False, True], dtype=bool)
```

Dans l'exemple, les éléments de la première et la troisième ligne sont inférieurs à 8, mais pas ceux de la deuxième ligne.

Je termine par un petit avertissement : nous avons indiqué dans la section de ce chapitre sur les agrégations que le langage Python dispose en interne des fonctions nommées `sum()`, `any()` et `all()` dont la syntaxe est différente de celles des versions NumPy. En particulier, elles produiront des résultats imprévus ou échoueront sur des tableaux multidimensionnels. Assurez-vous de bien utiliser `np.sum()`, `np.any()` et `np.all()` dans ces exemples !

Opérateurs booléens

Nous avons vu comment compter le nombre de jours ayant reçu moins de dix centimètres de pluie ou tous les jours en ayant reçu plus de cinq. Comment savoir quel jour il y a eu une précipitation supérieure à deux centimètres mais inférieure à dix ? Nous utilisons à cet effet les quatre opérateurs logiques binaires de Python `&`, `|`, `^` et `~`. Comme dans le cas des opérateurs arithmétiques, NumPy les redéfinit sous forme d'ufuncs travaillant élément par élément dans un tableau en général booléen.

Voici comment exprimer la recherche des jours avec une précipitation entre deux extrêmes :

```
In[23]: np.sum((pouces > 0.5) & (pouces < 1))
```

```
Out[23]: 29
```

Il y a donc eu 29 jours pendant lesquels il a plu entre 0,5 et 1 pouce.

Les parenthèses sont indispensables car elles permettent de contrôler l'ordre d'évaluation des différents opérateurs. En ne conservant qu'un jeu de parenthèses, comme dans l'exemple suivant, nous obtenons une erreur :

```
pouces > (0.5 & pouces) < 1
```

Si vous avez suivi un cours élémentaire de logique, vous savez que A **ET** B équivaut à **NON** (A **OU** B). Nous pouvons donc obtenir le même résultat exprimé d'une autre façon :

```
In[24]: np.sum(-(pouces <= 0.5) | (pouces >= 1))
```

```
Out[24]: 29
```

En combinant les opérateurs de comparaison et les opérateurs booléens, on peut rédiger un grand nombre d'opérations logiques de façon efficace.

Le tableau suivant présente les opérateurs booléens sur bit avec l'ufunc équivalente :

Opérateur Ufuncs équivalentes

&	np.bitwise_and
---	----------------

	np.bitwise_or
--	---------------

^	np.bitwise_xor
-	np.bitwise_not

Nous disposons maintenant d'un assez grand nombre d'outils pour répondre à toutes sortes de questions au sujet de la pluie. Voici quelques résultats obtenus en combinant le masquage avec l'agrégation :

```
In[25]: print("Jours sans pluie : ",  
np.sum(pouces == 0))  
        print("Jours pluvieux : ", np.sum(pouces  
!= 0))  
        print("Pluie de + de 0.5 pouces : ",  
np.sum(pouces > 0.5))  
        print("Pluie de moins de 0.1 pouces : ",  
np.sum((pouces > 0) & (pouces < 0.1)))
```

```
Jours sans pluie : 215  
Jours pluvieux : 150  
Pluie de plus de 0.5 pouces : 37  
Pluie de moins de 0.1 pouces : 46
```

Tableaux booléens et masques

Nous venons de voir comment calculer des agrégats à partir d'un tableau booléen. Une technique encore plus puissante consiste à se servir d'un tableau booléen comme masque

afin de sélectionner des sous-ensembles de données.
Reprenons le tableau x précédent pour nos exemples :

In[26]: x

Out[26]: array([[5, 0, 3, 3],
[7, 9, 3, 5],
[2, 4, 7, 6]])

Supposons que nous voulions obtenir un tableau contenant toutes les valeurs inférieures à 5 :

In[27]: $x < 5$

Out[27]: array([[False, True, True, True],
[False, False, True, False],
[True, True, False, False]],
dtype=bool)

Pour pouvoir sélectionner ces valeurs dans le tableau, nous faisons une indexation par rapport à ce tableau booléen, ce qui est l'opération de masquage :

In[28]: $x[x < 5]$

Out[28]: array([0, 3, 3, 3, 2, 4])

Le résultat est un tableau à une dimension contenant les valeurs qui satisfont à la condition, c'est-à-dire celles pour

lesquelles le masque contient au même endroit la valeur True.

D'autres traitements sont ensuite réalisables sur ces valeurs. Cela nous permet par exemple d'obtenir quelques statistiques intéressantes concernant la pluviométrie à Seattle :

```
In[29]: pluvieux = (pouces > 0)
         # construction d'un masque des jours
pluvieux (pouces > 0)
         # construction d'un masque des jours
ensoleillés (21 Juin = j. 172)
jours = np.arange(365)
estival = (jours > 172) & (jours < 262)

         print("Médiane des pluies des jours
pluvieux de 2014 (pouces): ",
np.median(pouces[pluvieux]))
         print("Médiane des pluies estivales 2014
(pouces):           ",
np.median(pouces[estival]))
         print("Maximale des pluies estivales 2014
(pouces):           ",
np.max(pouces[estival]))
         print("Médiane des pluies des jours
pluvieux hors été (pouces):",
np.median(pouces[pluvieux & ~estival]))
```

Médiane des pluies des jours pluvieux de 2014

(pouces): 0.19488188976377951

Médiane des pluies estivales 2014 (pouces):

0.0

Maximale des pluies estivales 2014 (pouces):

0.8503937007874016

Médiane des pluies des jours pluvieux hors été

(pouces): 0.20078740157480

Ce genre de question devient facile à traiter en combinant les opérations booléennes, les masques et les agrégats.

Mots-clés and/or ou bien opérateurs & | ?

Souvent, une confusion apparaît au moment de bien distinguer l'effet des mots-clés and et or par rapport aux opérateurs apparentés & et |. Quand faut-il utiliser les premiers ou les deuxièmes ?

La différence est simple : and et or testent la véracité d'un objet entier alors que & et | s'intéressent aux bits constituant l'objet.

Avec and et or, c'est comme si vous demandiez à Python de considérer l'objet comme une entité

booléenne. Toute valeur entière différente de zéro est évaluée à True :

```
In[30]: bool(42), bool(0)  
Out[30]: (True, False)
```

```
In[31]: bool(42 and 0)  
Out[31]: False
```

```
In[32]: bool(42 or 0)  
Out[32]: True
```

Si vous appliquez l'un des opérateurs & et | sur des entiers, ce sont les bits individuels de la ou des valeurs qui sont comparés avec un opérateur and ou or :

```
In[33]: bin(42)      # Pour pouvoir  
comprendre  
Out[33]: '0b101010'
```

```
In[34]: bin(59)      # Pour pouvoir  
comprendre  
Out[34]: '0b111011'
```

```
In[35]: bin(42 & 59)  
Out[35]: '0b101010'
```

```
In[36]: bin(42 | 59)
Out[36]: '0b111011'
```

Vous remarquez que les bits de la représentation binaire sont comparés dans l'ordre pour produire le résultat.

Un tableau de valeurs booléennes de NumPy peut être considéré comme une chaîne de bits, 1 valant True et 0 valant False. Les deux opérateurs & et | fonctionnent de la même façon qu'auparavant :

```
In[37]: A = np.array([1, 0, 1, 0, 1, 0],
dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1],
dtype=bool)
A | B

array([ True,  True,  True, False,  True,
True])
```

Si vous utilisez l'opérateur or sur ce tableau, vous demandez à savoir si l'objet tableau en entier est vrai ou pas, ce qui ne peut pas avoir de valeur bien définie :

```
In[38]: A or B
```

```
ValueError                                Traceback (most
recent call last)
<ipython-input-38-5d8e4f2e21c0> in <module>
()
----> 1 A or B
```

```
ValueError: The truth value of an array
with more than one element is
ambiguous. Use a.any() or a.all()
```

Quand vous appliquez une expression booléenne à un tableau, vous devez donc utiliser & et | au lieu de and et or :

```
In[39]: x = np.arange(10)
(x > 4) & (x < 8)
```

```
Out[39]: array([False, False, ..., True,
True, False, False], dtype=bool)
```

Si vous tentez d'évaluer la véracité du tableau complet, vous aurez la même erreur ValueError que plus haut :

```
In[40]: (x > 4) and (x < 8)
```

```
ValueError                                Traceback (most
recent call last)
<ipython-input-40-3d24f1ffd63d> in <module>
()
----> 1 (x > 4) and (x < 8)
```

```
ValueError: The truth value of an array
with more than one element is
ambiguous. Use a.any() or a.all()
```

Pour résumer : and et or réalisent une évaluation booléenne sur l'objet complet alors que & et | réalisent plusieurs évaluations booléennes sur les bits ou les octets qui constituent cet objet. Dans le cas des opérations sur les tableaux NumPy booléens, c'est en général cette seconde version qui est désirée.

2.7 : Indexation fancy

Nous avons vu dans les sections précédentes comment accéder à et intervenir sur une partie d'un tableau au moyen des index (`arr[0]`), des tranches (`arr[: 5]`) et des masques booléens (`arr[arr > 0]`). Découvrons maintenant un autre style d'indexation de tableau, appelée l'indexation *fancy*. Elle ressemble à l'indexation simple déjà vue, sauf que l'on fournit non pas des valeurs uniques scalaires, mais des tableaux d'index. Il devient ainsi possible de modifier aisément des sous-ensembles complexes de valeurs d'un tableau.

Découverte de l'indexation fancy

Le principe de cette indexation est fort simple : il s'agit de transmettre un tableau d'index pour accéder en une fois à plusieurs éléments d'un tableau. Partons du tableau suivant :

```
In[1]: import numpy as np  
rand = np.random.RandomState(42)  
x = rand.randint(100, size=10)  
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

Pour accéder à trois éléments, nous pouvons opérer ainsi :

```
In[2]: [x[3], x[7], x[2]]
```

```
Out[2]: [71, 86, 14]
```

Mais nous obtenons le même résultat en transmettant une liste ou un tableau d'index :

```
In[3]: ind = [3, 7, 4]
        x[ind]
```

```
Out[3]: array([71, 86, 60])
```

Dans l'indexation fancy, la forme du résultat reflète celle du tableau d'index et non celle du tableau traité :

```
In[4]: ind = np.array([[3, 7],
                      [4, 5]])
        x[ind]
```

```
Out[4]: array([[71, 86],
                  [60, 20]])
```

Cette indexation peut s'appliquer à plusieurs dimensions, par exemple :

```
In[5]: x = np.arange(12).reshape((3, 4))
        x
```

```
Out[5]: array([[ 0,  1,  2,  3],  
               [ 4,  5,  6,  7],  
               [ 8,  9, 10, 11]])
```

Comme avec l'indexation standard, le premier index désigne la ligne et le second la colonne :

```
In[6]: lig = np.array([0, 1, 2])  
        col = np.array([2, 1, 3])  
        X[lig, col]
```

```
Out[6]: array([ 2,  5, 11])
```

Vous remarquez que la première valeur du résultat vaut $X[0, 2]$, la deuxième vaut $X[1, 1]$ et la troisième $X[2, 3]$. La création de paires suit les mêmes règles de diffusion que celles présentées dans la section sur la diffusion de ce chapitre. Si nous combinons par exemple un vecteur de colonnes et un vecteur de lignes au moyen des index, nous obtenons un résultat en deux dimensions :

```
In[7]: X[lig[:, np.newaxis], col]
```

```
Out[7]: array([[ 2,  1,  3],  
               [ 6,  5,  7],  
               [10,  9, 11]])
```

Chacune des valeurs de ligne est confrontée à chaque vecteur de colonne, comme lors de la diffusion avec les opérations

arithmétiques. Un exemple :

```
In[8]: lig[:, np.newaxis] * col
```

```
Out[8]: array([[0, 0, 0],  
               [2, 1, 3],  
               [4, 2, 6]])
```

Dans le cas de l'indexation fancy, il ne faut jamais oublier que la valeur renvoyée prend la forme diffusée des index et non celle du tableau à indicer.

Indexation combinée

Des traitements plus puissants sont possibles en combinant l'indexation fancy et un autre mécanisme d'indexation. Voici nos données initiales :

```
In[9]: print(X)
```

```
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]]
```

Voici comment combiner indexation fancy et indexation simple :

```
In[10]: X[2, [2, 0, 1]]
```

```
Out[10]: array([10, 8, 9])
```

De même, on peut combiner l'indexation fancy avec le tranchage:

```
In[11]: X[1:, [2, 0, 1]]
```

```
Out[11]: array([[ 6, 4, 5],  
                 [10, 8, 9]])
```

Enfin, nous combinons l'indexation fancy avec un masque :

```
In[12]: masque = np.array([1, 0, 1, 0],  
                         dtype=bool)  
         X[lig[:, np.newaxis], masque]
```

```
Out[12]: array([[ 0, 2],  
                  [ 4, 6],  
                  [ 8, 10]])
```

Ces différentes approches d'indexation procurent une vaste gamme d'opérations très souples pour accéder et modifier les valeurs des tableaux.

Exemple : sélection de points aléatoires

L'indexation fancy est souvent utilisée pour sélectionner un sous-ensemble de lignes à partir d'une matrice. Partons

d'une matrice de N par D qui représente N points dans D dimensions. Les points suivants sont tirés d'une distribution normale en deux dimensions :

```
In[13]: mean = [0, 0]
          cov = [[1, 2],
                  [2, 5]]
          X = rand.multivariate_normal(mean, cov,
                                         100)
          X.shape
```

```
Out[13]: (100, 2)
```

Au moyen des outils de visualisation décrits dans le [Chapitre 4](#), nous pouvons tracer ces points sous forme d'un nuage ([Figure 2.7](#)) :

```
In[14]: %matplotlib inline
          import matplotlib.pyplot as plt
          import seaborn; seaborn.set() # style de
          tracé

          plt.scatter(X[:, 0], X[:, 1]);
```

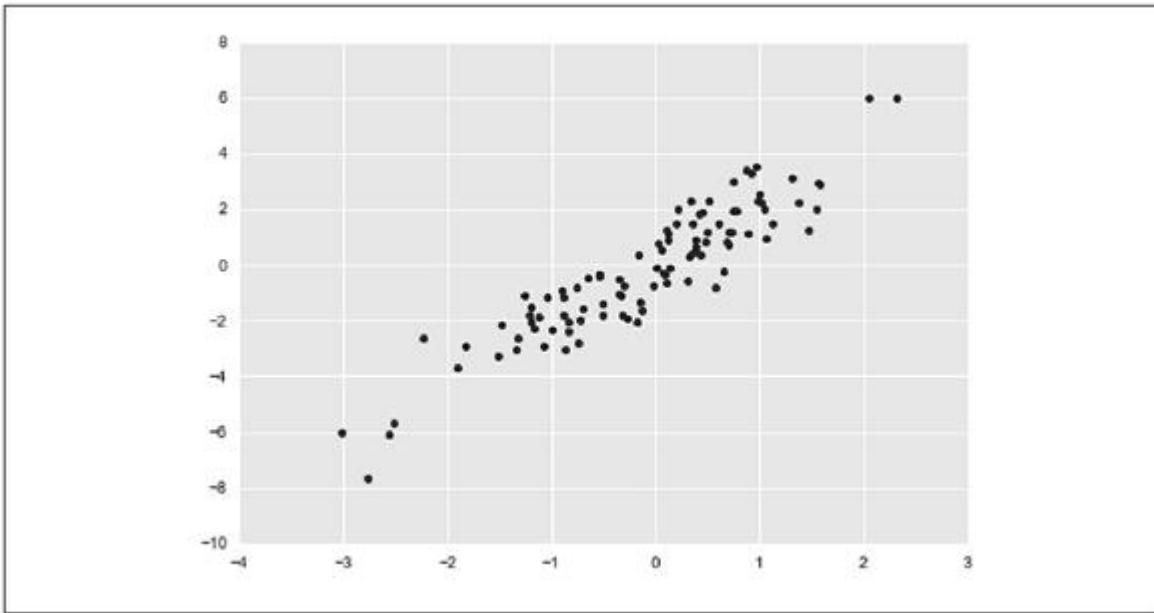


Figure 2.7 : Nuage de points en distribution normale.

Voyons comment sélectionner 20 points au hasard grâce à l'indexation fancy. Nous commençons par choisir 20 index au hasard sans répétition puis nous les appliquons pour sélectionner une partie du tableau de départ :

In[15]:

```
indices = np.random.choice(X.shape[0], 20,  
replace=False)  
indices
```

Out[15]: array([93, 45, 73, 81, 50, 10, 98, 94,
4, 64, 65, 89, 47, 84, 82,
80, 25, 90, 63, 20])

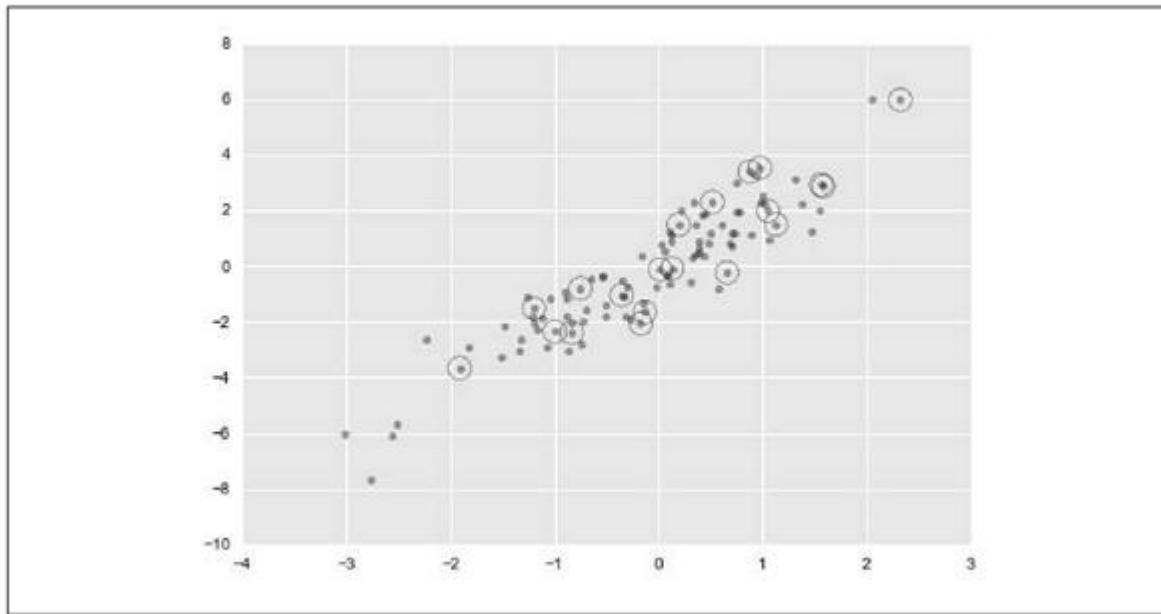
In[16]: selection = X[indices] # fancy indexing
here

```
selection.shape
```

```
Out[16]: (20, 2)
```

Pour rendre visible les points sélectionnés, nous ajoutons des cercles ([Figure 2.8](#)) :

```
In[17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
            plt.scatter(selection[:, 0], selection[:, 1],
                        facecolor='none', s=200);
```



[Figure 2.8](#) : Une sélection aléatoire parmi des points.

Cette technique est souvent adoptée pour partitionner rapidement un jeu de données. C'est une opération fréquemment requise pour distribuer les données entre jeux d'entraînement et jeux de test afin de valider un modèle

statistique (voyez la section sur les hyperparamètres dans le [Chapitre 5](#)) ainsi que dans les échantillonnages visant à répondre à des problèmes statistiques.

Modification de valeurs par indexation fancy

L'indexation fancy permet non seulement d'accéder aux éléments d'un tableau, mais également de les modifier. Partons d'un tableau d'index avec lequel nous voudrions donner une certaine valeur aux éléments correspondants d'un tableau :

```
In[18]: x = np.arange(10)
         i = np.array([2, 1, 8, 4])
         x[i] = 99
         print(x)

[ 0 99 99 3 99 5 6 7 99 9]
```

Tous les opérateurs d'affectation sont utilisables :

```
In[19]: x[i] -= 10
         print(x)

[ 0 89 89 3 89 5 6 7 89 9]
```

Notez que dans ces opérations une répétition d'index peut donner des résultats inattendus. Étudiez l'exemple suivant :

```
In[20]: x = np.zeros(10)
         x[[0, 0]] = [4, 6]
         print(x)
```

```
[ 6.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Où est partie la valeur 4 ? Cette opération affecte d'abord $x[0] = 4$ puis $x[0] = 6$. Voilà pourquoi $x[0]$ contient enfin la valeur 6.

Rien de plus normal, mais voyons cette autre opération :

```
In[21]:
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x
```

```
Out[21]: array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.])
```

Vous pouvez vous attendre à ce que $x[3]$ contienne 2, et que $x[4]$ contienne 3, puisque cela correspond au nombre de répétitions de chaque index. Pourquoi n'en est-il pas ainsi ? Il faut savoir que $x[i] += 1$ est une écriture abrégée de $x[i] = x[i] + 1$. L'augmentation $x[i] + 1$ est réalisée en premier puis le résultat est affecté aux index dans x . Ce n'est donc pas

l'augmentation qui est répétée, mais l'affectation, ce qui explique le résultat un peu perturbant.

Comment faire pour jouir de l'autre comportement en cas de répétitions ? Il suffit d'utiliser la méthode `at()` des fonctions universelles ainsi :

```
In[22]: x = np.zeros(10)
         np.add.at(x, i, 1)
         print(x)
```

```
[ 0.  0.  1.  2.  3.  0.  0.  0.  0.]
```

La méthode `at()` applique l'opérateur demandé sur place avec les index spécifiés, i dans l'exemple et la valeur spécifiée, 1. Une variante correspond à `reduceat()` dont vous trouverez les détails dans la documentation de NumPy.

Exemple : répartition de données (binning)

Ces techniques permettent par exemple de créer un histogramme manuellement en répartissant les données dans des bacs (opération de *binning*). Supposons 1 000 valeurs pour lesquelles nous voulons rapidement savoir comment elles se distribuent dans un tableau de bacs (*bins*). Nous pouvons faire le calcul au moyen de l'ufunc `at` comme ceci :

```
In[23]: np.random.seed(42)
         x = np.random.randn(100)

# Calcul manuel de l'histogramme
bacs = np.linspace(-5, 5, 20)
nombre = np.zeros_like(bacs)

# Trouver le bac approprié à chaque valeur x
i = np.searchsorted(bacs, x)

# Ajouter 1 à chaque bac
np.add.at(nombre, i, 1)
```

Le tableau *nombre* contient le nombre de points dans chaque bac, ce qui correspond à un histogramme ([Figure 2.9](#)) :

```
In[24]: # Tracer les résultats
         plt.plot(bacs, nombre,
drawstyle='steps');
```

Il serait peu pratique de devoir relancer ce genre de calcul avant chaque tracé d'un histogramme. Le paquetage Matplotlib fournit la fonction `plt.hist()` qui permet d'obtenir le même résultat en une seule ligne :

```
plt.hist(x, bacs, histtype='step');
```

Le diagramme que crée cette fonction est très similaire au précédent. Pour réaliser la répartition, Matplotlib se sert de

la fonction np.histogram() dont le traitement est proche de celui que nous avons fait manuellement. Comparons leurs performances :

```
In[25]: print("Routine NumPy :")
          %timeit nombre, edges = np.histogram(x,
bacs)
          print("Routine sur mesure :")
          %timeit np.add.at(nombre,
np.searchsorted(bacs, x), 1)
Routine NumPy :
10000 loops, best of 3: 97.6 µs per loop
Routine sur mesure :
10000 loops, best of 3: 19.5 µs per loop
```

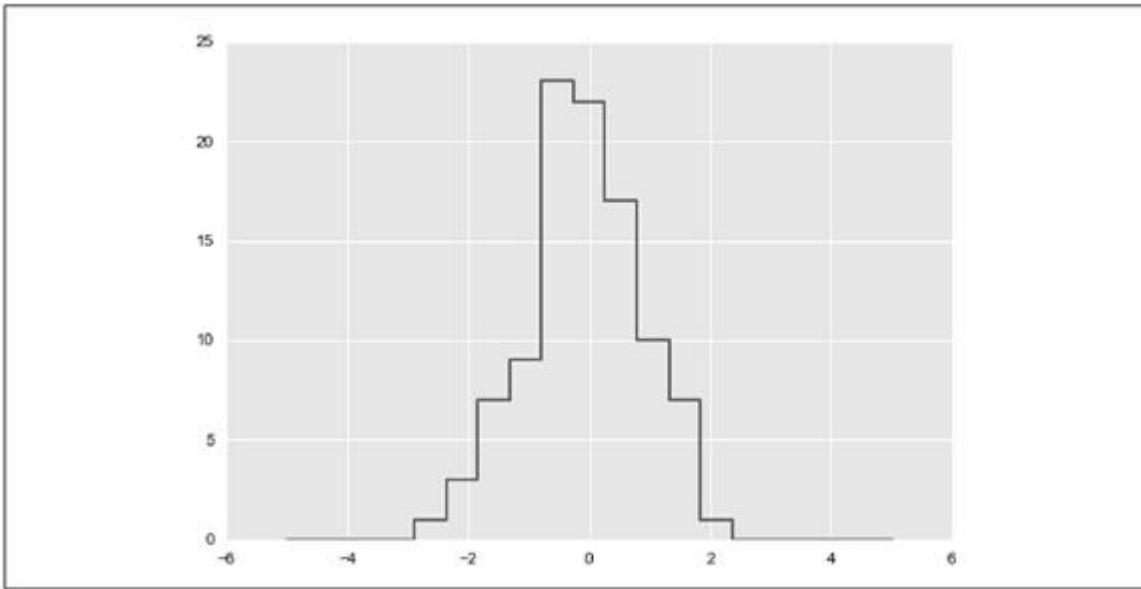


Figure 2.9 : Histogramme calculé manuellement.

Notre nouvel algorithme sur une ligne est bien plus rapide que l'algorithme optimisé de NumPy ! Comment est-ce possible ? En allant lire le code source de np.histogram (en tapant dans IPython np.histogram??), on peut voir que le traitement est un peu plus sophistiqué et ne se limite pas à une recherche et à un comptage. L'algorithme de NumPy est beaucoup plus souple et offre de meilleures performances dès que le nombre de points de données devient vraiment important :

```
In[26]: x = np.random.randn(1000000)
print("Routine NumPy :")
%timeit nombre, edges = np.histogram(x, bacs)
print("Routine sur mesure :")
%timeit np.add.at(nombre, np.searchsorted(bacs,
```

`x), 1)`

Routine NumPy :

```
10 loops, best of 3: 68.7 ms per loop
```

Routine sur mesure :

```
10 loops, best of 3: 135 ms per loop
```

On peut en conclure que l'efficacité des algorithmes n'est pas un sujet simple. Un algorithme qui se montre efficace pour un gros volume de données ne le sera sans doute pas pour un petit volume, et *vice versa* (voyez l'encadré sur la notation un peu plus loin). Il reste intéressant de coder l'algorithme vous-même, car vous acquerrez ainsi la maîtrise des méthodes élémentaires, et pourrez ensuite vous appuyer sur ce savoir pour créer des comportements spécifiques très intéressants. Une utilisation efficace de Python dans les applications de traitement lourd de données suppose de connaître les routines générales telles que `np.histogram()`, et de connaître les contextes dans lesquels elles sont appropriées, tout en sachant comment accéder aux fonctions à bas niveau lorsque le besoin s'en fait sentir.

2.8 : Tri de tableaux

Nous venons de voir comment accéder et modifier les données d'un tableau NumPy. Voyons maintenant comment trier ce genre de tableau. Les algorithmes de tri constituent un sujet de prédilection dans les premières années d'apprentissage de l'informatique. Si vous êtes passé par là, vous avez sans doute rêvé ou bien eu des cauchemars lorsque vous avez découvert les tris par insertion, par sélection, par fusion, les tris rapides, les tris à bulles, et bien d'autres. Toutes ces techniques cherchent le même objectif : trier les valeurs trouvées dans une liste ou un tableau.

Un simple tri par sélection cherche la plus petite valeur d'une liste et l'intervertit avec sa voisine jusqu'à aboutir à une liste triée. En Python, il suffit de calculer une ligne pour coder l'opération :

```
In[1]:  
import numpy as np  
  
def selection_sort(x):  
    for i in range(len(x)):  
        swap = i + np.argmin(x[i:])  
        (x[i], x[swap]) = (x[swap], x[i])  
    return x
```

```
In[2]:
```

```
x = np.array([2, 1, 4, 3, 5])
selection_sort(x)
```

```
Out[2]: array([1, 2, 3, 4, 5])
```

Tout étudiant en première année d'informatique le sait : le tri par sélection est simple, mais bien trop lent lorsque les tableaux sont grands. Pour une liste de N valeurs, il faut faire v tours de boucle, chaque tour réalisant de l'ordre de N comparaisons. En notation « Grand-O », qui sert à caractériser un algorithme, (voyez l'encadré sur la notation « Grand-O » plus loin), le tri par sélection demande en moyenne $O[N^2]$. Si vous doublez le nombre d'éléments dans la liste, la durée d'exécution va être multipliée par quatre.

Ceci dit, le tri par sélection reste bien meilleur que mon algorithme de tri favori, le bogosort :

```
In[3]:
```

```
def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x
```

```
In[4]:
```

```
x = np.array([2, 1, 4, 3, 5])
bogosort(x)
```

```
Out[4]: array([1, 2, 3, 4, 5])
```

Cette méthode de tri un peu grossière se fonde sur la chance en appliquant de façon répétée une répartition au hasard du contenu du tableau jusqu'à aboutir à un tableau trié. Avec une notation Grand-O $[N \times N!]$ (c'est-à-dire N fois la factorielle de N), il est évident que vous n'utiliserez jamais ce tri pour des travaux réels.

Python propose heureusement d'autres algorithmes de tri beaucoup plus efficaces. Commençons par ceux qui sont intégrés à Python. Nous verrons ensuite les routines proposées par NumPy, qui sont optimisées pour les tableaux NumPy.

Tri rapide dans NumPy avec `np.sort` et `np.argsort`

Python propose en standard les deux fonctions `sort` et `sorted` pour les listes, mais nous ne les décrirons pas ici. En effet, la fonction `np.sort` de NumPy est beaucoup plus efficace et adaptée à nos besoins. Par défaut, ses performances correspondent à $O[N \log N]$, avec un algorithme de tri rapide *quicksort*, mais les deux tris *mergesort* et *heapsort* sont également disponibles. Dans la plupart des cas, le tri *quicksort* par défaut suffira.

Pour obtenir une version triée d'un tableau sans modifier les données d'entrées, vous utilisez np.sort :

```
In[5]:  
x = np.array([2, 1, 4, 3, 5])  
np.sort(x)
```

```
Out[5]: array([1, 2, 3, 4, 5])
```

Si vous acceptez de trier le tableau d'entrées, vous utilisez la méthode sort du tableau :

```
In[6]:  
x.sort()  
print(x)
```

```
[1 2 3 4 5]
```

La variante argsort renvoie les index des éléments triés :

```
In[7]:  
x = np.array([2, 1, 4, 3, 5])  
i = np.argsort(x)  
print(i)
```

```
[1 0 3 2 4]
```

Le premier élément du résultat correspond à l'index du plus petit élément, et ainsi de suite. Vous pouvez ensuite utiliser

ces index avec une indexation fancy pour fabriquer le tableau trié :

```
In[8]: x[i]  
Out[8]: array([1, 2, 3, 4, 5])
```

Tri selon les lignes ou les colonnes

Les algorithmes de tri de NumPy offrent la possibilité de trier en fonction de certaines lignes ou colonnes dans un tableau multidimensionnel, grâce au paramètre axis :

```
In[9]:  
rand = np.random.RandomState(42)  
X = rand.randint(0, 10, (4, 6))  
print(X)
```

```
[[6 3 7 4 6 9]  
 [2 6 7 4 3 7]  
 [7 2 5 4 1 7]  
 [5 1 4 0 9 5]]
```

```
In[10]: # Tri chaque colonne de X  
np.sort(X, axis=0)
```

```
Out[10]:  
array([[2, 1, 4, 0, 1, 5],  
       [5, 2, 5, 4, 3, 7],  
       [6, 3, 7, 4, 6, 7],  
       [7, 6, 7, 4, 9, 9]])
```

```
In[11]: # Trie chaque ligne de X  
np.sort(X, axis=1)
```

```
Out[11]:  
array([[3, 4, 6, 6, 7, 9],  
       [2, 3, 4, 6, 7, 7],  
       [1, 2, 4, 5, 7, 7],  
       [0, 1, 4, 5, 5, 9]])
```

Vous devez savoir que cette opération considère chaque ligne ou colonne en tant que tableau indépendant. Les relations éventuelles entre lignes et colonnes sont perdues !

Tri partiel et partitionnement

Parfois, vous n'avez pas besoin de trier tout le tableau, mais simplement de découvrir quels sont les K plus petites valeurs. Vous pouvez utiliser la fonction np.partition qui attend un tableau en entrée ainsi qu'un nombre K. Elle produit un autre tableau contenant les K valeurs les plus petites du côté gauche de la partition et toutes les autres valeurs du côté droit, dans un ordre arbitraire :

```
In[12]:  
x = np.array([7, 2, 3, 1, 6, 5, 4])  
np.partition(x, 3)
```

```
Out[12]: array([2, 1, 3, 4, 6, 5, 7])
```

Vous constatez que les trois premières valeurs résultantes sont bien les trois valeurs les plus petites du tableau, les autres valeurs apparaissant à la suite. Les éléments ne sont triés dans aucune des deux partitions.

Comme pour le tri, nous pouvons demander un partitionnement selon un axe choisi d'un tableau multidimensionnel :

In[13]:

```
np.partition(X, 2, axis=1)
```

Out[13]:

```
array([[3, 4, 6, 7, 6, 9],  
       [2, 3, 4, 7, 6, 7],  
       [1, 2, 4, 5, 7, 7],  
       [0, 1, 4, 5, 9, 5]])
```

Nous obtenons un tableau dans lequel les deux premières cases de chaque ligne contiennent les plus petites valeurs pour cette ligne, les autres valeurs prenant place dans les cases suivantes.

Vous disposez ici aussi d'une variante qui porte le nom de np.argpartition qui permet d'obtenir les index de la partition (tout comme np.argsort le propose pour les index de tri). Nous allons l'utiliser dans la section qui suit.

Un exemple : les k plus proches voisins

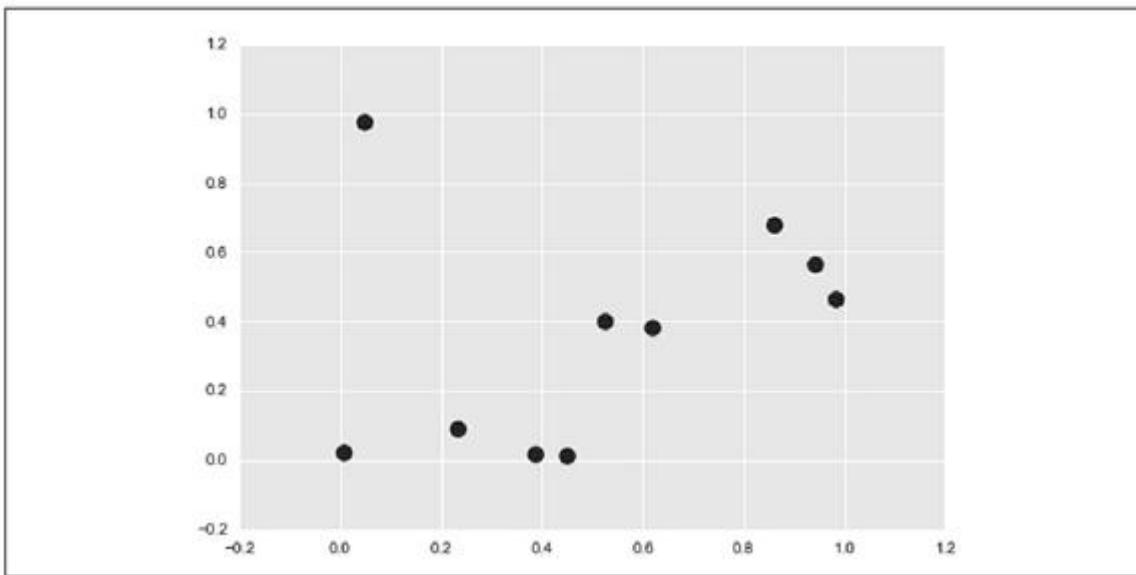
Servons-nous de la fonction argsort pour plusieurs axes afin de trouver les plus proches voisins de chaque point d'un ensemble. Nous commençons par créer un ensemble au hasard de 10 points en deux dimensions. Nous nous plions à la convention standard et choisissons une forme de tableau de 10×2 :

```
In[14]: X = rand.rand(10, 2)
```

Cherchons à voir à quoi ressemble cette répartition au moyen d'un tracé ([Figure 2.10](#)) :

```
In[15]:
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```



[Figure 2.10](#) : Visualisation des points d'un exemple de k-voisins.

Nous pouvons maintenant calculer la distance entre chaque paire de points. Je rappelle que le carré de la distance entre deux points correspond à la somme des carrés des différences dans chaque dimension. Nous pouvons générer la matrice des carrés des distances avec une seule ligne de code en nous servant des mécanismes de diffusion et d'agrégation vus dans ce chapitre :

In[16]:

```
dist_sq = np.sum((X[:,np.newaxis,:] -
                  X[np.newaxis,:,:,:]) ** 2, axis=-1)
```

Cette instruction est assez touffue et sa compréhension n'est pas évidente si vous ne connaissez pas les règles de

diffusion de NumPy. Il est donc intéressant de subdiviser ce traitement en plusieurs étapes :

```
In[17]: # Calculer les différences de
coordonnées pour chaque paire
differences = X[:, np.newaxis, :] -
X[np.newaxis, :, :]
differences.shape
```

```
Out[17]: (10, 10, 2)
```

```
In[18]: # Élever aux carrés les différences
q_differences = differences ** 2
sq_differences.shape
```

```
Out[18]: (10, 10, 2)
```

```
In[19]:
# Additionner les différences pour obtenir les
carrés de distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
```

```
Out[19]: (10, 10)
```

Pour garantir que nous sommes sur la bonne voie, nous devons pouvoir constater que la diagonale de la matrice, c'est-à-dire l'ensemble des distances entre chaque point et lui-même vaut zéro :

```
In[20]: dist_sq.diagonal()  
Out[20]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  
 0.,  0.])
```

Effectivement ! Après conversion des carrés de distance des paires, nous pouvons appliquer np.argsort pour lancer un tri selon les lignes. Les colonnes les plus à gauche contiendront ensuite les index des plus proches voisins :

```
In[21]:  
nearest = np.argsort(dist_sq, axis=1)  
print(nearest)
```

```
[[0 3 9 7 1 4 2 5 6 8]  
 [1 4 7 9 3 6 8 5 0 2]  
 [2 1 4 6 3 0 8 9 7 5]  
 [3 9 7 0 1 4 5 8 6 2]  
 [4 1 8 5 6 7 9 3 0 2]  
 [5 8 6 4 1 7 9 3 2 0]  
 [6 8 5 4 1 7 9 3 2 0]  
 [7 9 3 1 4 0 5 8 6 2]  
 [8 5 6 4 1 7 9 3 2 0]  
 [9 7 3 0 1 4 5 8 6 2]]
```

Vous constatez que dans la première colonne nous trouvons les valeurs entre 0 et 9 dans l'ordre. C'est normal, puisque le plus proche voisin de chaque point est lui-même.

Nous venons de faire un tri complet, ce qui est plus que ce que nous avions besoin. Si nous nous étions seulement intéressés aux K plus proches voisins, il aurait suffi de partitionner chaque ligne pour que les K + 1 plus petits carrés de distance arrivent en premier, les distances supérieures prenant place dans les autres positions du tableau. C'est réalisable avec la fonction

`np.argpartition()` :

```
In[22]:  
K = 2  
nearest_partition = np.argpartition(dist_sq, K +  
1, axis=1)
```

Visualisons ce réseau de voisins avec des lignes représentant les connexions entre chaque point et ses deux proches voisins ([Figure 2.11](#)) :

```
In[23]:  
plt.scatter(X[:, 0], X[:, 1], s=100)  
# Tracer de lignes entre un point et ces deux  
voisins  
K = 2  
  
for i in range(X.shape[0]):  
    for j in nearest_partition[i, :K+1]:  
        # Tracer d'une ligne entre X[i] et X[j]
```

```
# Utilisation d'une magie avec zip :  
plt.plot(*zip(X[j], X[i]), color='black')
```

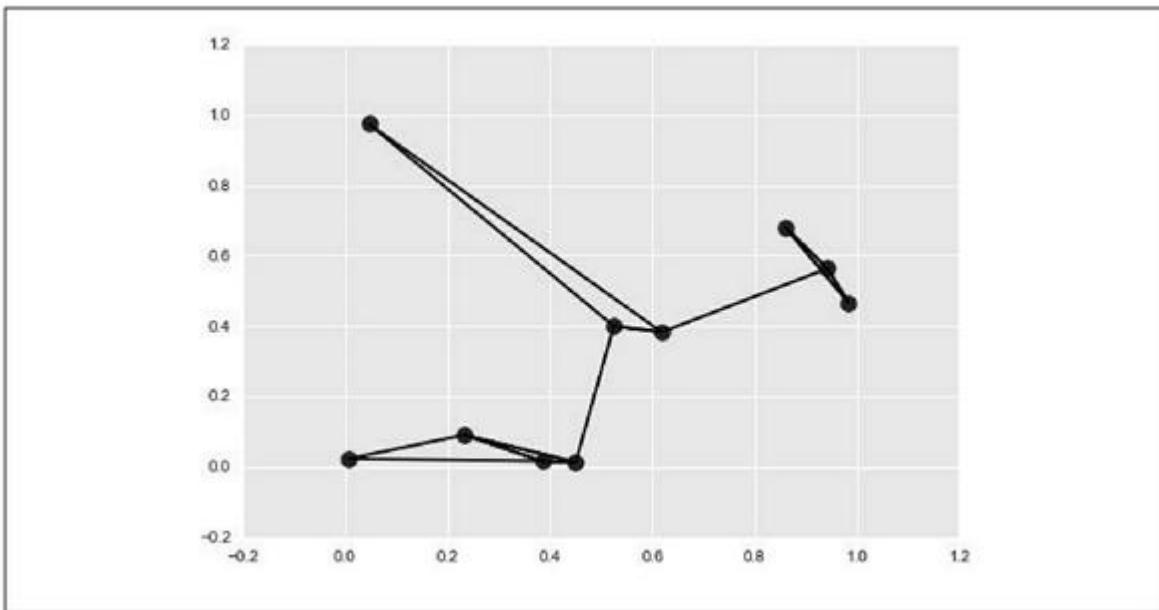


Figure 2.11 : Visualisation des plus proches voisins de chaque point.

Au premier regard, on pourrait croire que certains points sont reliés à plus de deux autres. Il suffit de se souvenir que lorsque le point A est l'un des deux plus proches voisins du point B, cela n'implique pas que le point B soit l'un des deux plus proches voisins du point A.

Cette approche consistant à utiliser la diffusion et le tri par ligne pourrait sembler moins simple que l'écriture d'une boucle, mais il s'avère que c'est une solution très efficace pour traiter ce genre de données dans Python. Vous pouvez être tenté d'obtenir le même résultat avec une boucle de répétition explicite en triant chaque couple de voisins

individuellement, mais il est probable que le traitement sera moins rapide qu'avec cette version vectorisée. L'autre intérêt de l'approche est que la façon dont elle est formulée n'est affectée ni par la taille, ni par le volume des données d'entrées. Vous pouvez aussi facilement trouver les voisins parmi 100 que parmi 1 000 000 de points, quel que soit le nombre de dimensions. Le code source sera le même.

Notons pour conclure que lorsque la recherche concerne un très grand nombre de voisins, il est possible d'utiliser des algorithmes à arborescence ou à approximation dont les performances sont de niveau $O[N \log N]$ ou encore meilleures, ce qui est préférable à $O[N^2]$ qui caractérise l'algorithme à force brute. C'est par exemple le cas de l'arbre KD-Tree, disponible dans Scikit-Learn (<http://bit.ly/2fSpdxI>).

Notation Grand-O (comparaison asymptotique)

La notation Grand-O permet de savoir de quelle façon le nombre d'opérations à réaliser par un algorithme évolue en fonction de l'augmentation de taille des données à traiter. L'utilisation précise de cette mesure suppose de plonger dans la théorie informatique, et d'apprendre notamment à distinguer cette notation de

ses collègues, petit- α , grand- θ (thêta), grand- Ω et leurs hybrides et mutants. Ces nuances sont indispensables pour être précis dans les formulations algorithmiques, mais vous les verrez rarement à l'œuvre en dehors des cours de théorie informatique et des diplômes qui les sanctionnent (ainsi que sur certains blogs pédants). La notation Grand-O est utilisée de façon beaucoup plus pragmatique à vaste échelle : elle permet de façon générale (même si moins précise) de prévoir comment un algorithme va se comporter lors d'une montée en charge. C'est cette interprétation allégée que nous adoptons dans ce livre, en présentant nos excuses aux théoriciens.

Dans ce sens relâché, la notation Grand-O permet de savoir combien de temps votre algorithme va demander pour s'exécuter lorsque vous augmentez le volume de données. En supposant un algorithme $O[N]$ (se lit « ordre N ») qui réclame 1 seconde pour traiter une liste de longueur $N=1\ 000$, vous pouvez vous attendre à ce qu'il prenne 5 secondes pour une liste $N=5\ 000$. Si l'algorithme est caractérisé en ordre $[N^2]$, et qu'il demande 1 seconde pour 1 000, il devrait réclamer 25 secondes pour $N=5\ 000$.

La valeur N dénote un aspect du volume de données (le nombre de points, de dimensions, *etc.*). Lorsque le volume se situe dans les milliards de valeurs ou plus, la différence de temps entre ordre N et ordre N^2 devient vraiment significative !

Sachez que la notation Grand-O n'informe par elle-même pas au sujet du temps physique d'exécution, mais seulement au sujet des proportions de variation en fonction de la charge N. Il est considéré qu'en général un algorithme ordre N montera mieux en charge qu'un algorithme ordre N^2 , pour des raisons évidentes. En revanche, ce ne sera pas nécessairement le cas pour de petits volumes de données. Dans certains problèmes, l'algorithme $O[N^2]$ pourra demander 0,01 seconde alors que l'algorithme plus performant $O[N]$ prendra 1 seconde. Mais si N passe à 1 000, l'algorithme $O[N]$ va surperformer l'autre.

Cette version peu académique de la notation Grand-O va s'avérer très utile pour comparer les performances des algorithmes. Nous allons avoir l'occasion de nous

en servir à plusieurs reprises quand nous nous soucierons des performances.

2.9 : Données structurées : les tableaux structurés de NumPy

Les données d'entrée peuvent souvent être placées dans un tableau de valeurs homogènes, mais pas toujours. Voyons comment utiliser les tableaux structurés et les tableaux d'enregistrements de NumPy. Ils permettent de stocker des données hétérogènes et composites. Les techniques présentées ici conviennent aux opérations simples, mais les contextes d'utilisation appellent rapidement à adopter le type DataFrame de Pandas, qui est décrit en détail dans le prochain chapitre.

Supposons plusieurs catégories de données décrivant des individus, par exemple avec le nom, l'âge et la taille. Nous avons besoin de stocker ces valeurs pour les faire traiter par un programme Python. Il est bien sûr possible de créer trois tableaux distincts :

In[2]:

```
pnom = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
poids = [55.0, 85.5, 68.0, 61.5]
```

Ce n'est pas très pratique. Rien ne relie logiquement les trois tableaux. Il serait préférable d'avoir une seule structure, ce

que NumPy autorise en définissant un tableau structuré.

Nous avions créé un tableau un peu plus haut avec l'expression suivante :

```
In[3]: x = np.zeros(4, dtype=int)
```

Nous pouvons facilement créer un tableau structuré comme ceci :

```
In[4]: # Type composite pour tableau structuré
data = np.zeros(4, dtype={'pnoms':('pnom',
'age', 'poids'),
'formats':('U10', 'i4',
'f8')})
print(data.dtype)

[('pnom', '<U10'), ('age', '<i4'), ('poids',
'<f8')]
```

Dans l'exemple, la notation 'U10' signifie « Chaîne Unicode de longueur maximale 10 », 'i4' désigne un entier sur 4 octets (ou 32 bits) et 'f8' un flottant sur 8 octets ou 64 bits. Nous verrons les autres options un peu plus loin.

Une fois que le tableau récepteur vide est créé, nous pouvons le remplir avec nos listes de valeurs :

```
In[5]:
data['pnom'] = pnom
```

```
data['age'] = age
data['poids'] = poids
print(data)

[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy',
37, 68.0) ('Doug', 19, 61.5)]
```

Dorénavant, nos données sont rassemblées dans le même bloc mémoire.

Un des avantages des tableaux structurés est qu'il est dorénavant possible d'accéder aux éléments par leur nom et par leur index :

```
In[6]: # Lit tous les prénoms
data['pnom']
```

```
Out[6]:
array(['Alice', 'Bob', 'Cathy', 'Doug'],
dtype='|<U10')
```

```
In[7]: # Lit la première ligne de données
data[0]
```

```
Out[7]:
('Alice', 25, 55.0)
```

```
In[8]: # Lit le prénom de la dernière ligne
data[-1]['pnom']
```

```
Out[8]: 'Doug'
```

En profitant du masquage booléen, vous pouvez réaliser des opérations encore plus sophistiquées, par exemple un filtrage en fonction de l'âge :

```
In[9]: # Lire les noms de ceux de moins de 30
ans
data[data['age'] < 30]['pnom']
```

```
Out[9]:
array(['Alice', 'Doug'], dtype='<U10')
```

Si vous envisagez des opérations encore plus complexes, il sera préférable de vous tourner vers le paquetage Pandas décrit en détail dans le prochain chapitre. En effet, Pandas propose l'objet nommé DataFrame, qui est une vraie structure bâtie à partir de tableaux NumPy et offrant de nombreuses possibilités de manipulations de données, avec une richesse opérationnelle incomparable.

Création d'un type de tableau structuré

Vous disposez de plusieurs méthodes pour créer des types de données structurées. Nous avions déjà vu la méthode basée sur le dictionnaire :

In[25]:

```
np.dtype({'pnoms':('pnom', 'age', 'poids'),  
'formats':('U10', 'i4', 'f8')})
```

Out[25]:

```
dtype([('pnom', '<U10'), ('age', '<i4'),  
( 'poids', '<f8')])
```

Les types numériques peuvent être désignés avec les noms de type Python ou avec les noms dtype de NumPy :

In[26]:

```
np.dtype({'names':('pnom', 'age', 'poids'),  
'formats':((np.str_, 10), int,  
np.float32)})
```

Out[26]:

```
dtype([('pnom', '<U10'), ('age', '<i8'),  
( 'poids', '<f4')])
```

Pour un type composite, vous pouvez spécifier une liste de tuples :

In[27]:

```
np.dtype([('pnom', 'S10'), ('age', 'i4'),  
( 'poids', 'f8')])
```

Out[27]:

```
dtype([('pnom', 'S10'), ('age', '<i4'),  
( 'poids', '<f8')])
```

Si les noms des types n'ont pas d'importance pour vous, vous pouvez égrener les types dans une chaîne à séparateur virgule :

In[28]:

```
np.dtype('S10,i4,f8')
```

Out[28]:

```
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

Ces codes abrégés peuvent perturber, mais leurs noms suivent une systématique simple. Le premier caractère facultatif peut être le signe < ou >, pour choisir entre « petit boutisme » ou « grand boutisme » (*endian*), c'est-à-dire le choix du stockage des bits de poids fort au début ou la fin de la valeur binaire. Le caractère suivant correspond au type de données : caractère, octet, entier, flottant, etc. ([Tableau 2.4](#)). Le ou les derniers caractères définissent la taille en octets pour les objets de ce type.

[Tableau 2.4](#) : Types de données de NumPy.

Caractère	Description	Exemple
'b'	Octet (byte)	np.dtype('b')
'i'	Entier signé	np.dtype('i4') == np.int32
'u'	Entier non signé	np.dtype('u1') == np.uint8

'f'	Flottant	<code>np.dtype('f8') == np.int64</code>
'c'	Flottant complexe	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	Chaîne de caractères	<code>np.dtype('S5')</code>
'U'	Chaîne Unicode	<code>np.dtype('U') == np.str_</code>
'V'	Donnée brute (void)	<code>np.dtype('V') == np void</code>

Types composites sophistiqués

Vous pouvez définir des types composés encore plus complexes, par exemple un type pour lequel chaque élément héberge un tableau ou une matrice de valeurs. Créons par exemple le type qui contient un composant nommé mat qui est une matrice de valeurs flottantes de 3×3 :

In[29]:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0]])
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Chaque élément de notre nouveau tableau X réunit un identifiant *id* et la matrice de 3×3 . Pourquoi utiliser ce genre de structure au lieu d'un tableau à plusieurs dimensions ou même d'un dictionnaire Python ? Il faut savoir que ce type *dtype* NumPy est directement implanté sous forme d'une structure C. Le tampon mémoire qui contient le tableau peut être atteint directement par un programme C adéquat. Si vous avez besoin d'ajouter une interface en Python à une librairie de fonctions existantes en C ou en Fortran, et que celle-ci manipule des données structurées, vous trouverez ces tableaux structurés NumPy particulièrement utiles !

RecordArrays : tableaux structurés évolués

NumPy propose enfin la classe nommée `np.recarray` qui ressemble à un tableau structuré, en y ajoutant la possibilité d'accéder aux champs par des attributs et non par des clés de dictionnaire. Rappelons la façon dont nous avons accédé aux valeurs *age* un peu plus haut :

```
In[30]: data['age']
```

```
Out[30]: array([25, 45, 37, 19], dtype=int32)
```

Si nous pouvons considérer les données comme contenues dans un tableau d'enregistrements (*record*), l'écriture des

accès devient beaucoup plus rapide :

In[31]:

```
data_rec = data.view(np.recarray)
data_rec.age
```

Out[31]: array([25, 45, 37, 19], dtype=int32)

L'inconvénient de ces tableaux d'enregistrements est l'ajout de code de gestion supplémentaire pour accéder aux champs, ce que nous pouvons constater ici :

In[32]:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
```

```
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

En fonction des contraintes de vos applications, vous choisirez entre privilégier la notation allégée ou les performances.

2.10 : En route vers Pandas

J'ai volontairement terminé ce chapitre sur les tableaux structurés et tableaux d'enregistrements parce que cela constitue une excellente transition vers la présentation du prochain paquetage : Pandas. En effet, les tableaux structurés que nous venons de voir s'avèrent pratiques dans certaines situations, notamment lorsqu'il faut faire interagir des tableaux NumPy avec des formats de données binaires tels que ceux rencontrés en langage C, en Fortran ou un autre langage. Pour une utilisation au quotidien des données structurées, il est largement préférable de se tourner vers le paquetage Pandas. Préparons-nous donc à plonger dans les détails correspondants dans le prochain chapitre.

CHAPITRE 3

Manipulation de données avec Pandas

Le chapitre précédent nous a permis de découvrir en détail la librairie NumPy et son objet central ndarray. Il permet de stocker et de traiter efficacement des tableaux Python. Découvrons maintenant les structures de données que propose la librairie Pandas qui est construite en se fondant sur la librairie NumPy. Elle présente une implémentation efficace d'une structure de données correspondant au type DataFrame. Il s'agit d'un tableau à plusieurs dimensions doté de labels pour les lignes et les colonnes ; plusieurs types peuvent être combinés et les données manquantes sont autorisées. La librairie Pandas permet donc de stocker des données nommées par des labels et offre toute une gamme d'opérations similaires à celles dont on dispose avec un moteur de base de données ou un tableau.

Nous avons vu que le type ndarray de NumPy permettait de gérer de façon bien organisée les données dans le domaine des traitements numériques. L'objectif principal est atteint

de façon satisfaisante, mais des limitations apparaissent lorsque l'on a besoin de plus de souplesse (en associant des labels aux données, en traitant les données manquantes, *etc.*), ainsi que lorsqu'il faut réaliser des opérations qui cadrent mal avec une approche par diffusion élément par élément (groupement, pivot, *etc.*). Ces besoins sont cruciaux lorsqu'il s'agit d'analyser des données moins structurées, nombreuses dans le monde réel. Pandas, et notamment ses deux objets Series et DataFrame, s'appuient sur la structure des tableaux NumPy afin d'offrir un accès efficace aux opérations de transformation de données qui représentent l'activité quotidienne d'un catalogue.

Nous allons nous concentrer dans ce chapitre sur les mécanismes permettant d'utiliser les structures Series, DataFrame et autres. Nous prendrons des exemples qui exploitent des jeux de données réels lorsque c'est possible, mais le contenu complet des exemples n'est pas toujours le sujet en cours de description.

3.1 : Installer et utiliser Pandas

Pour installer Pandas dans votre système, vous devez d'abord avoir installé la librairie NumPy. Si vous désirez reconstruire la librairie à partir du code source, vous devez avoir installé les outils pour compiler un programme C et Cython, qui sont les langages utilisés pour Pandas. Vous trouverez tous les détails au sujet de l'installation dans la documentation de Pandas (<http://pandas.pydata.org/>). Si vous avez suivi les conseils donnés dans la préface en adoptant l'ensemble Anaconda, la librairie Pandas est déjà installée.

Une fois Pandas en place, vous pouvez en demander l'importation en interrogeant le numéro de version :

```
In[1]:  
import pandas  
pandas.__version__
```

```
Out[1]:  
'0.18.1'
```

Pour la librairie NumPy, nous avons choisi par convention l'alias np. Pour Pandas, nous choisissons l'alias pd :

```
In[2]: import pandas as pd
```

Nous utiliserons cette convention dans toute la suite du livre.

Un rappel à propos de la documentation interne

Vous avez accès à une aide directe en utilisant le mécanisme basé sur la touche tabulation pendant que vous saisissez le nom d'un élément du langage. Pour de l'aide au sujet des fonctions, vous ajoutez le signe point d'interrogation (?). Tout cela est expliqué dans l'introduction du livre.

Pour voir la liste des noms connus dans l'espace de noms de Pandas, vous saisissez ceci :

In [3]: **pd.<TAB>**

Pour accéder à la documentation de Pandas, vous saisissez :

In [4]: **pd?**

La documentation complète avec des tutoriels et d'autres ressources est disponible à l'adresse <http://pandas.pydata.org/>.

3.2 : Présentation des objets de Pandas

De façon résumée, les objets Pandas constituent une sorte de version sophistiquée des tableaux structurés NumPy. Les lignes et les colonnes sont dorénavant identifiées par les labels et non simplement par des indices numériques. Nous verrons au cours du chapitre que Pandas propose une vaste gamme d'outils de méthodes et de fonctions pour les structures de données. Tout ce qui suit suppose de bien comprendre d'abord ces structures. C'est pourquoi nous allons d'abord découvrir en détail les trois structures fondamentales de Pandas que sont Series, DataFrame, et Index.

Nous commençons bien sûr nos sessions d'exemples avec les directives pour importer les deux librairies NumPy et Pandas :

```
In[1]:  
import numpy as np  
import pandas as pd
```

L'objet Series de Pandas

L'objet Series est un tableau à une dimension contenant des données indexées. Vous pouvez en créer un à partir d'une liste ou d'un tableau :

In[2]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Out[2]:

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Le résultat précédent montre que Series accueille une séquence de valeurs et une séquence d'indices, l'accès pouvant se faire par les attributs values et index. La partie values revient à un tableau NumPy classique :

In[3]: **data.values**

```
Out[3]: array([ 0.25, 0.5 , 0.75, 1. ])
```

L'attribut index est un objet de type tableau ayant le type exact pd.Index, que nous décrirons plus en détail plus loin :

```
In[4]: data.index
```

```
Out[4]: RangeIndex(start=0, stop=4, step=1)
```

Vous pouvez accéder aux données comme dans le cas d'un tableau NumPy, par l'index en utilisant la notation Python entre crochets :

```
In[5]: data[1]
```

```
Out[5]: 0.5
```

```
In[6]: data[1:3]
```

```
Out[6]:
```

```
1    0.50
```

```
2    0.75
```

```
dtype: float64
```

Nous verrons que l'objet Series de Pandas est beaucoup plus souple que le tableau à une dimension NumPy qu'il émule.

Series : un tableau NumPy généralisé

Ce que nous venons de voir pousserait à considérer l'objet Series comme l'équivalent d'un tableau NumPy à une dimension. La différence se situe au niveau de la présence de l'index : dans un tableau NumPy, il y a toujours un index entier défini par défaut et permettant d'accéder aux valeurs. Dans l'objet Series de Pandas, l'index est défini explicitement en relation avec les valeurs.

C'est cette définition explicite qui apporte de nouvelles possibilités à un objet Series. Notamment, l'index peut être non seulement de type numérique entier, mais de n'importe quel autre type. Nous pouvons tout à fait utiliser des chaînes comme index :

In[7]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
index=['a', 'b', 'c', 'd'])  
data
```

Out[7]:

```
a    0.25  
b    0.50  
c    0.75  
d    1.00  
dtype: float64
```

L'accès aux éléments fonctionne comme prévu :

In[8]:

```
data['b']
```

Out[8]: 0.5

Les indices peuvent même être non séquentiels ou non voisins :

```
In[9]:
```

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
index=[2, 5, 3, 7])  
data
```

```
Out[9]:
```

```
2    0.25  
5    0.50  
3    0.75  
7    1.00  
dtype: float64
```

```
In[10]: data[5]
```

```
Out[10]: 0.5
```

Series : un dictionnaire spécialisé

L'objet Series peut donc être vu comme une version spécialisée d'un dictionnaire Python. Rappelons qu'un dictionnaire est une structure qui associe des clés arbitraires à des valeurs arbitraires. La structure Series réunit des clés typées à des valeurs typées. L'existence de ces types est importante : nous savons que lorsque le code est compilé avec mention des types, cela permet souvent une exécution bien plus efficace ; c'est déjà le cas des tableaux NumPy en comparaison des listes Python dans certains traitements. De même, l'information de type d'un objet Series de Pandas le

rend beaucoup plus efficace dans certains traitements qu'un dictionnaire Python.

L'analogie entre dictionnaire et objet Series est encore plus évidente lorsque l'on tente de construire un objet Series directement depuis un dictionnaire Python :

In[11]:

```
population_dict = {'California': 38332521,
                   'Florida': 19552860,
                   'Illinois': 12882135,
                   'New York': 19651127,
                   'Texas': 26448193}
population = pd.Series(population_dict)
population
```

Out[11]:

```
California 38332521
Florida    19552860
Illinois   12882135
New York   19651127
Texas      26448193
dtype: int64
```

L'objet Series est créé par défaut en prenant les index à partir des clés triées. Il devient ensuite possible d'accéder aux éléments comme dans le cas d'un dictionnaire :

In[12]: `population['California']`

```
Out[12]: 38332521
```

En revanche, à la différence du dictionnaire Python, l'objet Series autorise les opérations de type tableau telles que les extractions par tranchage (*slicing*) :

```
In[13]: population['California':'Illinois']
```

```
Out[13]:
```

```
California    38332521
Florida      19552860
Illinois     12882135
dtype: int64
```

Nous verrons quelques pages plus loin les particularités de l'indexation et du tranchage dans la librairie Pandas.

Construction d'objets Series

Nous venons de voir comment construire un objet Series à partir de zéro. Dans tous les cas, c'est une variante de cette instruction :

```
>>> pd.Series(data, index=index)
```

Dans l'exemple, index est un paramètre facultatif et data est une entité choisie parmi de nombreuses possibilités.

C'est ainsi que data peut être une liste ou un tableau NumPy. Dans ces cas, index est une séquence de valeurs entières :

```
In[14]: pd.Series([2, 4, 6])
```

```
Out[14]:
```

```
0    2  
1    4  
2    6  
dtype: int64
```

data peut aussi être un scalaire que l'on répète pour remplir l'index spécifié :

```
In[15]: pd.Series(5, index=[100, 200, 300])
```

```
Out[15]:
```

```
100    5  
200    5  
300    5  
dtype: int64
```

data peut enfin être un dictionnaire. Dans ce cas, index correspond par défaut aux clés triées du dictionnaire :

```
In[16]: pd.Series({2:'a', 1:'b', 3:'c'})
```

```
Out[16]:
```

```
1    b  
2    a  
3    c  
dtype: object
```

Dans tous les cas, il reste possible de définir l'index de façon explicite lorsque l'on a besoin d'un autre résultat :

```
In[17]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

```
Out[17]:
```

```
3    c  
2    a  
dtype: object
```

Dans ce dernier exemple, l'objet Series n'est peuplé qu'avec les clés explicitement identifiées.

L'objet DataFrame de Pandas

La deuxième des trois structures fondamentales de Pandas est DataFrame. Comme Series que nous venons de présenter, DataFrame peut être considéré soit comme une généralisation d'un tableau NumPy, soit comme une spécialisation d'un dictionnaire Python. Voyons tour à tour ces deux points de vue.

DataFrame : un tableau NumPy généralisé

Nous avons dit que l'objet Series était l'équivalent d'un tableau à une dimension avec des indices souples. De même, l'objet DataFrame équivaut à un tableau à deux dimensions

offrant de la souplesse au niveau des index de lignes et des noms de colonnes. Un tableau à deux dimensions peut être considéré comme une séquence triée de colonnes en une dimension. De même, l'objet DataFrame peut être vu comme une séquence d'objets Series alignés. Le terme « aligné » signifie ici que les éléments partagent le même index.

Voyons cela en pratique en construisant d'abord un nouvel objet Series qui dresse la liste des codes postaux pour les cinq états déjà vus dans la section précédente :

In[18]:

```
area_dict = {'California': 423967, 'Florida':  
170312, 'Illinois': 149995,  
             'New York': 141297, 'Texas': 695662}  
area = pd.Series(area_dict)  
area
```

Out[18]:

```
California    423967  
Florida       170312  
Illinois      149995  
New York     141297  
Texas        695662  
dtype: int64
```

Une fois ceci défini et en réutilisant l'objet Series *population* de l'exemple précédent, nous pouvons utiliser un

dictionnaire pour construire un objet en deux dimensions pour accueillir ces informations :

In[19]:

```
states = pd.DataFrame({'population': population,
'area': area})
states
```

Out[19]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Comme l'objet Series, l'objet DataFrame possède un attribut nommé index qui permet d'accéder aux labels des indices :

In[20]: `states.index`

Out[20]:

```
Index(['California', 'Florida', 'Illinois', 'New
York', 'Texas'], dtype='object')
```

L'objet DataFrame possède l'attribut columns qui est un objet Index contenant les labels des colonnes :

In[21]: `states.columns`

```
Out[21]: Index(['area', 'population'],
dtype='object')
```

C'est pourquoi l'objet DataFrame peut être considéré comme une version généralisée d'un tableau NumPy à deux dimensions, les lignes et les colonnes disposant d'un index généralisé pour accéder aux données.

DataFrame : un dictionnaire spécialisé

L'objet DataFrame peut être considéré comme un dictionnaire spécialisé. Un dictionnaire associe une clé à une valeur ; de même, DataFrame associe un nom de colonne à un objet Series de colonne de données. On peut par exemple demander l'attribut area et récupérer l'objet Series qui contient les codes des zones postales :

```
In[22]: states['area']
```

```
Out[22]:
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Une confusion est possible ici : dans un tableau NumPy à deux dimensions, data[0] renvoie la première ligne. Pour un

objet DataFrame, l'écriture `data['colo']` renvoie la première colonne. C'est pourquoi il vaut mieux considérer un objet DataFrame comme un dictionnaire généralisé plutôt que comme un tableau généralisé, même si les deux approches peuvent être utiles. Nous verrons un peu plus loin d'autres techniques plus souples pour indiquer un objet DataFrame.

Construction d'un objet DataFrame

Plusieurs techniques permettent de construire un objet DataFrame Pandas.

À partir d'un seul objet Series. L'objet DataFrame est une collection d'objets Series. Un DataFrame contenant une seule colonne peut être construit à partir d'un objet Series unique :

```
In[23]: pd.DataFrame(population, columns=[ 'population'])
```

```
Out[23]:
```

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

À partir d'une liste de dictionnaires. N'importe quelle liste peut être convertie en un objet DataFrame. Créons quelques données avec une liste par compréhension :

In[24]:

```
data = [{ 'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

Out[24]:

	a	b
0	0	0
1	1	2
2	2	4

Si certaines clés du dictionnaire manquent, Pandas va insérer à leur place la pseudo-valeur NaN (<< n'est pas un nombre >>) :

In[25]:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[25]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

À partir d'un dictionnaire d'objets Series. Nous avons déjà vu qu'un objet DataFrame pouvait être construit à partir d'un dictionnaire d'objets Series :

In[26]:

```
pd.DataFrame({'population': population, 'area': area})
```

Out[26]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

À partir d'un tableau NumPy à deux dimensions. À partir d'un tel tableau, nous créons un objet DataFrame avec les colonnes spécifiées et les noms d'index. Si les noms ne sont pas fournis, une valeur entière est appliquée pour les indices :

In[27]:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Out[27]:

foo	bar
a	0.123456789
b	0.234567890
c	0.345678901

```
a      0.865257  0.213169  
b      0.442759  0.108267  
c      0.047110  0.905718
```

À partir d'un tableau structuré NumPy. Nous avons rencontré les tableaux structurés à la fin du chapitre précédent. L'objet DataFrame de Pandas s'utilise un peu comme un tableau structuré, et peut donc être créé directement à partir de celui-ci :

In[28]:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B',  
'f8')])  
A
```

Out[28]:

```
array([(0, 0.0), (0, 0.0), (0, 0.0)],  
      dtype=[('A', '<i8'), ('B', '<f8')])
```

In[29]: pd.DataFrame(A)

Out[29]:

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

L'objet Index de Pandas

Nous savons que les deux objets Series et DataFrame possèdent un index explicite qui permet de faire référence aux données et de les modifier. Cet objet est lui-même une structure intéressante que l'on peut considérer soit comme un tableau immuable, soit comme un ensemble trié. (Techniquement, c'est un multiensemble, car les objets Index peuvent contenir des valeurs répétées.) Ces deux perspectives ont des conséquences intéressantes au niveau des traitements applicables à l'objet Index. En guise d'exemple simple, construisons un objet Index à partir d'une liste d'entiers :

In[30]:

```
ind = pd.Index([2, 3, 5, 7, 11])  
ind
```

Out[30]:

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index : un tableau immuable

L'objet Index ressemble beaucoup à un tableau. Nous pouvons nous servir de la notation d'indice Python standard pour récupérer des valeurs ou des tranches :

In[31]: **ind[1]**

Out[31]: 3

In[32]: `ind[::-2]`

```
Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

L'objet Index dispose de nombreux attributs déjà connus pour les tableaux NumPy :

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

Une différence importante entre Index et un tableau NumPy est que les index sont immuables ; ils ne peuvent pas être modifiés par les méthodes habituelles :

In[34]: `ind[1] = 0`

```
TypeError Traceback (most recent call last)
<ipython-input-34-40e631c82e8a> in <module>()
-----> 1 ind[1] = 0
/Users/jakevdp/anaconda/lib/python3.5/site-
packages/pandas/indexes/base.py ...
1243
1244 def __setitem__(self, key, value):
-> 1245     raise TypeError("Index does not support
mutable operations")
1246
```

```
1247 def __getitem__(self, key):  
TypeError: Index does not support mutable  
operations
```

Cette immutabilité permet de partager des indices entre plusieurs objets DataFrame de façon plus sûre par rapport à des tableaux, sans risquer les effets secondaires résultant d'une modification involontaire d'un indice.

Index : un ensemble ordonné

Les objets Pandas sont conçus de sorte de simplifier les opérations de jointure entre ensembles de données (qui dépendent par de nombreux aspects de l'arithmétique des ensembles). L'objet Index obéit à la plupart des conventions en vigueur pour les structures de données ensembles (set) de Python, ce qui autorise les opérations classiques telles que les unions, les intersections, les différences et autres :

In[35]:

```
indA = pd.Index([1, 3, 5, 7, 9])  
indB = pd.Index([2, 3, 5, 7, 11])
```

In[36]:

```
indA & indB      # intersection
```

Out[36]:

```
Int64Index([3, 5, 7], dtype='int64')
```

```
In[37]:
```

```
indA | indB      # union
```

```
Out[37]:
```

```
Int64Index([1, 2, 3, 5, 7, 9, 11],  
dtype='int64')
```

```
In[38]:
```

```
indA ^ indB      # différence symétrique
```

```
Out[38]:
```

```
Int64Index([1, 2, 9, 11], dtype='int64')
```

Toutes ces opérations sont également possibles au moyen de méthodes de l'objet, par exemple

`indA.intersection(indB)`.

3.3 : Indexation et sélection de données

Nous avons vu en détail dans le [Chapitre 2](#) les différentes méthodes et outils pour lire et modifier les valeurs stockées dans des tableaux NumPy : indexation (par exemple `arr[2, 1]`), tranchage (p.e. `arr[:, 1:5]`), masquage (p.e. `arr[arr > 0]`), indexation fancy (p.e. `arr[0, [1, 5]]`) et leurs combinaisons (p.e. `arr[:, [1, 5]]`).

Découvrons des techniques similaires applicables à des objets Series et DataFrame de Pandas. Les conditions d'utilisation sont proches de celles que vous avez pu découvrir à propos de NumPy, sauf quelques points particuliers qui méritent de prendre quelques précautions.

Commençons par le cas le plus simple d'un objet Series à une dimension. Nous verrons dans un second temps ceux à deux dimensions.

Sélection de données dans Series

Nous venons de voir qu'un objet Series pouvait être considéré sous plusieurs aspects comme un tableau NumPy à une dimension, et sous de nombreux autres comme un dictionnaire Python standard. Conservons ces deux

analogies pour mieux comprendre les mécanismes d'indexation et de sélection de données dans ce genre d'objets.

L'objet Series : un dictionnaire

Comme un dictionnaire, l'objet Series fournit une association entre une collection de clés et une collection de valeurs :

In[1]:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
data
```

Out[1]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In[2]: data['b']

Out[2]: 0.5

Nous pouvons examiner les clés ou index et les valeurs avec des expressions de méthodes classiques des dictionnaires Python :

```
In[3]: 'a' in data
```

```
Out[3]: True
```

```
In[4]: data.keys()
```

```
Out[4]: Index(['a', 'b', 'c', 'd'],  
dtype='object')
```

```
In[5]: list(data.items())
```

```
Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75),  
( 'd', 1.0)]
```

Vous pouvez modifier un objet Series au moyen de la syntaxe des dictionnaires. De même que l'on peut agrandir un dictionnaire en assignant une nouvelle clé, on peut agrandir un objet Series en l'associant à une nouvelle valeur d'index :

```
In[6]:
```

```
data['e'] = 1.25
```

```
data
```

```
Out[6]:
```

```
a    0.25  
b    0.50  
c    0.75  
d    1.00  
e    1.25  
dtype: float64
```

Cette possibilité de modifier facilement l'objet est très pratique. C'est la librairie Pandas qui se charge de gérer l'implantation mémoire et les éventuelles copies de données ; en général, le programmeur n'a pas à s'en soucier.

L'objet Series : un tableau à une dimension

L'objet Series se fonde sur cette interface de style dictionnaire pour permettre la sélection d'éléments dans le même style que les tableaux NumPy. Vous disposez ainsi du tranchage, du masquage et de l'indexation fancy. Voici quelques exemples :

```
In[7]: # Tranchage avec index explicite  
data['a':'c']
```

```
Out[7]:  
a    0.25  
b    0.50  
c    0.75  
dtype: float64
```

```
In[8]: # Tranchage par index entier implicite  
data[0:2]
```

```
Out[8]:  
a    0.25  
b    0.50  
dtype: float64
```

```
In[9]: # Masquage  
data[(data > 0.3) & (data < 0.8)]
```

```
Out[9]:  
b    0.50  
c    0.75  
dtype: float64
```

```
In[10]: # Indexation fancy  
data[['a', 'e']]
```

```
Out[10]:  
a    0.25  
e    1.25  
dtype: float64
```

La principale cause de confusion concerne le tranchage. Lorsque vous demandez un tranchage avec un index explicite (par exemple `data['a' : 'c']`), l'index final est inclus dans la tranche, alors que si vous tranchez avec un index implicite (par exemple `data[0 : 2]`), cette borne finale est *exclue* de la tranche.

Les indexeurs `loc`, `iloc` et `ix`

Les divergences dans les conventions entre indexation et tranchage peuvent facilement entraîner des confusions. Par exemple, pour un objet Series doté d'un index entier

explicite, une opération d'indexation telle que `data[1]` utilisera l'index explicite alors qu'une opération de tranchage telle que `data[1 : 3]` utilisera l'index implicite de style Python :

In[11]:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[11]:

```
1    a
3    b
5    c
dtype: object
```

In[12]: # Index explicite à l'indexation

```
data[1]
```

Out[12]: 'a'

In[13]: # Index implicite au tranchage

```
data[1:3]
```

Out[13]:

```
3    b
5    c
dtype: object
```

Pandas propose des attributs appelés *indexeurs* pour éviter ce genre de confusion en relation avec des index entiers. Ils permettent de désigner explicitement une méthode d'indexation désirée. Ce ne sont pas des méthodes, mais des attributs qui déclarent et rendent accessibles une interface de tranchage pour les données de Series.

Le premier attribut se nomme loc. Il permet de formuler une indexation ou un tranchage en faisant systématiquement référence à l'index explicite :

```
In[14]: data.loc[1]  
Out[14]: 'a'
```

```
In[15]: data.loc[1:3]  
Out[15]:  
1    a  
3    b  
dtype: object
```

L'attribut iloc fait lui toujours référence à l'index implicite dans le style Python, pour l'indexation et le tranchage :

```
In[16]: data.iloc[1]  
Out[16]: 'b'
```

```
In[17]: data.iloc[1:3]  
Out[17]:  
3    b
```

```
5      c  
dtype: object
```

Il existe un troisième attribut nommé `ix` qui combine les deux précédents. Pour un objet `Series`, il équivaut à l'indexation classique basée sur la notation `[]`. L'intérêt de cet indexeur `ix` deviendra beaucoup plus évident quand nous décrirons les objets `DataFrame` un peu plus loin.

Un grand principe en Python est de toujours privilégier l'explicite par rapport à l'implicite. Les deux attributs `loc` et `iloc` deviennent donc précieux pour maintenir la lisibilité du code. Je vous conseille de vous en servir, notamment dans le cas des index entiers, car vous éviterez ainsi quelques bogues subtils qui sont la conséquence des divergences entre indexation et tranchage comme nous venons de le voir.

Sélection de données et `DataFrame`

Nous savons qu'un objet `DataFrame` peut être considéré comme un tableau structuré ou à deux dimensions, mais également comme un dictionnaire de structures `Series` qui partagent le même `index`. Ces deux analogies nous seront utiles pour voir comment faire une sélection de données avec cette structure.

L'objet DataFrame : un dictionnaire

Voyons d'abord comment exploiter un DataFrame sous forme d'un dictionnaire contenant des objets Series reliés. Reprenons notre exemple de populations et de codes :

In[18]:

```
area = pd.Series({'California': 423967, 'Texas':  
695662,  
                 'New York': 141297, 'Florida':  
170312,  
                 'Illinois': 149995})  
pop = pd.Series({'California': 38332521,  
'Texas': 26448193,  
                 'New York': 19651127, 'Florida':  
19552860,  
                 'Illinois': 12882135})  
data = pd.DataFrame({'area':area, 'pop':pop})  
data
```

Out[18]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

On accède à chacun des objets Series qui incarnent les colonnes de l'objet DataFrame au moyen d'une indexation

dans le style dictionnaire avec le nom de colonne :

```
In[19]: data['area']
```

Out[19]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas          695662
Name: area, dtype: int64
```

On peut aussi utiliser la syntaxe de type attribut pour les noms de colonnes qui sont des chaînes :

```
In[20]: data.area
```

Out[20]:

```
California    423967
Florida       170312
Illinois      149995
New York      141297
Texas          695662
Name: area, dtype: int64
```

Les deux syntaxes permettent d'accéder exactement au même objet, comme le confirme le test suivant :

```
In[21]: data.area is data['area']
```

```
Out[21]: True
```

Notez que la syntaxe basée sur l'attribut n'est pas toujours disponible. Si le nom de colonne n'est pas constitué d'une chaîne, ou si le nom de colonne entre en conflit avec celui d'une méthode de l'objet DataFrame, cette syntaxe est inaccessible. L'objet DataFrame possède par exemple une méthode `pop()`, et l'écriture `data.pop` va déclencher cette méthode et non accéder à la colonne nommée `pop` :

```
In[22]: data.pop is data['pop']  
Out[22]: False
```

Pour exclure tout souci, évitez toute affectation de colonne par la syntaxe d'attribut. Écrivez plutôt `data['pop'] = z` plutôt que `data.pop = z`.

Comme dans le cas d'un objet Series, la syntaxe de style dictionnaire permet tout aussi bien de modifier l'objet. Voici par exemple comment ajouter une colonne :

```
In[23]:
```

```
data['density'] = data['pop'] / data['area']  
data
```

```
Out[23]:
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

```
New York    141297  19651127  139.076746  
Texas      695662   26448193  38.018740
```

C'est une sorte d'avant-goût de la syntaxe très efficace de l'arithmétique par élément entre plusieurs objets Series que nous verrons dans quelques pages.

L'objet DataFrame : un tableau à deux dimensions

Nous avons dit que nous pouvions considérer un objet DataFrame comme une sorte de tableau à deux dimensions sophistiqué. Pour accéder au tableau de données sous-jacent, nous nous servons de l'attribut values :

```
In[24]: data.values
```

```
Out[24]:
```

```
array([[ 4.23967000e+05,  3.83325210e+07,  
 9.04139261e+01],  
       [ 1.70312000e+05,  1.95528600e+07,  
 1.14806121e+02],  
       [ 1.49995000e+05,  1.28821350e+07,  
 8.58837628e+01],  
       [ 1.41297000e+05,  1.96511270e+07,  
 1.39076746e+02],  
       [ 6.95662000e+05,  2.64481930e+07,  
 3.80187404e+01]])
```

Sachant cela, nous pouvons envisager sur un objet DataFrame les opérations habituelles aux tableaux. Nous pouvons par exemple intervertir les lignes et les colonnes de l'objet DataFrame :

```
In[25]: data.T
```

```
Out[25]:
```

	California	Florida	Illinois	New
York	Texas			
area	4.239670e+05	1.703120e+05	1.499950e+05	
	1.412970e+05	6.956620e+05		
pop	3.833252e+07	1.955286e+07	1.288214e+07	
	1.965113e+07	2.644819e+07		
density	9.041393e+01	1.148061e+02	8.588376e+01	
	1.390767e+02	3.801874e+01		

En ce qui concerne les opérations d'indexation, l'approche d'indexation de style dictionnaire pour les colonnes nous empêche de considérer un objet DataFrame comme un simple objet de type tableau NumPy. Rappelons que pour un tableau, en transmettant un seul index, on accède à une ligne :

```
In[26]: data.values[0]
```

```
Out[26]:
```

```
array([ 4.23967000e+05,      3.83325210e+07,
       9.04139261e+01])
```

En transmettant un seul index à un objet DataFrame, on accède à une colonne :

```
In[27]: data['area']
```

```
Out[27]:
```

```
California    423967  
Florida       170312  
Illinois      149995  
New York     141297  
Texas         695662  
Name: area, dtype: int64
```

Il nous faut donc une autre convention pour l'indexation dans le style des tableaux. Comme précédemment, Pandas propose les trois indexeurs loc, iloc et ix. Avec l'indexeur iloc, nous pouvons indexer le tableau contenu comme si c'était un simple tableau NumPy, donc avec le style d'indexation Python standard. En revanche, et heureusement, les index et les labels de colonnes de l'objet DataFrame sont conservés :

```
In[28]: data.iloc[:3, :2]
```

```
Out[28]:
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

```
In[29]: data.loc[:'Illinois', : 'pop']
```

```
Out[29]:
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

L'attribut indexeur `ix` offre une combinaison des deux approches :

```
In[30]: data.ix[:3, : 'pop']
```

```
Out[30]:
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Rappelons que dans le cas des indices entiers, `ix` peut constituer la même source de confusion que celle vue lorsque nous avons décrit les objets `Series` avec indexation par entiers.

Toutes les techniques d'accès aux données de style NumPy peuvent être utilisées avec les attributs indexeurs. Par exemple, avec `loc`, nous pouvons combiner masquage et indexation fancy :

```
In[31]: data.loc[data.density > 100, ['pop',  
'density']]
```

Out[31]:

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Ces différentes conventions permettent aussi de modifier les valeurs, en utilisant les mêmes techniques que celles présentées pour NumPy :

```
In[32]:
```

```
data.iloc[0, 2] = 90  
data
```

Out[32]:

	area	pop	density
California	423967	38332521	90.000000
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

Pour renforcer votre maîtrise de la manipulation des données Pandas, je vous conseille de vous entraîner un peu avec un objet DataFrame simple en testant les différents types d'indexation, de tranchage, de masquage et

d'indexation fancy que permettent ces techniques d'indexation.

Autres conventions d'indexation

Il existe enfin quelques conventions d'indexation qui semblent diverger avec ce que nous venons de présenter ; elles s'avèrent pourtant bien pratiques. Tout d'abord, alors que l'indexation fait référence aux colonnes, le tranchage fait référence aux lignes :

```
In[33]: data['Florida':'Illinois']
```

```
Out[33]:
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

De telles tranches peuvent désigner les lignes par un numéro de ligne plutôt que par l'index :

```
In[34]: data[1:3]
```

```
Out[34]:
```

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

De même, les opérations de masquage direct sont comprises par ligne et non par colonne :

```
In[35]: data[data.density > 100]
```

```
Out[35]:
```

	area	pop	density
Florida	170312	19552860	114.806121
New York	141297	19651127	139.076746

Ces deux conventions ressemblent en termes de syntaxe à celles valables pour un tableau NumPy. Elles ne sont pas en accord parfait avec les conventions de la librairie Pandas, mais elles sont assez utiles en pratique.

3.4 : Opérations sur les données Pandas

Un des points forts de NumPy est de permettre des opérations rapides au niveau des éléments au moyen d'opérations arithmétiques de base (addition, soustraction, *etc.*) ou plus complexes (trigonométrie, exponentielles, logarithmes, *etc.*). La librairie Pandas hérite de quasiment toutes ces fonctions auprès de NumPy, et elle tire notamment profit des fonctions universelles ufuncs présentées dans le chapitre précédent.

Les objets de Pandas étant plus complexes, quelques compléments deviennent disponibles : pour les opérations unaires, comme par exemple une négation ou une fonction trigonométrique, les fonctions ufuncs maintiennent les *index* et les *labels de colonnes* dans le résultat. Pour les opérations binaires (additions ou multiplications), Pandas aligne automatiquement les index lors du passage des objets à la fonction. La conséquence est que le maintien du contexte et la combinaison de données de plusieurs sources (opérations pouvant entraîner des erreurs dans le cas de tableaux NumPy élémentaires) deviennent des opérations quasiment sûres avec Pandas. Nous verrons aussi qu'il existe plusieurs opérations clairement définies entre des structures Series à

une dimension et des structures DataFrame à deux dimensions.

Préservation des index avec les ufuncs

Pandas est prévu et conçu pour travailler en combinaison avec NumPy. Toutes les fonctions ufuncs de NumPy peuvent s'appliquer à un objet Series ou DataFrame de Pandas. Commençons par définir un objet Series et un autre DataFrame simple pour nos démonstrations :

In[1]:

```
import pandas as pd  
import numpy as np
```

In[2]:

```
rng = np.random.RandomState(42)  
ser = pd.Series(rng.randint(0, 10, 4))  
ser
```

Out[2]:

```
0    6  
1    3  
2    7  
3    4  
dtype: int64
```

In[3]:

```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),  
                  columns=['A', 'B', 'C', 'D'])  
df
```

Out[3]:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

Si nous appliquons une ufunc de NumPy à un de ces objets, nous obtenons un autre objet Pandas dans lequel les *indices sont préservés* :

In[4]: np.exp(ser)

Out[4]:

0	403.428793
1	20.085537
2	1096.633158
3	54.598150

dtype: float64

Voici un calcul un peu plus complexe :

In[5]: np.sin(df * np.pi / 4)

Out[5]:

A	B	C	D
---	---	---	---

```
0 -1.000000 7.071068e-01 1.000000
-1.000000e+00
1 -0.707107 1.224647e-16 0.707107
-7.071068e-01
2 -0.707107 1.000000e+00 -0.707107
1.224647e-16
```

Toutes les fonctions ufuncs décrites dans le [Chapitre 2](#) peuvent être utilisées sur le même modèle.

Fonctions ufuncs et alignement d'index

Dans le cas d'une opération binaire concernant deux objets Series ou DataFrame, nous avons dit que Pandas alignait les index pendant l'opération. Cette possibilité est bienvenue lorsque vous travaillez avec des données incomplètes, ce que nous verrons dans certains des exemples suivants.

Alignement des index dans Series

Supposons que nous ayons trouvé deux sources de données différentes en vue de sélectionner les trois états des USA ayant la plus grande superficie et ceux ayant la plus grande population :

```
In[6]:
area = pd.Series({'Alaska': 1723337, 'Texas':
695662,
```

```
'California': 423967},  
name='area')  
population = pd.Series({'California': 38332521,  
'Texas': 26448193,  
'New York': 19651127},  
name='population')
```

Que se passe-t-il si nous tentons une division en vue d'obtenir la densité de population ?

In[7]:
population / area

Out[7]:

Alaska	NaN
California	90.413926
New York	NaN
Texas	38.018740

dtype: float64

Le tableau résultant contient l'*union* des index des deux tableaux d'entrées, nous pouvons les connaître au moyen de l'arithmétique d'ensemble Python standard appliquée à ces index :

In[8]:
area.index | population.index

Out[8]:

```
Index(['Alaska', 'California', 'New York',
       'Texas'], dtype='object')
```

Tout élément pour lequel il n'y a pas d'entrée dans l'autre source reçoit la pseudo-valeur NaN qui est la façon dont Pandas notifie d'un manque de données (nous revenons sur les données manquantes dans la prochaine section). Cette façon d'associer des index s'applique à toutes les expressions arithmétiques standard de Python. Chaque valeur manquante est remplacée par la notation NaN par défaut :

In[9]:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

Out[9]:

```
0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

Il est possible de choisir une autre pseudo-valeur d'absence, en utilisant la méthode d'objet appropriée à la place d'un opérateur. Par exemple, écrire A.add(B) est équivalent à A + B, mais permet aussi de choisir la pseudo-valeur d'absence pour tout élément non présent dans A ou B :

```
In[10]:
```

```
A.add(B, fill_value=0)
```

```
Out[10]:
```

```
0    2.0  
1    5.0  
2    9.0  
3    5.0  
dtype: float64
```

Alignement des index dans DataFrame

Le même mécanisme d'alignement s'applique simultanément aux colonnes et aux index dans les opérations sur un objet DataFrame :

```
In[11]:
```

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),  
columns=list('AB'))  
A
```

```
Out[11]:
```

```
   A   B  
0  1  11  
1  5   1
```

```
In[12]:
```

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),  
columns=list('BAC'))  
B
```

```
Out[12]:
```

```
    B   A   C  
0   4   0   9  
1   5   8   0  
2   9   2   6
```

```
In[13]:
```

```
A + B
```

```
Out[13]:
```

```
      A      B      C  
0    1.0    15.0    NaN  
1   13.0     6.0    NaN  
2    NaN     NaN    NaN
```

Notez bien que les index sont correctement alignés, quel que soit leur ordre dans les deux objets d'entrée et que les index sont triés dans le résultat. Comme pour l'objet Series, vous pouvez utiliser une méthode arithmétique pour spécifier une valeur d'absence par le paramètre `fill_value`. Dans l'exemple, nous remplissons avec la moyenne de toutes les valeurs trouvées dans A (calculées d'abord en empilant les lignes de A) :

```
In[14]:
```

```
fill = A.stack().mean()  
A.add(B, fill_value=fill)
```

Out[14] :

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

Le [Tableau 3.1](#) montre les opérateurs Python avec les méthodes d'objet Pandas équivalentes.

Tableau 3.1 : Opérateurs Python et méthodes Pandas.

Opérateur Python	Méthode Pandas
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div()
//	floordiv()
%	mod()
**	pow()

Opérations ufuncs entre DataFrame et Series

L'alignement des index et des colonnes est également préservé dans les opérations entre un objet DataFrame et un

objet Series. Ces opérations sont les mêmes que celles entre un tableau NumPy à une dimension et un autre à deux dimensions. Essayons une opération habituelle consistant à trouver la différence entre un tableau à deux dimensions et l'une de ses lignes :

In[15]:

```
A = rng.randint(10, size=(3, 4))  
A
```

Out[15]:

```
array([[3, 8, 2, 4],  
       [2, 6, 4, 8],  
       [6, 1, 3, 8]])
```

In[16]:

```
A - A[0]
```

Out[16]:

```
array([[ 0,  0,  0,  0],  
       [-1, -2,  2,  4],  
       [ 3, -7,  1,  4]])
```

D'après les règles de diffusion de NumPy vues dans le chapitre précédent, la soustraction entre un tableau à deux dimensions et l'une de ses lignes est réalisée dans le sens des lignes.

Par défaut dans Pandas, la convention demande de suivre aussi le sens des lignes :

In[17] :

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

Out[17] :

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

Pour opérer par colonnes, il suffit de recourir aux méthodes d'objets indiquées plus haut, en utilisant le mot-clé axis :

In[18] :

```
df.subtract(df['R'], axis=0)
```

Out[18] :

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

Comme les opérations précédentes, les index entre les deux éléments sont automatiquement alignés :

In[19] :

```
halfrow = df.iloc[0, ::2]
```

```
halfrow
```

```
Out[19]:
```

```
Q      3  
S      2  
Name: 0, dtype: int64
```

```
In[20]:
```

```
df - halfrow
```

```
Out[20]:
```

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	-1.0	NaN	2.0	NaN
2	3.0	NaN	1.0	NaN

La préservation et l'alignement des index et des colonnes ont pour conséquence que les opérations sur les données Pandas maintiennent toujours le contexte des données, ce qui épargne de faire les erreurs élémentaires qui peuvent survenir lorsque vous travaillez avec des données hétérogènes contenant des données manquantes dans les tableaux NumPy bruts.

3.5 : Gestion des données manquantes

Les données à traiter qui sont issues du monde réel se distinguent des données utilisées dans les exemples car elles sont rarement homogènes et propres. De nombreux jeux de données très intéressants comportent des « trous », des données manquantes. Pour ne pas simplifier les choses, la façon dont les données manquantes sont signalées varie d'une source de données à une autre. Voyons dans cette section dans les grandes lignes comment il faut gérer les données manquantes. Nous verrons comment elles sont symbolisées dans Pandas puis utiliserons quelques outils intégrés à Pandas pour gérer ces données manquantes. Dans la suite du livre, nous parlerons de données manquantes lorsque nous parlerons de données nulles, `NaN` (non numériques), ou `NA` (non accessibles).

Compromis conventionnels des données manquantes

Différentes stratégies ont été développées pour informer de l'absence de données dans une table ou dans un objet `DataFrame`. Les deux stratégies essentielles consistent soit à utiliser un *masque* pour indiquer globalement les valeurs

manquantes, soit à définir une valeur *sentinelle* qui correspond à une donnée manquante.

Dans l'approche utilisant le masque, on suppose que soit celui-ci est un tableau booléen indépendant, soit qu'il utilise un bit dans la représentation de données pour marquer la valeur comme étant non exploitable, nulle.

Dans l'approche sentinelle, il s'agit de choisir une valeur particulière selon une convention. On peut par exemple décider qu'une valeur entière manquante correspond à la valeur -9999 , ou que cela corresponde à un motif binaire spécifique. On peut aussi adopter une convention répandue, par exemple la convention IEEE concernant les valeurs à virgule flottante qui définit une valeur spéciale comme étant un Non-Nombre, NaN.

Chaque approche est un compromis : si vous optez pour un tableau de masquage, il faut mettre en place le tableau booléen complémentaire, ce qui réclame du temps et de l'espace de traitement. Avec une valeur sentinelle, cette valeur n'est plus utilisable pour les données pertinentes, et cela suppose en général d'écrire des traitements logiques (souvent non optimisés) au niveau du processeur CPU ou du coprocesseur arithmétique. Enfin, les pseudo-valeurs telles que NaN ne sont pas disponibles pour tous les types de données.

Lorsque aucun choix optimal ne peut s'appliquer de façon universelle, chaque langage et chaque système va adopter une convention spécifique. Le langage R utilise par exemple un patron binaire pour chaque type de données en tant que valeur sentinelle pour les données manquantes. Autre exemple, le système SciDB définit un octet supplémentaire associé à chaque cellule pour indiquer l'état NA, Non Accessible.

Données manquantes dans Pandas

Du fait que Pandas dépend du paquetage NumPy, sa gestion des données manquantes est contrainte. Le paquetage NumPy ne connaît pas la notion de valeur manquante pour les types de données autres que ceux à virgule flottante.

Pandas aurait pu adopter l'approche du langage R et définir des patrons binaires pour chaque type afin d'indiquer la nullité, mais l'approche aurait été assez complexe. En effet, R connaît quatre types de données élémentaires, alors que NumPy en supporte beaucoup plus. R ne gère qu'un type entier alors que NumPy en supporte quatorze (si l'on tient compte des différentes précisions, versions signées ou pas et choix du boutisme ou ordre d'encodage des bits). S'il avait fallu créer un patron binaire pour chacun des types NumPy, cela aurait entraîné une telle surcharge pour gérer les cas particuliers des différents types, qu'une nouvelle branche de

développement à partir du paquetage NumPy (*fork*) serait apparue en concurrence de Pandas. Enfin, sacrifier un bit pour ce masque pour les types compacts tels que les entiers sur 8 bits réduirait trop la plage de valeurs représentable (cela la divise par deux).

NumPy permet d'utiliser un tableau avec masque, c'est-à-dire un tableau auquel est associé un masque booléen pour marquer les données comme exploitables ou non exploitables. Pandas aurait pu s'en inspirer, mais ici aussi la surcharge rend ce choix peu attrayant, au niveau de l'espace de stockage, du temps de calcul et de la maintenance du code.

Une fois ces différentes contraintes énumérées, Pandas a choisi d'utiliser la technique des sentinelles, en réutilisant deux valeurs nulles qui existent déjà dans le langage Python : la valeur normalisée NaN pour les nombres à virgule flottante et l'objet None de Python. Ce choix a quelques effets secondaires, comme nous le verrons, mais il constitue en pratique un compromis acceptable dans la plupart des cas.

None : données manquantes de style Python

La première des deux valeurs sentinelles utilisées par Pandas s'écrit donc None (rien). C'est un objet singleton de Python qui est généralement utilisé pour la même raison

dans le code Python. Puisque c'est un objet Python, il n'est pas utilisable dans n'importe quel tableau NumPy ou Pandas, seulement dans les tableaux dont le type de données est object (c'est-à-dire des tableaux d'objets Python) :

In[1]:

```
import numpy as np
import pandas as pd
```

In[2]:

```
vals1 = np.array([1, None, 3, 4])
vals1
```

Out[2]:

```
array([1, None, 3, 4], dtype=object)
```

La mention `dtype=object` signifie qu'au sujet du contenu du tableau, NumPy peut seulement savoir que ce sont des objets Python. Dans certains usages, ce genre de tableau d'objets s'avère approprié, mais toutes les opérations sur les données vont être réalisées au niveau Python, avec donc un niveau de performances bien moindre que ce que l'on peut espérer avec un type natif :

In[3]:

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
print()
```

```
dtype = object  
10 loops, best of 3: 78.2 ms per loop
```

```
dtype = int  
100 loops, best of 3: 3.06 ms per loop
```

Si vous utilisez des objets Python dans un tableau, et demandez des agrégations avec `sum()` ou `min()` dans un tableau contenant une valeur `None`, vous déclencherez une erreur :

```
In[4]:
```

```
vals1.sum()
```

```
TypeError                                 Traceback (most  
recent call last)  
<ipython-input-4-749fd8ae6030> in <module>()  
----> 1 vals1.sum()
```

```
/Users/jakevdp/anaconda/lib/python3.5/site-  
packages/numpy/core/_methods.py ...
```

```
30
```

```
31 def _sum(a, axis=None, dtype=None,  
out=None, keepdims=False):  
---> 32 return umr_sum(a, axis, dtype, out,  
keepdims)
```

```
33
```

```
34 def _prod(a, axis=None, dtype=None,
```

```
out=None, keepdims=False):  
TypeError: unsupported operand type(s) for +:  
'int' and 'NoneType'
```

Ici, le résultat est simplement lié au fait que l'addition d'une valeur numérique entière à la pseudo-valeur None donne un résultat indéfini.

NaN : données numériques manquantes

La seconde convention de données manquantes dans Pandas correspond à NaN, le non-nombre. Dans ce cas, il s'agit d'une valeur standardisée pour tous les systèmes par l'institut de standardisation IEEE. Cela ne concerne que les valeurs numériques à virgule flottante :

```
In[5]:  
vals2 = np.array([1, np.nan, 3, 4])  
vals2.dtype
```

```
Out[5]:  
dtype('float64')
```

Vous constatez que NumPy a choisi un type flottant natif pour ce genre de tableau. À la différence du tableau de type objet que nous venons de voir, ce tableau permet de réaliser des opérations rapides grâce au code compilé. La pseudo-valeur NaN ressemble un peu à un virus de données, dans le sens où elle infecte tout objet qu'elle touche. Quelle que soit

l'opération, lorsque vous combinez arithmétiquement une valeur à NaN, vous obtenez un NaN :

```
In[6]: 1 + np.nan  
Out[6]: nan
```

```
In[7]: 0 * np.nan  
Out[7]: nan
```

Il en découle qu'un agrégat sur les valeurs reste défini, donc ne déclenche pas d'erreur, mais il n'est pas toujours utile :

```
In[8]:  
vals2.sum(), vals2.min(), vals2.max()
```

```
Out[8]:  
(nan, nan, nan)
```

NumPy a prévu quelques cas d'agrégations spéciaux qui ignorent les valeurs manquantes :

```
In[9]:  
np.nansum(vals2), np.nanmin(vals2),  
np.nanmax(vals2)
```

```
Out[9]:  
(8.0, 1.0, 4.0)
```

N'oubliez jamais que NaN ne concerne que les valeurs à virgule flottante. Il n'y a pas d'équivalent de cette pseudo-valeur pour les types entiers, les chaînes et les autres types.

NaN et None dans Pandas

Les deux pseudo-valeurs NaN et None ont leur intérêt, et Pandas est conçu de sorte de gérer les deux de façon quasiment interchangeable, en assurant une conversion de l'un à l'autre lorsque c'est approprié :

In[10]:

```
pd.Series([1, np.nan, 2, None])
```

Out[10]:

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

Dans le cas d'un type de données pour lequel il n'existe pas de valeur sentinelle, Pandas transtype automatiquement les valeurs manquantes (NA) qu'il rencontre. Si nous forçons par exemple une valeur d'un tableau d'entiers avec np.nan, cette valeur sera automatiquement transtypée vers le type flottant pour pouvoir prendre en compte la valeur NaN :

```
In[11]:  
x = pd.Series(range(2), dtype=int)  
x
```

```
Out[11]:  
0    0  
1    1  
dtype: int32
```

```
In[12]:  
x[0] = None  
x
```

```
Out[12]:  
0    NaN  
1    1.0  
dtype: float64
```

Non seulement Pandas transtype le tableau d'entiers vers un type à virgule flottante, mais il convertit également la pseudo-valeur None vers NaN. (Il est prévu de définir une pseudo-valeur dédiée aux entiers dans Pandas, mais cela n'est pas encore réalisé au moment d'écrire ces lignes.)

Ces conversions pourraient être considérées comme une sorte de bricolage, en comparaison de l'approche plus homogène des valeurs NA dans un langage spécifique tel que R. Pourtant, l'approche basée « sentinelles et transtypage »

de Pandas s'avère bien pratique et de ma propre expérience, entraîne rarement des problèmes.

Le [Tableau 3.2](#) rappelle les conventions de transtypage Pandas pour les valeurs manquantes (NA).

Tableau 3.2 : Gestion des valeurs manquantes par type dans Pandas.

Type	Transtypage si NA	Valeur sentinelle NA
Flottant	Non	np.nan
Objet	Non	None ou np.nan
Entier	Vers float64	np.nan
Booléen	Vers object	None ou np.nan

Rappelons que les données de type chaîne sont toujours stockées avec dtype object dans Pandas.

Opérations avec des valeurs nulles

Nous venons de voir que la librairie Pandas considère les deux pseudo-valeurs None et NaN de façon plus ou moins interchangeable pour marquer les valeurs manquantes ou nulles. Nous disposons de plusieurs méthodes alignées avec cette approche pour détecter, supprimer et remplacer les valeurs nulles dans une structure de données Pandas. Les voici :

`isnull()` Génère un masque booléen pour marquer les valeurs manquantes.

`notnull()` Opposé de `isnull()`.

`dropna()` Renvoie une version des données après filtrage.

`fillna()` Renvoie une copie des données une fois les valeurs manquantes remplacées ou imputées.

Terminons cette section en découvrant ces différentes routines au moyen d'exemples.

Détection de valeurs nulles

Pandas offre deux méthodes pour détecter les valeurs nulles : `isnull()` et `notnull()`. Toutes deux renvoient un masque booléen associé aux données :

In[13]:

```
data = pd.Series([1, np.nan, 'hello', None])
```

In[14]:

```
data.isnull()
```

Out[14]:

```
0    False
1    True
2    False
3    True
dtype: bool
```

Comme indiqué dans la section de début de chapitre consacrée à l'indexation et à la sélection de données, on peut utiliser un masque booléen directement sous forme d'un index pour Series

ou DataFrame :

```
In[15]: data[data.notnull()]
```

```
Out[15]:
```

```
0      1
2    hello
dtype: object
```

L'effet des deux méthodes `isnull()` et `notnull()` est similaire pour les objets DataFrame.

Élimination des valeurs nulles

En complément au masque binaire que nous venons de voir, vous disposez de deux méthodes de soutien `dropna()` (pour supprimer les valeurs NA) et `fillna()` (pour remplacer les pseudo-valeurs NA par des valeurs). Pour une structure Series, le résultat est évident :

```
In[16]: data.dropna()
```

```
Out[16]:
```

```
0    1  
2  hello  
dtype: object
```

En revanche, vous disposez de quelques options pour un objet DataFrame. Partons de celui-ci :

In[17]:

```
df = pd.DataFrame([[1,      np.nan, 2],  
                  [2,      3,      5],  
                  [np.nan, 4,      6]])
```

```
df
```

Out[17]:

```
   0    1  2  
0  1.0  NaN  2  
1  2.0  3.0  5  
2  NaN  4.0  6
```

Avec une structure DataFrame, il n'est pas possible de supprimer une valeur isolée, seulement une ligne ou une colonne complète. Puisque vous devez pouvoir choisir entre les deux ordres d'attaque, dropna() est doté d'un certain nombre d'options dans le cas d'une structure DataFrame.

Par défaut, dropna() abandonne toutes les lignes contenant au moins une valeur nulle :

In[18]: df.dropna()

Out[18] :

	0	1	2
1	2.0	3.0	5

Vous pouvez demander d'oublier les valeurs NA selon un autre axe. Avec axis=1, vous supprimez toutes les colonnes contenant au moins une valeur nulle :

In[19]: `df.dropna(axis='columns')`

Out[19] :

	2
0	2
1	5
2	6

Évidemment, vous écartez du même coup des données valables. Il peut être plus intéressant de ne supprimer de ligne et de colonne entière que si toutes les valeurs sont NA, ou une grande majorité d'entre elles. Vous pouvez le faire en utilisant les paramètres how ou thresh, ce qui permet de contrôler précisément le nombre de valeurs nulles que vous acceptez de conserver.

La valeur par défaut équivaut à `how='any'`. Dans ce cas, toute ligne ou colonne (selon la valeur de `axis`) qui contient une valeur nulle sera abandonnée. Vous pouvez aussi choisir `how='all'` qui va n'éliminer que les lignes ou colonnes qui ne contiennent que des valeurs nulles :

```
In[20]:
```

```
df[3] = np.nan  
df
```

```
Out[20]:
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
In[21]:
```

```
df.dropna(axis='columns', how='all')
```

```
Out[21]:
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

Pour encore plus de précision, vous utiliserez le paramètre `thresh` avec lequel vous pouvez spécifier combien de valeurs non nulles doivent être présentes au minimum dans la ligne ou dans la colonne pour la conserver :

```
In[22]: df.dropna(axis='rows', thresh=3)
```

```
Out[22]:
```

	0	1	2	3
1	2.0	3.0	5	NaN

Dans cet exemple, nous avons écarté la première et la dernière ligne parce qu'elles ne contenaient que deux valeurs non nulles.

Remplacement des valeurs nulles

Il est parfois plus intéressant de remplacer les valeurs manquantes par une valeur acceptable au lieu de les abandonner. Cela peut être une valeur numérique égale à zéro ou bien le fruit d'une interpolation entre des valeurs correctes. L'opération peut être réalisée sur place en utilisant la méthode `isnull()` en tant que masque. Cette opération est très fréquente en Pandas, et nous disposons de la méthode `fillna()` qui renvoie directement une copie du tableau dans lequel toutes les valeurs nulles ont été remplacées.

Partons de la structure Series suivante :

In[23]:

```
data = pd.Series([1, np.nan, 2, None, 3],  
index=list('abcde'))  
data
```

Out[23]:

a	1.0
b	NaN
c	2.0
d	NaN

```
e    3.0  
dtype: float64
```

Voici comment remplacer toutes les entrées NA par la valeur zéro :

```
In[24]: data.fillna(0)
```

```
Out[24]:  
a    1.0  
b    0.0  
c    2.0  
d    0.0  
e    3.0  
dtype: float64
```

Nous pouvons faire un remplissage aval « forward-fill » afin de propager la dernière valeur correcte dans les manquants :

```
In[25]: # Remplissage forward-fill  
data.fillna(method='ffill')
```

```
Out[25]:  
a    1.0  
b    1.0  
c    2.0  
d    2.0  
e    3.0  
dtype: float64
```

Nous pouvons même demander un remplissage amont « back-fill », propageant la prochaine valeur correcte en arrière :

```
In[26]: # Remplissage back-fill  
data.fillna(method='bfill')
```

```
Out[26]:  
a    1.0  
b    2.0  
c    2.0  
d    3.0  
e    3.0  
dtype: float64
```

Les mêmes options sont disponibles pour une structure DataFrame, mais nous pouvons également préciser un axe selon lequel effectuer le remplissage avec axis :

```
In[27]: df
```

```
Out[27]:
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
In[28]:  
df.fillna(method='ffill', axis=1)
```

Out[28]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Sachez que s'il n'y a pas de valeurs correctes précédentes lors d'un remplissage aval, la pseudo-valeur reste en place.

3.6 : Indexation hiérarchique

Pour le moment, nous n'avons traité que des données en une dimension et en deux dimensions, qui étaient stockées respectivement dans des objets Series et DataFrame de Pandas. Il est souvent nécessaire de traiter un plus grand nombre de dimensions, c'est-à-dire d'indexer les données avec plus d'une ou de deux clés. Il est possible dans Pandas d'utiliser les objets Panel et Panel4D pour gérer directement des données en trois et en quatre dimensions (nous le verrons dans la section sur les données de panneaux un peu plus loin). Une autre approche, beaucoup plus usitée, consiste à regrouper plusieurs niveaux d'index dans un seul index, ce qui correspond à l'indexation hiérarchique ou *multi-indexation*. Il devient aussi possible de représenter de façon compacte des données avec un plus grand nombre de dimensions, tout en continuant à utiliser les objets en une dimension Series et en deux dimensions DataFrame.

Nous allons découvrir comment créer directement un objet MultiIndex. Nous verrons comment indexer, trancher et obtenir des statistiques à partir de données à index multiples. Nous verrons quelques routines permettant de convertir les représentations des données entre le format simple et le format hiérarchique des index.

Commençons par demander nos deux importations standard :

```
In[1]:  
import pandas as pd  
import numpy as np
```

Un objet Series à index multiple

Voyons comment nous pourrions représenter des données en deux dimensions au moyen d'un objet Series en une dimension. En guise d'exemple, nous choisissons une série de données pour laquelle chaque point possède une chaîne de caractères et une clé numérique.

Une mauvaise approche

Imaginons que nous ayons besoin d'étudier les données de plusieurs états des USA pour deux années différentes. Nous pourrions essayer d'utiliser les outils Pandas déjà connus en utilisant des tuples Python en tant que clés :

```
In[2]:  
index = [('California', 2000), ('California',  
2010),  
        ('New York', 2000), ('New York', 2010),  
        ('Texas', 2000), ('Texas', 2010)]  
populations = [33871648, 37253956,
```

```
18976457, 19378102,  
20851820, 25145561]  
pop = pd.Series(populations, index=index)  
pop
```

Out[2]:

```
(California, 2000)    33871648  
(California, 2010)    37253956  
(New York, 2000)     18976457  
(New York, 2010)     19378102  
(Texas, 2000)        20851820  
(Texas, 2010)        25145561  
dtype: int64
```

Avec ce mode d'indexation, on peut aisément manipuler les index et trancher les données en fonction de cet index multiple :

In[3]:

```
pop['California', 2010]:('Texas', 2000)]
```

Out[3]:

```
(California, 2010)    37253956  
(New York, 2000)     18976457  
(New York, 2010)     19378102  
(Texas, 2000)        20851820  
dtype: int64
```

Mais c'est tout ce que l'on peut faire. Si vous avez besoin de sélectionner toutes les valeurs de 2010, vous êtes forcé

d'écrire une opération de sélection complexe et peu performante :

In[4]:

```
pop[[i for i in pop.index if i[1] == 2010]]
```

Out[4]:

```
(California, 2010)    37253956  
(New York, 2010)     19378102  
(Texas, 2010)        25145561  
dtype: int64
```

C'est le résultat désiré, mais il ne sera pas réalisé de façon suffisamment efficace dans le cas d'un grand volume de données. C'est moins intéressant que la syntaxe de tranchage que nous aimons utiliser dans Pandas.

Une meilleure approche : MultiIndex de Pandas

La librairie Pandas offre une autre approche. Notre index basé sur des tuples est une sorte de multi-index rudimentaire. Le type MultiIndex de Pandas nous permet de profiter du genre d'opérations que nous recherchons. Voici comment nous pouvons à partir des tuples créer un multi-index :

In[5]:

```
index = pd.MultiIndex.from_tuples(index)
```

index

Out[5]:

```
MultiIndex(levels=[['California', 'New York',
 'Texas'], [2000, 2010]],
            labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0,
 1, 0, 1]])
```

Vous remarquez que l'objet MultiIndex contient plusieurs niveaux d'indexation levels, le nom d'état state et les années, ainsi que plusieurs labels pour chaque point de données.

Nous accédons à la représentation hiérarchique des données si nous réindexons les séries avec cet objet MultiIndex :

In[6]:

```
pop = pop.reindex(index)
pop
```

Out[6]:

California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

```
dtype: int64
```

Les deux premières colonnes de la représentation de Series montrent les valeurs des index multiples, la troisième colonne montrant les données. Vous remarquez que la première colonne est parfois vide, ce qui correspond ici à un contenu identique à la ligne précédente.

Si nous cherchons à accéder à toutes les données pour lesquelles le second index vaut 2010, nous pouvons nous servir de la notation de tranchage Pandas :

```
In[7]: pop[:, 2010]
```

```
Out[7]:
```

```
California    37253956
New York      19378102
Texas          25145561
dtype: int64
```

Nous obtenons un tableau à index unique ne contenant que les clés recherchées. L'opération est beaucoup plus efficace que la solution de multi-indexation basée tuples vue au départ, et beaucoup plus pratique à utiliser. Entrons maintenant dans les détails des opérations d'indexation sur des données à indexation hiérarchique.

MultIndex en tant que dimension supplémentaire

Vous avez peut-être noté que nous aurions pu obtenir le même stockage de données avec un objet DataFrame doté d'index et de labels de colonnes. Il est vrai que Pandas a été conçu en sorte de préserver cette équivalence. Vous pouvez convertir un objet Series à index multiple en un objet DataFrame à index conventionnel au moyen de la méthode `unstack()` :

In[8]:

```
pop_df = pop.unstack()  
pop_df
```

Out[8]:

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

Vous pouvez bien sûr réaliser l'opération inverse avec la méthode `stack()` :

In[9]:

```
pop_df.stack()
```

Out[9]:

```
California 2000    33871648
```

```
    2010    37253956  
New York   2000    18976457  
              2010    19378102  
Texas      2000    20851820  
              2010    25145561  
dtype: int64
```

Considérant la simplicité de ces deux opérations, vous vous demandez peut-être pourquoi nous nous embêtons à réaliser des indexations hiérarchiques. La raison est fort simple : nous avons pu représenter des données en deux dimensions sous la forme d'un objet en une dimension Series, et nous pouvons donc représenter des données à trois ou plus dimensions dans un objet Series ou DataFrame. Chaque niveau supplémentaire d'un multi-index incarne une dimension supplémentaire des données. Cette relation organique nous offre bien plus de souplesse au niveau des types de données représentables. Nous pouvons tout à fait ajouter une colonne de données démographiques pour chaque état et année, par exemple pour la seule population âgée de moins de 18 ans. Avec un objet MultiIndex, cela se résume à ajouter une colonne à l'objet DataFrame :

```
In[10]:  
pop_df = pd.DataFrame({'total': pop,  
                      'under18': [9267089,  
9284094,  
4687374,
```

```
4318033,  
5906301,  
6879014])  
pop_df
```

Out[10]:

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

De plus, toutes les fonctions universelles ufuncs et commandes spécifiques vues en début de chapitre dans la section sur le traitement des données Pandas sont applicables aux index hiérarchiques. Voici par exemple comment trouver le pourcentage de mineurs par année :

In[11]:

```
f_u18 = pop_df['under18'] / pop_df['total']  
f_u18.unstack()
```

Out[11]:

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

Nous pouvons donc facilement et rapidement manipuler et analyser des données à grand nombre de dimensions.

Méthode de création de MultiIndex

La façon la plus simple de construire un objet Series ou DataFrame à index multiple consiste à lui transmettre une liste de deux tableaux d'index ou plus à destination du constructeur :

In[12]:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[[ 'a', 'a', 'b', 'b'],
                          [1, 2, 1, 2]],
                  columns=[ 'data1', 'data2'])
df
```

Out[12]:

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

La création proprement dite de MultiIndex est réalisée en coulisses.

Si vous transmettez un dictionnaire dans lequel les clés sont les tuples appropriés, Pandas va détecter automatiquement

ce format et sélectionner un MultiIndex par défaut :

In[13]:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

Out[13]:

```
California 2000 33871648
            2010 37253956
New York   2000 18976457
            2010 19378102
Texas      2000 20851820
            2010 25145561
dtype: int64
```

Il reste cependant parfois nécessaire de créer le multi-index de façon explicite. Voyons quelques méthodes permettant d'y parvenir.

Constructeur explicite de multi-index

Pour contrôler en détail la construction d'un index, vous pouvez vous servir des constructeurs de classe définis dans l'objet pd.MultiIndex. Nous pouvons par exemple construire

un MultiIndex à partir d'une liste de tableaux, en définissant les valeurs d'index dans chaque niveau :

In[14]:

```
pd.MultiIndex.from_arrays([[ 'a', 'a', 'b', 'b'],
[1, 2, 1, 2]])
```

Out[14]:

```
MultiIndex([( 'a', 1),
( 'a', 2),
( 'b', 1),
( 'b', 2)],
)
```

Vous pouvez évidemment partir d'une liste de tuples qui donne les valeurs d'index multiple pour chaque point :

In[15]:

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2),
('b', 1), ('b', 2)])
```

Out[15]:

```
MultiIndex([( 'a', 1),
( 'a', 2),
( 'b', 1),
( 'b', 2)],
)
```

Vous obtenez le même résultat en partant du produit cartésien des index simples :

In[16]:

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

Out[16]:

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

Vous pouvez enfin construire un multi-index directement en vous conformant à son codage interne. Il suffit de lui transmettre un paramètre levels qui est la liste des listes contenant les valeurs d'index pour chaque niveau et un paramètre code qui est la liste des listes qui font référence à ces niveaux :

In[17]:

```
pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
               codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Out[17]:

```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

N'importe lequel de ces objets peut être transmis en tant que paramètre index pour créer un objet Series ou DataFrame, ou à la méthode reindex d'un tel objet existant.

Noms des niveaux de MultiIndex

Il est dans certains cas judicieux de pouvoir donner des noms aux différents niveaux de multi-index, ce qui se réalise facilement en fournissant le paramètre names à un constructeur de multi-index ou en définissant l'attribut names de l'index après coup :

In[18]:

```
pop.index.names = ['region', 'an']
pop
```

Out[18]:

```
region      an
California  2000    33871648
              2010    37253956
New York    2000    18976457
              2010    19378102
Texas       2000    20851820
              2010    25145561
dtype: int64
```

Lorsque le jeu de données comporte de nombreux index, cela permet plus facilement de garder le contrôle sur la signification de chaque index.

MultIndex de colonnes

Dans un objet DataFrame, les lignes et les colonnes sont absolument symétriques. Plusieurs niveaux d'indices sont possibles aussi bien pour les lignes que pour les colonnes. Partons d'un petit exemple de données de santé avec fréquence cardiaque et température corporelle :

```
In[19]:  
# Index et colonnes hiérarchisés  
index = pd.MultiIndex.from_product([[2013,  
2014], [1, 2]],  
                                     names=['an',  
'visite'])  
columns = pd.MultiIndex.from_product([['Robert',  
'Guy', 'Sophie'],  
                                      ['Cardio',  
'Temp']],  
                                     names=  
['sujet', 'type'])  
# Données de simulation  

```

```
Out[19]:
```

		Robert		Guy		
sujet		Cardio	Temp	Cardio	Temp	
Sophie						
type						
Cardio	Temp					
an	visite					
2013	1	22.0	37.4	41.0	36.2	44.0
	35.0					
	2	24.0	37.5	39.0	37.3	32.0
	35.3					
2014	1	43.0	39.7	52.0	37.1	49.0
	37.9					
	2	36.0	36.4	17.0	38.1	25.0
	38.1					

Les multi-index pour les lignes et pour les colonnes se montrent très pratiques. Il s'agit effectivement ici de données à quatre dimensions : le patient ou sujet, le type de mesure, l'année et le numéro de visite. Nous pouvons ensuite indexer la colonne de plus haut niveau selon le nom du patient et obtenir un objet DataFrame complet, uniquement pour cette personne :

```
In[20]: health_data['Guy']
```

```
Out[20]:
```

		Cardio	Temp
type		Cardio	Temp
an	visite		
2013	1	26.0	37.2
	2	37.0	37.9

```
2014      1    32.0    37.8  
          2    50.0    36.2
```

L'indexation hiérarchique des lignes et des colonnes peut se montrer extrêmement utile dans le cas des enregistrements complexes avec plusieurs séries de données nommées, à plusieurs reprises et pour plusieurs sujets d'intérêt (personnes, pays, villes, *etc.*).

Indexation et tranchage d'un MultiIndex

Les opérations d'indexation et de tranchage d'un objet MultiIndex ont été conçues afin d'être intuitives. Vous pouvez considérer les indices comme des dimensions complémentaires. Voyons comment indexer d'abord un objet Series multi-index, puis un objet DataFrame.

Objets Series multi-indexés

Repartons de l'objet Series multi-indexé des populations de plusieurs États américains :

```
In[21]: pop  
Out[21]:  
region      an  
California  2000    33871648  
              2010    37253956
```

```
New York    2000    18976457  
              2010    19378102  
Texas      2000    20851820  
              2010    25145561  
dtype: int64
```

Pour accéder à un élément, nous pouvons demander une indexation multiple :

```
In[22]: pop['California', 2000]
```

```
Out[22]: 33871648
```

MultiIndex permet l'indexation partielle, c'est-à-dire sur un seul des niveaux d'index. Cela produit un autre objet Series, dans lequel n'y est maintenu que l'indice de plus bas niveau :

```
In[23]: pop['California']
```

```
Out[23]:  
an  
2000    33871648  
2010    37253956  
dtype: int64
```

Pour un tranchage partiel, il est nécessaire que l'objet MultiIndex soit trié d'abord (nous reparlons du tri des index un peu plus loin) :

```
In[24]: pop.loc['California':'New York']
```

```
Out[24]:
```

```
region      an
California  2000    33871648
              2010    37253956
New York    2000    18976457
              2010    19378102
dtype: int64
```

Une fois les index triés, nous demandons l'indexation partielle sur le niveau le plus bas en transmettant comme premier index une tranche vide :

```
In[25]: pop[:, 2000]
```

```
Out[25]:
```

```
region
California    33871648
New York      18976457
Texas          20851820
dtype: int64
```

Vous avez également accès à d'autres types d'indexation et de sélection (nous les avons présentés dans la section sur l'indexation des objets en début de chapitre). Voici par exemple une sélection à partir d'un masque booléen :

```
In[26]: pop[pop > 22000000]
```

```
Out[26]:
```

```
region      an
California  2000    33871648
              2010    37253956
Texas       2010    25145561
dtype: int64
```

Vous pouvez enfin faire une sélection avec une indexation fancy :

```
In[27]: pop[['California', 'Texas']]
```

```
Out[27]:
```

```
region      an
California  2000    33871648
              2010    37253956
Texas       2000    20851820
              2010    25145561
dtype: int64
```

Objets DataFrame multi-indexés

Le raisonnement est similaire pour un objet multi-indexé DataFrame. Reprenons notre jeu de données de santé :

```
In[28]: health_data
```

```
Out[28]:
```

	sujet	Robert		Guy		Sophie
	type	Cardio	Temp	Cardio	Temp	
	Cardio	Temp				
	an	visite				
2013	1	43.0	37.3	26.0	37.2	59.0
	37.1					
	2	33.0	37.1	37.0	37.9	35.0
	36.8					
2014	1	31.0	36.1	32.0	37.8	39.0
	36.9					
	2	33.0	35.8	50.0	36.2	10.0
	37.6					

Nous n'oublions pas que les colonnes sont primaires dans un objet DataFrame. La syntaxe valable pour un objet Series multi-index s'applique donc aux colonnes. Voici par exemple comment récupérer la fréquence cardiaque de Guy avec une simple opération :

```
In[29]: health_data['Guy', 'Cardio']
```

```
Out[29]:
```

	an	visite
2013	1	32.0
	2	50.0
2014	1	39.0
	2	48.0

```
Name: (Guido, HR), dtype: float64
```

Comme dans le code à un seul index, nous pouvons nous servir des indexeurs loc, iloc et ix présentés dans la section antérieure sur la sélection et l'indexation. Voici un exemple :

```
In[30]: health_data.iloc[:2, :2]
```

```
Out[30]:
```

```
sujet      Robert
type      Cardio  Temp
an   visite
2013  1      39.0  38.5
      2      41.0  39.4
```

Grâce à ces indexeurs, vous obtenez une vue de type tableau des données à deux dimensions, mais chaque index individuel dans loc ou dans iloc peut recevoir un tuple d'index multiples, comme ceci :

```
In[31]: health_data.loc[:, ('Robert', 'Cardio')]
```

```
Out[31]:
```

```
an   visite
2013  1      39.0
      2      41.0
2014  1      24.0
      2      34.0
Name: (Robert, Cardio), dtype: float64
```

Ceci dit, ce n'est pas très pratique de travailler avec des tranches dans ces tuples d'index. D'ailleurs, si vous tentez de créer une tranche dans un tuple, vous aurez une erreur de syntaxe :

```
In[32]: health_data.loc[:, :, 'Cardio'])
```

```
File "<ipython-input-32-c64acebd2d69>", line 1
    health_data.loc[:, :, 'Cardio')]
          ^
SyntaxError: invalid syntax
```

Vous pourriez contourner le problème en demandant de générer d'abord la tranche explicitement au moyen de la fonction slice() de Python. Dans ce contexte, mieux vaut utiliser un objet IndexSlice que Pandas définit exactement à cette intention. Voici un exemple :

```
In[33]:
```

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'Cardio']]
```

```
Out[33]:
```

sujet	Robert	Guy	Sophie
type	Cardio	Cardio	Cardio
an	visite		
2013	1	39.0	32.0
2014	1	24.0	32.0
			52.0

Les possibilités d’interaction avec les données dans un objet Series ou DataFrame multi-indexé sont énormes. Comme de nombreux autres outils présentés dans ce livre, le mieux pour en acquérir la maîtrise consiste à les essayer !

Réorganisation d’un multi-index

Pour bien exploiter des données multi-indexées, il faut savoir comment transformer ces données de façon efficace. Un certain nombre d’opérations préservent les informations présentes, tout en les réorganisant pour les préparer à différents traitements. Nous avons déjà rencontré deux méthodes, stack() et unstack(), mais il existe bien d’autres moyens pour contrôler précisément l’organisation des données au niveau des index hiérarchiques et des colonnes. Découvrons ces techniques.

Index triés et non triés

Nous avions déjà réalisé cette mise en garde plus haut, mais il est nécessaire de la répéter. Un grand nombre d’opérations de tranchage sur multi-index échouent si l’index n’est pas trié. Voyons cela en détail.

Nous commençons par créer un lot de données multi-indexées dans lequel les index ne sont pas triés en termes lexicaux :

```
In[34]:  
index = pd.MultiIndex.from_product([('a', 'c',  
'b'), [1, 2]])  
data = pd.Series(np.random.rand(6), index=index)  
data.index.names = ['char', 'int']  
data
```

```
Out[34]:  
char int  
a    1    0.185229  
      2    0.352673  
c    1    0.274921  
      2    0.076575  
b    1    0.562786  
      2    0.124449  
dtype: float64
```

Nous allons déclencher une erreur si nous essayons d'obtenir une tranche partielle de cet index :

```
In[35]:  
try:  
    data['a':'b']  
except KeyError as e:  
    print(type(e))  
    print(e)  
  
<class 'KeyError'>  
'Key length (1) was greater than MultiIndex  
lexsort depth (0)'
```

Le message d'erreur indique clairement que c'est un problème d'index non trié. Les opérations de tri partiel (parmi d'autres) ont besoin que les niveaux d'index soient triés. Heureusement, Pandas propose des routines permettant d'y parvenir. Pour l'objet DataFrame, nous disposons par exemple de `sort_index()` et de `sortlevel()`. Voyons comment utiliser la première, `sort_index()`:

```
In[36]:
```

```
data = data.sort_index()  
data
```

```
Out[36]:
```

```
char int  
a    1    0.185229  
      2    0.352673  
b    1    0.562786  
      2    0.124449  
c    1    0.274921  
      2    0.076575  
dtype: float64
```

Une fois l'index trié, le tranchage partiel fonctionne comme prévu :

```
In[37]: data['a':'b']
```

```
Out[37]:  
char int  
a    1    0.185229  
      2    0.352673  
b    1    0.562786  
      2    0.124449  
dtype: float64
```

Empilement et dépilement d'index

Nous avons vu un peu plus haut que l'on pouvait convertir un jeu de données du format multi-indexé empilé vers une représentation plus simple en deux dimensions, en précisant éventuellement le niveau à utiliser :

```
In[38]: pop.unstack(level=0)
```

```
Out[38]:  
region    California   New York      Texas  
an  
2000        33871648  18976457  20851820  
2010        37253956  19378102  25145561
```

```
In[39]: pop.unstack(level=1)
```

```
Out[39]:
```

```
an          2000          2010  
region  
California  33871648  37253956
```

```
New York    18976457    19378102  
Texas      20851820    25145561
```

La méthode complémentaire de `unstack()` se nomme `stack()`. Elle permet de retrouver la série de départ :

```
In[40]: pop.unstack().stack()
```

```
Out[40]:
```

```
region      an  
California  2000    33871648  
              2010    37253956  
New York   2000    18976457  
              2010    19378102  
Texas      2000    20851820  
              2010    25145561  
dtype: int64
```

set_index et reset_index

Une autre façon de réorganiser des données hiérarchiques consiste à transformer les labels d'index en colonnes au moyen de la méthode `reset_index`. Si nous appliquons cette technique à notre dictionnaire de population, nous obtenons un objet `DataFrame` dans lequel les deux colonnes pour la région (`state`) et pour l'année contiennent les informations qui se trouvaient dans l'index. Nous pouvons en option indiquer le nom des données pour la représentation en colonnes :

```
In[41]:
```

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

```
Out[41]:
```

	region	an	population
0	California	2000	33871648
1	California	2010	37253956
2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

Lorsque vous devez traiter des données du monde réel, vous partirez souvent de données d'entrées brutes se présentant sous ce format. Il est dans ce cas très utile de pouvoir construire un multi-index à partir des valeurs des colonnes au moyen de la méthode `set_index` de l'objet DataFrame qui renvoie un objet multi-indexé :

```
In[42]: pop_flat.set_index(['region', 'an'])
```

```
Out[42]:
```

		population
region	an	
California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102

Texas	2000	20851820
	2010	25145561

Je trouve que ce genre de réindexation constitue l'une des techniques les plus utiles qui soient pour bien prendre en compte un jeu de données du monde réel.

Agrégation de données sur multi-index

Nous avons vu que Pandas offrait des méthodes d'agrégation de données telles que `mean()`, `sum()` et `max()`. Lorsque vous traitez des données à indexation hiérarchique, vous pouvez transmettre à ces méthodes le paramètre `level` pour choisir quel sous-ensemble de données doit être traité par l'opération d'agrégation.

Repartons de notre petit jeu de données de santé :

In[43]: `health_data`

Out[43]:

sujet	Robert	Guy
Sophie		
type	Cardio	Temp
Cardio	Cardio	Temp
Temp		
an	visite	
2013	1	43.0 37.3 26.0 37.2
		59.0 37.1

```
      2          33.0  37.1      37.0  37.9  
35.0  36.8  
2014   1          31.0  36.1      32.0  37.8  
39.0  36.9  
      2          33.0  35.8      50.0  36.2  
10.0  37.6
```

Cherchons à obtenir la moyenne des valeurs pour les deux visites annuelles. Il suffit de donner le nom du niveau d'index à explorer, qui est ici l'année :

In[44]:

```
data_mean = health_data.mean(level='an')  
data_mean
```

Out[44]:

	Robert		Guy		Sophie
sujet					
type	Cardio	Temp	Cardio	Temp	Cardio
Temp					
an					
2013	38.0	37.20	31.5	37.55	47.0
36.95					
2014	32.0	35.95	41.0	37.00	24.5
37.25					

Au moyen du mot-clé axis, nous pouvons obtenir la moyenne parmi les niveaux des colonnes :

In[45]: `data_mean.mean(axis=1, level='type')`

Out[45] :

```
type      Cardio      Temp  
an  
2013      38.833333    37.233333  
2014      32.500000    36.733333
```

Autrement dit, deux lignes ont suffi à trouver le rythme cardiaque et la température moyenne pour tous les patients de toutes les visites par année. Cette façon d'écrire est une version abrégée de ce que permet le mécanisme GroupBy qui sera décrit dans la section sur l'agrégation et le groupement plus loin dans ce chapitre. Cet exemple est très modeste, mais de nombreux jeux de données du monde réel présentent ce genre de structure hiérarchique.

Données de panneaux

Pandas offre deux autres structures de données fondamentales dont nous n'avons pas parlé et qui se nomment pd.Panel et pd.Panel4D. Ce sont des généralisations respectivement à trois et à quatre dimensions des objets à une dimension Series et à deux dimensions DataFrame. Une fois que vous connaissez les opérations d'indexation et de manipulation pour les objets Series et DataFrame, les opérations pour pd.Panel et pd.Panel4D sont simples à utiliser. Vous

pouvez par exemple utiliser directement les indexeurs ix, loc et iloc déjà rencontrés.

Nous ne décrirons pas ces structures de type panneau, car j'ai constaté qu'en général la multi-indexation s'avérait plus utile et offrait des représentations plus simples pour concevoir des données dans un plus grand nombre de dimensions. En outre, les données de panneaux sont des représentations denses des données alors que la multi-indexation est une représentation allégée ou éparse. Lorsque le nombre de dimensions augmente, une représentation dense s'avère très inefficace dans la majorité des cas. Ceci dit, ce genre de structure peut s'avérer utile dans des cas particuliers. Pour en savoir plus à leur sujet, voyez les références fournies en toute fin de chapitre.

3.7 : Combinaison de jeux de données avec concat et append

Une des approches les plus fertiles en matière de datalogie consiste à combiner plusieurs sources de données. Il peut s'agir d'une simple fusion ou concaténation de deux jeux ou d'une séquence sophistiquée d'opérations de jointure et de fusion dans le style des actions habituelles dans une base de données, avec gestion correcte des éventuels chevauchements. Les deux objets Series et DataFrame sont aptes à une exploitation dans cet état d'esprit. La librairie Pandas offre des fonctions et des méthodes permettant d'effectuer ce genre de traitement vite et sans hésitation.

Commençons par une concaténation simple d'objets Series et DataFrame au moyen de la fonction pd.concat. Nous verrons dans un deuxième temps des opérations de fusion et de jointure mémoire sophistiquées.

Nous commençons bien sûr par les directives d'importation habituelles :

```
In[1]:  
import pandas as pd  
import numpy as np
```

Pour plus de confort, nous définissons une fonction pour créer un objet DataFrame que nous pourrons utiliser :

In[2] :

```
def make_df(cols, ind):
    """Crée un objet d'étude DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# DataFrame d'étude
make_df('ABC', range(3))
```

Out[2] :

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

Pour plus de confort, nous définissons une classe d'affichage `display` qui permet de montrer plusieurs tableaux DataFrame résultants côté à côté dans le calepin Jupyter. Le code exploite la méthode spéciale `_repr_html_` dont IPython se sert pour ses propres affichages.

```
class display(object):
    """Affiche le HTML de plusieurs objets côté à côté"""
```

```

template = """<div style="float: left;
padding: 10px;">
<p style='font-family:"Courier New", Courier,
monospace'>{0}</p>{1}
</div>"""
def __init__(self, *args):
    self.args = args
def __repr_html__(self):
    return '\n'.join(self.template.format(a,
eval(a).__repr_html__())
                  for a in self.args)

def __repr__(self):
    return '\n\n'.join(a + '\n' + repr(eval(a))
                      for a in self.args)

```

Si vous travaillez sur la ligne de commande hors accès au HTML, remplacez dans la suite les appels à `display()` par des appels à `print()`.

Un rappel sur la concaténation des tableaux NumPy

L'opération de concaténation d'objets Series ou DataFrame ressemble beaucoup à celle des tableaux NumPy. Nous avions vu dans le chapitre précédent qu'il suffisait d'utiliser la fonction `np.concatenate`. Rappelons que nous pouvions

ainsi produire un tableau unique à partir de deux tableaux ou plus :

```
In[4]:  
x = [1, 2, 3]  
y = [4, 5, 6]  
z = [7, 8, 9]  
np.concatenate([x, y, z])
```

```
Out[4]:  
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Le premier paramètre doit être une liste ou un tuple de tableaux. Un paramètre facultatif axis permet de choisir l'axe selon lequel l'opération doit être réalisée :

```
In[5]:  
x = [[1, 2],  
     [3, 4]]  
np.concatenate([x, x], axis=1)
```

```
Out[5]:  
array([[1, 2, 1, 2],  
      [3, 4, 3, 4]])
```

Concaténation simple avec pd.concat()

Pandas offre la fonction pd.concat() dont l'utilisation est proche de celle de np.concatenate, hormis un certain nombre d'options que nous verrons plus loin et que voici :

```
# Signature allégée de la fonction
pd.concat(objs, axis=0, join='outer',
           ignore_index=False,
           keys=None, levels=None, names=None,
           verify_integrity=False, sort=False,
           copy=True)
```

Cette fonction permet de réaliser des concaténations simples pour des objets Series ou DataFrame :

In[6]:

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Out[6]:

1	A
2	B
3	C
4	D

```
5      E  
6      F  
dtype: object
```

Vous pouvez également concaténer des objets ayant plus de dimensions, par exemple des objets DataFrame :

In[7] :

```
df1 = make_df('AB', [1, 2])  
df2 = make_df('AB', [3, 4])  
display('df1', 'df2', 'pd.concat([df1, df2])')
```

Out[7] :

```
df1  
A   B  
1  A1  B1  
2  A2  B2
```

```
df2  
A   B  
3  A3  B3  
4  A4  B4
```

```
pd.concat([df1, df2])  
A   B  
1  A1  B1  
2  A2  B2  
3  A3  B3  
4  A4  B4
```

L'opération est réalisée par ligne dans l'objet DataFrame, sauf mention contraire (ce qui équivaut à axis=0). Vous pouvez spécifier un autre axe de concaténation comme dans l'exemple suivant :

In[8] :

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4],
axis='columns')")
```

Out[8] :

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

```
pd.concat([df3, df4], axis='columns')
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Nous aurions pu indiquer explicitement axis=1, mais l'expression axis='columns' est plus intuitive.

Index dupliqués

Une différence essentielle entre np.concatenate et pd.concat est liée au fait que la concaténation dans Pandas maintient les index, même s'ils se retrouvent en double dans le résultat ! Partons de cet exemple simple :

In[9] :

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index # Avec index dupliqués!
display('x', 'y', 'pd.concat([x, y])')
```

Out[9] :

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y])

	A	B
0	A0	B0

```
1    A1    B1
0    A2    B2
1    A3    B3
```

Vous constatez que les index sont répétés dans le résultat. Ceci dit, il est souvent interdit de laisser des doublons dans les données au niveau des index. Il y a plusieurs solutions pour se conformer à cette attente.

Déclencher une erreur en cas de répétition. Pour garantir que les résultats d'une concaténation ne contiennent pas de doublons, il suffit d'activer l'indicateur `verify_integrity`. Si vous lui donnez la valeur `True`, une exception sera déclenchée dès qu'un index apparaîtra en double. Dans l'exemple suivant, nous capturons l'erreur et affichons un message :

```
In[10]:
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values:
Int64Index([0, 1], dtype='int64')
```

Ignorer l'index. Dans certains cas, la valeur de l'index d'entrée n'a pas d'importance, et vous acceptez de l'ignorer en produisant un nouveau. Il suffit d'utiliser l'indicateur

`ignore_index` en lui donnant la valeur `True`. Le résultat va contenir un nouvel index entier pour l'objet `Series` résultant :

In[11]:

```
display('x', 'y', 'pd.concat([x, y],  
ignore_index=True)')
```

Out[11]:

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y], ignore_index=True)
```

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Ajouter des clés multi-index. Une autre approche consiste à utiliser l'option `keys` avec laquelle vous spécifiez un label

pour les sources de données. Le résultat est un index hiérarchisé :

```
In[12]:  
display('x', 'y', "pd.concat([x, y], keys=['x',  
'y']))")
```

```
Out[12]:
```

```
x  
      A      B  
0    A0    B0  
1    A1    B1
```

```
y  
      A      B  
0    A2    B2  
1    A3    B3
```

```
pd.concat([x, y], keys=['x', 'y'])  
      A      B  
x  0  A0  B0  
   1  A1  B1  
y  0  A2  B2  
   1  A3  B3
```

Nous obtenons un objet DataFrame à index multiples. Les données peuvent ensuite être transformées selon vos

besoins au moyen des outils décrits dans la section de ce chapitre dédiée à l'indexation hiérarchique.

Concaténation avec jointures

Les exemples que nous venons de voir se limitent à une concaténation d'objets DataFrame dont les noms de colonnes sont identiques. En réalité, les noms des colonnes ne vont pas coïncider entre plusieurs sources de données. pd.concat offre plusieurs options pour gérer cette situation. Voici comment concaténer deux objets DataFrame dont seules certaines colonnes sont identiques :

In[13]:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

Out[13]:

```
df5
   A   B   C
1  A1  B1  C1
2  A2  B2  C2
```

```
df6
   B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

```
pd.concat([df5, df6])
```

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

Par défaut, les entrées pour lesquelles il n'y a pas de données reçoivent la pseudo-valeur NA. Il est possible de choisir un autre comportement au moyen des paramètres join et join_axes. Par défaut, la jointure correspond à une union des colonnes d'entrée (join='outer'), mais vous pouvez demander une intersection des colonnes avec join='inner' :

In[14]:

```
display('df5', 'df6', "pd.concat([df5, df6],  
join='inner'))")
```

Out[14]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6], join='inner')  
      B   C  
1   B1  C1  
2   B2  C2  
3   B3  C3  
4   B4  C4
```

On peut aussi choisir quelles colonnes doivent faire partie du résultat au moyen du paramètre `join_axes` à qui l'on fournit une liste d'objets `index`. Dans l'exemple suivant, nous voulons obtenir en sortie les données des colonnes existant dans la première source de données :

```
In[15]:  
df7 = pd.concat([df5, df6])  
df7 = df7.reindex(df5.index)  
display('df5', 'df6', 'df7')
```

Out[15]:

```
df5  
      A   B   C  
1   A1  B1  C1  
2   A2  B2  C2
```

```
df6  
      B   C   D  
3   B3  C3  D3  
4   B4  C4  D4
```

```
df7
```

```
A   B   C   D  
1  A1  B1  C1  NaN  
2  A2  B2  C2  NaN
```

En combinant les options de la fonction pd.concat, on dispose donc d'un vaste choix de traitements pour joindre deux jeux de données. N'hésitez pas à y avoir recours.

La méthode append()

La concaténation directe de tableaux est une opération courante. C'est pourquoi les objets Series et DataFrame définissent la méthode append() qui permet d'y parvenir plus rapidement. Au lieu d'appeler pd.concat([df1, df2]), vous pouvez dorénavant écrire df1.append(df2) :

In[16]:

```
display('df1', 'df2', 'df1.append(df2)')
```

Out[16]:

```
df1  
A   B  
1  A1  B1  
2  A2  B2
```

```
df2  
A   B  
3  A3  B3  
4  A4  B4
```

```
df1.append(df2)
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Vous savez que la méthode `append()` des listes Python et sa collègue `extend()` modifient l'objet originel. Ce n'est pas le cas de la méthode `append()` de Pandas. Elle crée un nouvel objet. La méthode n'est pas très efficace, parce qu'elle oblige à créer un nouvel index et un nouveau tampon de données. Lorsque vous avez besoin de réaliser plusieurs opérations d'ajout, il est préférable de construire une liste d'objets DataFrame puis de les transmettre tous ensemble à la fonction `concat()`.

Nous allons maintenant découvrir une autre approche plus puissante encore pour combiner les données de plusieurs sources. Il s'agit des fusions et des jointures dans le style des bases de données que propose la fonction `pd.merge`. Rappelons que pour tout autre détail au sujet des fonctions de cette section, vous disposez de la documentation de Pandas.

3.8 : Combinaison de jeux de données avec merge et join

La librairie Pandas se distingue notamment par ses opérations de fusion et de jointure en mémoire à hautes performances. Ce genre de traitement de données vous semblera familier si vous avez déjà utilisé une base de données relationnelle. Les traitements correspondants se fondent principalement sur la fonction pd.merge. Voyons cela par quelques exemples.

In[1]:

```
import pandas as pd  
import numpy as np
```

```
class display(object):  
    # etc. Nous ne répétons pas la définition de  
    # cette classe
```

Algèbre relationnelle

Le comportement qui est implémenté dans pd.merge est un sous-ensemble de l'algèbre relationnelle. Il s'agit d'un jeu formel de règles pour manipuler des données structurées au format relationnel. Il constitue la fondation théorique des opérations applicables dans la plupart des bases de données.

La grande force de cette algèbre est liée au fait qu'elle propose quelques opérations primitives qui deviennent des blocs opératoires permettant de construire des opérations complexes sur un jeu de données. Lorsqu'une série d'opérations fondamentales a été correctement et efficacement incorporée à un moteur de base de données, il devient possible de réaliser des opérations composites sophistiquées.

La librairie Pandas a incorporé plusieurs de ces blocs élémentaires dans la fonction pd.merge et dans la méthode join des objets Series et DataFrame. Nous verrons que cela permet d'établir efficacement des liens entre différentes sources.

Catégorie de jointures

La fonction pd.merge propose trois types de jointures : univers-un, plusieurs-vers-un et plusieurs-vers-plusieurs. Les trois opérations utilisent le même appel à l'interface pd.merge, le type de jointure réalisé ne dépendant que du format des données d'entrée. Voyons un exemple de chacun de ces trois types d'opérations ; nous verrons ensuite quelques options.

Jointure un-vers-un

L'opération la plus simple est la jointure un-vers-un, qui ressemble beaucoup à l'opération de concaténation par colonnes que nous avons vue dans la section précédente décrivant concat et append. Partons en guise d'exemple de deux objets DataFrame contenant des informations au sujet de plusieurs salariés d'une entreprise :

In[2]:

```
df1 = pd.DataFrame({'employé': ['Robert',
 'Jake', 'Lise', 'Sophie'],
 'groupe': ['Compta',
 'Ingénierie', 'Ingénierie', 'RH']})
df2 = pd.DataFrame({'employé': ['Lise',
 'Robert', 'Jake', 'Sophie'],
 'dt_embau': [2004, 2008,
 2012, 2014]})
display('df1', 'df2')
```

Out[2]:

```
df1
    employé      groupe
0   Robert      Compta
1     Jake  Ingénierie
2     Lise  Ingénierie
3   Sophie        RH
```

```
df2
```

```
employé dt_embau
0      Lise      2004
1    Robert      2008
2      Jake      2012
3   Sophie      2014
```

Pour obtenir un seul objet DataFrame, nous nous servons de notre fonction pd.merge() :

```
In[3] :
df3 = pd.merge(df1, df2)
df3
```

```
Out[3] :
employé      groupe dt_embau
0  Robert      Compta      2008
1    Jake  Ingénierie      2012
2    Lise  Ingénierie      2004
3   Sophie        RH      2014
```

La fonction pd.merge() détecte que chacun des deux objets DataFrame d'entrée possède une colonne nommée *employé* ; elle se sert de cette colonne comme clé de jointure. Le résultat est un objet DataFrame résultant de la fusion. Notez que l'ordre des entrées dans les colonnes n'est pas obligatoirement maintenu. Dans l'exemple, l'ordre dans la colonne *employé* n'est pas le même dans *df1* et *df2* et pd.merge() prend cela en compte. Notez également que la fusion supprime en général l'index, sauf lorsque vous

demandez une fusion par l'index comme nous le verrons un peu plus loin avec les options left_index et right_index.

Jointure plusieurs-vers-un

Dans ce type de jointure, une des deux colonnes de clés contient des entrées en double. Les doublons sont conservés le cas échéant dans l'objet DataFrame résultant. Voici un exemple :

In[4] :

```
df4 = pd.DataFrame({'groupe': ['Compta',  
'Ingénierie', 'RH'],  
'superviseur': ['Caroline',  
'Guy', 'Steve']})  
display('df3', 'df4', 'pd.merge(df3, df4)')
```

Out[4] :

df3

	employé	groupe	dt_embau
0	Robert	Compta	2008
1	Jake	Ingénierie	2012
2	Lise	Ingénierie	2004
3	Sophie	RH	2014

df4

	groupe	superviseur
0	Compta	Caroline
1	Ingénierie	Guy
2	RH	Steve

```
pd.merge(df3, df4)
    employé      groupe dt_embau superviseur
0   Robert      Compta      2008   Caroline
1     Jake  Ingénierie      2012        Guy
2     Lise  Ingénierie      2004        Guy
3   Sophie        RH      2014       Steve
```

Le résultat contient une colonne supplémentaire avec l'information pour le superviseur. Cette information est répétée en fonction des données d'entrées.

Jointure plusieurs-vers-plusieurs

Le fonctionnement de ce type de jointure n'est pas évident de prime abord. Pourtant, les règles sont bien définies : lorsque dans les deux tableaux d'entrée, la colonne de clés contient des doublons, le résultat est une fusion plusieurs-vers-plusieurs. Voyons cela par un exemple. Partons d'un objet DataFrame contenant des domaines de compétences associés à des salariés.

Nous pouvons obtenir les compétences que possède n'importe quel salarié en réalisant une jointure plusieurs-vers-plusieurs :

In[5] :

```
df5 = pd.DataFrame({'groupe': ['Compta',
 'Compta',
```

```
'Ingénierie',
'Ingénierie', 'RH', 'RH'],
'savoirs': ['math',
'finance', 'codage', 'linux',
'finances',
'organisation']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

Out[5]:

```
df1
employé      groupe
0   Robert     Compta
1   Jake Ingénierie
2   Lise Ingénierie
3   Sophie      RH
```

df5

```
        groupe      savoirs
0       Compta      math
1       Compta      finance
2  Ingénierie      codage
3  Ingénierie      linux
4           RH      finances
5           RH  organisation
pd.merge(df1, df5)
```

```
employé      groupe      savoirs
0   Robert     Compta      math
1   Robert     Compta      finance
2   Jake Ingénierie      codage
3   Jake Ingénierie      linux
```

```
4    Lise Ingénierie      codage
5    Lise Ingénierie      linux
6    Sophie      RH      finances
7    Sophie      RH organisation
```

Les trois types de jointures que nous venons de voir peuvent être combinés à d'autres outils Pandas pour offrir une grande richesse fonctionnelle. En réalité, les jeux de données sont rarement aussi propres que ceux qui nous ont servi d'exemples. Voyons donc les options proposées par `pd.merge()` pour résoudre les problèmes spécifiques que peuvent poser les données d'entrée.

Spécification de la clé de fusion

Nous connaissons le comportement normal de `pd.merge()` qui est de chercher des noms de colonnes identiques entre les deux jeux d'entrée, puis de se servir de celles-ci comme clés. Souvent, les noms de colonnes ne coïncident pas ; `pd.merge()` propose plusieurs options pour surmonter ces soucis.

Le mot-clé `on`

Vous pouvez tout simplement indiquer explicitement le nom de la colonne servant de clé au moyen du mot réservé `on`. Vous lui fournissez un nom de colonne ou une liste de noms :

In[6]:

```
display('df1', 'df2', "pd.merge(df1, df2,  
on='employé')")
```

Out[6]:

df1

	employé	groupe
0	Robert	Compta
1	Jake	Ingénierie
2	Lise	Ingénierie
3	Sophie	RH

df2

	employé	dt_embau
0	Lise	2004
1	Robert	2008
2	Jake	2012
3	Sophie	2014

pd.merge(df1, df2, on='employé')

	employé	groupe	dt_embau
0	Robert	Compta	2008
1	Jake	Ingénierie	2012
2	Lise	Ingénierie	2004
3	Sophie	RH	2014

Notez que cette option n'est utilisable que si les deux objets d'entrée contiennent le même nom de colonne.

Les mots réservés `left_on` et `right_on`

Il vous arrivera d'avoir à fusionner deux jeux de données dont les noms de colonnes ne coïncident pas. Un jeu source peut par exemple avoir la mention *prénom* pour le nom du salarié, alors que l'autre utilise *employé*. Nous pouvons dans ce cas utiliser les mots réservés `left_on` et `right_on` pour désigner les deux noms de colonnes :

In[7]:

```
df3 = pd.DataFrame({'prénom': ['Robert', 'Jake',
'Lise', 'Sophie'],
'salaire': [70000, 80000,
120000, 90000]})

display('df1', 'df3', 'pd.merge(df1, df3,
left_on="employé", right_on="prénom")')
```

Out[7]:

```
df1
 employé      groupe
0   Robert     Compta
1     Jake    Ingénierie
2     Lise    Ingénierie
3   Sophie        RH
```

df3

```
 prénom  salaire
0   Robert    70000
1     Jake    80000
2     Lise   120000
```

```
3 Sophie      90000

pd.merge(df1, df3, left_on="employé",
right_on="prénom")
   employé      groupe prénom    salaire
0    Robert      Compta Robert     70000
1     Jake  Ingénierie   Jake     80000
2     Lise  Ingénierie   Lise    120000
3   Sophie          RH Sophie     90000
```

Le résultat va contenir une colonne redondante que nous pouvons éliminer au besoin par exemple avec la méthode drop() d'un objet DataFrame :

In[8] :

```
pd.merge(df1, df3, left_on="employé",
right_on="prénom").drop('prénom', axis=1)
```

Out[8] :

```
   employé      groupe    salaire
0    Robert      Compta     70000
1     Jake  Ingénierie     80000
2     Lise  Ingénierie    120000
3   Sophie          RH     90000
```

Les mots réservés `left_index` et `right_index`

Vous voudrez parfois fusionner sur un index plutôt que sur une colonne. Partons de la préparation de données suivante :

In[9] :

```
df1a = df1.set_index('employé')
df2a = df2.set_index('employé')
display('df1a', 'df2a')
```

Out[9] :

```
df1a
      groupe
employé
Robert    Compta
Jake     Ingénierie
Lise     Ingénierie
Sophie       RH
```

```
df2a
      dt_embau
employé
Lise        2004
Robert      2008
Jake        2012
Sophie      2014
```

Nous pouvons nous servir de l'index comme clé de fusion au moyen des options left_index et/

ou right_index de pd.merge :

In[10] :

```
print(df1a);
display('df1a', 'df2a',
```

```
"pd.merge(df1a, df2a, left_index=True,  
right_index=True)")
```

Out[10]:

```
df1a  
      groupe  
employé  
Robert    Compta  
Jake     Ingénierie  
Lise     Ingénierie  
Sophie       RH  
df2a  
      dt_embau  
employé  
Lise        2004  
Robert      2008  
Jake        2012  
Sophie      2014
```

```
pd.merge(df1a, df2a, left_index=True,  
right_index=True)
```

```
      groupe  dt_embau  
employé  
Robert    Compta      2008  
Jake     Ingénierie   2012  
Lise     Ingénierie   2004  
Sophie       RH       2014
```

L'objet DataFrame possède également la méthode join avec laquelle la fusion est réalisée par défaut sous forme de

jointure des index :

In[11]:

```
display('df1a', 'df2a', 'df1a.join(df2a)')
```

Out[11]:

```
df1a
      groupe
employé
Robert    Compta
Jake     Ingénierie
Lise     Ingénierie
Sophie       RH

df2a
      dt_embau
employé
Lise        2004
Robert      2008
Jake        2012
Sophie      2014

df1a.join(df2a)
      groupe  dt_embau
employé
Robert    Compta      2008
Jake     Ingénierie   2012
Lise     Ingénierie   2004
Sophie       RH       2014
```

Vous pouvez combiner index et colonne en stipulant left_index en même temps que right_on ou left_on en même temps que right_index :

In[12]:

```
display('df1a', 'df3', "pd.merge(df1a, df3,  
left_index=True, right_on='prénom')")
```

Out[12]:

```
df1a  
      groupe  
employé  
Robert      Compta  
Jake       Ingénierie  
Lise       Ingénierie  
Sophie        RH
```

```
df3  
    prénom   salaire  
0  Robert    70000  
1    Jake    80000  
2    Lise   120000  
3 Sophie    90000
```

```
pd.merge(df1a, df3, left_index=True,  
right_on='prénom')  
      groupe  prénom   salaire  
0      Compta  Robert    70000
```

```
1 Ingénierie Jake 80000
2 Ingénierie Lise 120000
3 RH Sophie 90000
```

Ces différentes options acceptent toutes plusieurs index ou colonnes. La syntaxe en est très intuitive. Pour d'autres détails, voyez dans la documentation de Pandas la section « Merge, Join, and Concatenate » (<http://pandas.pydata.org/pandas-docs/stable/merging.html>).

Arithmétique des ensembles et fusions

Dans tous les exemples précédents concernant les jointures, nous avions mis l'accent sur un point essentiel : le choix du type d'arithmétique d'ensembles à utiliser. Il s'agit de la façon de procéder lorsqu'une valeur est trouvée dans une colonne clé mais pas dans l'autre. Partons d'un exemple :

```
In[13]:
df6 = pd.DataFrame({'prénom': ['Pierre', 'Paul',
'Marie'],
'aliment': ['poisson',
'légume', 'pain']},
columns=['prénom', 'aliment'])
df7 = pd.DataFrame({'prénom': ['Marie',
'Joseph'],
'boisson': ['vin', 'bière']}),
```

```
        columns=['prénom', 'boisson'])
display('df6', 'df7', pd.merge(df6, df7))
```

Out[13]:

```
df6
  prénom   aliment
0  Pierre    poisson
1    Paul    légume
2   Marie     pain
df7
  prénom   boisson
0   Marie      vin
1  Joseph    bière
```

```
pd.merge(df6, df7)
  prénom   aliment   boisson
0   Marie     pain      vin
```

Le prénom *Marie* est le seul qui existe dans les deux jeux de données. Le résultat produit par défaut correspond à l'intersection des deux jeux, et donc uniquement aux données de cette entrée. Il s'agit d'une jointure interne. Pour être plus explicite, il suffit d'utiliser le mot-clé `how` dont la valeur par défaut est '`inner`' :

```
In[14]:
pd.merge(df6, df7, how='inner')
```

Out[14]:

```
    prénom   aliment   boisson  
0      Marie       pain       vin
```

Le mot `how` accepte les autres options '`outer`', '`left`' et '`right`'. La jointure externe que permet `outer` correspond à une union des colonnes d'entrées, avec remplissage des valeurs manquantes par une pseudo-valeur :

In[15]:

```
display('df6', 'df7', "pd.merge(df6, df7,  
how='outer')")
```

Out[15]:

```
df6  
    prénom   aliment  
0  Pierre   poisson  
1  Paul     légume  
2  Marie     pain
```

```
df7  
    prénom   boisson  
0  Marie     vin  
1  Joseph    bière
```

```
pd.merge(df6, df7, how='outer')  
    prénom   aliment   boisson  
0  Pierre   poisson     NaN  
1  Paul     légume     NaN  
2  Marie     pain      vin  
3  Joseph    NaN       bière
```

Enfin, les jointures gauche et droite produisent en sortie le contenu des entrées gauches ou des entrées droites respectivement :

In[16]:

```
display('df6', 'df7', "pd.merge(df6, df7,  
how='left')")
```

Out[16]:

```
df6  
  prénom   aliment  
0  Pierre    poisson  
1  Paul     légume  
2  Marie      pain
```

```
df7  
  prénom   boisson  
0  Marie       vin  
1  Joseph     bière
```

```
pd.merge(df6, df7, how='left')  
  prénom   aliment   boisson  
0  Pierre    poisson     NaN  
1  Paul     légume     NaN  
2  Marie      pain      vin
```

Dans cet exemple, nous obtenons les entrées du premier jeu de données. Le raisonnement est le même pour `how='right'`.

Ces différentes options s'appliquent directement à tous les types de jointures présentés.

Suffixes et conflits de noms de colonnes

Dans certains cas, vos deux objets d'entrées DataFrame contiennent au moins un nom de colonne qui entre en conflit, comme dans cet exemple :

In[17]:

```
df8 = pd.DataFrame({'prénom': ['Robert', 'Jake',
'Lise', 'Sophie'],
'rang': [1, 2, 3, 4]})

df9 = pd.DataFrame({'prénom': ['Robert', 'Jake',
'Lise', 'Sophie'],
'rang': [3, 1, 4, 2]})

display('df8', 'df9', 'pd.merge(df8, df9,
on="prénom")')
```

Out[17]:

df8

	nom	rang
0	Robert	1
1	Jake	2
2	Lise	3
3	Sophie	4

df9

```
    nom      rang
0  Robert    3
1  Jake     1
2  Lise     4
3 Sophie    2
pd.merge(df8, df9, on="nom")
   prénom  rang_x  rang_y
0  Robert    1        3
1  Jake     2        1
2  Lise     3        4
3 Sophie    4        2
```

Pour éviter toute confusion dans les résultats, la fonction de fusion ajoute automatiquement en suffixe une mention `_x` ou `_y`. Si ce comportement ne vous convient pas, vous pouvez choisir votre propre suffixe au moyen du mot réservé `suffixes` :

In[18]:

```
display('df8', 'df9', 'pd.merge(df8, df9,
on="prénom", suffixes=["_G", "_D"]))')
```

Out[18]:

```
df8
   prénom  rang
0  Robert    1
1  Jake     2
2  Lise     3
3 Sophie    4
```

```
df9  
  prénom  rang  
0  Robert   3  
1  Jake     1  
2  Lise     4  
3 Sophie    2
```

```
pd.merge(df8, df9, on="prénom", suffixes=["_G",  
                                         "_D"])  
  prénom  rang_G  rang_D  
0  Robert   1      3  
1  Jake     2      1  
2  Lise     3      4  
3 Sophie    4      2
```

Cette qualification s'applique à toutes les jointures possibles, même lorsque plusieurs colonnes sont en conflit.

Pour d'autres détails, vous verrez la prochaine section relative à l'agrégation et aux groupements. Voyez également la section « Merge, Join, and Concatenate » dans la documentation de Pandas.

Exemple : données démographiques des USA

On a d'abord recours aux opérations de fusion et jointure lorsqu'il s'agit de combiner des données provenant de sources différentes. Dans notre exemple, nous allons utiliser

des données concernant les États des USA et leur population. Les fichiers correspondants se trouvent à l'adresse <http://github.com/jakevdp/data-USstates/> :

In[19]:

```
# Téléchargement des données par commandes shell
# !curl -O
https://raw.githubusercontent.com/jakevdp/data-
USstates/master/state-population.csv
# !curl -O
https://raw.githubusercontent.com/jakevdp/data-
USstates/master/state-areas.csv
# !curl -O
https://raw.githubusercontent.com/jakevdp/data-
USstates/master/state-abbrevs.csv
```

Voyons d'abord le contenu des trois jeux de données au moyen de la fonction `read_csv()` de Pandas :

In[20]:

```
pop = pd.read_csv('data/state-
population.csv')
areas = pd.read_csv('data/state-areas.csv')
abbrevs = pd.read_csv('data/state-abbrevs.csv')

display('pop.head()', 'areas.head()',
'abbrevs.head()')
```

Out[20]:

```
pop.head()
```

```
state/region    ages      year  population
0   AL           under18  2012  1117489.0
1   AL           total    2012  4817528.0
2   AL           under18  2010  1130966.0
3   AL           total    2010  4785570.0
4   AL           under18  2011  1125763.0
```

```
areas.head()
  state      area (sq. mi)
0  Alabama     52423
1  Alaska      656425
2  Arizona     114006
3  Arkansas    53182
4  California   163707
```

```
abbrvs.head()
  state      abbreviation
0  Alabama        AL
1  Alaska         AK
2  Arizona        AZ
3  Arkansas       AR
4  California     CA
```

Supposons que nous cherchions un résultat assez évident : nous voulons classer les états et les territoires en fonction des densités de population décroissantes en 2010. Toutes les données dont nous avons besoin sont présentes, mais il va s'agir de les combiner correctement.

Nous commençons par une fusion plusieurs-vers-un afin de disposer du nom complet de chaque état dans le cadre de données DataFrame de la population. Nous réalisons la fusion en fonction de la colonne state/region du bloc pop, avec la colonne abbreviation de abrevs. Nous demandons une jointure externe avec how='outer' afin de ne pas laisser perdre des données en cas de labels mal orthographiés :

In[21]:

```
merged = pd.merge(pop, abrevs, how='outer',
                  left_on='state/region',
                  right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop
des duplicates
merged.head()
```

Out[21]:

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	
	Alabama				
1	AL	total	2012	4817528.0	
	Alabama				
2	AL	under18	2010	1130966.0	
	Alabama				
3	AL	total	2010	4785570.0	
	Alabama				
4	AL	under18	2011	1125763.0	
	Alabama				

Vérifions justement s'il y avait des possibilités de conflits en demandant les lignes contenant des valeurs nulles :

In[22]:

```
merged.isnull().any()
```

Out[22]:

```
state/region    False
ages           False
year            False
population     True
state           True
dtype: bool
```

Effectivement, certaines données de population sont vides ; voyons lesquelles :

In[23]:

```
merged[merged['population'].isnull()].head()
```

Out[23]:

```
      state/region  ages   year  population  state
2448    PR          under18  1990      NaN      NaN
2449    PR          total    1990      NaN      NaN
2450    PR          total    1991      NaN      NaN
2451    PR          under18  1991      NaN      NaN
2452    PR          total    1993      NaN      NaN
```

Il semble que les valeurs nulles correspondent à la population de Puerto Rico (PR) avant l'an 2000. Ces données

ne devaient sans doute pas être disponibles à cette époque.

Nous constatons par ailleurs que certaines entrées de *state* sont nulles elles aussi, ce qui signifie qu'il n'y avait pas d'entrées correspondantes dans *abrevs*. Voyons quelles régions sont concernées :

In[24]:

```
merged.loc[merged['state'].isnull(),  
           'state/region'].unique()
```

Out[24]:

```
array(['PR', 'USA'], dtype=object)
```

Nous pouvons ainsi voir où se trouve le souci : les données de population comportent des entrées pour Puerto Rico (PR) et pour les États-Unis en tant que tels (USA). En revanche, ces entrées n'existent pas dans la clé d'abréviation d'états. Nous pouvons corriger cela en remplaçant les entrées concernées :

In[25]:

```
merged.loc[merged['state/region'] == 'PR',  
           'state'] = 'Puerto Rico'  
merged.loc[merged['state/region'] == 'USA',  
           'state'] = 'United States'  
merged.isnull().any()
```

Out[25]:

```
state/region    False
ages           False
year            False
population     True
state           False
dtype: bool
```

Dorénavant, la recherche de valeurs nulles devient fausse pour la colonne *state*. Nous sommes prêts !

Nous allons maintenant fusionner le résultat avec les données de superficie *area*, en utilisant la même procédure. Nous pourrons ensuite faire une jointure sur la colonne *state* des deux tables :

In[26]:

```
final = pd.merge(merged, areas, on='state',
how='left')
final.head()
```

Out[26]:

	state/region	ages	year	population	state
area (sq. mi)					
0	AL	under18	2012	1117489.0	
	Alabama	52423.0			
1	AL	total	2012	4817528.0	
	Alabama	52423.0			
2	AL	under18	2010	1130966.0	
	Alabama	52423.0			
3	AL	total	2010	4785570.0	

```
Alabama 52423.0
4 AL under18 2011 1125763.0
Alabama 52423.0
```

Vérifions ici aussi s'il y a des valeurs nulles et des divergences de noms :

```
In[27]:
final.isnull().any()
```

```
Out[27]:
state/region    False
ages           False
year            False
population     True
state           False
area (sq. mi)  True
dtype: bool
```

Il y a effectivement des valeurs nulles dans la colonne *area*.
Voyons quelles régions ont été ignorées de ce fait :

```
In[28]:
final['state'][final['area (sq.
mi)'].isnull()].unique()
```

```
Out[28]:
array(['United States'], dtype=object)
```

Il semble qu'il n'y ait pas de données de superficie globale pour le pays USA en entier. Nous pourrions insérer une valeur, par exemple en calculant la somme de toutes les superficies des états, mais nous allons faire plus simple. Nous abandonnons les valeurs nulles parce que la densité de population globale des USA ne nous intéresse pas ici :

In[29]:

```
final.dropna(inplace=True)  
final.head()
```

Out[29]:

	state/region	ages	year	population	state
	area (sq. mi)				
0	AL	under18	2012	1117489.0	
	Alabama	52423.0			
1	AL	total	2012	4817528.0	
	Alabama	52423.0			
2	AL	under18	2010	1130966.0	
	Alabama	52423.0			
3	AL	total	2010	4785570.0	
	Alabama	52423.0			
4	AL	under18	2011	1125763.0	
	Alabama	52423.0			

Toutes les données dont nous avons besoin sont maintenant prêtes et nous allons pouvoir répondre à la question. Commençons par sélectionner la portion des données pour l'année 2000, toutes tranches d'âges confondues, avec leurs

totaux. Nous nous servons de la fonction `query`, mais sachez qu'il faut installer le paquetage `numexpr` (voyez la section sur `eval` et `query` en toute fin de ce chapitre) :

In[30]:

```
data2010 = final.query("year == 2010 & ages ==  
'total'")  
data2010.head()
```

Out[30]:

	state/region	ages	year	population
state	area (sq. mi)			
3	AL	total	2010	4785570.0
	Alabama	52423.0		
91	AK	total	2010	713868.0
	Alaska	656425.0		
101	AZ	total	2010	6408790.0
	Arizona	114006.0		
189	AR	total	2010	2922280.0
	Arkansas	53182.0		
197	CA	total	2010	37333601.0
	California	163707.0		

Nous pouvons maintenant demander le calcul de la densité de population. Nous allons réindexer les données par État avant de calculer le résultat :

In[31]:

```
data2010.set_index('state', inplace=True)  
density = data2010['population'] /
```

```
data2010['area (sq. mi)']

In[32]:
density.sort_values(ascending=False,
 inplace=True)
density.head()
```

```
Out[32]:
state
District of Columbia    8898.897059
Puerto Rico              1058.665149
New Jersey                1009.253268
Rhode Island               681.339159
Connecticut                 645.600649
dtype: float64
```

Nous obtenons une liste des États américains ainsi que Washington DC et Puerto Rico dans l'ordre décroissant des densités de population en 2010, en nombre de personnes par mille carré (environ 2,5 km²). Le district de Columbia où se situe Washington est la région la plus dense, mais l'État le plus dense est le New Jersey. Affichons la fin de la liste :

```
In[33]: density.tail()
```

```
Out[33]:
state
South Dakota      10.583512
North Dakota       9.537565
Montana            6.736171
```

```
Wyoming      5.768079  
Alaska       1.087509  
dtype: float64
```

L'État le moins dense est évidemment l'Alaska, avec à peine plus d'une personne par mille carré.

Ce genre d'opération de fusion et de jointure en plusieurs étapes est fréquemment utilisé pour répondre à des questions du monde réel. J'espère que l'exemple vous a donné une idée de ce qu'il est possible de faire avec les outils présentés pour faire parler vos données !

3.9 : Agrégations et groupements

Un domaine de traitement essentiel lorsqu'il s'agit d'analyser des données en grand volume est la création de résumés ou synthèses efficaces. Les opérations d'agrégation offertes par des méthodes telles que `sum()`, `mean()`, `median()`, `min()` et `max()` produisent une valeur unique qui permet d'en savoir plus sur la nature exacte des données d'entrée. Nous allons découvrir dans cette section les opérations d'agrégation offertes par Pandas, qu'il s'agisse d'opérations simples ressemblant à celles vues avec les tableaux NumPy ou d'opérations plus complexes fondées sur le concept d'objet de groupement `GroupBy`.



(N.d.T.) Comme au début de la section précédente, nous ne répétons pas le code source de la fonction de confort `display()` définie plus haut.

Données des exoplanètes

Nous allons nous servir du jeu de données nommé `planets` qui est incorporé au paquetage Seaborn (<http://seaborn.pydata.org>). Nous reviendrons en détail sur la visualisation avec Seaborn dans le [Chapitre 4](#). Ce jeu de données contient des mesures concernant les exoplanètes

découvertes dans un secteur choisi du ciel. Commençons par télécharger le fichier avec la commande Seaborn suivante :

In[2]:

```
import numpy as np
import pandas as pd

import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

Out[2]:

```
(1035, 6)
```

In[3]:

```
planets.head()
```

Out[3]:

	method	number	orbital_period	mass
	distance year			
0	Radial Velocity	1	269.300	7.10
77.40	2006			
1	Radial Velocity	1	874.774	2.21
56.95	2008			
2	Radial Velocity	1	763.000	2.60
19.84	2011			
3	Radial Velocity	1	326.030	19.40
110.62	2007			
4	Radial Velocity	1	516.220	10.50
119.47	2009			

Dans la version de 2014 de ces découvertes, il y a plus de 1 000 exoplanètes listées.

Une agrégation simple dans Pandas

Nous avons découvert dans le [Chapitre 2](#) certaines opérations d'agrégation pour les tableaux NumPy. Comme dans le cas d'un tableau NumPy à une dimension, un objet Series de Pandas renvoie par agrégation une valeur unique :

In[4] :

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

Out[4] :

```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

In[5] : `ser.sum()`

Out[5] : 2.8119254917081569

In[6] : `ser.mean()`

Out[6] : 0.56238509834163142

Pour un objet DataFrame, l'opération d'agrégation renvoie par défaut les résultats pour chaque colonne :

```
In[7]:  
df = pd.DataFrame({'A': rng.rand(5),  
                   'B': rng.rand(5)})  
df
```

Out[7] :

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

```
In[8]: df.mean()
```

```
Out[8]: A    0.477888  
        B    0.443420  
       dtype: float64
```

Vous pouvez demander l'agrégation pour chaque ligne en ajoutant le paramètre axis :

```
In[9]:  
df.mean(axis='columns')
```

```
Out[9]:  
0    0.088290
```

```
1    0.513997  
2    0.849309  
3    0.406727  
4    0.444949  
dtype: float64
```

Les objets Series et DataFrame de Pandas disposent de toutes les opérations d'agrégation vues pour les objets NumPy dans le [Chapitre 2](#). Ils y ajoutent une méthode polyvalente nommée describe() qui permet d'obtenir différents agrégats par colonne en renvoyant un résultat. Appliquons cette méthode à nos données d'exoplanètes, en supprimant les lignes contenant des manques :

```
In[10]:  
planets.dropna().describe()
```

```
Out[10]:
```

	number	orbital_period	mass
distance		year	
count	498.00000	498.000000	498.000000
	498.000000	498.000000	
mean	1.73494	835.778671	2.509320
	52.068213	2007.377510	
std	1.17572	1469.128259	3.636274
	46.596041	4.167284	
min	1.00000	1.328300	0.003600
	1.350000	1989.000000	
25%	1.00000	38.272250	0.212500

```

24.497500 2005.000000
50%      1.000000    357.000000    1.245000
39.940000 2009.000000
75%      2.000000    999.600000    2.867500
59.332500 2011.000000
max      6.000000   17337.500000   25.000000
354.000000 2014.000000

```

Cette première opération permet de se faire une idée générale des propriétés du jeu de données. Nous voyons par exemple que la colonne de l'année year montre que quasiment 50 % des exoplanètes ont été découvertes à partir de 2010. C'est tout simplement le résultat de la mission Kepler, un télescope spatial qui avait été envoyé expressément pour trouver des exoplanètes autour de certaines étoiles.

Le [Tableau 3.3](#) montre quelques autres méthodes d'agrégation standard de Pandas.

Tableau 3.3 : Méthodes d'agrégation de Pandas.

Agrégation	Description
count()	Dénombrement des éléments
first(), last()	Premier, dernier élément
mean(), median()	Moyenne et médiane
min(), max()	Minimum et maximum
std(), var()	Écart type et variance

mad()	Déviation absolue moyenne
prod()	Produit de tous les éléments
sum()	Somme de tous les éléments

Les deux objets DataFrame et Series disposent de toutes ces méthodes.

Les possibilités d'analyse au moyen de ces méthodes sont néanmoins limitées. Pour entrer plus en détail dans la création d'une vision synthétique des données, vous aurez recours à l'opération GroupBy qui permet de créer des agrégats à partir de sous-ensembles de données, de façon rapide et efficace.

GroupBy : Split-Apply-Combine

Vous aurez souvent besoin d'agréger vos données de façon conditionnelle, en fonction d'un label ou d'un index. C'est ce que permet l'opération proposée par GroupBy. Le nom provient de celui d'une commande du langage de gestion de bases de données SQL. Ce groupement conditionnel comporte trois étapes qui ont été indiquées au départ par Hadley Wickham, fameux spécialiste du langage R : sélectionner (ou *split*), appliquer et combiner.

Sélectionner, appliquer, combiner

La [Figure 3.1](#) donne un exemple générique de ce triplet d'opérations. Dans l'exemple, la partie « application » est une opération de sommation.

Cette [Figure 3.1](#) montre clairement le traitement réalisé par GroupBy :

- L'étape de sélection split consiste à séparer les données d'un DataFrame puis à les regrouper en fonction de la valeur d'une clé.
- L'étape d'application correspond à la réalisation d'un traitement, en général un agrégat, une transformation ou un filtrage, sur les différents groupes individuels.
- Enfin, l'opération de combinaison fusionne les résultats des opérations pour aboutir à un tableau en sortie.

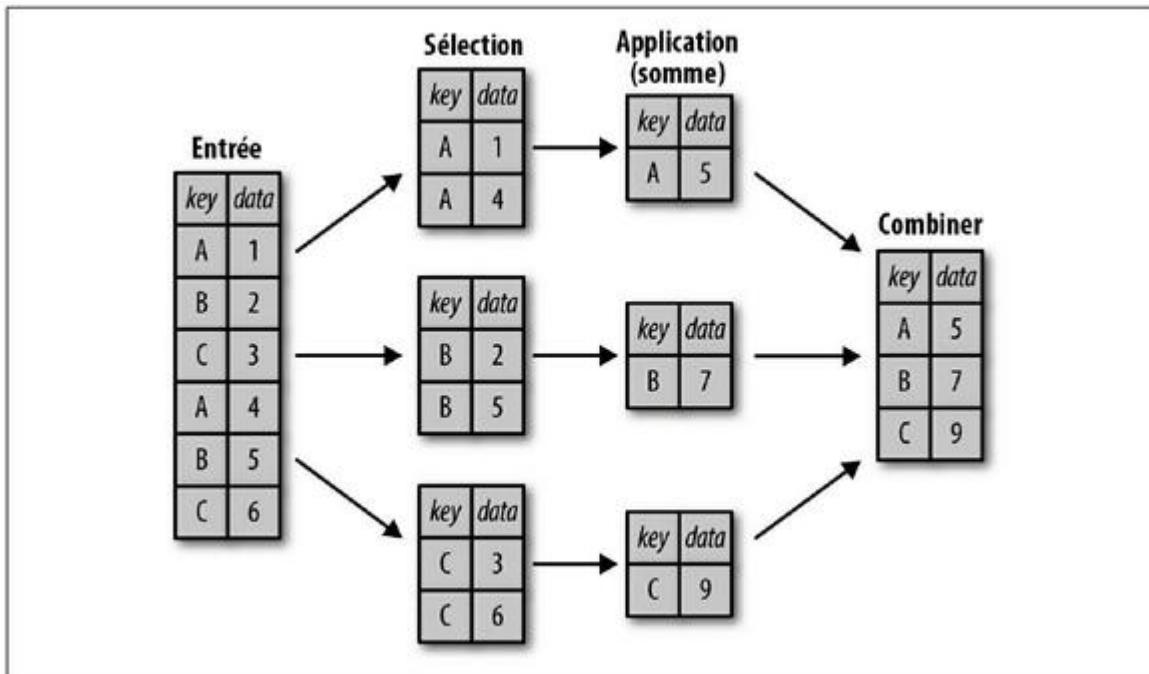


Figure 3.1 : Représentation visuelle d'une opération de groupement conditionnel GroupBy.

Nous pourrions arriver au même résultat manuellement en combinant des opérations de masquage, d'agrégation et de fusion. L'énorme différence est que les sous-groupes intermédiaires n'ont ici pas besoin d'être instanciés de façon explicite. En général, GroupBy s'en charge en réalisant une seule passe sur les données, tout en mettant à jour la somme, la moyenne, le nombre, le minimum et les autres valeurs d'agrégats pour chaque groupe. GroupBy permet donc de vous libérer de ces étapes. Vous n'avez plus besoin de savoir comment les traitements sont réalisés, et pouvez donc rester concentré sur votre vision globale de l'opération.

Prenons un exemple concret en voyant comment utiliser Pandas pour réaliser le calcul de la [Figure 3.1](#). Nous commençons par créer le jeu de données DataFrame d'entrée :

```
In[11]:  
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A',  
    'B', 'C'],  
    'data': range(6)}, columns=  
    ['key', 'data'])  
df
```

Out[11]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

Nous pouvons nous servir de la méthode groupby() de DataFrame pour faire réaliser l'opération split-apply-combine la plus élémentaire. Il suffit de transmettre le nom de la colonne devant servir de clé :

```
In[12]:  
df.groupby('key')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy  
object at 0x0001247DE29E20>
```

Notez que ce qui est renvoyé n'est pas un jeu d'objets DataFrame, mais un objet unique DataFrameGroupBy. C'est cet objet qui contient toute la partie magique : vous pouvez imaginer que c'est une vue spéciale de l'objet DataFrame, capable d'analyser les différents groupes, sans réaliser les calculs tant que l'opération d'agrégation n'est pas demandée. Cette approche d'évaluation retardée (*lazy*) permet de mettre en place des opérations d'agrégation classique de façon très efficace, et presque sans effort.

Pour obtenir un résultat, il suffit d'appliquer une opération d'agrégation à l'objet DataFrameGroupBy. Nous obtiendrons le résultat des étapes d'application et de combinaison appropriées :

In[13]:

```
df.groupby('key').sum()
```

Out[13]:

```
data  
key  
A      3  
B      5  
C      7
```

L'exemple utilise la méthode `sum()`, mais vous pouvez appliquer quasiment toutes les fonctions d'agrégation classique de Pandas ou de NumPy et quasiment toutes les opérations valides sur un objet DataFrame, comme nous allons le voir.

L'objet GroupBy

L'objet GroupBy propose une abstraction très souple que vous pouvez considérer comme une collection d'objets DataFrame qui réalise d'elle-même les opérations complexes. Voyons cela en action à partir des données des exoplanètes.

Les opérations les plus importantes que propose un objet GroupBy sont les agrégations, les filtrages, les transformations et les applications. Nous le verrons en détail dans les sections suivantes. Pour nous y préparer, découvrons d'abord quelques possibilités fonctionnelles que permettent les opérations fondamentales de GroupBy.

Indexation des colonnes. L'objet GroupBy permet d'indexer les colonnes de la même façon que l'objet DataFrame, en renvoyant un objet GroupBy modifié. Voici un exemple :

```
In[15]: planets.groupby('method')
<pandas.core.groupby.generic.DataFrameGroupBy
object at 0x000001247DE4A0A0>
```

```
In[15]: planets.groupby('method')
['orbital_period']
<pandas.core.groupby.generic.SeriesGroupBy
object at 0x000001247DE4AA90>
```

Nous venons de sélectionner un groupe Series à partir du groupe DataFrame de départ en citant un nom de colonne. Comme pour l'objet GroupBy, aucun calcul n'est réalisé tant que nous n'avons pas demandé de réaliser une agrégation sur l'objet :

```
In[16]:
planets.groupby('method')
['orbital_period'].median()
```

```
Out[16]:
method
Astrometry           631.180000
Eclipse Timing Variations 4343.500000
Imaging              27500.000000
Microlensing          3300.000000
Orbital Brightness Modulation 0.342887
Pulsar Timing          66.541900
Pulsation Timing Variations 1170.000000
Radial Velocity        360.200000
Transit                5.714932
Transit Timing Variations 57.011000
Name: orbital_period, dtype: float64
```

Ces traitements nous donnent une première idée des périodes orbitales (exprimées en jours) concernées par les différentes méthodes de mesure astronomique.

Itération parmi les groupes. L'objet GroupBy permet d'itérer parmi les groupes, en renvoyant chaque groupe en tant qu'objet Series ou DataFrame :

In[17]:

```
for (method, group) in
planets.groupby('method'):
    print("{0}:30s} shape={1}".format(method,
group.shape))
```

Astrometry	shape=(2, 6)
Eclipse Timing Variations	shape=(9, 6)
Imaging	shape=(38, 6)
Microlensing	shape=(23, 6)
Orbital Brightness Modulation	shape=(3, 6)
Pulsar Timing	shape=(5, 6)
Pulsation Timing Variations	shape=(1, 6)
Radial Velocity	shape=(553, 6)
Transit	shape=(397, 6)
Transit Timing Variations	shape=(4, 6)

Cette itération peut s'avérer utile pour réaliser certains traitements manuellement, mais il sera en général plus rapide de se servir de la fonction apply que nous allons décrire un peu plus loin.

Méthodes de distribution. Grâce au puissant mécanisme des classes Python, toute méthode qui n'est pas connue directement par l'objet GroupBy est retransmise pour être appliquée au groupe, qu'il s'agisse d'objets DataFrame ou Series. Il est ainsi possible d'utiliser la méthode describe() de DataFrame pour demander une série d'agrégations afin de décrire chacun des groupes de données :

In[18]:

```
planets.groupby('method')['year'].describe()
```

			count		mean
std	min	25%	50%	75%	max
method					
Astrometry			2.0	2011.500000	
2.121320	2010.0	2010.75	2011.5	2012.25	
2013.0					
Eclipse Timing Variations		9.0	2010.000000		
1.414214	2008.0	2009.00	2010.0	2011.00	
2012.0					
Imaging			38.0	2009.131579	
2.781901	2004.0	2008.00	2009.0	2011.00	
2013.0					
Microlensing			23.0	2009.782609	
2.859697	2004.0	2008.00	2010.0	2012.00	
2013.0					
Orbital Brightness Modul.		3.0	2011.666667		
1.154701	2011.0	2011.00	2011.0	2012.00	
2013.0					

Pulsar Timing		5.0	1998.400000		
8.384510	1992.0	1992.00	1994.0	2003.00	
2011.0					
Pulsation Timing Var.		1.0	2007.000000		
NaN	2007.0	2007.00	2007.0	2007.00	2007.0
Radial Velocity		553.0	2007.518987		
4.249052	1989.0	2005.00	2009.0	2011.00	
2014.0					
Transit		397.0	2011.236776		
2.077867	2002.0	2010.00	2012.0	2013.00	
2014.0					
Transit Timing Var.		4.0	2012.500000		
1.290994	2011.0	2011.75	2012.5	2013.25	
2014.0					

Ce tableau nous permet d'en apprendre plus au sujet des données : la colonne *count* permet de conclure que la plupart des planètes ont été découvertes par la mesure de la vitesse radiale ou du transit, la dernière de ces deux méthodes étant devenue plus efficace récemment (fruit d'une plus grande précision du télescope). Les deux méthodes de mesure les plus récemment introduites (pas avant 2011) semblent être les variations de temps de transit et la variation de luminosité orbitale.

Ce n'est qu'un exemple de l'intérêt des méthodes de distribution. Sachez que ces méthodes s'appliquent à chaque groupe individuel, avant que les résultats soient recombinés dans `GroupBy` pour être renvoyés. Toute méthode valide

pour DataFrame ou Series peut être appliquée à l'objet GroupBy correspondant, ce qui laisse entrevoir des opérations très polyvalentes et puissantes !

Agréger, filtrer, transformer, appliquer

Nous venons de nous intéresser aux opérations de combinaison d'un agrégat, mais d'autres traitements sont disponibles. Tout objet GroupBy dispose des quatre méthodes aggregate(), filter(), transform() et apply() qui permettent d'appliquer un grand nombre de traitements avant de recombiner les données groupées.

Nous allons nous servir de l'objet DataFrame suivant pour les prochains exemples :

In[19]:

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A',
'B', 'C'],
'data1': range(6),
'data2': rng.randint(0, 10,
6)},
columns = ['key', 'data1',
'data2'])
df
```

Out[19]:

key	data1	data2
A	0	9
B	1	2
C	2	4
A	3	5
B	4	7
C	5	8

0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Agrégation. Nous avons découvert les opérations d'agrégation élémentaire de GroupBy que sont sum(), median() et autres. La méthode aggregate() offre encore plus de souplesse. Elle accepte en paramètre une chaîne, une fonction ou une liste puis calcule tous les agrégats en une fois. Voici un exemple qui combine ces opérations :

In[20]:

```
df.groupby('key').aggregate(['min', np.median,
                           max])
```

Out[20]:

key	data1			data2		
	min	median	max	min	median	max
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Vous pouvez également transmettre un dictionnaire qui associe à chaque nom de colonne l'opération qui est à appliquer aux données de cette colonne :

```
In[21]:
```

```
df.groupby('key').aggregate({'data1': 'min',
'data2': 'max'})
```

```
Out[21]:
```

	data1	data2
key		
A	0	5
B	1	7
C	2	9

Filtrage. L'opération de filtrage permet d'éliminer des données en fonction des propriétés du groupe. Nous pouvons par exemple ne conserver que les groupes pour lesquels l'écart type est supérieur à une valeur choisie :

```
In[22]:
```

```
def filter_func(x):
    return x['data2'].std() > 4
```

```
display('df', "df.groupby('key').std()", "df.groupby('key').filter(filter_func)")
```

```
Out[22]
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3

```
4     B      4      7
5     C      5      9
```



```
df.groupby('key').std()
    data1      data2
key
A  2.12132  1.414214
B  2.12132  4.949747
C  2.12132  4.242641
```

```
df.groupby('key').filter(filter_func)
   key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

La fonction filter() doit renvoyer une valeur booléenne qui permet de conserver le groupe ou pas. Dans l'exemple, le groupe A n'ayant pas de valeur supérieure à 4 pour l'écart type, il n'apparaît pas dans le résultat.

Transformation. L'opération d'agrégation renvoie une version réduite des données alors que la transformation permet d'obtenir une version modifiée du jeu complet prêt à être recombiné. Les données de sortie ont le même format que les données d'entrée. Un exemple fréquent consiste à centrer les données en soustrayant la moyenne de groupe :

In[23]:

```
df.groupby('key').transform(lambda x: x -  
x.mean())
```

Out[23]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

La méthode apply(). La méthode apply() permet d'appliquer une fonction au résultat du groupe. Cette fonction doit attendre en entrée un objet DataFrame pour renvoyer soit un objet Pandas (DataFrame ou Series), soit un scalaire. L'opération de combinaison sera adaptée au type de résultat produit.

Voici par exemple comment apply() permet de normaliser le contenu de la première colonne en fonction de la somme de la deuxième colonne :

In[24]:

```
def norm_by_data2(x):  
    # x est un DataFrame de valeurs de groupe  
    x['data1'] /= x['data2'].sum()  
    return x
```

```
display('df',
"df.groupby('key').apply(norm_by_data2)")
```

Out[24]:

```
df
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

```
df.groupby('key').apply(norm_by_data2)
```

```
  key      data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

La méthode `apply()` de `GroupBy` est très souple puisque le seul critère demandé est que la fonction accepte en entrée un objet `DataFrame` et renvoie un objet Pandas ou un scalaire. Les traitements que vous demandez ne sont limités que par votre imagination !

Spécification de la clé de distribution split

Les exemples élémentaires que nous venons de voir comportaient une distribution des données de DataFrame en fonction d'un nom de colonne. Ce n'est qu'une des nombreuses options de définition de groupe possibles. Voyons-en quelques autres.

Liste, tableau, série ou index comme clés de regroupement.
La clé peut être n'importe quelle série ou liste dont la longueur coïncide avec celle de l'objet DataFrame. Voici un exemple :

In[25]:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
```

Out[25]:

```
df
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

```
df.groupby(L).sum()
```

	data1	data2
0	7	17

```
1      4          3  
2      4          7
```

Il existe donc une façon plus longue à écrire de l'opération df.groupby('key') vue plus haut :

In[26]:

```
display('df', "df.groupby(df['key']).sum()")
```

Out[26]:

```
df  
  key  data1  data2  
0   A      0      5  
1   B      1      0  
2   C      2      3  
3   A      3      3  
4   B      4      7  
5   C      5      9
```

```
df.groupby(df['key']).sum()
```

```
    data1  data2  
key  
A      3      8  
B      5      7  
C      7     12
```

Dictionnaire ou séries associant index et groupe. Une autre technique consiste à fournir un dictionnaire qui associe des valeurs d'index à des clés de groupe :

```
In[27]:
```

```
df2 = df.set_index('key')
mapping = {'A': 'voyelle', 'B': 'consonne', 'C': 'consonne'}
display('df2', 'df2.groupby(mapping).sum()')
# print(df2); print(df2.groupby(mapping).sum())
```

```
Out[27]:
```

```
df2
      data1  data2
key
  A    0    5
  B    1    0
  C    2    3
  A    3    3
  B    4    7
  C    5    9
```

```
df2.groupby(mapping).sum()
```

```
      data1  data2
consonne      12    19
voyelle       3     8
```

Une fonction Python. Pour un résultat similaire, vous pouvez transmettre une fonction Python qui prend les valeurs d'index en entrée pour produire le groupe :

```
In[28]:
```

```
display('df2', 'df2.groupby(str.lower).mean()')
```

```
Out[28]:  
df2  
    data1  data2  
key  
A      0      5  
B      1      0  
C      2      3  
A      3      3  
B      4      7  
C      5      9
```

```
df2.groupby(str.lower).mean()  
    data1  data2  
a    1.5    4.0  
b    2.5    3.5  
c    3.5    6.0
```

Une liste de clés valides. Les choix de clés précédents peuvent être combinés pour demander un groupement en fonction d'un multi-index :

```
In[29]:  
df2.groupby([str.lower, mapping]).mean()
```

```
Out[29]:  
    data1  data2  
a vowel    1.5    4.0  
b consonant  2.5    3.5  
c consonant  3.5    6.0
```

Exemple de groupement

En quelques lignes de code Python, nous pouvons dénombrer les exoplanètes découvertes par méthode de mesure et par décennie :

In[30] :

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])
['number'].sum().unstack().fillna(0)
```

Out[30] :

		1980s	1990s
decade			
2000s	2010s		
method			
Astrometry		0.0	0.0
2.0			0.0
Eclipse Timing Variations		0.0	0.0
10.0			5.0
Imaging		0.0	0.0
21.0			29.0
Microlensing		0.0	0.0
15.0			12.0
Orbital Brightness Modulation		0.0	0.0
0.0	5.0		
Pulsar Timing		0.0	9.0
1.0			1.0
Pulsation Timing Variations		0.0	0.0
			1.0

```
0.0
Radial Velocity           1.0      52.0
475.0   424.0
Transit                   0.0      0.0      64.0
712.0
Transit Timing Variations 0.0      0.0      0.0
9.0
```

L'exemple montre la puissance offerte par les différentes opérations lorsqu'elles sont appliquées à des jeux de données réels. Nous accédons ainsi rapidement à une compréhension générale des dates et des moyens utilisés pour découvrir les exoplanètes au cours des dernières décennies !

Je vous suggère d'étudier les instructions de cet exemple avec soin ; cherchez à identifier chaque étape afin de comprendre en quoi elle participe au résultat. Ce n'est évidemment pas un exemple trop simple, et c'est pourquoi en prenant le temps de l'analyser, vous saurez plus facilement analyser vos propres données.

3.10 : Tableaux croisés dynamiques ou tableaux pivots

Nous venons de voir en quoi l'abstraction GroupBy permettait d'étudier les relations dans un jeu de données. Le concept de tableau croisé dynamique ou tableau pivot, est bien connu des utilisateurs de tableurs. Rappelons qu'un tel tableau prend en entrée des données en colonnes pour les regrouper dans un tableau de sortie à deux dimensions qui produit des valeurs dans plusieurs dimensions. Une confusion reste possible entre les tableaux croisés et GroupBy. Personnellement, je considère les tableaux croisés comme une version à plusieurs dimensions de l'agrégation GroupBy. Il s'agit toujours de créer des sous-groupes par sélection (split), d'appliquer puis de recombiner, mais les deux opérations d'éclatement et de recombinaison ne s'appliquent plus seulement à un index à une dimension, mais à une grille à deux dimensions.

Données de test des tableaux pivots

Tous les exemples de cette section vont se baser sur la fameuse liste des passagers du Titanic qui est intégrée à la

librairie Seaborn (nous reverrons Seaborn dans le chapitre suivant) :

In[1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
```

In[2]:

```
titanic.head()
```

Out[2]:

	survived	pclass	sex	age	sibsp	parch
	fare	embarked	class			
0	0	3	male	22.0	1	0
7.2500		S	Third			
1	1	1	female	38.0	1	0
71.2833		C	First			
2	1	3	female	26.0	0	0
7.9250		S	Third			
3	1	1	female	35.0	1	0
53.1000		S	First			
4	0	3	male	35.0	0	0
8.0500		S	Third			
alone	who	adult_male	deck	embark_town	alive	
man		True	NaN	Southampton		no
False						

woman	False	C	Cherbourg	yes
False				
woman	False	NaN	Southampton	yes
True				
woman	False	C	Southampton	yes
False				
man	True	NaN	Southampton	no
True				

Cette table contient toutes sortes d'informations concernant les passagers de ce voyage fatidique, et notamment le sexe, l'âge, la classe, le prix du billet et bien d'autres choses.

Tableaux croisés manuels

Pour commencer à extraire du sens de ces données tragiques, nous pouvons commencer par les regrouper en fonction du genre (du sexe), du statut de survie ou d'une combinaison de ces deux critères. Puisque vous avez lu la section précédente, vous pourriez utiliser GroupBy pour par exemple connaître le taux de survie par sexe :

In[3] :

```
titanic.groupby('sex')[['survived']].mean()
```

Out[3] :

survived
sex

```
female 0.742038  
male 0.188908
```

Nous apprenons ainsi que les trois quarts des femmes ont survécu, alors que seul un homme sur cinq n'a pas péri !

Allons un peu plus loin en étudiant la relation entre sexe et classe du passager. En utilisant GroupBy, nous devons rédiger une séquence de traitements qui ressemble à ceci : nous regroupons d'abord par classe et par sexe, nous sélectionnons le statut de survie, nous appliquons une agrégation de moyenne, nous recombinons les groupes résultants puis nous dépilons par unstack() l'index hiérarchisé pour obtenir les dimensions cachées. Voici l'instruction :

In[4]:

```
titanic.groupby(['sex', 'class'])  
['survived'].aggregate('mean').unstack()
```

Out[4]:

	class	First	Second	Third
sex				
female	0.968085	0.921053	0.500000	
male	0.368852	0.157407	0.135447	

Nous voyons ainsi que le prix du billet a un effet important sur le taux de survie. En revanche, le code commence à être un peu touffu et l'instruction est assez longue. Chaque étape

de ce tuyau de traitement est bien à sa place, puisque tous ces outils ont été présentés plus haut. En revanche, l'instruction commence à être malaisée à lire. Du fait que ce genre de travail à deux dimensions dans l'esprit GroupBy est assez habituel, Pandas a prévu une routine polyvalente nommée pivot_table qui permet de gérer de façon plus efficace ce type d'agrégation à plusieurs dimensions.

Syntaxe des tableaux croisés

Voici comment reformuler le traitement précédent avec la méthode pivot_table d'un objet DataFrame :

In[5] :

```
titanic.pivot_table('survived', index='sex',  
columns='class')
```

Out[5] :

	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Le résultat est strictement le même qu'avec GroupBy, mais l'instruction est beaucoup plus facile à lire et à rédiger. Comme vous pouvez vous y attendre pour une traversée de l'Atlantique au début du xx^e siècle, le taux de survie favorise les femmes, et surtout les femmes riches. Les femmes

voyageant en première classe ont pratiquement toutes survécu (bonjour Rose !), alors que seul un homme sur dix de la troisième classe a survécu (désolé, Jack !).

Tableaux croisés multiniveaux

Comme avec GroupBy, on peut spécifier plusieurs niveaux pour le groupement dans un tableau croisé, et préciser un certain nombre d'options. Nous pouvons par exemple voir l'impact sur les classes d'âge. Créons des baquets (*bins*) pour les différents âges au moyen de la fonction pd.cut :

In[6] :

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', 'age'],
'class')
```

Out[6] :

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

La même technique peut s'appliquer aux colonnes. Ajoutons des informations concernant le prix du ticket avec pd.qcut

afin de générer automatiquement des quantiles :

In[7] :

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', 'age'],
['fare', 'class'])
```

Out[7] :

```
fare      (-0.001, 14.454]  (14.454, 512.329]
          class      First   Second   Third   First
Second   Third
       sex     age
female  (0, 18]   NaN  1.000000  0.714286
0.909091  1.000000  0.318182
                  (18, 80]   NaN  0.880000  0.444444  0.972973
0.914286  0.391304
       male  (0, 18]   NaN  0.000000  0.260870  0.800000
0.818182  0.178571
                  (18, 80]   0.0  0.098039  0.125000  0.391304
0.030303  0.192308
```

Nous obtenons ainsi un agrégat à quatre dimensions avec des indices hiérarchisés (revoyez les index hiérarchisés en début de chapitre si nécessaire). La grille obtenue permet de voir les relations entre les différentes valeurs.

Autres options des tableaux croisés

Voici la syntaxe formelle de la méthode pivot_table de DataFrame :

```
# Syntaxe d'appel depuis Pandas 0.18
DataFrame.pivot_table(data, values=None,
index=None, columns=None,
                      aggfunc='mean',
fill_value=None, margins=False,
dropna=True,
margins_name='All')
```

Nous avons déjà utilisé les trois premiers paramètres dans les exemples précédents. Parmi les autres, deux, `fill_value` et `dropna`, sont des options pour contrôler les données manquantes. Leur utilisation est évidente et nous n'en donnerons pas d'exemple ici.

Intéressons-nous en revanche au mot-clé `aggfunc` qui détermine le type d'agrégation à réaliser. Par défaut, c'est une moyenne, *mean*. Comme avec `GroupBy`, vous pouvez indiquer une chaîne pour l'un des traitements les plus fréquents (`sum`, `mean`, `count`, `min`, `max`, *etc.*) ou bien citer le nom d'une fonction d'agrégation (`np.sum()`, `min()`, `sum()`, *etc.*). Vous pouvez même utiliser une entrée de dictionnaire qui associe une colonne à l'une des options précédentes :

```
In[8]:
titanic.pivot_table(index='sex',
columns='class',
                      aggfunc={'survived':sum,
'fare':'mean'})
```

Out[8]:

```
          fare
survived
class   First        Second       Third      First
Second  Third
sex
female  106.125798  21.970121  16.118810  91
70      72
male    67.226127   19.741782  12.661633  45
17      47
```

Notez bien que nous n'avons pas utilisé le mot-clé values ici. En effet, lorsque vous utilisez le style dictionnaire pour aggfunc, cet élément est déterminé automatiquement.

Lorsque vous avez besoin de faire calculer des sous-totaux pour chaque groupement, vous activez l'option margins :

In[9]:

```
titanic.pivot_table('survived', index='sex',
columns='class', margins=True)
```

Out[9]:

```
class   First        Second       Third      All
sex
female  0.968085   0.921053   0.500000   0.742038
male    0.368852   0.157407   0.135447   0.188908
All     0.629630   0.472826   0.242363   0.383838
```

Nous obtenons ainsi facilement en ligne All des informations concernant le taux de survie par sexe, toutes classes confondues, puis le taux de survie par classe, quel que soit le sexe, et enfin le taux de survie global qui s'établit à 38 %. Le label des données de la ligne de sous-totaux peut même être personnalisé au moyen de margins_name, la valeur par défaut étant All.

Un exemple : naissances au cours du temps

Pour un exemple concret, partons des données librement disponibles concernant les naissances aux USA. Elles sont proposées par les CDC, *Centers for Disease Control*. Les données sont disponibles à l'adresse suivante :

<https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv> (ce jeu de données a été analysé en détail par Andrew Gelman et ses collègues, comme le montre son blog, <http://andrewgelman.com/2012/06/14/cool-ass-signal-processing-using-gaussian-processes/>) :

In[10]:

```
# Commande shell pour télécharger les données :  
# !curl -O  
https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv
```

```
In[11]:
```

```
births = pd.read_csv('data/births.csv')
```

Lorsque nous demandons les cinq premières lignes du fichier, nous constatons qu'il contient tout simplement le nombre de naissances par date et par sexe :

```
In[12]: births.head()
```

```
Out[12]:
```

	year	month	day	gender	births
0	1969	1	1.0	F	4046
1	1969	1	1.0	M	4440
2	1969	1	2.0	F	4454
3	1969	1	2.0	M	4548
4	1969	1	3.0	F	4548

Démarrons notre investigation par un tableau croisé. Nous ajoutons une colonne de décennies et demandons à voir les naissances des garçons et des filles pour chaque décennie :

```
In[13]:
```

```
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade',
columns='gender',
aggfunc='sum')
```

```
Out[13]:
```

gender	F	M
decade		

1960	1753634	1846572
1970	16263075	17121550
1980	18310351	19243452
1990	19479454	20420553
2000	18229309	19106428

Nous constatons que les garçons sont plus nombreux que les filles dans toutes les décennies. Nous pouvons demander une représentation graphique en nous servant des outils de tracé de Pandas. Nous pouvons ainsi voir le nombre de naissances par an et par sexe (voir en [Figure 3.2](#) ; le [Chapitre 4](#) décrit en détail les tracés avec Matplotlib) :

In[14]:

```
%matplotlib inline
import matplotlib.pyplot as plt
sns.set()          # styles Seaborn
births.pivot_table('births', index='year',
columns='gender', aggfunc='sum').plot()
plt.ylabel('total de naissances par an');
```

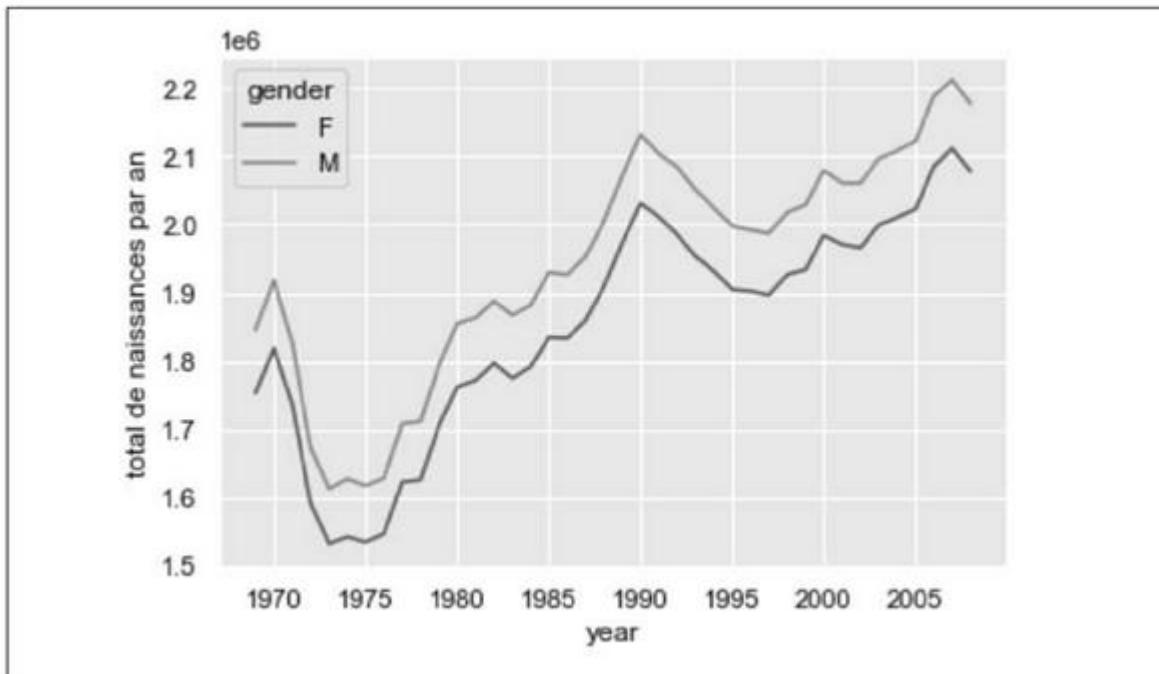


Figure 3.2 : Nombre de naissances aux USA par année et par sexe.

Nous voyons donc immédiatement les tendances des naissances par sexe avec un tableau croisé et la méthode plot(). On peut deviner qu'au cours des cinquante années passées il y a eu environ 5 % de naissances de garçons de plus que de naissances de filles.

Exploration détaillée

Bien que cela ne concerne plus spécifiquement les tableaux croisés, quelques autres traitements sont possibles pour faire parler ce jeu de données au moyen des outils Pandas déjà présentés. Nous pouvons commencer par nettoyer les données en supprimant les valeurs errantes qui peuvent être le résultat de fautes de saisie (par exemple « June 31 ») ou

de valeurs absentes. Une technique efficace pour supprimer ces éléments parasites consiste à couper les valeurs errantes. Nous allons appliquer une robuste opération dite sigma-clipping: (#1)



Vous pouvez en apprendre plus au sujet du sigma-clipping dans un livre que j'ai écrit avec Željko Ivezić, Andrew J. Connolly et Alexander Gray : *Statistics, Data Mining, and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data* (Princeton University Press, 2014).

In[15]:

```
quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.74 * (quartiles[2] - quartiles[0])
```

La dernière ligne offre une estimation solide de la moyenne de l'échantillon. La valeur 0.74 est liée à la plage interquartile d'une distribution gaussienne. Nous pouvons ainsi utiliser la méthode query() (déttaillée en fin de chapitre en même temps que eval() et query()) pour écarter par filtrage les lignes avec des naissances hors de ces valeurs limites :

In[16]:

```
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

La colonne des jours *day* est pour l'instant au format chaîne parce que certaines colonnes du jeu de données contenaient la pseudo-valeur 'null'. Nous devons la convertir vers le type entier :

In[17]:

```
# Force colonne 'day' vers integer; string au
# départ à cause des nulls
births['day'] = births['day'].astype(int)
```

Nous combinons enfin le jour, le mois et l'année pour obtenir un index de date, ce qui permet de calculer facilement le jour de semaine qui correspond à chaque ligne (nous décrivons les séries temporelles dans la prochaine section de ce chapitre) :

In[18]:

```
# Crée un index datetime avec an, mois, jour
births.index = pd.to_datetime(10000 *
                             births.year +
                             100 * births.month
                             +
                             births.day,
                             format='%Y%m%d')

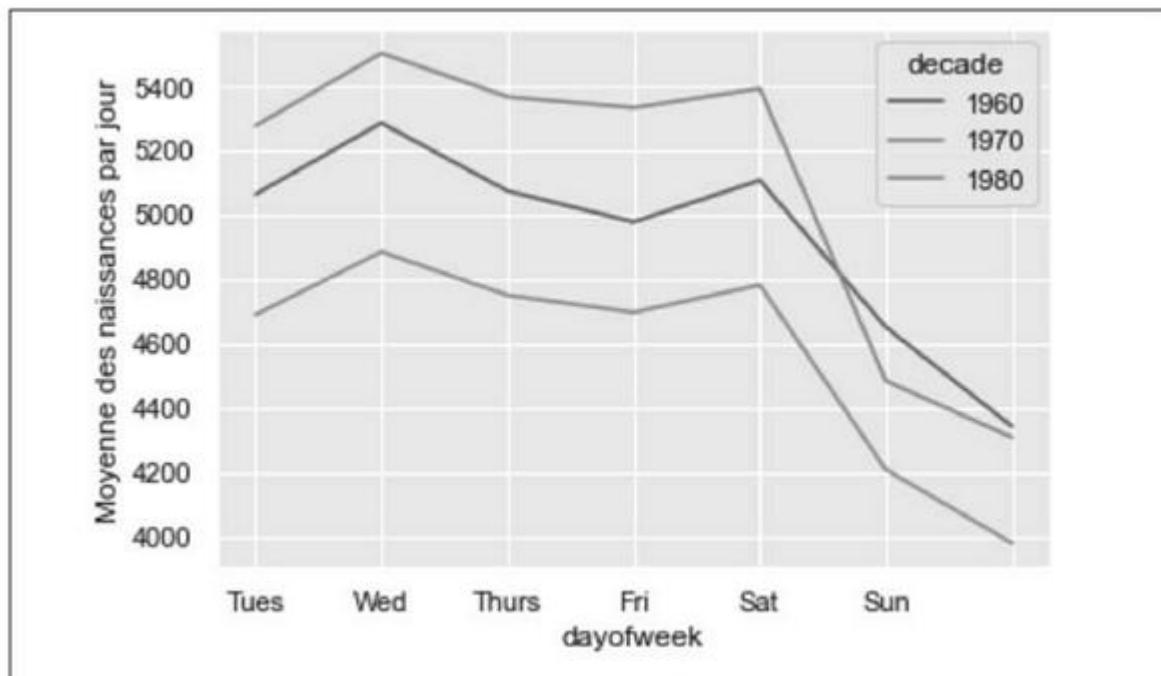
births['dayofweek'] = births.index.dayofweek
```

Cela nous permet de demander un affichage des naissances par jour de la semaine sur plusieurs décennies ([Figure 3.3](#)) :

In[19]:

```
import matplotlib.pyplot as plt
import matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                   columns='decade',
                   aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed',
                           'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('total de naissances par an');
```



[Figure 3.3](#) : Moyenne des naissances par jour de semaine sur plusieurs décennies.

Il semble qu'il y ait un peu moins de naissances le weekend ! Les valeurs pour les années 1990 et 2000 ne sont pas disponibles parce que l'institut CDC ne conserve que le mois de naissance à partir de 1989.

Il nous serait également utile de connaître la moyenne des naissances par jour de l'année. Commençons par regrouper les données séparément par mois et par jour :

In[20]:

```
births_by_date = births.pivot_table('births',  
[births.index.month, births.index.day])  
births_by_date.head()
```

Out[20]:

```
          births  
1 1 4009.225  
2 4247.400  
3 4500.900  
4 4571.350  
5 4603.625  
Name: births, dtype: float64
```

Nous obtenons un multi-index par rapport au mois et au jour. Pour simplifier le tracé graphique, convertissons ces deux index en une date en les associant avec une variable d'année servant de modèle (nous prenons soin de choisir une année bissextile afin que le 29 février soit géré correctement !) :

In[21]:

```
births_by_date.index = [pd.datetime(2012, month,  
day)]
```

```
for (month, day) in  
births_by_date.index]  
births_by_date.head()
```

Out[21]:

```
          births  
2012-01-01    4009.225  
2012-01-02    4247.400  
2012-01-03    4500.900  
2012-01-04    4571.350  
2012-01-05    4603.625  
Name: births, dtype: float64
```

En nous concentrant seulement sur le mois et le jour, nous obtenons une série temporelle qui montre le nombre moyen de naissances par jour dans l'année. Nous nous servons de la méthode plot() pour obtenir le tracé graphique ([Figure 3.4](#)). Nous pouvons en déduire quelques faits remarquables :

In[22]:

```
fig, ax = plt.subplots(figsize=(12, 4))  
births_by_date.plot(ax=ax);
```

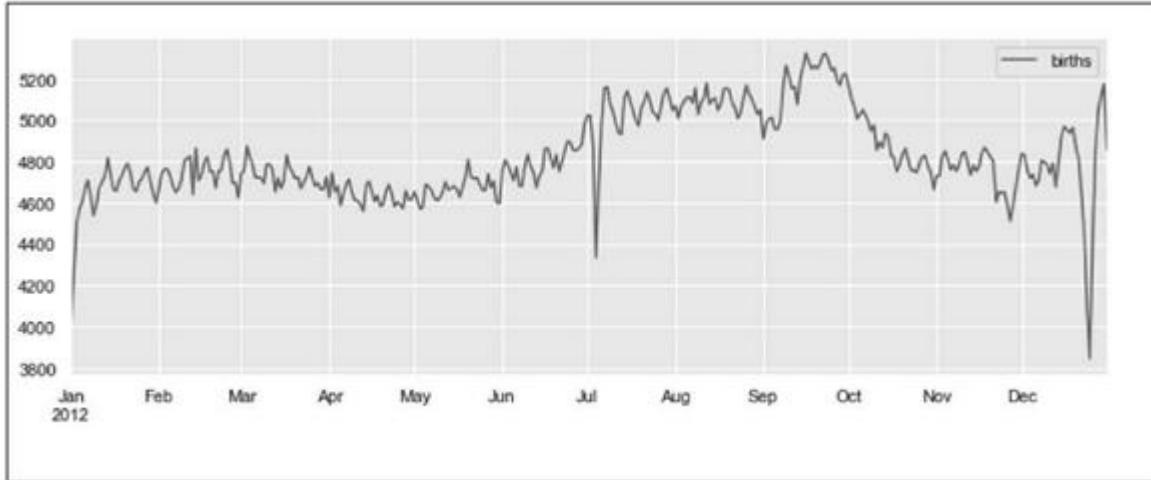


Figure 3.4 : Naissances moyennes par jour de l'année.

On remarque sur le champ la chute des naissances pendant les jours fériés aux USA (jour de l'Indépendance, fête du Travail, Thanksgiving, Noël, nouvel an). Il est peu probable qu'il y ait un effet psychosomatique ; c'est plutôt le résultat de choix de planification des naissances. Pour en savoir plus à ce sujet, voyez le poste d'Andrew Gelman (<http://bit.ly/2fZzW8K>). Nous reviendrons sur ce comportement dans la section du [Chapitre 4](#) concernant le texte et les annotations. Ce bref exemple suffit à montrer que vous pouvez combiner de nombreux outils de Python et de Pandas pour en apprendre plus au sujet des différents jeux de données. Dans les prochaines sections, nous allons voir quelques applications assez sophistiquées de ces possibilités de manipulation des données.

3.11 : Opérations vectorisées sur les chaînes

Python permet assez aisément de traiter les données de type chaîne de caractères. La librairie Pandas en tire profit et propose un jeu complet d'opérations vectorisées sur les chaînes. Ces traitements sont indispensables dans le cadre des opérations de nettoyage des données réelles que vous devez explorer. Découvrons quelques-unes des opérations Pandas concernant les chaînes en voyant comment nettoyer quelque peu un jeu de données très bruité, celui d'une série de recettes disponibles sur Internet.

Aperçu des opérations chaînes Pandas

Nous avons vu dans les sections précédentes comment les outils tels que NumPy et Pandas savaient généraliser les opérations arithmétiques. C'est ce qui permet d'appliquer vite et bien la même opération à un grand nombre d'éléments de tableau. Rappelons un exemple :

```
In[1]:  
import numpy as np  
x = np.array([2, 3, 5, 7, 11, 13])
```

```
x * 2
```

```
Out[1]:
```

```
array([ 4, 6, 10, 14, 22, 26])
```

Il s'agit d'une vectorisation des opérations qui simplifie l'écriture des traitements de tableaux. Nous n'avons en effet plus besoin de nous soucier de la taille ou de la forme du tableau, et pouvons nous concentrer sur l'opération à appliquer. NumPy n'offre pas un accès aussi simple dans le cas des tableaux de chaînes, ce qui nous oblige à utiliser une syntaxe de boucle de traitement moins compacte :

```
In[2]:
```

```
data = ['pierre', 'Paul', 'MARIE', 'gUY']
[s.capitalize() for s in data]
```

```
Out[2]:
```

```
['Pierre', 'Paul', 'Marie', 'Guy']
```

Cela peut suffire dans certains cas, mais l'opération va échouer dès qu'elle détecte une valeur manquante. Par exemple :

```
In[3]:
```

```
data = ['pierre', 'Paul', None, 'MARIE', 'gUY']
[s.capitalize() for s in data]
```

```
-----  
AttributeError  
Traceback (most recent call last)  
<ipython-input-9-bf52d8d2bce2> in <module>  
    1 data = ['pierre', 'Paul', None, 'MARIE',  
    'gUY']  
----> 2 [s.capitalize() for s in data]  
  
<ipython-input-9-bf52d8d2bce2> in <listcomp>(.0)  
    1 data = ['pierre', 'Paul', None, 'MARIE',  
    'gUY']  
----> 2 [s.capitalize() for s in data]  
  
AttributeError: 'NoneType' object has no  
attribute 'capitalize'
```

Pandas répond aux deux besoins de vectorisation des opérations chaînes et de gestion correcte des valeurs manquantes au moyen de l'attribut nommé str des objets Series et Index lorsqu'ils contiennent des chaînes. Nous pouvons par exemple créer un objet Series de Pandas à partir de nos données précédentes :

```
In[4]:  
import pandas as pd  
names = pd.Series(data)  
names
```

```
Out[4]:  
0    pierre
```

```
1      Paul
2      None
3    MARIE
4      gUY
dtype: object
```

Nous pouvons ensuite appeler une méthode pour normaliser toutes les entrées avec une majuscule sans nous soucier de la valeur manquante :

```
In[5]:
names.str.capitalize()
```

```
Out[5]:
0    Pierre
1      Paul
2      None
3    Marie
4      Guy
dtype: object
```

Vous pouvez utiliser l'aide à la saisie par la touche Tab pour l'attribut str afin d'obtenir la liste de toutes les méthodes de vectorisation sur chaîne disponibles dans Pandas.

Liste des méthodes de chaînes Pandas

Si vous avez un minimum d'expérience avec les manipulations de chaînes en Python, vous n'aurez aucun mal à exploiter la syntaxe des chaînes Pandas. C'est pourquoi nous nous limitons à fournir la liste des méthodes disponibles. Nous verrons dans un deuxième temps quelques subtilités remarquables. Les exemples de cette section utilisent la série de noms de comédiens suivante :

```
In[6]: monty = pd.Series(['Graham Chapman',
 'John Cleese',
 'Terry Gilliam', 'Eric
 Idle',
 'Terry Jones', 'Michael
 Palin'])
```

Méthodes similaires aux méthodes de chaînes Python

Quasiment toutes les méthodes de chaînes standard de Python existent dans une version pour chaînes vectorisées Pandas. Voici la liste des méthodes de str correspondantes :

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()

```
center()    rfind()        isalnum()      isdecimal()
zfill()     index()        isalpha()       split()
strip()     rindex()       isdigit()      rsplit()
rstrip()    capitalize()   isspace()      partition()
lstrip()    swapcase()    istitle()      rpartition()
```

Notez que les valeurs renvoyées sont variables. Par exemple, `lower()` renvoie une série de chaînes :

```
In[7]: monty.str.lower()
```

```
Out[7]:
```

```
0  graham chapman
1  john cleese
2  terry gilliam
3  eric idle
4  terry jones
5  michael palin
dtype: object
```

D'autres méthodes renvoient des valeurs numériques :

```
In[8]: monty.str.len()
```

```
Out[8]:
```

```
0  14
1  11
2  13
3  9
4  11
```

```
5  13  
dtype: int64
```

D'autres enfin renvoient des valeurs booléennes :

```
In[9]: monty.str.startswith('T')
```

```
Out[9]:
```

```
0  False  
1  False  
2  True  
3  False  
4  True  
5  False  
dtype: bool
```

Enfin, certaines méthodes renvoient une liste ou valeur composite pour chaque élément :

```
In[10]: monty.str.split()
```

```
Out[10]
```

```
0  [Graham, Chapman]  
1  [John, Cleese]  
2  [Terry, Gilliam]  
3  [Eric, Idle]  
4  [Terry, Jones]  
5  [Michael, Palin]  
dtype: object
```

Nous verrons d'autres manipulations d'un objet de type série de listes dans la suite.

Méthodes exploitant les expressions régulières

Vous disposez d'une autre série de méthodes qui reconnaissent une expression régulière, ce qui permet d'analyser le contenu d'un élément chaîne. Elles obéissent à certaines conventions d'appel API du module standard de Python appelé `re` ([Tableau 3.4](#)).

[Tableau 3.4](#) : Équivalence entre les méthodes de Pandas et les fonctions du module `re` de Python.

Méthode	Description
<code>match()</code>	Applique <code>re.match()</code> à chaque élément et renvoie un booléen.
<code>extract()</code>	Applique <code>re.match()</code> à chaque élément et renvoie des chaînes des groupes trouvés.
<code>findall()</code>	Applique <code>re.findall()</code> à chaque élément.
<code>replace()</code>	Remplace les occurrences du motif par une chaîne.
<code>contains()</code>	Appelle <code>re.search()</code> pour chaque élément et renvoie un booléen.
<code>count()</code>	Compte les occurrences du motif.
<code>split()</code>	Équivalent à <code>str.split()</code> , mais accepte les regexps.
<code>rsplit()</code>	Équivalent à <code>str.rsplit()</code> , mais accepte les regexps.

Ces méthodes permettent de réaliser des traitements très intéressants. Voici par exemple comment récupérer le prénom de chaque comédien en demandant de capturer le premier groupe de caractères continus dans chaque élément :

```
In[11]: monty.str.extract('([A-Za-z]+)')
```

Out[11]:

```
0    Graham  
1      John  
2    Terry  
3     Eric  
4    Terry  
5  Michael  
dtype: object
```

Pour un traitement un peu plus complexe, cherchons tous les noms qui commencent et se terminent par une consonne. Nous utilisons les métacaractères de début de chaîne (\wedge) et de fin de chaîne ($\$$) :

```
In[12]: monty.str.findall(r'^[^AEIOU].*  
[^aeiou]$')
```

Out[12]:

```
0  [Graham Chapman]  
1          []  
2  [Terry Gilliam]
```

```
3          []
4      [Terry Jones]
5      [Michael Palin]
dtype: object
```

Cette possibilité d'appliquer des expressions régulières aux entrées d'un objet Series ou DataFrame ouvre évidemment de nombreuses possibilités d'analyse et de préparation des jeux de données.

Autres méthodes chaînes

Vous disposez enfin d'une série de méthodes pour réaliser certaines opérations particulières ([Tableau 3.5](#)).

[Tableau 3.5](#) : Autres méthodes chaînes de Pandas.

Méthode	Description
get()	Indexe les éléments.
slice()	Tranche chaque élément.
slice_replace()	Remplace la tranche dans chaque élément par la valeur fournie.
cat()	Concatène des chaînes.
repeat()	Répète des valeurs.
normalize()	Renvoie le code Unicode d'une chaîne.
pad()	Ajoute de l'espace à gauche, à droite ou aux deux extrémités d'une chaîne.

wrap()	Brise une longue chaîne en lignes de longueur limitée.
join()	Réunit les chaînes de chaque élément de la série avec un séparateur fourni.
get_dummies()	Extrait des variables de simulation en tant que DataFrame.

Accès et sélection vectorisés aux éléments. Vous pouvez accéder de façon vectorisée aux différents éléments, en particulier grâce aux deux méthodes get() et slice(). Nous pouvons facilement extraire les trois premiers caractères de chaque chaîne en écrivant str.slice(0, 3). En ce qui concerne cette opération simple, le même résultat est obtenu au moyen de la syntaxe d'indexation habituelle de Python, df.str[0 : 3] :

```
In[13]:  
monty.str[0:3]
```

```
Out[13]:  
0    Gra  
1    Joh  
2    Ter  
3    Eri  
4    Ter  
5    Mic  
dtype: object
```

La même équivalence fonctionnelle concerne l'indexation, avec df.str.get(i) ou df.str[i].

Les deux méthodes `get()` et `slice()` permettent aussi d'accéder aux éléments de tableaux tels qu'ils sont renvoyés par la méthode `split()`. Voici par exemple comment combiner `split()` et `get()` pour récupérer le patronyme de chaque entrée, c'est-à-dire le dernier mot :

```
In[14]: monty.str.split().str.get(-1)
```

```
Out[14]:
```

```
0    Chapman
1    Cleese
2    Gilliam
3      Idle
4    Jones
5    Palin
dtype: object
```

Variables indicatrices. La méthode `get_dummies()` mérite quelques explications. Elle est utile lorsque les données comportent une ou plusieurs colonnes dont la valeur est symbolique, sous forme d'un indicateur. Dans l'exemple suivant, nous utilisons plusieurs lettres majuscules pour coder différentes informations spécifiques : A = « Né aux USA », B = « Né au Royaume-Uni », C = « Aime le fromage », D = « Aime le jambon » :

```
In[15]:
```

```
full_monty = pd.DataFrame({'name': monty,
```

```
        'info': ['B|C|D',
'B|D', 'A|C', 'B|D', 'B|C',
'B|C|D']})
full_monty
```

Out[15]:

```
    info name
0  B|C|D Graham Chapman
1  B|D      John Cleese
2  A|C      Terry Gilliam
3  B|D      Eric Idle
4  B|C      Terry Jones
5  B|C|D  Michael Palin
```

La méthode `get_dummies()` permet de créer en un geste un objet DataFrame à partir de ses variables indicatrices :

In[16]:

```
full_monty['info'].str.get_dummies('|')
```

Out[16]:

```
   A  B  C  D
0  0  1  1  1
1  0  1  0  1
2  1  0  1  0
3  0  1  0  1
4  0  1  1  0
5  0  1  1  1
```

En vous servant de ces cinquante méthodes comme blocs de construction, vous pouvez bâtir des traitements de chaînes très sophistiqués pour nettoyer vos données d'entrées.

Nous n'irons pas plus loin dans la découverte de ces méthodes. Je vous invite à consulter la documentation en ligne de Pandas (<http://pandas.pydata.org/pandas-docs/stable/text.html>) ou à vous servir des ressources fournies en fin de chapitre.

3.12 : Traitement des données temporelles

Au départ, Pandas a été conçu pour servir dans un contexte de modélisation financière. Vous pouvez donc vous attendre à découvrir un jeu complet d'outils pour traiter les dates, les heures et toutes les données basées sur le temps. Les données de date et d'heure sont disponibles dans différents formats :

- Des *horodatages* (*time stamps*) qui correspondent à des moments précis (par exemple, July 4th, 2015, at 7:00 a.m.).
- Des *intervalles* et des périodes temporelles qui désignent des durées entre un début et une fin, par exemple l'année 2015 en entier. En général, une période est un intervalle temporel spécial alors qu'un intervalle est un découpage de longueur uniforme sans chevauchement (par exemple des périodes de 24 heures qui constituent des jours).
- Des *durées absolues* ou *deltas* qui désignent une durée exacte, par exemple 22,56 secondes.

Nous allons voir comment travailler avec ces différents types de données dans Pandas. Cette section ne prétend pas décrire de façon exhaustive les outils temporels de Python ou de Pandas. Nous allons rappeler quels outils sont disponibles pour les dates et heures dans Python, avant de présenter ceux qu'y ajoute Pandas. Nous creuserons ensuite des points particuliers avant de passer par quelques exemples de manipulations des données temporelles dans Pandas.

Les dates et les heures dans Python

Le langage Python offre un certain nombre de représentations pour les dates, les heures, les périodes et les durées. Les outils temporels de Pandas sont beaucoup plus efficaces dans le domaine de la datalogie, mais il n'est pas inutile de voir quelles relations ils maintiennent avec les outils standard de Python.

Dates et heures natives (datetime et dateutil)

Les objets temporels de Python sont définis dans le module standard `datetime`. Vous pouvez créer des traitements intéressants en le combinant avec le module `dateutil`. Voici par exemple comment construire manuellement une date à partir du type `datetime` :

In[1]:

```
from datetime import datetime  
datetime(year=2015, month=7, day=4)
```

Out[1]: datetime.datetime(2015, 7, 4, 0, 0)

Avec le module dateutil, vous pouvez analyser une date dans différents formats de chaînes :

In[2]:

```
from dateutil import parser  
date = parser.parse("4th of July, 2015")  
date
```

Out[2]: datetime.datetime(2015, 7, 4, 0, 0)

Une fois que vous disposez d'un objet de type datetime, vous pouvez par exemple s'afficher le jour de la semaine :

In[3]: date.strftime('%A')

Out[3]: 'Saturday'

Dans le dernier exemple, nous nous servons d'un des codes de format de chaîne standard pour les dates (<<%A>>). Pour en savoir plus, consultez la section de la méthode strftime (<https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior>), la section de documentation de datetime dans Python (<https://docs.python.org/3/library/datetime.html>) et

celle de dateutil (<http://labix.org/python-dateutil>). Ne négligez pas enfin le paquetage nommé pytz (<http://pytz.sourceforge.net/>) qui réunit des outils pour mieux gérer la principale cause de migraine dans le domaine temporel que sont les fuseaux horaires, *time zones*.

Les deux modules datetime et dateutil sont attrayants parce qu'ils offrent une grande souplesse et simplicité d'écriture pour toutes sortes de traitements concernant les dates et les heures. Le problème surgit lorsque vous devez traiter de vastes volumes de données temporelles. Le même souci de performances que pour les grands tableaux de données numériques s'invite dans le débat. De même que les tableaux numériques de style NumPy ont résolu ce problème de performances pour les données numériques, les objets temporels typés que propose NumPy résolvent les soucis de performances des traitements temporels.

Tableaux temporels typés : le type datetime64 de NumPy

La lourdeur d'exécution du format datetime standard de Python a poussé les développeurs de NumPy à inventer une nouvelle série de types de données temporelles. Le type nommé datetime64 encode les dates sous forme d'entiers sur 64 bits, ce qui permet de stocker une date de façon très

compacte. Ce type `datetime64` doit être utilisé avec un format d'entrée très spécifique :

In[4]:

```
import numpy as np
date = np.array('2015-07-04',
dtype=np.datetime64)
date
```

Out[4]: array(datetime.date(2015, 7, 4),
dtype='datetime64[D']')

Une fois que la date a ainsi été retypée, nous pouvons immédiatement appliquer des opérations vectorisées :

In[5]:

```
date + np.arange(12)
```

Out[5]:

```
array(['2015-07-04', '2015-07-05', '2015-07-06',
'2015-07-07',
'2015-07-08', '2015-07-09', '2015-07-10',
'2015-07-11',
'2015-07-12', '2015-07-13', '2015-07-14',
'2015-07-15'],
dtype='datetime64[D']')
```

Du fait qu'un tableau `datetime64` de NumPy contient un seul type, les opérations demandées peuvent être réalisées bien

plus vite qu'avec un objet `datetime` standard de Python, ce qui se ressent d'autant plus que le tableau est volumineux (nous avons vu ce genre de vectorisation lorsque nous avons abordé les fonctions `ufuncs` dans le [Chapitre 2](#)).

Il faut savoir que les objets `datetime64` et `timedelta64` dépendent d'une unité temporelle fondamentale. Du fait que le codage se fait sur 64 bits, la plage de temps qui peut être encodée est égale à 2^{64} fois l'unité fondamentale. Autrement dit, vous devez faire un choix entre précision temporelle et taille de la plage de valeurs possible pour `datetime64`.

Si vous avez besoin par exemple d'une précision d'une nanoseconde, vous ne pourrez coder que jusqu'à 2^{64} nanosecondes, ce qui équivaut à un peu moins de 600 ans. Notez que NumPy va déduire l'unité appropriée à partir de valeurs fournies. Voici par exemple une valeur avec l'unité jour :

```
In[6]: np.datetime64('2015-07-04')
```

```
Out[6]: numpy.datetime64('2015-07-04')
```

Voici une autre valeur dont l'unité est égale à la minute :

```
In[7]: np.datetime64('2015-07-04 12:00')
```

```
Out[7]: numpy.datetime64('2015-07-04T12:00')
```

Notez que le fuseau horaire est aligné automatiquement avec celui en vigueur sur l'ordinateur qui exécute le code. Par ailleurs, vous pouvez changer l'unité fondamentale en la spécifiant en second paramètre. Voici par exemple comment forcer l'unité nanoseconde :

```
In[8]: np.datetime64('2015-07-04 12:59:59.50',  
'ns')
```

```
Out[8]: numpy.datetime64('2015-07-  
04T12:59:59.500000000')
```

Le [Tableau 3.6](#) issu de la documentation de `datetime64` de NumPy

(<http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html>)

rappelle les codes de format disponibles avec les plages temporelles relatives et absolues correspondantes.

Tableau 3.6 : Codes des unités temporelles de `datetime64`.

Code	Description	Envergure (relative)	Envergure (absolue)
Y	Année	$\pm 9.2\text{e}18$ ans	[9.2e18 av. J.-C., 9.2e18 apr. J.-C.]
M	Mois	$\pm 7.6\text{e}17$ ans	[7.6e17 av. J.-C., 7.6e17 apr. J.-C.]
W	Semaine	$\pm 1.7\text{e}17$ ans	[1.7e17 av. J.-C., 1.7e17 apr. J.-C.]
D	Jour	$\pm 2.5\text{e}16$ ans	[2.5e16 av. J.-C., 2.5e16 apr. J.-C.]

h	Heure	$\pm 1.0\text{e}15$ ans	[$1.0\text{e}15$ av. J.-C., $1.0\text{e}15$ apr. J.-C.]
m	Minute	$\pm 1.7\text{e}13$ ans	[$1.7\text{e}13$ av. J.-C., $1.7\text{e}13$ apr. J.-C.]
s	Seconde	$\pm 2.9\text{e}12$ ans	[$2.9\text{e}9$ av. J.-C., $2.9\text{e}9$ apr. J.-C.]
ms	Milliseconde	$\pm 2.9\text{e}9$ ans	[$2.9\text{e}6$ av. J.-C., $2.9\text{e}6$ apr. J.-C.]
us	Microseconde	$\pm 2.9\text{e}6$ ans	[290301 av. J.-C., 294241 apr. J.-C.]
ns	Nanoseconde	± 292 ans	[1678 apr. J.-C., 2262 apr. J.-C.]
ps	Picoseconde	± 106 jours	[1969 apr. J.-C., 1970 apr. J.-C.]
fs	Femtoseconde	± 2.6 heures	[1969 apr. J.-C., 1970 apr. J.-C.]
as	Attoseconde	± 9.2 secondes	[1969 apr. J.-C., 1970 apr. J.-C.]

Pour la majorité des travaux dans le monde réel, le format qui semble le plus approprié est `datetime64[ns]`, qui offre une plage de valeurs suffisante avec une précision correcte.

Le type temporel `datetime64` résout certains des défauts du type `datetime` standard de Python, mais il ne dispose pas d'un aussi grand nombre de méthodes et de fonctions, notamment celles de `dateutil`. Vous en saurez plus dans la documentation de `datetime64` (<http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html>).

Les dates et les heures de Pandas : le meilleur des deux mondes

La librairie Pandas va plus loin en réutilisant les outils que nous venons de décrire afin de proposer un objet nommé `Timestamp` qui combine la souplesse de `datetime` et `dateutil` avec l'efficacité de l'interface vectorisée de `numpy.datetime64`. Pandas peut construire à partir d'un groupe d'objets `Timestamp` un objet `DatetimeIndex` avec lequel il devient possible d'indexer les données dans un objet `Series` ou `DataFrame`. Nous en verrons plusieurs exemples.

Voici comment profiter de Pandas pour reformuler la démonstration précédente. Nous analysons une chaîne de dates ayant un format particulier en profitant des codes de format pour afficher le jour de la semaine :

In[9]:

```
import pandas as pd  
date = pd.to_datetime("4th of July, 2015")  
date
```

Out[9]: `Timestamp('2015-07-04 00:00:00')`

In[10]: `date.strftime('%A')`

Out[10]: `'Saturday'`

Nous pouvons appliquer directement des opérations vectorisées dans le style NumPy sur ce même objet :

In[11]:

```
date + pd.to_timedelta(np.arange(12), 'D')
```

Out[11]:

```
DatetimeIndex(['2015-07-04', '2015-07-05',
                 '2015-07-06', '2015-07-07',
                 '2015-07-08', '2015-07-09', '2015-
07-10', '2015-07-11',
                 '2015-07-12', '2015-07-13', '2015-
07-14', '2015-07-15'],
                dtype='datetime64[ns]', freq=None)
```

Découvrons dans la prochaine sous-section quelques opérations plus sophistiquées manipulant des séries temporelles avec les outils de Pandas.

Séries temporelles de Pandas : chrono-indexation

Les outils temporels de Pandas montrent vraiment leurs avantages quand vous avez besoin d'indexer des données en fonction du temps. Voici comment construire un objet Series ainsi chrono-indexé :

```
In[12]:  
index = pd.DatetimeIndex(['2014-07-04', '2014-  
08-04',  
                           '2015-07-04', '2015-08-  
04'])  
data = pd.Series([0, 1, 2, 3], index=index)  
data
```

```
Out[12]:  
2014-07-04 0  
2014-08-04 1  
2015-07-04 2  
2015-08-04 3  
dtype: int64
```

Les données se trouvent dorénavant dans un objet Series et nous pouvons utiliser l'une des syntaxes d'indexation de Series déjà rencontrées, en transmettant les valeurs qui doivent être transformées en dates :

```
In[13]:  
data['2014-07-04':'2015-07-04']
```

```
Out[13]:  
2014-07-04 0  
2014-08-04 1  
2015-07-04 2  
dtype: int64
```

Vous disposez d'opérations d'indexation spécifiques aux dates, ce qui permet par exemple de fournir une année en vue d'obtenir la sélection de toutes les données pour cette année :

```
In[14]:  
data['2015']
```

```
Out[14]:  
2015-07-04 2  
2015-08-04 3  
dtype: int64
```

Nous verrons plus loin des exemples d'utilisation des dates comme index. Voyons d'abord quelles sont les structures de données temporelles disponibles.

Structures de données temporelles de Pandas

Voici les structures de données temporelles fondamentales qui sont définies par Pandas :

- Pour incarner un moment unique dans le temps (horodatage), la librairie Pandas propose le type nommé `Timestamp`. Il vient en remplacement du type natif de Python nommé `datetime`, et se fonde sur le type

sophistiqué et efficace `numpy.datetime64`. La structure d'index associée se nomme `DatetimeIndex`.

- Pour les périodes temporelles, Pandas propose le type `Period` qui code un intervalle temporel à fréquence fixe toujours basé sur `numpy.datetime64`. La structure d'index associée se nomme `PeriodIndex`.
- Pour les durées ou quantités temporelles, Pandas propose le type `Timedelta` qui remplace avantageusement le type natif de Python `datetime.timedelta`, et se base aussi sur `numpy.timedelta64`. La structure d'index associée porte le nom `TimedeltaIndex`.

Parmi ces chrono-objets, les deux fondamentaux sont les deux premiers : `Timestamp` et `DatetimeIndex`. Il est possible de les invoquer directement, mais en général, vous utiliserez plutôt la fonction `pd.to_datetime()` qui est capable de prendre en compte de nombreux formats. Lorsque vous lui transmettez une seule date, la fonction renvoie un objet `Timestamp`. Lorsque vous renvoyez une série de dates, elle renvoie par défaut un index `DatetimeIndex` :

In[15] :

```
dates = pd.to_datetime([datetime(2015, 7, 3),  
'4th of July, 2015',  
'2015-Jul-6', '07-07-
```

```
2015', '20150708'])
```

```
dates
```

```
Out[15]:
```

```
DatetimeIndex(['2015-07-03', '2015-07-04',
'2015-07-06', '2015-07-07',
'2015-07-08'],
dtype='datetime64[ns]', freq=None)
```

Un index de moment DatetimeIndex peut être converti en index de période PeriodIndex au moyen de la fonction `to_period()`, en spécifiant un code de fréquence. Dans l'exemple, nous utilisons le code 'D' qui stipule une fréquence quotidienne par jour calendaire :

```
In[16]:
```

```
dates.to_period('D')
```

```
Out[16]:
```

```
PeriodIndex(['2015-07-03', '2015-07-04', '2015-
07-06', '2015-07-07',
'2015-07-08'],
dtype='int64', freq='D')
```

Vous obtenez un objet TimedeltaIndex lorsque vous demandez la soustraction d'une date à une autre :

```
In[17]:
```

```
dates - dates[0]
```

```
Out[17]:  
TimedeltaIndex(['0 days', '1 days', '3 days', '4  
days', '5 days'],  
dtype='timedelta64[ns]', freq=None)
```

Séquences temporelles homogènes avec pd.date_range()

Pandas simplifie la création de séquences de dates homogènes au moyen de plusieurs fonctions. pd.date_range() sert aux horodatages, pd.period_range() aux périodes et pd.timedelta_

range() aux durées ou deltas. Nous savons que les méthodes range() de Python et np.arange() de NumPy travaillent à partir d'un point de départ, d'un point d'arrivée et d'une taille de pas facultative pour produire une séquence temporelle. Il en va de même pour pd.date_range() qui doit recevoir en entrée une date de départ, une date de fin et un code de fréquence facultatif pour produire une séquence homogène. La fréquence par défaut est de un jour :

```
In[18]:  
pd.date_range('2015-07-03', '2015-07-10')
```

```
Out[18]:  
DatetimeIndex(['2015-07-03', '2015-07-04',  
'2015-07-05', '2015-07-06',
```

```
'2015-07-07', '2015-07-08', '2015-  
07-09', '2015-07-10'],  
dtype='datetime64[ns]', freq='D')
```

Vous pouvez également ne spécifier que la date de début et un nombre de périodes, donc sans date de fin :

In[19]:

```
pd.date_range('2015-07-03', periods=8)
```

Out[19]:

```
DatetimeIndex(['2015-07-03', '2015-07-04',  
'2015-07-05', '2015-07-06',  
               '2015-07-07', '2015-07-08', '2015-  
07-09', '2015-07-10'],  
dtype='datetime64[ns]', freq='D')
```

Vous contrôlez la largeur de période avec le paramètre freq qui vaut D par défaut. Voici comment générer une série d'horodatages :

In[20]:

```
pd.date_range('2015-07-03', periods=8, freq='H')
```

Out[20]:

```
DatetimeIndex(['2015-07-03 00:00:00', '2015-07-  
03 01:00:00',  
               '2015-07-03 02:00:00', '2015-07-03  
03:00:00',  
               '2015-07-03 04:00:00', '2015-07-03
```

```
05:00:00',
       '2015-07-03 06:00:00', '2015-07-03
07:00:00'],
      dtype='datetime64[ns]', freq='H')
```

Pour créer une séquence homogène de périodes ou de deltas temporels, vous vous servez des fonctions `pd.period_range()` et `pd.timedelta_range()`. Voici comment produire quelques périodes mensuelles :

In[21]:

```
pd.period_range('2015-07', periods=8, freq='M')
```

Out[21]:

```
PeriodIndex(['2015-07', '2015-08', '2015-09',
'2015-10', '2015-11',
       '2015-12', '2016-01', '2016-02'],
      dtype='period[M]', freq='M')
```

Et voici une séquence de durées horaires :

In[22]:

```
pd.timedelta_range(0, periods=10, freq='H')
```

Out[22]:

```
TimedeltaIndex(['0 days 00:00:00', '0 days
01:00:00', '0 days 02:00:00',
       '0 days 03:00:00', '0 days
04:00:00', '0 days 05:00:00',
       '0 days 06:00:00', '0 days
07:00:00', '0 days 08:00:00',
       '0 days 09:00:00'],
      dtype='timedelta64[ns]', freq='H')
```

```

07:00:00', '0 days 08:00:00',
'0 days 09:00:00'],
dtype='timedelta64[ns]', freq='H')

```

Tout cela suppose de connaître les codes de fréquence reconnus par Pandas, ce qui fait l'objet de la prochaine section.

Codes de fréquence et décalages

Les outils temporels de Pandas sont fortement dépendants de la fréquence choisie ou d'un décalage de date. Nous avons déjà découvert les codes D pour le jour calendaire et H pour l'heure. Le [Tableau 3.7](#) décrit les principaux codes disponibles.

Tableau 3.7 : Liste des codes de fréquence Pandas.

Code	Description	Code	Description
D	Jour calendaire	B	Jour ouvré
W	Hebdomadaire		
M	Fin de mois	BM	Fin de mois Business
Q	Fin de trimestre	BQ	Fin de trimestre Business
A	Fin d'année	BA	Fin d'année Business
H	Heure	BH	Heures ouvrées
T	Minute		

S	Seconde
L	Milliseconde
U	Microseconde
N	Nanoseconde

Notez que les fréquences mensuelles, trimestrielles et annuelles sont alignées sur la fin de la période. Pour les faire démarrer en début de période, il faut ajouter S en deuxième lettre ([Tableau 3.8](#)).

[Tableau 3.8](#) : Liste des codes de fréquence pour indexation en début de période.

Code	Description
MS	Début de mois
BMS	Début de mois Business
QS	Début de trimestre
BQS	Début de trimestre Business
AS	Début d'année
BAS	Début d'année Business

Vous pouvez même personnaliser le mois servant de charnière de période trimestrielle ou annuelle en ajoutant en suffixe le code de mois en anglais sur trois lettres :

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

De même, vous pouvez changer le jour de début de semaine en ajoutant un code de jour en anglais sur trois lettres :

- W-SUN, W-MON, W-TUE, W-WED, etc.

Vous pouvez enfin combiner les codes de fréquence pour obtenir des fréquences spécifiques. Voici par exemple comment combiner le code d'heure H et le code de minute T pour générer des périodes de 2 h 30 :

In[23]:

```
pd.timedelta_range(0, periods=9, freq="2H30T")
```

Out[23]:

```
TimedeltaIndex(['0 days 00:00:00', '0 days  
02:30:00', '0 days 05:00:00',  
               '0 days 07:30:00', '0 days  
10:00:00', '0 days 12:30:00',  
               '0 days 15:00:00', '0 days  
17:30:00', '0 days 20:00:00',  
               '0 days 22:30:00', '1 days  
01:00:00', '1 days 03:30:00'],  
              dtype='timedelta64[ns]',  
              freq='150T')
```

Tous ces codes correspondent à des fonctions de décalage temporelles spécifiques qui sont réunies dans le module pd.tseries.offsets. Voici par exemple comment créer un décalage de jour ouvré directement :

In[24]:

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=5,
freq=BDay())
```

Out[24]:

```
DatetimeIndex(['2015-07-01', '2015-07-02',
'2015-07-03', '2015-07-06',
'2015-07-07'],
dtype='datetime64[ns]', freq='B')
```

Pour tous détails au sujet des fréquences et des décalages, vous irez voir la section abordant les objets DateOffset dans la documentation en ligne de Pandas (<http://pandas.pydata.org/pandas-docs/stable/timeseries.html#dateoffset-objects>).

Rééchantillonage, décalage et fenêtrage

Un apport essentiel des outils temporels de Pandas est la possibilité qu'ils offrent de se servir de dates et d'heures

comme index pour organiser et manipuler les données. Les avantages habituels de l'indexation (alignement automatique pendant les traitements, tranchage et accès facile aux données, etc.) restent applicables ; Pandas y ajoute plusieurs opérations spécifiques aux données chronologiques.

Nous allons découvrir certaines de ces fonctions en utilisant en exemple un cours de Bourse. La librairie Pandas a été développée principalement dans le cadre d'activités financières, ce qui explique l'existence d'outils spécifiques à ce domaine. Vous disposez par exemple du paquetage complémentaire nommé pandas-datareader (que vous pouvez insérer avec conda install pandas-datareader) qui sait rapatrier lui-même des données financières auprès de différentes sources, et notamment Yahoo Finance. Voici comment charger l'historique des prix de clôture de l'action Apple :

In[25] :

```
from pandas_datareader import data
aapl = data.DataReader('AAPL', start='2004',
end='2018',
data_source='yahoo')
aapl.head()
```

Out[25] :

High	Low	Open
------	-----	------

Close	Volume	Adj Close
Date		
2004-01-02	1.553571	1.512857
1.520000	36153292.0	1.449212
2004-01-05	1.599285	1.530000
1.583571	97228992.0	1.509823
2004-01-06	1.601428	1.550714
1.577857	125841674.0	1.504375
2004-01-07	1.630714	1.566428
1.613571	145578342.0	1.538426
2004-01-08	1.695000	1.617857
1.668571	113857212.0	1.590864

Pour nous simplifier la vie, nous ne conservons que le cours de clôture :

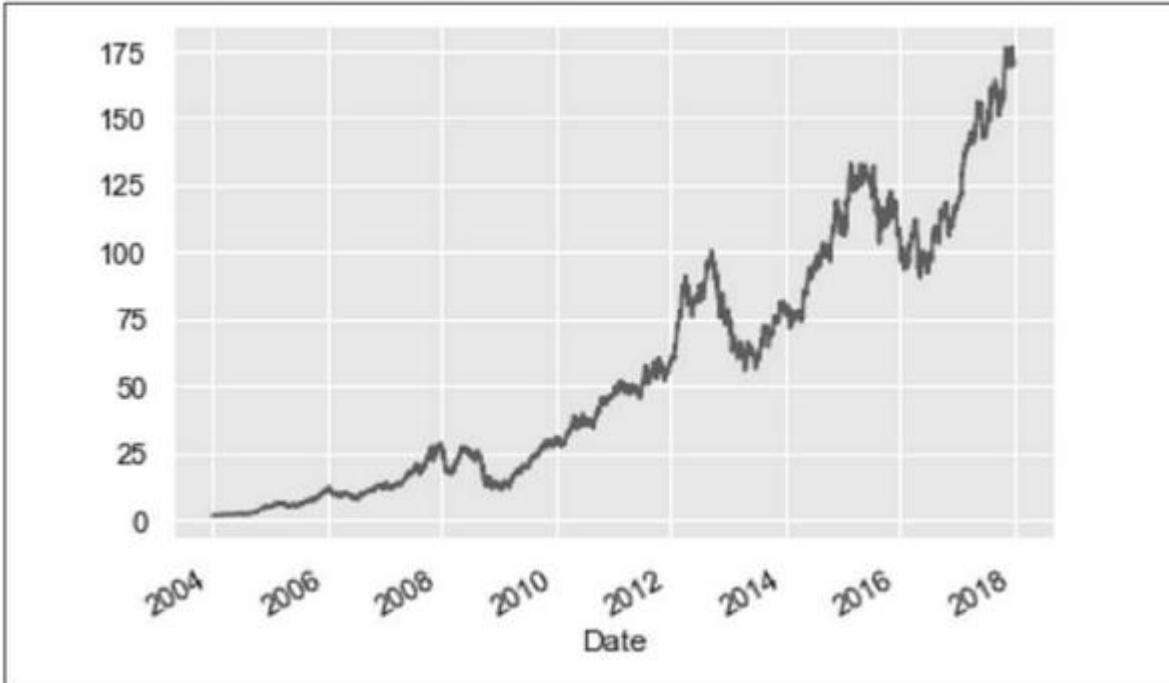
```
In[26]: aapl = aapl['Close']
```

Demandons immédiatement un graphique au moyen de la méthode `plot()`, après avoir réalisé la configuration Matplotlib indispensable ([Figure 3.5](#)) :

```
In[27]:
```

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
```

```
In[28]: aapl.plot();
```



[Figure 3.5](#) : Cours de clôture d'une action au cours du temps.

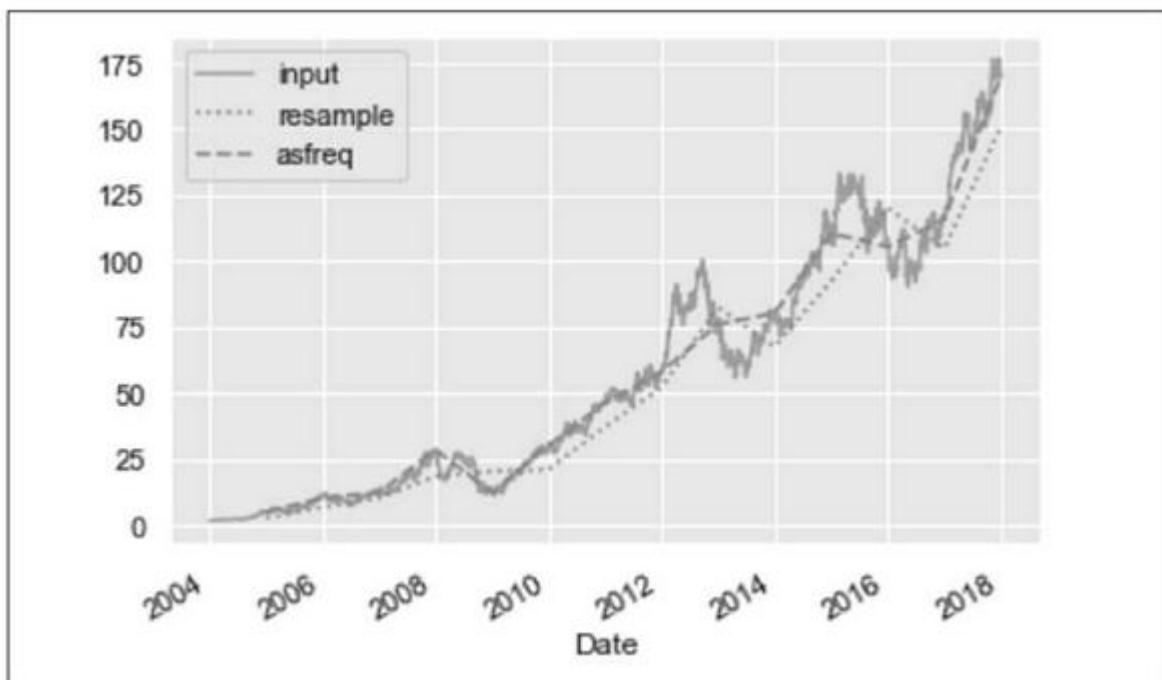
Rééchantillonage et conversion de fréquences

Les données temporelles ont souvent besoin d'être rééchantillonnées, c'est-à-dire reformulées à fréquence supérieure ou inférieure. Vous disposez à cet effet des deux méthodes `resample()` et `asfreq()`. Elles se distinguent notamment par le fait que `resample()` est une agrégation de données alors que `asfreq()`, qui est plus simple, est une sélection de données.

Comparons l'effet des deux méthodes avec notre cours de Bourse, pour un sous-échantillonage. Nous demandons un rééchantillonement pour la fin de chaque année fiscale ([Figure 3.6](#)) :

In[29]:

```
aapl.plot(alpha=0.5, style='--')
aapl.resample('BA').mean().plot(style=':')
aapl.asfreq('BA').plot(style='---');
plt.legend(['input', 'resample', 'asfreq'],
          loc='upper left');
```



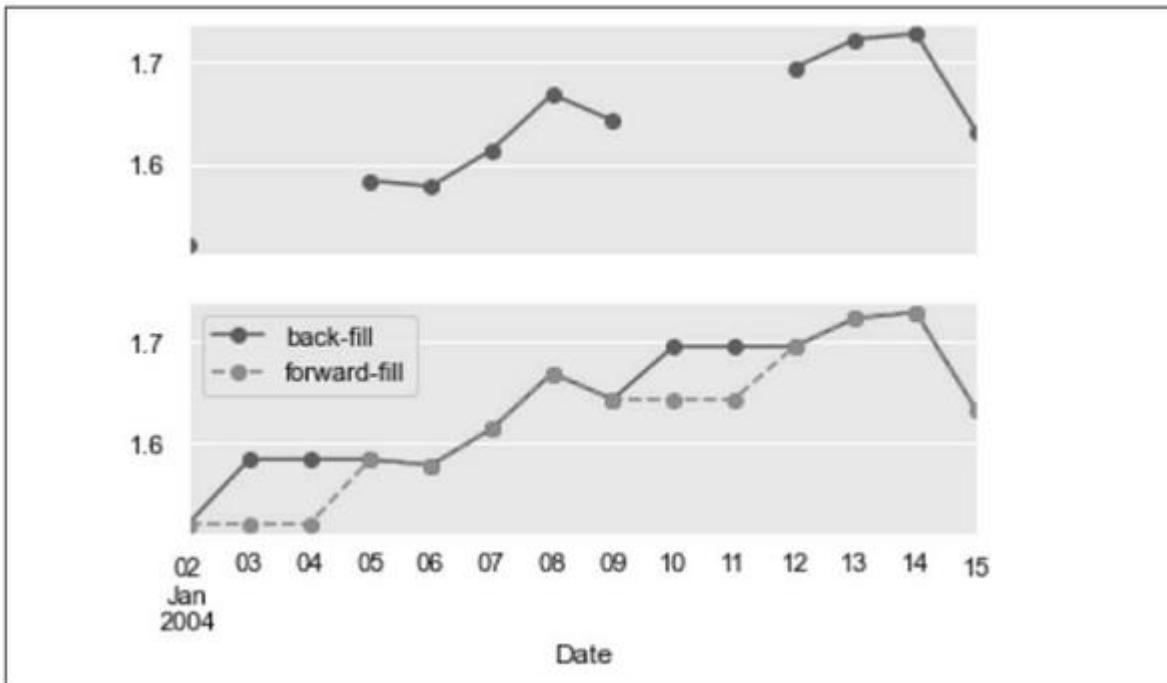
[Figure 3.6](#) : Rééchantillonage d'un cours de Bourse vers une fréquence moindre.

Notez bien la différence : pour chaque point, resample indique la moyenne de l'année précédente alors que asfreq rappelle la valeur en fin d'année.

Dans le cas d'une augmentation de fréquence (suréchantillonnage), les deux méthodes sont quasi équivalentes, bien que resample() offre plus d'options. Dans la configuration par défaut, les deux méthodes laissent les

nouveaux points vides, c'est-à-dire avec une valeur NA. Comme pour la fonction pd.fillna(), vous pouvez ajouter un paramètre méthode à asfreq() pour décider d'un autre traitement des nouvelles valeurs. Voici par exemple comment rééchantillonner des données limitées aux jours ouvrés vers les jours calendaires c'est-à-dire avec les week-ends ([Figure 3.7](#)) :

```
In[30]:  
fig, ax = plt.subplots(2, sharex=True)  
data = aapl.iloc[:10]  
  
data.asfreq('D').plot(ax=ax[0], marker='o')  
  
data.asfreq('D', method='bfill').plot(ax=ax[1],  
style=' -o')  
data.asfreq('D', method='ffill').plot(ax=ax[1],  
style='--o')  
ax[1].legend(["back-fill", "forward-fill"]);
```



[Figure 3.7](#) : Comparaison d'interpolations avec remplissage par l'aval et par l'amont.

Le diagramme du haut montre la situation initiale : les jours non ouvrés correspondent à des valeurs NA, qui ne sont donc pas représentées. Dans le panneau inférieur, nous voyons la différence entre les deux techniques de remplissage des trous : par l'aval ou par l'amont.

Décalage temporel

Un autre domaine de traitement temporel habituel consiste à décaler les données dans le temps. Pandas offre deux méthodes dans ce domaine : `shift()` et `tshift()`. La principale différence entre elles est que `shift()` décale les données alors

que tshift() décale les index. Dans les deux cas, le décalage est spécifié sous forme d'un multiple de la fréquence.

Voici par exemple comment utiliser shift() et tshift() pour un décalage de 900 jours ([Figure 3.8](#)) :

In[31]:

```
fig, ax = plt.subplots(3, sharey=True)

# apply a frequency to the data
aapl = aapl.asfreq('D', method='pad')

aapl.plot(ax=ax[0])
aapl.shift(900).plot(ax=ax[1])
aapl.tshift(900).plot(ax=ax[2])
# legends and annotations
local_max = pd.to_datetime('2007-11-05')
offset = pd.Timedelta(900, 'D')
ax[0].legend(['input'], loc=2)
ax[0].get_xticklabels()[4].set(weight='heavy',
color='red')
ax[0].axvline(local_max, alpha=0.3, color='red')

ax[1].legend(['shift(900)'], loc=2)
ax[1].get_xticklabels()[4].set(weight='heavy',
color='red')
ax[1].axvline(local_max + offset, alpha=0.3,
color='red')

ax[2].legend(['tshift(900)'], loc=2)
```

```

ax[2].get_xticklabels()[1].set(weight='heavy',
color='red')
ax[2].axvline(local_max + offset, alpha=0.3,
color='red');

```

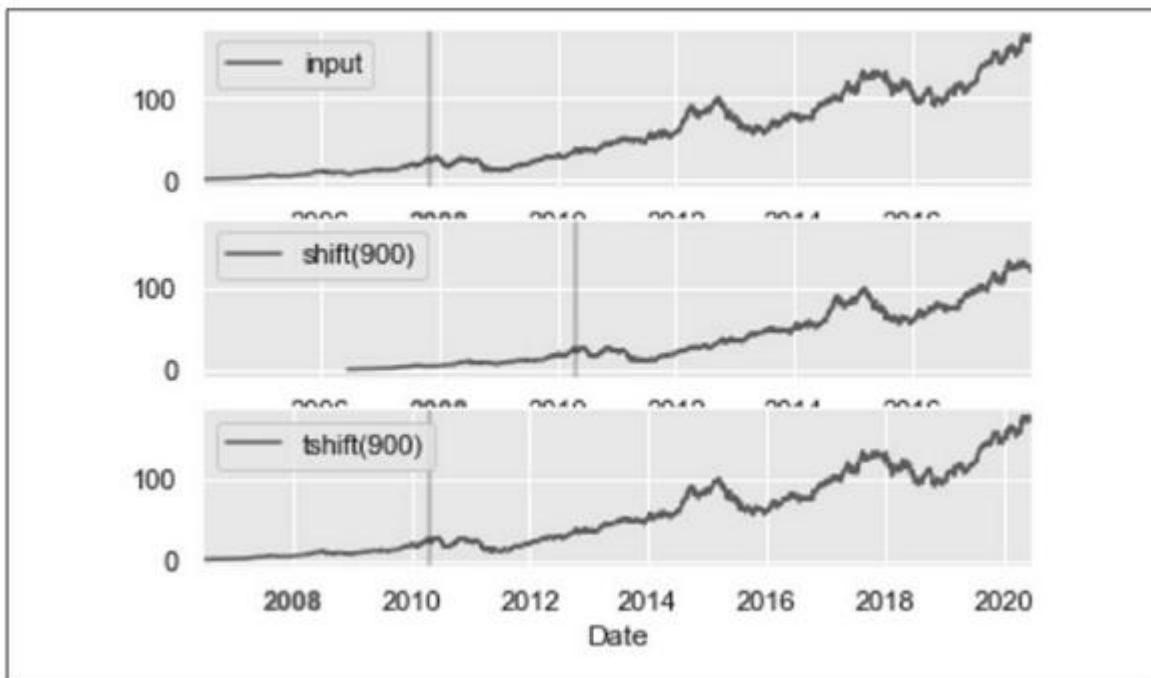


Figure 3.8 : Comparaison entre les décalages shift et tshift.

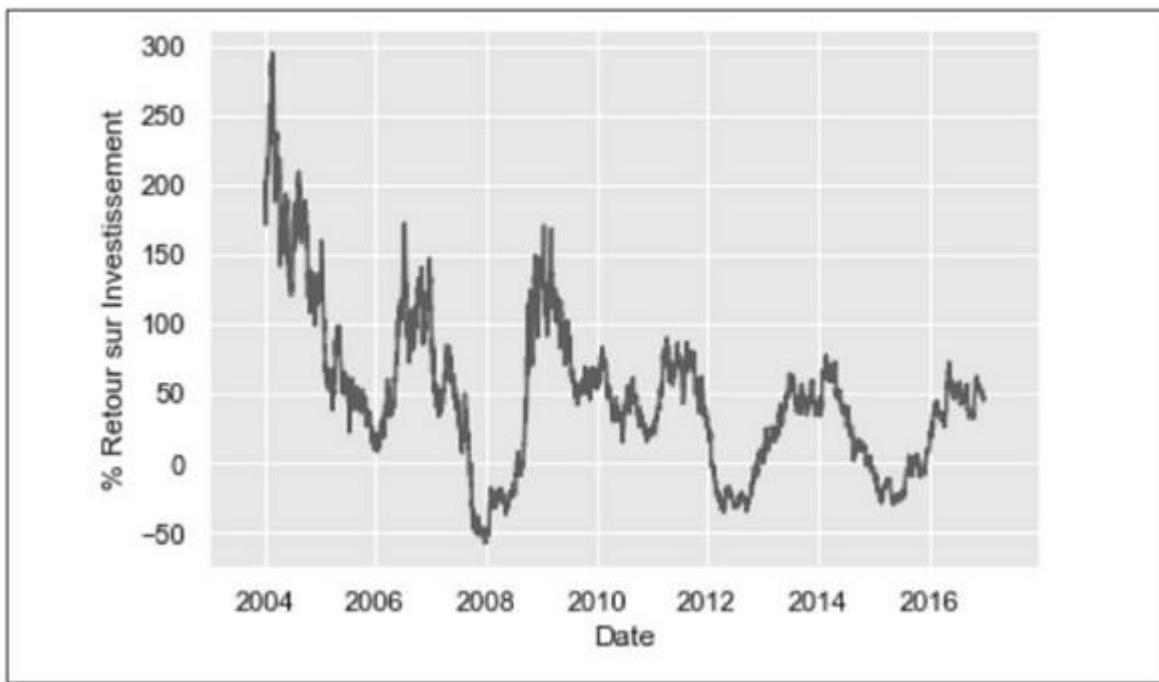
Nous voyons que `shift(900)` a décalé les données de 900 jours ; les plus récentes sont sorties du graphique par la droite et des valeurs NA ont été insérées par la gauche. L'autre méthode, `tshift(900)` a décalé les valeurs d'index de 900 jours.

Ce genre de décalage sert notamment à calculer des différences au cours du temps. Voici comment connaître le retour sur investissement sur un an pour l'action concernée

et sur la durée disponible dans le jeu de données ([Figure 3.9](#)) :

In[32]:

```
ROI = 100 * (aapl.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Retour sur Investissement');
```



[Figure 3.9](#) : Calcul du retour sur investissement annuel pour une action en Bourse.

Fenêtre mobile

Le troisième genre d'opérations temporelles de Pandas correspond aux statistiques mobiles qui dépendent de l'attribut nommé `rolling()` des objets `Series` et `DataFrame`. Vous obtenez ainsi une vue qui ressemble à celle fournie par

les opérations Groupby vues dans la section sur les agrégats et les groupements. La vue mobile donne accès à un certain nombre d'opérations d'agrégation par défaut.

Voici par exemple comment obtenir la moyenne et l'écart type mobile sur un an de notre action en Bourse ([Figure 3.10](#)) :

```
In[33]:
```

```
rolling = aapl.rolling(365, center=True)
data = pd.DataFrame({'input': aapl,
                     'one-year rolling_mean':
                     rolling.mean(),
                     'one-year rolling_std':
                     rolling.std()})
ax = data.plot(style=['-', '--', ':'])
ax.lines[0].set_alpha(.3)
```

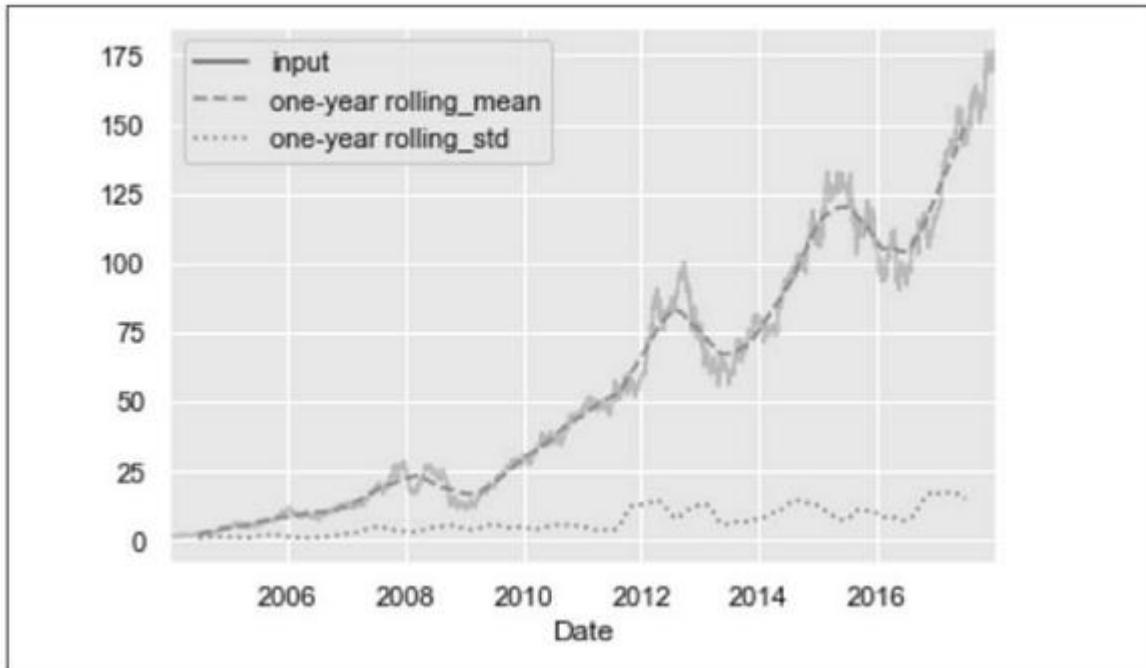


Figure 3.10 : Statistiques mobile d'une action en Bourse.

Comme avec GroupBy, vous pouvez vous servir de aggregate() et de apply() pour personnaliser vos calculs de valeurs mobiles.

Pour en savoir plus

Cette courte visite guidée n'a présenté que les fonctions essentielles concernant les données temporelles de Pandas. Pour en savoir plus, voyez bien sûr la documentation en ligne de Pandas, dans la rubrique « Time Series/Date » (<http://pandas.pydata.org/pandas-docs/stable/timeseries.html>).

Voyez également dans la même collection et chez le même éditeur le livre de Wes McKinney, *Analyse de données avec*

Python. Il présente les outils temporels dans le cadre économique et financier, et donne des détails au niveau de la gestion des fuseaux horaires et des jours ouvrés.

N'hésitez pas enfin à profiter de l'aide intégrée à IPython pour tester les différentes options des méthodes. Personnellement, je trouve que c'est souvent la meilleure façon d'appréhender un nouvel outil Python.

Exemple : comptage des bicyclettes à Seattle

Pour offrir un exemple un peu plus ambitieux de traitement de données temporelles, partons du comptage des bicyclettes empruntant le Fremont Bridge de Seattle (<http://www.openstreetmap.org/#map=17/47.64813/-122.34965>).

Les données proviennent d'un système de comptage automatique qui avait été placé dans les pistes cyclables du pont en 2012. Le comptage peut être téléchargé à l'adresse <http://data.seattle.gov/>. Voici le lien direct vers le jeu de données :

<https://data.seattle.gov/Transportation/Fremont-Bridge-Hourly-Bicycle-Counts-by-Month-Octo/65db-xm6k>

Les données sont fournies dans l'archive des exemples qui accompagne le livre. Voici les instructions pour récupérer le fichier CSV si nécessaire :

In[34] :

```
# !curl -o FremontBridge.csv
# https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```



(N.d.T.) Voyez s'il n'est pas nécessaire de renommer le fichier une fois téléchargé. Le nom utilisé dans la suite est *FremontBridge.csv*.

Nous nous servons de Pandas pour demander le chargement du contenu du fichier CSV dans un objet DataFrame. Nous demandons d'utiliser le champ *Date* comme index et nous voulons faire analyser automatiquement les dates :

In[35] :

```
data = pd.read_csv('FremontBridge.csv',
index_col='Date', parse_dates=True)
data.head()
```

Out[35] :

	Fremont Bridge Total	Fremont Bridge East Sidewalk	Fremont Bridge West Sidewalk
Date			
2019-11-01 00:00:00	12.0	7.0	5.0
2019-11-01 01:00:00	7.0	0.0	7.0

2019-11-01 02:00:00	1.0	0.0	1.0
2019-11-01 03:00:00	6.0	6.0	0.0
2019-11-01 04:00:00	6.0	5.0	1.0

Pour plus de confort, nous abrégeons les noms des colonnes :

In[36] :

```
data.columns = ['Total', 'Est', 'Ouest']
```

Prenons connaissance des statistiques techniques de ce jeu de données :

In[37] :

```
data.dropna().describe()
```

Out[37] :

	Total	Est	Ouest
count	144352.000000	144352.000000	144352.000000
mean	111.188013	50.529241	60.658772
std	141.260941	65.050403	87.812023
min	0.000000	0.000000	0.000000
25%	14.000000	6.000000	7.000000

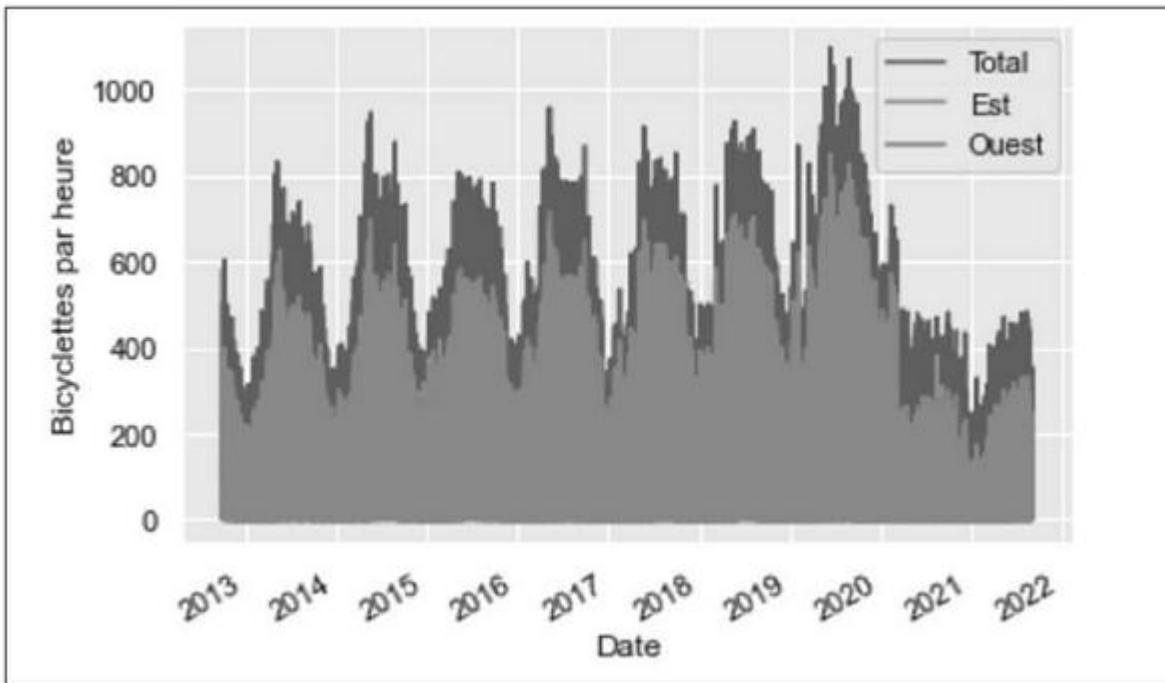
```
50%   60.000000    28.000000    30.000000  
75%   146.000000    68.000000    75.000000  
max   1097.000000   698.000000   850.000000
```

Visualisation des données

N'attendons pas pour avoir une première idée du contenu : demandons un tracé des données brutes ([Figure 3.11](#)) :

```
In[38]:  
%matplotlib inline  
import seaborn; seaborn.set()
```

```
In[39]:  
data.plot()  
plt.ylabel('Bicyclettes par heure');
```

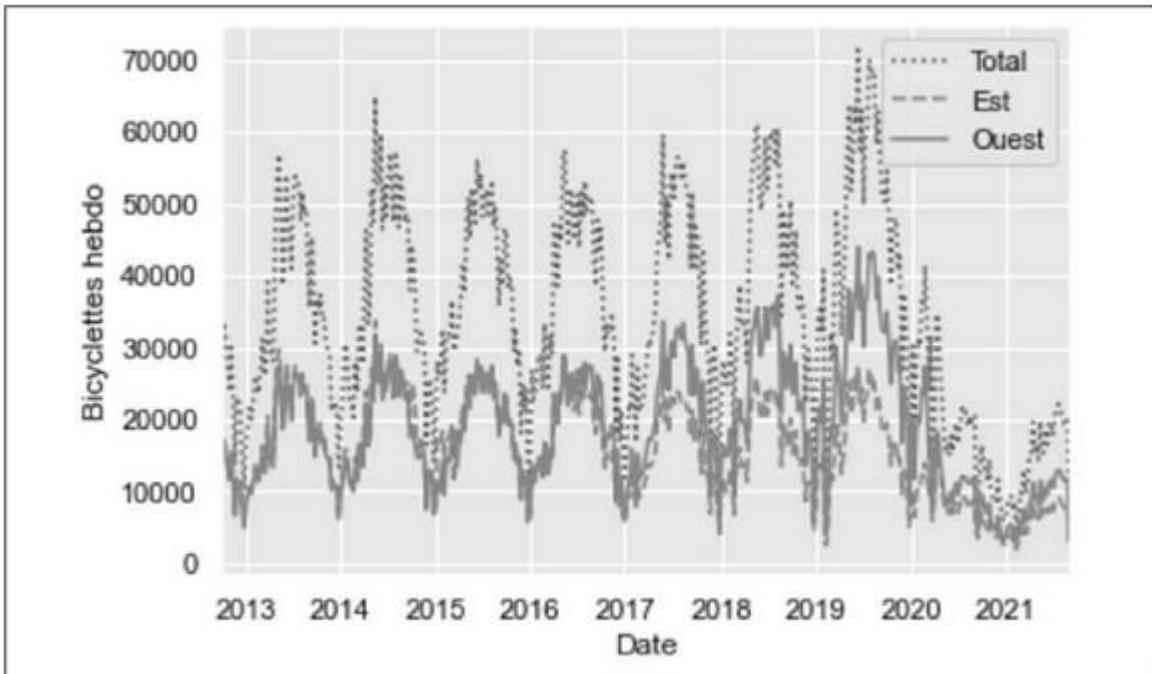


[Figure 3.11](#) : Comptage horaire des bicyclettes empruntant le Fremont Bridge.

La fréquence horaire sur une période de plusieurs années nous offre une précision qui dépasse nos besoins. Demandons donc un rééchantillonage par semaine ([Figure 3.12](#)) :

In[40]:

```
hebdo = data.resample('W').sum()  
hebdo.plot(style=[':', '--', '-'])  
plt.ylabel('Bicyclettes hebdo');
```



[Figure 3.12](#) : Sous-échantillonnage hebdomadaire des comptages de bicyclettes.

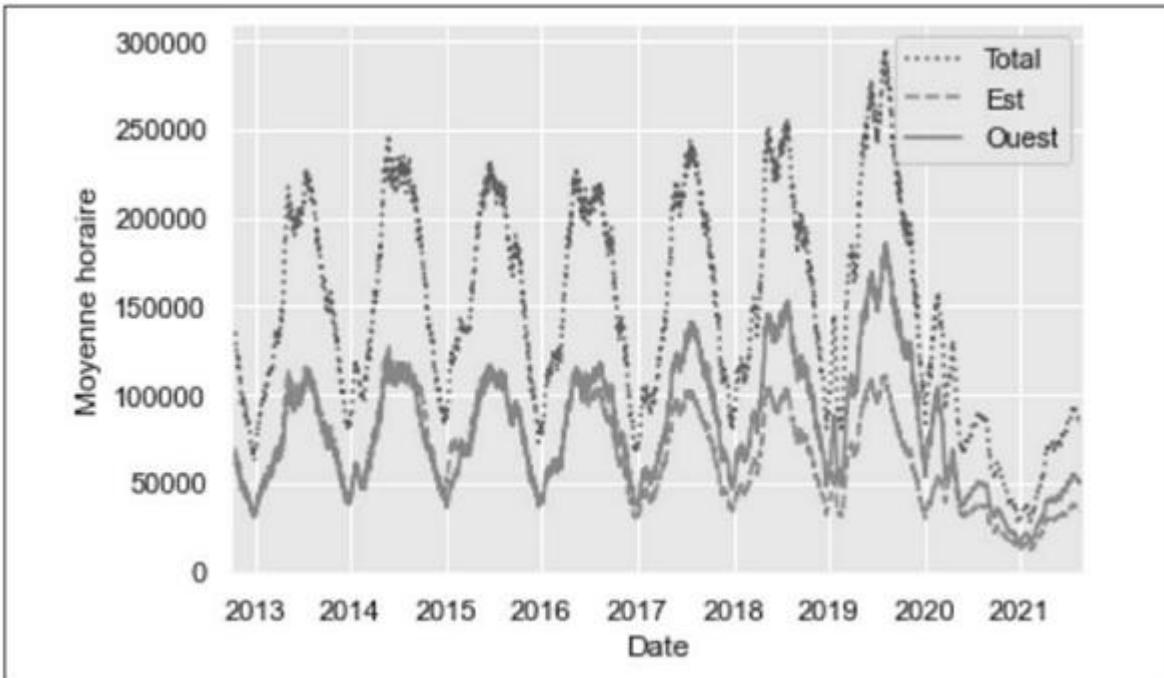
Une saisonnalité devient évidente : les gens sont plus nombreux à faire du vélo l’été que l’hiver, et l’utilisation varie d’une semaine à l’autre, sans doute en fonction de la météo. (Voyez également la description détaillant les régressions linéaires du [Chapitre 5](#) à ce sujet).

Voyons comment agréger les données avec une moyenne mobile, grâce à la fonction `pd.rolling_mean()`. Nous demandons une moyenne mobile sur 30 jours en prenant soin de bien centrer la fenêtre d’étude ([Figure 3.13](#)) :

In[41]:

```
quoti = data.resample('D').sum()
quoti.rolling(30, center=True).sum().plot(style=
```

```
[':', '--', '-'])  
plt.ylabel('Moyenne horaire');
```



[Figure 3.13](#) : Moyenne mobile des passages hebdomadaires.

Les courbes résultantes sont très hachées à cause de la valeur de rupture fixe (cutoff) de la fenêtre. Nous pouvons nous servir d'une fonction de fenêtrage pour une courbe moins mouvementée, par exemple une fenêtre gaussienne. L'exemple suivant et la [Figure 3.14](#) spécifient la largeur de fenêtre, 50 jours dans l'exemple, et la largeur de la gaussienne dans la fenêtre (nous avons choisi 10 jours) :

```
In[42]:  
quoti.rolling(50, center=True,
```

```
win_type='gaussian').sum(std=10).plot(style=[':', '--', '-']);
```

Plongée dans les données

La [Figure 3.14](#) permet de se faire une première idée de la tendance de passage, mais elle ne révèle pas la structure qui nous intéresse. Nous aimerais par exemple connaître le trafic moyen en fonction de l'heure dans le jour. Nous pouvons nous servir de GroupBy à cet effet (décrite plus haut dans ce chapitre) :

In[43]:

```
par_heure = data.groupby(data.index.time).mean()
tops_heure = 4 * 60 * 60 * np.arange(6)
par_heure.plot(xticks=tops_heure, style=[':', '-',
                                         '-']);
```

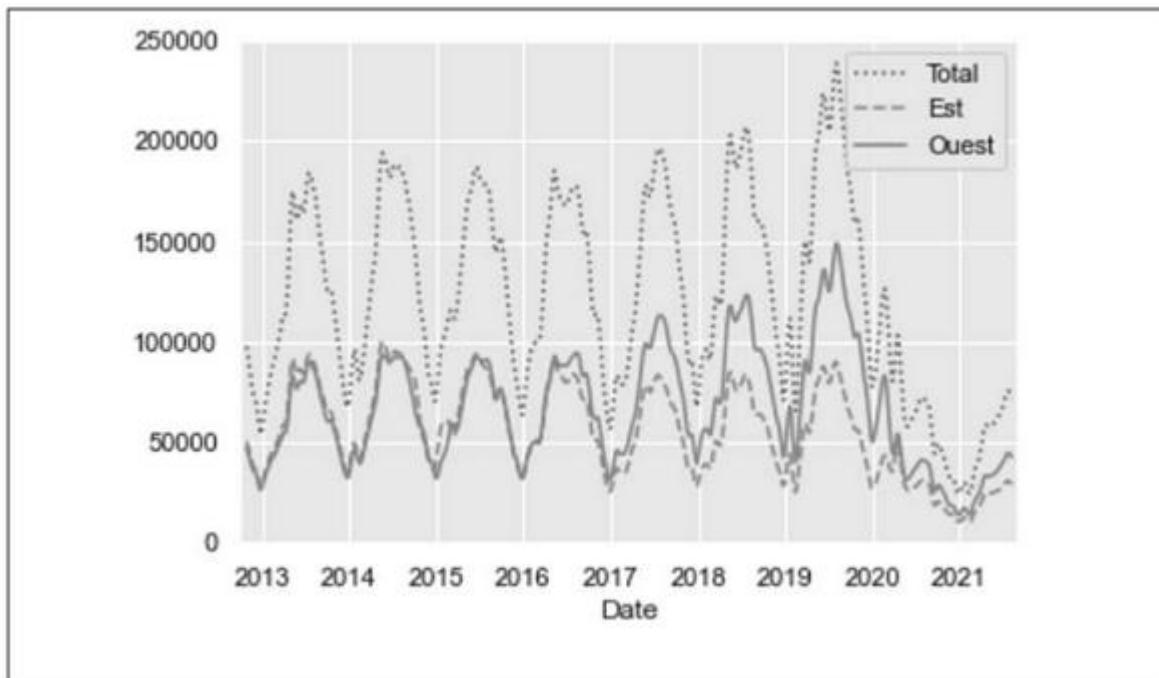


Figure 3.14 : Comptage hebdomadaire après adoucissement gaussien.

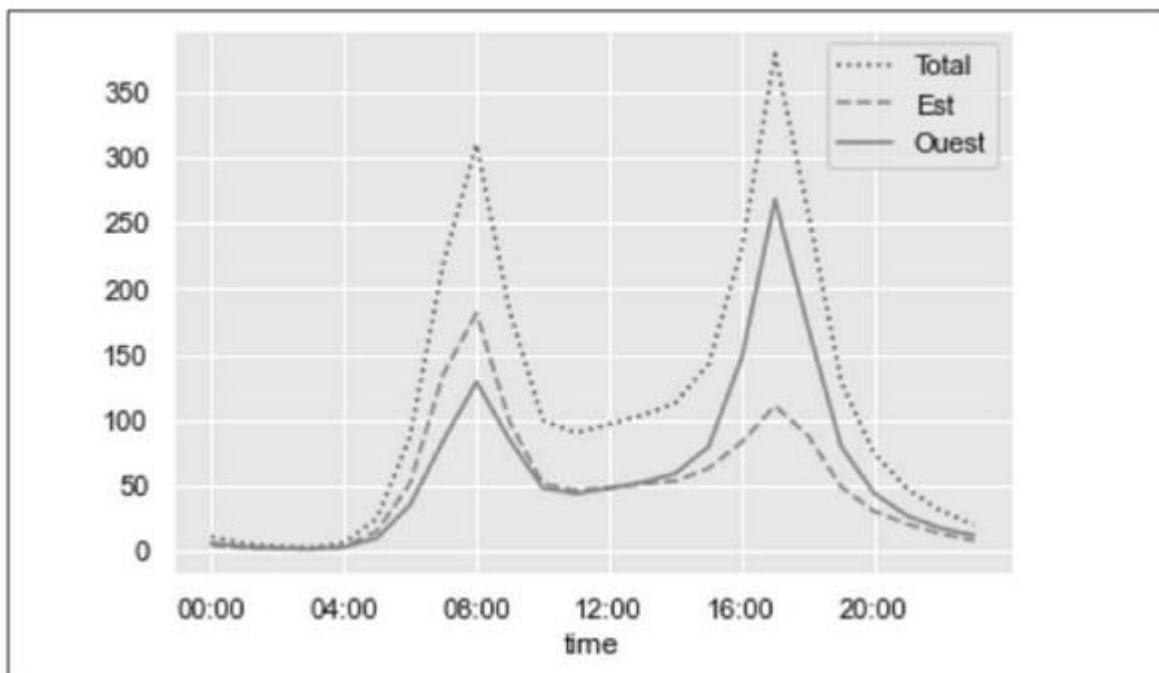


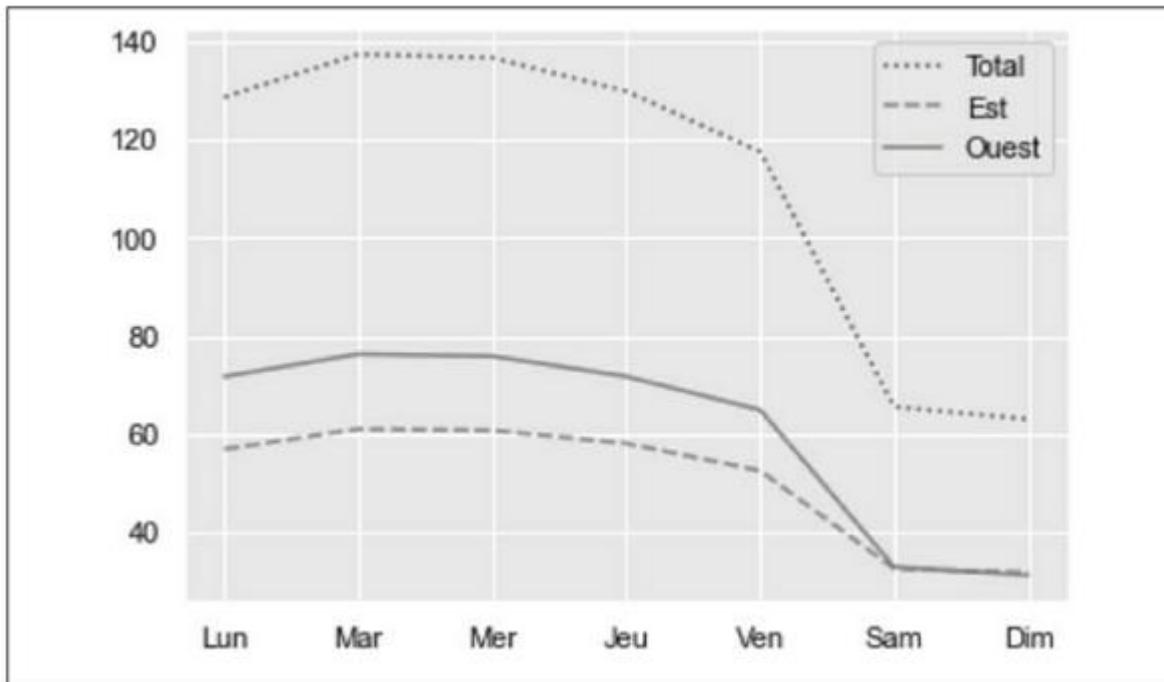
Figure 3.15 : Comptage moyen horaire des passages.

Le trafic horaire est clairement bimodal, avec des pointes à 8 heures du matin et à 17 heures, clairement liées aux horaires de bureau. D'ailleurs, la piste ouest qui sert à aller au centre-ville de Seattle voit sa pointe le matin alors que la piste est est plus fréquentée soir.

Voyons également les différences selon le jour de la semaine. Nous nous servons d'une simple opération groupby ([Figure 3.16](#)) :

In[44]:

```
par_jour =  
data.groupby(data.index.dayofweek).mean()  
par_jour.index = ['Lun', 'Mar', 'Mer', 'Jeu',  
'Ven', 'Sam', 'Dim']  
par_jour.plot(style=[':', '--', '-']);
```



[Figure 3.16](#) : Comptage moyen selon le jour de semaine.

La différence entre jours ouvrés et week-ends est évidente, puisqu'il y a quasiment deux fois plus de passages du lundi au vendredi que le samedi et le dimanche.

Allons encore plus loin en demandant un groupement groupby composé pour obtenir la tendance horaire la semaine et le week-end. Nous définissons un indicateur pour marquer le week-end et nous demandons un groupement selon cet indicateur et l'heure dans la journée :

In[45]:

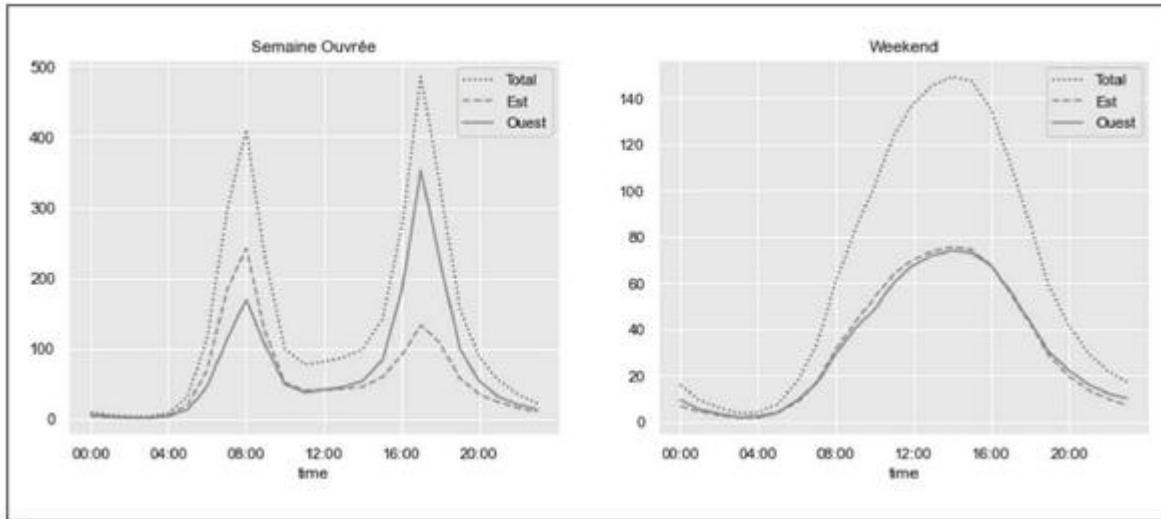
```
weekend = np.where(data.index.weekday < 5,  
'Weekday', 'Weekend')
```

```
par_heure = data.groupby([weekend,  
data.index.time]).mean()
```

Nous nous servons ensuite des outils Matplotlib qui sont décrits dans la section sur les sous-graphiques multiples du [Chapitre 4](#) pour présenter les deux diagrammes côté à côté ([Figure 3.17](#)) :

In[46]:

```
import matplotlib.pyplot as plt  
fig, ax = plt.subplots(1, 2, figsize=(14, 5))  
par_heure.loc['Weekday'].plot(ax=ax[0],  
title='Semaine Ouvrée',  
                    xticks=tops_heure,  
style=[':', '--', '-'])  
par_heure.loc['Weekend'].plot(ax=ax[1],  
title='Weekend',  
                    xticks=tops_heure,  
style=[':', '--', '-']);
```



[Figure 3.17](#) : Comparaison des passages par heure la semaine et le week-end.

Le résultat n'est pas surprenant : pendant les jours ouvrés, le motif est bimodal alors que le week-end, il est unimodal. Il serait intéressant de creuser encore plus cette situation en cherchant l'impact de la météo, de la température, et d'autres facteurs jouant sur les comportements des cyclistes. (J'ai écrit un article à ce sujet sur mon blog, « *Is Seattle Really Seeing an Uptick In Cycling?* »). Nous allons réutiliser ce jeu de données dans le [Chapitre 5](#) lorsque nous verrons en détail les régressions linéaires.

3.13 : Hautes performances Pandas avec eval() et query()

Nous avons vu au cours des Chapitres 2 et 3 que la technologie PyData profitait des possibilités offertes par les deux librairies NumPy et Pandas pour réaliser certaines opérations directement en langage C, en utilisant une syntaxe aisée. C'est notamment le cas des opérations à vectorisation et à diffusion de NumPy et des opérations de groupement de Pandas. Pouvoir combiner bonnes performances et haut niveau d'abstraction convient parfaitement dans de nombreux cas, mais il faut rappeler que cela suppose normalement de créer des objets temporaires intermédiaires, ce qui a malgré tout un impact négatif sur l'empreinte mémoire et les durées de traitement.

Depuis sa version 0.13, la librairie Pandas dispose d'outils expérimentaux qui permettent de profiter directement des performances de niveau C sans avoir besoin de créer des tableaux intermédiaires. Ce sont les deux fonctions nommées eval() et query() qui se fondent sur le paquetage numexpr (<https://github.com/pydata/numexpr>). Voyons leurs principales conditions d'utilisation et les contextes dans lesquels vous pourrez y avoir recours.

query() et eval() pour les expressions composites

Nous savons qu'aussi bien NumPy que Pandas permettent de réaliser des opérations vectorisées performantes. C'est par exemple le cas pour additionner les éléments de deux tableaux :

```
In[1]:  
import numpy as np  
rng = np.random.RandomState(42)  
x = rng.rand(1E6)  
y = rng.rand(1E6)  
%timeit x + y
```

```
6.83 ms ± 425 µs per loop (mean ± std. dev. of 7  
runs, 100 loops each)
```

Comme vu lors de la présentation des ufuncs dans le [Chapitre 2](#), cette approche est bien plus rapide que celle qui se base sur une boucle de répétition Python ou une liste par compréhension :

```
In[2]:  
%timeit np.fromiter((xi + yi for xi, yi in  
zip(x, y)),  
dtype=x.dtype, count=len(x))
```

```
309 ms ± 3.69 ms per loop (mean ± std. dev. of 7
runs, 1 loop each)
```

Ce niveau d'abstraction devient néanmoins moins efficace lorsqu'il s'agit de traiter des expressions composées, comme dans l'exemple suivant :

```
In[3]: mask = (x > 0.5) & (y < 0.5)
```

Ici, NumPy doit évaluer chaque sous-expression, et le traitement équivaut à peu près à ceci :

```
In[4]: tmp1 = (x > 0.5) tmp2 = (y < 0.5) mask =
tmp1 & tmp2
```

Chaque étape intermédiaire fait l'objet d'une réservation d'espace mémoire. La surcharge en matière d'espace et de temps de traitement peut devenir énorme si les deux tableaux x et y sont volumineux. Grâce à la librairie Numexpr, vous pouvez dorénavant réaliser ce type de traitement d'expressions composées élément par élément sans qu'il y ait réservation de tableaux intermédiaires. Vous trouverez d'autres détails dans la documentation de Numexpr. Il nous suffit de savoir que la librairie accepte en entrée une chaîne contenant l'expression à traiter en style NumPy :

```
In[5]:
import numexpr
```

```
mask_numexpr = numexpr.evaluate(' (x > 0.5) & (y  
< 0.5) ')  
np.allclose(mask, mask_numexpr)
```

Out[5]: True

Les deux outils de Pandas nommés eval() et query() fonctionnent sur le même principe puisqu'ils sont basés sur le paquetage Numexpr.

Opérations efficaces avec pandas.eval()

La fonction eval() de Pandas est alimentée par une expression de type chaîne pour réaliser des calculs avec des objets DataFrame. Commençons par créer quatre objets DataFrame :

In[6]:

```
import pandas as pd  
nrows, ncols = 100000, 100  
rng = np.random.RandomState(42)  
df1, df2, df3, df4 =  
(pd.DataFrame(rng.rand(nrows, ncols))  
  
for i in range(4))
```

Nous pouvons obtenir la somme des quatre objets en Pandas classique en décrivant l'opération :

In[7]:

```
%timeit df1 + df2 + df3 + df4
```

```
191 ms ± 3.24 ms per loop (mean ± std. dev. of 7
runs, 1 loop each)
```

Pour obtenir le même résultat avec pd.eval, nous construisons une chaîne pour l'expression :

In[8]:

```
%timeit pd.eval('df1 + df2 + df3 + df4')
```

```
89.7 ms ± 3.74 ms per loop (mean ± std. dev. of
7 runs, 10 loops each)
```

La version basée sur eval() est environ 50 % plus rapide et consomme bien moins d'espace mémoire, tout en donnant le même résultat :



(N.d.T.) La fonction allclose() permet de comparer le contenu des deux objets.

In[9]:

```
np.allclose(df1 + df2 + df3 + df4,
pd.eval('df1 + df2 + df3 + df4'))
```

Out[9]: True

Quelques opérations proposées par pd.eval()

La fonction pd.eval() offre une belle polyvalence. Pour nos exemples, nous allons partir des objets DataFrame d'entiers suivants :

```
In[10]:  
df1, df2, df3, df4, df5 =  
(pd.DataFrame(rng.randint(0, 1000, (100, 3)))  
  
for i in range(5))
```

Opérateurs arithmétiques. pd.eval() reconnaît tous les opérateurs arithmétiques, par exemple :

```
In[11]:  
result1 = -df1 * df2 / (df3 + df4) - df5  
result2 = pd.eval('-df1 * df2 / (df3 + df4) -  
df5')  
np.allclose(result1, result2)
```

```
Out[11]: True
```

Opérateurs de comparaison. pd.eval() accepte tous les opérateurs relationnels, y compris les expressions chaînées :

```
In[12]:  
result1 = (df1 < df2) & (df2 <= df3) & (df3 !=  
df4)  
result2 = pd.eval('df1 < df2 <= df3 != df4')
```

```
np.allclose(result1, result2)
```

```
Out[12]: True
```

Opérateurs sur bits. pd.eval() reconnaît les trois opérateurs &, | et ~ :

```
In[13]:
```

```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 <  
df4)  
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) |  
(df3 < df4)')  
np.allclose(result1, result2)
```

```
Out[13]: True
```

La notation littérale des mêmes opérateurs dans les expressions booléennes est possible (and, or et not) :

```
In[14]:
```

```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5)  
or (df3 < df4)')  
np.allclose(result1, result3)
```

```
Out[14]: True
```

Attribut d'objets et index. pd.eval() permet d'accéder aux attributs des objets au moyen de la syntaxe obj.attr et aux index en écrivant obj[index] :

In[15]:

```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)
```

Out[15]: True

Autres opérations. D'autres opérations, et notamment les appels de fonction, les instructions conditionnelles et les boucles sont ou seront prochainement ajoutés à la définition de pd.eval(). Si vous avez besoin d'une possibilité non encore disponible par Pandas, vous vous servirez directement de la librairie Numexpr.

DataFrame.eval() pour les opérations par colonnes

L'objet DataFrame possède sa propre méthode eval() qui fonctionne de façon similaire à pd.eval(). Elle offre l'avantage de permettre de faire référence aux colonnes par *leur nom*. Mettons d'abord en place un tableau avec des labels :

In[16]:

```
df = pd.DataFrame(rng.rand(1000, 3), columns=
['A', 'B', 'C'])
df.head()
```

Out[16]:

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

Nous pouvons maintenant désigner les colonnes pour effectuer un calcul avec pd.eval() :

In[17]:

```
result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)
```

Out[17]: True

Avec la méthode DataFrame.eval(), l'écriture peut être plus compacte, car nous citons directement les noms des colonnes :

In[18]:

```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
```

Out[18]: True

Vous constatez que nous utilisons les noms des colonnes comme noms de variables dans l'expression.

Affectations dans DataFrame.eval()

DataFrame.eval() permet également de modifier les valeurs par affectation à une colonne. Reprenons l'objet DataFrame précédent avec ses trois colonnes 'A', 'B' et 'C' :

In[19] :

```
df.head()
```

Out[19] :

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

Nous pouvons nous servir de df.eval() pour ajouter une colonne 'D' et la peupler avec une valeur calculée à partir des trois autres :

In[20] :

```
df.eval('D = (A + B) / C', inplace=True)  
df.head()
```

Out[20] :

	A	B	C	D
0	0.375506	0.406939	0.069938	11.187620

```
1 0.069087 0.235615 0.154374 1.973796
2 0.677945 0.433839 0.652324 1.704344
3 0.264038 0.808055 0.347197 3.087857
4 0.589161 0.252418 0.557789 1.508776
```

Il est bien sûr également possible de modifier le contenu d'une colonne existante :

In[21]:

```
df.eval('D = (A - B) / C', inplace=True)
df.head()
```

Out[21]:

	A	B	C	D
0	0.375506	0.406939	0.069938	-0.449425
1	0.069087	0.235615	0.154374	-1.078728
2	0.677945	0.433839	0.652324	0.374209
3	0.264038	0.808055	0.347197	-1.566886
4	0.589161	0.252418	0.557789	0.603708

Les variables locales dans DataFrame.eval()

La méthode DataFrame.eval() autorise même l'utilisation de variables Python locales :

In[22]:

```
moy_colonne = df.mean(1)
result1 = df['A'] + moy_colonne
result2 = df.eval('A + @moy_colonne')
np.allclose(result1, result2)
```

```
Out[22]: True
```

Le nom de variable locale doit être précédé du symbole @, ce qui permet d'évaluer des expressions qui exploitent des valeurs dans les deux espaces de noms : celui des colonnes et celui des objets Python. Précisons que le métacaractère @ n'est reconnu que par la *méthode* DataFrame.eval(), pas par la *fonction* pandas.eval(), car cette dernière n'a accès qu'à un espace de noms Python.

La méthode DataFrame.query()

Les objets DataFrame disposent d'une seconde méthode fonctionnant par évaluation d'une chaîne. Il s'agit de la méthode query(). Partons du traitement suivant :

```
In[23]:
```

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = pd.eval('df[(df.A < 0.5) & (df.B <
0.5)]')
np.allclose(result1, result2)
```

```
Out[23]: True
```

Nous utilisons effectivement des noms de colonnes de l'objet DataFrame, comme nous l'avons vu pour DataFrame.eval(). En revanche, il n'est pas possible

d'utiliser la syntaxe de DataFrame. eval(). Pour réussir ce genre de filtrage, il faut recourir à la méthode query() :

In[24]:

```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
```

Out[24]: True

Non seulement le traitement est plus efficace, mais la lecture de l'instruction est plus simple que celle avec l'expression de masquage. Ajoutons que la méthode query() accepte elle aussi de citer des variables locales avec @ :

In[25]:

```
Cmean = df['C'].mean()
result1 = df[(df.A < Cmean) & (df.B < Cmean)]
result2 = df.query('A < @Cmean and B < @Cmean')
np.allclose(result1, result2)
```

Out[25]: True

Domaine d'utilisation de ces fonctions

Deux critères sont à considérer lorsque vous vous apprêtez à utiliser les deux fonctions mentionnées : le temps de calcul et l'empreinte mémoire. Les besoins en mémoire sont les

plus faciles à estimer. Nous savons que toutes expressions composées qui concernent un tableau NumPy ou un objet DataFrame de Pandas va entraîner la création de tableaux temporaires. L'instruction suivante :

```
In[26]: x = df[(df.A < 0.5) & (df.B < 0.5)]
```

est à peu près équivalente à celles-ci :

```
In[27]:  
tmp1 = df.A < 0.5  
tmp2 = df.B < 0.5  
tmp3 = tmp1 & tmp2  
x = df[tmp3]
```

Dès que la taille prévue pour les objets DataFrame intermédiaires menace d'atteindre la quantité d'espace mémoire disponible, c'est-à-dire plusieurs gigaoctets, il est conseillé de recourir à une expression eval() ou query(). Rappelons que vous pouvez écrire ceci pour connaître la taille approximative de votre tableau en octets :

```
In[28]: df.values.nbytes
```

```
Out[28]: 32000
```

Du côté des performances, eval() peut se montrer plus rapide même si vous n'êtes pas en risque de débordement

mémoire système. Tout dépend de la relation entre la taille de vos objets DataFrame temporaires et la taille des zones de mémoire cache L2 et L1 du processeur, qui sont de quelques mégaoctets. Lorsque les objets intermédiaires sont beaucoup plus volumineux, eval() peut éviter les transferts lents des valeurs entre les différents niveaux de cache. À titre personnel, je n'ai pas vu d'énormes différences entre la méthode classique et l'utilisation d'eval() ou de query(). D'ailleurs, la méthode classique est même plus rapide pour les petits tableaux ! Le premier avantage de eval() et query() est l'économie d'espace mémoire, ainsi que la syntaxe plus lisible.

Nous venons de présenter un certain nombre de détails concernant eval() et query() ; vous irez consulter la documentation de Pandas pour tout complément. Vous y verrez que vous pouvez notamment mettre en place un analyseur ou un moteur pour exécuter les requêtes, comme décrit dans la section consacrée aux performances dans la documentation (<http://pandas.pydata.org/pandas-docs/dev/enhancingperf.html>).

3.14 : Autres ressources

Ce chapitre nous a permis de passer en revue les principaux outils offerts par la librairie Pandas pour une analyse efficace des données. Nous n'avons pas pu tout dire. Je vous invite à vous reporter aux ressources suivantes pour en savoir plus au sujet de Pandas !

Documentation en ligne de Pandas

Il s'agit bien sûr de la référence principale (<http://pandas.pydata.org/>). Les exemples que vous trouverez correspondent en général à de petits volumes de données, mais les options sont décrites en détail et permettent aisément de comprendre comment utiliser les différentes fonctions.

Analyse de données avec Python par Wes McKinney

Rédigé par Wes McKinney qui est le créateur de la librairie Pandas, cet autre livre va beaucoup plus en détail sur ce paquetage, en s'intéressant notamment de façon fouillée aux données temporelles dont Wes avait énormément besoin en tant que consultant financier. Le livre regorge d'exemples

montrant comment appliquer Pandas à des situations du monde réel. Il est disponible en version française chez First.

Stack Overflow

Le nombre d'adeptes de la librairie Pandas est tel que quasiment toute question que vous pourriez vous poser trouvera sa réponse sur le site Stack Overflow (<http://stackoverflow.com/questions/tagged/pandas>). Une approche efficace consiste à d'abord utiliser le moteur de recherches Google pour poser votre question, si possible en anglais, ou votre message d'erreur. Vous devrez normalement trouver la page du site Stack Overflow qui répondra à votre attente.

Pandas sur PyVideo

De nombreuses conférences ont été l'occasion d'exposer par des développeurs et des utilisateurs experts de Pandas, qu'ils s'agissent de PyCon, SciPy ou de PyData. Voyez notamment les tutoriels de PyCon qui ont été conçus par des présentateurs bien armés (<http://pyvideo.org/tag/pandas/>).

J'espère que ces ressources et les informations que vous avez pu réunir au long de ce chapitre vous aideront à bien exploiter Pandas pour résoudre les problèmes d'analyse de données que vous aurez à affronter !

CHAPITRE 4

Visualisation avec Matplotlib

Nous pouvons maintenant plonger dans la découverte de l'outil Python de visualisation de données : Matplotlib. Il s'agit d'une librairie multiplateforme fondée sur les tableaux de données NumPy. On l'exploite dans le cadre de l'ensemble fonctionnel SciPy. Matplotlib a été conçu en 2002 par John Hunter ; il ne désirait au départ qu'un enrichissement d'IPython pour pouvoir réaliser des tracés de graphiques interactifs dans le style MatLab en utilisant gnuplot depuis la ligne de commande IPython. À la même époque, le créateur d'IPython, Fernando Perez, mettait les bouchées doubles pour boucler sa thèse de doctorat. Il avait fait savoir à John qu'il n'aurait pas le temps de s'occuper de sa partie du projet pendant plusieurs mois. C'est ainsi que John avait décidé de reprendre l'ensemble à son compte, ce qui a produit le paquetage Matplotlib. La version 0.1 était sortie en 2003. La librairie a rapidement profité d'un énorme coup de projecteur lorsque le *Space Telescope Science Institute* (responsable de la mise en place du télescope spatial Hubble) l'a choisi pour générer ses graphiques. Matplotlib a

ainsi pu bénéficier d'un support financier et d'enrichissements fonctionnels soutenus.

Un des points forts de Matplotlib est sa capacité à interagir correctement avec de nombreux systèmes d'exploitation et sous-systèmes graphiques. La librairie accepte en effet des dizaines de types d'affichage et de moteurs de rendu, ce qui vous garantit de pouvoir l'exploiter, quel que soit le système d'exploitation ou le mode d'affichage dont vous avez besoin. Cette approche multiplateforme polyvalente reste l'un des grands avantages de Matplotlib. Elle a favorisé son adoption à vaste échelle, ce qui a permis de faire éclore une active communauté de développeurs qui a diffusé les puissants outils de Matplotlib et promu sa versatilité dans le monde scientifique utilisant le langage Python.

Au bout de quasiment deux décennies d'évolution, l'interface et le style de Matplotlib commencent néanmoins à être un peu surannés. Les modalités d'utilisation commencent à dater en comparaison de celles des outils plus récents tels que ggplot et ggviz pour le langage R, ou des outils de visualisation Web basés sur D3js ou HTML5. Je reste convaincu qu'il ne faut pas pour autant écarter Matplotlib qui continue à constituer un moteur de rendu graphique multiplateforme solide. Les plus récentes versions permettent d'ailleurs de définir facilement de nouveaux styles de tracés (une section de ce chapitre aborde cette

personnalisation de la configuration). De nouveaux paquetages sont apparus, s'appuyant sur les mécanismes puissants de Matplotlib en proposant une interface API plus moderne. C'est notamment le cas de Seaborn (une section lui est consacrée dans le chapitre), de ggplot (<http://yhat.github.io/ggplot>), de HoloViews (<http://holoviews.org>), de Altair (<http://altair-viz.github.io>) et même de la librairie Pandas qui peut être utilisée comme enveloppe autour de l'interface API de Matplotlib. Ceci dit, même lorsque l'on utilise une telle surcouche fonctionnelle, il reste indispensable de découvrir la syntaxe de Matplotlib pour pouvoir effectuer des réglages fins. Je pense que Matplotlib va rester un composant essentiel de la trousse à outils de visualisation de données, même si les outils plus récents vont amener les développeurs à moins recourir aux accès directs à l'API de Matplotlib.

4.1 : Techniques Matplotlib fondamentales

Avant de découvrir comment créer des graphiques avec Matplotlib, prenons le temps d'acquérir les techniques de bonne utilisation du paquetage.

Importation de Matplotlib

De même qu'une convention invite à utiliser l'abréviation np pour la librairie NumPy et pd pour Pandas, nous allons suivre deux conventions pour les imports Matplotlib :

```
In[1]:  
import matplotlib as mpl  
import matplotlib.pyplot as plt
```

L'interface nommée plt est celle que nous utiliserons le plus souvent dans ce chapitre.

Choix des styles

Le choix du style visuel de nos graphiques sera réalisé au moyen de la directive plt.style. Voici comment opter pour le style classique de Matplotlib :

```
In[2]: plt.style.use('classic')
```

Nous retoucherons ce style en fonction des besoins. Notez qu'avant la version 1.5 de Matplotlib, on ne disposait que du style par défaut. Vers la fin du chapitre, nous reviendrons sur la personnalisation de Matplotlib.

Afficher les graphiques : avec ou sans `show()` ?

Visualiser un graphique suppose deux étapes : le créer en mémoire puis le rendre visible sur un écran ou un terminal. La façon dont vous demandez l'affichage de votre rendu dépend du contexte. Vous avez le choix entre trois contextes d'utilisation : dans un script, dans un terminal IPython ou dans un calepin IPython Web.

Tracé depuis un script

Pour profiter de Matplotlib depuis un script Python, vous vous servez de la fonction `plt.show()` qui démarre une boucle événementielle, cherche tous les objets de figures actifs puis ouvre une ou plusieurs fenêtres interactives pour y positionner ces figures.

Partons d'un fichier portant le nom `monplot.py` contenant les lignes suivantes :

```
# file: monplot.py
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

Si vous démarrez ce script depuis la ligne de commande comme ci-dessous, vous verrez apparaître une fenêtre contenant la figure demandée :

```
$ python monplot.py
```

En coulisses, la fonction plt.show() réalise un certain nombre d'opérations, puisqu'elle doit entrer en interaction avec le sous-système d'affichage graphique. Les détails varient évidemment selon le système d'exploitation et même selon l'installation ; Matplotlib fait de son mieux pour vous épargner tout souci à ce niveau de détails.

Une mise en garde importante : vous ne devez appeler la fonction plt.show() qu'*une seule fois* par session Python. En règle générale, cet appel se trouve tout à la fin de votre script. Si vous appelez plusieurs fois show(), vous allez déclencher un comportement imprévisible et différent selon

l'infrastructure concernée. Vous prenez donc soin d'éviter cette erreur.

Tracé depuis une session d'interpréteur IPython

Vous pouvez profiter de Matplotlib de façon interactive depuis l'interpréteur IPython (vu dans le [Chapitre 1](#)). Il suffit de demander de basculer en mode Matplotlib au moyen de la commande magique %matplotlib que vous saisissez juste après avoir démarré IPython :

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg
```

```
In [2]: import matplotlib.pyplot as plt
```

Une fois dans ce mode, n'importe quelle commande de tracé plt va faire s'ouvrir une fenêtre graphique dans laquelle vous allez pouvoir visualiser le résultat d'autres commandes. Certaines modifications ne seront pas prises en compte automatiquement, et notamment la modification des propriétés de tracés existants. Il suffira dans ce cas de demander une mise à jour avec plt.draw(). Notez que vous n'avez pas besoin de la fonction plt.show() dans le mode Matplotlib.

Tracé depuis un calepin IPython

Vous connaissez bien sûr le principe des calepins IPython qui s'utilisent dans une fenêtre de navigateur Web et permettent de combiner du code Python, des graphiques, des éléments HTML et des commentaires (revoyez le [Chapitre 1](#) si nécessaire).

Pour accéder à un affichage graphique interactif dans un calepin IPython, vous utilisez la commande magique `%matplotlib`, comme dans l'interpréteur IPython. Dans le calepin IPython, vous pouvez également insérer directement les figures réalisées dans le calepin, et vous avez deux options à ce niveau :

- `%matplotlib notebook` insère un graphique interactif dans le calepin ;
- `%matplotlib inline` insère une figure statique du graphique dans le calepin.

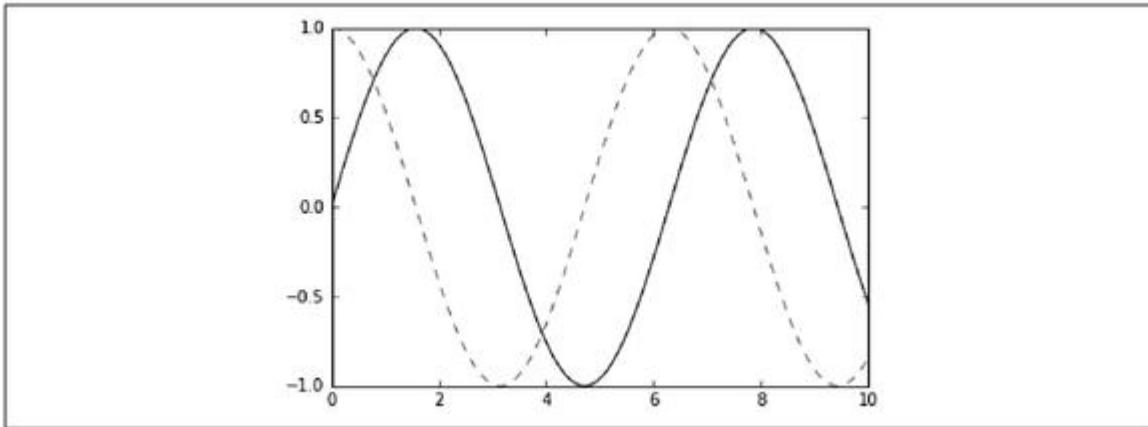
Dans la suite du livre, nous utiliserons en général la formule statique avec `%matplotlib inline` :

```
In[3]: %matplotlib inline
```

Vous n'exécutez cette commande qu'une fois par session ou noyau d'exécution. Toutes les cellules qui comportent une commande de création de graphique vont être suivies d'une

image au format PNG du graphique correspondant ([Figure 4.1](#)) :

```
In[41]:  
import numpy as np  
x = np.linspace(0, 10, 100)  
  
fig = plt.figure()  
plt.plot(x, np.sin(x), '-')  
plt.plot(x, np.cos(x), '--');
```



[Figure 4.1](#) : Exemple simple de tracé graphique.

Enregistrer une figure dans un fichier

La librairie Matplotlib permet d'enregistrer un graphique dans un des nombreux formats de fichiers disponibles au moyen de la fonction `savefig()`. Voici par exemple comment produire un fichier PNG à partir de la figure courante :

```
In[5]: fig.savefig('ma_figure.png')
```

Nous disposons ensuite d'un fichier portant le nom *ma_figure.png* stocké dans le répertoire courant :

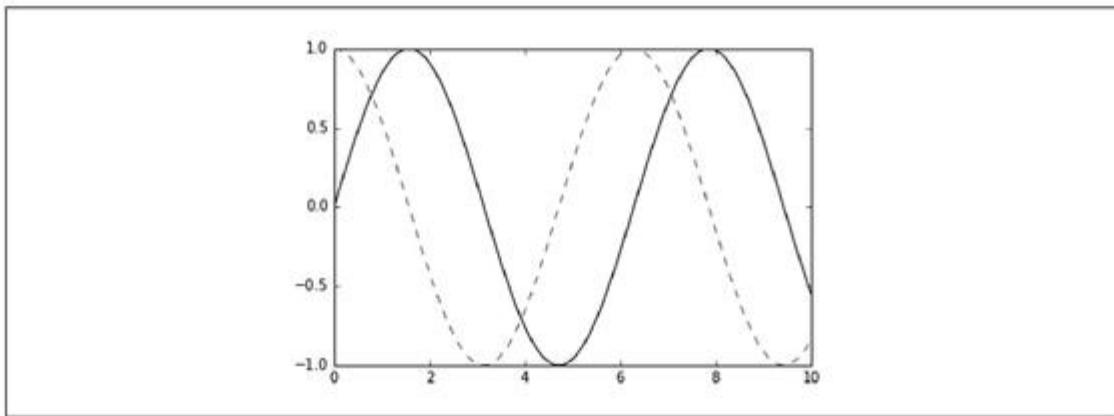
```
In[6]: !ls -lh my_figure.png
```

```
-rw-r--r-- 1 jakevdp staff 16K Aug 11 10:59
ma_figure.png
```

Nous confirmons que le fichier contient cette figure en nous servant d'un objet Image d'IPython pour afficher le contenu du fichier ([Figure 4.2](#)) :

```
In[7]:
```

```
from IPython.display import Image
Image('ma_figure.png')
```



[Figure 4.2](#) : Affichage du contenu d'un fichier PNG.

La fonction `savefig()` détermine le format du fichier à partir des lettres de l'extension de nom du fichier. Le nombre de

formats disponibles dépend des composants de support *backends* installés. Vous obtenez la liste des types reconnus sur votre système en utilisant la méthode suivante sur l'objet canvas de la figure :

In[8]:

```
fig.canvas.get_supported_filetypes()
```

Out[8]:

```
{'eps': 'Encapsulated Postscript',
'jpeg': 'Joint Photographic Experts Group',
'jpg': 'Joint Photographic Experts Group',
'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX',
'png': 'Portable Network Graphics',
'ps': 'Postscript',
'raw': 'Raw RGBA bitmap',
'rgba': 'Raw RGBA bitmap',
'svg': 'Scalable Vector Graphics',
'svgz': 'Scalable Vector Graphics',
'tif': 'Tagged Image File Format',
'tiff': 'Tagged Image File Format'}
```

Notez que vous n'avez pas besoin d'utiliser plt.show() pour montrer la figure, ni aucune commande apparentée lorsque vous n'avez besoin que d'enregistrer la figure.

4.2 : Deux interfaces pour le prix d'une

Pour éviter certaines confusions, il faut savoir que Matplotlib propose deux interfaces : une interface de style MATLAB avec conservation d'état, et une interface orientée objet plus puissante. Découvrons les grandes lignes de ce qui les distingue.

Interface de style MATLAB

Au départ, Matplotlib avait été conçu comme une alternative à Python pour les utilisateurs de la librairie mathématique MATLAB, ce qui se devine dans la syntaxe choisie. Les outils dans le style MATLAB sont réunis dans l'interface nommée pyplot (plt). L'exemple suivant semblera familier à n'importe quel utilisateur de MATLAB ([Figure 4.3](#)) :

```
In[9]: # Créer un tracé
plt.figure()

# Création du 1er contenu de panneau et réglage
des axes
plt.subplot(2, 1, 1) # (rows, columns, panel
number)
plt.plot(x, np.sin(x))
```

```
# Idem pour second panneau  
plt.subplot(2, 1, 2)  
plt.plot(x, np.cos(x));
```

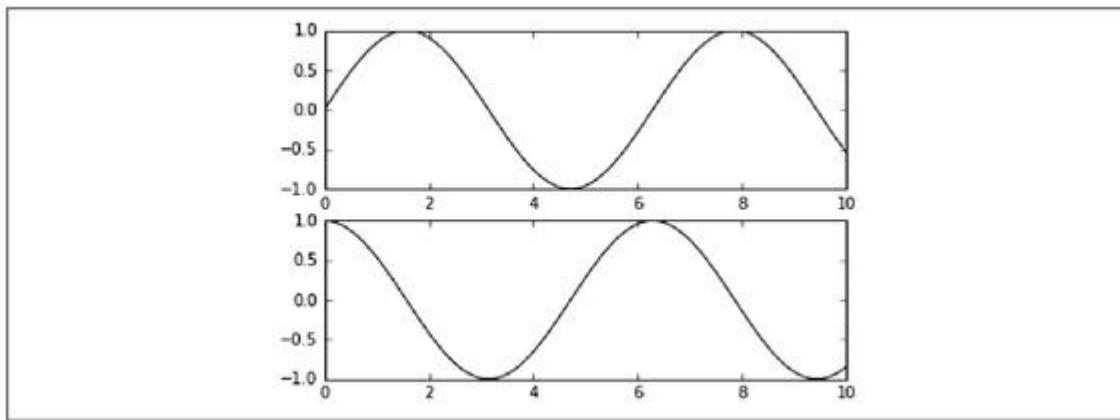


Figure 4.3 : Deux sous-graphiques créés avec l'interface de style MATLAB.

Souvenez-vous que cette interface dispose d'une conservation d'état, c'est-à-dire qu'elle garde en mémoire l'accès à la figure et aux axes affichés, et c'est sur ces éléments que toutes les commandes plt s'appliquent. Vous pouvez obtenir une référence à ces éléments au moyen de la fonction plt.gcf() (*get current figure*) et de plt.gca() (*get current axes*).

Cette approche convient bien aux tracés simples, mais des problèmes peuvent ensuite apparaître. Dans notre exemple, une fois que le second sous-graphique est produit, comment pouvons-nous intervenir à nouveau sur le premier ? Ce n'est

pas impossible avec l'interface MATLAB, mais l'opération est malaisée. Il existe une solution plus efficace.

Interface orientée objet

Dès que le rendu à produire est plus complexe ou lorsque vous avez besoin de mieux contrôler les éléments d'une figure, vous opterez pour l'interface orientée objet. Vous ne dépendez dans ce cas plus d'éléments courants tels qu'une figure ou un axe. Dans l'orientation objet, les fonctions de tracé sont des méthodes d'objets existants de type figure et axe. Voici comment obtenir le même groupe de deux sous-graphiques selon cette approche ([Figure 4.4](#)) :

```
In[10]:  
# création d'une grille de points  
# ax est un tableau de deux objets axes  
fig, ax = plt.subplots(2)  
  
# On appelle plot() sur l'objet désiré  
ax[0].plot(x, np.sin(x))  
ax[1].plot(x, np.cos(x));
```

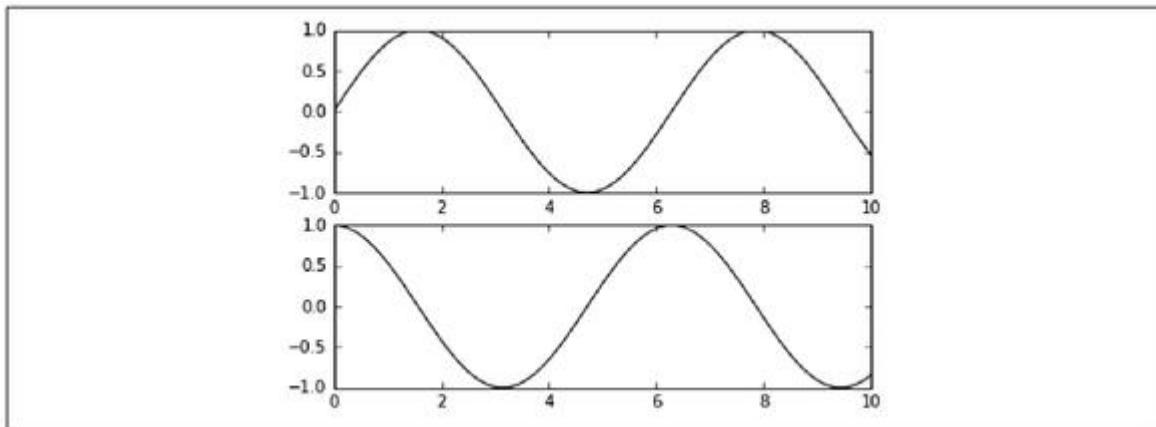


Figure 4.4 : Deux sous-graphiques produits avec l'interface orientée objet.

Tout au long du chapitre, nous opterons selon le cas pour l'approche MATLAB ou pour l'approche orientée objet en fonction des besoins. Dans la plupart des cas, la différence se limite à remplacer `plt.plot()` par `ax.plot()`. Il y a cependant quelques petits pièges que nous soulignerons au moment opportun dans les sections suivantes.

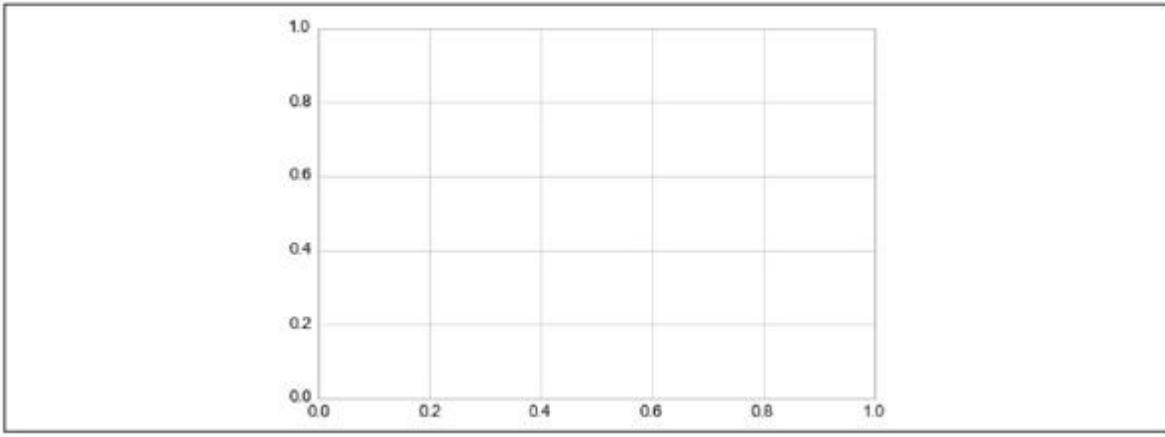
4.3 : Tracé de lignes simples

L'un des tracés les plus simples que l'on puisse imaginer correspond à la fonction mathématique $y = f(x)$. Voyons comment créer un graphique de ce type. Comme dans toutes les sections suivantes, nous commençons par mettre en place le calepin de tracé en important les fonctions dont nous aurons besoin :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
plt.style.use('seaborn-whitegrid')  
import numpy as np
```

Pour un graphique Matplotlib, nous commençons toujours par créer une figure et un repère que nous appelons *ax*. Voici la façon la plus simple de les créer ([Figure 4.5](#)) :

```
In[2]:  
fig = plt.figure()  
ax = plt.axes()
```



[Figure 4.5](#) : Un fond de figure avec un repère orthonormé à deux axes.

Dans Matplotlib, l'élément que l'on nomme figure est une instance de la classe plt.Figure. Vous pouvez le considérer comme un conteneur pour tous les objets qui vont incarner les axes, les graphiques, les textes et les labels. Les éléments nommés axes sont des instances de la classe plt. Axes. Ce sont les axes et le cadre ainsi que les graduations et labels qui vont ensuite accueillir les éléments variables du tracé. Dans la suite du chapitre, nous utiliserons le nom de variable *fig* pour l'instance de la figure et le nom *ax* pour l'instance des axes ou groupes d'axes.

Une fois que nous avons créé un jeu d'axes, nous pouvons utiliser la fonction ax.plot() pour y faire tracer les données. Commençons par une sinusoïde ([Figure 4.6](#)) :

In[3] :

```
fig = plt.figure()  
ax = plt.axes()
```

```
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

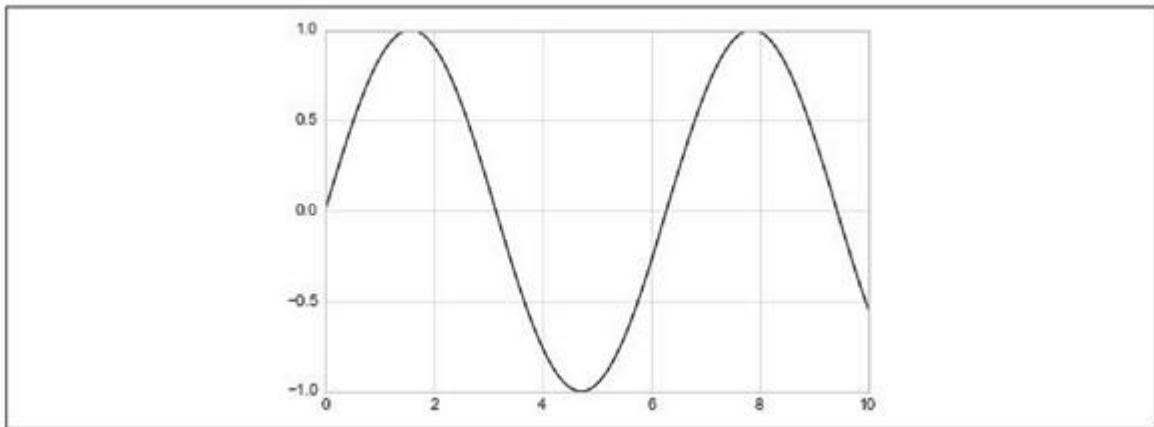


Figure 4.6 : Une sinusoïde simple.

Si nous avions choisi l'approche MATLAB comme décrit dans la section précédente, nous aurions ainsi laissé la figure et les axes être produits automatiquement et l'instruction suivante aurait suffi à obtenir le même résultat :

```
In[4]: plt.plot(x, np.sin(x));
```

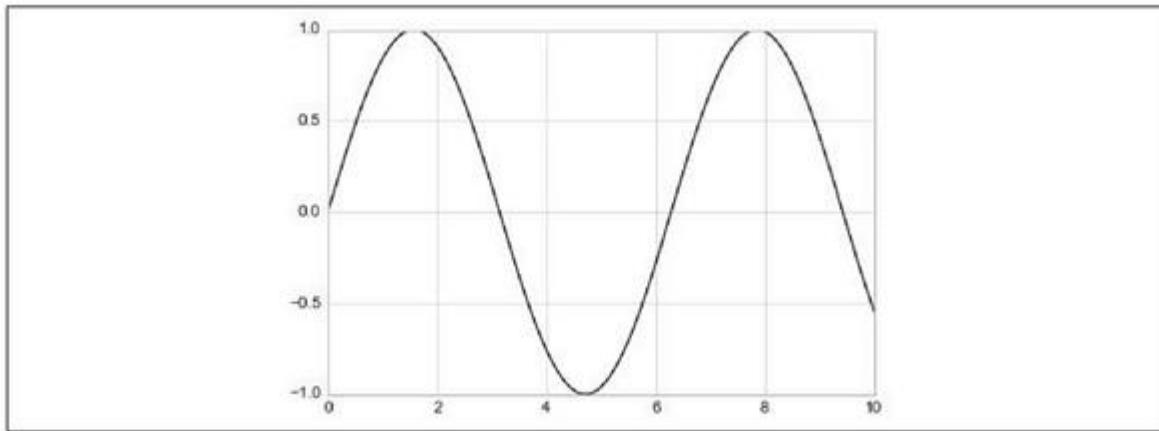


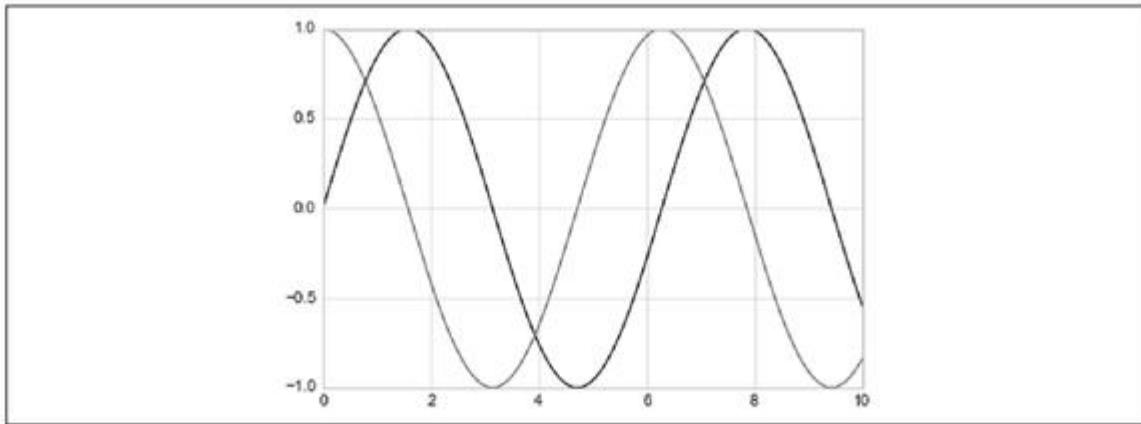
Figure 4.7 : La même sinusoïde en utilisant l'interface MATLAB.

Si nous avons besoin d'afficher plusieurs courbes, il suffit d'appeler plusieurs fois la même fonction `plot()` ([Figure 4.8](#)) :

In[5] :

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

C'est tout ce qu'il y a à dire pour les fonctions de tracé simple dans Matplotlib ! Allons voir les détails de l'aspect des axes et des lignes.



[Figure 4.8](#) : Tracé de plusieurs courbes.

Couleurs et styles de tracé des lignes

La première action de personnalisation d'un graphique concerne en général la couleur et le style de tracé des lignes. La fonction plt.plot() accepte des paramètres supplémentaires à cet effet. Le mot-clé color concerne la couleur de tracé, et vous pouvez lui fournir une chaîne de caractères. Plusieurs syntaxes sont possibles pour définir la couleur ([Figure 4.9](#)) :

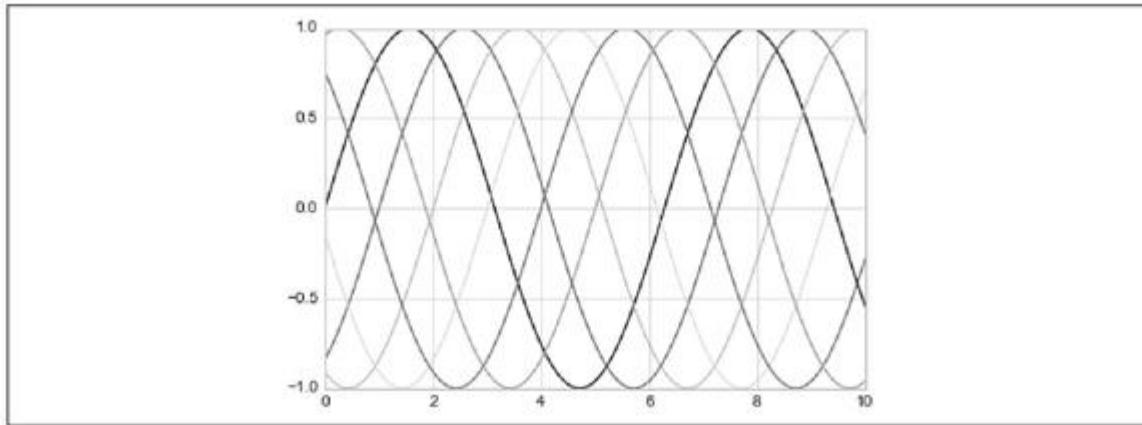
In[6] :

```
plt.plot(x, np.sin(x - 0), color='blue') # Choix par nom  
plt.plot(x, np.sin(x - 1), color='g')    # Choix par code (rgbcmyk)  
plt.plot(x, np.sin(x - 2), color='0.75') # Niveau x de gris entre 0 et 1
```

```

plt.plot(x, np.sin(x - 3), color='#FFDD44') #
Code hexa code (RRGGBB 00-FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3))
# Tuple RGB, 0 ou 1
plt.plot(x, np.sin(x - 5), color='chartreuse');
# Nom de couleur HTML

```



[Figure 4.9](#) : Contrôle de la couleur des lignes à tracer.

Si vous ne spécifiez aucune couleur, Matplotlib va sélectionner une couleur différente pour chaque ligne.

Pour intervenir sur le style de tracé, vous disposez du mot-clé linestyle ([Figure 4.10](#)) :

In[7] :

```

plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

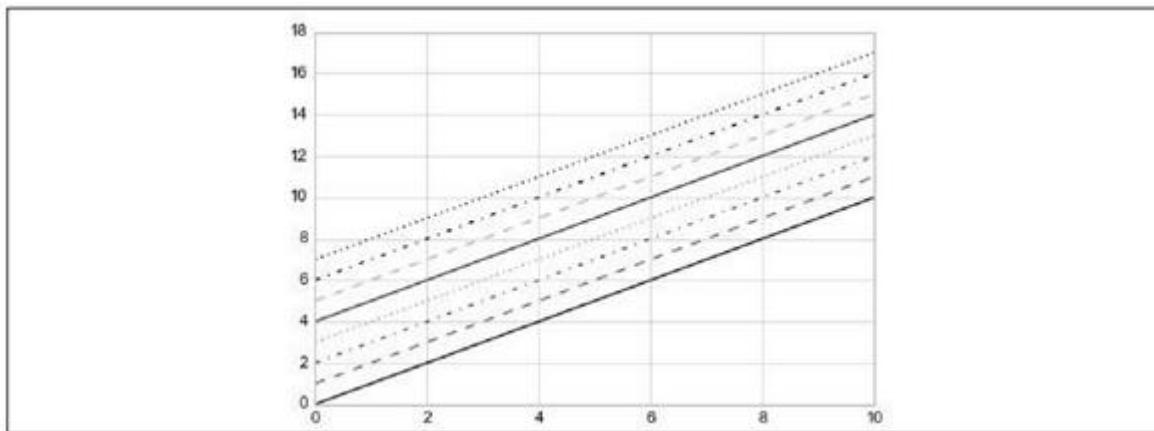
```

Raccourcis de styles de tracé :

```

plt.plot(x, x + 4, linestyle='-' ) # solide
plt.plot(x, x + 5, linestyle='--' ) # tireté
(dashed)
plt.plot(x, x + 6, linestyle='-.') # tiret-point
plt.plot(x, x + 7, linestyle=':' ); # pointillé
(dotted)

```



[Figure 4.10](#) : Quelques exemples de styles de tracé.

Vous pouvez même combiner les codes linestyle et color sous forme d'un code reconnu par la fonction plt.plot() ([Figure 4.11](#)) :

In[8]:

```

plt.plot(x, x + 0, '-g') # solide vert
plt.plot(x, x + 1, '--c') # tireté cyan
plt.plot(x, x + 2, '-.k') # tiret-point noir
plt.plot(x, x + 3, ':r'); # pointillé rouge

```

Les codes couleur sur un caractère correspondent aux abréviations standardisées des systèmes colorimétriques

RGB (Rouge/Verte/Bleu) et CMYK (Cyan/Magenta/Jaune/Noir), répandus dans le monde de la quadrichromie.

D'autres paramètres facultatifs sont disponibles pour personnaliser encore plus l'aspect d'un tracé. Commencez par consulter la chaîne documentaire de la fonction plt.plot() accessible par l'outil d'aide d'IPython (décrit au début du [Chapitre 1](#)).

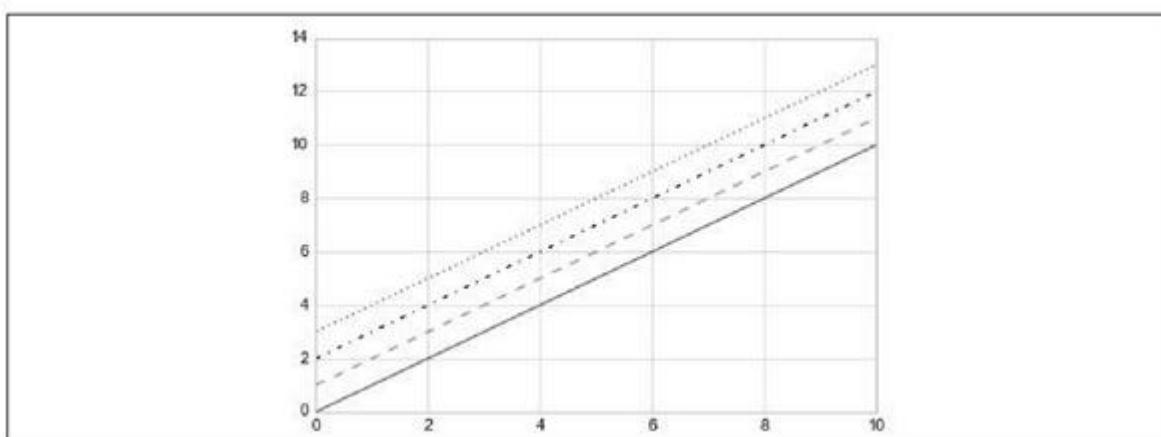
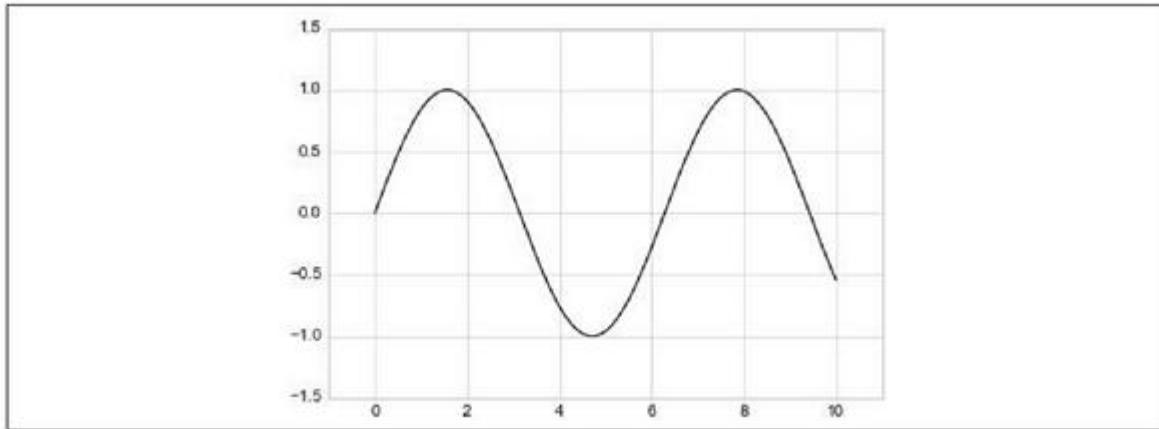


Figure 4.11 : Contrôle des couleurs des styles avec la syntaxe par code abrégé.

Ajustement des limites d'axes d'un tracé

En règle générale, Matplotlib choisit de façon correcte les limites des tracés, mais il est parfois intéressant de les contrôler plus explicitement. Vous disposez à cet effet des deux méthodes plt.xlim() et plt.ylim() ([Figure 4.12](#)) :

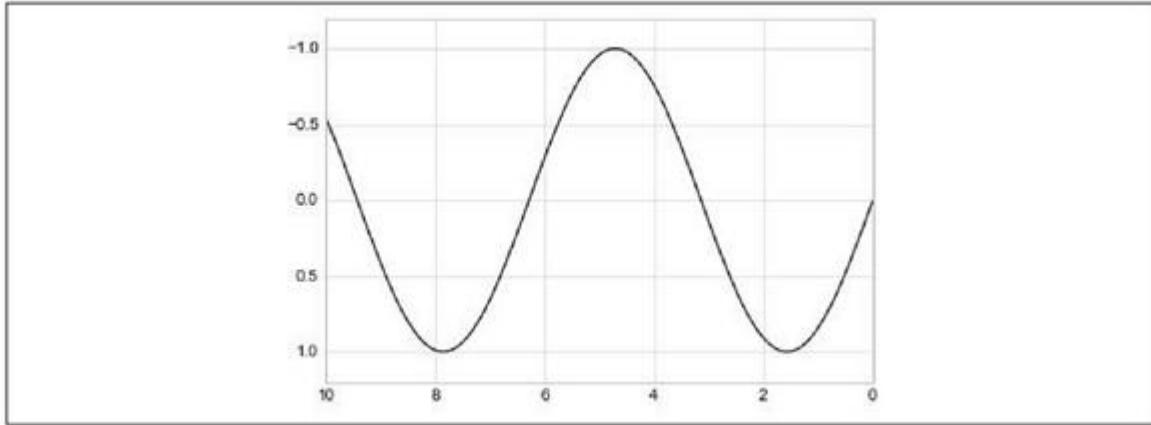
```
In[9]: plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```



[Figure 4.12](#) : Exemple de limites d'axes explicites.

En fournissant la valeur positive d'abord, vous pouvez effectivement renverser l'axe ([Figure 4.13](#)) :

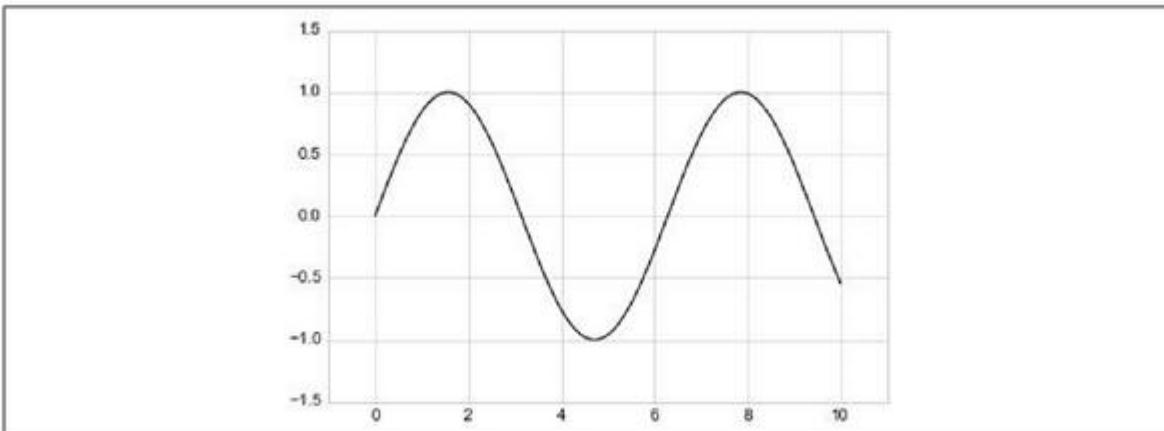
```
In[10]: plt.plot(x, np.sin(x))
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```



[Figure 4.13](#) : Exemple d'inversion de l'axe y.

Vous disposez également de la méthode plt.axis() qui combine le réglage des limites pour les deux axes en un seul appel. Soyez vigilant : il ne faut pas confondre cette méthode axis() (avec un i) avec la méthode axes() (avec un e). Vous fournissez les paramètres dans l'ordre [xmin, xmax, ymin, ymax] ([Figure 4.14](#)) :

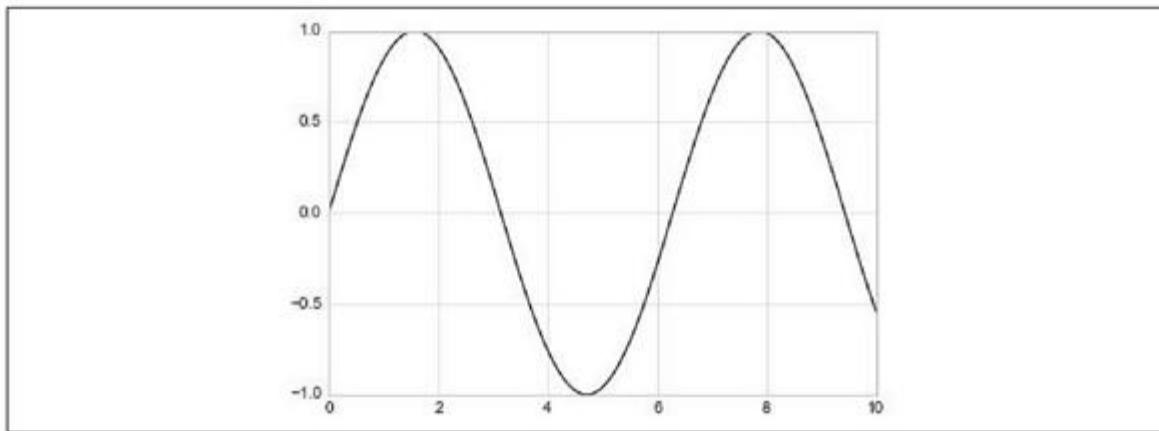
```
In[11]: plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```



[Figure 4.14](#) : Réglage des limites d'axes avec plt.axis.

Cette méthode offre d'autres possibilités, et notamment le recadrage automatique du cadre de tracé en fonction des limites demandées ([Figure 4.15](#)) :

```
In[12]: plt.plot(x, np.sin(x))
plt.axis('tight');
```

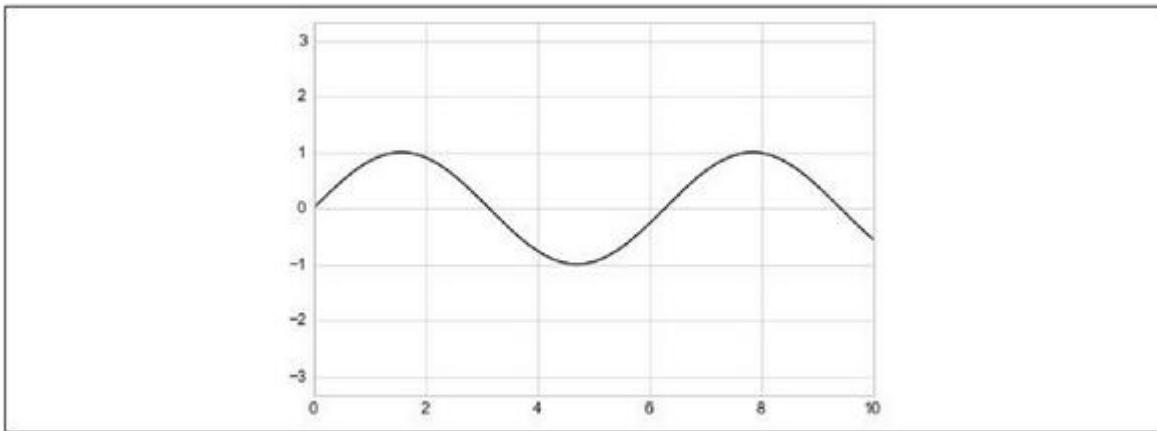


[Figure 4.15](#) : Exemple de recadrage serré en fonction des limites.

Cette même méthode permet en outre de demander l'utilisation de la même échelle pour les deux axes x et y

([Figure 4.16](#)) :

```
In[13]: plt.plot(x, np.sin(x))
plt.axis('equal');
```



[Figure 4.16](#) : Effet de l'option « equal » pour homogénéiser les unités des axes.

Reportez-vous à la chaîne documentaire de plt.axis() pour en savoir plus au sujet des limites d'axes.

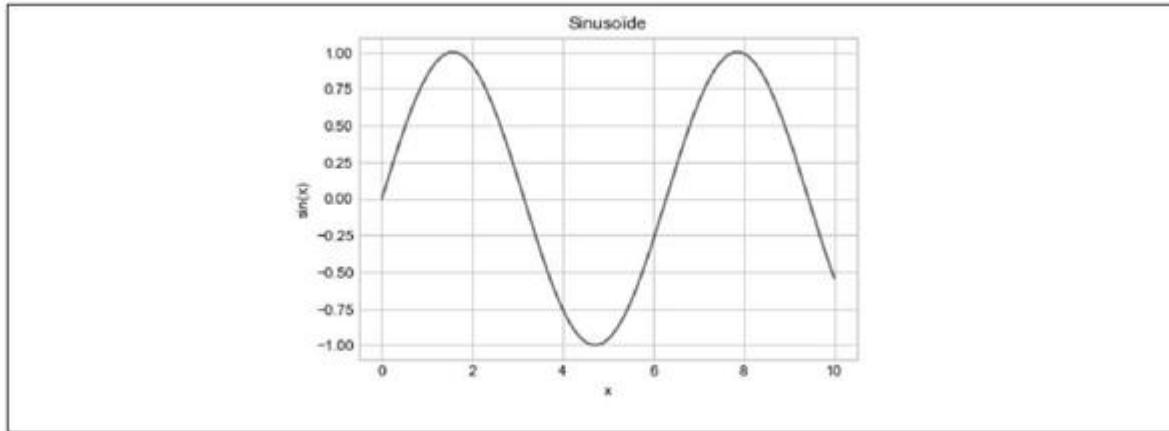
Titres et labels de tracé

Découvrons pour terminer cette section l'ajout de titres et sous-titres pour le tracé, les axes et le cartouche de légende.

Pour les titres et labels d'axes, vous disposez de méthodes dédiées à leur personnalisation ([Figure 4.17](#)) :

```
In[14]:
plt.plot(x, np.sin(x))
plt.title("Sinusoïde")
```

```
plt.xlabel("x")
plt.ylabel("sin(x)");
```



[Figure 4.17](#) : Ajout d'un titre de tracé et de labels d'axes.

Plusieurs paramètres facultatifs permettent de retoucher la position, la taille et le style des légendes. Voyez la documentation de Matplotlib et celle de chaque fonction pour en savoir plus.

Dès que vous présentez plusieurs séries de valeurs dans un même tracé, il devient intéressant de prévoir un cartouche de légende pour identifier chaque ligne. Il suffit d'utiliser la méthode dédiée plt.legend(). Plusieurs techniques sont utilisables. Je trouve que la plus confortable consiste à définir le label de chaque ligne au moyen du mot-clé label de la fonction de tracé plot() ([Figure 4.18](#)) :

In[15]:

```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
```

```
plt.plot(x, np.cos(x), ':b', label='cos(x)')  
plt.axis('equal')  
plt.legend();
```

Vous constatez que la méthode plt.legend() tient compte du style et de la couleur de tracé et affiche le label approprié. Voyez la documentation de cette méthode pour en savoir plus. Nous reviendrons sur la personnalisation des légendes de tracé un peu plus loin dans ce même chapitre.

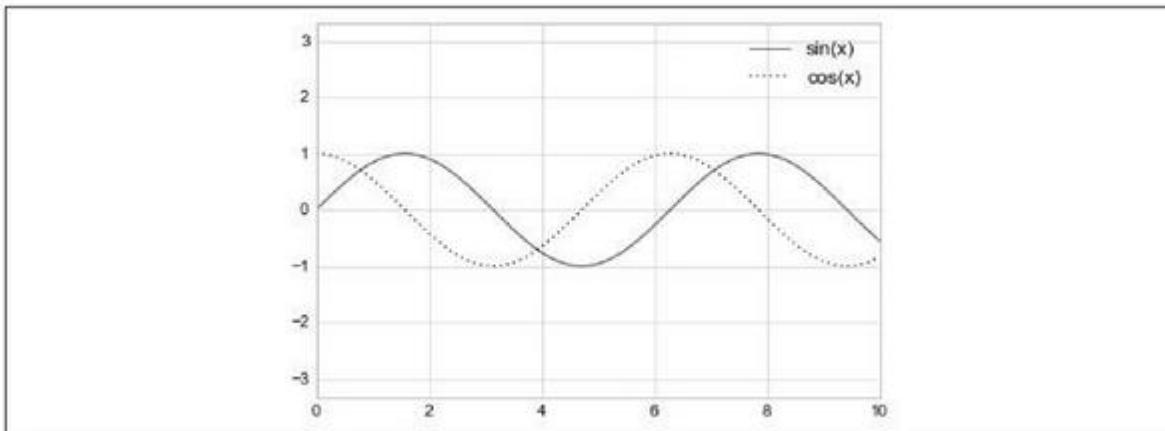


Figure 4.18 : Exemple de cartouche de légende d'un tracé.

4.4 : Tracé en nuage de points

Un genre de tracé proche du tracé de courbe est le tracé par points ou nuage de points (*scatter*). Au lieu de montrer une ligne continue, il présente une succession de points, le symbole utilisé pouvant être personnalisé. Commençons par préparer notre calepin pour effectuer des tracés et importons les fonctions dont nous aurons besoin :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
plt.style.use('seaborn-whitegrid')  
import numpy as np
```

Lignes pointillées avec plt.plot

Nous avons vu dans la section précédente comment tracer des lignes continues avec plt.plot() ou ax.plot(). La même fonction permet de produire un tracé en pointillé ([Figure 4.20](#)) :

```
In[2]:  
x = np.linspace(0, 10, 30)  
y = np.sin(x)  
plt.plot(x, y, 'o', color='black');
```

Faux homonymes dans Matplotlib

La plupart des méthodes de la famille plt portent le même nom dans la famille ax, par exemple plt.plot() → ax.plot(), plt.legend() → ax.legend(), etc., mais ce n'est pas le cas de toutes. Celles qui concernent les limites, les labels et les titres ne sont pas homonymes. Voici l'équivalence entre les noms des fonctions dans le mode MATLAB et celles dans le mode orienté objet :

plt.xlabel() → ax.set_xlabel()

plt.ylabel() → ax.set_ylabel()

plt.xlim() → ax.set_xlim()

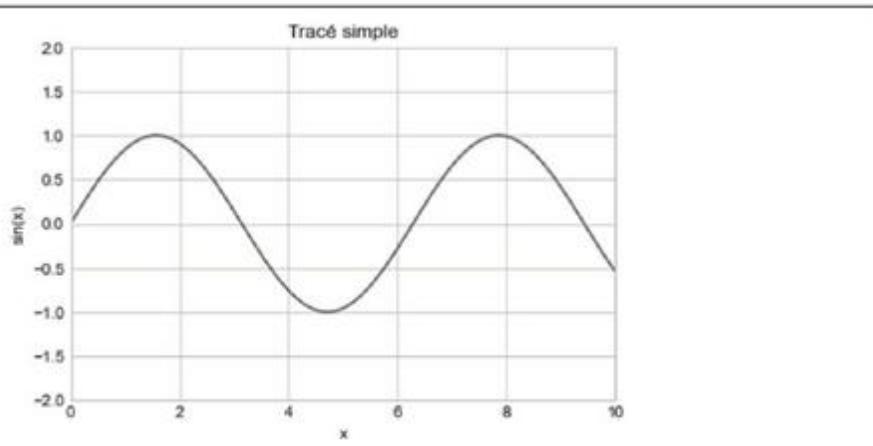
plt.ylim() → ax.set_ylim()

plt.title() → ax.set_title()

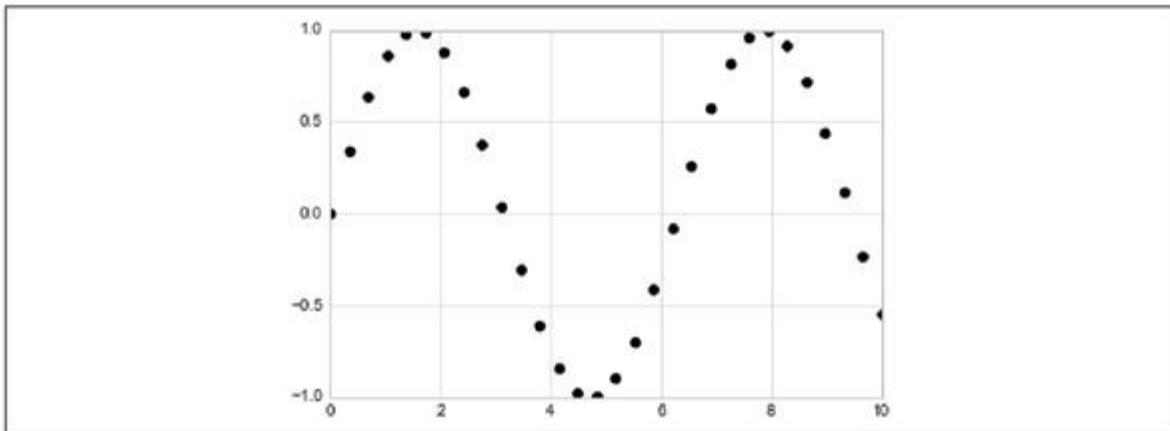
Dans l'approche orientée objet, il est souvent plus pratique d'utiliser au lieu de ces différentes fonctions individuelles la méthode ax.set() qui permet de régler tous les paramètres en une fois ([Figure 4.19](#)) :

In[16]:

```
ax = plt.axes()
ax.plot(x, np.sin(x))
ax.set(xlim = (0, 10), ylim = (-2, 2),
       xlabel = 'x',      ylabel = 'sin(x)',
       title  = 'Un tracé simple');
```



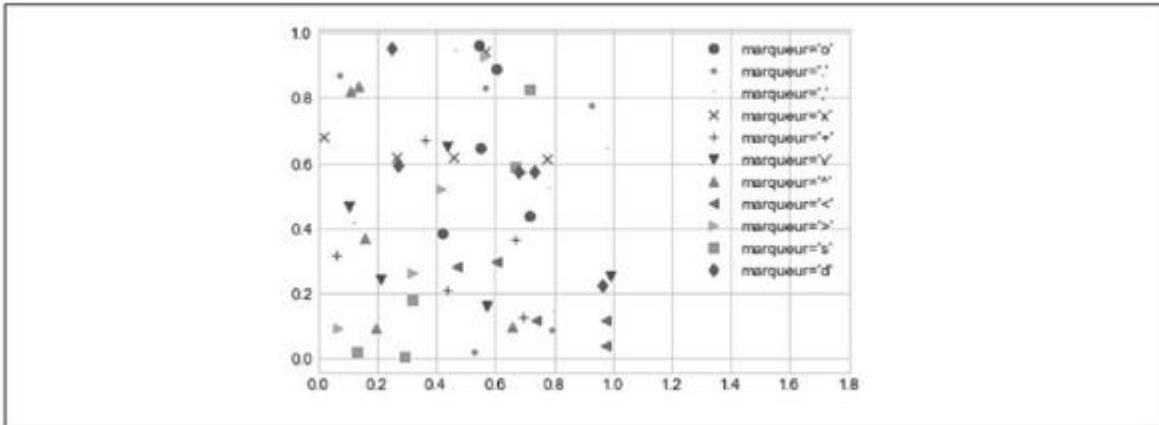
[Figure 4.19](#) : Réglage de plusieurs propriétés par un seul appel à `ax.set()`.



[Figure 4.20](#) : Exemple de tracé d'une ligne pointillée.

Le troisième paramètre de la fonction permet de choisir le type de symbole qui doit représenter le point. Vous disposez d'options telles que'-' et '--' pour contrôler le style de tracé d'une ligne continue ; il existe de même des codes pour les styles de pointillés. La liste complète est fournie dans la documentation de plt.plot() et dans celle de Matplotlib. La plupart des options sont faciles à deviner. Le test suivant montre l'utilisation d'un certain nombre de ces symboles ([Figure 4.21](#)) :

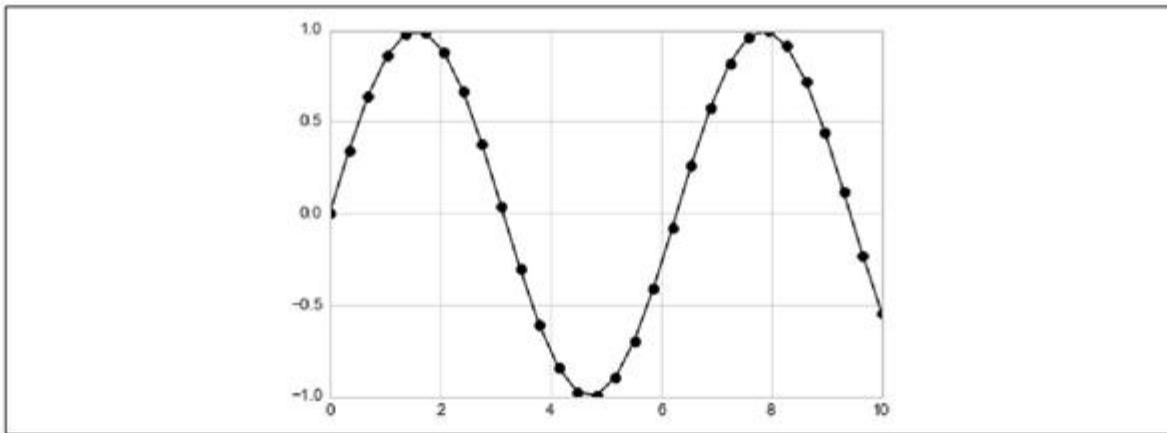
```
In[3] :  
rng = np.random.RandomState(0)  
for marker in ['o', '.', ',', 'x', '+', 'v',  
'^', '<', '>', 's', 'd']:   
    plt.plot(rng.rand(5), rng.rand(5), marker,  
             label = "marqueur='{}'".format(marker))  
  
plt.legend(numpoints=1)  
plt.xlim(0, 1.8);
```



[Figure 4.21](#) : Passage en revue de symboles de points.

Ces codes de caractères peuvent être combinés avec des codes de lignes pleines et de couleur pour sur-imposer les points à une ligne continue ([Figure 4.22](#)) :

```
In[4]: plt.plot(x, y, '-ok');      # ligne (-),
 cercle (o), noir (k)
```



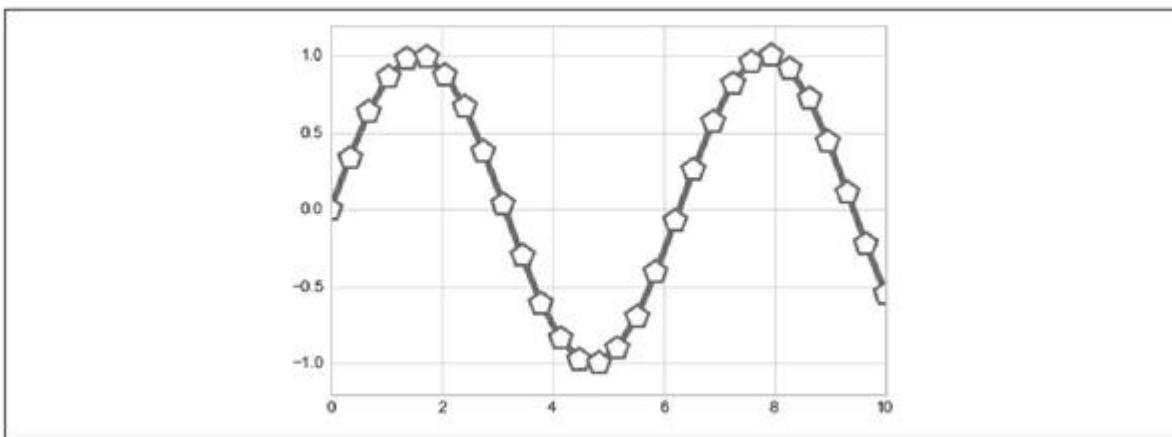
[Figure 4.22](#) : Combinaison d'une ligne continue et marqueur de points.

D'autres paramètres nominatifs pour plt.plot() permettent d'affiner les choix en terme de lignes et de marqueurs

([Figure 4.23](#)) :

In[5]:

```
plt.plot(x, y, '-p', color = 'gray',
          markersize = 15, linewidth = 4,
          markerfacecolor = 'white',
          markeredgecolor = 'gray',
          markeredgewidth = 2)
plt.ylim(-1.2, 1.2);
```



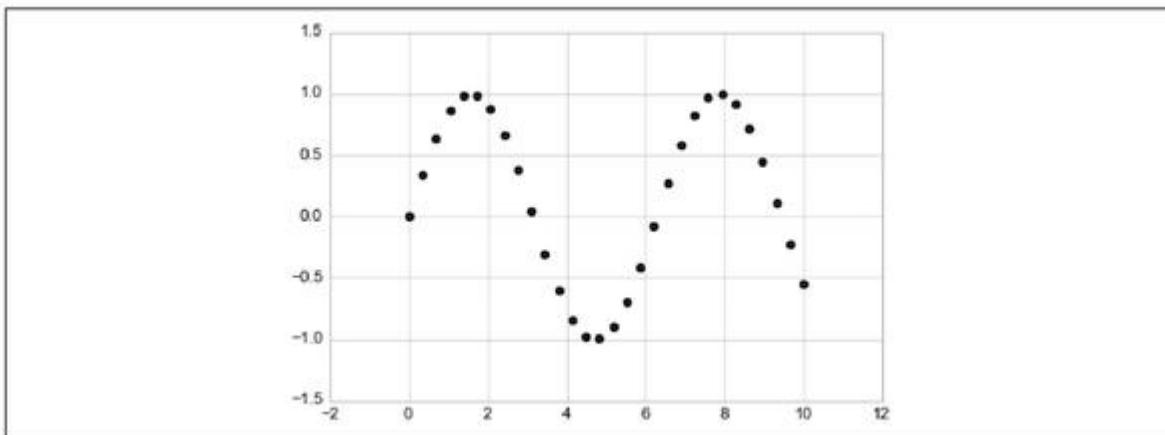
[Figure 4.23](#) : Personnalisation d'une ligne et des points.

Les possibilités de personnalisation sont énormes. Pour tout autre détail, voyez la documentation de plt.plot().

Ligne pointillée avec plt.scatter

Une solution encore plus efficace pour créer des lignes pointillées est le recours à la fonction plt.scatter(), dont l'utilisation est très proche de celle de plt.plot() ([Figure 4.24](#)) :

```
In[6]: plt.scatter(x, y, marker='o');
```



[Figure 4.24](#) : Tracé d'une ligne pointillée simple.

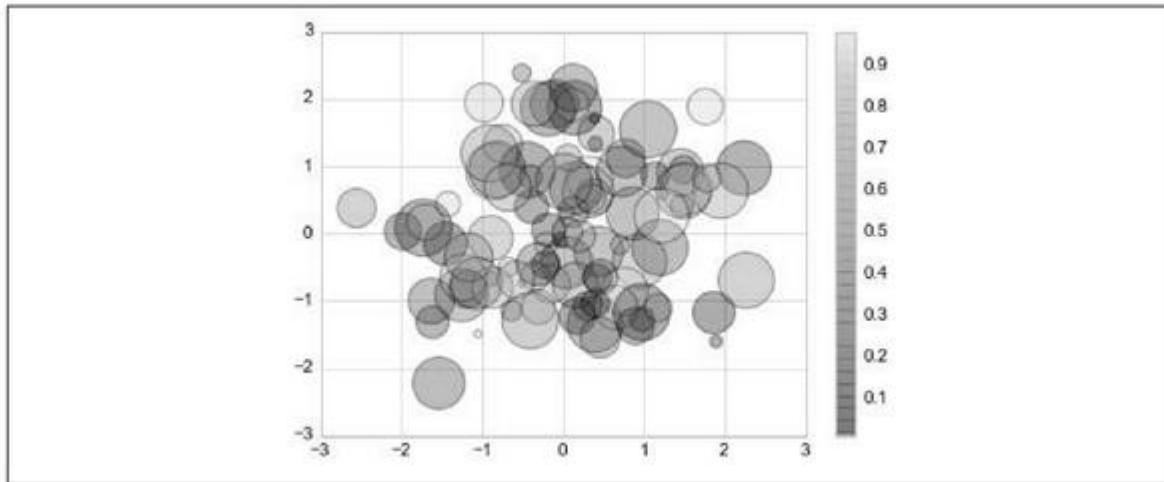
La méthode `plt.scatter()` est une sorte de version dédiée de `plt.plot()`, car elle permet de choisir précisément les propriétés des points, un à un : taille, couleur de remplissage et de contour, *etc.* Ces propriétés peuvent même être mises en relation avec les données à visualiser.

Voyons cela par un exemple comportant des points d'une grande variété de couleurs et de tailles. Pour montrer les chevauchements qui vont en résulter, nous nous servons du paramètre de transparence alpha ([Figure 4.25](#)) :

```
In[7]:
```

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)
```

```
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,  
cmap='viridis')  
plt.colorbar(); # show color scale
```



[Figure 4.25](#) : Variation de taille, de couleur et de transparence d'un nuage de points.

Vous remarquez que l'argument de couleur est associé d'office à une échelle de couleurs qui est rendue visible par la commande `colorbar()`. Le paramètre de taille est exprimé en pixels. Ces deux paramètres permettent donc d'ajouter des informations au sujet de deux autres dimensions des données d'entrée.

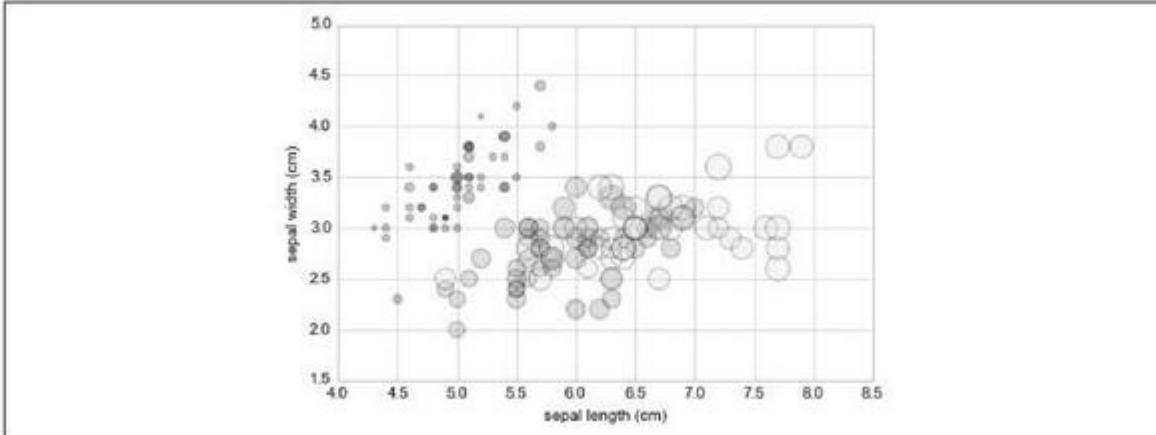
Nous réutilisons le jeu de données des iris de Scikit-Learn. Chaque échantillon correspond à un type de fleur parmi trois, et nous connaissons la taille des pétales et des sépales ([Figure 4.26](#)) :

In[8]:

```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T

plt.scatter(features[0], features[1], alpha=0.2,
            s=100*features[3], c=iris.target,
            cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1]);
```

Ce graphique visualise effectivement quatre dimensions des données : les deux dimensions de base en x et y montrent la longueur et la largeur des sépales ; le diamètre de chaque point montre la largeur des pétales, et la couleur correspond à l'espèce de fleur concernée. De tels nuages de points à plusieurs dimensions permettent une exploration et une présentation très efficace des données.



[Figure 4.26](#) : Utilisation des propriétés des points pour ajouter des caractéristiques aux données des iris.

Performances de `plot()` et de `scatter()`

Puisque `plt.scatter()` offre bien plus de possibilités que `plt.plot()`, il semble logique de se tourner vers elle pour tous les nuages de points. Pour de petits volumes de données, cela reste vrai, mais dès qu'il y a plus que quelques milliers de points à afficher, `plt.plot()` sera bien plus efficace. En effet, `plt.scatter()` doit préparer chaque point au niveau taille et couleur avant son affichage. La méthode `plt.plot()` utilise un format unique pour tous les points, ce qui permet de configurer le point une fois en début de traitement. Lorsque le jeu de données est imposant, les performances sont bien meilleures avec `plt.plot()`.

4.5 : Visualisation des erreurs

Dans tout travail scientifique, il est souvent presque aussi important, et parfois même plus, de bien traiter et afficher les marges d'erreur, plutôt que les valeurs elles-mêmes. Supposez qu'il nous faille réaliser des observations astrophysiques pour obtenir une estimation de la constante de Hubble, qui est la mesure locale du taux d'expansion de l'univers. Les recherches académiques actuelles donnent une valeur d'environ 71 km/s/Mpc . De mon côté, j'ai mesuré une valeur de 74 km/s/Mpc avec mon approche. Les deux valeurs sont-elles cohérentes ? Si je n'ai pas d'autres informations, il n'y a aucun moyen de le savoir.

Supposons maintenant que je puisse connaître les marges d'erreur : la recherche actuelle indique une valeur de $71 \pm 2.5 \text{ (km/s)/Mpc}$, et mes proches recherches trouvent $74 \pm 5 \text{ (km/s)/Mpc}$. Les valeurs sont-elles dorénavant cohérentes ? Il devient possible de répondre de façon quantitative.

La possibilité de visualiser les marges d'erreur permet de présenter des informations plus exhaustives dans les résultats des traitements.

Barre d'erreurs simple

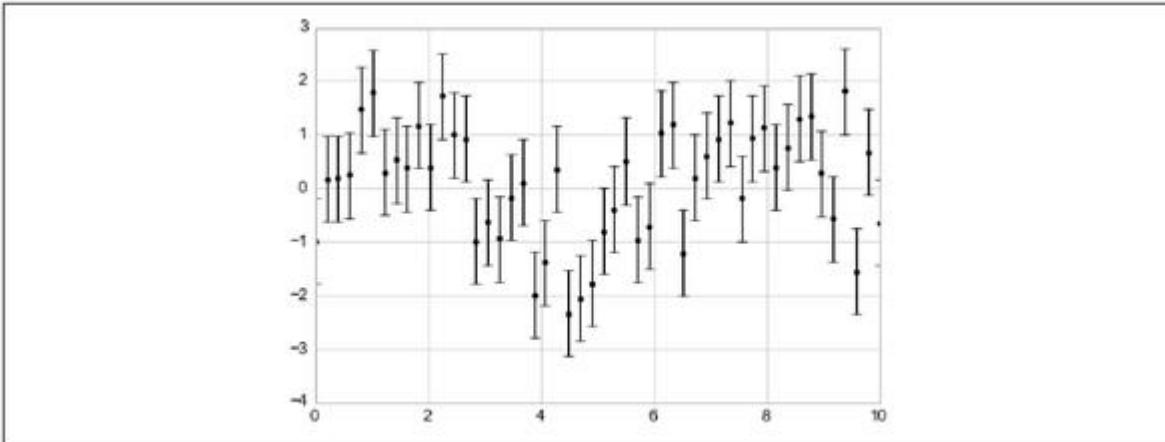
Vous pouvez ajouter une barre d'erreurs simple à un diagramme par un appel à la méthode plt. errorbar() ([Figure 4.27](#)) :

In[1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

In[2]:

```
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



[Figure 4.27](#) : Exemple d'ajout de barre d'erreurs.

Le paramètre `fmt` est un code de format pour choisir l'aspect des lignes et des points. Sa syntaxe est la même que pour le raccourci utilisé dans `plt.plot()` et vu dans la section précédente.

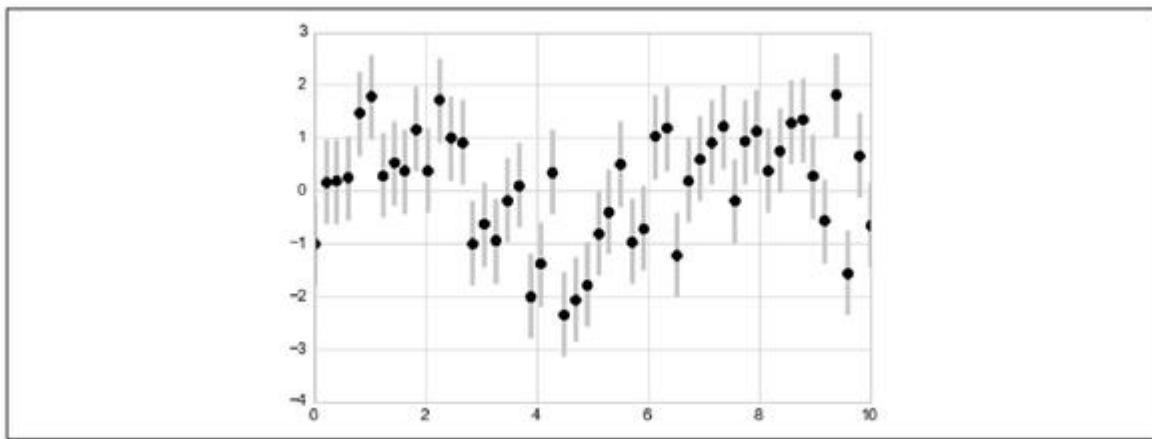
La méthode `errorbar()` offre toute une foule d'options pour personnaliser le résultat. J'ai souvent besoin de rendre les barres d'erreurs en gris moyen ou clair pour ne pas surcharger l'affichage des points de mesure ([Figure 4.28](#)) :

In[3] :

```
plt.errorbar(x, y, yerr=dy, fmt='o',
              color='black',
              ecolor='lightgray', elinewidth=3,
              capsize=0);
```

D'autres options sont disponibles : affichage des barres d'erreurs horizontales avec `xerr`, barres décentrées, *etc.* Pour

tout détail, voyez la documentation de plt.errorbar().



[Figure 4.28](#) : Affichage des barres d'erreurs en gris.

Plage d'erreurs continues

Vous aurez parfois besoin de montrer des barres d'erreurs pour des quantités continues. Bien que la librairie Matplotlib n'offre pas de routine pour répondre à ce besoin, on peut assez aisément aboutir au même résultat en combinant des primitives telles que plt.plot() et plt.fill_between(). La [Figure 4.29](#) en montre un exemple.

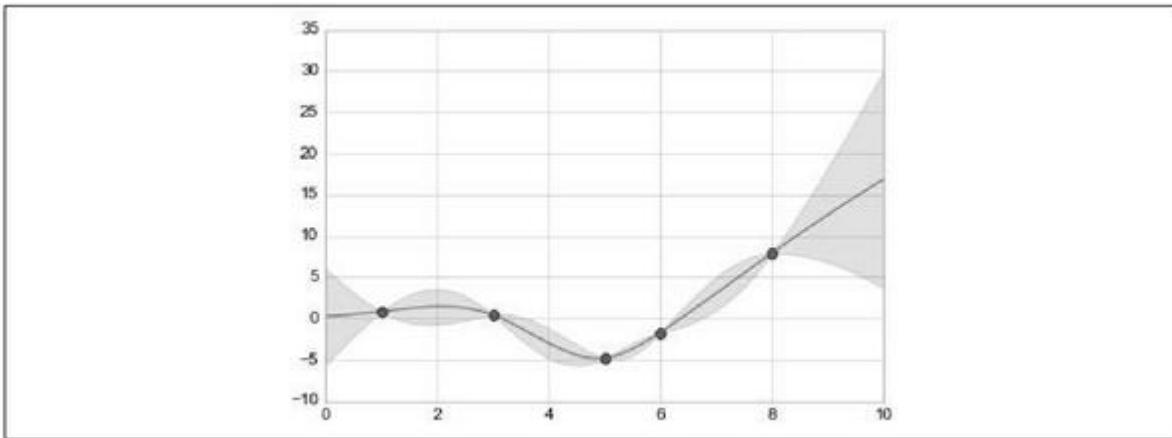


Figure 4.29 : Remplissage de la plage de marge d'erreurs continues.

4.6 : Tracé de densités et de contours

Il est dans certains cas intéressant d'afficher en deux dimensions des données qui sont en trois dimensions, en utilisant la variation de contours ou de couleurs de surface. Nous disposons de trois fonctions Matplotlib à cet effet : plt.contour() pour tracer des contours, plt.contourf() pour des contours avec remplissage et plt.imshow() pour afficher des images. Nous allons découvrir plusieurs exemples, en commençant bien sûr par préparer le calepin en important les fonctions dont nous aurons besoin :

In[1] :

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
```

Affichage d'une fonction à trois dimensions

Voyons d'abord comment créer un tracé de contours en partant d'une fonction $z = f(x,y)$. Voici la définition de cette fonction :

```
In[2]:
```

```
def f(x, y):  
    return np.sin(x) ** 10 + np.cos(10 + y * x) *  
    np.cos(x)
```

La fonction plt.contour() que nous allons utiliser attend trois paramètres d'entrée : une matrice de valeurs en x, une matrice de valeurs en y et une matrice de valeurs en z. Les valeurs en x et y correspondent à des positions sur le graphique et la valeur en z à des niveaux de contours. Pour disposer de ce format de données, nous nous servons de la fonction np.meshgrid() qui construit une matrice à deux dimensions à partir d'un tableau à une dimension :

```
In[3]:
```

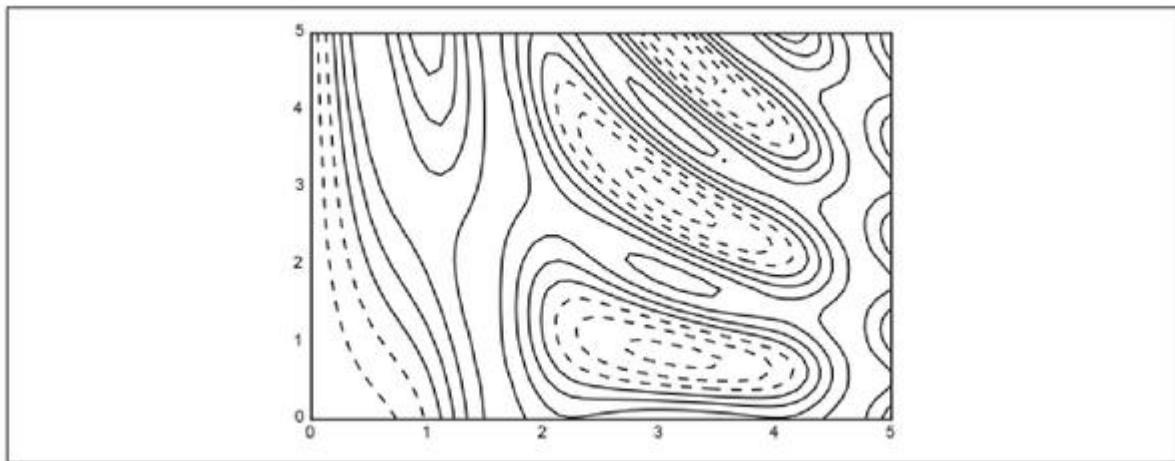
```
x = np.linspace(0, 5, 50)  
y = np.linspace(0, 5, 40)  
  
X, Y = np.meshgrid(x, y)  
Z = f(X, Y)
```

Nous pouvons maintenant demander la création d'un tracé de contours avec notre fonction ([Figure 4.30](#)) :

```
In[4]: plt.contour(X, Y, Z, colors='black');
```

Lorsqu'une seule couleur est utilisée par défaut, les valeurs négatives sont des lignes tiretées et les valeurs positives des

lignes continues. Vous pouvez bien sûr choisir des codes couleur en précisant une carte de couleurs grâce au paramètre `cmap`. Profitons-en pour afficher un plus grand nombre de lignes, en l'occurrence 20, régulièrement espacées au sein de la plage de valeurs des données ([Figure 4.31](#)) :



[Figure 4.30](#) : Visualisation d'une troisième dimension grâce aux contours.

```
In[5]: plt.contour(X, Y, Z, 20, cmap='RdGy');
```

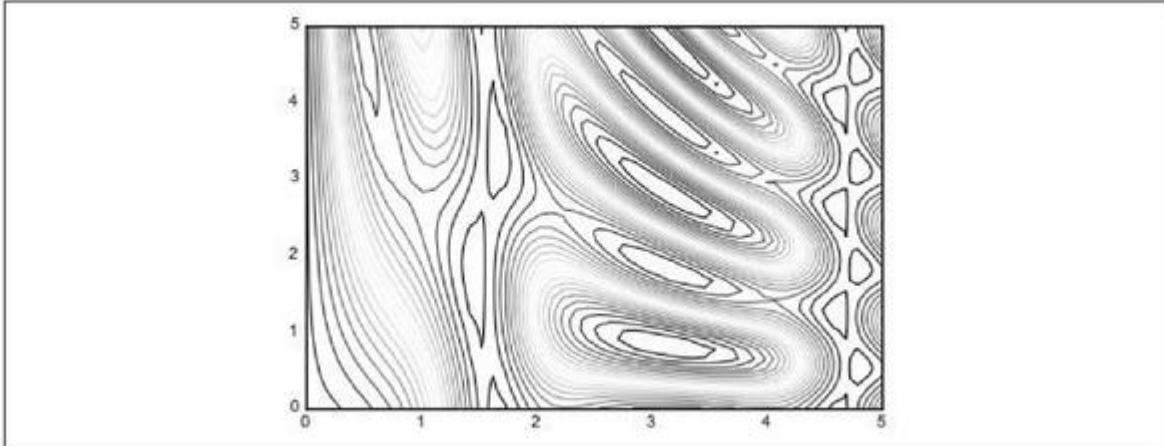


Figure 4.31 : Trois dimensions de données avec des contours colorés.

Dans l'exemple, nous utilisons la carte de couleurs RdGy (qui signifie Rouge-Gris). Ce jeu de couleurs convient bien aux données centrées. La librairie offre plusieurs autres cartes de couleurs dont vous pouvez connaître la liste en demandant l'accès à l'aide du module plt.cm ainsi :

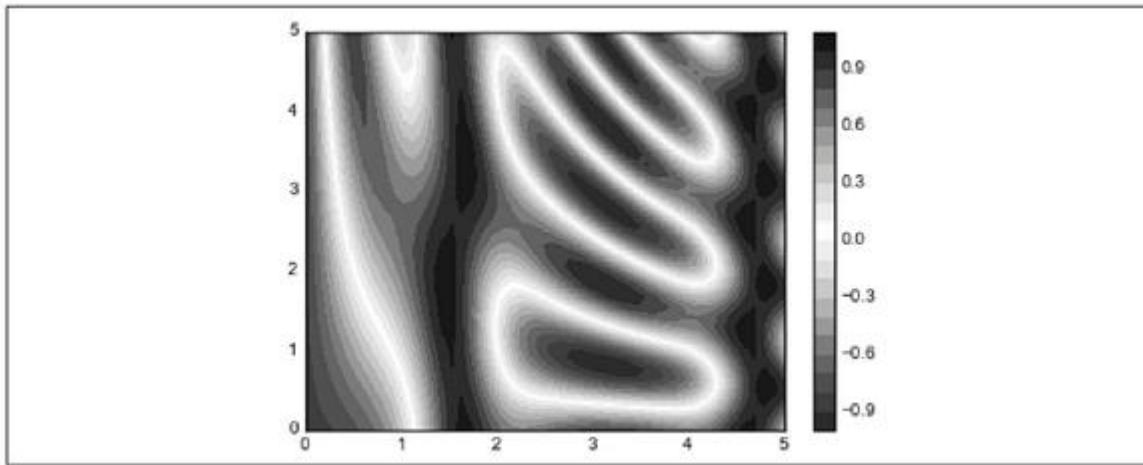
```
plt.cm.<TAB>
```

Le dernier tracé est devenu assez précis, mais les espaces entre les lignes peuvent déranger. Il suffit d'utiliser la fonction plt.contourf() pour remplir entre les contours. La syntaxe est très proche de celle de plt.contour().

Nous n'oublions pas d'ajouter la commande plt.colorbar() pour faire afficher sur la droite une légende des couleurs (Figure 4.32) :

In[6] :

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')  
plt.colorbar();
```



[Figure 4.32](#) : Tracé de contours avec remplissage.

La barre de couleurs à droite permet de savoir que les régions sombres sont des crêtes alors que les régions rouges sont des vallées.

Ce mode de tracé des contours souffre d'un effet d'escalier entre les valeurs. On peut y remédier en demandant un très grand nombre de contours, mais cela rend la création du graphique peu efficace, puisque Matplotlib doit générer un nouveau polygone pour chaque niveau. Il est préférable d'utiliser dans ce cas la fonction plt.imshow() qui interprète une grille de données à deux dimensions en tant qu'image.

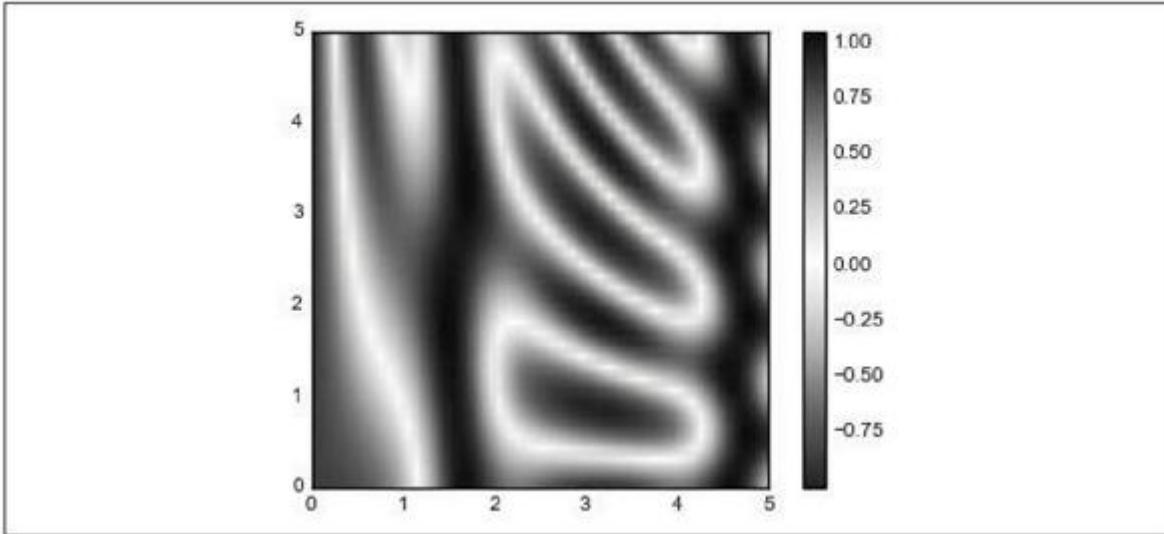
Le code suivant produit ce que montre la [Figure 4.33](#) :

In[7]:

```
plt.imshow(Z, extent=[0, 5, 0, 5],  
origin='lower', cmap='RdGy')  
plt.colorbar()
```

La méthode imshow() souffre de quelques limitations :

- plt.imshow() ne peut traiter directement une grille en x et y. Vous devez spécifier les plages de l'image sur le tracé sous forme [xmin, xmax, ymin, ymax].
- Par défaut, plt.imshow() utilise le coin supérieur gauche comme origine des coordonnées et non le coin inférieur gauche habituellement utilisé dans les tracés de contours. Un repositionnement est donc à réaliser avant d'afficher les données.
- plt.imshow() ajuste automatiquement les proportions des axes pour les faire coïncider avec celles des données d'entrée.



[Figure 4.33](#) : Affichage de données en trois dimensions sous forme d'image.

Il est dans certains cas intéressant de combiner un tracé de contours avec un tracé d'images. Pour obtenir ce que montre la [Figure 4.34](#), nous utilisons une image de fond avec une transparence définie par le paramètre alpha sur laquelle nous faisons tracer des contours qui portent les labels grâce à la fonction plt.clabel() :

In[8] :

```
contours = plt.contour(X, Y, Z, 3,
colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[0, 5, 0, 5],
origin='lower', cmap='RdGy', alpha=0.5)

plt.colorbar();
```

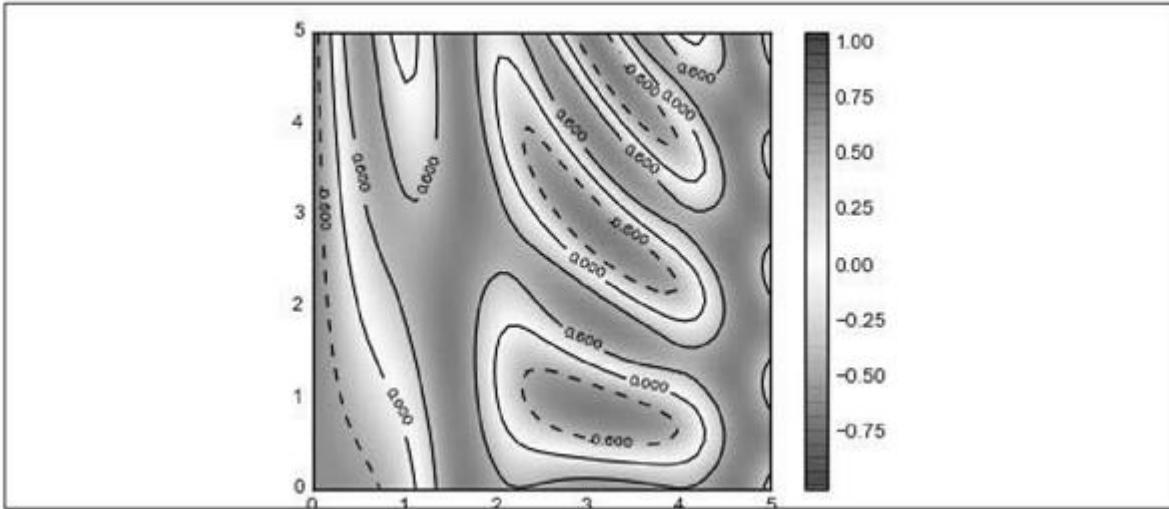


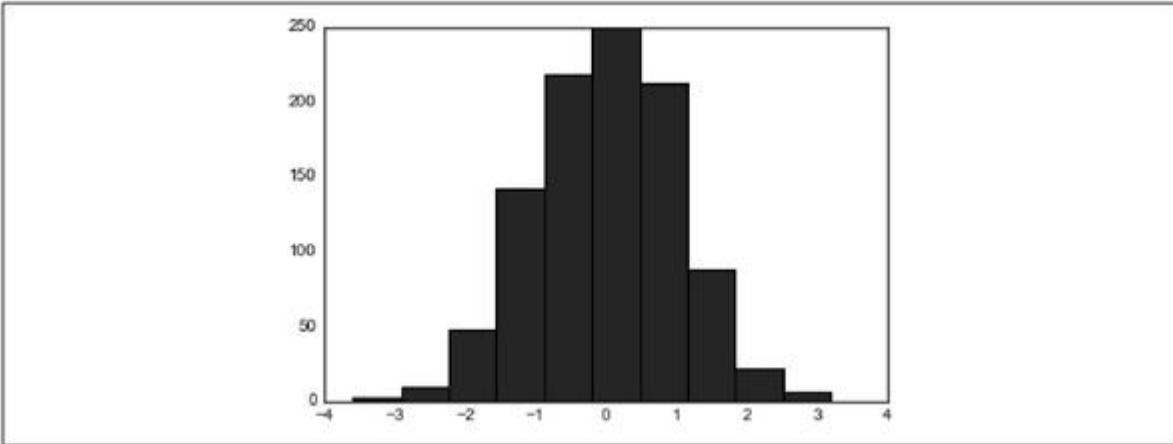
Figure 4.34 : Des contours avec labels sur-imposés à une image.

En exploitant ces trois fonctions `plt.contour()`, `plt.contourf()` et `plt.imshow()`, vous pouvez contrôler en détail la manière dont vous affichez des données en trois dimensions dans le cadre d'un graphique en deux dimensions. Vous utiliserez la chaîne de documentation des fonctions pour en savoir plus. Voyez aussi la section ultérieure du chapitre qui aborde les tracés en trois dimensions dans Matplotlib.

4.7 : Histogrammes, bins et densité

Souvent, une excellente première approche d'un jeu de données consiste à demander l'affichage d'un histogramme ou diagramme à barres verticales. Nous avons déjà vu la fonction de tracé d'histogramme de Matplotlib dans le chapitre précédent. Après avoir demandé les imports nécessaires, nous avions réussi à créer un histogramme en une seule instruction ([Figure 4.35](#)) :

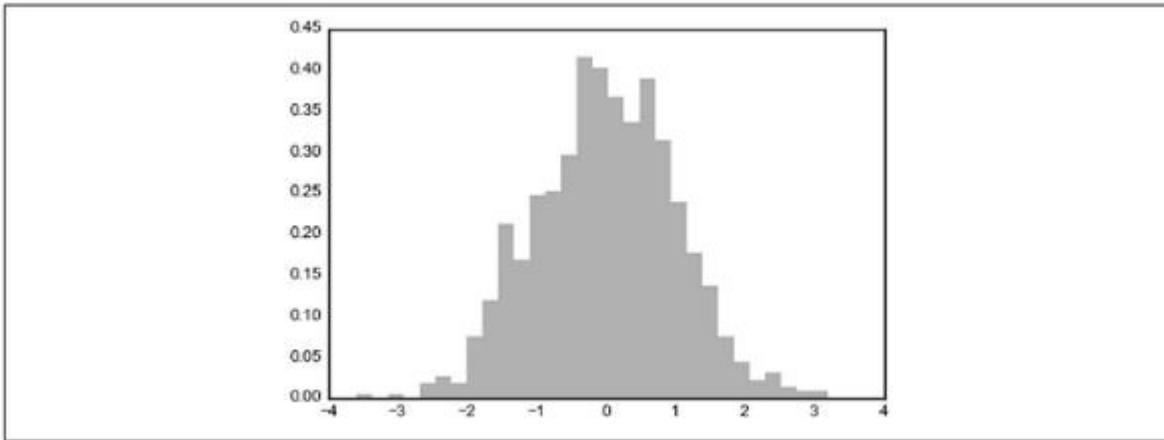
```
In[1]:  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt  
plt.style.use('seaborn-white')  
  
data = np.random.randn(1000)  
  
In[2]: plt.hist(data);
```



[Figure 4.35](#) : Un histogramme simple.

La fonction nommée `hist()` dispose de nombreuses options pour contrôler aussi bien les calculs que l'aspect. Voici une version plus personnalisée d'un histogramme ([Figure 4.36](#)) :

```
In[3]:  
plt.hist(data, bins=30, density=True, alpha=0.5,  
         histtype='stepfilled',  
         color='steelblue',  
         edgecolor='none');
```



[Figure 4.36](#) : Un histogramme personnalisé.

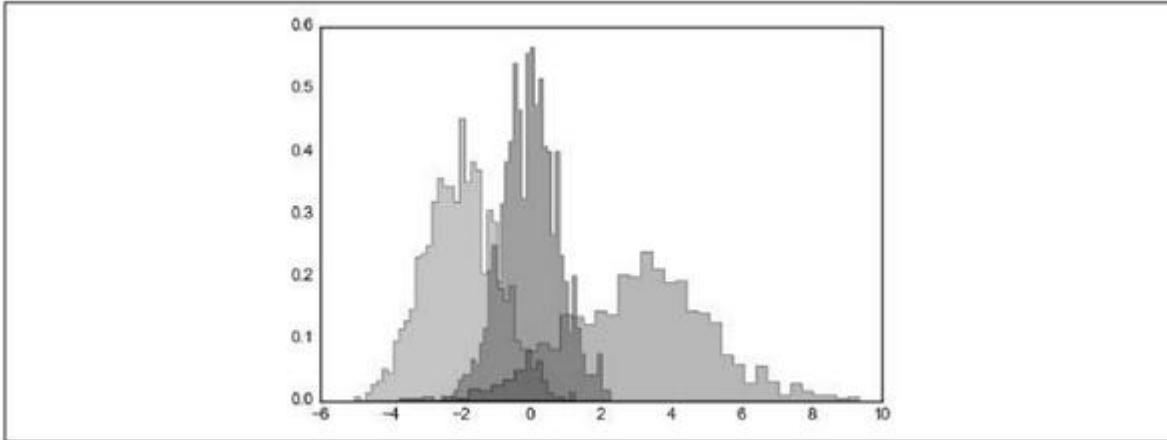
La documentation du code source de plt.hist() donne la liste des autres options. Lorsque j'ai besoin de comparer les histogrammes de plusieurs distributions, j'aime combiner histtype='stepfilled' avec un certain niveau de transparence alpha ([Figure 4.37](#)) :

In[4] :

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3,
density=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



[Figure 4.37](#) : Combinaison de plusieurs histogrammes.

Lorsque vous n'avez besoin que de connaître le nombre de points dans un des bacs d'un histogramme sans l'afficher, vous pouvez vous servir de la fonction np.histogram() :

In[5] :

```
counts, bin_edges = np.histogram(data, bins=5)
```

```
print(counts)
```

```
[ 12 190 468 301 29]
```

Histogrammes en deux dimensions avec bins

Nous venons de voir comment créer un histogramme à une dimension en distribuant la série de valeurs dans plusieurs bacs appelés bins. De même, on peut créer un histogramme à deux dimensions en distribuant les points dans deux bacs.

Nous allons découvrir plusieurs approches possibles, en commençant par définir deux tableaux en x et en y à partir d'une distribution gaussienne multivariée :

In[6]:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov,
10000).T
```

Histogramme en deux dimensions avec plt.hist2d

Une solution simple pour produire un histogramme en deux dimensions correspond à la fonction plt.hist2d() de Matplotlib ([Figure 4.38](#)) :

In[7]:

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('Nombres dans bin')
```

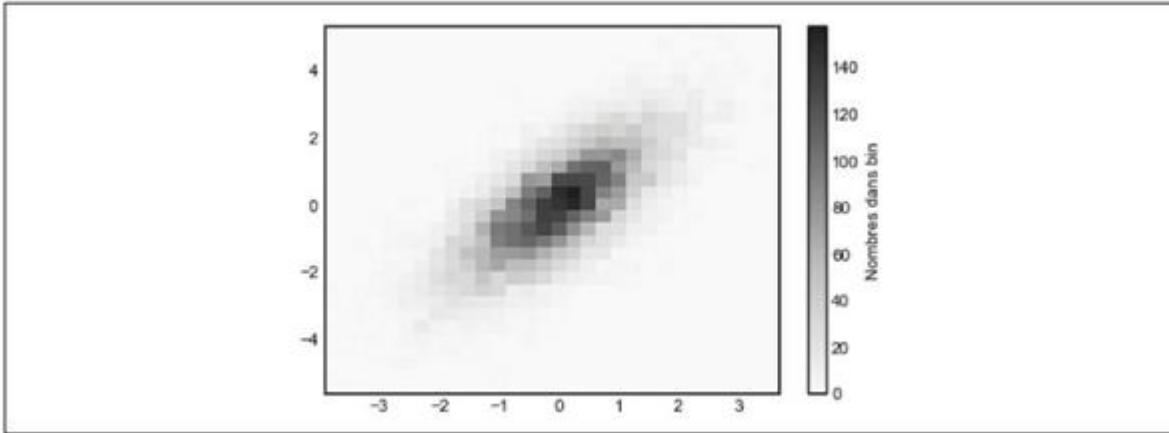


Figure 4.38 : Un histogramme en deux dimensions produit avec plt.hist2d.

Comme sa collègue plt.hist(), la fonction plt.hist2d() propose toute une série d'options décrites dans la chaîne de documentation de la fonction. Vous disposez ici aussi d'une fonction apparentée qui se nomme np.histogram2d(), et qui s'utilise ainsi :

```
In[8]: counts, xedges, yedges =  
np.histogram2d(x, y, bins=30)
```

Lorsque vous avez besoin d'afficher un plus grand nombre de dimensions, vous vous tournez vers la fonction np.histogramdd().

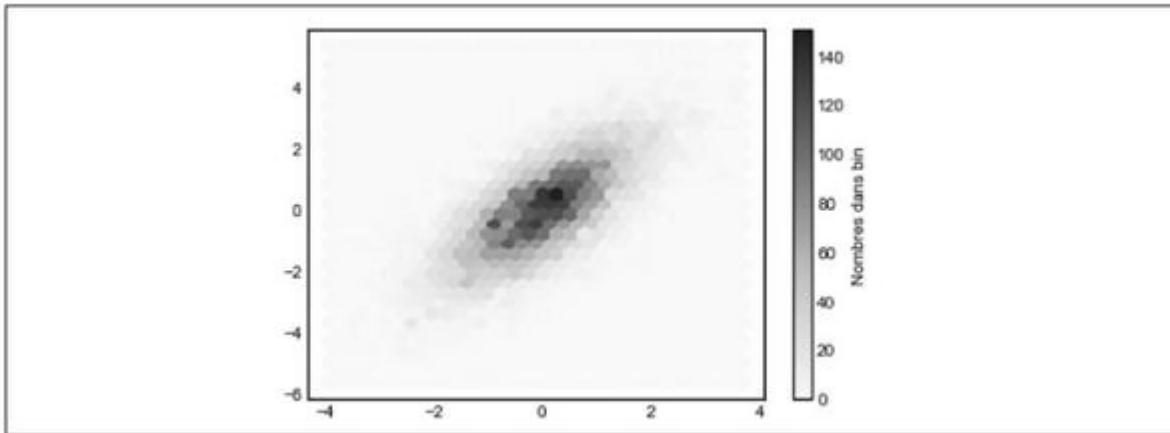
Distribution en bins hexagonaux avec plt.hexbin

Notre histogramme à deux dimensions produit un carrelage de carrés selon les axes. Un autre motif peut être utilisé :

l'hexagone. Il suffit d'utiliser la routine plt.hexbin() qui distribue un jeu de données en deux dimensions dans un réseau d'hexagones ([Figure 4.39](#)) :

In[9] :

```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='Nombres dans bin')
```



[Figure 4.39](#) : Un histogramme à deux dimensions avec plt.hexbin.

Parmi les nombreuses options de plt.hexbin(), citons la possibilité de contrôler le poids relatif de chaque point et le choix des éléments dans chaque bac pour y montrer l'un des agrégats de NumPy (la moyenne des poids, l'écart-type des poids, etc.).

Estimation de densité de noyau KDE

Une autre technique permettant de juger des densités relatives dans plusieurs dimensions correspond à

l'estimation de densité de noyaux KDE (*kernel density estimation*). Nous y reviendrons en détail dans la section correspondante du [Chapitre 5](#). Contentons-nous pour l'instant de découvrir que KDE permet de distribuer les points dans l'espace puis de leur appliquer un traitement pour obtenir un lissage. Le paquetage `scipy.stats` offre une implémentation simple et efficace de KDE. Voici comment utiliser cette méthode d'estimation avec ses données ([voir Figure 4.40](#)) :

```
In[10]:
```

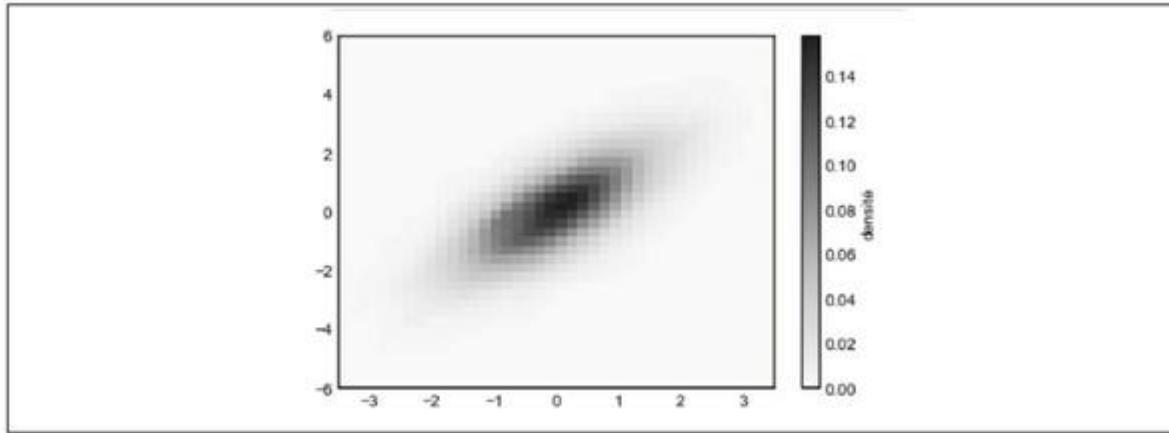
```
from scipy.stats import gaussian_kde

# adapte un tableau de taille [Ndim, Nsamples]
data = np.vstack([x, y])
kde = gaussian_kde(data)

# Évalue sur matrice régulière
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(),
Ygrid.ravel()]))

# Tracé du résultat
plt.imshow(Z.reshape(Xgrid.shape),
origin='lower', aspect='auto',
extent=[-3.5, 3.5, -6, 6],
cmap='Blues')
```

```
cb = plt.colorbar()  
cb.set_label("density")
```



[Figure 4.40](#) : Représentation KDE d'une distribution.

KDE gère une longueur de lissage qui permet de régler finement le choix entre niveau de détails et lissage (c'est un exemple de l'arbitrage habituel entre biais et variance). Le bon choix de la longueur de lissage a fait l'objet de nombreuses études : la fonction `gaussian_kde()` applique des règles en vue de trouver une longueur de lissage optimale en fonction des données d'entrée.

L'écosystème SciPy dispose d'autres formulations de KDE, et chacune a ses forces et ses faiblesses. Voyez notamment `sklearn.neighbors.KernelDensity` et `statsmodels.nonparametric.`

`kernel_density.KDEMultivariate`. Pour des visualisations KDE, Matplotlib demande d'écrire relativement beaucoup d'instructions. L'API de Seaborn est beaucoup plus directe,

comme nous le verrons dans la section qui lui est consacrée en fin de chapitre.

4.8 : Personnalisation des légendes de tracé

Les légendes sont indispensables pour donner du sens à un graphique, en donnant des noms aux différents éléments. Nous avons déjà vu comment créer une légende simple. Voyons comment personnaliser l'aspect et la position des légendes avec Matplotlib.

Pour créer une légende simple, nous utilisons plt.legend() qui ajoute automatiquement une légende pour tout élément qui en dispose ([Figure 4.41](#)) :

In[1] :

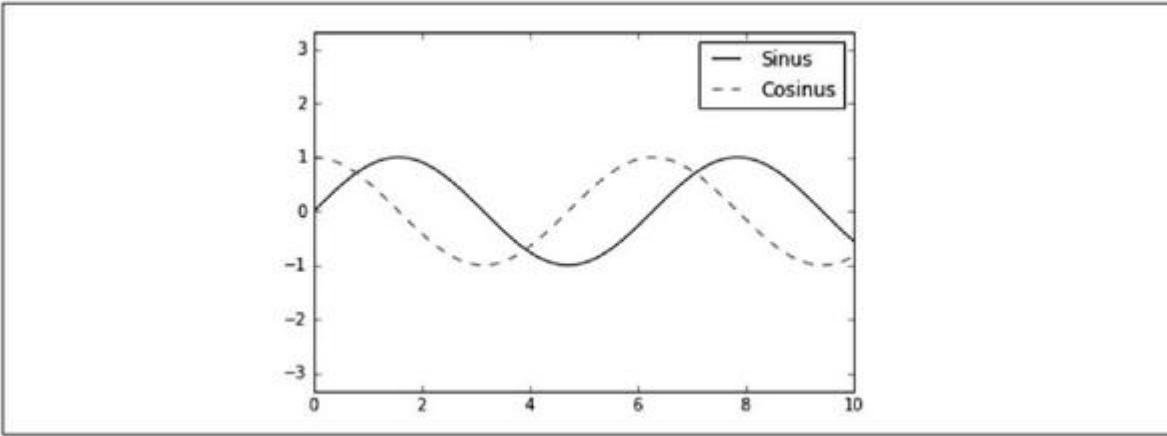
```
import matplotlib.pyplot as plt
plt.style.use('classic')
```

In[2] :

```
%matplotlib inline
import numpy as np
```

In[3] :

```
x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()
ax.plot(x, np.sin(x), '-b', label='Sinus')
ax.plot(x, np.cos(x), '--r', label='Cosinus')
ax.axis('equal')
leg = ax.legend();
```

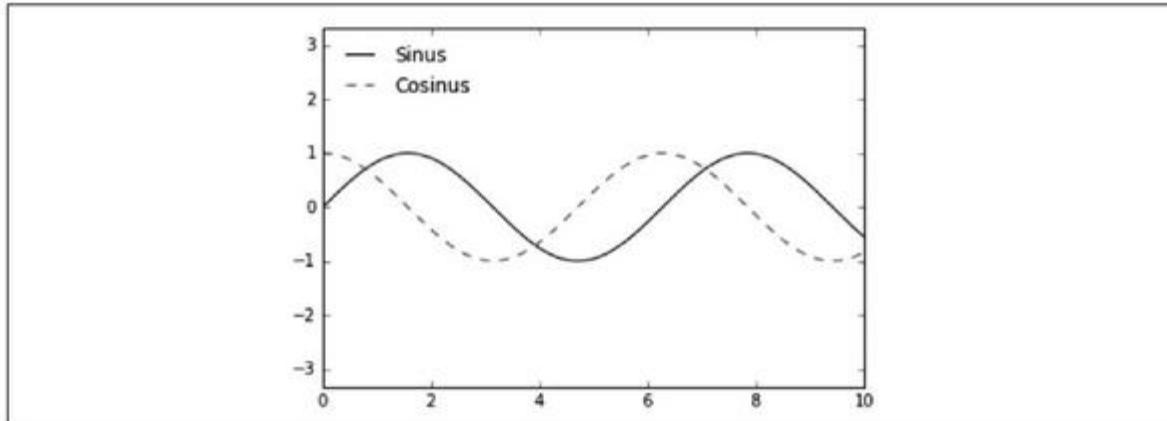


[Figure 4.41](#) : Cartouche de légendes positionné par défaut.

Le cartouche peut être repositionné et affiché sans cadre ([Figure 4.42](#)) :

In[4]:

```
ax.legend(loc='upper left', frameon=False)
fig
```

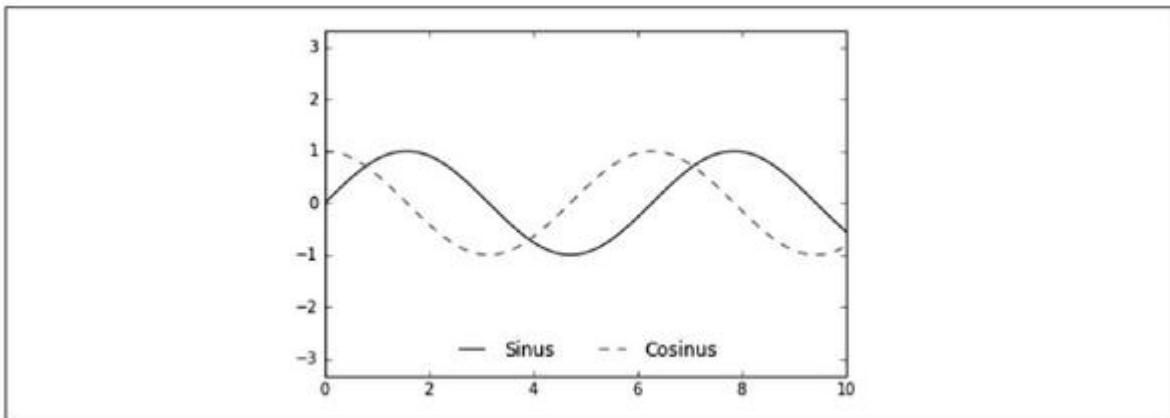


[Figure 4.42](#) : Cartouche de légendes personnalisé.

Le paramètre ncol permet de changer le nombre de colonnes dans le cartouche ([Figure 4.43](#)) :

```
In[5]:
```

```
ax.legend(frameon=False, loc='lower center',
ncol=2)
fig
```

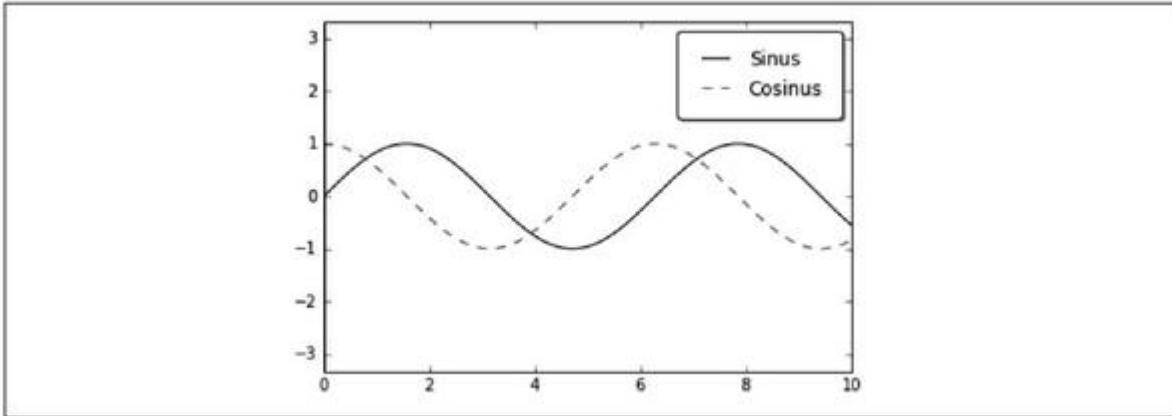


[Figure 4.43](#) : Cartouche sur deux colonnes sans cadre.

Nous pouvons aussi choisir d'arrondir les coins du cadre ou d'ajouter une ombre, de régler la transparence alpha ou de contrôler l'espacement autour ([Figure 4.44](#)) :

```
In[6]:
```

```
ax.legend(fancybox=True, framealpha=1,
shadow=True, borderpad=1)
fig
```



[Figure 4.44](#) : Cartouche arrondi grâce au paramètre `fancybox`.

Voyez la documentation du code source de `plt.legend()` pour en savoir plus.

Choix des éléments à légendier

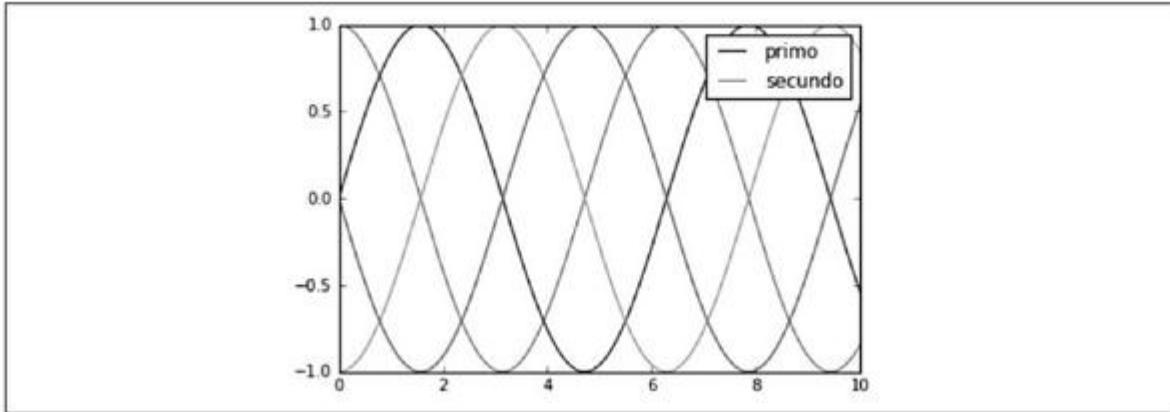
Par défaut, tous les éléments portant un label sont légendés. Lorsque ce n'est pas ce que vous voulez, il suffit de décider quels éléments et labels doivent apparaître en vous servant des objets renvoyés par les commandes de tracé. La commande `plt.plot()` permet de créer plusieurs lignes à la fois, et renvoie une liste des instances de lignes qui ont été créées. Il suffit de transmettre ces éléments à `plt.legend()` pour lui demander de les identifier, en indiquant éventuellement les labels à montrer ([Figure 4.45](#)) :

In[7]:

```
y = np.sin(x[:, np.newaxis] + np.pi *  
np.arange(0, 2, 0.5))
```

```
lines = plt.plot(x, y)

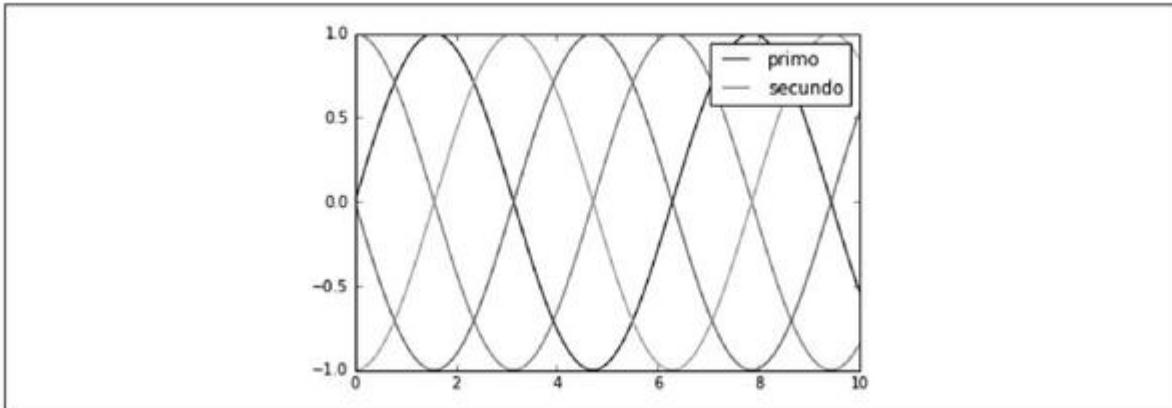
# lines est une liste d'instances de plt.Line2D
plt.legend(lines[:2], ['primo', 'secundo']);
```



[Figure 4.45](#) : Personnalisation du cartouche de légendes.

Personnellement, je préfère utiliser la première méthode en choisissant les labels en paramètre des commandes de tracé ([Figure 4.46](#)) :

```
In[8] :
plt.plot(x, y[:, 0], label='primo')
plt.plot(x, y[:, 1], label='secundo')
plt.plot(x, y[:, 2:])
plt.legend(framealpha=1, frameon=True);
```



[Figure 4.46](#) : Autres méthodes de personnalisation d'un cartouche.

Notez que par défaut la fonction de légendage ignore tous les éléments dont l'attribut label n'est pas renseigné.

Légende des tailles de points

Parfois, les légendes par défaut ne suffisent pas. Lorsque vous exploitez les diamètres des points pour représenter une variable, vous avez besoin de faire connaître la signification de ces points. Dans l'exemple suivant, nous indiquons ainsi la population de plusieurs villes en Californie. La légende doit indiquer le niveau de population en fonction des diamètres. Pour y parvenir, nous traçons des données avec labels mais sans entrées ([Figure 4.47](#)) :

In[9] :

```
import pandas as pd
cities =
pd.read_csv('data/california_cities.csv')
```

```
# Extraction des données désirées
lat, lon = cities['latd'], cities['longd']
population, area = cities['population_total'],
cities['area_total_km2']

# Nuage de points avec diamètres et couleurs,
# mais sans labels
plt.scatter(lon, lat, label=None,
c=np.log10(population), cmap='viridis',
s=area, linewidth=0, alpha=0.5)
plt.axis(aspect='equal')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='log_{10}(population)')
plt.clim(3, 7)

# Création de légende :
# tracé de listes vides de tailles et labels
choisis
for area in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.3, s=area,
                label=str(area) + ' km^2')
plt.legend(scatterpoints=1, frameon=False,
           labelspacing=1, title='City Area')
plt.title('California Cities: Area and
Population');
```

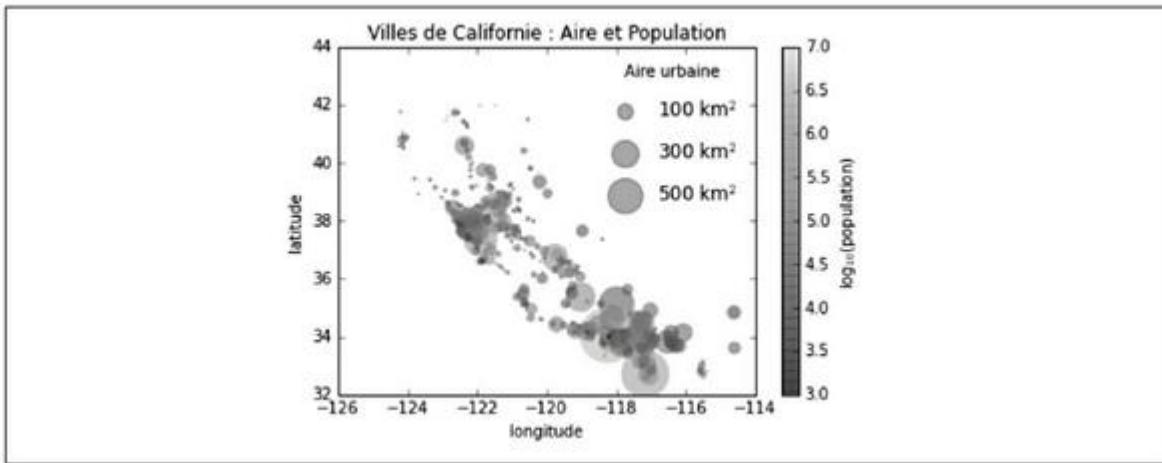


Figure 4.47 : Coordonnées spatiales, surfaces et population des villes de Californie.

Les légendes ne peuvent faire référence qu'à des objets qui sont tracés. Il faut donc tracer tout élément que l'on veut légender. Dans notre exemple, les cercles gris ne font pas partie du tracé, et c'est pourquoi nous les simulons par des listes vides. Vous remarquez ici aussi que seuls les éléments qui possèdent un label sont légendés.

Grâce à cette astuce de listes vides, nous obtenons des objets de tracé que la fonction de légendage utilise pour transmettre une information utile. Cette technique permet de créer des visualisations très sophistiquées.

Notez au passage qu'avec de telles données géographiques, il aurait été intéressant de pouvoir afficher les limites d'états et autres détails géographiques. Il existe un très bon outil complémentaire à Matplotlib pour répondre à ce besoin : Basemap. Nous le verrons en fin de chapitre lorsque nous aborderons Basemap dans le contexte géographique.

Cartouches de légendes multiples

Vous aurez parfois besoin de répartir les légendes dans plusieurs cartouches pour vos axes. Ce n'est pas possible directement dans Matplotlib avec l'interface standard legend. Si vous tentez de créer une seconde légende avec plt.legend() ou avec ax.legend(), vous remplacez la première. La solution consiste à créer un objet de légende que l'on appelle un artiste puis à appeler la méthode de bas niveau ax.add_artist() pour ajouter le second artiste au premier ([Figure 4.48](#)) :

```
In[10]:  
fig, ax = plt.subplots()  
  
lines = []  
styles = ['-', '--', '-.', ':']  
x = np.linspace(0, 10, 1000)  
  
for i in range(4):  
    lines += ax.plot(x, np.sin(x - i * np.pi / 2),  
                     styles[i], color='black')  
    ax.axis('equal')  
  
# Choix des lignes et labels du premier  
cartouche  
ax.legend(lines[:2], ['line A', 'line B'],  
         loc='upper right', frameon=False)
```

```

# Création manuelle du second cartouche et ajout
de l'artiste
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line
D'],
loc='lower right', frameon=False)
ax.add_artist(leg);

```

Cet appel à bas niveau permet de découvrir les rouages internes utilisés par Matplotlib pour générer les tracés. Vous pouvez afficher le code source de `ax.legend()` (vous savez qu'il suffit de saisir `ax.legend??`). Vous verrez que la fonction se résume en fait à une série d'appels à l'objet artiste nommé `Legend` qui est stocké dans l'attribut `legend_` puis ajouté dans la figure à tracer.

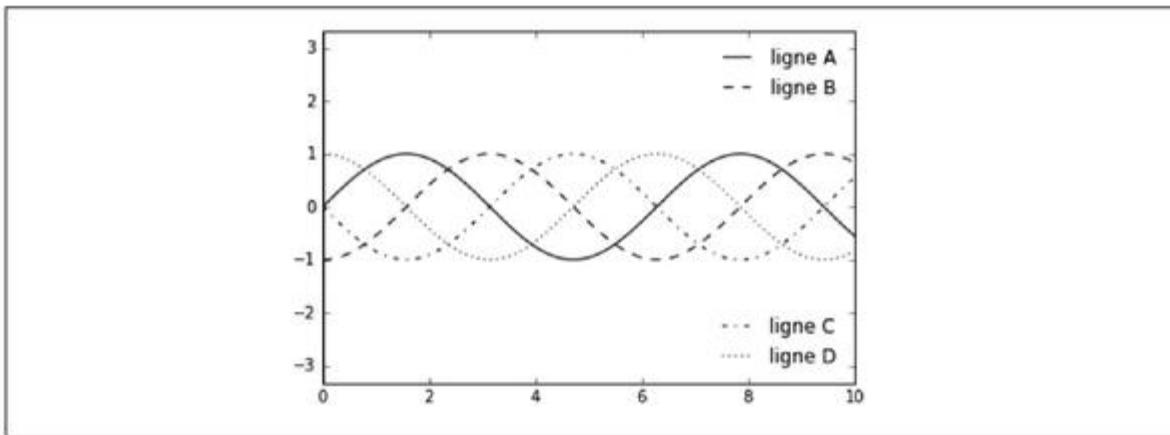


Figure 4.48 : Cartouche en deux parties.

4.9 : Personnalisation des barres de couleurs

Les légendes servent à nommer des valeurs distinctes, mais pour des valeurs continues telles que les couleurs des points, des lignes ou des régions, il faut utiliser une barre de couleurs légendée. Dans Matplotlib, la barre de couleurs est un composant séparé du tracé qui permet de fournir la signification des couleurs utilisées. Ce livre est imprimé en noir et blanc, et c'est pourquoi nous avons prévu une planche couleur au format PDF permettant de visualiser en couleur les différentes couleurs du chapitre (<https://github.com/jakevdp/PythonDataScienceHandbook>).

Commençons par mettre en place les éléments pour pouvoir réaliser les tracés et importer les fonctions dont nous aurons besoin :

In[1]:

```
import matplotlib.pyplot as plt  
plt.style.use('classic')
```

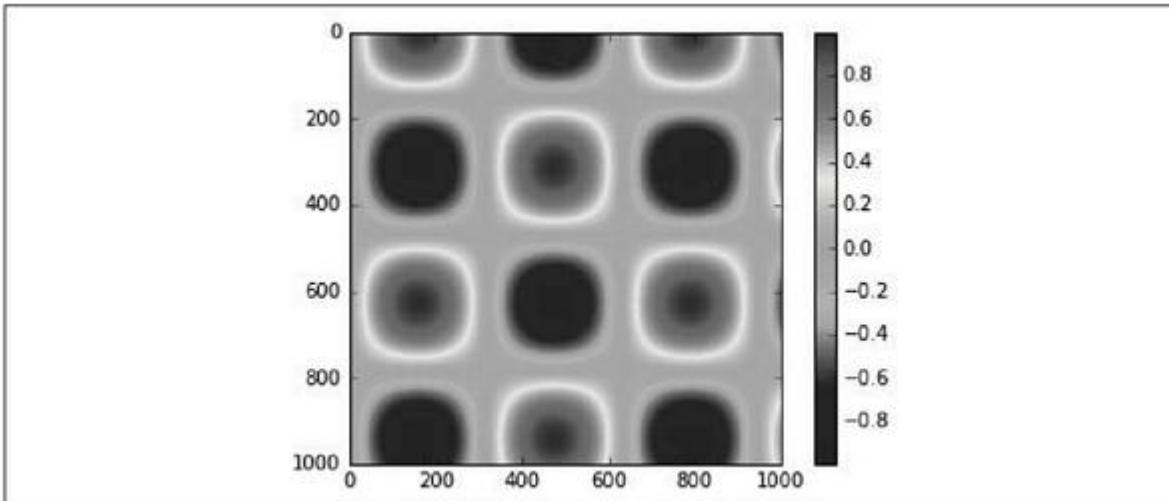
In[2]:

```
%matplotlib inline  
import numpy as np
```

Nous avons déjà utilisé à plusieurs reprises la barre de couleurs standard qui s'affiche par un appel à la fonction plt.colorbar() ([Figure 4.49](#)) :

In[3] :

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
plt.imshow(I)
plt.colorbar();
```



[Figure 4.49](#) : Une barre de couleurs par défaut.

Voyons donc comment personnaliser cette barre de couleurs pour l'adapter à différentes situations.

Style des barres de couleurs

La plage de couleurs correspond au paramètre nommé cmap fourni à la fonction de tracé ([Figure 4.50](#)) :

```
In[4]: plt.imshow(I, cmap='gray');
```

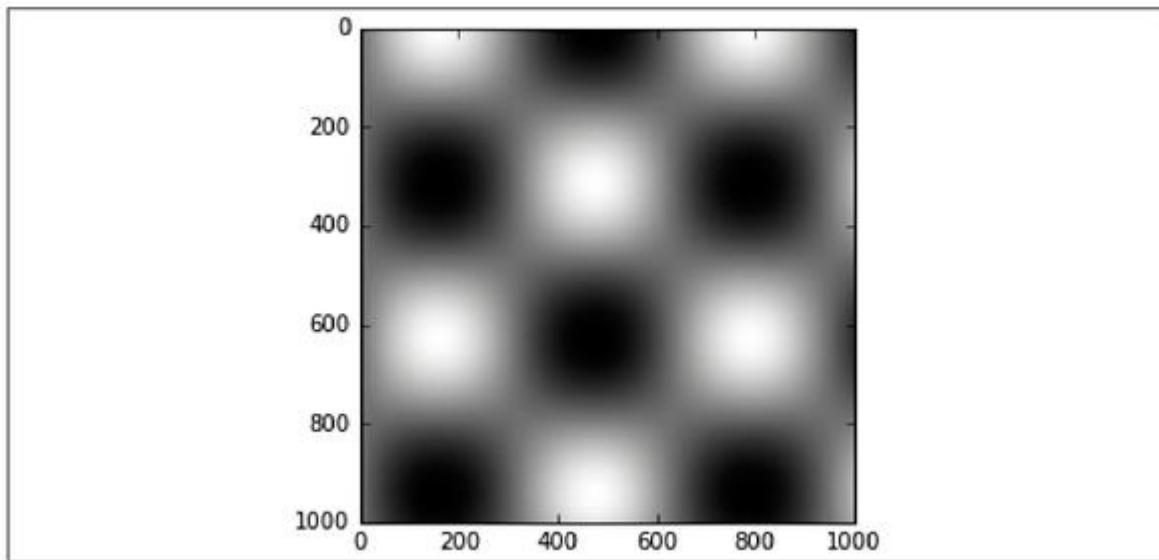


Figure 4.50 : Une plage de couleurs en niveaux de gris.

L'affichage de la barre de couleurs permet donc de connaître la plage de couleurs en vigueur. Toutes les plages de couleurs sont accessibles dans l'espace de noms plt.cm. Vous pouvez dresser la liste des possibilités au moyen de la fonction d'aide d'IPython basée sur la touche tabulation :

```
plt.cm.<TAB>
```

Savoir sélectionner une plage de couleurs est une chose, mais il faut d'abord apprendre à choisir parmi les nombreuses possibilités ! Les critères à prendre en compte sont bien plus subtils que l'on pourrait s'y attendre.

Sélection d'une plage de couleurs

Ce livre ne présentera pas les détails concernant le choix des plages de couleurs. Vous vous reporterez si nécessaire à la documentation de Matplotlib.

Vous disposez de trois catégories de plages de couleurs :

Plage de couleurs séquentielles

Il s'agit d'une séquence de couleurs continues (binary, viridis, etc.).

Plage de couleurs divergentes

En général, elle comporte deux couleurs pour montrer les valeurs positives et négatives par rapport à une moyenne (RdBu, PuOr, etc.).

Plage de couleurs qualitatives

Il s'agit d'une sélection de couleurs sans ordre significatif (rainbow, jet, etc.).

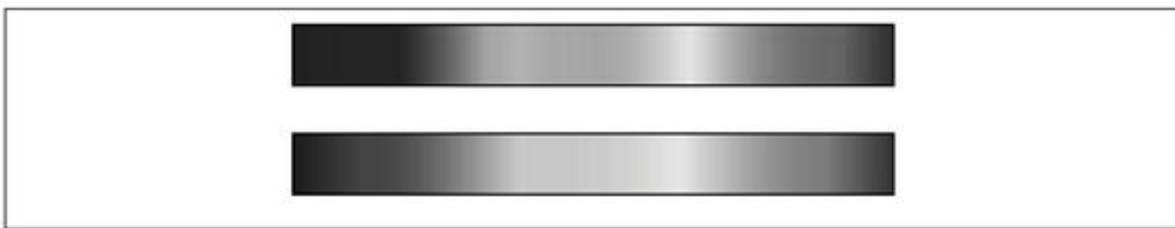
Le nuancier jet était celui en vigueur par défaut dans les versions antérieures à 2.0 de Matplotlib, ce qui n'était pas un choix avisé parce qu'un nuancier qualitatif ne convient pas à des données quantitatives. Il ne permet en effet pas de montrer un dégradé entre deux nuances en fonction de l'évolution progressive des valeurs.

Il suffit de convertir ce nuancier jet en noir et blanc pour le confirmer ([Figure 4.51](#)) :

```
In[5]:  
from matplotlib.colors import  
LinearSegmentedColormap  
  
def grayscale_cmap(cmap):  
    """Renvoie une version grise de la plage de  
couleurs fournie"""  
    cmap = plt.cm.get_cmap(cmap)  
    colors = cmap(np.arange(cmap.N))  
  
    # Conversion RGBA vers luminance gris perçu  
    # voir http://alienryderflex.com/hsp.html  
    RGB_weight = [0.299, 0.587, 0.114]  
    luminance = np.sqrt(np.dot(colors[:, :3] **  
2, RGB_weight))  
    colors[:, :3] = luminance[:, np.newaxis]  
    return  
LinearSegmentedColormap.from_list(cmap.name +  
"_gray", colors, cmap.N)  
def view_colormap(cmap):  
    """Trace une plage de couleurs et  
l'équivalent gris"""  
    cmap = plt.cm.get_cmap(cmap)  
    colors = cmap(np.arange(cmap.N))  
    cmap = grayscale_cmap(cmap)  
    grayscale = cmap(np.arange(cmap.N))  
    fig, ax = plt.subplots(2, figsize=(6, 2),
```

```
subplot_kw=dict(xticks=[], yticks=[]))  
ax[0].imshow([colors], extent=[0, 10, 0, 1])  
ax[1].imshow([grayscale], extent=[0, 10, 0, 1]
```

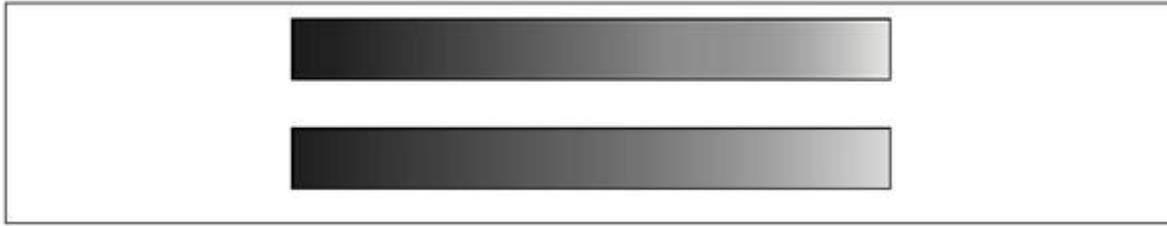
In[6]: `view_colormap('jet')`



[Figure 4.51](#) : Le nuancier jet et son échelle de luminance déséquilibrée.

Vous remarquez les bandes très claires dans la version en niveaux de gris. Même dans la version en couleurs, cette partie très brillante va attirer l'œil vers certaines parties de la palette de couleurs, ce qui fait ressortir inutilement des parties peu importantes du jeu de données. Il est préférable d'adopter un nuancier tel que viridis qui est celui par défaut dorénavant. Il a été conçu pour offrir une variation de luminosité continue sur toute la plage de couleurs. Il s'adapte donc bien à la perception humaine des couleurs tout en se convertissant correctement pour une impression en niveaux de gris ([Figure 4.52](#)) :

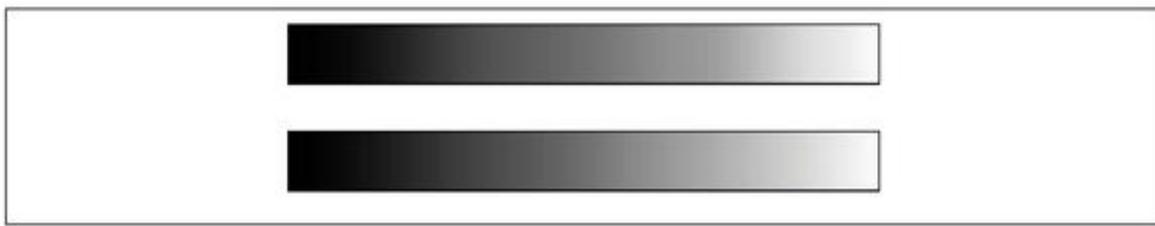
In[7]: `view_colormap('viridis')`



[Figure 4.52](#) : Le nuancier viridis et son échelle de luminances progressives.

Si vous appréciez les nuanciers de types arc-en-ciel, vous pouvez adopter le nuancier cubehelix pour les données continues ([Figure 4.53](#)) :

```
In[8]: view_colormap('cubehelix')
```



[Figure 4.53](#) : Le nuancier cubehelix et sa luminance.

Les barres de couleurs à deux couleurs telles que RdBu (rouge et bleu) conviennent particulièrement à la visualisation des écarts positifs et négatifs par rapport à une moyenne. Sachez cependant, comme le montre la [Figure 4.54](#), que la distinction entre positif et négatif est perdue lors de la conversion vers les niveaux de gris !

```
In[9]: view_colormap('RdBu')
```

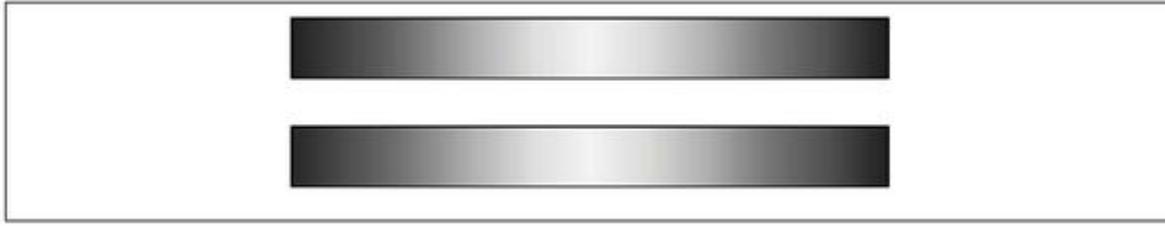


Figure 4.54 : Deux barres de couleurs rouge-bleu RdBu, en couleurs et en niveaux de gris.

Nous utiliserons ce style de nuancier dans la suite du livre.

Matplotlib offre un grand choix de plages de couleurs dont vous pouvez vous faire une idée en accédant aux détails du sous-module plt.cm depuis IPython. Pour une description plus générale de la gestion des couleurs dans Python, voyez la documentation et les outils de la librairie Seaborn que nous abordons à la fin de ce chapitre.

LIMITES ET EXTENSIONS DE COULEURS

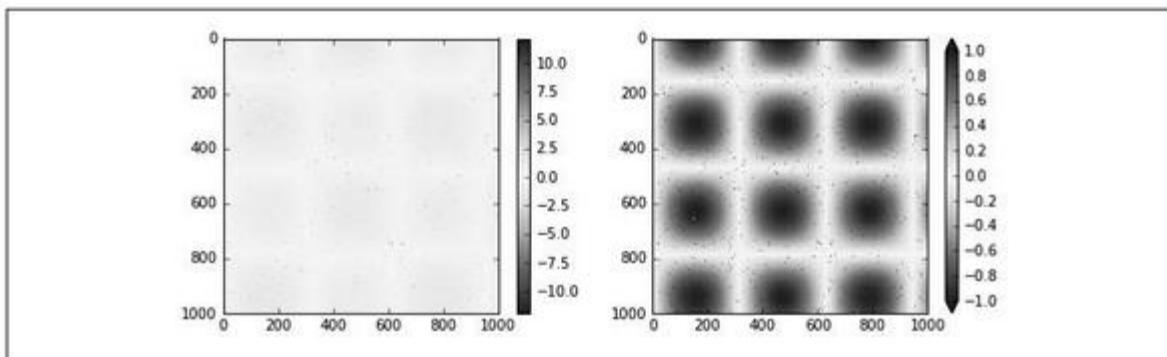
Vous pouvez aisément personnaliser les barres de couleurs Matplotlib. En effet, l'objet barre de couleurs est une instance de plt.Axes. De ce fait, tout le paramétrage des axes et des graduations reste applicable. Les barres de couleurs offrent des options intéressantes ; vous pouvez par exemple limiter la palette de couleurs et choisir d'afficher les valeurs hors limites en terminant la barre par une flèche en haut et en bas, au moyen de la propriété extend. Cette possibilité est

très pratique par exemple lorsque vous devez afficher une image contenant beaucoup de bruit ([Figure 4.55](#)) :

```
In[10]: # Génère du bruit dans 1 % des pixels
speckles = (np.random.random(I.shape) < 0.01)
I[speckles] = np.random.normal(0, 3,
np.count_nonzero(speckles))
plt.figure(figsize=(10, 3.5))

plt.subplot(1, 2, 1)
plt.imshow(I, cmap='RdBu')
plt.colorbar()

plt.subplot(1, 2, 2)
plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1);
```



[Figure 4.55](#) : Contrôle des extensions et limites d'une barre de couleurs.

Vous remarquez que dans le volet gauche, les pixels de bruit sont pris en compte dans les limites de couleurs : le résultat

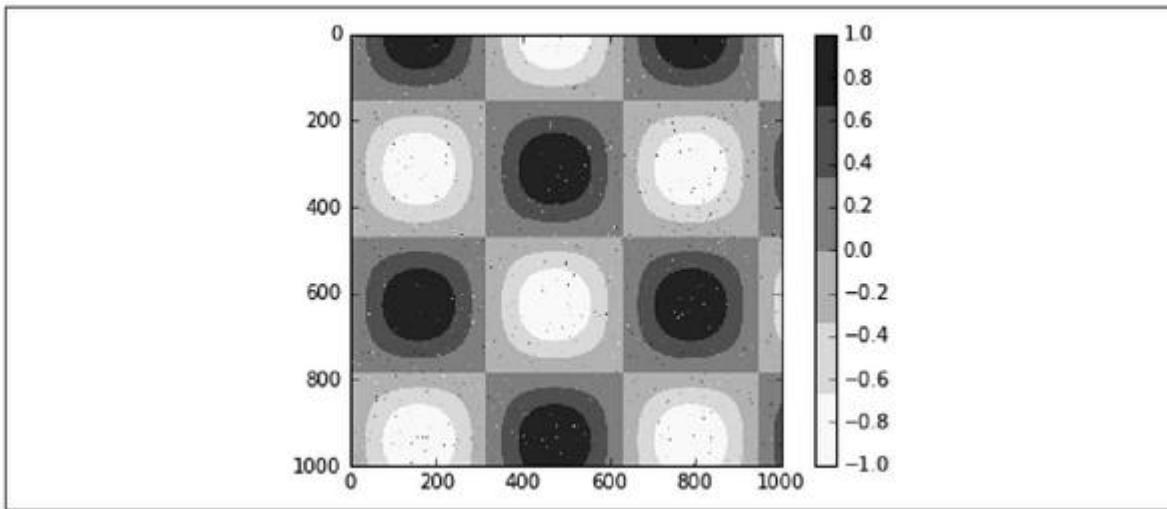
est que le motif que nous cherchions à voir est totalement délavé. Dans le volet droit, nous avons contrôlé les limites de couleurs et ajouté les extensions pour spécifier les valeurs qui sont hors limites par le haut ou par le bas. Le résultat est une visualisation bien plus exploitable malgré le bruit.

Barres de couleurs discontinues

Par défaut, les nuanciers sont continus, mais vous aurez parfois besoin d'utiliser des valeurs discontinues. Vous pouvez vous servir de la fonction plt.cm.get_cmap() en fournissant le nom d'un nuancier ainsi que le nombre de bacs de répartition désiré ([Figure 4.56](#)) :

```
In[11]:  
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))  
plt.colorbar()  
plt.clim(-1, 1);
```

Les conditions d'utilisation d'un tel nuancier restent les mêmes que pour un nuancier classique.



[Figure 4.56](#) : Utilisation d'un nuancier discontinu.

Exemple : chiffres manuscrits

Découvrons l'utilité des nuanciers de barres de couleurs par la visualisation d'un lot de chiffres manuscrits. Ce jeu de données est intégré au paquetage Scikit-Learn. Il est constitué d'environ 2 000 vignettes graphiques de 8×8 représentant des chiffres écrits à la main.

Nous devons d'abord télécharger les données puis visualiser quelques images d'exemples avec plt.imshow() ([Figure 4.57](#)) :

```
In[12]: # Chargement images de 0 à 5 et
         visualisation partielle
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)
```

```
fig, ax = plt.subplots(8, 8, figsize=(6, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
    axi.set(xticks=[], yticks=[])

```

Chaque cellule de chiffres correspond à 64 niveaux de luminance des pixels constitutifs. On peut donc considérer que chaque chiffre est un point dans un espace à 64 dimensions, chaque dimension correspondant à la luminosité d'un pixel. Il n'est évidemment pas simple du tout de visualiser les relations avec un tel nombre de dimensions. Une solution consiste à exploiter la technique de réduction de dimensions, comme par exemple l'apprentissage par manifold. On réduit ainsi le nombre de dimensions tout en conservant les relations qui nous intéressent. Cette technique est une technique d'apprentissage machine non supervisée (nous y reviendrons dans la section sur l'apprentissage manifold du [Chapitre 5](#)).

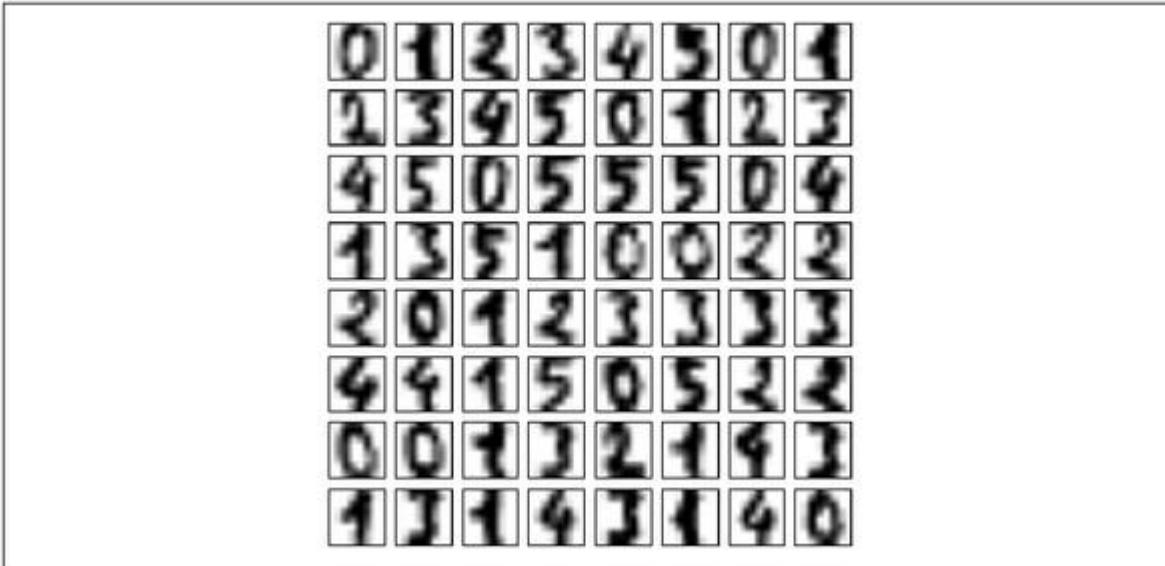


Figure 4.57 : Échantillon des données de chiffres manuscrits.

Sans entrer dans les détails, voyons immédiatement quel aspect prend une projection par apprentissage manifold sur deux dimensions de ces données de chiffres manuscrits :

```
In[13]: # Projection des chiffres en 2D avec  
IsoMap  
from sklearn.manifold import Isomap  
iso = Isomap(n_components=2)  
projection = iso.fit_transform(digits.data)
```

Nous utilisons un nuancier discontinu et réglons le paramètre ticks pour améliorer l'aspect de la barre de couleurs ([Figure 4.58](#)) :

```
In[14]:  
plt.scatter(projection[:, 0], projection[:, 1],  
lw=0.1, c=digits.target,
```

```

cmap=plt.cm.get_cmap('cubehelix', 6))
plt.colorbar(ticks=range(6), label='Valeur de
chiffre')
plt.clim(-0.5, 5.5)

```

Cette projection nous fournit des détails intéressants quant aux relations entre les données du jeu : nous constatons que les plages des chiffres 5 et 3 se chevauchent quasiment, ce qui confirme que ces deux chiffres seront difficiles à distinguer, et donc souvent confondus par un algorithme de classification automatique. D'autres chiffres sont plus écartés, par exemple 0 et 1, et seront donc plus aisément distingués. Il suffit de voir la figure précédente pour convenir que les 5 et les 3 se ressemblent beaucoup plus que les 0 et les 1.

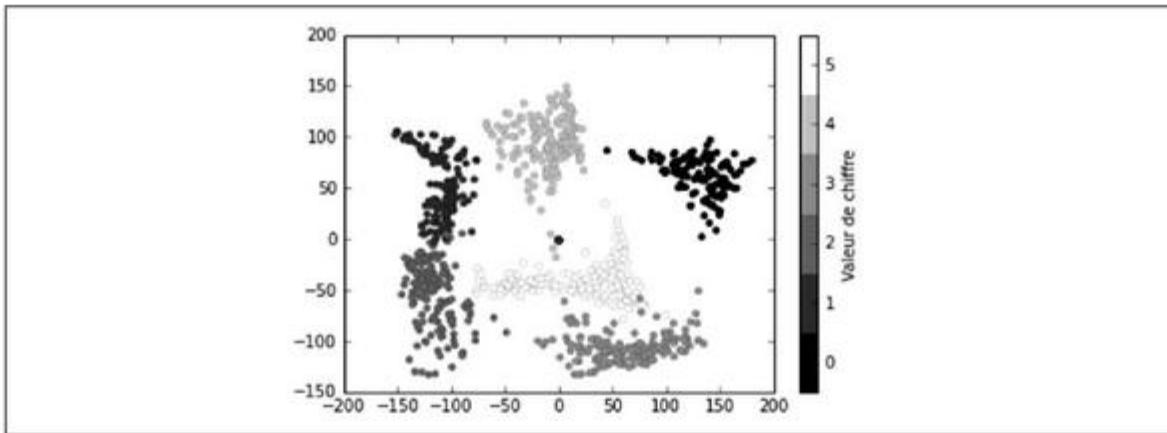


Figure 4.58 : Traitement de type manifold des pixels de chiffres manuscrits.

4.10 : Sous-tracés multiples

Il est parfois utile de pouvoir visualiser les mêmes données sous plusieurs angles. Matplotlib gère le principe de sous-tracés ou *subplots* qui consiste à combiner plusieurs tracés dans la même figure. Chaque sous-tracé peut être un élément inséré, un repère orthonormé ou même une configuration plus complexe. Nous allons découvrir quatre routines de création de sous-tracés avec Matplotlib. Commençons par préparer le calepin et importer les fonctions dont nous aurons besoin :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
plt.style.use('seaborn-white')  
import numpy as np
```

Sous-tracé manuel avec plt.axes

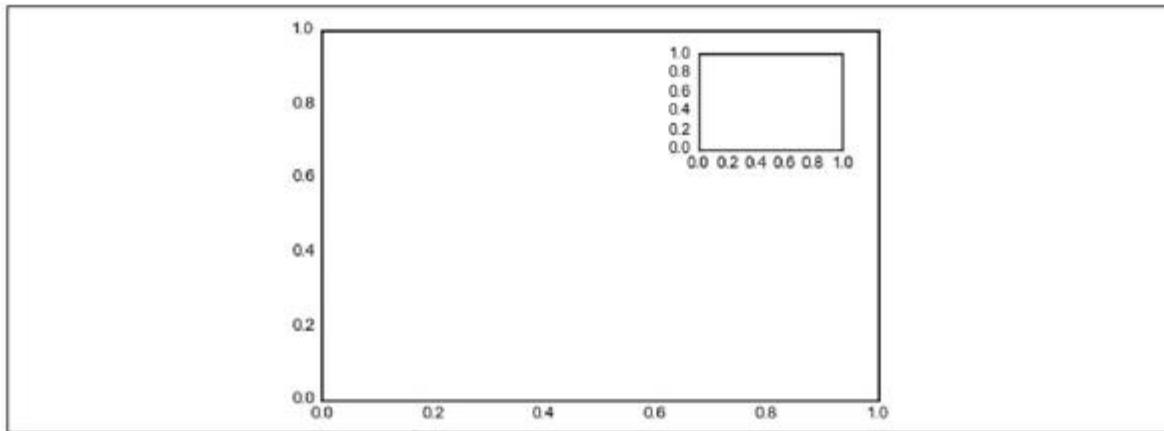
La méthode élémentaire pour créer un tracé correspond à la fonction `plt.axes()`. On obtient ainsi un objet standard qui remplit totalement l'espace de la figure. Un paramètre facultatif permet de spécifier quatre nombres dans les coordonnées de la figure pour incarner respectivement les quatre mesures [bas, gauche, largeur, hauteur]. La

valeur 0 correspond au coin inférieur gauche de la figure et la valeur 1 au coin supérieur droit.

Nous pouvons, grâce à ces paramètres, créer un sous-tracé dans l'angle supérieur droit d'un autre en choisissant pour x et y les valeurs 0.65 qui demandent de commencer à 65 % de la largeur et de la hauteur. Pour les étendues de x et y, nous choisissons 0.2, ce qui signifie que le tracé occupera 20 % de la largeur et de la hauteur de la figure totale. Le résultat du code suivant est montré dans la [Figure 4.59](#) :

In[2] :

```
ax1 = plt.axes() # Axes standard  
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```



[Figure 4.59](#) : Exemple d'insertion d'un sous-tracé.

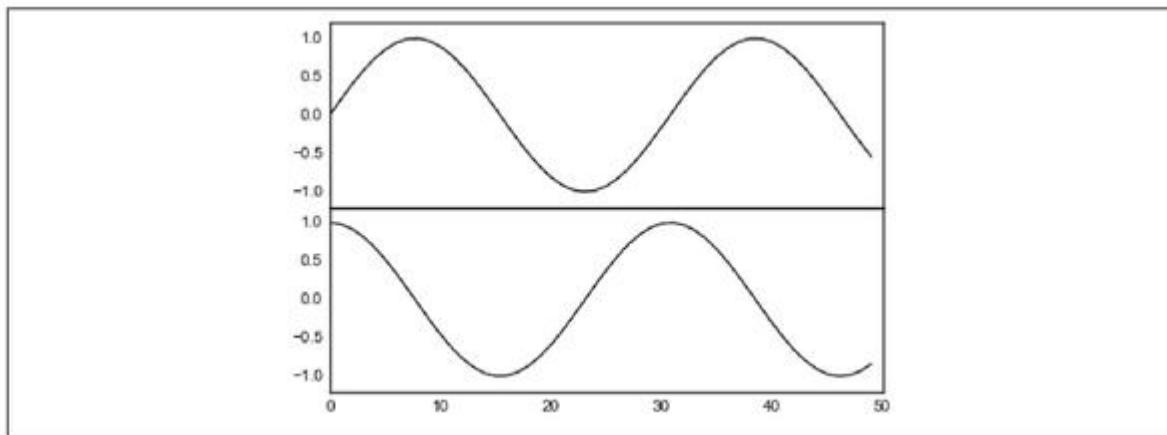
Si l'on utilise l'interface orientée objet, on obtient le même résultat au moyen de la méthode `fig.add_axes()`. Nous nous

en servons pour créer deux sous-tracés empilés verticalement ([Figure 4.60](#)) :

In[3] :

```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
xticklabels=[], ylim=(-1.2, 1.2))
ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4], ylim=
(-1.2, 1.2))

x = np.linspace(0, 10)
ax1.plot(np.sin(x))
ax2.plot(np.cos(x));
```



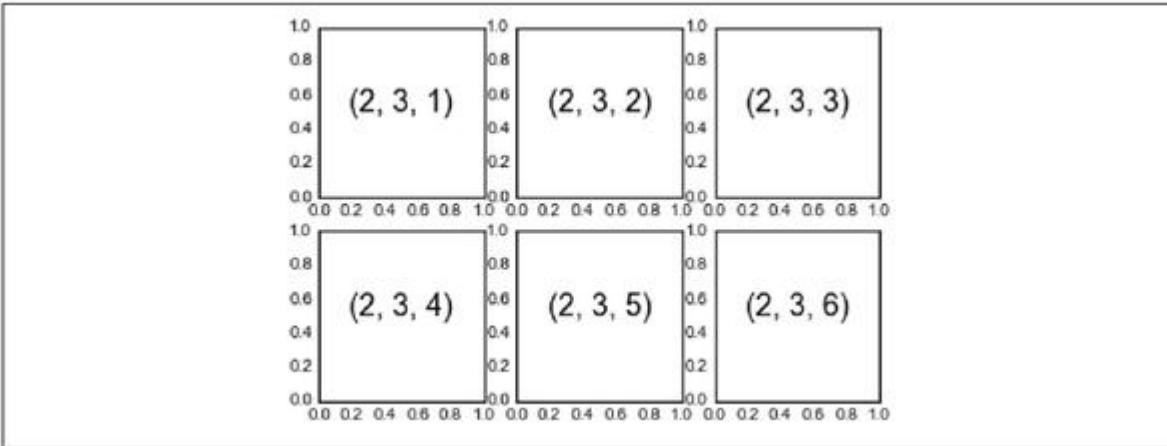
[Figure 4.60](#) : Deux tracés empilés verticalement.

Le tracé du haut n'a pas de graduations inférieures et les deux tracés se touchent. Le bas du volet supérieur à la position 0.5 coïncide avec le haut du volet inférieur qui est en position 0.1 + 0.4.

Une grille de sous-tracés avec plt.subplot()

Afficher des lignes et des colonnes de sous-tracés est un besoin suffisamment fréquent pour que Matplotlib ait prévu plusieurs routines pour y répondre. Celle de plus bas niveau correspond à plt.subplot() qui se contente d'installer un sous-tracé dans un quadrillage. Cette routine travaille avec trois paramètres entiers : le nombre de lignes, le nombre de colonnes et l'index de la cellule devant recevoir le sous-tracé. Les cellules sont comptées en commençant par un, dans le coin supérieur gauche, en allant vers la fin de la ligne, et ainsi jusqu'au coin inférieur droit ([Figure 4.61](#)) :

```
In[4]:  
for i in range(1, 7):  
    plt.subplot(2, 3, i)  
    plt.text(0.5, 0.5, str((2, 3, i)),  
            fontsize=18, ha='center')
```



[Figure 4.61](#) : Exemple de sous-tracés avec `plt.subplot()`.

Vous pouvez modifier l'espacement entre les sous-tracés au moyen de la commande `plt.subplots_adjust`. Dans l'exemple qui suit, nous utilisons l'équivalent orienté objet, `fig.add_subplot()` ([Figure 4.62](#)) :

```
In[5]:  
fig = plt.figure()  
fig.subplots_adjust(hspace=0.4, wspace=0.4)  
for i in range(1, 7):  
    ax = fig.add_subplot(2, 3, i)  
    ax.text(0.5, 0.5, str((2, 3, i)),  
    fontsize=18, ha='center')
```

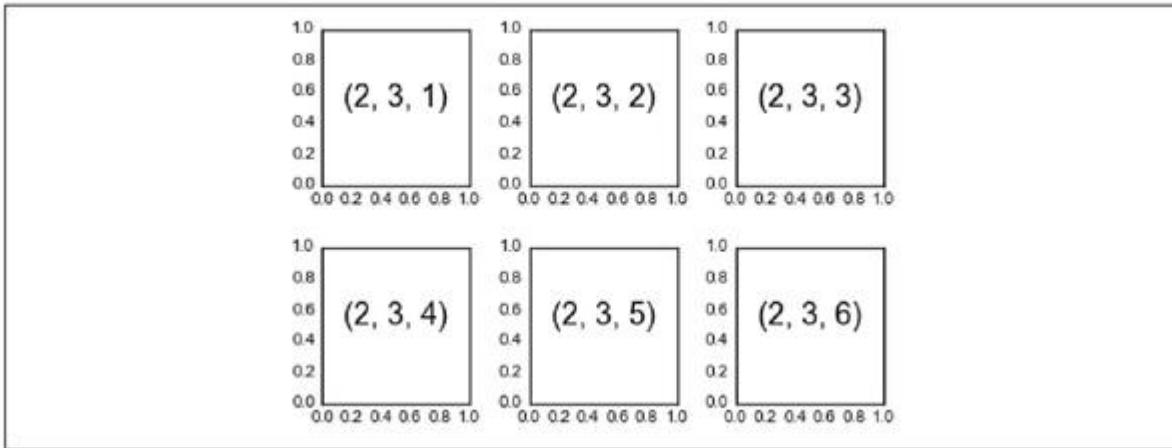


Figure 4.62 : Augmentation des marges entre sous-tracés.

Les paramètres nommés `hspace` et `wspace` de `plt.subplots_adjust` permettent de décider de l'espacement dans les sens de la hauteur et de la largeur. Les valeurs sont exprimées en unités de la taille du sous-tracé. Dans l'exemple, nous spécifions 40 % de la largeur et de la hauteur du sous-tracé.

Tout un ensemble en une fois avec `plt.subplots`

La technique que nous venons de voir pour combiner plusieurs tracés devient malaisé à employer, notamment lorsque vous avez besoin de masquer les légendes d'axes x et y de certains des tracés. Dans ce cas, vous utilisez la fonction nommée `plt.subplots()` (notez bien le **S** qui marque le pluriel). Cette fonction crée en une seule instruction une

grille complète de sous-tracés et renvoie le résultat sous forme d'un tableau NumPy. Vous donnez en paramètre le nombre de lignes et de colonnes avec en option les mots-clés `sharex` et `sharey` pour désigner les lignes ou les colonnes qui doivent partager la même légende d'axe.

Voici par exemple comment créer une grille de deux tracés sur trois avec une seule légende en y pour chaque ligne et une seule légende en x pour chaque colonne ([Figure 4.63](#)) :

```
In[6]: fig, ax = plt.subplots(2, 3,  
sharex='col', sharey='row')
```

Les deux paramètres `sharex` et `sharey` ont permis de supprimer les légendes inutiles pour certains sous-tracés. Le quadrillage résultant est envoyé dans un tableau NumPy, ce qui permet de l'utiliser pour en définir le contenu au moyen de la notation de tableau avec index standard ([Figure 4.64](#)) :

```
In[7]: # axes dans tableau 2D indexé par [lig,  
col]  
for i in range(2):  
    for j in range(3):  
        ax[i, j].text(0.5, 0.5, str((i, j)),  
fontsize=18, ha='center')  
fig
```

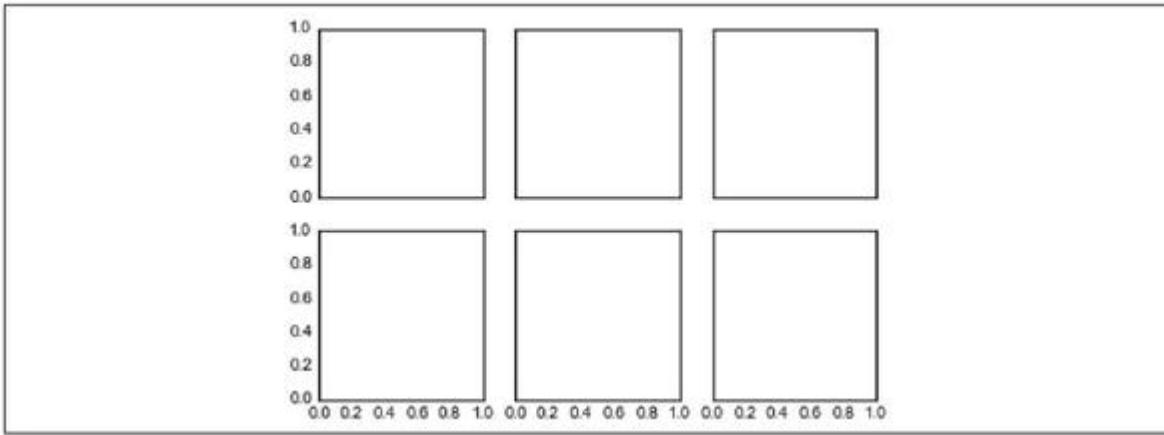


Figure 4.63 : Partage des légendes d'axes avec plt.subplots().

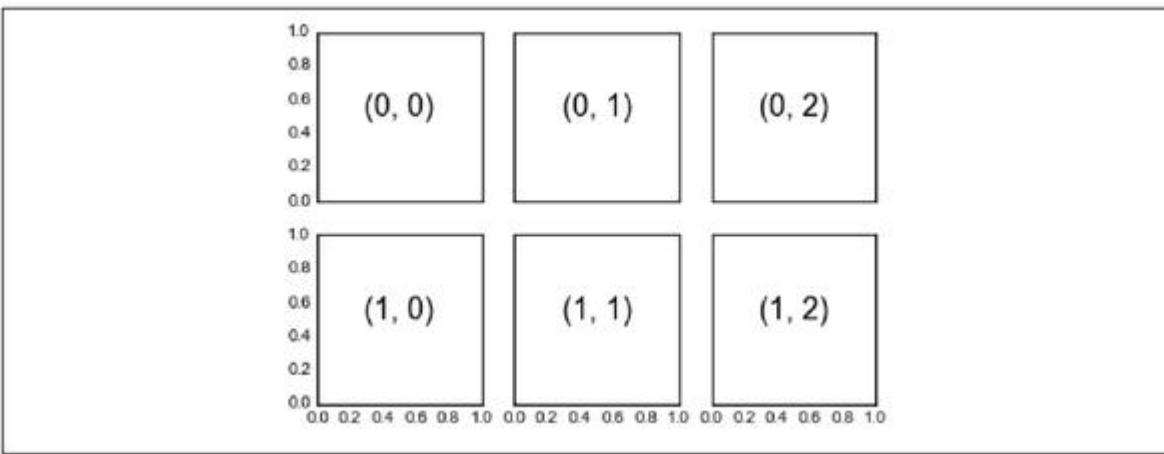


Figure 4.64 : Identification des sous-tracés d'un quadrillage.

Notez au passage que plt.subplots() est cohérent avec la convention Python consistant à commencer la numérotation à zéro, ce qui n'est pas le cas de plt.subplot().

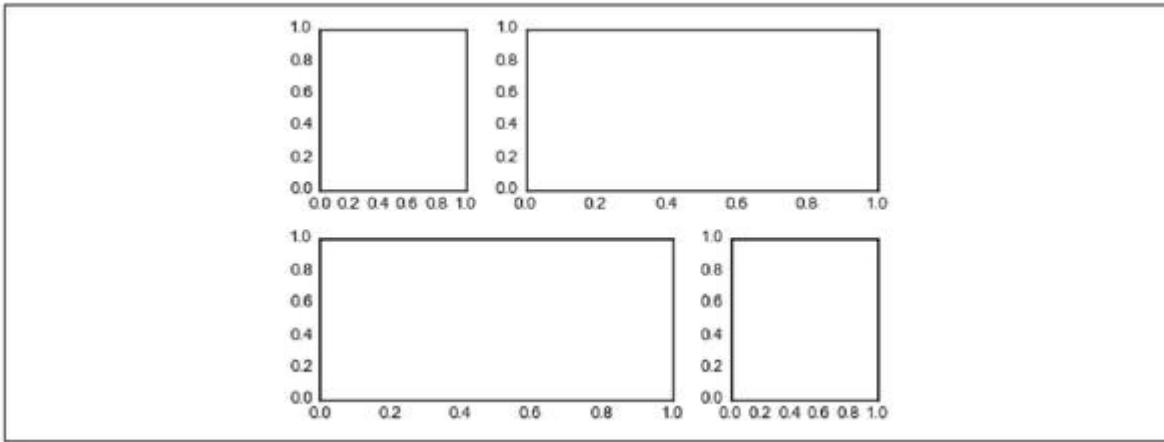
Arrangements sophistiqués avec plt.GridSpec

Pour arranger de façon vraiment personnelle plusieurs sous-tracés, vous utilisez plt.GridSpec(). L'objet correspondant n'incarne pas un tracé. C'est une interface exploitable par la fonction plt. subplot(). Voici par exemple comment spécifier une grille de deux lignes sur trois colonnes en choisissant l'espace en largeur et en hauteur :

```
In[8]: grid = plt.GridSpec(2, 3, wspace=0.4,  
                         hspace=0.3)
```

Nous pouvons ensuite définir les positions et les étendues des sous-tracés au moyen de la syntaxe de tranchage Python classique ([Figure 4.65](#)) :

```
In[9]:  
plt.subplot(grid[0, 0])  
plt.subplot(grid[0, 1:])  
plt.subplot(grid[1, :2])  
plt.subplot(grid[1, 2]);
```



[Figure 4.65](#) : Un arrangement irrégulier de sous-tracés avec `plt.GridSpec`.

Cette véritable mise en page de plusieurs tracés trouve de nombreux domaines d'emploi. Je m'en sers souvent pour créer des histogrammes multiples tels que celui de la

[Figure 4.66](#) :

```
In[10]: # Génère des données en distribution normale
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov,
3000).T

# Configure les axes avec GridSpec
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2,
wspace=0.2)
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0],
xticklabels=[], sharey=main_ax)
```

```
x_hist = fig.add_subplot(grid[-1, 1:],  
                         yticklabels=[], sharex=main_ax)  
  
# Nuages de points sur axes principaux  
main_ax.plot(x, y, 'ok', markersize=3,  
alpha=0.2)  
  
# Histogramme sur axes associés  
x_hist.hist(x, 40, histtype='stepfilled',  
            orientation='vertical',  
            color='gray')  
x_hist.invert_yaxis()  
y_hist.hist(y, 40, histtype='stepfilled',  
            orientation='horizontal',  
            color='gray')  
y_hist.invert_xaxis()
```

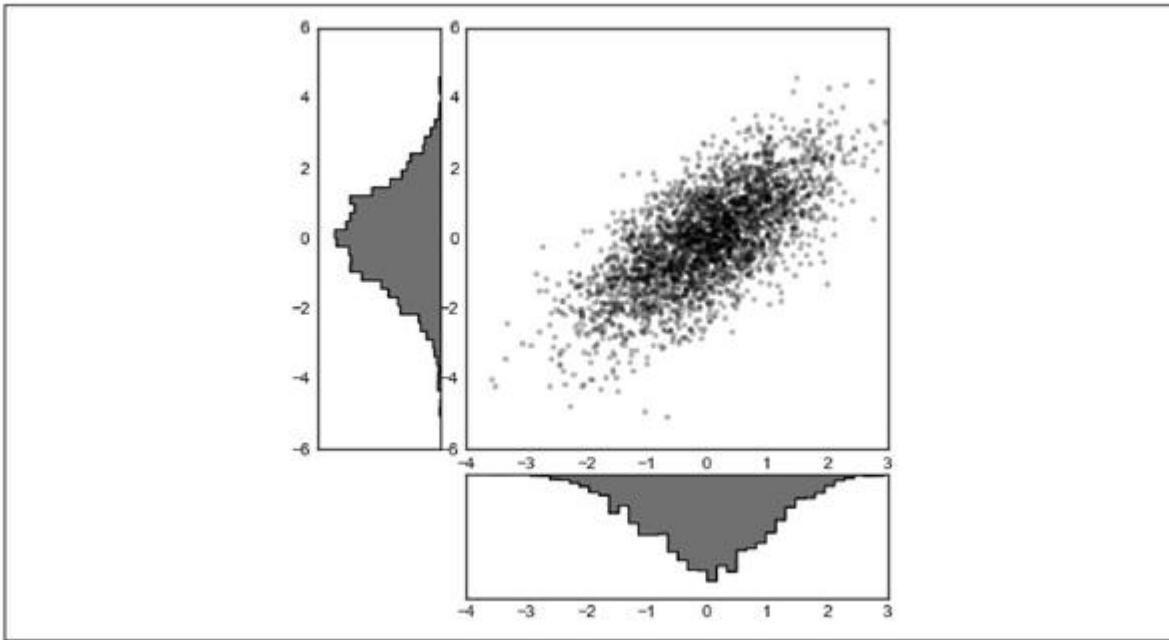


Figure 4.66 : Une distribution à plusieurs dimensions avec `plt.GridSpec`.

Ce genre de visualisation de la distribution le long des deux axes est suffisamment demandé pour que le paquetage Seaborn lui consacre une interface API dédiée. Nous verrons cela dans la dernière section de ce chapitre consacrée à la visualisation avec Seaborn.

4.11 : Texte et annotations

Pour qu'une visualisation de données soit efficace, elle doit guider le lecteur pour qu'il en retire toute la signification. Parfois, les éléments graphiques suffisent, sans ajouter de texte, mais il est souvent nécessaire d'ajouter des légendes et des messages. Les légendes d'axes et les titres ainsi que les cartouches sont déjà des annotations, mais d'autres éléments peuvent être ajoutés. Voyons comment enrichir nos graphiques pour qu'ils transmettent des informations utiles. Nous commençons par mettre en place notre calepin pour effectuer des tracés et importons les fonctions dont nous aurons besoin :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
import matplotlib as mpl  
plt.style.use('seaborn-whitegrid')  
import numpy as np  
import pandas as pd
```

Un exemple : naissances aux USA et congés

Nous allons reprendre les données déjà vues dans le [Chapitre 3](#) quand nous avons abordé les tableaux croisés dynamiques. Ces données permettent de connaître le nombre de naissances par jour dans une année aux USA. Rappelons que le fichier peut être téléchargé à l'adresse <https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv>.

Il est cependant fourni dans le fichier archive des exemples.

Nous commençons par quelques préparatifs puis traçons les résultats ([Figure 4.67](#)) :

```
In[2]:  
births = pd.read_csv('births.csv')  
  
quartiles = np.percentile(births['births'], [25, 50, 75])  
mu, sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])  
births = births.query('(births > @mu - 5 * @sig)  
& (births < @mu + 5 * @sig)')  
  
births['day'] = births['day'].astype(int)  
births.index = pd.to_datetime(10000 *  
births.year + 100 * births.month +
```

```

births.day,
format='%Y%m%d' )
births_by_date = births.pivot_table('births',
[births.index.month, births.index.day])
births_by_date.index = [pd.datetime(2012, month,
day)
for (month, day) in
births_by_date.index]
In[3]:
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax);

```

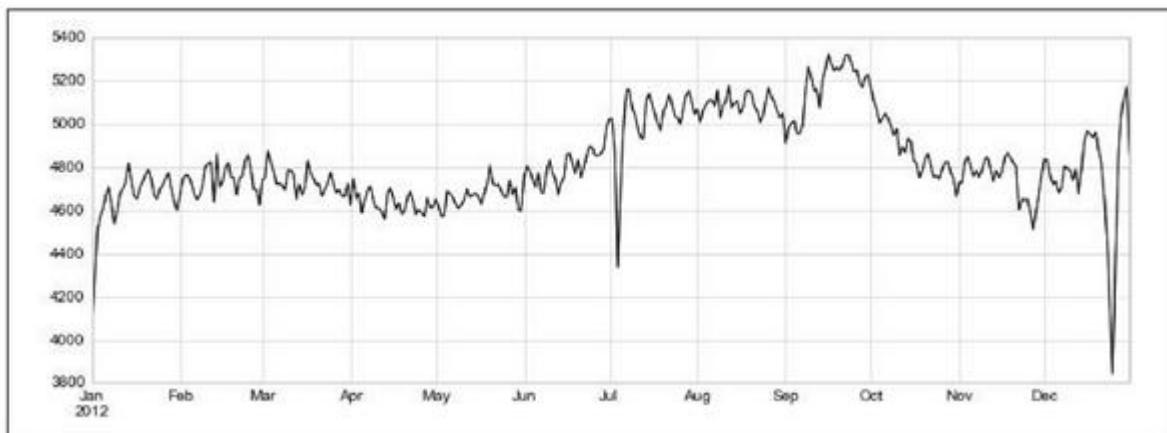
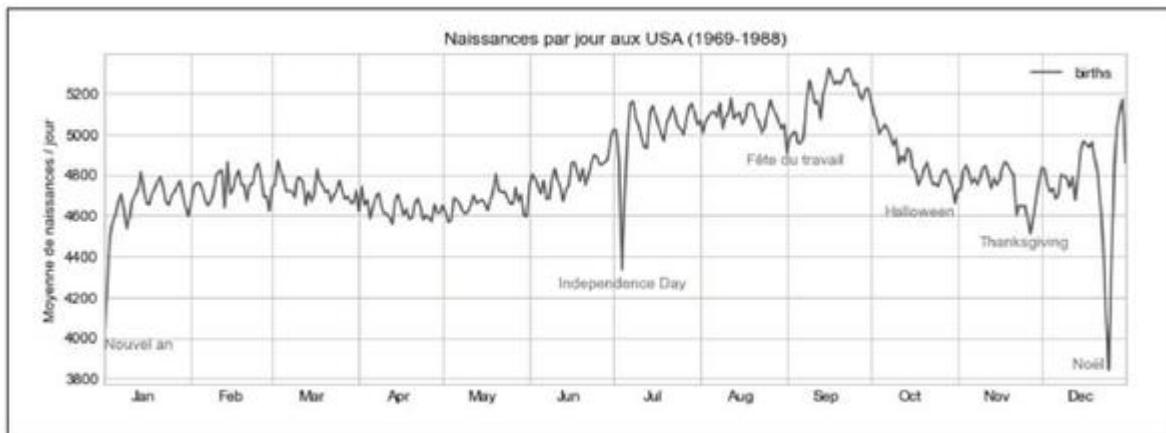


Figure 4.67 : Moyenne des naissances par jour de l'année.

Les lecteurs de ce genre de visualisation trouveront avantage à bénéficier d'annotations et de commentaires pour attirer l'attention sur certains points. Vous pouvez ajouter manuellement des éléments avec plt.text ou ax.text, pour positionner du texte à certaines coordonnées en x et y (Figure 4.68) :

```
In[4]:  
fig, ax = plt.subplots(figsize=(12, 4))  
births_by_date.plot(ax=ax)  
  
# Ajout de labels  
style = dict(size=10, color='gray')  
ax.text('2012-1-1', 3950, "Nouvel an",  
**style)  
ax.text('2012-7-4', 4250, "Independence Day",  
ha='center', **style)  
ax.text('2012-9-4', 4850, "Fête du travail",  
ha='center', **style)  
ax.text('2012-10-31', 4600, "Halloween",  
ha='right', **style)  
ax.text('2012-11-25', 4450, "Thanksgiving",  
ha='center', **style)  
ax.text('2012-12-25', 3850, "Noël ", ha='right',  
**style)  
  
# Label des axes  
ax.set(title='Naissance par jour aux USA (1969-  
1988)',  
      ylabel='Moyenne de naissances/jour')  
  
# Formatage de l'axe x avec noms de mois centrés  
ax.xaxis.set_major_locator(mpl.dates.MonthLocato  
r())  
ax.xaxis.set_minor_locator(mpl.dates.MonthLocato  
r(bymonthday=15))  
ax.xaxis.set_major_formatter(plt.NullFormatter())
```

```
)  
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```



[Figure 4.68](#) : Annotations manuelles du graphique des naissances.

La méthode `ax.text` attend en paramètre la position x, celle en y, une chaîne et des mots-clés facultatifs pour la couleur, la taille, le style et l’alignement et autres propriétés du texte. Dans l'exemple, nous avons utilisé `ha='right'` et `ha='center'` pour l’alignement horizontal. Pour les autres options, voyez la chaîne documentaire de `plt.text()` ou `mpl.text.Text()`.

Transformations et positions du texte

Les annotations que nous venons d'ajouter ont été positionnées par rapport à une valeur de données. Parfois, il

faut choisir une position indépendante d'une valeur, ce qui passe par l'objet de transformation de Matplotlib.

Un logiciel de rendu graphique doit toujours disposer d'un mécanisme pour effectuer une conversion entre des systèmes de coordonnées différents. Si les coordonnées d'un point (x,y) correspondent à (1,1), il faut convertir ces valeurs par rapport aux coordonnées de la figure, c'est-à-dire à des pixels relativement au cadre. Ces transformations correspondent à des opérations mathématiques assez simples ; Matplotlib dispose d'un jeu d'outils complet pour réaliser ces conversions. Vous pouvez étudier ces outils dans le sous-module `matplotlib.transforms`.

Les besoins habituels ne requièrent pas de plonger dans les détails de ces transformations. Il reste cependant intéressant de savoir comment cela fonctionne pour le positionnement d'un texte sur un graphique. Trois transformations prédéfinies sont utiles dans ce cadre :

`ax.transData`

Transformation reposant sur les coordonnées des données à visualiser.

`ax.transAxes`

Transformation dépendante des axes exprimée en unités d'axes.

```
fig.transFigure
```

Transformation associée à la surface complète de la figure en unités de la figure.

L'exemple qui suit affiche du texte en différentes positions au moyen de ces transformations ([Figure 4.69](#)) :

In[5] :

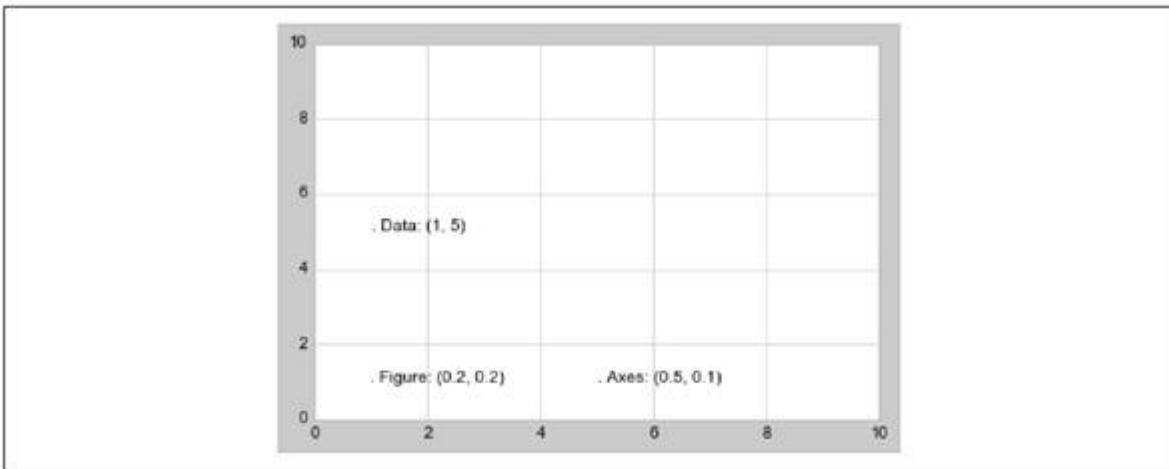
```
fig, ax = plt.subplots(facecolor='lightgray')
ax.axis([0, 10, 0, 10])

# transform=ax.transData est par défaut, mais
# soyons explicites
ax.text(1, 5, ". Data: (1, 5)",
        transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)",
        transform=ax.transAxes)
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)",
        transform=fig.transFigure);
```

Par défaut, le texte est aligné à gauche et au-dessus du point de coordonnées concerné, qui correspond ici au point (<< . >>) placé au début de chaque chaîne.

Les coordonnées transData coïncident avec les coordonnées de l'abscisse et de l'ordonnée en x et en y. Les coordonnées

`transAxes` sont relatives à l'angle inférieur gauche des axes, c'est-à-dire au coin de la zone à fond blanc. Elles correspondent à une fraction de la taille des axes. Enfin, les coordonnées `transFigure` sont relatives à l'angle inférieur gauche de la figure, en pourcentage de la taille de celle-ci, la figure allant jusqu'aux limites de la zone en gris.



[Figure 4.69](#) : Les trois contextes de coordonnées de Matplotlib.

Si nous modifions la limite des axes, seules les coordonnées `transData` changent, les autres n'en dépendant pas ([Figure 4.70](#)) :

```
In[6]:  
ax.set_xlim(0, 2)  
ax.set_ylim(-6, 6)  
fig
```

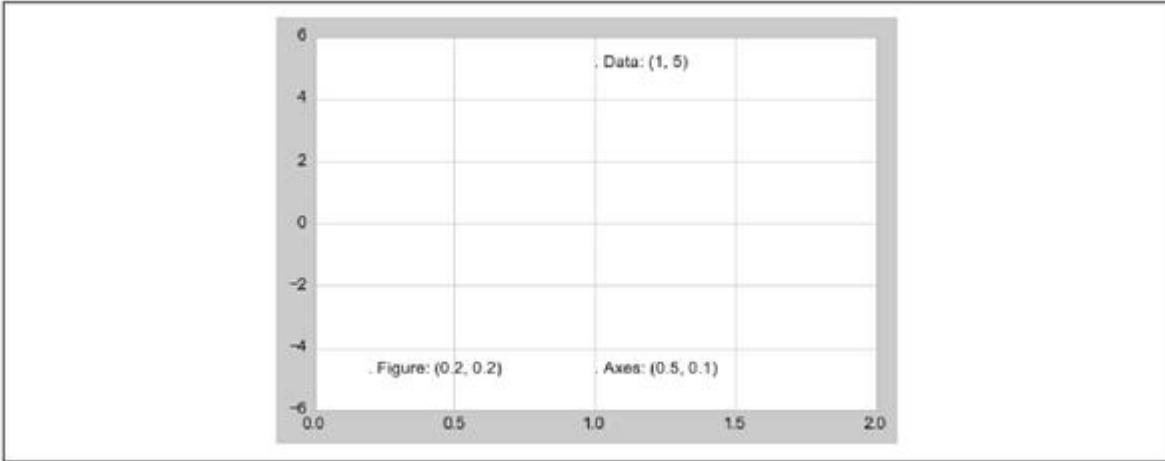


Figure 4.70 : Les coordonnées dépendantes des limites d'axes.

Vous pouvez vérifier cette dépendance de façon interactive en exécutant le code dans un calepin, et en basculant le mode `%matplotlib inline` en `%matplotlib notebook`. Vous pourrez alors utiliser le menu pour interagir avec le tracé.

Flèches et annotations

En complément des graduations et des textes volants, une annotation très utile consiste à ajouter des flèches.

L'opération est moins simple que prévu dans Matplotlib. Il existe une fonction nommée `plt.arrow()` mais je déconseille son emploi, car elle produit des objets SVG dont la taille va dépendre des proportions de vos tracés, et cela sera rarement satisfaisant. Je vous conseille plutôt de vous servir de la fonction `plt.annotate()` qui permet d'insérer du texte,

mais également une flèche, avec un vaste choix au niveau de son aspect.

Voyons quelques options de cette fonction `annotate()` ([Figure 4.71](#)) :

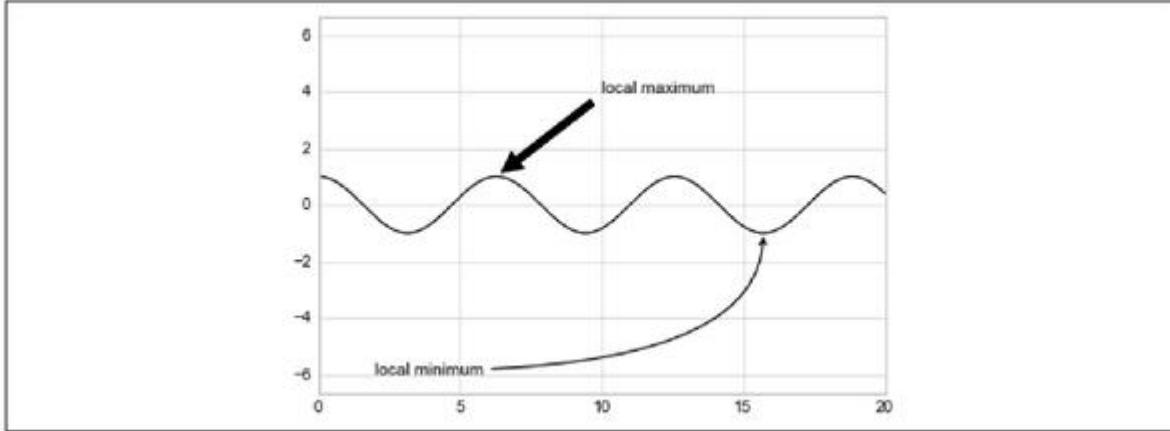
In[7] :

```
%matplotlib inline

fig, ax = plt.subplots()

x = np.linspace(0, 20, 1000)
ax.plot(x, np.cos(x))
ax.axis('equal')

ax.annotate('local maximum', xy=(6.28, 1),
            xytext=(10, 4),
            arrowprops=dict(facecolor='black',
                            shrink=0.05))
ax.annotate('local minimum', xy=(5 * np.pi, -1),
            xytext=(2, -6),
            arrowprops=dict(arrowsyle= '-> ',
                           connectionstyle=
angle3, angleA=0, angleB=-90 ));
```



[Figure 4.71](#) : Deux exemples d'annotations avec des flèches.

Vous contrôlez le style de la flèche au moyen du dictionnaire nommé `arrowprops`. Les nombreuses options sont décrites dans la documentation en ligne de Matplotlib. Contentons-nous d'en découvrir quelques-unes, en repartant de l'exemple des naissances aux USA ([Figure 4.72](#)) :

In[8] :

```
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Ajout de labels
ax.annotate("Nouvel An", xy=('2012-1-1', 4100),
            xycoords='data',
            xytext=(50, -30), textcoords='offset
points',
            arrowprops=dict(arrowstyle='-> ,
                           connectionstyle=
                           arc3, rad=-0.2 ))
```

```
ax.annotate("Independence Day", xy=( '2012-7-4' ,  
4250), xycoords='data',  
bbox=dict(boxstyle= round , fc= none  
, ec= gray ),  
xytext=(10, -40), textcoords='offset  
points', ha='center',  
arrowprops=dict(arrowstyle="->"))  
  
ax.annotate("Fête du travail", xy=( '2012-9-4' ,  
4850), xycoords='data',  
ha='center', xytext=(0, -20),  
textcoords='offset points')  
  
ax.annotate(' ', xy=( '2012-9-1' , 4850), xytext=  
( '2012-9-7' , 4850),  
xycoords='data', textcoords='data',  
arrowprops={'arrowstyle': '|-'  
, widthA=0.2, widthB=0.2', })  
  
ax.annotate("Halloween", xy=( '2012-10-31' ,  
4600), xycoords='data',  
xytext=(-80, -40), textcoords='offset  
points',  
arrowprops=dict(arrowstyle="fancy",  
fc="0.6", ec="none",  
connectionstyle="angle3,angleA=0,angleB=-90"))  
  
ax.annotate("Thanksgiving", xy=( '2012-11-25' ,  
4500), xycoords='data',
```

```
xytext=(-120, -60),
textcoords='offset points',
bbox=dict(boxstyle="round4, pad=.5",
fc="0.9"),
arrowprops=dict(arrowstyle="->",

connectionstyle="angle, angleA=0, angleB=80, rad=20
"))

ax.annotate("Noël", xy=('2012-12-25', 3850),
xycoords='data',
xytext=(-30, 0), textcoords='offset
points',
size=13, ha='right', va="center",
bbox=dict(boxstyle="round",
alpha=0.1),

arrowprops=dict(arrowstyle="wedge, tail_width=0.5
", alpha=0.1));

# Label des axes
ax.set(title='Naissances aux USA par jour (1969-
1988)',
ylabel='average daily births')

# Format de l'axe x avec labels de mois centrés
ax.xaxis.set_major_locator(mpl.dates.MonthLocato
r())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocato
r(bymonthday=15))
```

```

ax.xaxis.set_major_formatter(plt.NullFormatter())
)
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
ax.set_xlim(3600, 5400);

```

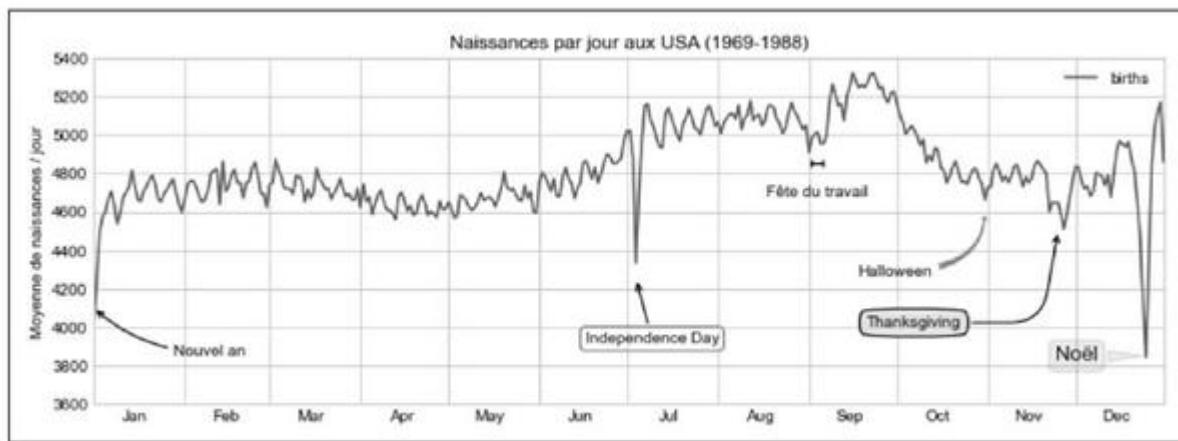


Figure 4.72 : Annotations pour les jours fériés aux USA.

Vous constatez que la définition des textes d'annotation et des flèches est très détaillée. Mais bien profiter de cette précision pour obtenir un rendu professionnel suppose de faire des essais manuels, ce qui peut prendre du temps ! Précisons que les choix de format des flèches et du texte de l'exemple sont disparates pour des raisons pédagogiques.

Vous trouverez d'autres exemples concernant les flèches et les annotations dans la galerie de Matplotlib, à l'adresse http://matplotlib.org/examples/pylab_examples/annotation_demo2.html.

4.12 : Personnalisation des graduations

Les marques de graduations, d'échelles et les traits de repérage que Matplotlib propose par défaut conviennent dans la plupart des cas, mais des personnalisations sont souvent les bienvenues. Nous allons voir comment ajuster la position des graduations et des éléments de format en fonction du besoin de vos tracés.

Avant de passer à des exemples, il n'est pas inutile de mieux comprendre la hiérarchie des objets d'un tracé Matplotlib. Un objet Python principal regroupe tout ce qui apparaît dans le tracé. Vous vous souvenez que l'objet nommé `figure` représente le cadre autour des éléments du tracé. Un objet Matplotlib peut à son tour contenir des sous-objets. Bien sûr, `figure` contient un ou plusieurs objets de type `axes`, chacun d'eux contenant des objets qui incarnent le contenu à tracer.

En ce qui concerne les graduations, chaque objet `axes` possède deux attributs `xaxis` et `yaxis` dont les attributs permettent de définir les propriétés des lignes, des graduations et des labels des axes.

Graduations majeures et mineures

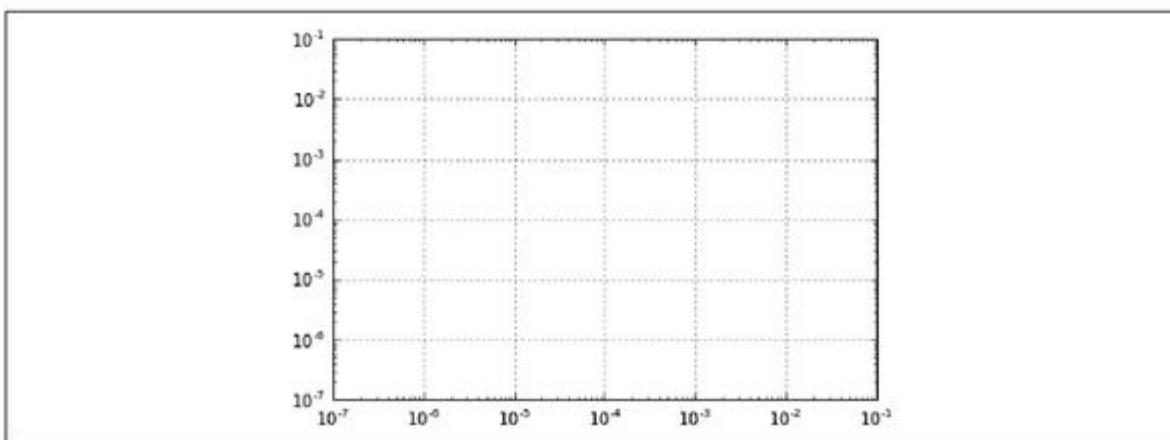
Chaque axe peut porter des graduations majeures et mineures, les premières étaient en général plus épaisses. Matplotlib affiche rarement les graduations mineures, sauf en particulier pour les tracés logarithmiques ([Figure 4.73](#)) :

In[1] :

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

In[2] :

```
ax = plt.axes(xscale='log',yscale='log')
```



[Figure 4.73](#) : Exemple de graduations et de labels logarithmiques.

On constate que les graduations majeures sont prolongées par un quadrillage alors que les graduations mineures n'en bénéficient pas (et n'ont pas de label).

Vous pouvez intervenir sur les propriétés des graduations, c'est-à-dire leur position, leur nombre et leur légende en utilisant les objets formatter et locator de l'axe. Voici comment personnaliser ces éléments pour l'axe x des abscisses :

In[3] :

```
print(ax.xaxis.get_major_locator())
print(ax.xaxis.get_minor_locator())
```

```
<matplotlib.ticker.LogLocator object at
0x0000027A1B35C700>
<matplotlib.ticker.LogLocator object at
0x0000027A18187520>
```

In[4] :

```
print(ax.xaxis.get_major_formatter())
print(ax.xaxis.get_minor_formatter())
```

```
<matplotlib.ticker.LogFormatterMathtext object
at 0x107512780>
<matplotlib.ticker.NullFormatter object at
0x10752dc18>
```

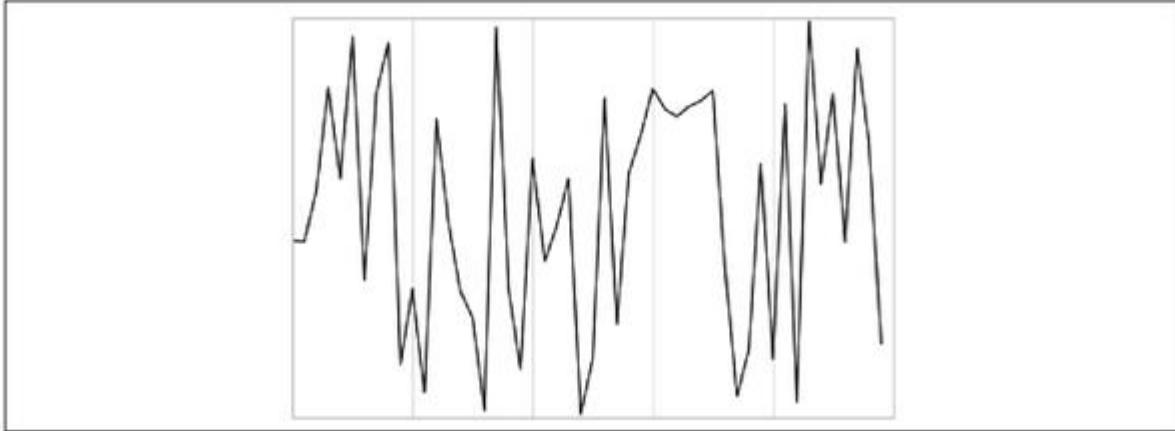
Nous utilisons l'objet nommé LogLocator pour les labels des graduations (ce qui est logique pour un tracé logarithmique). Vous remarquez que les graduations mineures sont gérées au niveau du label par un objet NullFormatter, ce qui signifie qu'il n'y a pas de labels.

Voyons les adaptations possibles de ces deux objets pour d'autres formats de tracé.

Masquer les graduations ou les labels

Une des opérations les plus fréquentes consiste à masquer les graduations ou seulement les labels au moyen des deux fonctions plt.NullLocator() et plt.NullFormatter() ([Figure 4.74](#)) :

```
In[5]:  
ax = plt.axes()  
ax.plot(np.random.rand(50))  
ax.yaxis.set_major_locator(plt.NullLocator())  
ax.xaxis.set_major_formatter(plt.NullFormatter())  
)
```



[Figure 4.74](#) : Masquage des labels sur l'axe x et des graduations sur l'axe y.

Dans l'exemple, les graduations restent visibles pour l'axe horizontal x alors que tout a disparu pour l'axe y. Dans certains cas, il est utile de masquer toutes les graduations, par exemple lorsque vous voulez afficher une mosaïque d'images comme celle de la [Figure 4.75](#) (il s'agit d'un exemple d'apprentissage machine supervisé que nous verrons dans le [Chapitre 5](#)) :

In[6]:

```
fig, ax = plt.subplots(5, 5, figsize=(5, 5))
fig.subplots_adjust(hspace=0, wspace=0)
```

```
# Prélèvement de données de visages depuis
scikit-learn
```

```
from sklearn.datasets import
```

```
fetch_olivetti_faces
```

```
faces = fetch_olivetti_faces().images
```

```
for i in range(5):
```

```
for j in range(5):
    ax[i,
j].xaxis.set_major_locator(plt.NullLocator())
    ax[i,
j].yaxis.set_major_locator(plt.NullLocator())
    ax[i, j].imshow(faces[10 * i + j],
cmap="bone")
```

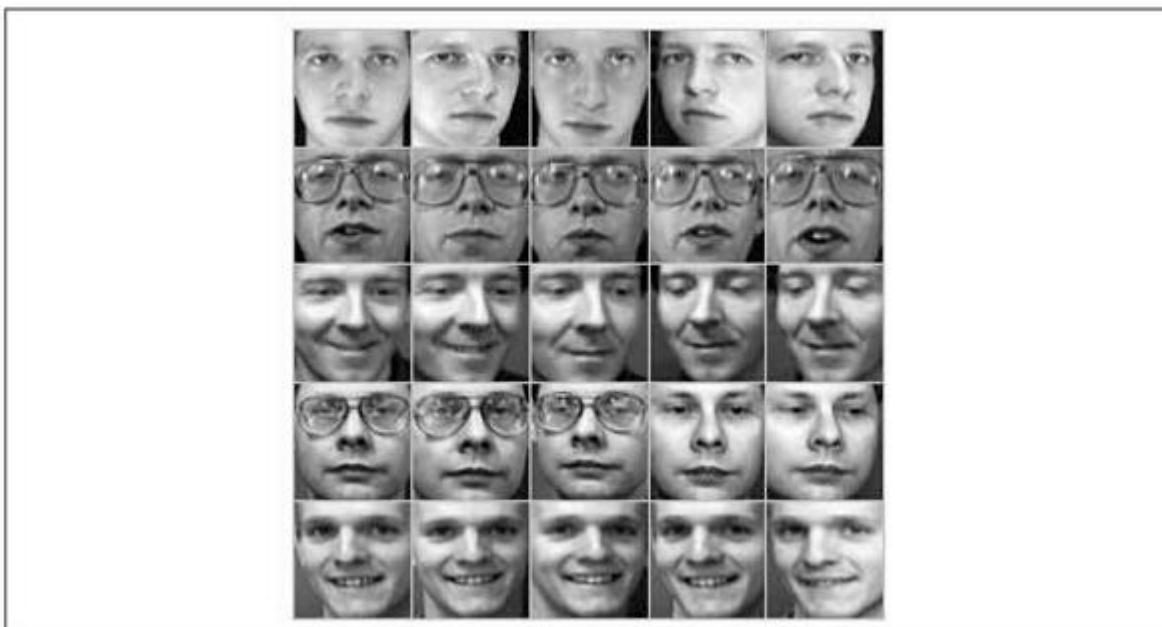


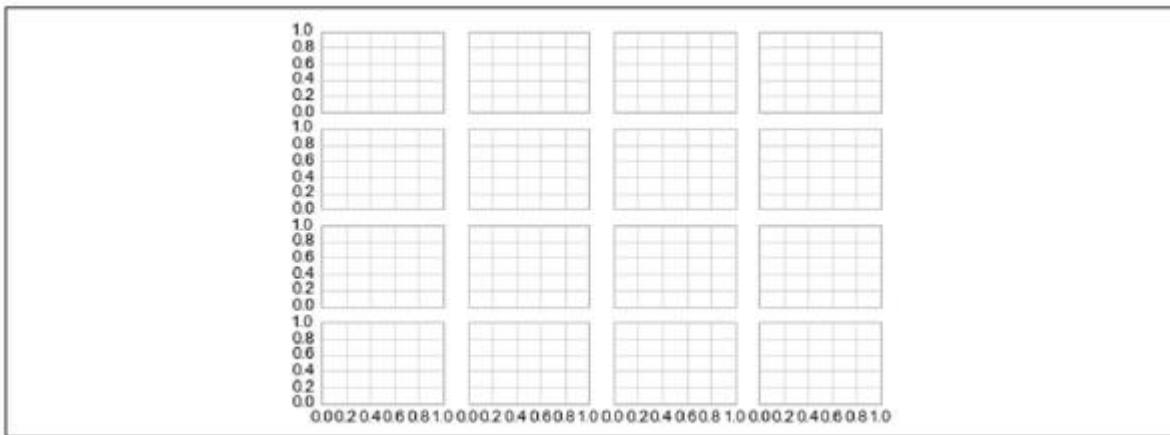
Figure 4.75 : Masquage de toutes les graduations d'axes pour une mosaïque d'images.

Notez que chaque image est placée dans son propre repère orthonormé. Nous avons masqué les graduations parce qu'elles n'apportent aucune information utile (ce sont des numéros de pixels).

Contrôle de la densité des graduations

Lorsque vous composez une visualisation à partir de plusieurs tracés, les choix par défaut peuvent aboutir à des graduations qui se chevauchent comme en [Figure 4.76](#) :

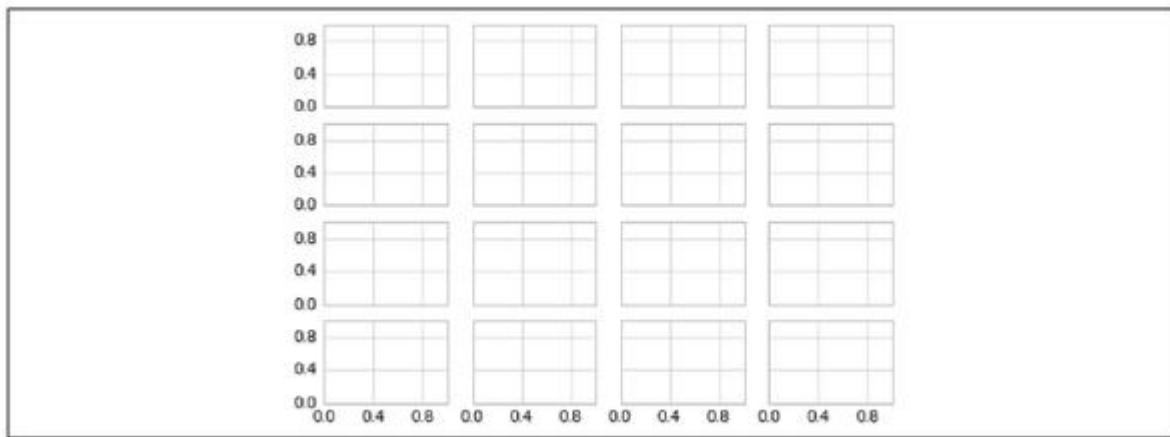
```
In[7]: fig, ax = plt.subplots(4, 4, sharex=True,  
sharey=True)
```



[Figure 4.76](#) : Des légendes d'axes trop denses en chevauchement.

Les valeurs numériques se chevauchent souvent, notamment sur l'axe x, ce qui les rend illisibles. Vous pouvez décider du nombre de graduations au moyen de la fonction plt.MaxNLocator(). Matplotlib va se débrouiller pour choisir les positions des graduations en fonction du nombre maximum indiqué ([Figure 4.77](#)) :

```
In[8]:  
for axi in ax.flat:  
  
    axi.xaxis.set_major_locator(plt.MaxNLocator(3))  
  
    axi.yaxis.set_major_locator(plt.MaxNLocator(3))  
fig
```



[Figure 4.77](#) : Réduction du nombre de graduations.

L'image devient beaucoup plus lisible. La position et la légende des graduations peuvent également être personnalisées avec plt.MultipleLocator, comme nous allons le voir dans quelques lignes.

Format de graduations personnalisé

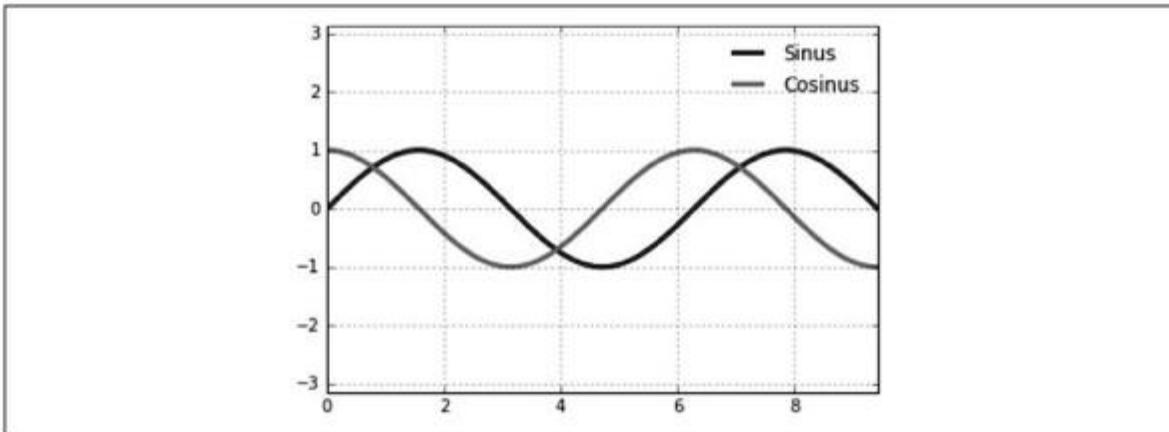
Le format par défaut des graduations de Matplotlib n'est pas toujours idéal. Partons d'un classique tracé de sinus et

cosinus ([Figure 4.78](#)) :

In[9] :

```
fig, ax = plt.subplots()
x = np.linspace(0, 3 * np.pi, 1000)
ax.plot(x, np.sin(x), lw=3, label='Sinus')
ax.plot(x, np.cos(x), lw=3, label='Cosinus')

# Choix de la grille, des légendes et des
# limites
ax.grid(True)
ax.legend(frameon=False)
ax.axis('equal')
ax.set_xlim(0, 3 * np.pi);
```



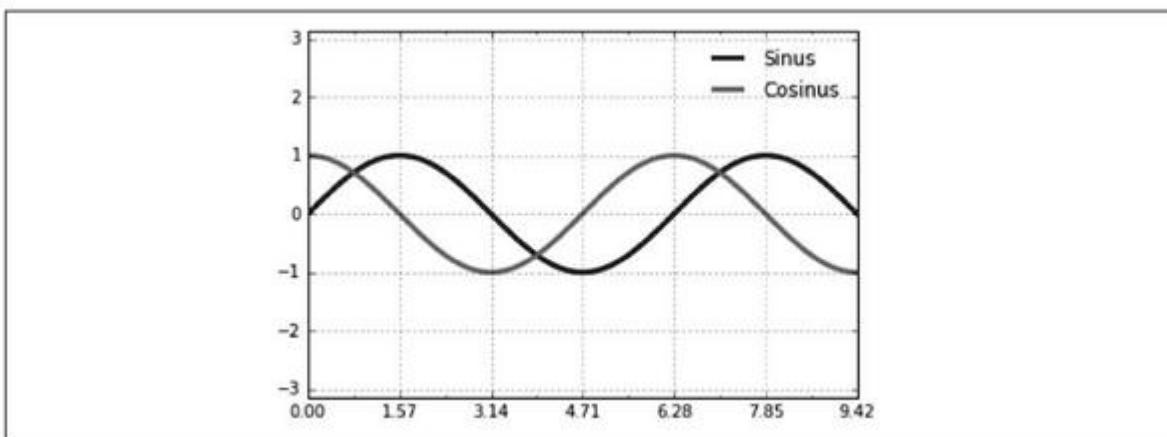
[Figure 4.78](#) : Tracé avec graduations par défaut.

La première amélioration possible consiste à utiliser des multiples de π pour les graduations horizontales. Il suffit d'utiliser MultipleLocator(). Nous en profitons pour régler

les graduations majeures et mineures en multiples de $\pi/4$ ([Figure 4.79](#)) :

In[10]:

```
ax.xaxis.set_major_locator(plt.MultipleLocator(n  
p.pi / 2))  
ax.xaxis.set_minor_locator(plt.MultipleLocator(n  
p.pi / 4))  
fig
```



[Figure 4.79](#) : Graduations exprimées en multiples de $\pi/2$.

Le résultat n'est pas encore idéal : nous devinons qu'il s'agit de multiples de π , mais la représentation décimale n'est pas la meilleure. Nous pouvons intervenir sur le formateur de graduations, c'est-à-dire plt.FuncFormatter(). Vous pouvez lui transmettre une fonction définie par le programmeur pour contrôler précisément l'aspect des graduations ([Figure 4.80](#)) :

```
In[11]:
```

```
def format_func(value, tick_number):
    N = int(np.round(2 * value / np.pi))
    if N == 0:
        return "0"
    elif N == 1:
        return r"\pi/2"
    elif N == 2:
        return r"\pi"
    elif N % 2 > 0:
        return r"\{0}\pi/2".format(N)
    else:
        return r"\{0}\pi".format(N // 2)
ax.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
fig
```

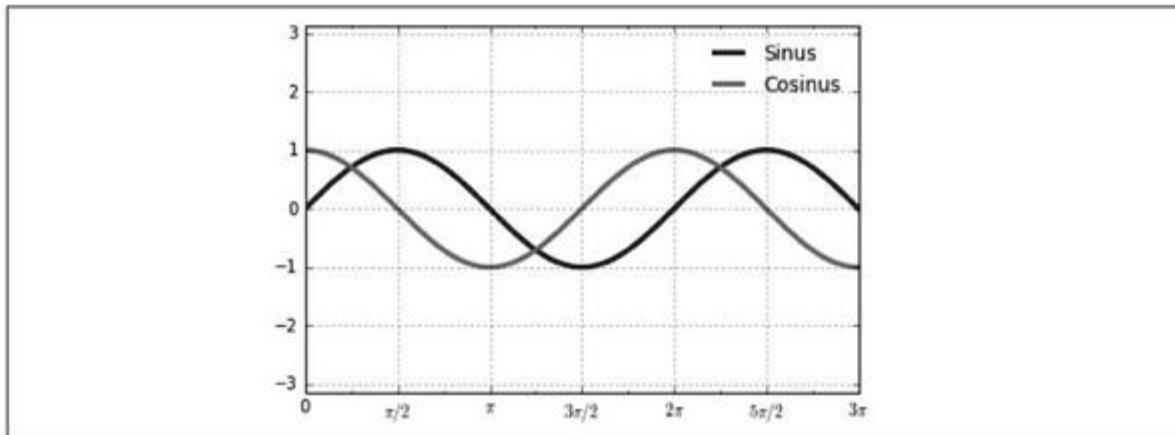


Figure 4.80 : Personnalisation complète des labels de graduations.

Le résultat devient vraiment sympathique. Je précise que nous avons profité du support du langage de mise en page

LaTeX de Matplotlib en entourant la chaîne de spécification par des signes dollar. Cette syntaxe est très pratique pour les symboles mathématiques et les formules. Dans l'exemple, la chaîne `\$\\pi$` permet d'afficher le symbole grec π .

Cette fonction `plt.FuncFormatter()` permet de contrôler précisément les graduations des tracés, et vous serez ravi d'y avoir recours pour vos tracés destinés à des conférences ou à des publications.

Liste de formateurs et de locateurs

Les deux tableaux qui suivent présentent successivement les locateurs et les formateurs utilisables pour contrôler les graduations des axes. Pour d'autres détails, vous vous reporterez à la documentation de Matplotlib. Notez que tous ces objets font partie de l'espace de noms `plt` :

Classe Locator	Description
<code>NullLocator</code>	Aucune graduation
<code>FixedLocator</code>	Positions des graduations fixes
<code>IndexLocator</code>	Locateur pour les points d'index (p.ex., quand <code>x = range(len(y))</code>)
<code>LinearLocator</code>	Graduations équidistantes entre min et max
<code>LogLocator</code>	Graduations logarithmiques entre min et max
<code>MultipleLocator</code>	Graduations et plage multiples de base

MaxNLocator	Trouve un nombre de graduations maximal en des points appropriés
AutoLocator	(Par défaut) MaxNLocator avec choix par défaut simples
AutoMinorLocator	Locateur pour les sous-graduations

Classe Formatter	Description
NullFormatter	Aucun label de graduation
IndexFormatter	Sélection des chaînes depuis une liste de labels
FixedFormatter	Sélection manuelle des chaînes de labels
FuncFormatter	Labels produits par une fonction définie par le programmeur
FormatStrFormatter	Utilise une chaîne de format pour chaque valeur
ScalarFormatter	(Par défaut) Formateur pour scalaires
LogFormatter	Formateur par défaut pour axes log

Nous utiliserons certains de ces objets dans les exemples de la suite du livre.

4.13 : Configuration et feuilles de styles de Matplotlib

Le paramétrage implicite ou par défaut du tracé avec Matplotlib ne convient que rarement. Les choses se sont bien améliorées depuis la version 2.0, mais il reste indispensable de pouvoir personnaliser le paramétrage.

Découvrons donc les options de configuration d'exécution rc (*runtime configuration*) puis le mécanisme des feuilles de styles qui permet de mettre en place toute une palette d'options en un geste.

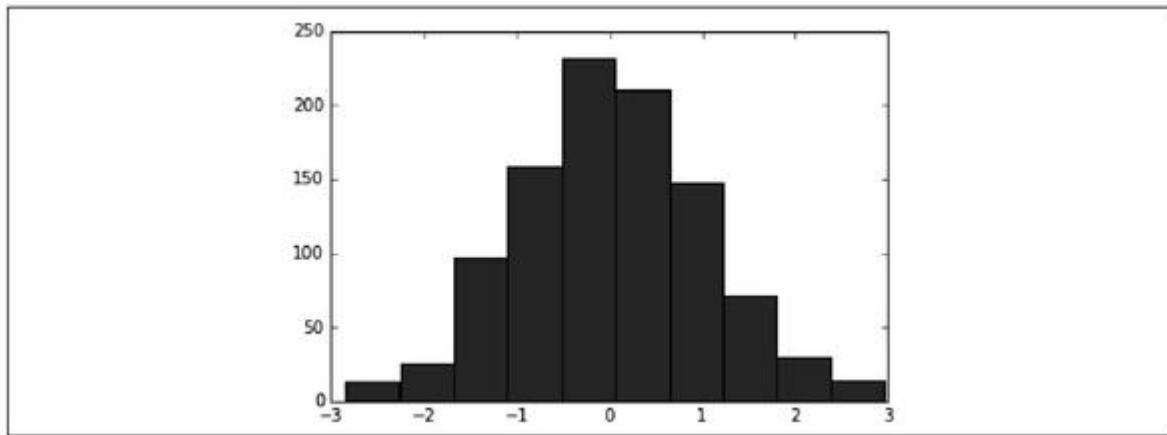
Personnalisation manuelle

Depuis le début du chapitre, nous avons vu comment retoucher l'un ou l'autre des paramètres d'un tracé pour l'adapter. Ce réglage est possible pour chaque tracé individuel. Partons d'un histogramme par défaut assez peu attrayant ([Figure 4.81](#)) :

```
In[1]:  
import matplotlib.pyplot as plt  
plt.style.use('classic')  
import numpy as np  
%matplotlib inline
```

```
In[2]:
```

```
x = np.random.randn(1000)  
plt.hist(x);
```



[Figure 4.81](#) : Histogramme Matplotlib dans le style par défaut.

Pour rendre le tracé plus agréable, nous allons retoucher certains paramètres manuellement ([Figure 4.82](#)) :

```
In[3]:
```

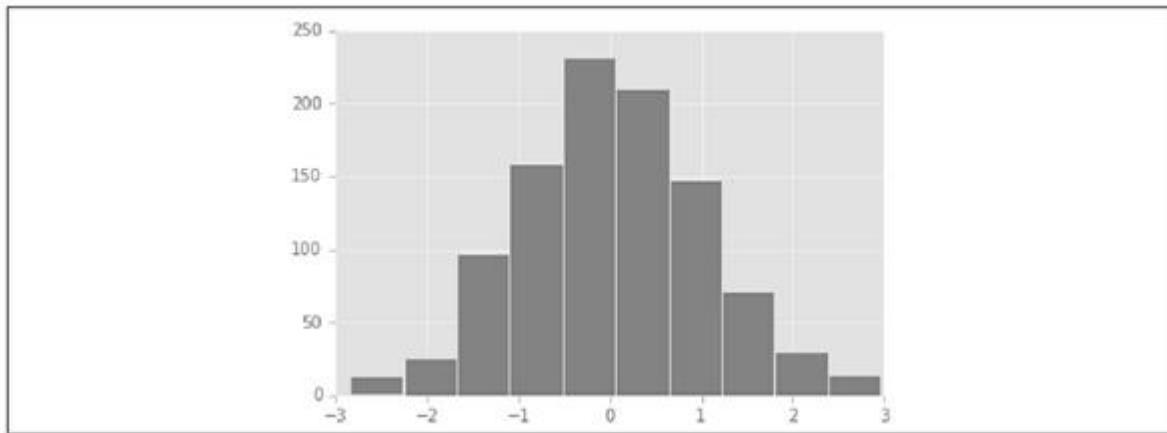
```
# Fond gris  
ax = plt.axes(facecolor='#E6E6E6')  
ax.set_axisbelow(True)  
  
# Traits blancs épais  
plt.grid(color='w', linestyle='solid')  
  
# Marques d'axes masquées  
for spine in ax.spines.values():  
    spine.set_visible(False)  
  
# Graduations du haut et de droite masquées
```

```

ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

# Graduations et labels éclaircis
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')
# Couleur de face et de bord des barres
d'histogramme
ax.hist(x, edgecolor='#E6E6E6',
color='#EE6666');

```



[Figure 4.82](#) : Le même histogramme après personnalisation d'aspect.

Le résultat est moins terne, et certains remarqueront la parenté avec ce que produit le paquetage de visualisation ggplot du langage R. Mais cela nous a obligés à écrire beaucoup de lignes ! Pour ne pas devoir répéter ce

paramétrage avant chaque tracé, des solutions existent pour réutiliser un jeu de paramètres dans plusieurs tracés.

Changement des valeurs par défaut avec rcParams

Lors de son démarrage, Matplotlib exploite une configuration d'exécution rc dans laquelle est défini l'ensemble des styles par défaut pour les tracés. Vous pouvez vous servir de la routine technique plt.rc() pour modifier la configuration. Voyons comment instaurer par défaut les paramètres pour obtenir notre tracé attrayant.

Nous prenons soin de créer une copie de sécurité du dictionnaire rcParams actuel pour pouvoir le restaurer au cas où nos modifications ne seraient pas satisfaisantes :

```
In[4]: IPython_default = plt.rcParams.copy()
```

Nous pouvons ensuite modifier les paramètres par défaut par différents appels à la fonction plt.

rc() :

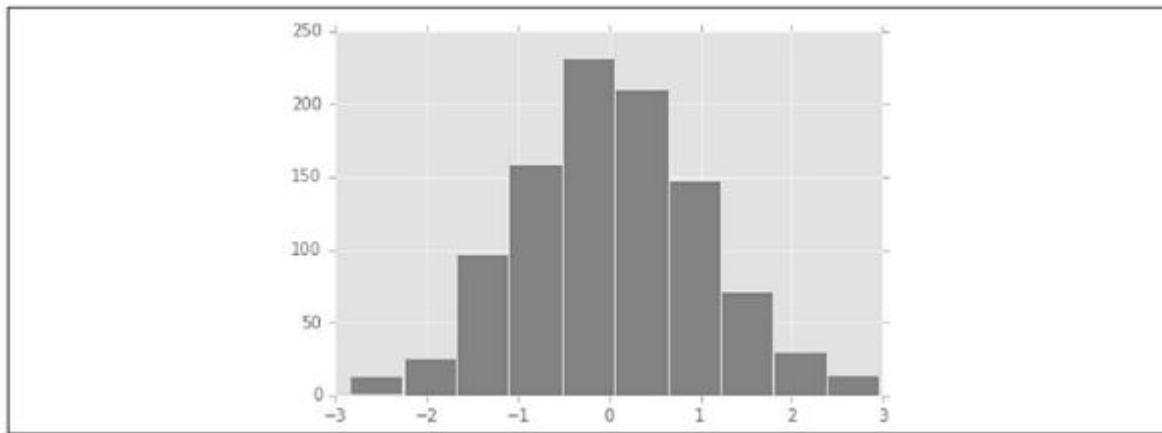
```
In[5]:
```

```
from matplotlib import cycler
colors = cycler('color',
                ['#EE6666', '#3388BB', '#9988DD',
                 '#EECC55', '#88BB44', '#FFBBBB'])
```

```
plt.rc('axes', facecolor='#E6E6E6',
      edgecolor='none',
      axisbelow=True, grid=True,
      prop_cycle=colors)
plt.rc('grid', color='w', linestyle='solid')
plt.rc('xtick', direction='out', color='gray')
plt.rc('ytick', direction='out', color='gray')
plt.rc('patch', edgecolor='#E6E6E6')
plt.rc('lines', linewidth=2)
```

Il ne reste plus qu'à tester ces paramètres en demandant un tracé ([Figure 4.83](#)) :

```
In[6]: plt.hist(x);
```



[Figure 4.83](#) : Le même histogramme personnalisé grâce aux paramétrages rc.

Voyons à quoi ressemble une série de lignes avec les paramètres rc par défaut ([Figure 4.84](#)) :

```
In[7]:
for i in range(4):
```

```
plt.plot(np.random.rand(10))
```

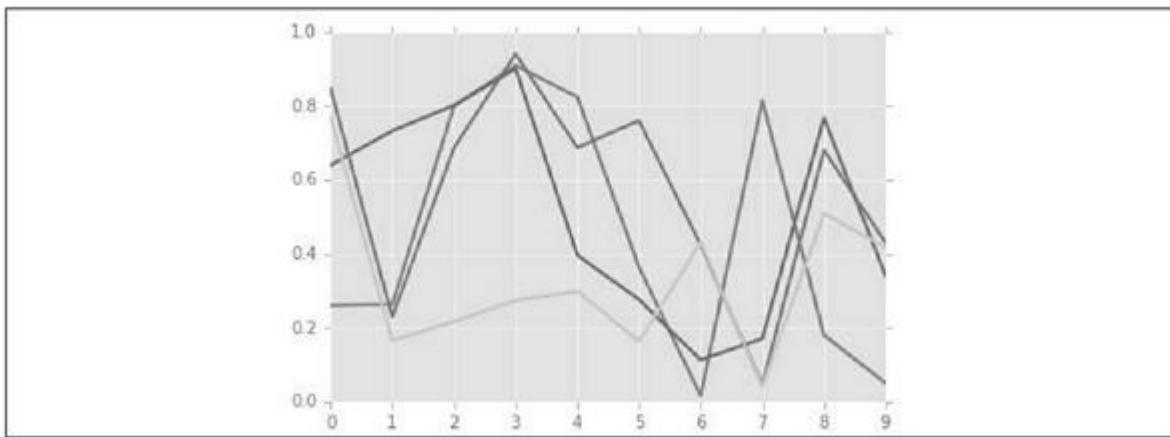


Figure 4.84 : Un tracé de lignes personnalisé.

N'hésitez pas à retoucher les paramètres par défaut en fonction de vos besoins. Vous stockez un jeu de paramètres dans un fichier avec une extension `.matplotlibrc` comme indiqué dans la documentation de Matplotlib. Ceci dit, je préfère encore personnaliser mes tracés Matplotlib en me servant des feuilles de styles.

Feuilles de styles

Depuis sa version 1.4, Matplotlib dispose d'un module nommé `style` qui est livré avec un certain nombre de styles par défaut, tout en permettant d'en créer et d'en adapter d'autres. Ces feuilles de styles se présentent au même format que les fichiers `.matplotlibrc` déjà mentionnés, mais le nom de fichier doit se terminer par l'extension `.mplstyle`.

Avant de voir comment créer vos propres styles, passons en revue les nombreux styles fournis. Vous en obtenez la liste au moyen de plt.style.available :

In[8] :

```
plt.style.available[:15]
```

Out[8] :

```
['Solarize_Light2',
 '_classic_test_patch',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn',
 'seaborn-bright',
 'seaborn-colorblind',
 'seaborn-dark',
 'seaborn-dark-palette',
 'seaborn-darkgrid']
```

Pour activer un nouveau style, vous appelez la fonction use() ainsi :

```
plt.style.use('nomstyle')
```

Ce basculement s'applique jusqu'à la fin de la session courante ! Pour ne changer de feuilles de styles que pour un tracé par exemple, il suffit d'utiliser le gestionnaire de contexte de style :

```
with plt.style.context('nomstyle'):  
    make_a_plot()
```

Pour passer en revue plusieurs des styles par défaut, définissons d'abord une fonction qui va produire côté à côté deux tracés élémentaires :

```
In[9]:  
def hist_and_lines():  
    np.random.seed(0)  
    fig, ax = plt.subplots(1, 2, figsize=(11, 4))  
    ax[0].hist(np.random.randn(1000))  
    for i in range(3):  
        ax[1].plot(np.random.rand(10))  
        ax[1].legend(['a', 'b', 'c'], loc='lower  
left')
```

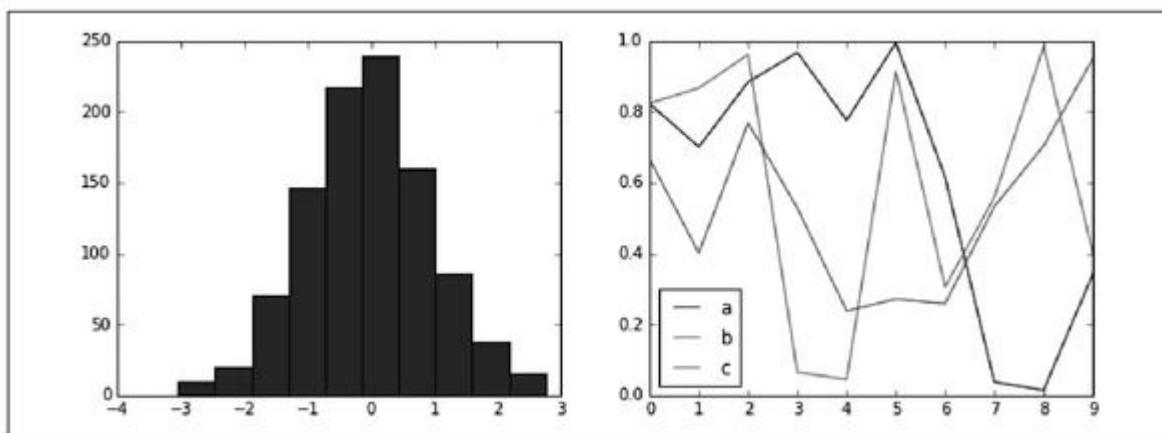
Style par défaut

Le style par défaut est celui que nous avons vu à l'œuvre depuis le début du livre. Revoyons-le en réinitialisant la configuration d'exécution rc à la valeur par défaut Ipython_default :

```
In[10]: # reset rcParams  
plt.rcParams.update(IPython_default);
```

Demandons notre premier tracé ([Figure 4.85](#)) :

```
In[11]: hist_and_lines()
```

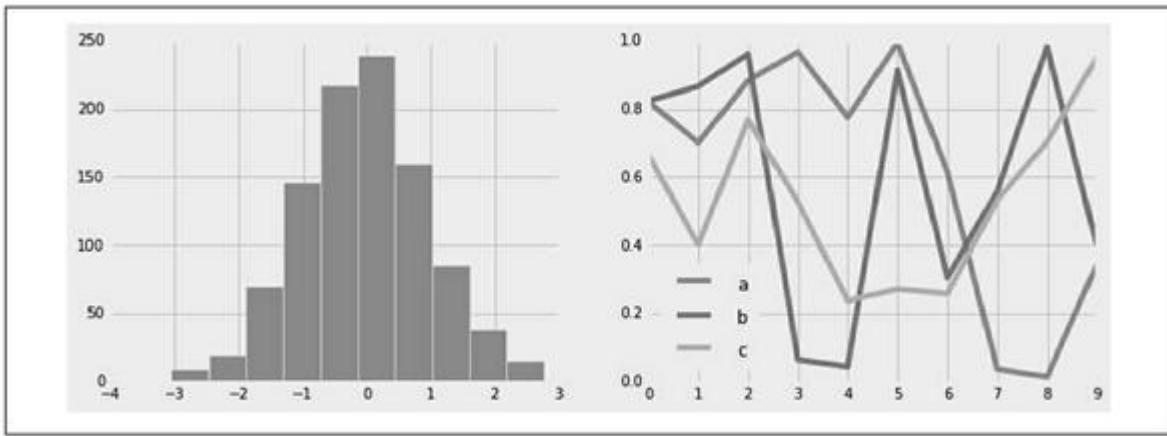


[Figure 4.85](#) : Style de tracé par défaut de Matplotlib.

Style FiveThirtyEight

Ce style s'inspire de l'aspect global du site Web nommé *FiveThirtyEight* (<https://fivethirtyeight.com/>). Il utilise des couleurs franches, des traits épais et des axes transparents ([Figure 4.86](#)) :

```
In[12]:  
with plt.style.context('fivethirtyeight'):  
    hist_and_lines()
```



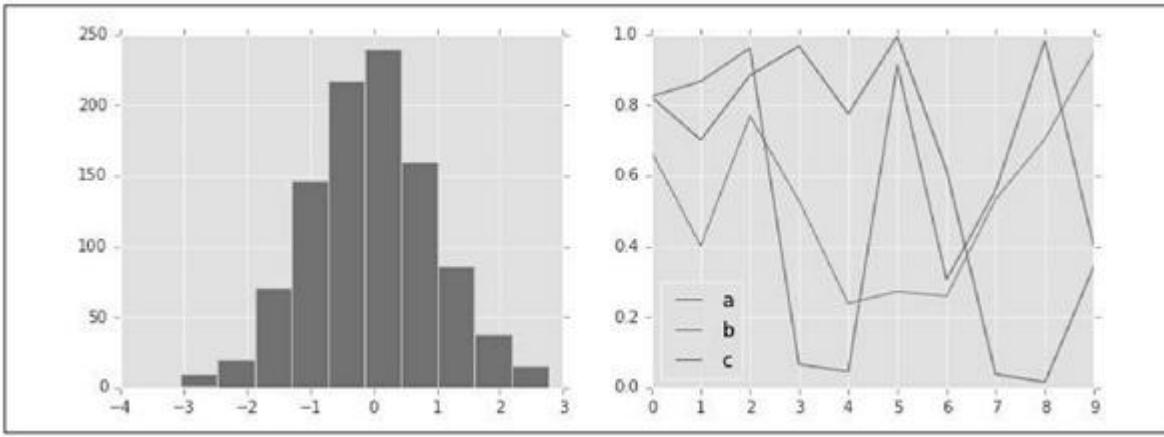
[Figure 4.86](#) : Style de tracé FiveThirtyEight.

ggplot

Ce style s'inspire des styles par défaut du paquetage ggplot très connu de ceux qui utilisent le langage R ([Figure 4.87](#)) :

In[13]:

```
with plt.style.context('ggplot'):  
    hist_and_lines()
```

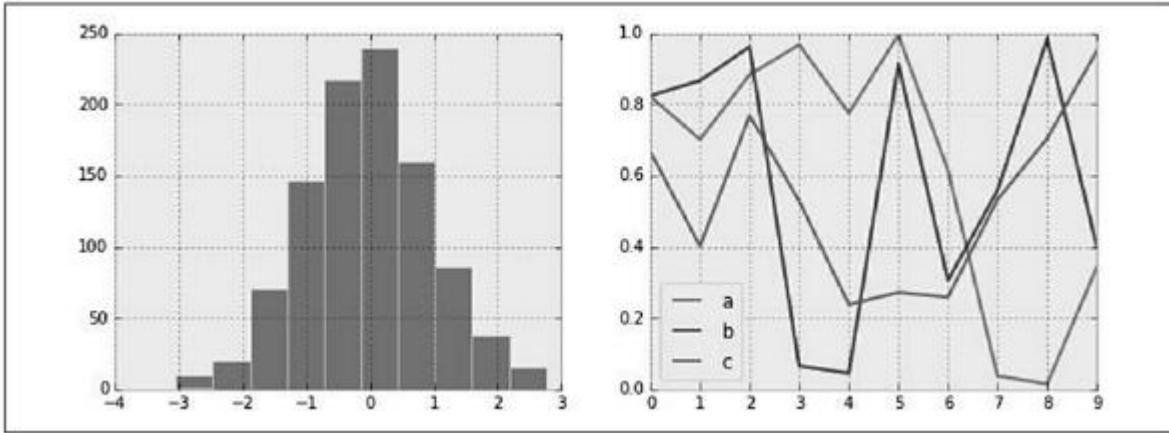


[Figure 4.87](#) : Style de tracé ggplot.

Style bmh (Bayesian Methods for Hackers)

Ce style s'inspire des figures créées avec Matplotlib dans un livre en ligne, *Probabilistic Programming and Bayesian Methods for Hackers* (<http://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/>). Le jeu de paramètres choisi donne un résultat attrayant et cohérent que reproduit le style nommé bmh ([Figure 4.88](#)) :

```
In[14]:  
with plt.style.context('bmh'):  
    hist_and_lines()
```



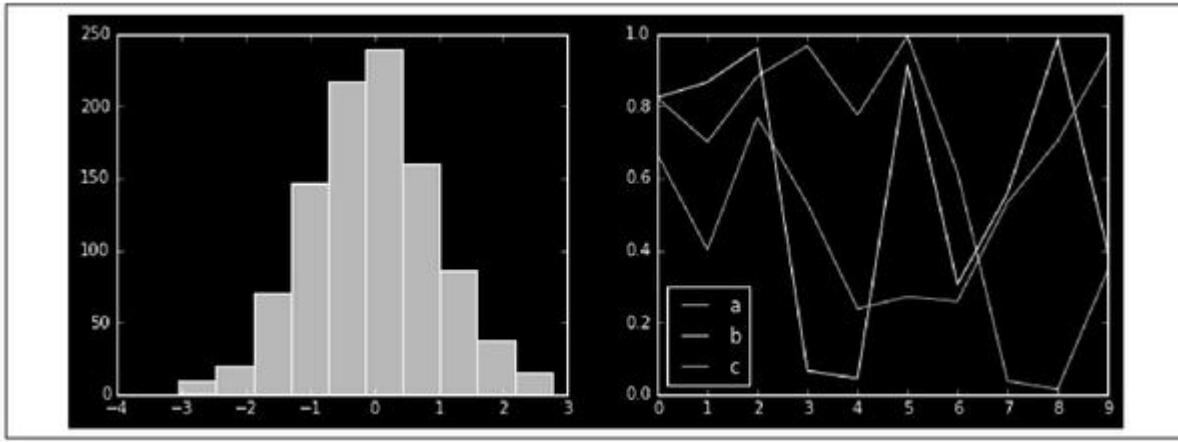
[Figure 4.88](#) : Style de tracé `bmh`.

Arrière-plan sombre (dark_background)

Il est souvent utile de pouvoir faire ressortir des images sur un fond sombre pour les figures des présentations, ce que permet le style `dark_background` ([Figure 4.89](#)) :

In[15]:

```
with plt.style.context('dark_background'):
    hist_and_lines()
```

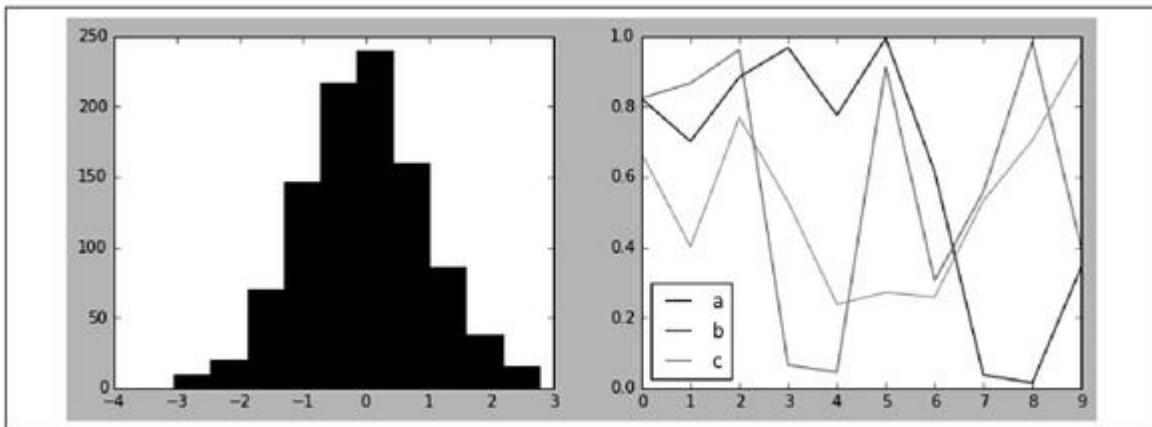


[Figure 4.89](#) : Style de tracé dark_background.

Nuances de gris (grayscale)

Lorsque vous préparez des figures qui sont destinées à être imprimées en noir et blanc, vous utiliserez ce style grayscale ([Figure 4.90](#)) :

```
In[16]:  
with plt.style.context('grayscale'):  
hist_and_lines()
```



[Figure 4.90](#) : Style de tracé grayscale.

Styles Seaborn

Matplotlib offre plus d'une quinzaine de feuilles de styles qui s'inspirent de la librairie de visualisation Seaborn que nous verrons en détail dans la dernière section de ce chapitre. Ces styles sont automatiquement chargés dès que vous importez la librairie Seaborn dans un calepin. J'ai tendance à privilégier ces styles par défaut dans mes explorations de données ([Figure 4.91](#)) :

```
In[17]:  
import seaborn  
hist_and_lines()
```

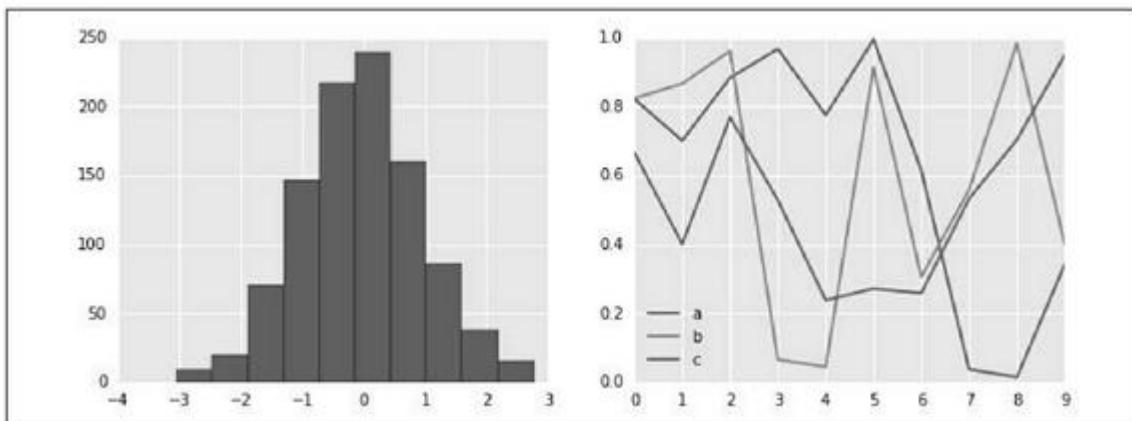


Figure 4.91 : Un style de tracé de la famille Seaborn.

Toutes ces options permettent d'exploiter Matplotlib aussi bien pour les visualisations interactives que pour préparer des figures destinées à être publiées. J'utilise bien sûr ces différents éléments de style dans les exemples du livre.

4.14 : Tracés Matplotlib en 3D

Au départ, Matplotlib ne pouvait travailler qu'en deux dimensions, mais dès la sortie de la version 1.0, des possibilités de tracé en trois dimensions ont été ajoutées. On dispose ainsi de possibilités de visualisation suffisantes, même si quelque peu limitées. Pour en profiter, il suffit de demander l'importation du module mplot3d qui est installé en même temps que Matplotlib ([Figure 4.92](#)) :

```
In[1]: from mpl_toolkits import mplot3d
```

Une fois le module importé, vous demandez une vue trois axes en spécifiant le mot-clé `projection='3d'` dans un appel de fonction de création d'axes :

```
In[2]:  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt
```

```
In[3]:  
fig = plt.figure()  
ax = plt.axes(projection='3d')
```

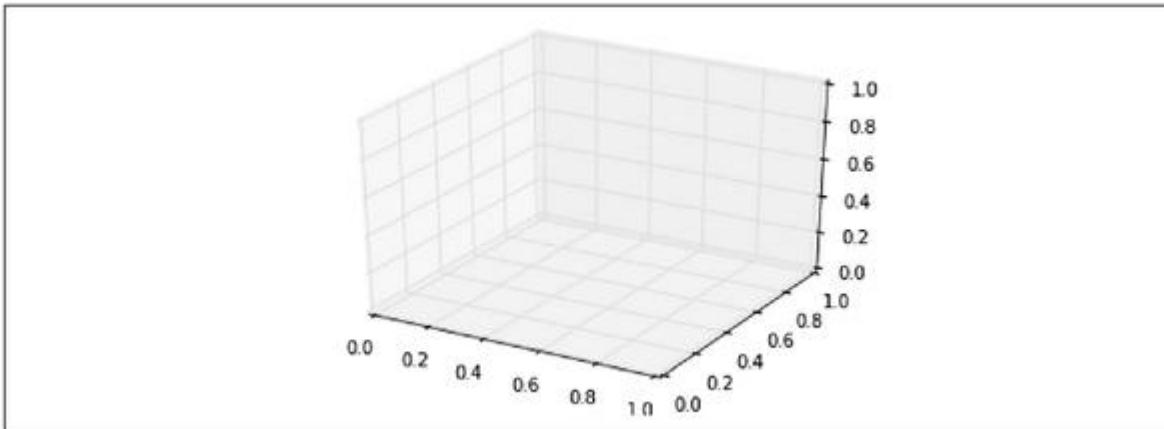


Figure 4.92 : Visualisation de trois axes sans données.

Une fois ce repère 3D en place, nous pouvons y faire apparaître différents types de tracés. Notez que le tracé en trois dimensions tire encore plus profit que celui en deux dimensions d'une utilisation en mode interactif dans le calepin. Je rappelle que l'on accède au mode interactif en spécifiant `%matplotlib notebook` au lieu de `%matplotlib inline` lors de l'exécution du code.

Tracé de points et de lignes en trois dimensions

Les tracés en 3D les plus simples sont ceux qui traitent des lignes ou des nuages de points à partir de triplets de valeurs (x , y , z). Les fonctions correspondantes portent d'ailleurs des noms proches de celles pour deux dimensions : `ax.plot3D()` et `ax.scatter3D()`. La syntaxe d'appel est quasiment la même. Vous pouvez donc vous reporter aux

sections de ce même chapitre qui ont décrit les tracés de lignes et de points. En guise d'exemple, produisons une spirale trigonométrique le long de laquelle nous déposons quelques points au hasard ([Figure 4.93](#)) :

In[4]:

```
ax = plt.axes(projection='3d')

# Données pour tracé 3D en lignes
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Données pour tracé 3D en nuages de points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 *
np.random.randn(100)
ydata = np.cos(zdata) + 0.1 *
np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata,
cmap='Greens');
```

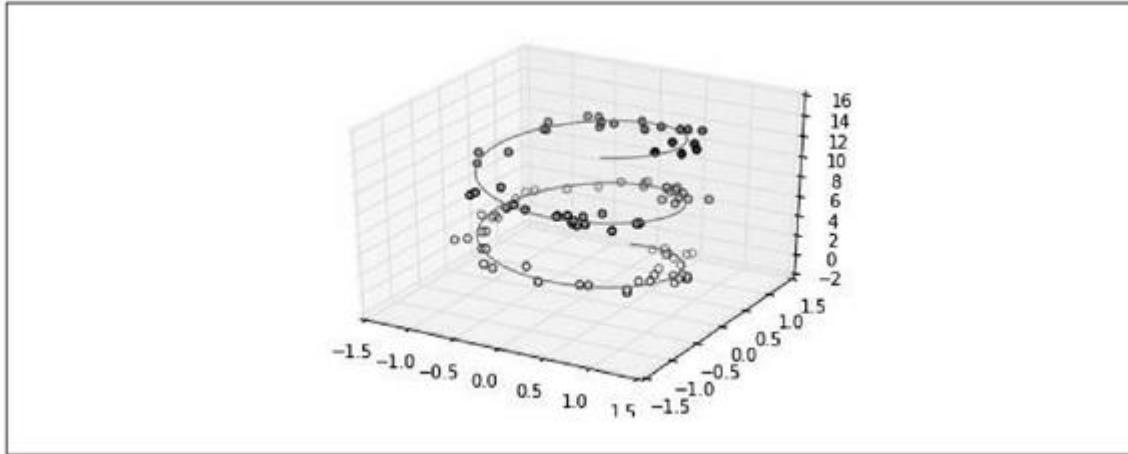


Figure 4.93 : Tracés de points et de lignes en trois dimensions.

La transparence des points est ajustée automatiquement pour conserver une bonne lisibilité du graphique. La vue statique du graphique en trois dimensions ne permet pas toujours de bien accéder à toutes les informations, et c'est pourquoi la vue interactive est recommandée pour pouvoir changer de point de vue.

Tracés de contours en trois dimensions

Nous avons déjà découvert les tracés de contours et de densités dans ce chapitre. Le module mplot3d permet de produire le même genre de tracé en trois dimensions au moyen de la fonction `ax.contour3D()`. Les données d'entrée doivent être des grilles régulières à deux dimensions, avec une évaluation de la donnée pour l'axe z à chaque point.

Voici comment créer un tracé de contours pour une fonction sinusoïdale ([Figure 4.94](#)) :

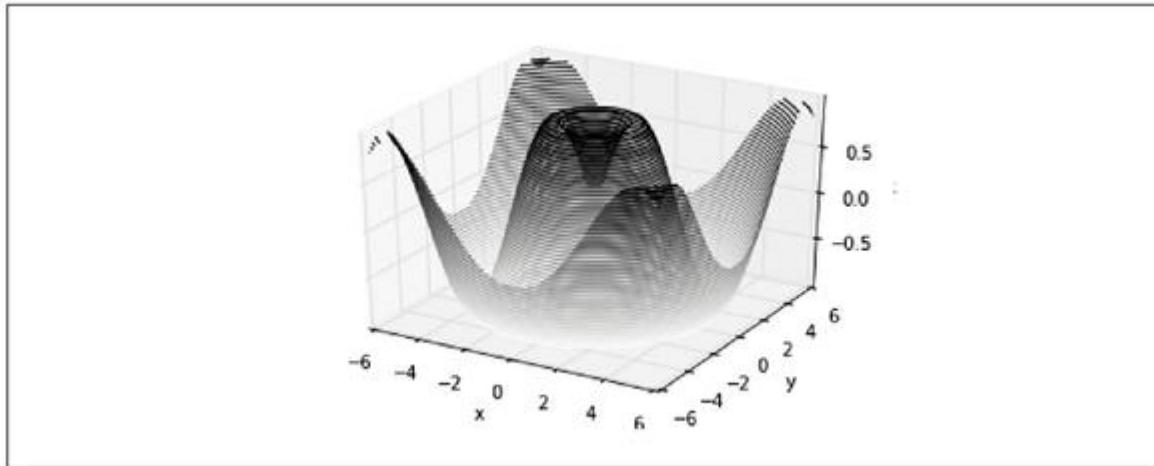
In[5]:

```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))

x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

In[6]:

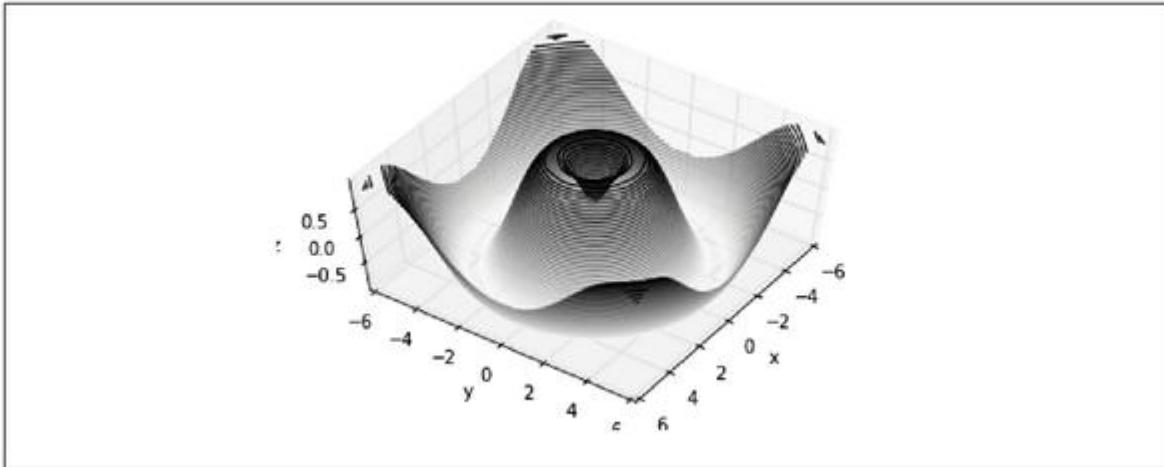
```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



[Figure 4.94](#) : Un tracé de contours en trois dimensions.

L'angle de vue sélectionné par défaut n'est pas toujours le meilleur, mais vous pouvez utiliser la méthode `view_init()` pour changer l'élévation et l'azimut. La [Figure 4.95](#) montre le résultat d'une élévation de 60 degrés au-dessus du plan x-y avec un azimut de 35 degrés, c'est-à-dire 35 degrés en sens antihoraire selon l'axe z :

```
In[7]:  
ax.view_init(60, 35)  
fig
```



[Figure 4.95](#) : Ajustement de l'angle de vue d'un tracé 3D.

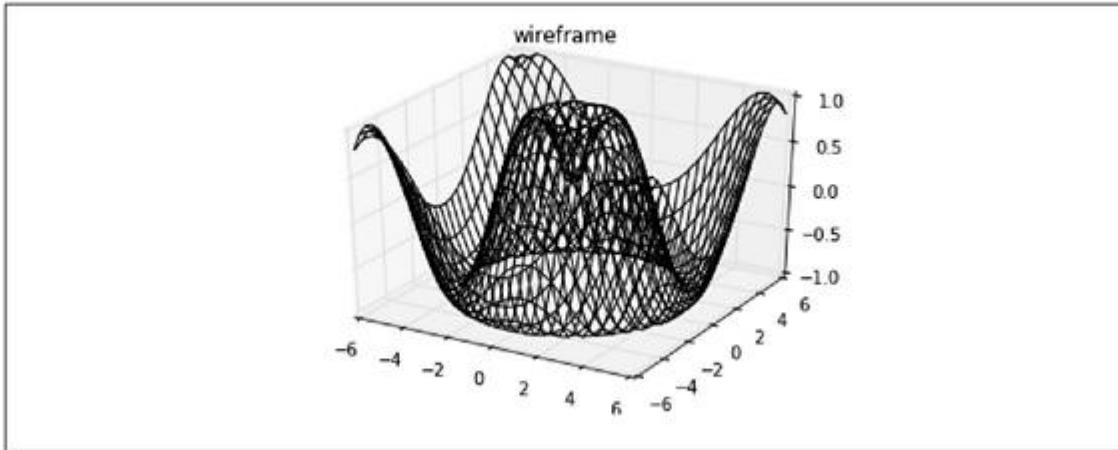
Je rappelle que ce genre de rotation peut être réalisé interactivement à la souris si vous utilisez le mode interactif de Matplotlib.

Tracés de fils de fer et de surfaces

Les données tabulaires sont correctement représentées dans des tracés filaires et de surfaces. Les valeurs sont distribuées sur la surface à trois dimensions, pour donner un résultat souvent très lisible. Voici d'abord un exemple de fil de fer ([Figure 4.96](#)) :

In[8] :

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```

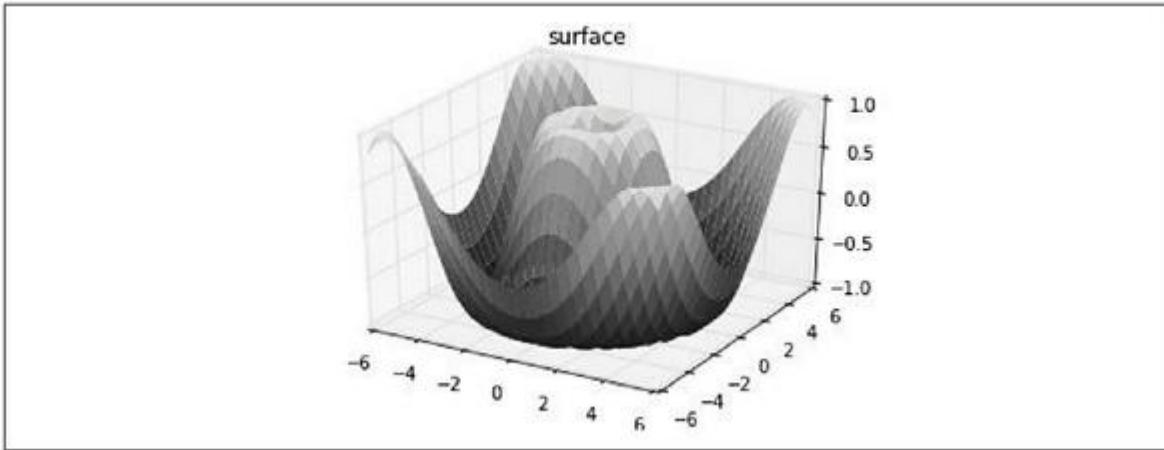


[Figure 4.96](#) : Un tracé fil de fer (wireframe).

Les tracés de surface sont très proches sauf que chaque face est remplie par des polygones. Vous pouvez spécifier un nuancier à associer à ces polygones de remplissage pour rendre plus visible la topologie de la surface ([Figure 4.97](#)) :

In[9] :

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
cmap='viridis', edgecolor='none')
ax.set_title('surface');
```



[Figure 4.97](#) : Tracé d'une surface en 3D.

Notez que le jeu de valeurs pour la surface doit être à deux dimensions. Voici un exemple utilisant un jeu de données polaire partiel. Lorsqu'il est utilisé avec `surface3D`, nous obtenons une tranche pour voir dans la fonction que nous visualisons ([Figure 4.98](#)) :

In[10]:

```
r = np.linspace(0, 6, 20)
theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi,
40)
r, theta = np.meshgrid(r, theta)

X = r * np.sin(theta)
Y = r * np.cos(theta)
Z = f(X, Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
```

```
cmap='viridis',  
edgecolor='none' );
```

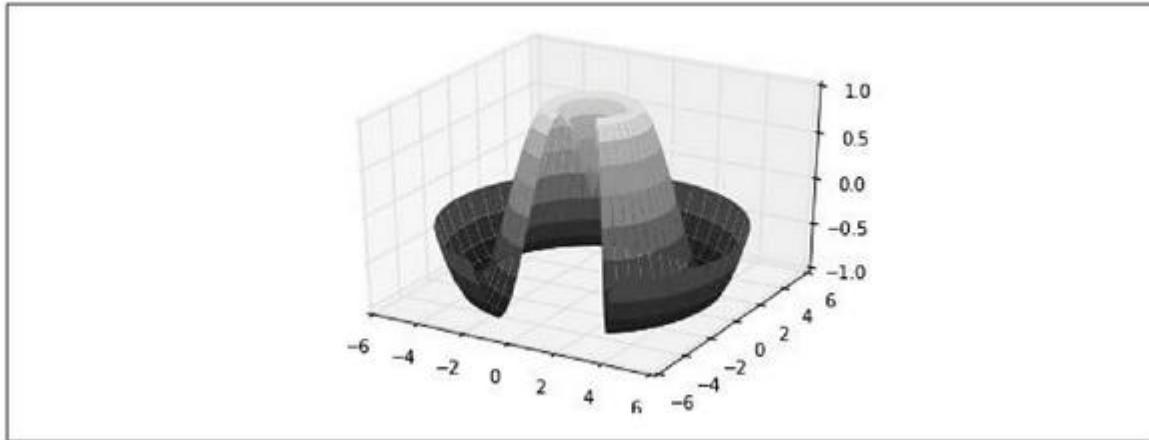


Figure 4.98 : Tracé de surface polaire.

Triangulation de surface

Le réseau de valeurs réparti de façon régulière qui alimente les routines précédentes ne convient pas dans certaines applications. Vous pouvez alors opter pour les tracés à triangulation. Au lieu d'un tracé régulier, dans un repère cartésien ou polaire, nous pouvons utiliser des données à répartition aléatoire :

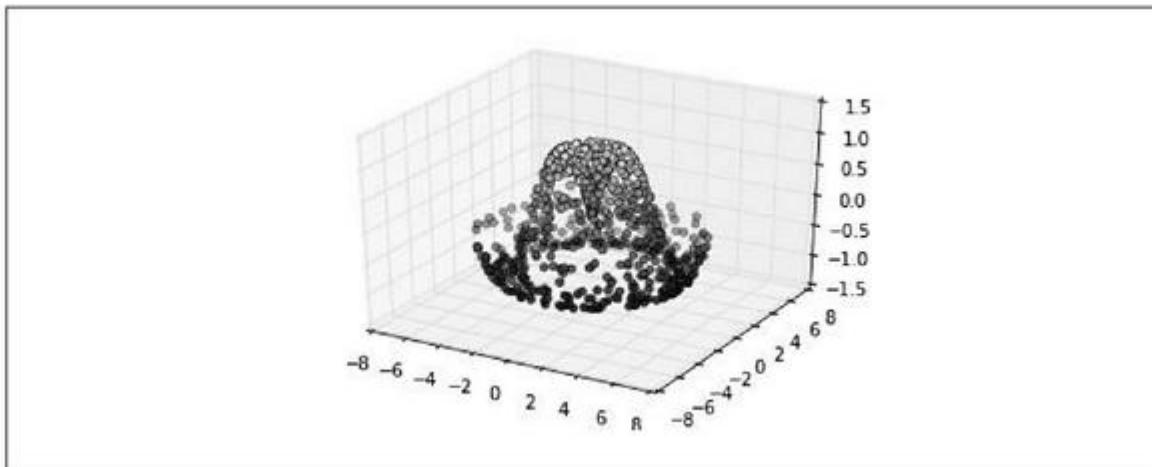
In[11]:

```
theta = 2 * np.pi * np.random.random(1000)  
r = 6 * np.random.random(1000)  
x = np.ravel(r * np.sin(theta))  
y = np.ravel(r * np.cos(theta))  
z = f(x, y)
```

Nous pouvons alors créer un tracé des points pour voir l'aspect de la surface à partir de laquelle nous échantillonnons ([Figure 4.99](#)) :

In[12]:

```
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='viridis',
linewidth=0.5);
```



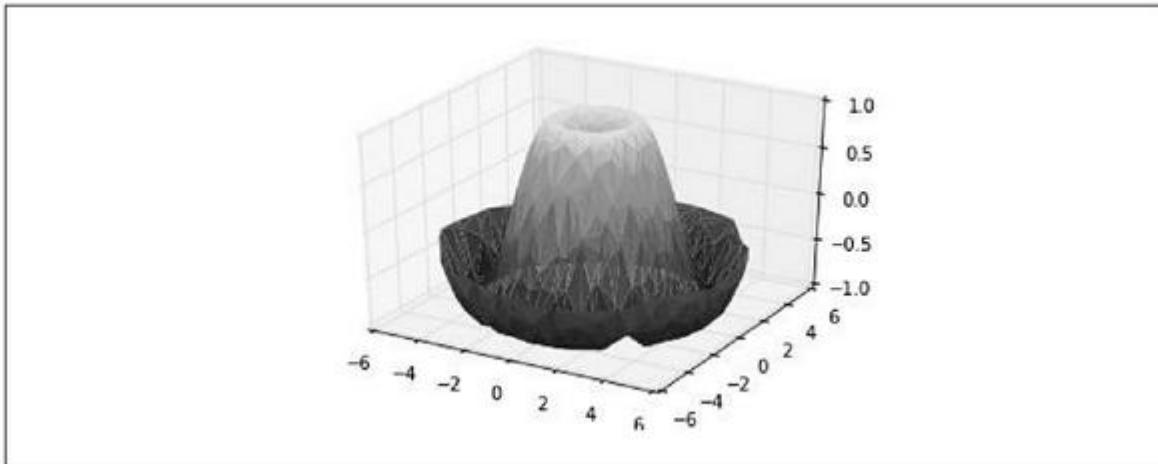
[Figure 4.99](#) : Une surface 3D échantillonnée.

Ce n'est qu'un résultat intermédiaire. Servons-nous de la fonction `ax.plot_trisurf()` qui crée une surface après avoir cherché un jeu de triangles entre les points voisins. La [Figure 4.100](#) montre le résultat ; souvenez-vous que `x`, `y` et `z` sont des tableaux à une dimension :

In[13]:

```
ax = plt.axes(projection='3d')
```

```
ax.plot_trisurf(x, y, z, cmap='viridis',  
edgecolor='none');
```



[Figure 4.100](#) : Tracé d'une surface par triangulation.

Le résultat n'est bien sûr pas aussi lisse qu'avec une grille de valeurs, mais la triangulation offre une souplesse pour créer certains tracés 3D. On peut par exemple créer une vue d'un ruban de Möbius en 3D, comme nous allons le faire dans l'exemple.

Exemple : visualisation d'un ruban de Möbius

Un ruban de Möbius peut se représenter avec une bande de papier collée après avoir tourné un des bouts d'un demi-tour. Malgré les apparences, ce genre de surface n'a qu'un côté ! Effectuons un rendu d'un tel objet avec les outils 3D de Matplotlib. Il faut d'abord s'intéresser à la création des paramètres. Puisqu'il s'agit d'un ruban à deux dimensions,

il nous faut deux dimensions initiales que nous appelons θ (thêta) qui va de 0 à 2π autour de la boucle, et w qui va de -1 à +1 sur la largeur du ruban :

In[14]:

```
theta = np.linspace(0, 2 * np.pi, 30)
w = np.linspace(-0.25, 0.25, 8)
w, theta = np.meshgrid(w, theta)
```

En partant de ce paramétrage, nous devons déterminer les positions en x, y, z du ruban.

En étudiant la situation, on comprend qu'il y a deux rotations : la première correspond à la position à chaque moment de la boucle par rapport à son centre vide, et cela correspond à notre variable θ . L'autre rotation est le vrillage du ruban selon son axe, et nous l'appellerons ϕ (ϕ). Un ruban de Möbius suppose de réaliser un vrillage d'un demi-tour pendant un tour de boucle complet, c'est-à-dire $\Delta\phi = \Delta\theta/2$.

In[15]: **phi = 0.5 * theta**

Un peu de trigonométrie va nous permettre de faire calculer les coordonnées en 3D qui sont imbriquées. Nous utilisons r comme distance par rapport au centre, ce qui permet de trouver les coordonnées x, y, z :

```
In[16]: # Rayon dans le plan x-y  
r = 1 + w * np.cos(phi)  
  
x = np.ravel(r * np.cos(theta))  
y = np.ravel(r * np.sin(theta))  
z = np.ravel(w * np.sin(phi))
```

Pour réaliser le tracé, nous devons vérifier que notre triangulation est correcte. La meilleure technique est de définir la triangulation dans le cadre de la paramétrisation sous-jacente. Nous pouvons ensuite laisser Matplotlib projeter la même triangulation dans l'espace en 3D du ruban de Möbius, ce qui est possible de cette façon ([Figure 4.101](#)) :

```
In[17]: # Triangulation dans le paramétrage  
sous-jacent  
from matplotlib.tri import Triangulation  
tri = Triangulation(np.ravel(w),  
np.ravel(theta))  
  
ax = plt.axes(projection='3d')  
ax.plot_trisurf(x, y, z,  
triangles=tri.triangles,  
cmap='viridis', linewidths=0.2);  
  
ax.set_xlim(-1, 1); ax.set_ylim(-1, 1);  
ax.set_zlim(-1, 1);
```

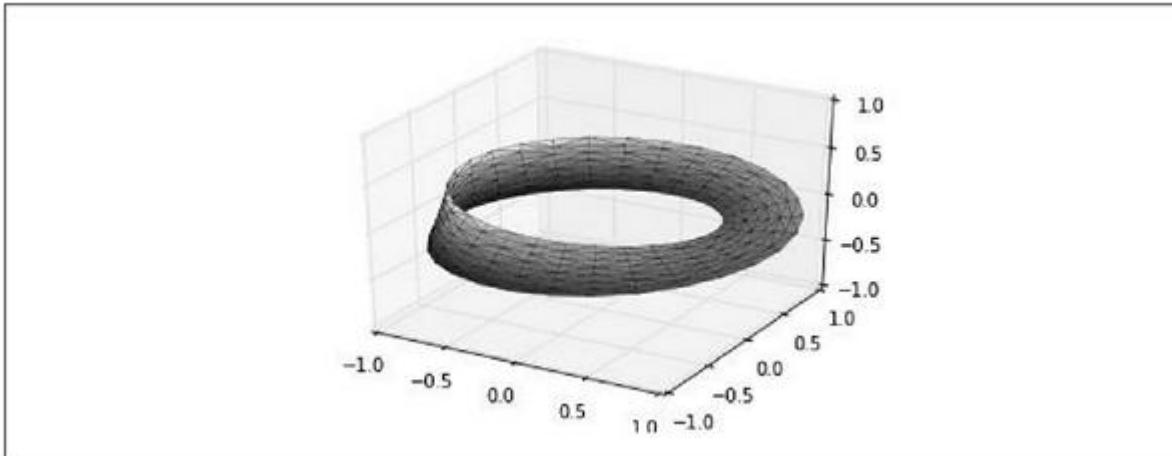


Figure 4.101 : Visualisation d'un ruban de Möbius en 3D.

Matplotlib permet donc de réaliser un certain nombre de visualisations en 3D grâce à ces différentes techniques.

4.15 : Données géographiques avec Basemap

La datalogie utilise fréquemment des données géographiques. Dans Matplotlib, l'outil principal pour cet emploi se nomme Basemap. C'est l'un des outils réunis dans l'espace de noms `mpl_toolkits`. Son utilisation peut sembler un peu primitive, et même les visualisations simples demandent un certain temps pour leur rendu. Des solutions plus modernes sont disponibles, par exemple `leaflet` ou l'API de Google Maps, et vous les privilégierez pour les travaux intensifs. Ceci dit, Basemap reste utile pour un usage général avec Python. Découvrons quelques exemples de visualisation de cartes qu'il permet.

Pour installer Basemap, si vous disposez de `conda`, vous pouvez utiliser la commande suivante pour télécharger et installer :

```
$ $ conda install -c conda-forge basemap
```

Il suffit d'ajouter deux directives d'import à notre en-tête de préparation habituel :

In[1] :

```
%matplotlib inline  
import os # Ajouté pour
```

```
fonctions système
import numpy as np
import matplotlib.pyplot as plt
os.environ['PROJ_LIB']=r'C:\X\DATALOGY\ANACONDA\
Library\share\basemap'
from mpl_toolkits.basemap import Basemap
```



(N.d.T.) La définition de la variable PROJ_LIB en dernière ligne ci-dessus s'est avérée nécessaire lorsque nous avons créé la version française. Faites un essai sans cette instruction. Ajoutez-la si l'exemple refuse de fonctionner.

Une fois notre outil Basemap installé et reconnu, nous pouvons obtenir immédiatement une vue géographique (la figure suivante réclame également le paquetage pillow pour Python 3) :

```
In[2]:
plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```

Nous revenons plus loin sur les paramètres de Basemap.

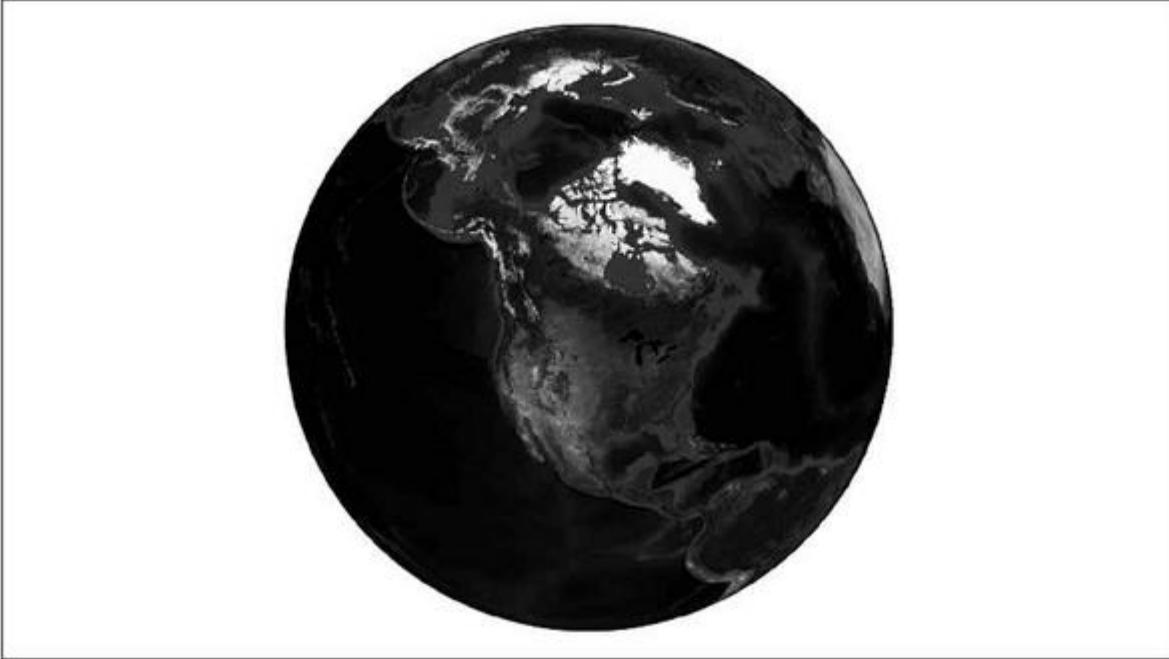


Figure 4.102 : Vue de « l'orange bleue ».

Apprenez d'abord que le globe ainsi visualisé n'est pas une image statique mais un système d'axes Matplotlib fonctionnel qui sait interpréter des coordonnées sphériques et permet donc d'ajouter des données sur le fond de carte ! Nous pouvons par exemple choisir une autre projection pour zoomer sur une région et positionner une ville, par exemple Seattle. Comme fond de carte, nous choisissons une image *etopo* qui montre la topographie sur les terres et au fond des mers ([Figure 4.103](#)) :

In[3] :

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
            width=8E6, height=8E6,
```

```
    lat_0=45, lon_0=-100, )  
m.etopo(scale=0.5, alpha=0.5)  
  
# Mappe (long, lat) vers (x, y) pour tracer  
x, y = m(-122.3, 47.6)  
plt.plot(x, y, 'ok', markersize=5)  
plt.text(x, y, 'Seattle', fontsize=12);
```

Cela vous donne un aperçu de ce qu'il est possible de faire avec quelques lignes de code Python. Découvrons maintenant les différents points de vue ou projections possibles, la gestion du fond de carte et le tracé des valeurs sur ce fond. Vous pourrez ainsi créer quasiment n'importe quelle visualisation.

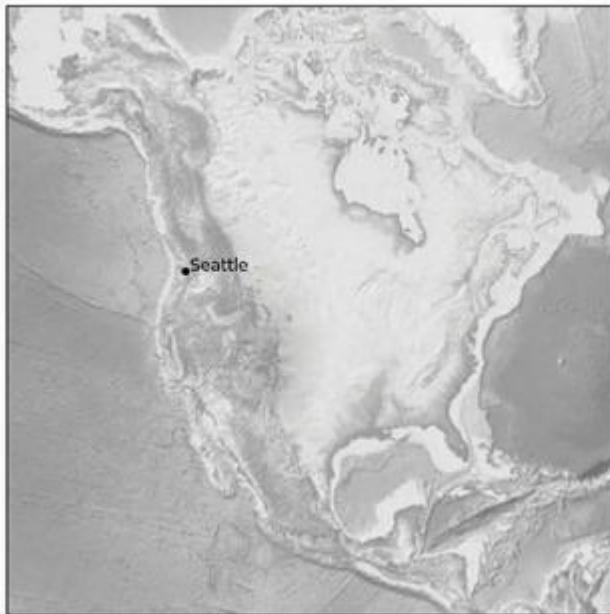


Figure 4.103 : Ajout de données et de labels sur un fond de carte.

Projections géographiques

La première chose à faire consiste à choisir une projection. Vous savez qu'il est impossible de projeter un objet sphérique tel que la Terre sur une surface plane sans provoquer des distorsions ou des déchirures. Différentes projections ont été inventées pour privilégier un aspect plutôt qu'un autre ; le choix est vaste ! C'est à vous de choisir votre projection selon que vous désiriez maintenir la fidélité des orientations, des surfaces, des distances ou des formes.

Le paquetage Basemap offre une douzaine de projections qui sont sélectionnables par un code. Découvrons les plus

usitées.

Nous définissons d'abord une petite fonction qui va redessiner la carte avec les lignes de latitude et de longitude :

In[4] :

```
from itertools import chain

def draw_map(m, scale=0.2):
    # Avec ombres de relief
    m.shadedrelief(scale=scale)

    # Latitudes et longitudes renvoyées dans un
    # dictionnaire
    lats = m.drawparallels(np.linspace(-90, 90,
    13))
    lons = m.drawmeridians(np.linspace(-180, 180,
    13))

    # Les clés contiennent les instances de
    plt.Line2D
    lat_lines = chain(*(tup[1][0] for tup in
    lats.items()))
    lon_lines = chain(*(tup[1][0] for tup in
    lons.items()))
    all_lines = chain(lat_lines, lon_lines)

    # On parcourt les lignes pour définir le
```

```
style
    for line in all_lines:
        line.set(linestyle='-', alpha=0.3,
color='w')
```

Projections cylindriques

Dans les projections cylindriques, les lignes de latitude et de longitude correspondent au quadrillage horizontal et vertical. Les régions proches de l'équateur ne sont pas très déformées, mais les régions polaires le sont énormément. L'espacement entre les parallèles varie d'une projection cylindrique à une autre, ce qui influe sur le niveau de distorsion au niveau des pôles. La [Figure 4.104](#) montre une projection cylindrique équidistante qui cherche à préserver les distances le long des méridiens. Vous disposez également de la projection Mercator (`projection='merc'`) et de la projection cylindrique à surfaces équivalentes (`projection='cea'`).

In[5]:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='cyl', resolution=None,
            llcrnrlat=-90, urcrnrlat=90,
            llcrnrlon=-180, urcrnrlon=180, )
draw_map(m)
```



[Figure 4.104](#) : Projection cylindrique à surfaces équivalentes.

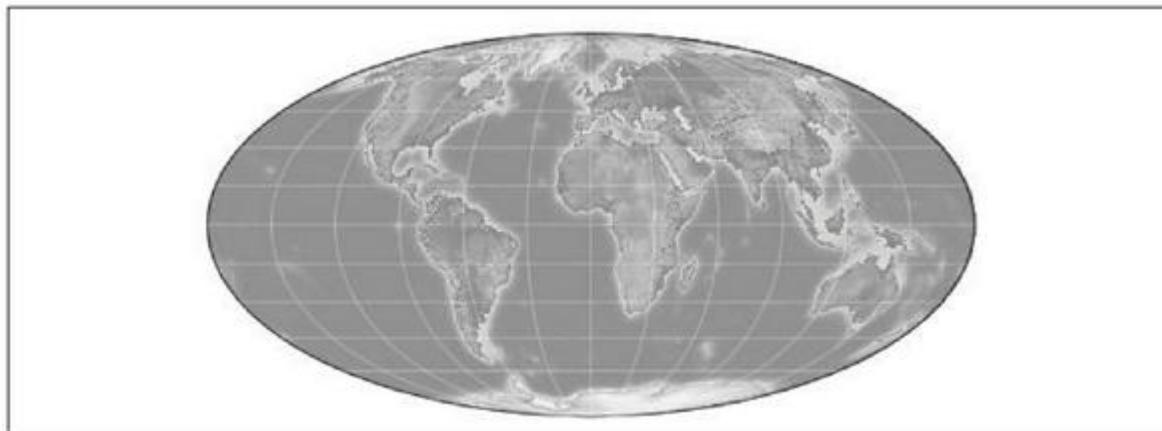
Les paramètres spécifiques à cette projection sont la latitude (lat) et la longitude (lon) pour d'une part l'angle inférieur gauche (llcrnr) et de l'autre l'angle supérieur droit (urcrnr), exprimés en degrés.

Projections pseudo-cylindriques

Dans ces projections, les lignes de longitudes que sont les méridiens doivent rester verticales par rapport à la surface, ce qui améliore les propriétés en se rapprochant des pôles. Une des variantes correspond à la projection de Mollweide (projection='moll') : tous les méridiens sont des arcs d'ellipse ([Figure 4.105](#)). Cette projection préserve les surfaces, ce qui entraîne des déformations près des pôles pour que les surfaces soient fidèlement représentées. D'autres variantes de cette projection sont la sinusoïde (projection='sinu') et la Robinson (projection='robin').

In[6]:

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None,
            lat_0=0, lon_0=0)
draw_map(m)
```



[Figure 4.105](#) : Projection de Mollweide.

Les paramètres complémentaires pour ces projections sont la latitude du centre (`lat_0`) et la longitude du centre (`lon_0`).

Projections en perspective

Ces projections sont construites après avoir choisi un point de vue, comme si vous vouliez photographier la Terre depuis l'espace (pour certains points de vue, ils sont situés en fait dans la Terre !). Ainsi la projection orthographique (`projection='ortho'`) montre le globe terrestre comme s'il était vu depuis très loin, ce qui ne peut donc montrer qu'un

hémisphère. D'autres projections en perspective sont la gnomonique (`projection='gnom'`) et la stéréographique (`projection='stere'`). Ce sont les deux plus fréquemment utilisées pour montrer des petites portions d'une carte.

Voici un exemple de projection orthographique ([Figure 4.106](#)) :

In[7]:

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
            lat_0=50, lon_0=0)
draw_map(m);
```



Figure 4.106 : Exemple de projection orthographique.

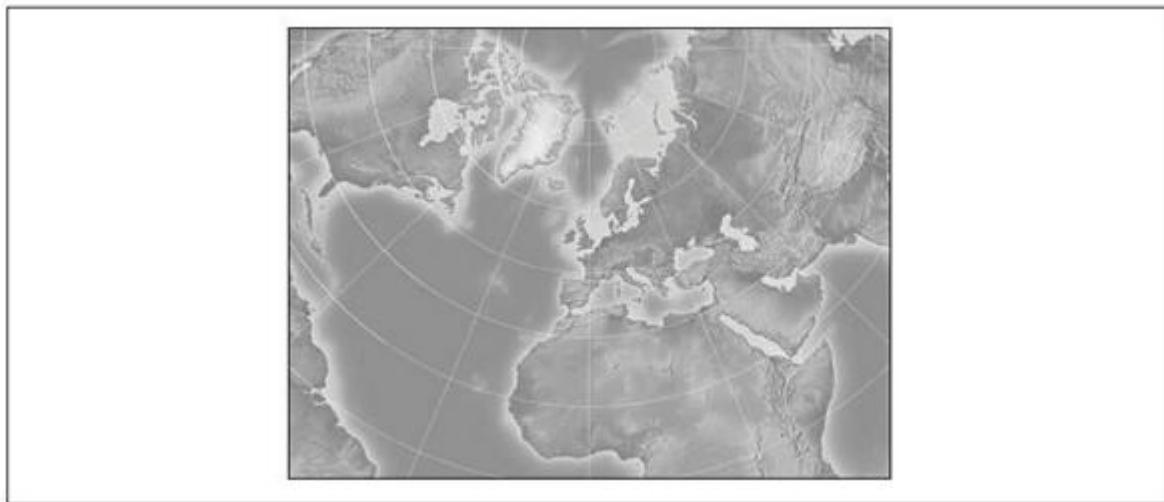
Projections coniques

Ce genre de projection construit la carte sur un cône qui est ensuite déroulé, ce qui privilégie certaines propriétés locales, en déformant énormément les régions qui en sont très éloignées. C'est le cas de la projection conique de Lambert (projection='lcc') qui nous a servi pour la carte des USA dans un autre exemple. Elle projette la carte sur un cône de telle façon que deux parallèles sont à des distances fidèlement représentées (ils correspondent aux deux paramètres `lat_1` et `lat_2` dans Basemap). L'échelle diminue entre ces deux repères et augmente à l'extérieur. D'autres variations de cette projection conique sont la projection

conique équidistante (`projection='eqdc'`) et la projection équivalente d'Albers (`projection='aea'`) ([Figure 4.107](#)). Comme les projections en perspective, les projections coniques sont bien adaptées pour montrer une vue partielle d'un globe.

In[8] :

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution=None,
lon_0=0,
        lat_0=50, lat_1=45,
lat_2=55, width=1.6E7, height=1.2E7)
draw_map(m)
```



[Figure 4.107](#) : Exemple de projection équivalente d'Albers.

Si vous prévoyez d'avoir souvent à réaliser des graphiques géographiques, je vous conseille de chercher d'autres projections disponibles, car chacune a ses avantages. Voyez

d'abord celles qui sont disponibles dans le paquetage Basemap (<http://matplotlib.org/basemap/users/mapsetup.html>). Si vous vous intéressez à ce domaine, vous allez découvrir toute une sous-culture de fanatiques de la visualisation et de la représentation, chacun argumentant pour sa projection préférée.

Tracé d'un fond de carte

Nous avons défini voici quelques pages une fonction nommée `draw_map()` qui utilisait les méthodes `bluemarble()` et `shadedrelief()` pour sur-imposer des images sur le fond de carte, ainsi que les méthodes `drawparallels()` et `drawmeridians()` pour ajouter les parallèles et les méridiens. Vous disposez dans Basemap de toute une palette de fonctions pour tracer les frontières physiques et politiques, les continents, les océans, les rivières ainsi que les états et régions. Voici une sélection de fonctions de tracé. Utilisez le système d'aide de IPython pour en savoir plus à leur sujet :

Limites physiques et maritimes

<code>drawcoastlines()</code>	Trace les côtes des continents.
<code>drawlsmask()</code>	Ajoute un masque entre terre et mer pour pouvoir ajouter une image sur l'une ou sur l'autre.

drawmapboundary()	Trace les frontières de la carte, avec la couleur de remplissage pour les océans.
drawrivers()	Trace les rivières.
fillcontinents()	Colorie les continents dans la couleur spécifiée, et les lacs avec une autre couleur éventuelle.

Frontières politiques

drawcountries()	Trace les limites des pays.
drawstates()	Trace les limites des États des USA.
drawcounties()	Trace les limites des comtés ou régions des USA.

Fonctions géométriques

drawgreatcircle()	Trace un cercle à partir de deux points.
drawparallels()	Trace des lignes de latitude constante.
drawmeridians()	Trace des lignes de longitude constante.
drawmapscale()	Ajoute une indication d'échelle linéaire.

Image du globe

bluemarble()	Image de bille bleue du projet de la NASA.
shadedrelief()	Ajoute une image avec ombrage de relief.
etopo()	Ajoute une image en relief de type etopo.
warpimage()	Ajoute une image fournie par l'utilisateur.

Pour toutes les fonctions concernant les frontières, vous devez sélectionner une résolution au moment de la création de l'image Basemap. Cela suppose d'utiliser le paramètre `resolution` de la classe Basemap qui peut prendre une valeur parmi cinq : 'c' (*crude*), 'l' (*low*), 'i' (*intermediate*), 'h' (*high*), 'f' (*full*) ou bien la valeur `None` si vous ne voulez pas de frontières. Ne demandez pas la plus haute qualité sans raison car la préparation de l'image pour une carte imposante peut prendre beaucoup de temps.



(N.d.T.) En standard, seules les deux résolutions les plus basses sont installées. Si vous avez besoin d'installer les autres, utilisez la commande suivante :

```
conda install -c conda-forge basemap-data-hires
```

L'exemple suivant compare deux résolutions sur les frontières d'un pays, en l'occurrence l'Écosse et en particulier l'île de Skye qui est située en 57,3° N, 6,2° W. Comme taille de notre carte, nous choisissons 90 × 120 kilomètres ([Figure 4.108](#)) :

In[9]:

```
fig, ax = plt.subplots(1, 2, figsize=(12, 8))
for i, res in enumerate(['l', 'h']):
    m = Basemap(projection='gnom', lat_0=57.3,
                lon_0=-6.2,
                width=90000, height=120000,
```

```

resolution=res, ax=ax[i])
    m.fillcontinents(color="#FFDDCC",
lake_color='#DDEEFF')
    m.drawmapboundary(fill_color="#DDEEFF")
    m.drawcoastlines()

ax[i].set_title("Résolution='{}'".format(res));

```



Figure 4.108 : Frontières d'une île en deux résolutions.

Vous constatez que la résolution la plus faible ne convient pas à cette échelle alors que l'autre devient acceptable. La basse résolution ne conviendrait qu'à une vision globale, mais elle serait beaucoup plus rapide à calculer, notamment pour la totalité de la planète ! Vous devrez peut-être tâtonner pour trouver la résolution la plus appropriée à

votre point de vue. Commencez toujours par une résolution faible et augmentez jusqu'à obtenir satisfaction.

Ajout des données sur une carte

Préparer un fond de carte est une chose, mais l'essentiel est bien de pouvoir ajouter des données sur ce fond. Pour les tracés et les textes simples, vous pouvez utiliser n'importe quelle fonction de la famille plt. Vous vous servez de l'instance de Basemap pour associer les coordonnées en latitude et en longitude à des coordonnées (x, y) puis pour tracer avec plt, comme nous l'avons vu dans l'exemple de la ville de Seattle.

Vous disposez en outre d'une série de fonctions spécifiques qui sont des méthodes d'une instance de la classe Basemap. Leur syntaxe ressemble beaucoup aux méthodes équivalentes de Matplotlib, mais elles disposent d'un paramètre booléen complémentaire nommé latlon. Lorsqu'il vaut True, vous pouvez transmettre des latitudes et des longitudes brutes à la méthode au lieu de coordonnées (x, y).

Voici une sélection de ces méthodes spécifiques aux cartes :

contour()/contourf() Trace des lignes de contours vides ou remplies.

imshow() Affiche une image.

pcolor()/pcolormesh()	Crée un tracé en pseudo-couleur avec un maillage régulier ou non.
plot()	Trace des lignes et/ou des marqueurs.
scatter()	Trace des points avec les marqueurs.
quiver()	Trace des vecteurs.
barbs()	Trace des symboles de force du vent (barbules).
drawgreatcircle()	Trace un grand cercle.

Nous utiliserons certaines de ces fonctions dans les exemples. Pour tout détail, reportez-vous comme d'habitude à la documentation en ligne (<http://matplotlib.org/basemap/>).

Exemple : villes de Californie

Lorsque nous avons produit la [Figure 4.47](#) de ce chapitre, nous avions vu comment contrôler la taille et la couleur des légendes pour ajouter des informations concernant la population des villes de Californie. Nous allons reprendre les mêmes données, mais en utilisant Basemap.

Nous commençons bien sûr par charger nos données :

```
In[10]:  
import pandas as pd  
cities =  
pd.read_csv('data/california_cities.csv')
```

```

# Extraction des données
lat = cities['latd'].values
lon = cities['longd'].values
population = cities['population_total'].values
area = cities['area_total_km2'].values

```

Nous devons ensuite préparer la projection, distribuer les données et ajouter une barre de couleurs et une légende ([Figure 4.109](#)) :

```

In[11]: # 1. Tracé du fond de carte
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution='h',
            lat_0=37.5, lon_0=-119,
            width=1E6, height=1.2E6)
m.shadedrelief()
m.drawcoastlines(color='gray')
m.drawcountries(color='gray')
m.drawstates(color='gray')

# 2. Ajout des données urbaines. La couleur code
# pour la population
# et le diamètre pour l'aire.
m.scatter(lon, lat, latlon=True,
          c=np.log10(population), s=area,
          cmap='Reds', alpha=0.5)

# 3. Création des barres de couleurs et légendes
plt.colorbar(label=r'$\log_{10}(\{ \rm population \})$')

```

```

plt.clim(3, 7)

# Construction de la légende avec points
factices
for a in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.5, s=a,
                label=str(a) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False,
           labelspacing=1, loc='lower left');

```

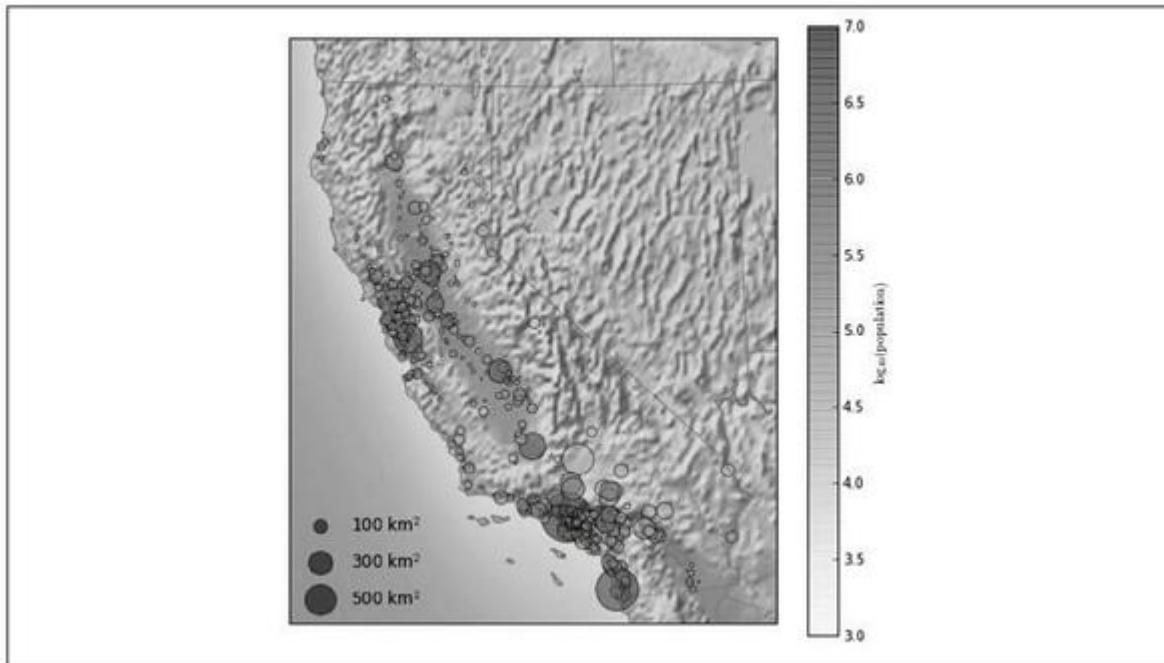


Figure 4.109 : Un tracé de points sur un fond de carte.

On voit ainsi clairement où les populations se sont installées : sur la côte dans les deux régions de Los Angeles et de San Francisco, et étalées le long de la vallée qui relie ces deux villes (la Silicon Valley). Il n'y a pratiquement personne dans les montagnes de l'arrière-pays.

Exemple : températures de surface

Passons à un exemple à plus grande échelle, celle d'un continent. Nous utilisons les données de l'anomalie climatique qui a frappé l'est des États-Unis en janvier 2014, anomalie du « polar vortex ». L'institut Goddard des études spatiales de la NASA (<http://data.giss.nasa.gov/>) est une excellente source de données climatiques. Nous pouvons y récupérer les données des températures GIS 250. Les commandes du shell pour le téléchargement doivent être changées sous Windows. Nous téléchargerons à l'heure actuelle (début 2022) un fichier pesant environ 50 Mo :

```
In[12]:  
# !curl -O  
http://data.giss.nasa.gov/pub/gistemp/gistemp250  
_GHCN.nc.gz  
# !gunzip gistemp250.nc.gz
```

Les données sont fournies dans le format NetCDF que Python peut exploiter au moyen de la librairie netCDF4, librairie que vous installez ainsi :

```
$ conda install -c conda-forge netcdf4
```

Nous procédons à la lecture des données :

In[13]:

```
from netCDF4 import Dataset  
data = Dataset('gistemp250.nc')
```

Le fichier contient de nombreuses mesures de température pour différentes dates. Nous sélectionnons l'index de la seule date qui nous intéresse, le 15 janvier 2014 :

In[14]:

```
from netCDF4 import date2index  
from datetime import datetime  
timeindex = date2index(datetime(2014, 1, 15),  
data.variables['time'])
```

Nous pouvons ensuite charger les données de latitude et de longitude ainsi que l'anomalie de température correspondant à l'index :

In[15]:

```
lat = data.variables['lat'][:]  
lon = data.variables['lon'][:]  
lon, lat = np.meshgrid(lon, lat)  
temp_anomaly = data.variables['tempanomaly'][timeindex]
```

Nous nous servons de la méthode `pcolormesh()` pour ajouter un maillage coloré des données. Nous nous limitons à l'Amérique du Nord en ajoutant une carte avec ombres de relief. Nous choisissons une barre de couleurs divergente,

avec une couleur neutre pour les valeurs nulles et deux couleurs contrastées pour les valeurs positives et négatives ([Figure 4.110](#)). Nous soulignons les traits de côtes pour un meilleur repérage :

```
In[16]:  
fig = plt.figure(figsize=(10, 8))  
m = Basemap(projection='lcc', resolution='c',  
            width=8E6, height=8E6,  
            lat_0=45, lon_0=-100,)  
  
m.shadedrelief(scale=0.5)  
m.pcolormesh(lon, lat, temp_anomaly,  
             latlon=True, cmap='RdBu_r')  
plt.clim(-8, 8)  
m.drawcoastlines(color='lightgray')  
  
plt.title('Anomalie de température de Janvier  
2014')  
plt.colorbar(label='anomalie (°C)');
```

Le résultat montre les anomalies de températures locales extrêmes au cours du mois concerné. La moitié est des USA a été beaucoup plus froide que d'habitude, alors que la moitié ouest ainsi que l'Alaska ont été beaucoup plus chaudes. Vous voyez le fond de carte dans les régions pour lesquelles il n'y a pas de températures mesurées.

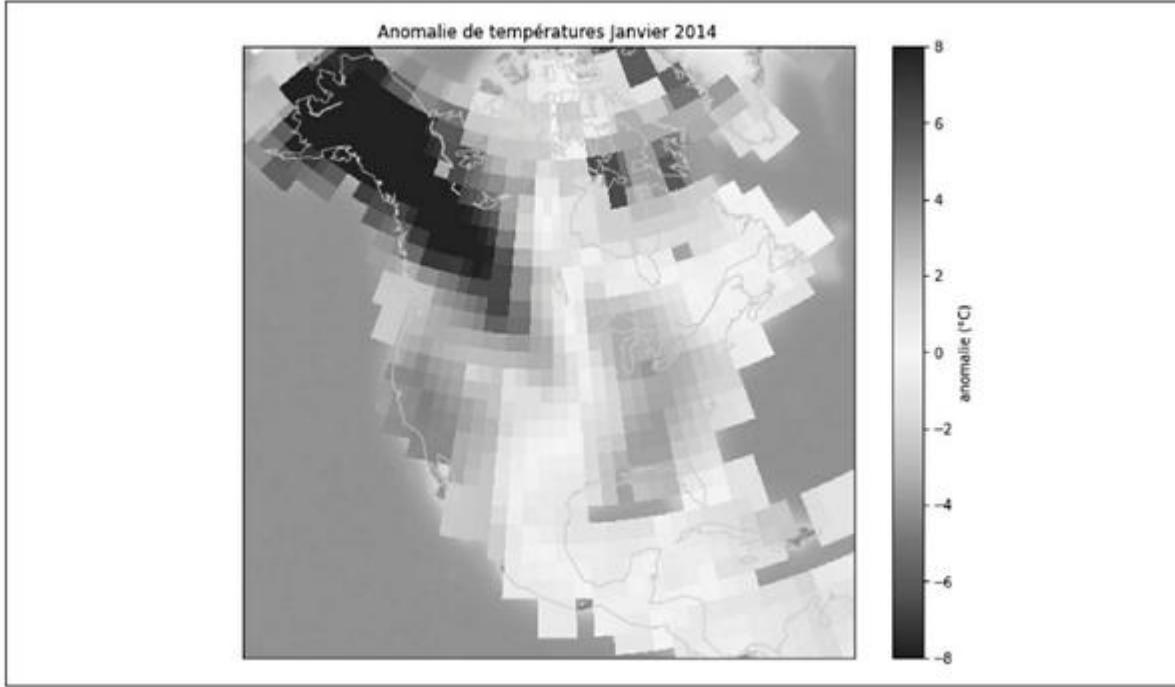


Figure 4.110 : Anomalies de températures de janvier 2014.

4.16 : Visualisation avec Seaborn

Matplotlib est une librairie de création de diagrammes très puissante et très répandue, mais même ces défenseurs reconnaissent qu'elle comporte quelques manques. Voici certaines des critiques que l'on entend au sujet de Matplotlib :

- Avant sa version 2.0, Matplotlib ne proposait pas de valeurs par défaut très judicieuses, car la librairie s'était basée sur la version de MATLAB de 1999.
- L'interface API de Matplotlib est relativement bas niveau. Les visualisations statistiques complexes sont accessibles, mais elles supposent souvent beaucoup de code de préparation.
- Matplotlib est apparu plus de dix ans avant les librairies Pandas, et n'a donc pas pu avoir été conçu pour exploiter les objets DataFrame de Pandas. Cela oblige à extraire le contenu de chaque objet Series pour les réunir dans le format approprié. Il serait très pratique de pouvoir utiliser directement les objets DataFrame dans les tracés.

Pour répondre à ces problèmes, nous disposons de Seaborn (<http://seaborn.pydata.org>). La librairie Seaborn propose des

choix par défaut de styles et de couleurs de tracé très efficaces ; elle définit des fonctions à haut niveau pour les besoins de tracé les plus communs en statistiques et s'intègre correctement avec la mécanique des objets DataFrames de Pandas.

Ceci dit, depuis la version 2.0, Matplotlib commence à corriger ses manques. L'apparition des outils de la famille plt.style (que nous avons décrite lorsque nous avons vu comment personnaliser Matplotlib dans ce chapitre) rend l'utilisation de Pandas plus facile.

Seaborn comparé à Matplotlib

Créons d'abord dans Matplotlib un tracé avec des valeurs aléatoires en utilisant les paramètres de format et de couleur standard. Nous commençons par les opérations d'importation :

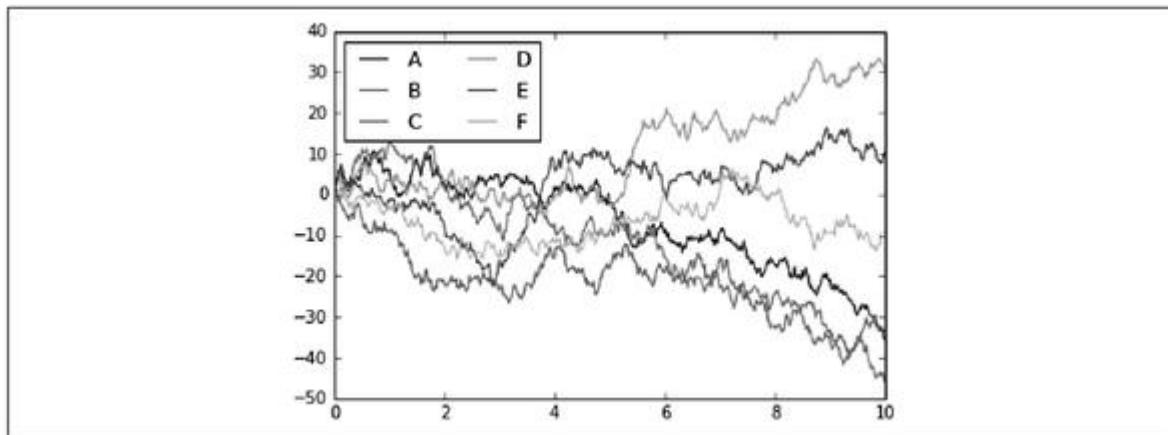
```
In[1]:  
import matplotlib.pyplot as plt  
plt.style.use('classic')  
%matplotlib inline  
import numpy as np  
import pandas as pd
```

Nous générerons quelques données au hasard :

```
In[2]: # Générer des données  
rng = np.random.RandomState(0)  
x = np.linspace(0, 10, 500)  
y = np.cumsum(rng.randn(500, 6), 0)
```

Nous pouvons maintenant demander notre tracé ([Figure 4.111](#)) :

```
In[3]: # Tracer avec valeurs par défaut de  
Matplotlib  
plt.plot(x, y)  
plt.legend('ABCDEF', ncol=2, loc='upper left');
```



[Figure 4.111](#) : Un exemple de tracé Matplotlib avec les valeurs par défaut.

Le résultat montre bien toutes les informations, mais au niveau esthétique, les choses pourraient être améliorées. L'aspect général semble même un peu daté par rapport à ce que l'on visualise des données au xxie siècle.

Voyons ce que propose Seaborn dans les mêmes conditions. Seaborn offre des routines à haut niveau mais peut également réutiliser et modifier les valeurs par défaut de Matplotlib. Autrement dit, même un script Matplotlib simple peut être amélioré. Nous choisissons le style au moyen de la méthode `set()` de Seaborn. Le nom abrégé de la librairie Seaborn est choisi par convention comme `sns` :

In[4] :

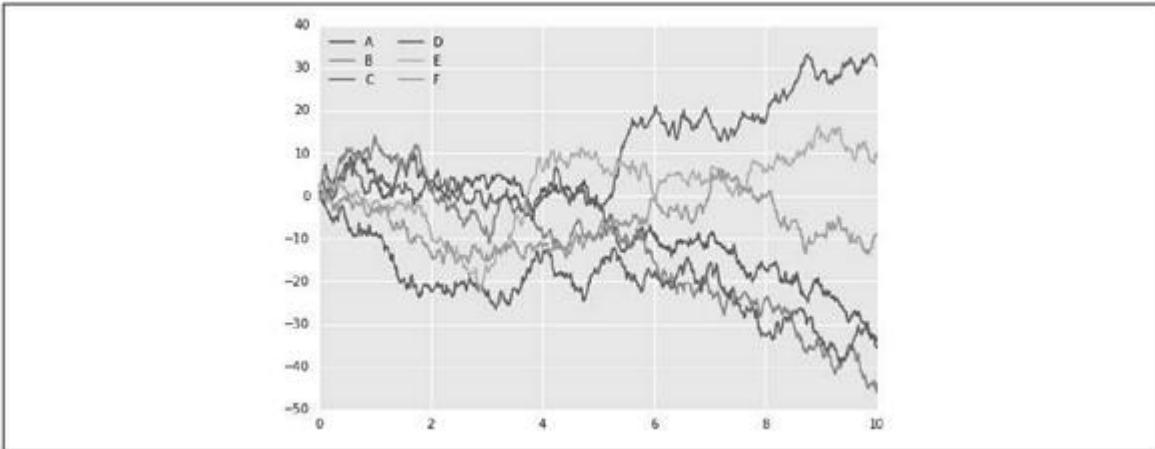
```
import seaborn as sns  
sns.set()
```

Nous exécutons les deux mêmes instructions pour créer le tracé ([Figure 4.112](#)) :

In[5] :

```
plt.plot(x, y)  
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

C'est beaucoup plus lisible, n'est-ce pas ? !



[Figure 4.112](#) : Aspect du tracé avec les styles par défaut de Seaborn.

Découverte des tracés Seaborn

L'idée maîtresse de Seaborn est de fournir des commandes à haut niveau pour pouvoir créer toute une gamme de tracés pour les explorations statistiques et même pour la recherche de modèles statistiques.

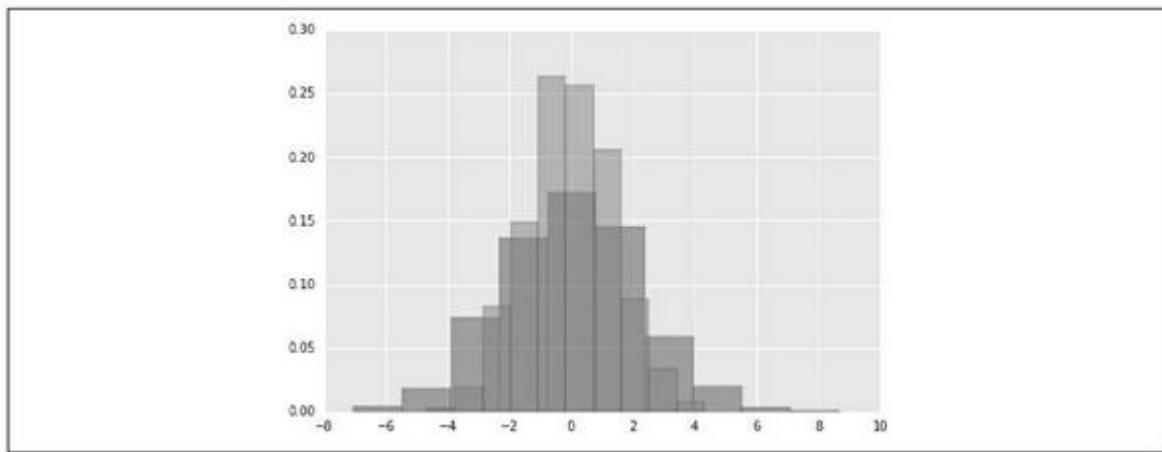
Découvrons quelques-uns des types de tracés et jeux de données disponibles dans Seaborn. Rappelons que tout ce que les pages suivantes vont montrer pourrait être réalisé avec des commandes Matplotlib élémentaires, et c'est justement ce que vous évite Seaborn. L'interface API de Seaborn est beaucoup plus simple d'emploi.

Histogrammes, KDE et densités

Lorsque vous visualisez des données statistiques, vous avez souvent besoin de tracer des histogrammes et la distribution

des variables après fusion. Nous avons vu qu'avec Matplotlib, l'opération était relativement simple ([Figure 4.113](#)) :

```
In[6]:  
data = np.random.multivariate_normal([0, 0],  
[[5, 2], [2, 2]], size=2000)  
data = pd.DataFrame(data, columns=['x', 'y'])  
  
for col in 'xy':  
    plt.hist(data[col], density=True, alpha=0.5)
```

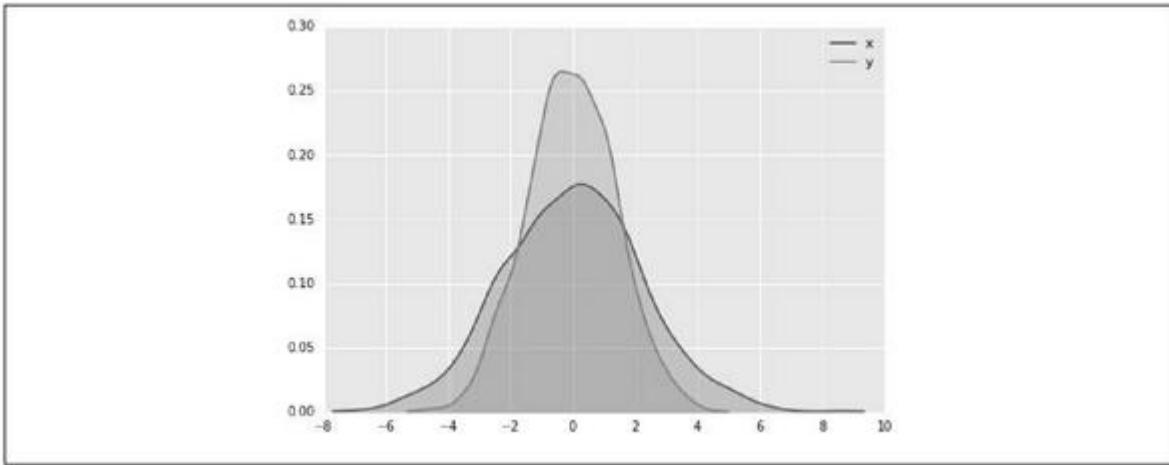


[Figure 4.113](#) : Histogramme de visualisation d'une distribution.

La librairie Seaborn permet d'obtenir une estimation lissée de la distribution grâce à une estimation de type kde, que vous obtenez en utilisant sns.kdeplot ([Figure 4.114](#)) :

```
In[7]:  
for col in 'xy':
```

```
sns.kdeplot(data[col], shade=True)
```



[Figure 4.114](#) : Estimation kde pour visualiser une distribution.

Vous pouvez même combiner un histogramme et une estimation kde avec histplot() ([Figure 4.115](#)) :

In[8] :

```
sns.histplot(data['x'])  
sns.histplot(data['y']);
```

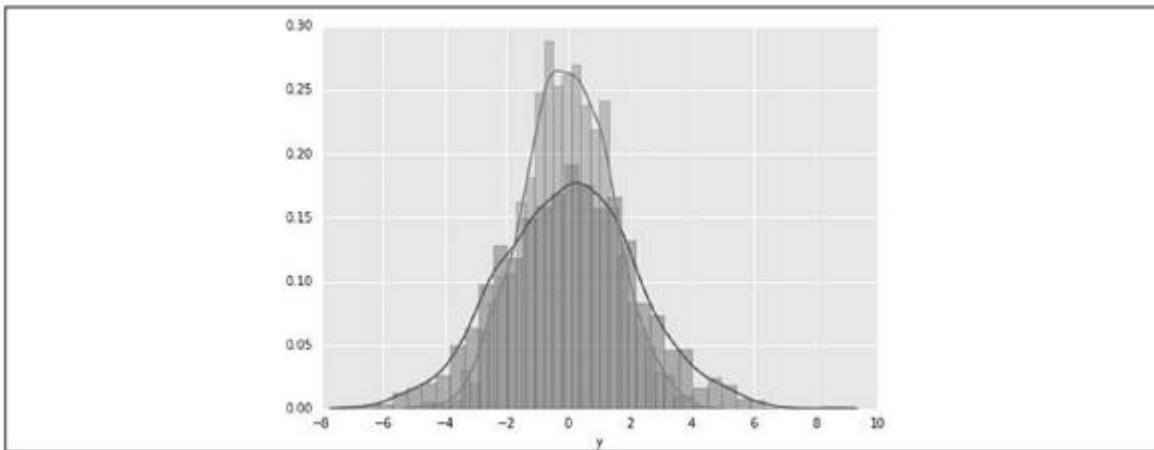


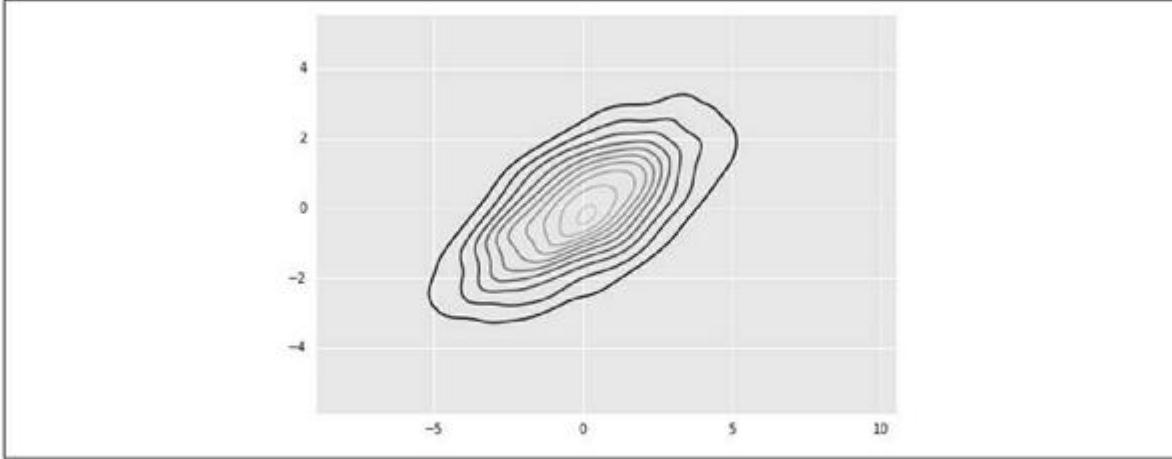
Figure 4.115 : Combinaison d'une densité kde et d'un histogramme.

Nous pouvons transmettre le jeu de données à deux dimensions à `kdeplot()` pour obtenir une visualisation en deux dimensions de nos données (Figure 4.116) :

```
In[9]: sns.kdeplot(data);
```



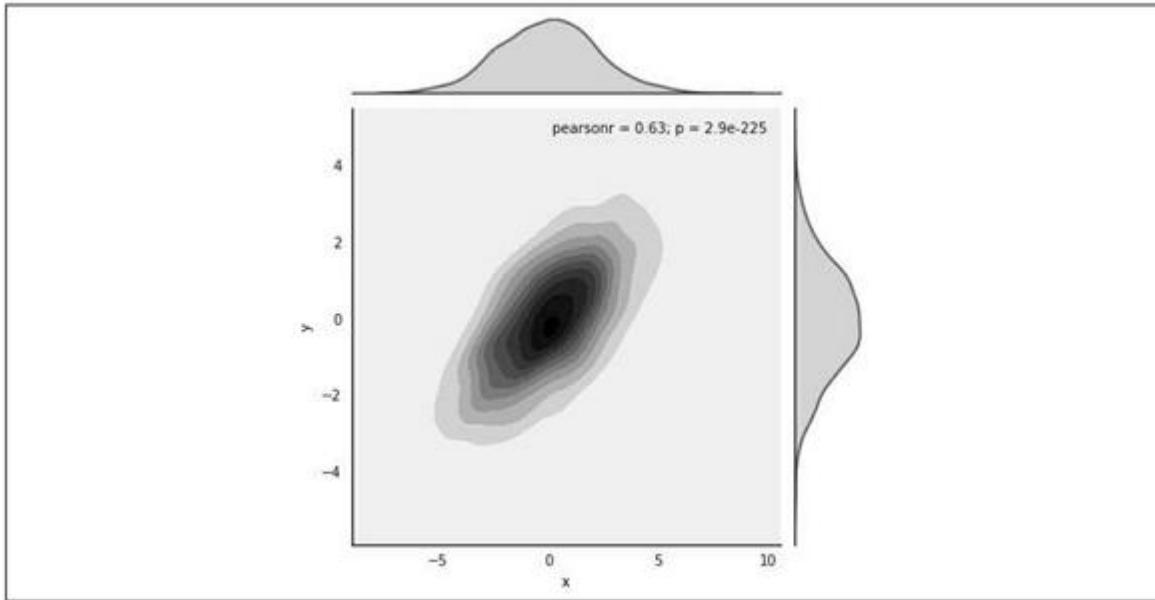
(N.d.T.) Il est possible que cet exemple ne fonctionne plus dans les versions plus récentes de Seaborn.



[Figure 4.116](#) : Visualisation kde à deux dimensions.

Pour visualiser en même temps la distribution jointe et les distributions marginales, nous utilisons sns.jointplot(). Nous choisissons un arrière-plan blanc ([Figure 4.117](#)) :

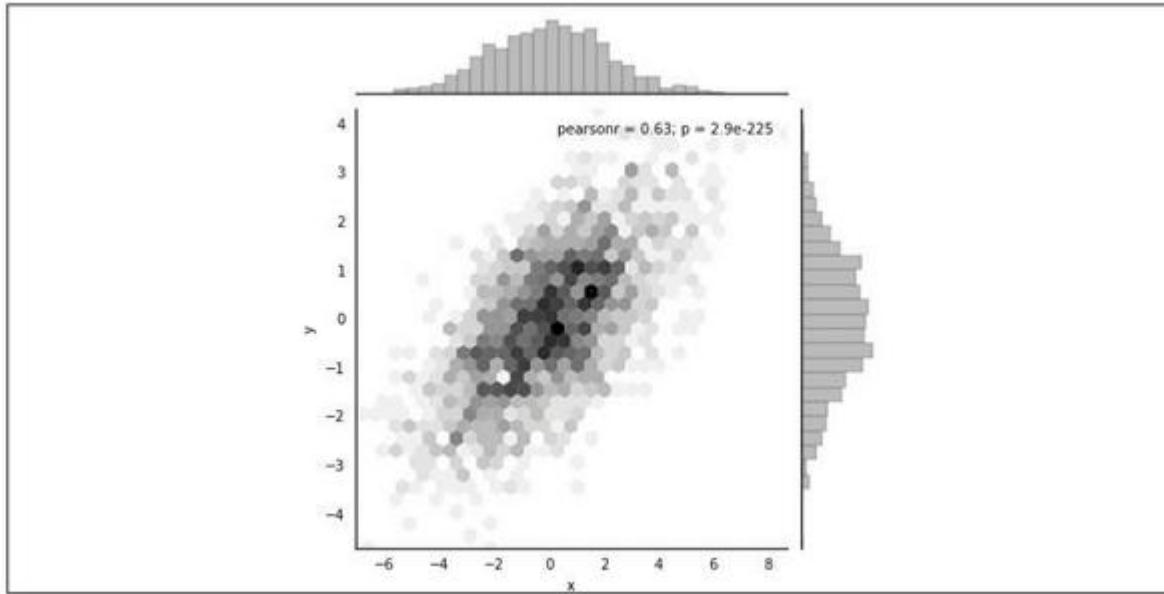
```
In[10]:  
with sns.axes_style('white'):  
    sns.jointplot("x", "y", data, kind='kde');
```



[Figure 4.117](#) : Tracé d'une distribution jointe avec estimation kde à deux dimensions.

La fonction `jointplot()` accepte d'autres paramètres. Nous pouvons par exemple demander un histogramme avec des valeurs à base hexagonale ([Figure 4.118](#)) :

```
In[11]:  
with sns.axes_style('white'):  
    sns.jointplot(x="x", y="y", data, kind='hex')
```



[Figure 4.118](#) : Tracé d'une distribution jointe avec représentation hexagonale des bacs.



(N.d.T.) Il est possible que cet exemple ne fonctionne plus dans les versions plus récentes de Seaborn.

Tracé de paires

Lorsque vous augmentez le nombre de dimensions d'un jeu de données pour lequel vous tracez des distributions jointes, vous obtenez des tracés de paires, qui sont tout à fait adaptés à la recherche de corrélations entre jeux de données à plusieurs dimensions. Vous comparez les paires de valeurs les unes aux autres.

Pour notre exemple, nous allons utiliser le jeu de données des iris qui donne les longueurs et largeurs des pétales et des sépales de trois espèces d'iris :

In[12]:

```
iris = sns.load_dataset("iris")
iris.head()
```

Out[12]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Il suffit d'utiliser `sns.pairplot()` pour accéder à une visualisation des relations entre les échantillons avec plusieurs dimensions ([Figure 4.119](#)) :

In[13]: `sns.pairplot(iris, hue='species', height=2.5);`

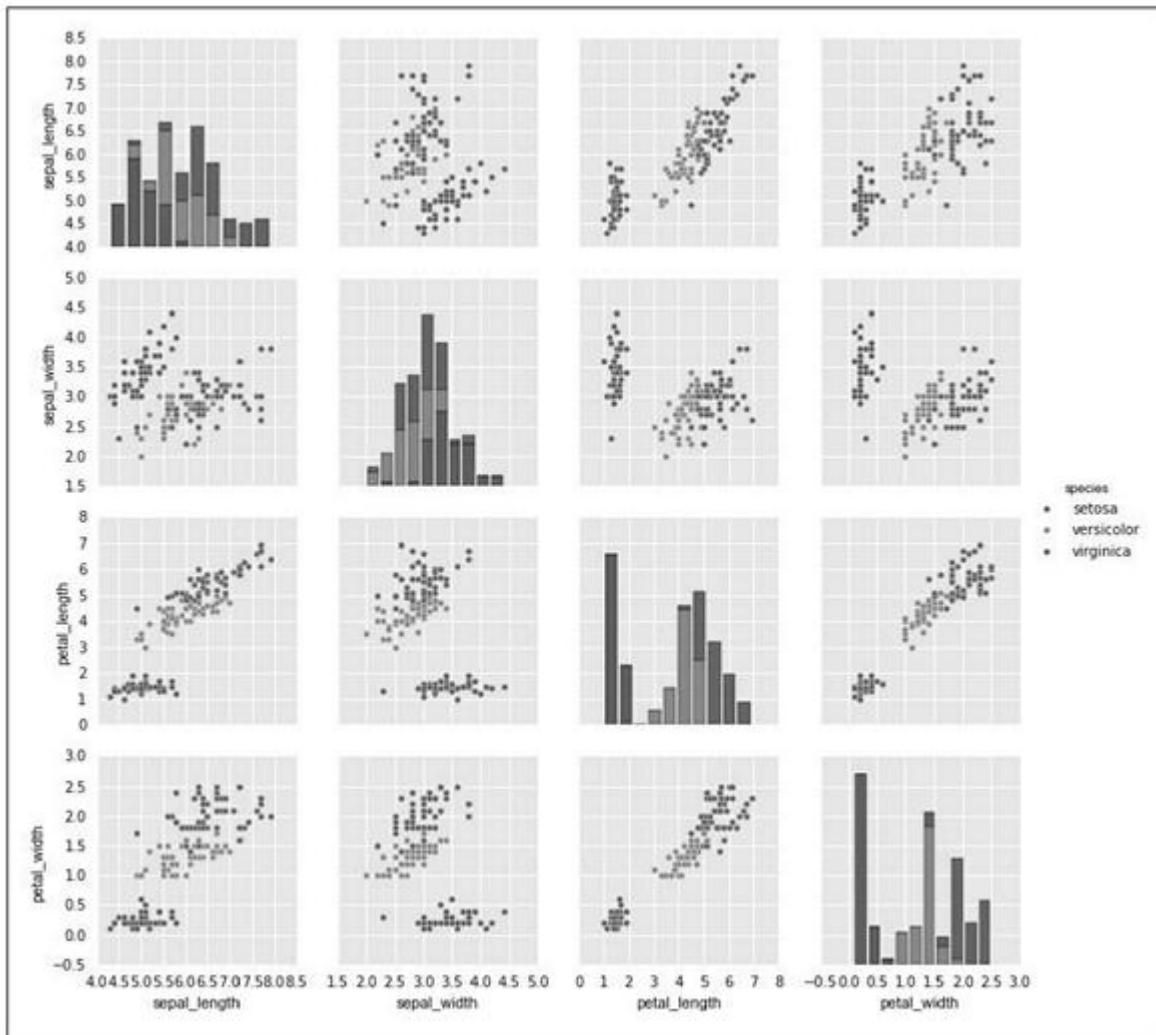


Figure 4.119 : Un tracé de paires montrant les relations entre quatre variables.

Histogrammes à facettes

Parfois, la meilleure représentation des données consiste à créer des sous-ensembles sous forme d'histogrammes, ce que permet facilement la méthode FacetGrid() de Seaborn (mefiez-vous des deux lettres capitales dans le nom). Voyons en guise d'exemple quels pourboires reçoivent les

serveurs d'un restaurant en fonction de différents critères ([Figure 4.120](#)) :

In[14]:

```
tips = sns.load_dataset('tips')
tips.head()
```

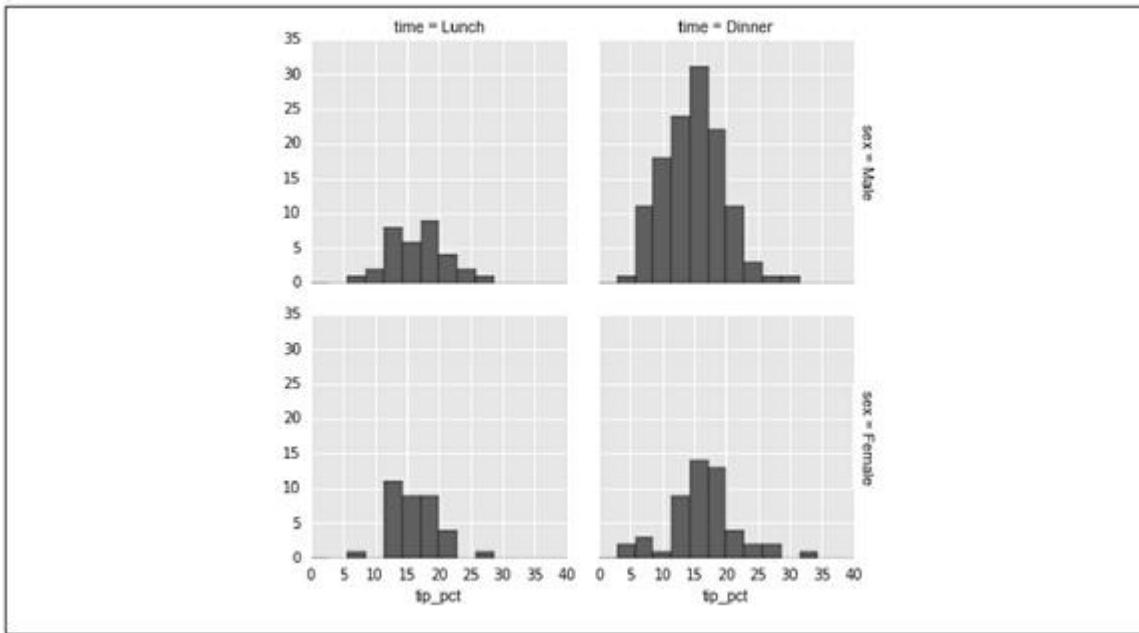
Out[14]:

	total_bill	tip	sex	smoker	day	time
size						
0	16.99	1.01	Female		No	Sun Dinner
1	10.34	1.66	Male		No	Sun Dinner
2	21.01	3.50	Male		No	Sun Dinner
3	23.68	3.31	Male		No	Sun Dinner
4	24.59	3.61	Female		No	Sun Dinner

In[15]:

```
tips['tip_pct'] = 100 * tips['tip'] /
tips['total_bill']
```

```
grid = sns.FacetGrid(tips, row="sex",
col="time", margin_titles=True)
grid.map(plt.hist, "tip_pct",
bins=np.linspace(0, 40, 15));
```



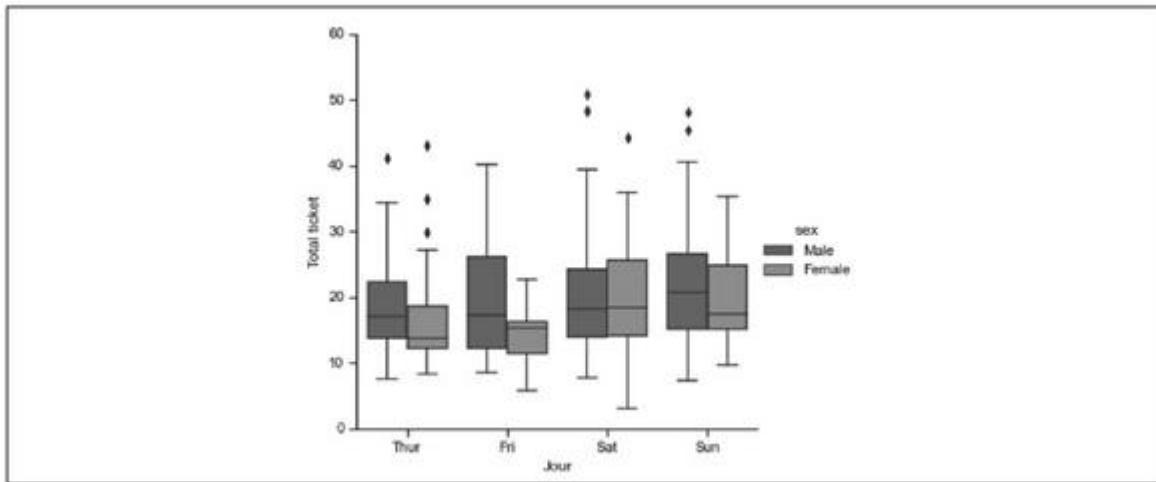
[Figure 4.120](#) : Un histogramme à quatre facettes.

Tracé de catégories

Pour visualiser la distribution d'une valeur servant de paramètre parmi un ensemble de bacs qui sont définis par un autre paramètre, vous pouvez utiliser les tracés catégoriels avec `catplot()` ([Figure 4.121](#)) :

In[6] :

```
with sns.axes_style(style='ticks'):
    g = sns.catplot(x="day", y="total_bill",
                     hue="sex",
                     data=tips, kind="box")
    g.set_axis_labels("Jour", "Total ticket");
```

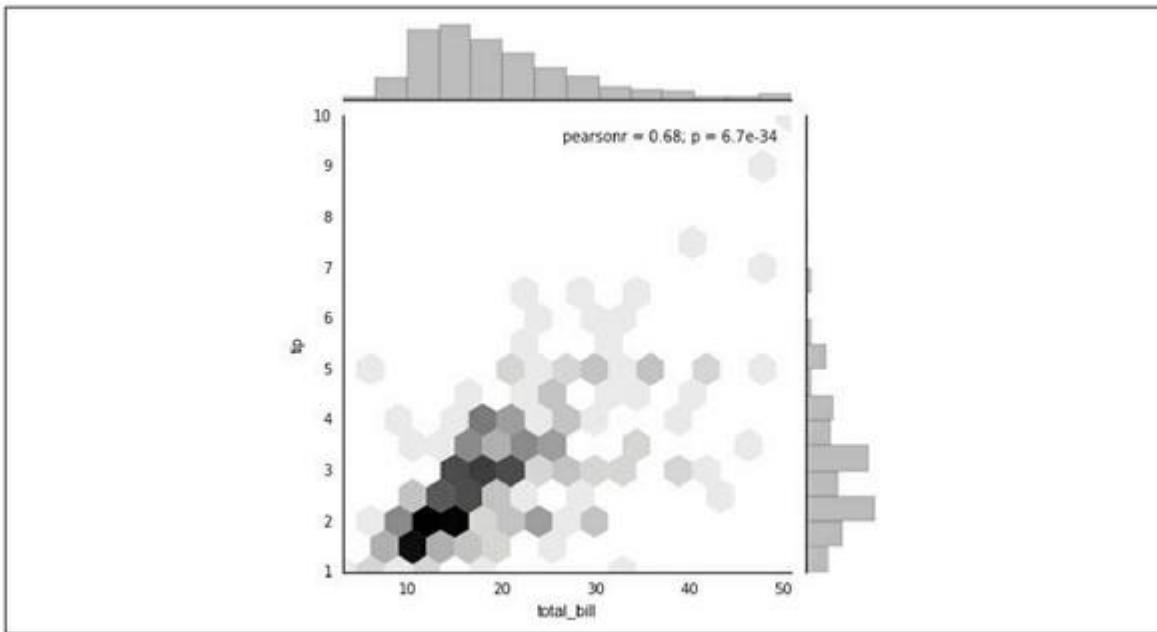


[Figure 4.121](#) : Un tracé catégoriel qui compare des distributions en fonction de critères tranchés.

Distributions jointes

Nous pouvons visualiser la corrélation de distribution entre deux jeux de données un peu comme nous l'avions fait avec le tracé de paires. Il suffit d'utiliser `sns.jointplot()` en spécifiant les distributions marginales associées ([Figure 4.122](#)) :

```
In[17]:  
with sns.axes_style('white'):  
    sns.jointplot("total_bill", "tip", data=tips,  
kind='hex')
```

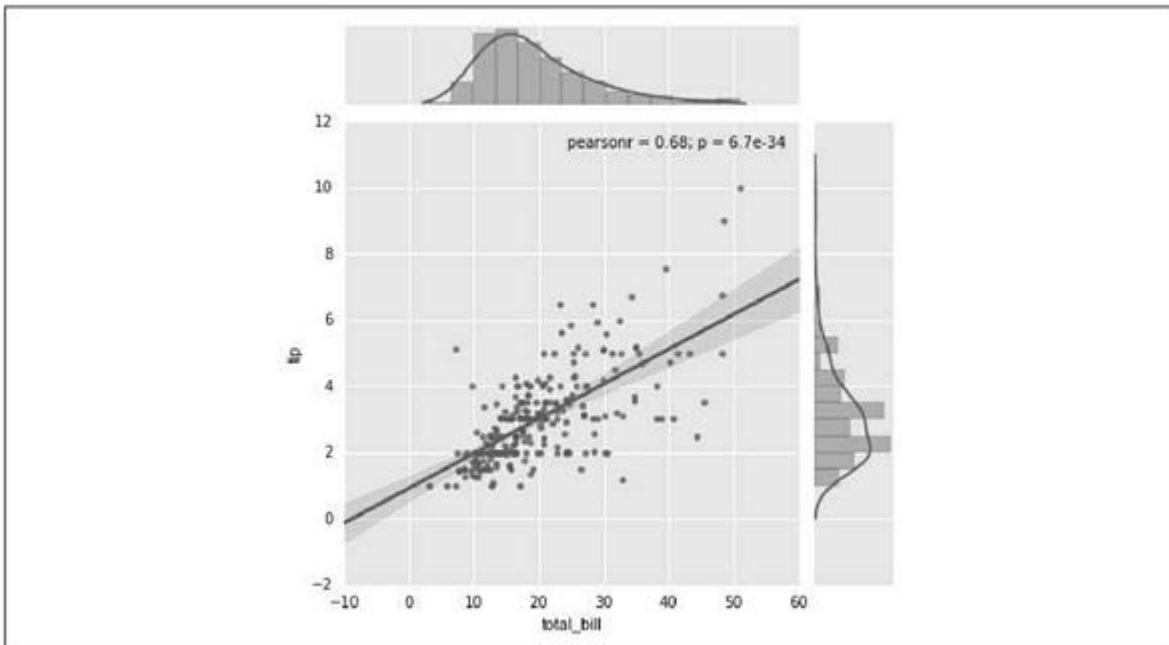


[Figure 4.122](#) : Tracé de distributions jointes.

La même méthode permet de demander une estimation de densité de noyau et une régression ([Figure 4.123](#)) :

In[18]:

```
sns.jointplot("total_bill", "tip", data=tips,  
kind='reg');
```



[Figure 4.123](#) : Tracé de distributions jointes avec régression fit.

Diagrammes en barres

Pour vos séries temporelles, vous pouvez utiliser `sns.catplot()` (qui remplace la méthode `catplot()`). Dans l'exemple de la [Figure 4.124](#), nous réutilisons les données des planètes que nous avions déjà utilisées lors des opérations de groupement du [Chapitre 3](#) :

In[19]:

```
planets = sns.load_dataset('planets')
planets.head()
```

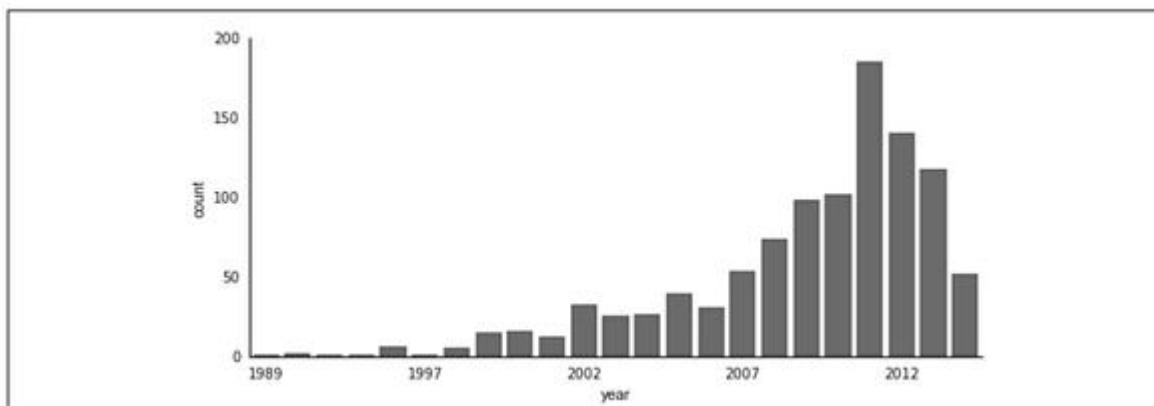
Out[19]:

	method	number	orbital_period	mass
distance				
Mercury	transit	1	88	3.3e+26
Venus	transit	2	225	4.8e+26
Earth	transit	3	365	5.9e+26
Mars	transit	4	687	6.4e+26
Jupiter	transit	5	12	1.9e+27
Saturn	transit	6	29	5.7e+26
Uranus	transit	7	84	8.7e+26
Neptune	transit	8	165	1.0e+27
Pluto	transit	9	248	1.3e+26
Haumea	transit	10	285	6.5e+26
Makemake	transit	11	500	1.4e+26
Eris	transit	12	550	2.1e+26

0	Radial Velocity	1	269.300	7.10
77.40	2006			
1	Radial Velocity	1	874.774	2.21
56.95	2008			
2	Radial Velocity	1	763.000	2.60
19.84	2011			
3	Radial Velocity	1	326.030	19.40
110.62	2007			
4	Radial Velocity	1	516.220	10.50
119.47	2009			

In[20]:

```
with sns.axes_style('white'):
    g = sns.catplot("year", data=planets,
aspect=2,
                 kind="count",
color='steelblue')
    g.set_xticklabels(step=5)
```



[Figure 4.124](#) : Un histogramme en tant que cas particulier de tracés factoriels.

Pour en savoir plus, nous pouvons observer la façon dont chaque planète est découverte, comme le montre la [Figure 4.125](#) :

In[21]:

```
with sns.axes_style('white'):
    g = sns.catplot("year", data=planets,
                    aspect=4.0, kind='count',
                    hue='method', order=range(2001, 2015))
    g.set_ylabels('Nombre de planètes
découvertes')
```

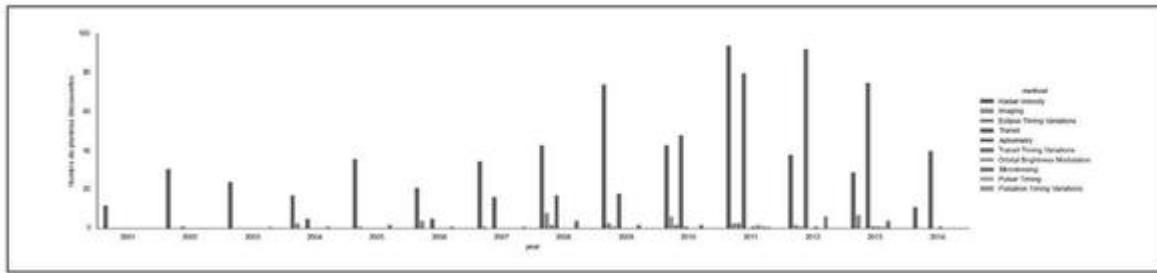


Figure 4.125 : Nombre de planètes découvertes par année et par type.



(N.d.T.) Voir le fichier des planches en couleurs pour une vue à grande échelle.

Pour de plus amples détails concernant Seaborn, voyez d'abord la documentation (<http://seaborn.pydata.org/>) et le tutoriel (<http://seaborn.pydata.org/>), sans oublier la galerie d'exemples (<http://seaborn.pydata.org/examples/index.html>).

Exemple : résultat d'un marathon

Nous allons profiter de Seaborn pour visualiser les temps d'un marathon afin de bien les analyser. Ces données ont été obtenues sur le Web et regroupées. J'ai ensuite supprimé les informations nominatives. Le fichier est disponible sur GitHub. Si vous avez besoin de récolter des données sur le Web, je vous conseille le livre de Ryan Mitchell *Scraping with Python*. Nous commençons par récupérer les données du Web pour les charger dans Pandas :

```
In[22]:  
# !curl -O  
https://raw.githubusercontent.com/jakevdp/marathon-data/  
# master/marathon-data.csv
```

```
In[23]:  
data = pd.read_csv('marathon-data.csv')  
data.head()
```

Out[23]:

	age	gender	split	final
0	33	M	01:05:38	02:08:51
1	32	M	01:06:26	02:09:28
2	31	M	01:06:49	02:10:42
3	38	M	01:06:16	02:13:45
4	31	M	01:06:32	02:13:59

Vous savez que par défaut, Pandas charge les colonnes contenant des temps sous forme de chaînes Python de type object. Nous le vérifions en demandant de connaître l'attribut dtype du

DataFrame :

```
In[24]:  
data.dtypes
```

```
Out[24]:  
age      int64
```

```
gender      object
split       object
final       object
dtype:      object
```

Nous devons corriger cela au moyen d'une fonction de conversion :

```
In[25]:  
import datetime  
def convert_time(s):  
    h, m, s = map(int, s.split(':'))  
    return datetime.timedelta(hours=h, minutes=m,  
seconds=s)  
  
data = pd.read_csv('marathon-data.csv',  
                   converters=  
{'split':convert_time, 'final':convert_time})  
data.head()
```

Out[25]:

	age	gender	split	final
0	33	M	0 days 01:05:38	0 days 02:08:51
1	32	M	0 days 01:06:26	0 days 02:09:28
2	31	M	0 days 01:06:49	0 days 02:10:42

```
3    38      M  0 days 01:06:16      0 days  
02:13:45  
4    31      M  0 days 01:06:32      0 days  
02:13:59
```

In[26]: `data.dtypes`

Out[26]:

```
age        int64  
gender     object  
split     timedelta64[ns]  
final     timedelta64[ns]  
dtype: object
```

C'est bien meilleur. Ajoutons des colonnes avec les temps en secondes, cela nous servira pour les outils de tracés Seaborn :

In[27]:

```
data['split_sec'] = data['split'].astype(int64)  
/ 1E9  
data['final_sec'] = data['final'].astype(int64)  
/ 1E9  
data.head()
```

Out[27]:

	age	gender	split	final
split_sec				
0	33	M	0 days 01:05:38	0 days 02:08:51
	3938.0		7731.0	

```
1   32      M  0 days 01:06:26  0 days 02:09:28
3986.0    7768.0
2   31      M  0 days 01:06:49  0 days 02:10:42
4009.0    7842.0
3   38      M  0 days 01:06:16  0 days 02:13:45
3976.0    8025.0
4   31      M  0 days 01:06:32  0 days 02:13:59
3992.0    8039.0
```

Demandons un tracé en jointure des données pour nous en faire une première idée ([Figure 4.126](#)) :

```
In[28]:
with sns.axes_style('white'):
    g = sns.jointplot("split_sec", "final_sec",
data, kind='hex')
    g.ax_joint.plot(np.linspace(4000, 16000),
                    np.linspace(8000, 32000),
'k')
```

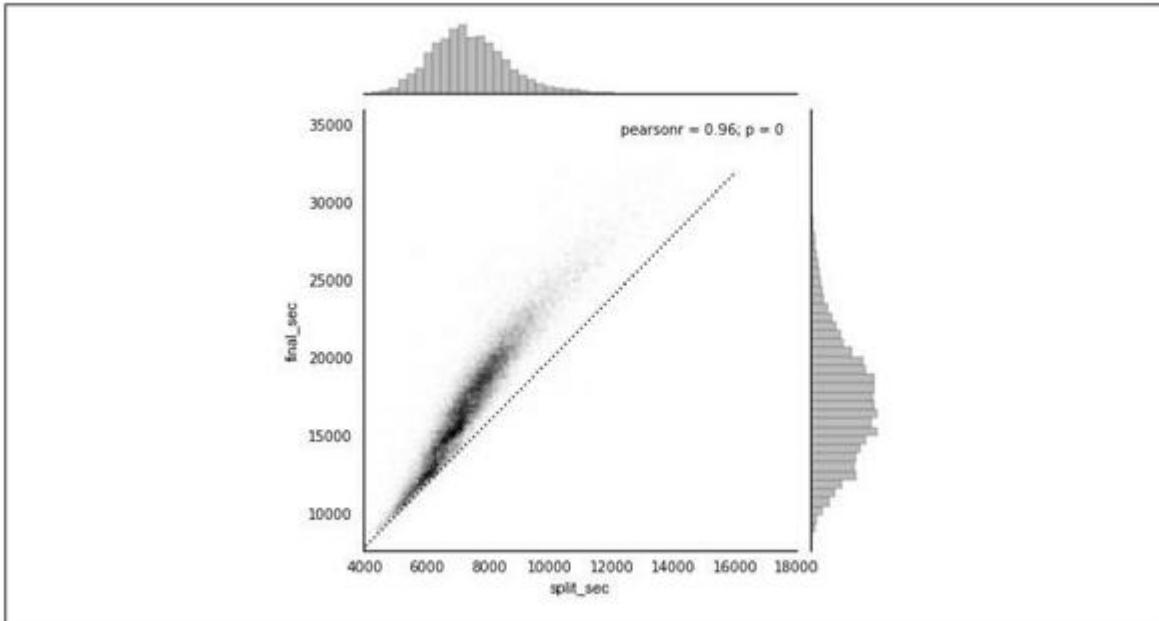


Figure 4.126 : Affichage des rapports entre les temps de la première moitié du marathon et le temps d'arrivée.

Dans cette figure, la ligne en pointillé indique où se situerait le temps d'une personne courant toujours à la même vitesse. L'essentiel des valeurs se situe au-dessus de cette ligne comme vous pouvez vous y attendre, parce que la plupart des gens ralentissent. Ceux qui accélèrent pendant la seconde moitié de la course ont un split négatif.

Ajoutons une autre colonne pour la fraction de temps split, qui permet de savoir à quel point chaque coureur a couru plus vite dans la seconde que dans la première moitié de sa course :

In[29]:

```
data['split_frac'] = 1 - 2 * data['split_sec'] /
```

```
data['final_sec']  
data.head()
```

Out[29]:

	age	gender	split	final
	split_sec	final_sec	split_frac	age_dec
0	33	M	0 days 01:05:38	0 days 02:08:51
3938.0		7731.0	-0.018756	30
1	32	M	0 days 01:06:26	0 days 02:09:28
3986.0		7768.0	-0.026262	30
2	31	M	0 days 01:06:49	0 days 02:10:42
4009.0		7842.0	-0.022443	30
3	38	M	0 days 01:06:16	0 days 02:13:45
3976.0		8025.0	0.009097	30
4	31	M	0 days 01:06:32	0 days 02:13:59
3992.0		8039.0	0.006842	30

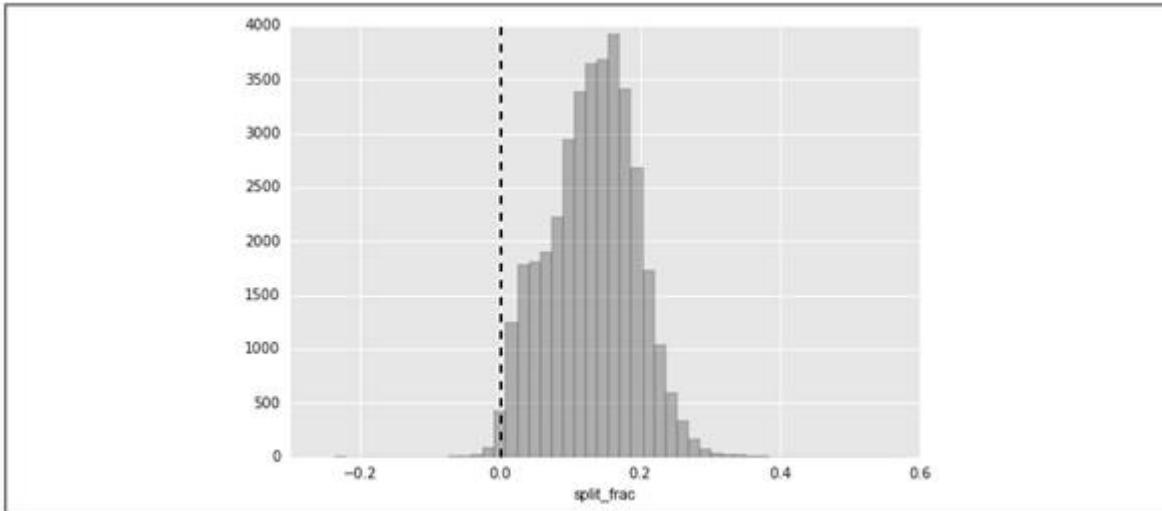
Lorsque la valeur dans cette nouvelle colonne `split_frac` est négative, c'est que la personne a accéléré dans la seconde moitié. Nous pouvons demander un tracé de distribution de cette valeur ([Figure 4.127](#)). La valeur 0.0 correspond à un coureur ayant pris autant de temps pour les deux moitiés de sa course. :

In[30]:

```
sns.distplot(data['split_frac'], kde=False);  
plt.axvline(0, color="k", linestyle="--");
```



(N.d.T.) Cette fonction `distplot()` sera prochainement remplacée par `displot()` et `histplot()`.



[Figure 4.127](#) : Distribution de l'avance ou du retard en seconde moitié par `split_frac`.

```
In[31]: sum(data.split_frac < 0)
```

```
Out[31]: 251
```

Parmi à peu près 40 000 coureurs, moins de 1 % environ ont accéléré pendant la seconde moitié (ont eu un split négatif).

Voyons s'il y a une corrélation entre cette fraction en split et d'autres variables. Nous allons nous servir d'un objet `PairGrid` qui sert à tracer toute une gamme d'estimations de corrélations ([Figure 4.128](#)) :

```
In[32]:
```

```
g = sns.PairGrid(data, vars=['age', 'split_sec',
```

```

'final_sec',
'split_frac'],
hue='gender', palette='RdBu_r')
g.map(plt.scatter, alpha=0.8)
g.add_legend();

```

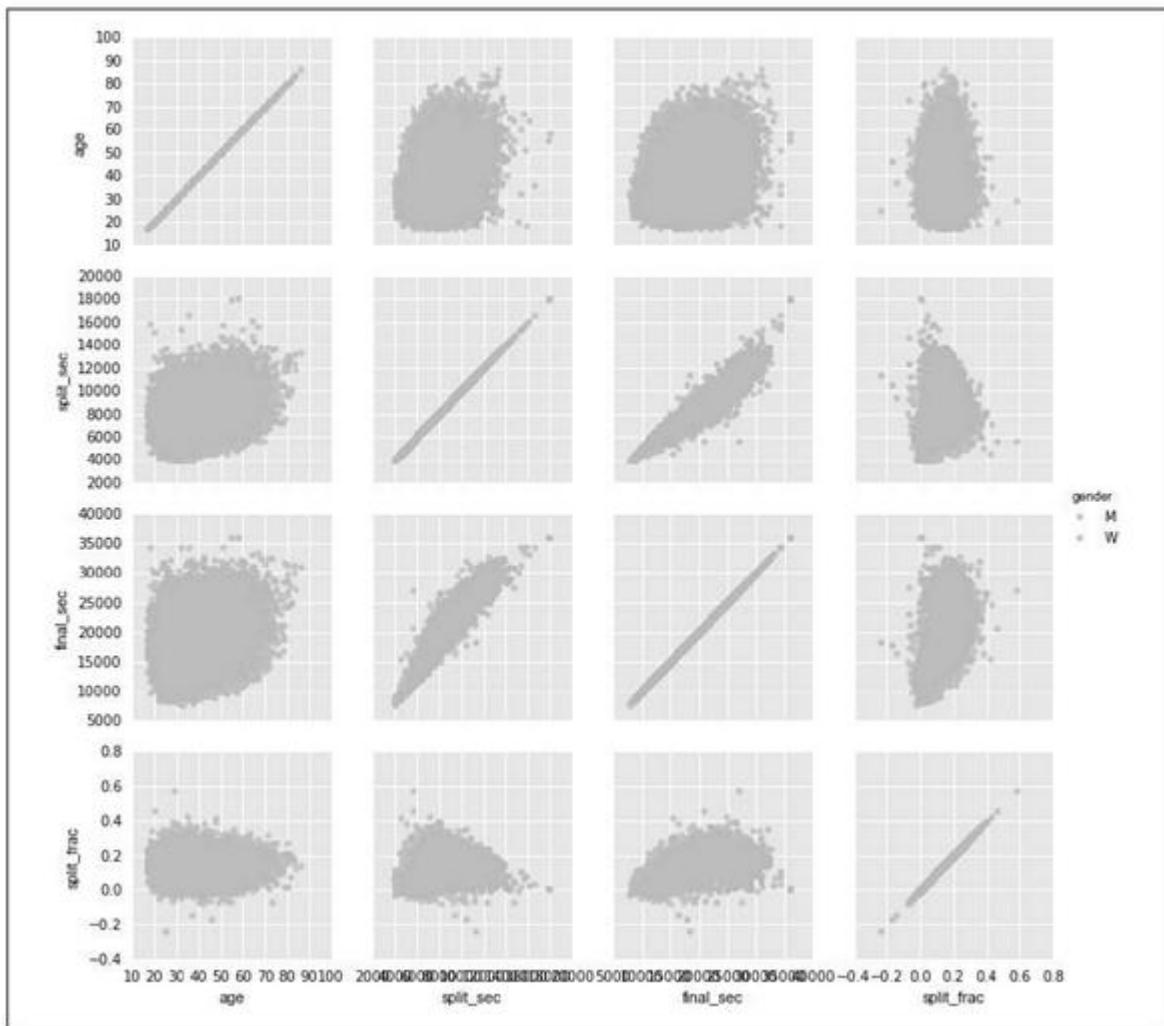


Figure 4.128 : Relations entre quantités des données du marathon.

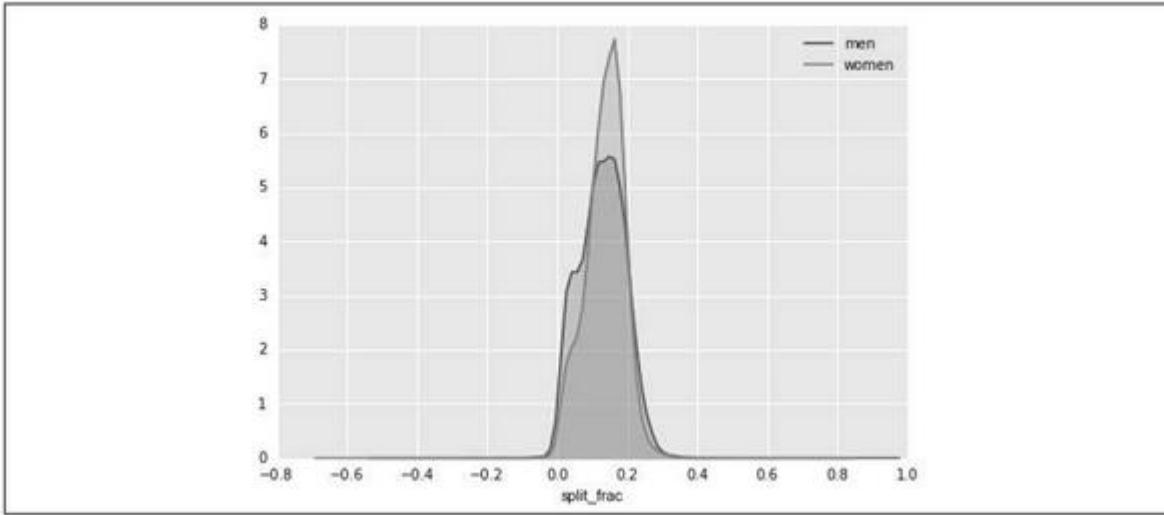
Notre critère split ne semble pas être particulièrement corrélé avec l'âge, mais il l'est avec le temps final : les coureurs les plus rapides ont tendance à bien équilibrer leurs

deux moitiés de course. (Nous constatons au passage que la librairie Seaborn ne résout pas les soucis de chevauchement de labels d'axes de Matplotlib. Nous pouvons cependant réduire ces soucis en nous servant des techniques présentées dans la partie sur la personnalisation des graduations de ce même chapitre.)

La différence entre hommes et femmes est intéressante. Visualisons le critère de fraction `split_frac` pour les deux groupes ([Figure 4.129](#)) :

In[33]:

```
sns.kdeplot(data.split_frac[data.gender=='M'],
label='men', shade=True)
sns.kdeplot(data.split_frac[data.gender=='W'],
label='women', shade=True)
plt.xlabel('split_frac');
```



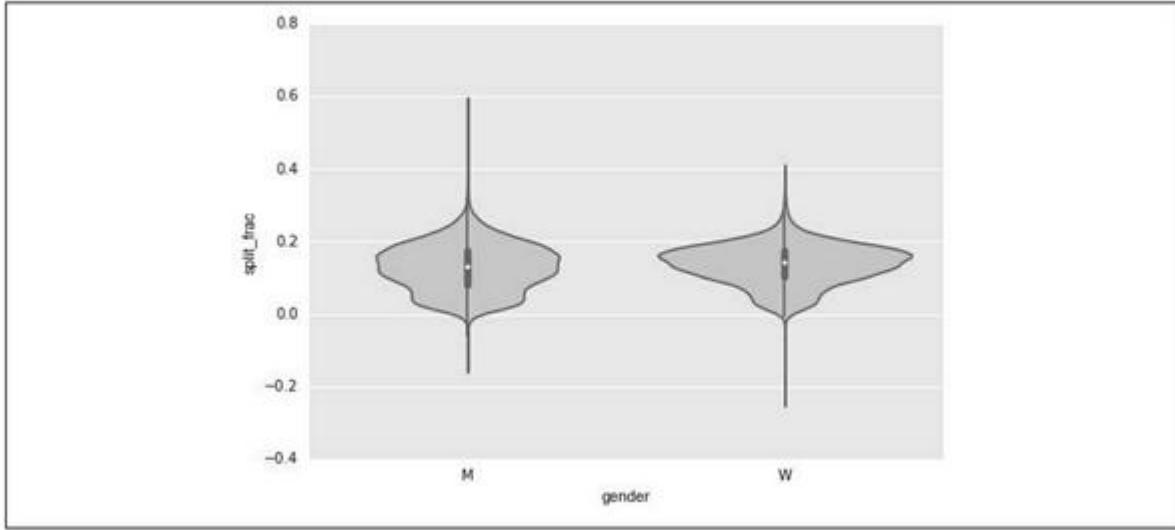
[Figure 4.129](#) : Distribution des valeurs split par genre.

Il y a beaucoup plus d'hommes que de femmes qui équilibrent quasiment leurs deux moitiés de course ! On pourrait penser à une distribution bimodale entre hommes et femmes. Voyons si nous pourrions en apprendre plus en demandant les distributions en fonction de l'âge.

Une bonne façon de comparer les distributions consiste à adopter un tracé en violon ([Figure 4.130](#)) :

In[34]:

```
sns.violinplot("gender", "split_frac",
data=data,
palette=["lightblue",
"lightpink"]);
```



[Figure 4.130](#) : Un tracé en violon montrant la fraction de split par genre.

Créons une nouvelle colonne dans notre tableau pour connaître la décennie de chaque participant ([Figure 4.131](#)) :

In[35]:

```
data['age_dec'] = data.age.map(lambda age: 10 *  
(age // 10))  
data.head()
```

Out[35]:

	age	gender	split	final
	split_sec	final_sec	split_frac	age_dec
0	33	M	0 days 01:05:38	0 days 02:08:51
3938.0	7731.0		-0.018756	30
1	32	M	0 days 01:06:26	0 days 02:09:28
3986.0	7768.0		-0.026262	30
2	31	M	0 days 01:06:49	0 days 02:10:42
4009.0	7842.0		-0.022443	30
3	38	M	0 days 01:06:16	0 days 02:13:45

```

3976.0      8025.0      0.009097      30
4    31      M  0 days 01:06:32  0 days 02:13:59
3992.0      8039.0      0.006842      30

```

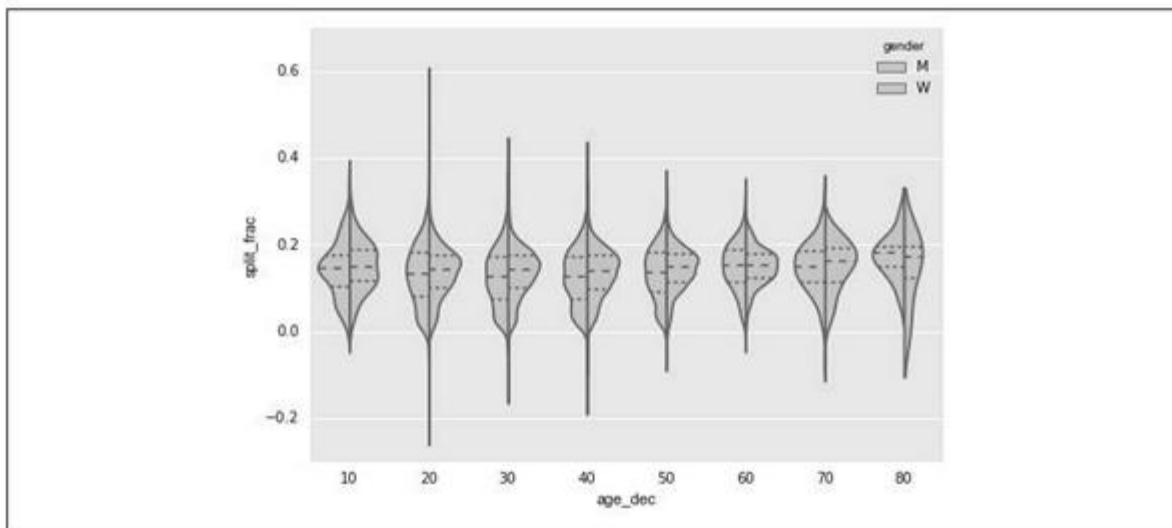
In[36]:

```

men = (data.gender == 'M')
women = (data.gender == 'W')

with sns.axes_style(style=None):
    sns.violinplot(x="age_dec", y="split_frac",
hue="gender", data=data,
                  split=True, inner="quartile",
                  palette=["lightblue",
"lightpink"]);

```



[Figure 4.131](#) : Diagramme montrant la fraction split par genre et par âge.

Nous pouvons voir que les distributions des hommes de leur vingtaine jusqu'à leur cinquantaine montrent plus de valeurs de split basses en comparaison des femmes de

mêmes tranches d'âges (ou de n'importe quel âge d'ailleurs).

On note de façon surprenante que les femmes très âgées semblent battre tout le monde au niveau du temps de split négatif. C'est sans doute lié au faible nombre de concurrentes de cette tranche d'âges :

```
In[37]: (data.age > 65).sum()
```

```
Out[37]: 4
```

Revenons aux hommes et voyons qui sont ceux avec un split négatif. Est-ce que cette caractéristique est corrélée avec un bon temps final ? Nous pouvons facilement en faire un tracé en nous servant de lmplot(), qui va automatiquement adapter une régression linéaire à nos données ([Figure 4.132](#)) :

```
In[38]:
```

```
g = sns.lmplot('final_sec', 'split_frac',
                col='gender',
                data=data, markers=".",
                scatter_kws=dict(color='c'))
g.map(plt.axhline, y=0.1, color="k", ls=":");
```

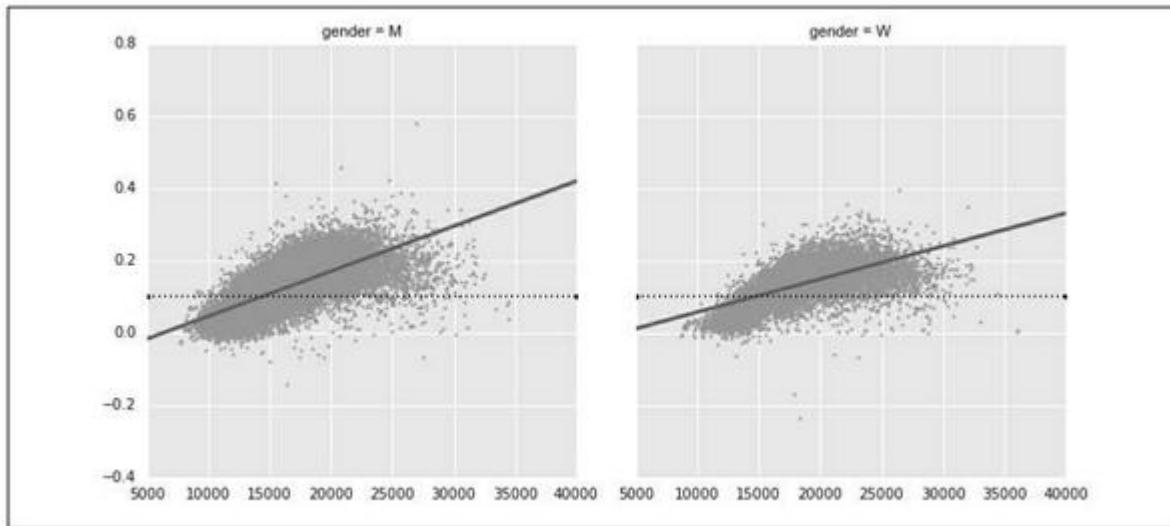


Figure 4.132 : Comparaison entre fraction split et temps final par genre.

Apparemment, ceux qui ont un split rapide sont également ceux qui ont terminé en moins de 15 000 secondes (soit 4 heures environ). Ceux moins rapides sont beaucoup moins nombreux à avoir une seconde moitié de course rapide.

4.17 : Autres ressources

Ressources Matplotlib

Un seul chapitre, même assez complet, ne peut couvrir toutes les possibilités d'une librairie telle que Matplotlib. N'hésitez pas à profiter des fonctions d'aide disponibles dans IPython pour explorer l'interface API de Matplotlib. Voyez également la documentation en ligne (<http://matplotlib.org/>).

Visitez également la galerie (<http://matplotlib.org/gallery.html>) qui montre des centaines de tracés différents, chacun étant associé au code Python qui a permis de le générer. Cela vous permet de découvrir tous les styles de visualisation et les techniques associées.

Pour une description encore plus détaillée de Matplotlib, je vous conseille le livre écrit par un des développeurs principaux de Matplotlib, Ben Root, intitulé *Interactive Applications Using Matplotlib* (<http://bit.ly/2fSqwQ>).

Autres librairies graphiques Python

Matplotlib est la librairie de visualisation principale de Python, mais il existe quelques outils qui sont apparus pour

répondre aux mêmes besoins. J'en cite quelques-uns :

- **Bokeh** (<http://bokeh.pydata.org>). Cette librairie JavaScript avec une interface Python sert à créer des visualisations interactives à partir de jeux de données volumineux et en flux. La partie Python génère une structure de données au format JSON qui est exploitable par le moteur Bokeh JS.
- **Plotly** (<http://plot.ly>). Logiciel open source produit par la société du même nom et similaire à Bokeh. La start-up qui développe ce produit déploie d'intenses efforts de développement ; la librairie reste pourtant d'utilisation gratuite.
- **Vispy** (<http://vispy.org/>). Projet en pleine effervescence qui propose des visualisations dynamiques de jeux de données volumineux. Le logiciel sert à exploiter OpenGL et à tirer profit des coprocesseurs graphiques, ce qui lui permet de réaliser des visualisations aussi importantes que stupéfiantes.
- **Vega** (<https://vega.github.io/>) et **Vega-Lite** (<https://vega.github.io/vega-lite>). Ces produits sont des représentations graphiques déclaratives, et constituent le fruit d'années de recherche dans le langage fondamental de la visualisation de données.

L'implémentation de référence pour le rendu est écrite en JavaScript, mais l'interface API est indépendante du langage. Il y a d'ailleurs une API Python en cours de développement dans le paquetage Altair (<http://altair-viz.github.io/>).

L'univers de la visualisation Python est très dynamique et je suis quasiment certain que les librairies que je viens de citer vont être rejoints par d'autres. Restez à l'affût !

CHAPITRE 5

Apprentissage machine

C'est souvent à l'apprentissage machine que pense le grand public lorsqu'il entend parler de sciences des données ou *datalogie*. L'apprentissage machine est ce domaine à la confluence entre algorithmique et statistique qui regroupe plus d'une dizaine d'approches pour l'exploration des données et la recherche de tendances significatives ; il s'agit plus de traitements réels que de théorie.

Le terme « apprentissage machine » est souvent présenté comme une pilule magique : il vous suffirait d'appliquer l'apprentissage machine à vos données et tous vos problèmes seraient résolus ! Vous vous doutez que la réalité est rarement aussi simple. Les différentes méthodes proposées sont très puissantes, mais elles ne seront efficaces que si vous connaissez les points forts et les points faibles de chacune d'elles, après avoir assimilé les concepts fondamentaux tels que les biais et les erreurs systématiques, la variance, les sur- et sous-ajustements, et bien d'autres aspects.

Pour présenter en pratique l'apprentissage machine, nous allons dans ce chapitre nous appuyer principalement sur le paquetage pour Python nommé Scikit-Learn (<http://scikit-learn.org>). Nous ne prétendons pas décrire tout le domaine de l'apprentissage machine qui est un sujet bien trop vaste et suppose de plonger bien plus avant dans les détails techniques. Il ne s'agit pas non plus d'un manuel de référence du paquetage Scikit-Learn. Pour en savoir plus à son sujet, vous puiserez dans les suggestions données en fin de chapitre. Voici les objectifs que nous nous assignons :

- une introduction au vocabulaire et aux concepts fondamentaux de l'apprentissage machine ;
- une présentation de l'interface fonctionnelle API de Scikit-Learn avec des exemples ;
- une présentation plus détaillée d'une dizaine d'approches d'apprentissage machine, afin d'aider à comprendre leur fonctionnement et les domaines et moments adéquats à chacune des techniques.

L'essentiel du contenu du chapitre est le fruit de mes tutoriels et ateliers réalisés en différentes occasions lors des conférences PyCon, SciPy et PyData. Si les pages qui suivent ont su éviter certaines ambiguïtés, je le dois aux nombreux participants aux ateliers et collègues formateurs qui m'ont fourni leur avis au cours de ces années.

Si vous avez besoin d'une description plus technique de n'importe lequel des sujets abordés, vous exploitez la liste fournie en fin de chapitre (N.d.T. : mais peu de titres sont disponibles en français).

5.1 : L'apprentissage machine, c'est quoi ?

Avant de découvrir en détail les différentes techniques, voyons ce qu'est l'apprentissage machine et ce qu'il n'est pas. Souvent, cette activité est considérée comme étant un sous-domaine de l'intelligence artificielle, mais je pense que cela peut induire en erreur. Il est évident que l'apprentissage machine est apparu suite à des recherches dans ce contexte, mais lorsqu'il s'agit d'appliquer ces techniques à la datalogie, il est plus fructueux de considérer que l'apprentissage machine est un moyen permettant de construire des modèles à partir des données.

Il s'agit en effet de construire des modèles mathématiques qui permettent de mieux comprendre les données. La notion d'apprentissage découle de la possibilité de fournir à ces modèles des paramètres de travail ajustables pour observer les données. En cherchant à optimiser ces paramètres, on ajuste le programme aux données. Une fois qu'un modèle a été ajusté le mieux possible à des données existantes, il devient possible de l'appliquer à de nouvelles données. Je laisse les lecteurs décider si cet apprentissage basé sur des modèles très mathématiques peut se comparer à l'apprentissage tel que le pratique le cerveau humain.

Pour pouvoir utiliser ce genre d'outils de façon efficace, il est indispensable de connaître le contexte du problème posé. Nous allons donc commencer par découvrir les grandes catégories d'approches disponibles.

Catégories d'apprentissage machine

Il existe deux catégories principales d'apprentissage machine : l'apprentissage supervisé et l'apprentissage non supervisé.

L'apprentissage supervisé revient à modéliser les relations qui existent entre les caractéristiques des données et les labels ou légendes associés aux données. Une fois le modèle déterminé, il peut servir à associer de nouveaux labels à des données actuellement inconnues. Cette catégorie se répartit en tâches de *classification* et tâches de *régression*. Dans la classification, les labels sont des catégories distinctes les unes des autres alors que dans la régression, ce sont des quantités continues, souvent numériques. Nous verrons des exemples des deux types d'apprentissage supervisé dans la suite.

L'apprentissage non supervisé consiste à modéliser les caractéristiques d'un jeu de données sans recourir à aucun label. On parle souvent d'action pour faire parler les données

à propos d'elles-mêmes. Cette catégorie regroupe les tâches de regroupement ou partitionnement (*clustering*) et celles de réduction dimensionnelle. Les algorithmes de regroupement cherchent à identifier des groupes de données alors que ceux de réduction dimensionnelle ont pour but de créer des synthèses en réduisant le nombre de caractéristiques. Nous verrons des exemples des deux types d'apprentissage non supervisé dans la suite.

Il existe également une catégorie d'apprentissage semi-supervisé, qui est à cheval entre les deux précédentes. Les techniques correspondantes sont en général appliquées lorsque l'on dispose de labels incomplets.

Exemples qualitatifs d'apprentissage machine

Voyons d'abord quelques exemples des résultats que l'on peut obtenir avec l'apprentissage machine. Ces premiers exemples proposent une vue qualitative et non quantitative. Nous verrons dans les sections suivantes des exemples concrets. Leur code source Python est disponible dans l'archive du livre à télécharger depuis le site de l'éditeur (voir l'introduction).

Classification : prédiction de labels discontinus

Commençons par une tâche de classification. À partir d'un ensemble de points labellisés, nous allons essayer d'associer des labels à des points qui n'en ont pas encore.

Nous travaillons sur le jeu de données visualisé dans la [Figure 5.1](#).

Nos données sont à deux dimensions : pour chaque point, nous disposons de deux caractéristiques qui sont représentées par les coordonnées (x , y) dans le plan. Pour chaque point, nous possédons également un *label de classe* parmi deux qui correspond aux couleurs des points. À partir de ces caractéristiques et de ces labels, nous allons chercher à créer un modèle qui permettra de décider pour d'autres points encore inconnus s'ils doivent être classés plutôt du côté « bleu » ou plutôt du côté « rouge ».

Plusieurs modèles sont utilisables pour cette classification, mais nous allons nous limiter à un modèle très simple. Nous supposons que les deux groupes peuvent être séparés en traçant une ligne droite dans le plan de sorte que les points d'un côté soient tous dans le même groupe. Notre modèle va incarner cette ligne droite de séparation grâce aux paramètres qui vont correspondre à des valeurs numériques permettant de choisir la position et l'orientation de la droite.

Les valeurs les plus appropriées pour les *paramètres* vont être obtenues en analysant les données, ce qui correspond à la phase d'apprentissage. L'opération consiste à entraîner le modèle.

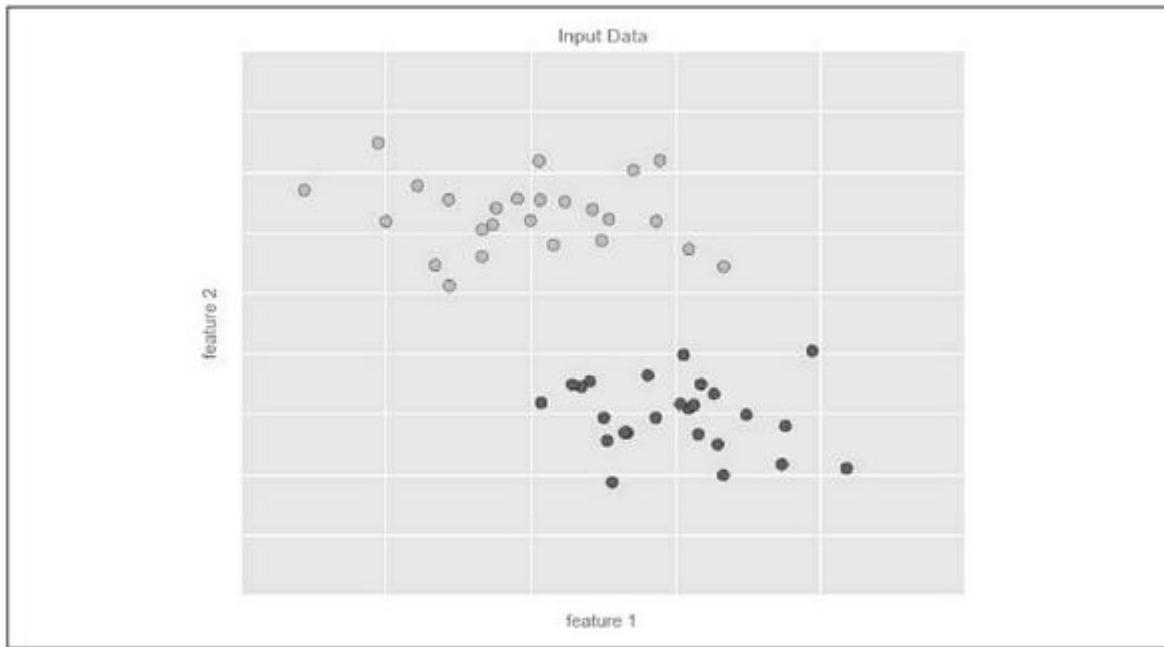


Figure 5.1 : Un jeu de données simple pour une classification.

La [Figure 5.2](#) montre le modèle une fois entraîné pour les mêmes données.

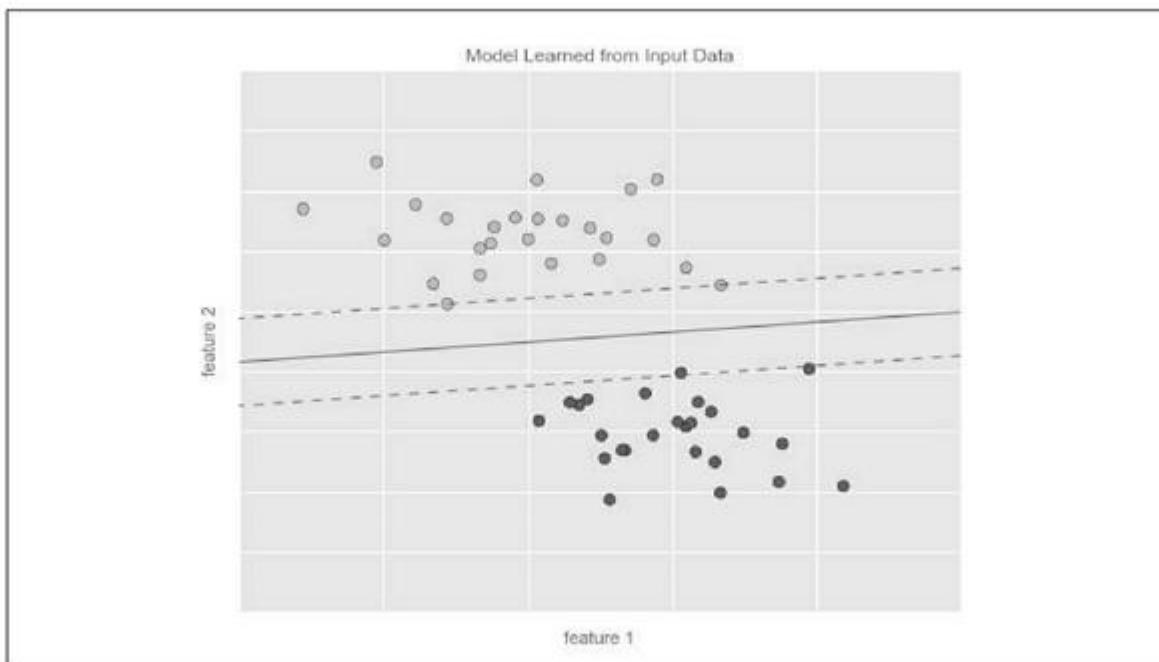


Figure 5.2 : Un modèle de classification simple.

Une fois le modèle entraîné, nous allons pouvoir l'appliquer à des données nouvelles sans labels. Nous lui proposons un nouveau jeu de données, nous faisons mettre en place la séparation par le modèle puis nous affectons des labels aux nouveaux points selon leur position. Cette étape correspond à la prédiction. Voyez la [Figure 5.3](#).

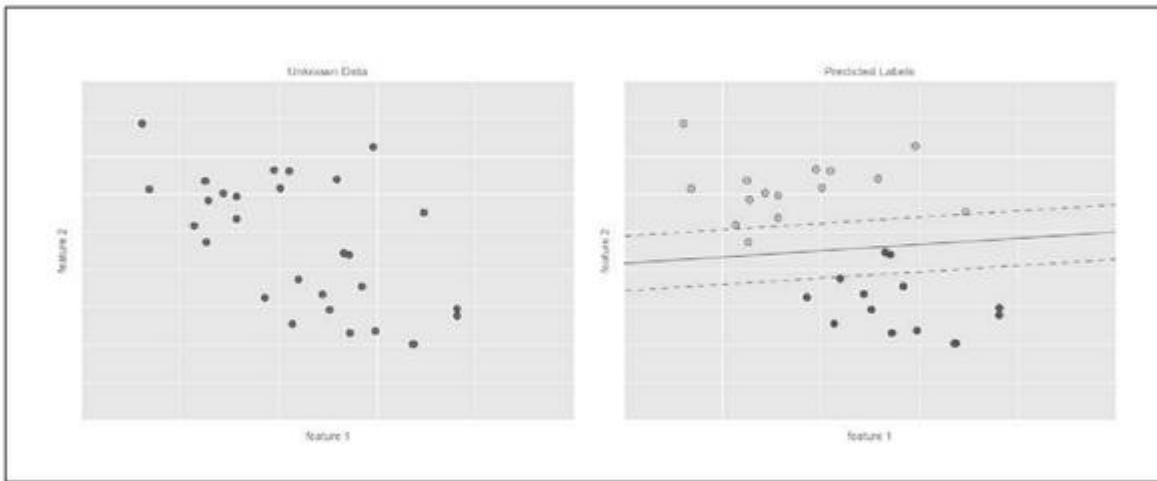


Figure 5.3 : Application d'un modèle de classification à de nouvelles données.

Nous venons de voir ce qu'est une tâche de classification en apprentissage machine. Le terme « classification » rappelle que les données concernées sont dotées de labels de classes discontinues. Dans notre exemple, la séparation aurait pu très facilement se faire visuellement. L'avantage de l'apprentissage machine est que ce même modèle peut être appliqué à un volume de données bien supérieur avec un plus grand nombre de dimensions.

La détection automatique des indésirables ou spams dans votre courriel est une tâche fort similaire. Dans ce cas, nous utiliserions les caractéristiques et labels suivants :

- *caractéristique 1, caractéristique 2, etc.* : nombre lissé de mots ou de phrases particuliers tels que « Viagra », « Prince nigérien », « maladie », etc.
- *label* : « indésirables » ou « désirables ».

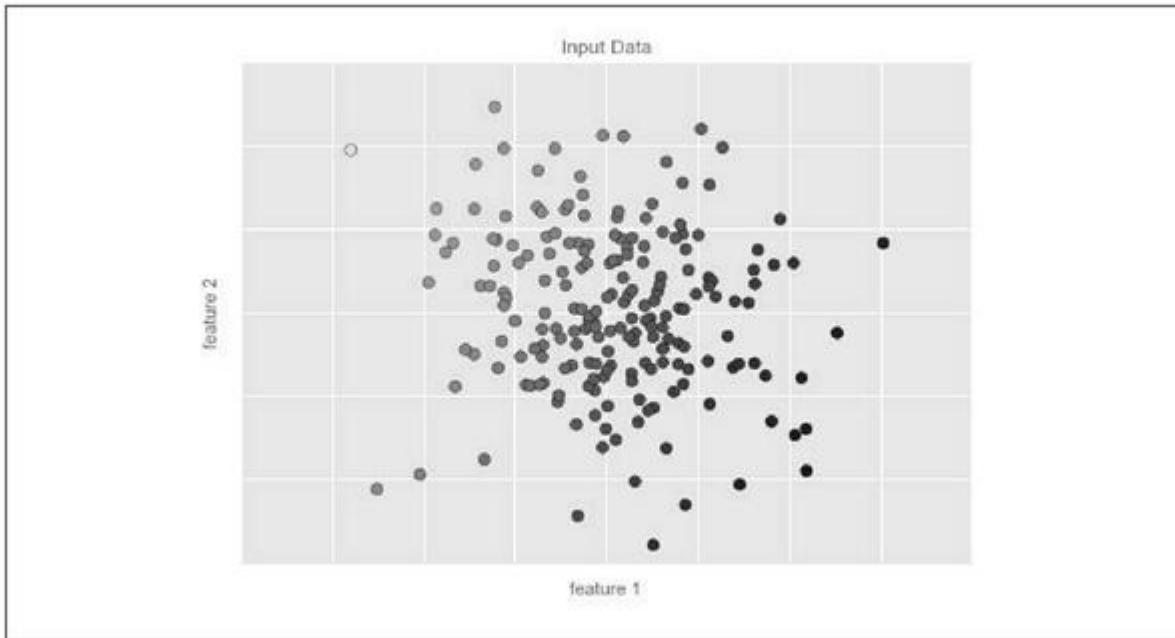
Pour créer le jeu d'entraînement, nous pouvons inspecter individuellement un échantillon de courriels pour trouver les labels puis utiliser le modèle ainsi préparé pour traiter le lot complet de courriels et chercher les labels. Cette approche peut se montrer très efficace si l'algorithme de classification est correctement entraîné et si vous disposez de caractéristiques correctement définies (un socle de milliers ou de millions de mots ou de phrases). Nous verrons un exemple de classification basée sur le texte dans la section du chapitre qui entre en détail dans la classification bayésienne naïve.

Nous verrons trois algorithmes de classification importants : bayésienne naïve gaussienne, machine à vecteurs de support et classification par forêt aléatoire, qui correspondent à trois sections de ce chapitre.

Régression : prédiction de labels continus

En opposition à l'algorithme de classification qui travaille sur des labels distincts, nous allons découvrir une tâche de régression qui utilise des labels sous forme de quantités continues.

Nous partons des données montrées dans la [Figure 5.4](#). Il s'agit d'un jeu de points, chacun doté d'un label continu.

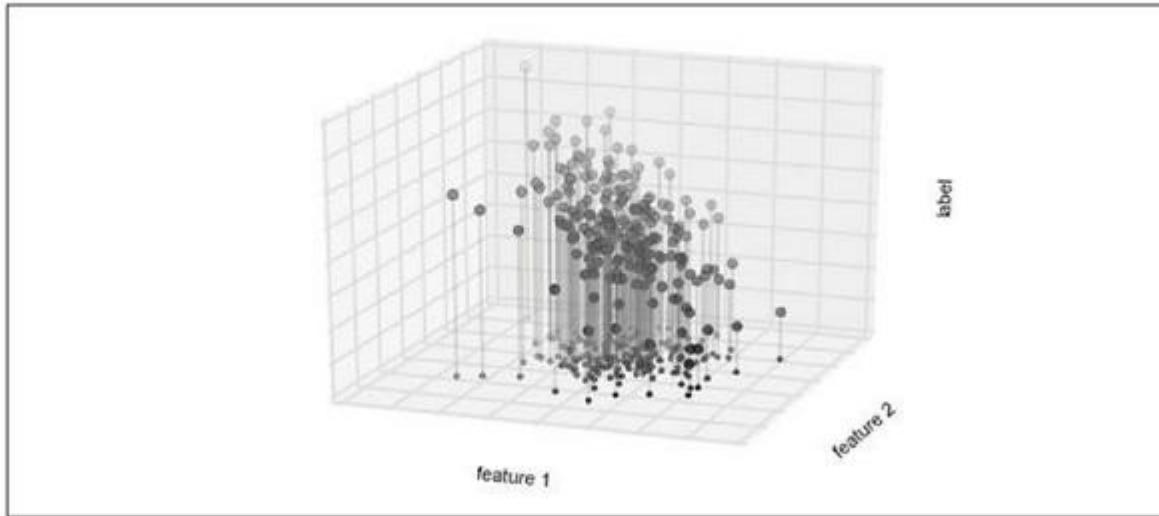


[Figure 5.4](#) : Jeu de données simple pour une régression.

Ici aussi, nous partons de données à deux dimensions, c'est-à-dire qu'il y a deux caractéristiques associées à chaque point. Le label continu est la couleur du point.

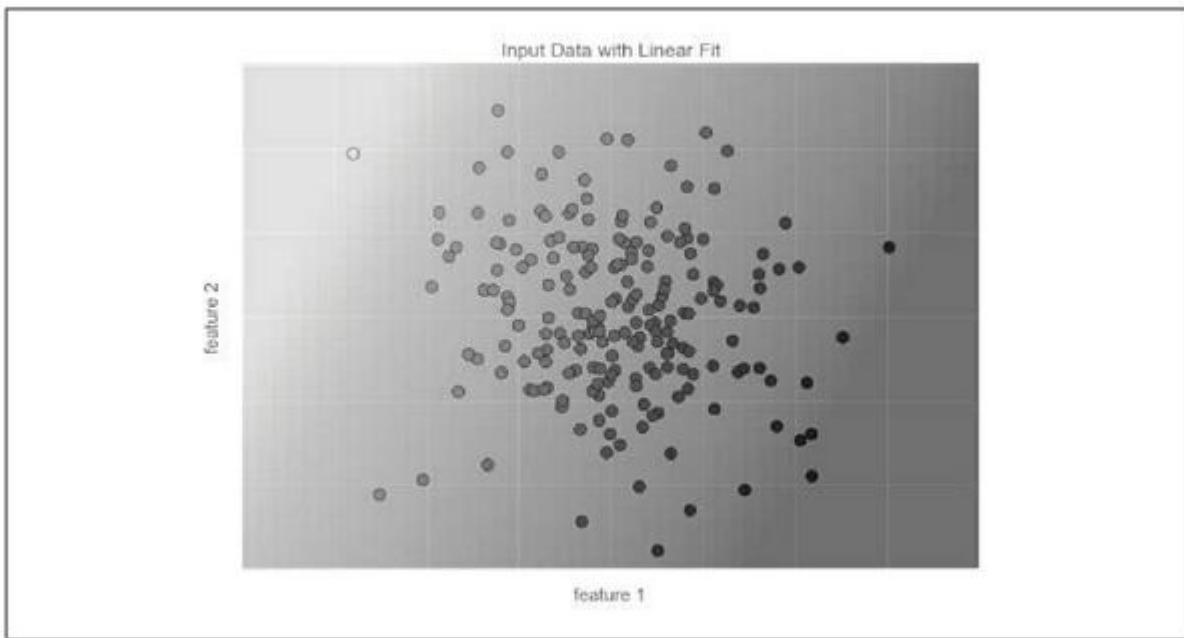
Plusieurs modèles de régression sont disponibles, mais nous allons nous contenter d'un modèle linéaire simple. Dans ce modèle, nous supposons que s'il est possible de traiter les labels comme une troisième dimension, nous pouvons les associer sous forme d'un plan aux données. C'est une génération à plus haut niveau du problème bien connu consistant à ajuster une ligne à des données sous forme de deux coordonnées.

Cette mise en place correspond à ce que montre la [Figure 5.5](#).



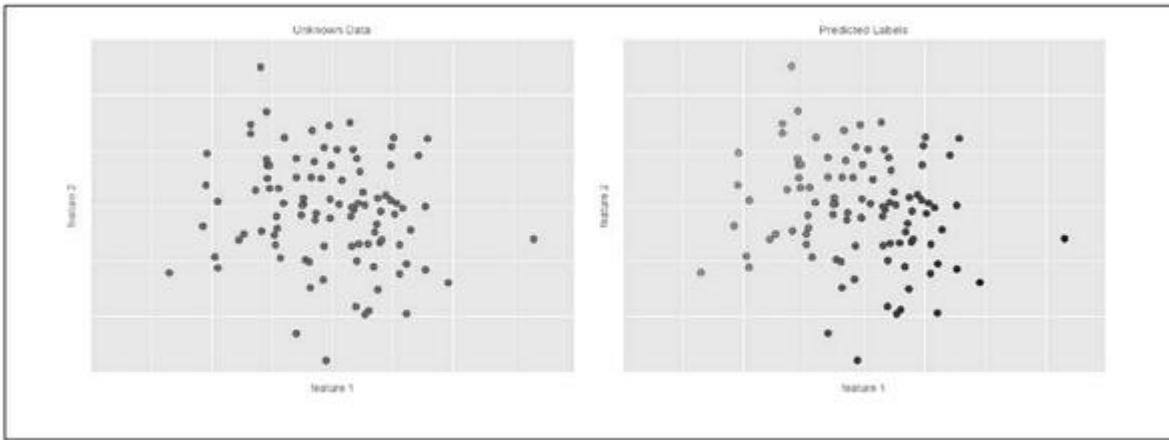
[Figure 5.5](#) : Vue en trois dimensions des données de régression.

Le plan horizontal qui réunit *caractéristique 1* et *caractéristique 2* est le même que celui du plan de la figure précédente. En revanche, nous avons visualisé les labels par une couleur et par une position selon un troisième axe. Cette vue laisse deviner qu'il doit être possible d'ajuster un plan dans cette représentation en 3D pour prédire les labels en fonction de n'importe quel jeu de paramètres d'entrée. Si nous revenons à une projection en 2D, nous obtenons ce que montre la [Figure 5.6](#) lorsque nous procédons à l'ajustement du plan.



[Figure 5.6](#) : Représentation du modèle de régression.

Ce plan d'ajustement constitue ce qui nous permet de prédire les labels pour de nouveaux points de données. Nous obtenons le résultat montré en [Figure 5.7](#).



[Figure 5.7](#) : Application du modèle de régression à de nouvelles données.

Comme pour l'exemple de classification, lorsque les données et les dimensions sont peu nombreuses, l'opération paraît très simple. Ce qui est important, c'est que ce modèle peut être appliqué à des données beaucoup plus complexes.

Cette technique permet par exemple de calculer les distances des galaxies vues dans un télescope. Les caractéristiques et les labels correspondants pourraient se présenter ainsi :

- *caractéristique 1, caractéristique 2, etc.* : luminosité de chaque galaxie à une longueur d'onde ou couleur.
- *label* : distance ou décalage dans le rouge de la galaxie.

Pour l'entraînement, il s'agirait de faire des observations normalement plus longues et coûteuses pour un échantillon des galaxies. Il suffit ensuite d'utiliser le bon modèle de régression pour estimer les distances des autres galaxies sans avoir à les observer une à une. Dans le monde de

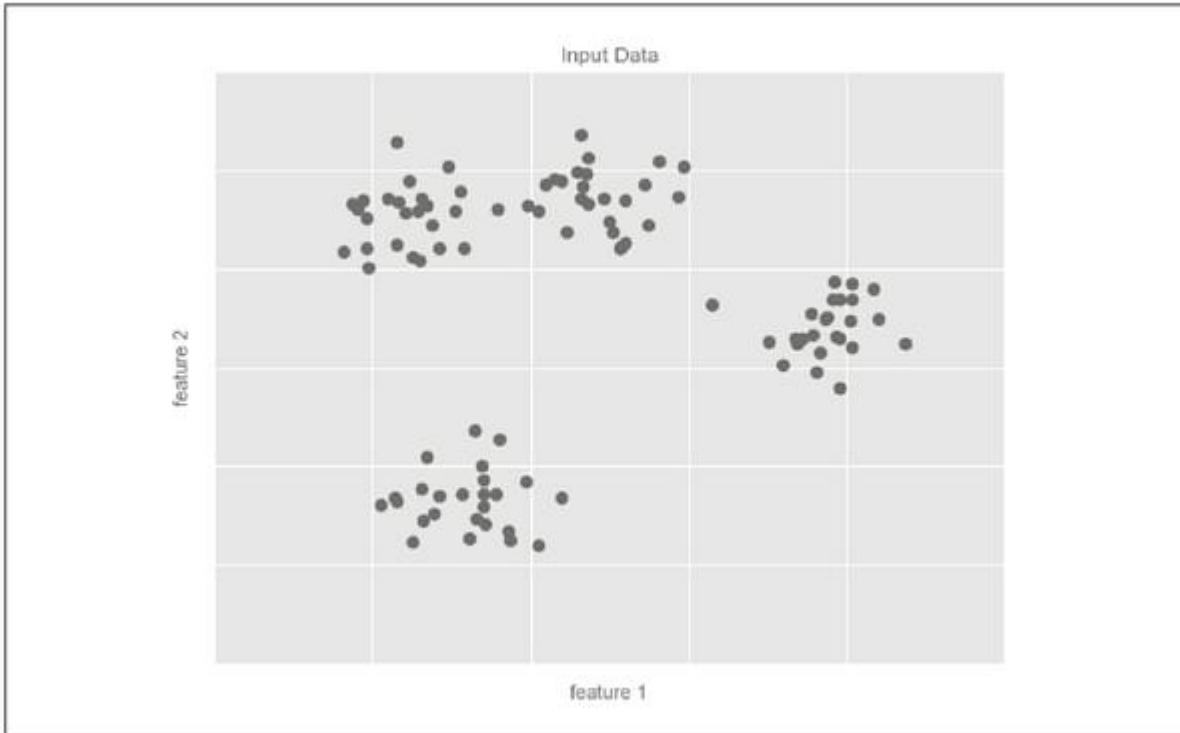
l'astronomie, cela correspond au problème du décalage dans le rouge photométrique (*redshift*).

Nous verrons en détail trois modèles de régression : la régression linéaire, les machines à vecteurs de support et la régression par forêt aléatoire.

Regroupement ou clustering : répartition de labels

Les deux algorithmes de classification et de régression que nous venons de voir correspondent à de l'apprentissage supervisé. La première étape consiste à construire le modèle qui va servir à prédire de nouveaux labels. Dans l'apprentissage non supervisé, le modèle ne dispose d'aucun label connu au départ.

Une technique d'apprentissage non supervisé très répandue correspond au regroupement ou « clustering ». Il s'agit d'associer automatiquement les données à différents groupes distincts. Partons par exemple des données en 2D suivantes ([Figure 5.8](#)).



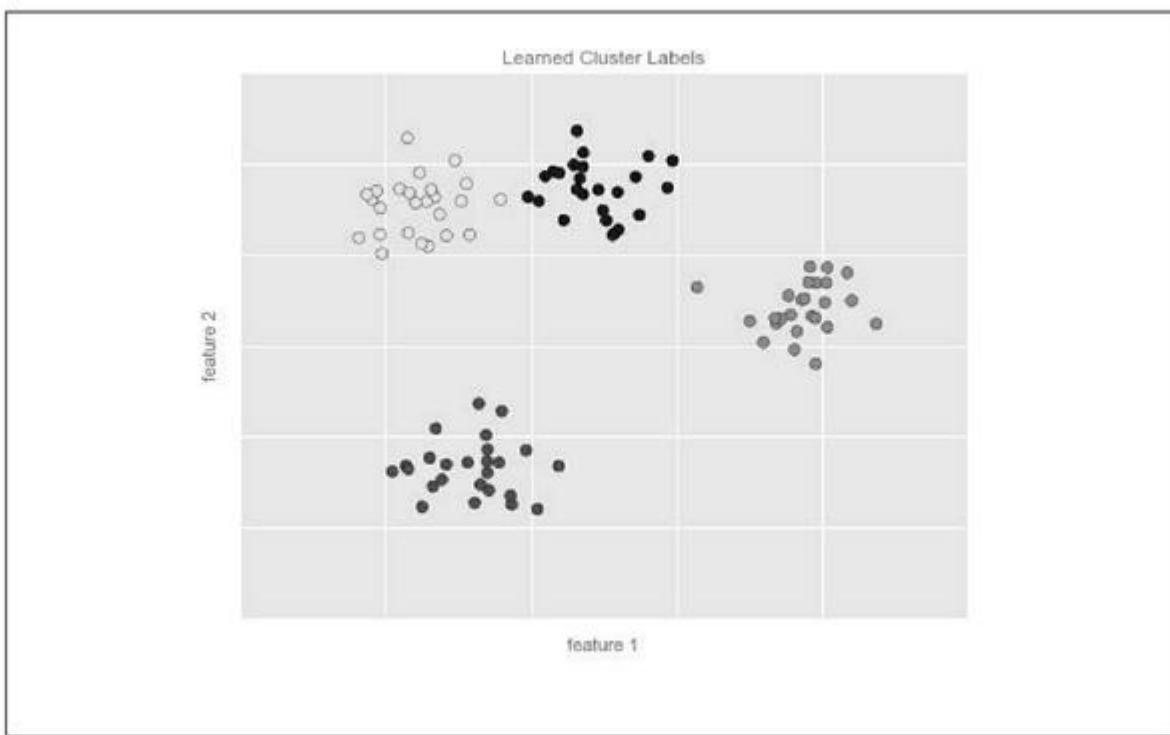
[Figure 5.8](#) : Données d'exemple pour un regroupement.

En observant l'image, on devine que les points font partie de plusieurs groupes. Le modèle par regroupement utilise la structure native des données pour décider à quel groupe appartient chaque point. Nous trouvons les clusters montrés dans la [Figure 5.9](#) en utilisant l'algorithme de k-moyennes, très rapide et très facile d'emploi.

L'approche par k-moyennes cherche à ajuster un modèle qui est constitué de k centres de groupes. Chaque centre optimal est le point pour lequel les distances sont les plus petites depuis chaque point du groupe et le centre. En deux dimensions, l'opération semble facile, mais il en va

autrement lorsque les données deviennent volumineuses et complexes.

Nous verrons plusieurs algorithmes de regroupement dans ce chapitre : l'algorithme par k-moyennes, le modèle mixte gaussien ; le partitionnement spectral est décrit dans la documentation de Scikit-Learn.



[Figure 5.9](#) : Données avec *labels* par application du modèle de regroupement par k-moyennes.

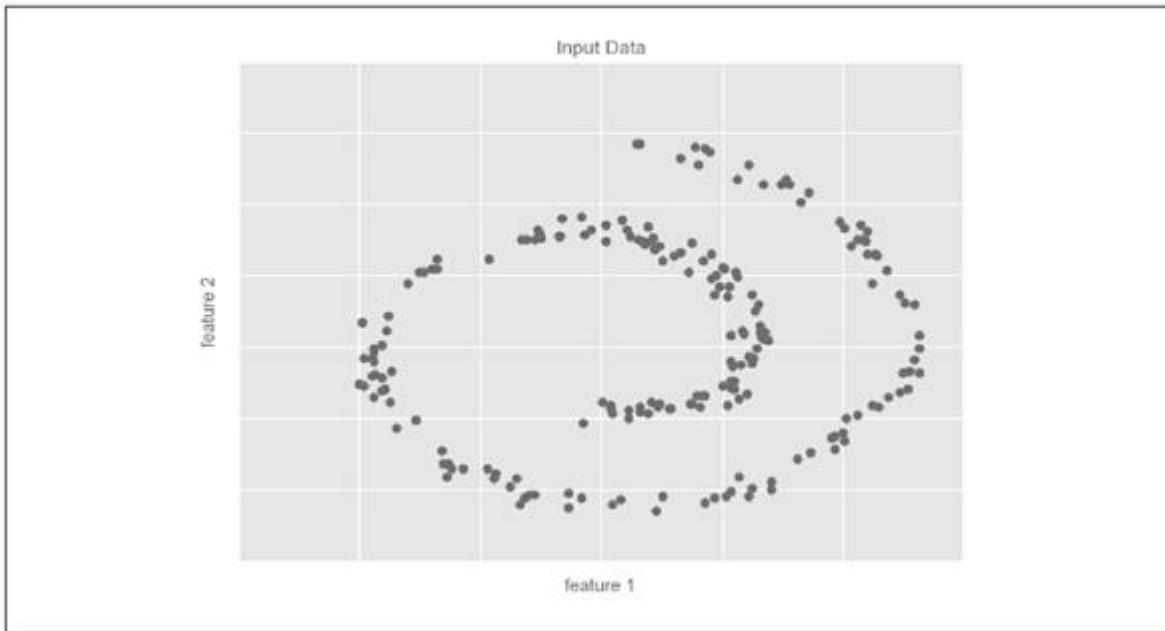
Réduction dimensionnelle : découverte de la structure de données sans labels

Une autre approche d'apprentissage non supervisé correspond à la *réduction dimensionnelle*. Dans ce cas, il s'agit

de découvrir les labels ou une autre information par analyse de la structure du jeu de données. Cette réduction dimensionnelle est un peu plus complexe et abstraite que les approches précédentes, car elle cherche à obtenir une représentation des données avec moins de dimensions tout en préservant les qualités du jeu de données complet. Selon la routine de réduction dimensionnelle, les qualités sont mesurées de façon différente, comme nous le verrons dans la section sur l'apprentissage par variété (*manifold*) de ce chapitre.

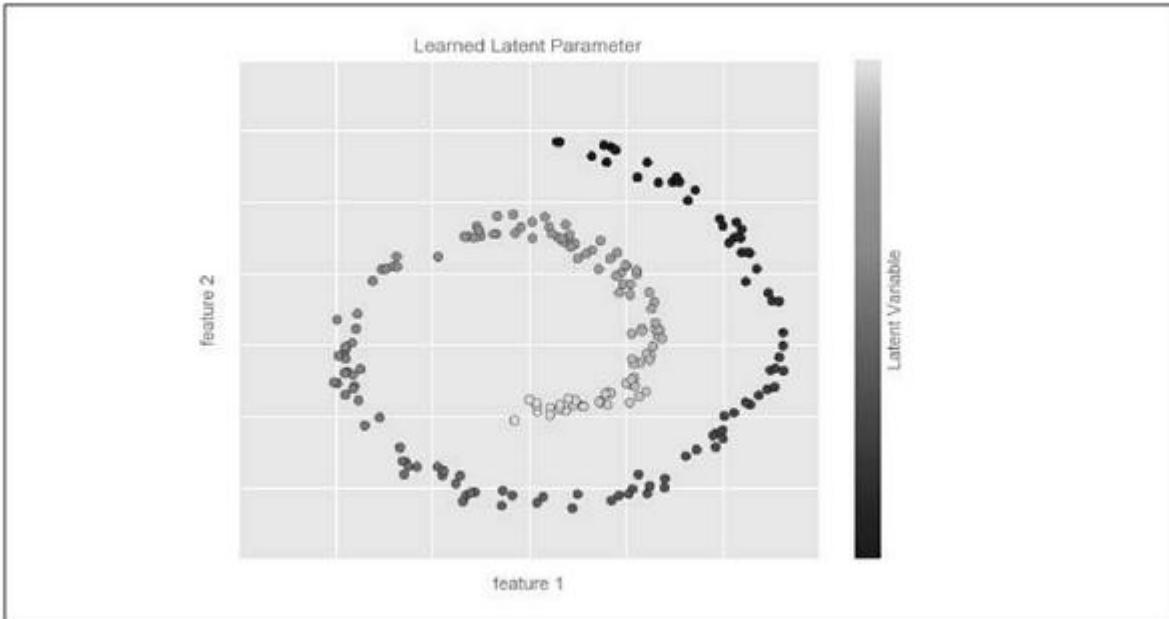
Partons des données montrées dans la [Figure 5.10](#).

Au premier regard, on comprend qu'il y a une certaine structure dans ces données. Elles sont réparties le long d'une ligne enroulée en spirale dans un espace en 2D. D'une certaine manière, on pourrait considérer que les données n'ont qu'une dimension, et que cette dimension unique est intégrée dans un espace offrant plus de dimensions. Un modèle de réduction dimensionnelle qui conviendrait dans ce cas devrait être capable de détecter cette structure interne non linéaire pour en extraire une représentation.



[Figure 5.10](#) : Données d'exemple pour une réduction dimensionnelle.

La [Figure 5.11](#) montre les résultats de l'algorithme nommé Isomap qui est un algorithme d'apprentissage par variété convenant parfaitement à cet exemple.



[Figure 5.11](#) : Différenciation des données par un label de couleur dans une réduction dimensionnelle.

Les couleurs qui incarnent la variable latente à une dimension qui a été extraite varient de façon uniforme tout au long de la spirale, ce qui confirme que l'algorithme a bien découvert la structure que nous pouvons voir. Comme dans les exemples précédents, l'intérêt de ce genre d'algorithme devient évident lorsque le nombre de dimensions augmente. Supposez que vous ayez besoin de visualiser quelques relations dans un jeu de données qui en possède 100 ou 1 000. Pour ne pas devoir visualiser mille dimensions, nous pouvons recourir à une réduction dimensionnelle pour aboutir à deux ou trois dimensions intéressantes.

Deux sections du chapitre se concentrent sur les algorithmes de réduction dimensionnelle : l'analyse par composantes principales PCA et différents algorithmes par variété, en particulier Isomap et l'intégration localement linéaire.

Synthèse

Nous venons de passer en revue des exemples extrêmement simples montrant les différentes approches d'apprentissage machine. Ce serait faire affront aux lecteurs que de préciser que nous avons volontairement passé sous silence d'importants détails pratiques. J'espère néanmoins que cette mise en appétit vous donne une idée générale des genres de problèmes que l'on peut résoudre grâce à la technique de l'apprentissage machine.

Revoyons les grandes lignes de ce qui a été découvert :

Apprentissage supervisé

Ce sont des modèles qui permettent de trouver des labels ou des noms à partir de données d'entraînement qui en possèdent déjà.

- *Classification* Modèle trouvant des labels en tant que catégories discontinues (discrètes).
- *Régression* Modèle qui trouve des labels en tant que valeurs continues (numériques).

Apprentissage non supervisé

Modèles qui détectent une structure dans des données sans labels.

- *Regroupement ou Clustering* Modèle détectant puis identifiant des groupes de données distincts parmi les données.
- *Réduction dimensionnelle* Modèle détectant et identifiant une structure avec un nombre de dimensions réduit.

Tous les résultats de cette visite guidée peuvent être générés chez vous en ayant recours aux calepins de code fournis avec le fichier des archives du livre ainsi qu'en version anglaise à l'adresse suivante :

<http://github.com/jakevdp/PythonDataScienceHandbook>.

5.2 : Présentation de Scikit-Learn

Plusieurs librairies de fonctions Python peuvent répondre aux besoins d'algorithmes d'apprentissage machine. L'une des plus utilisées, Scikit-Learn (<http://scikit-learn.org>), offre une interface fonctionnelle API claire, uniforme et facile d'emploi, et s'appuie sur une documentation en ligne très complète (même si en anglais seulement). La grande homogénéité de la librairie vous permet d'apprendre les bases et la syntaxe de Scikit-Learn pour un premier genre de modèle, puis d'utiliser très facilement les autres modèles et algorithmes.

Voyons les grandes lignes de l'API de Scikit-Learn. L'acquisition du contenu de cette section vous apportera une base solide pour contrôler les détails pratiques très techniques des différents algorithmes d'apprentissage qui font l'objet des sections suivantes de ce chapitre.

Nous allons d'abord voir comment les données sont représentées pour Scikit-Learn puis découvrirons l'interface API d'estimateur. Nous verrons ensuite un exemple concret consistant à reconnaître des chiffres manuscrits.

Représentation des données dans Scikit-Learn

Le but de l'apprentissage machine est de construire un modèle à partir de données. Nous allons donc d'abord voir comment les données d'entrée doivent être organisées pour qu'elles puissent être exploitées au mieux par l'algorithme. Le format le plus approprié est celui d'un tableau de données.

Des données en tableaux

Ce que l'on appelle tableau en informatique est une matrice dans laquelle les lignes correspondent à différents éléments d'un jeu de données, et les colonnes à des quantités ou des valeurs caractérisant chaque élément. Partons du jeu de données d'exemple floral et très connu des iris de Fisher (https://fr.wikipedia.org/wiki/Iris_de_Fisher) qui avait été étudié au départ par Ronald Fisher dès 1936. Nous pouvons préparer ce jeu de données au format d'une structure DataFrame Pandas en nous servant de la librairie Seaborn (<https://stanford.edu/~mwaskom/software/seaborn/>) :

```
In[1]:  
import seaborn as sns  
iris = sns.load_dataset('iris')  
iris.head()
```

Out[1]:

	sepal_length	sepal_width	petal_length	
	petal_width	species		
0	5.1	3.5	1.4	
0.2	setosa			
1	4.9	3.0	1.4	
0.2	setosa			
2	4.7	3.2	1.3	
0.2	setosa			
3	4.6	3.1	1.5	
0.2	setosa			
4	5.0	3.6	1.4	
0.2	setosa			

Chaque ligne de l'extrait affiché correspond à une fleur telle qu'elle a été observée. Le nombre total de lignes correspond au nombre d'observations. En général, les lignes d'une matrice sont des *échantillons* et la variable `n_samples` contient le nombre d'échantillons.

Les colonnes du tableau contiennent des quantités relatives à chaque échantillon. En général, les colonnes sont des *caractéristiques* et le nombre de colonnes est stocké dans une variable nommée `n_features`.

Matrice de caractéristiques

Ce tableau à deux dimensions porte également le nom de *matrice de caractéristiques*. Par convention, on utilise la

variable nommée X (majuscule) pour stocker cette matrice de données d'entrée. Le format de la matrice peut s'exprimer sous la forme [n_samples, n_features]. En général, elle sera stockée dans un tableau NumPy ou un cadre DataFrame de Pandas, même si certains modèles Scikit-Learn acceptent également de travailler sur des matrices éparses (creuses) de SciPy.

Les lignes (les échantillons) correspondent toujours aux objets individuels décrits par le jeu de données. Un échantillon peut être une fleur, une personne, une image, un document, un son, une vidéo, un objet cosmique ou toute chose (non unique) que vous pouvez décrire en lui associant une série de valeurs quantitatives.

Les colonnes ou caractéristiques correspondent aux observations qui fournissent des informations concernant chaque échantillon. En général, ces caractéristiques sont des valeurs réelles numériques, mais il peut également s'agir de valeurs booléennes (oui ou non) ou de valeurs discontinues, donc qualitatives.

Tableau cible

En plus de la matrice de caractéristiques X (nous abrégerons parfois en *matcar*), il faut souvent également disposer d'un tableau cible ou tableau de labels que nous appelons par convention y . En général, le tableau ne possède qu'une

dimension, avec une longueur égale à `n_samples`. Il est normalement stocké dans un tableau NumPy ou un objet Series de Pandas. Il peut contenir des valeurs numériques continues ou des noms de classes ou de labels discontinus (discrets). Nous allons principalement travailler avec des tableaux cibles à une dimension, tout en sachant que certaines fonctions d'estimateurs de Scikit-Learn savent traiter des valeurs cibles sous forme de tableaux à deux dimensions (`[n_samples, n_targets]`).

Il arrive trop souvent que l'on se méprenne sur la différence de nature entre le tableau cible et les autres colonnes de caractéristiques, celles d'entrée. Le tableau cible est normalement la donnée quantitative que l'on cherche à prédire à partir des données disponibles ; dans le langage des statistiques, il s'agit de la *variable dépendante*. Dans l'exemple précédent, nous pourrions chercher à construire un modèle capable de trouver par prédition l'espèce de fleur à partir des caractéristiques (mesures). Dans ce cas, la colonne `species` serait la *caractéristique cible*.

Nous pouvons nous servir de Seaborn pour obtenir une visualisation de ce tableau cible pour les différentes espèces d'iris ([Figure 5.12](#)) :

```
In[2]:  
%matplotlib inline
```

```
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```

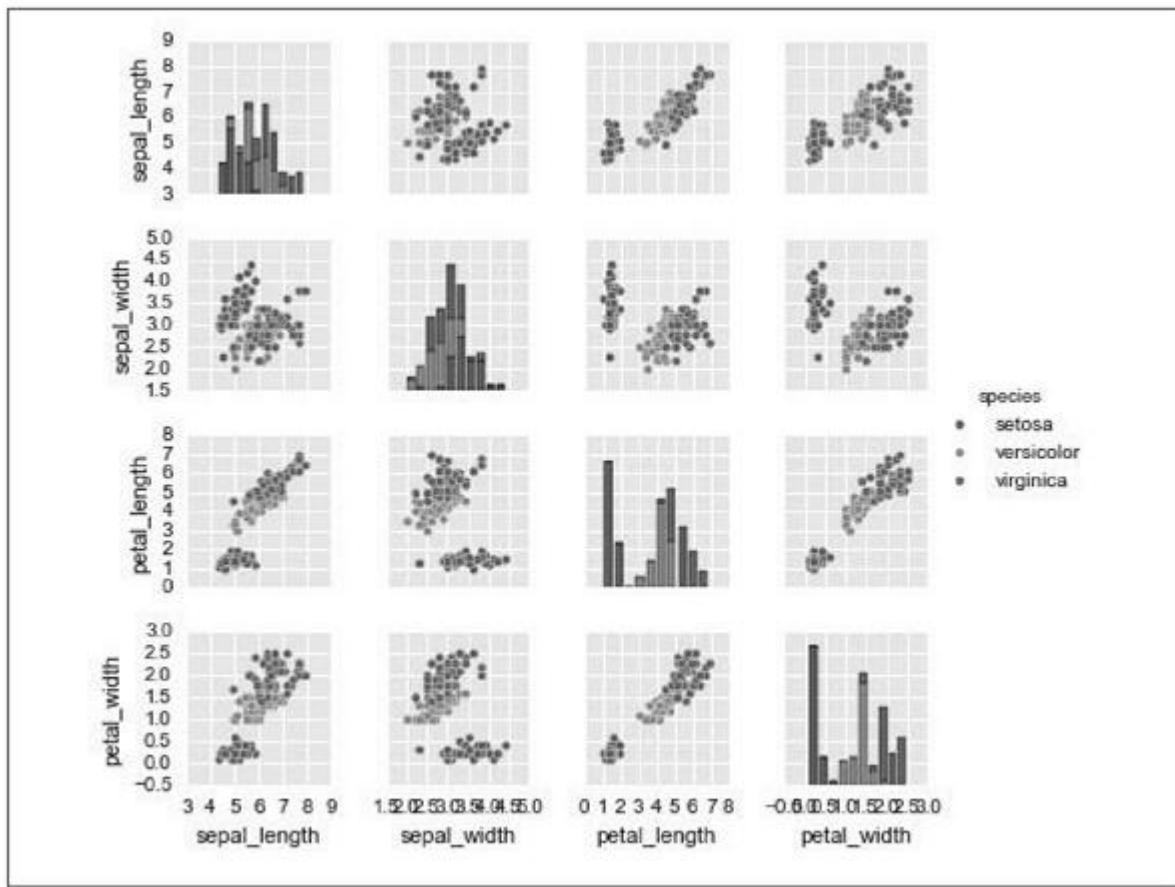


Figure 5.12 : Visualisation du jeu de données Iris.

Pour une exploitation avec Scikit-Learn, nous décidons d'extraire la matrice de caractéristiques et le tableau cible d'une structure DataFrame. Nous nous servons à cet effet des fonctions disponibles avec DataFrame, présentées dans le [Chapitre 3](#) :

In[3] :

```
X_iris = iris.drop('species', axis=1)
```

```
X_iris.shape
```

Out[3]:

```
(150, 4)
```

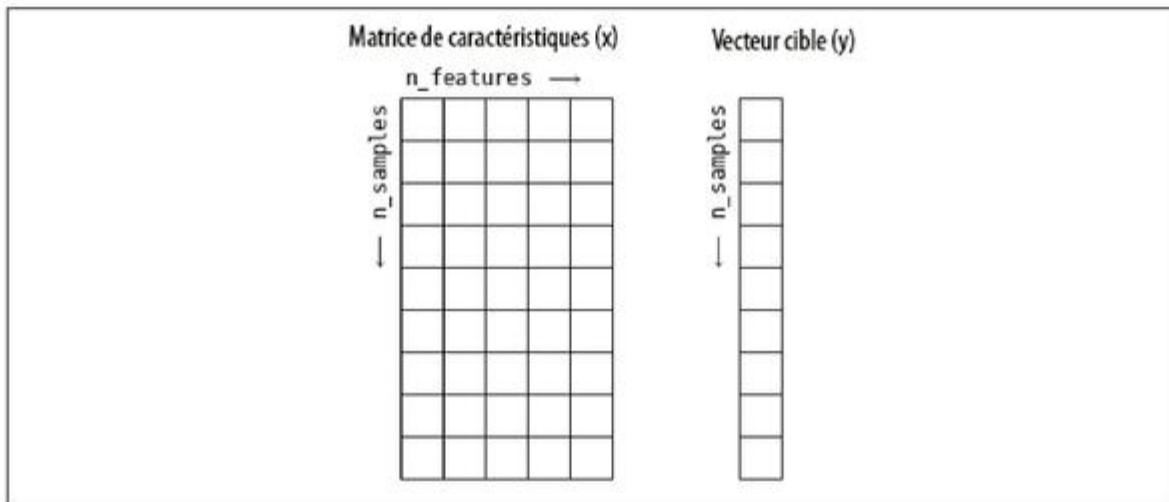
In[4]:

```
y_iris = iris['species']  
y_iris.shape
```

Out[4]:

```
(150, )
```

Le format des deux tables de caractéristiques et de cibles est représenté dans la [Figure 5.13](#).



[Figure 5.13](#) : Format des données pour Scikit-Learn.

Une fois les données correctement préparées, nous pouvons nous intéresser à l'interface API estimateur de Scikit-Learn.

API estimateur de Scikit-Learn

Voici les grandes règles qui ont présidé à la construction de l'interface API de Scikit-Learn :

Cohérence

Tous les objets partagent une même interface constituée d'un ensemble limité de méthodes, avec une bonne documentation.

Inspection

Toutes les valeurs de paramètres sont disponibles en tant qu'attributs publics.

Hiérarchie d'objets limitée

Seuls les algorithmes correspondent à des classes Python. Les jeux de données sont présentés dans des formats standard (tableaux de NumPy, structures DataFrame de Pandas, matrices éparses de SciPy) et les noms de paramètres sont des chaînes Python standard.

Composition

Un grand nombre d'actions d'apprentissage machine peut être exprimé sous forme d'une séquence d'algorithmes élémentaire, et Scikit-Learn propose cette approche dès que possible.

Valeurs par défaut sensées

Lorsqu'un modèle a besoin de paramètres spécifiés par le programmeur, une valeur par défaut acceptable est définie par la librairie.

Ces quelques principes sont à la base de la facilité d'emploi de Scikit-Learn, mais ils doivent être bien compris. Chacun des algorithmes d'apprentissage machine de Scikit-Learn utilise l'interface API Estimateur qui offre donc une approche cohérente pour un vaste nombre d'applications dans ce domaine.

Principes de l'interface API

Pour profiter de l'interface API de l'estimateur de Scikit-Learn, vous allez dérouler en général les cinq étapes suivantes (nous verrons des exemples détaillés dans la suite) :

1. Choix d'une classe de modèles en important la classe d'estimation appropriée depuis Scikit-Learn.
2. Sélection des hyperparamètres du modèle en créant une instance de la classe avec les bonnes valeurs.
3. Réorganisation éventuelle des données pour obtenir une matrice de caractéristiques et un vecteur cible comme indiqué ci-dessus.
4. Ajustement du modèle aux données par appel à la méthode fit() de l'instance du modèle.

5. Application du modèle à de nouvelles données :

- Dans le cadre d'un apprentissage supervisé, il s'agit en général de trouver des labels pour des données inconnues au moyen de la méthode predict().
- Dans le cadre d'un apprentissage non supervisé, il s'agit en général de transformer ou de déduire des propriétés des données au moyen d'une des deux méthodes transform() ou predict().

Découvrons quelques exemples d'application de ces méthodes pour l'apprentissage supervisé et non supervisé.

Apprentissage supervisé, exemple 1 : régression linéaire simple

Commençons par une régression linéaire simple qui consiste à ajuster une ligne à une paire de données en x, y. Nous choisissons de générer des données au hasard comme point de départ ([Figure 5.14](#)) :

In[5] :

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
rng = np.random.RandomState(42)
```

```
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```

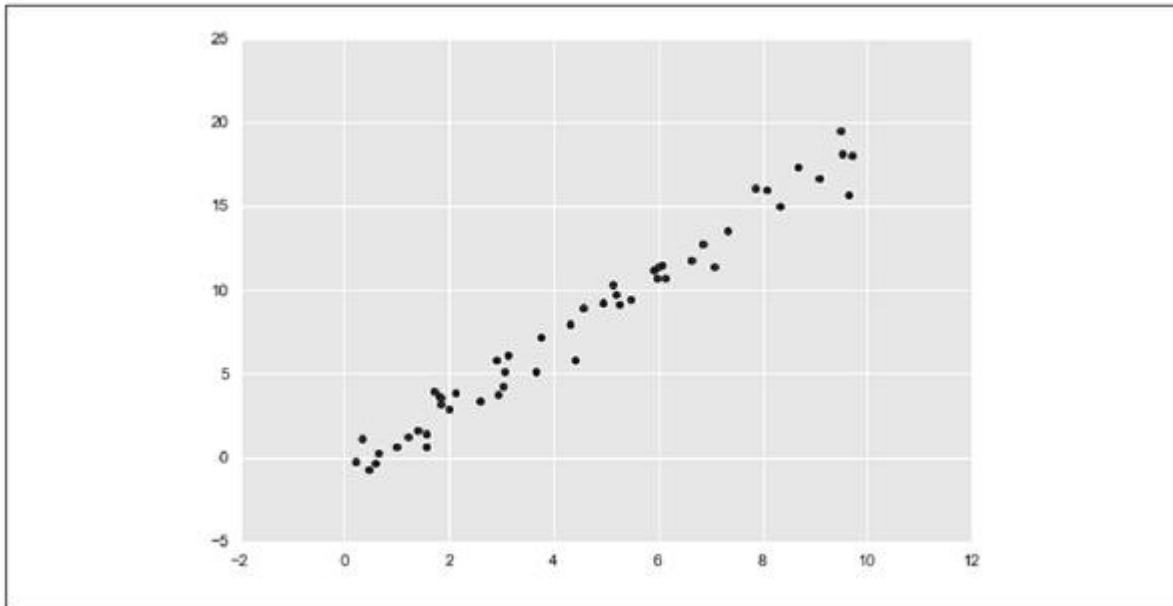


Figure 5.14 : Données d'entrée d'une régression linéaire.

Une fois ces données en place, nous allons appliquer la recette indiquée plus haut, en suivant les cinq étapes :

1. Choix de la classe de modèles.

Tous les modèles Scikit-Learn sont des classes Python. Pour notre régression linéaire, nous demandons d'importer la classe correspondante :

```
In[6]: from sklearn.linear_model import
LinearRegression
```

Il existe d'autres modèles de régression linéaire. Vous pouvez en apprendre plus en vous renseignant au sujet du module `sklearn.linear_model`.

2. Choix des hyperparamètres du modèle.

Il faut bien distinguer la classe de modèles et l'instance de modèle.

Plusieurs options sont disponibles au moment de créer l'instance. En fonction de la classe choisie, nous aurons à répondre à une ou plusieurs des questions suivantes :

- Veut-on ajuster pour le décalage offset (donc intercepter) ?
- Veut-on normaliser le modèle ?
- Veut-on prétraiter les caractéristiques pour rendre le modèle plus souple ?
- Quel degré de régularisation voulons-nous utiliser dans le modèle ?
- Combien de composants de modèle voulons-nous utiliser ?

Il ne s'agit ici que de quelques exemples de choix importants qui doivent être faits juste après sélection de la classe de modèles. Ces choix correspondent normalement à des

hyperparamètres qui doivent être réglés avant de tenter d'ajuster le modèle aux données. Dans Scikit-Learn, les hyperparamètres sont renseignés en fournissant des valeurs au moment de l'instanciation du modèle. Nous verrons dans la prochaine section de ce chapitre comment choisir leurs valeurs dans le cadre de la validation du modèle.

Nous avions dit que pour notre exemple de régression linéaire, nous allions créer une instance de la classe `LinearRegression`. Nous choisissons d'ajuster pour l'interception au moyen de l'hyperparamètre nommé `fit_intercept` :

In[7]:

```
model = LinearRegression(fit_intercept=True)
model
```

Out[7]:

```
LinearRegression(copy_X=True,
fit_intercept=True, n_jobs=1,
normalize=False)
```

L'action d'instanciation du modèle n'a pas d'autre effet que la mise en place des hyperparamètres valorisés. Le modèle n'a pas été appliqué à des données. Dans l'API de Scikit-Learn, il est clairement rappelé qu'il y a une différence entre choix du modèle et application du modèle aux données.

3. Réorganisation des données en une matrice de caractéristiques et un vecteur cible.

Nous avons dit que le format attendu par Scikit-Learn pour les données était celui d'une matrice de caractéristiques à deux dimensions et d'un tableau cible à une dimension. Notre variable cible `y` est déjà dans le bon format, celui d'un tableau de longueur `n_samples`. Il nous faut encore préparer les données d'entrée `X` pour obtenir une matrice au format `[n_samples, n_features]`. Dans notre exemple, cela se résume à reformer notre tableau qui est à une dimension :

In[8]:

```
x = x[:, np.newaxis]  
x.shape
```

Out[8]: (50, 1)

4. Ajustement du modèle aux données.

Nous pouvons maintenant appliquer le modèle en utilisant sa méthode `fit()` :

In[9]:

```
model.fit(x, y)
```

Out[9]:

```
LinearRegression(copy_X=True,
```

```
fit_intercept=True, n_jobs=1,  
normalize=False)
```

La méthode `fit()` provoque l'enchaînement de plusieurs calculs internes suivi du stockage des résultats dans des attributs spécifiques au modèle, valeur que vous pouvez ensuite explorer. Dans Scikit-Learn, tous les paramètres récoltés avec `fit()` portent des noms qui se terminent par un caractère souligné. Dans ce modèle linéaire, nous obtenons notamment :

```
In[10]: model.coef_
```

```
Out[10]: array([ 1.9776566])
```

```
In[11]: model.intercept_
```

```
Out[11]: -0.90331072553111635
```

Ces deux paramètres correspondent à la pente et au point d'interception de l'ajustement linéaire simple par rapport aux données. Les valeurs s'approchent de 2 pour la pente d'entrée et de -1 pour l'interception.

Les gens se demandent souvent quel est le degré d'incertitude dans ces paramètres de modèle. Généralement, Scikit-Learn ne prévoit aucun outil pour lancer des investigations au niveau des paramètres internes du modèle. Cette activité relève plus de la modélisation statistique que

de l'apprentissage machine. Ce qui intéresse l'apprentissage machine, c'est ce que le modèle va prédire. Si vous avez envie de plonger dans les détails du paramétrage d'ajustement d'un modèle, vous vous tournez vers d'autres outils, par exemple le paquetage Python nommé StatsModels (<http://statsmodels.sourceforge.net/>).

5. Application du modèle à de nouvelles données inconnues.

Une fois le modèle entraîné, ajusté, il ne reste plus qu'à l'évaluer en lui demandant de travailler avec des données non présentes dans le jeu d'entraînement. Dans Scikit-Learn, nous utilisons la méthode predict(). Pour notre exemple, notre « jeu de nouvelles données » va se résumer à une grille de valeurs x, et nous demanderons quelles valeurs y le modèle va pouvoir prédire :

```
In[12]: xfit = np.linspace(-1, 11)
```

Comme auparavant, nous devons reformater les valeurs x pour obtenir une matrice [n_samples, n_features]. Nous pouvons ensuite soumettre cette matrice à notre modèle :

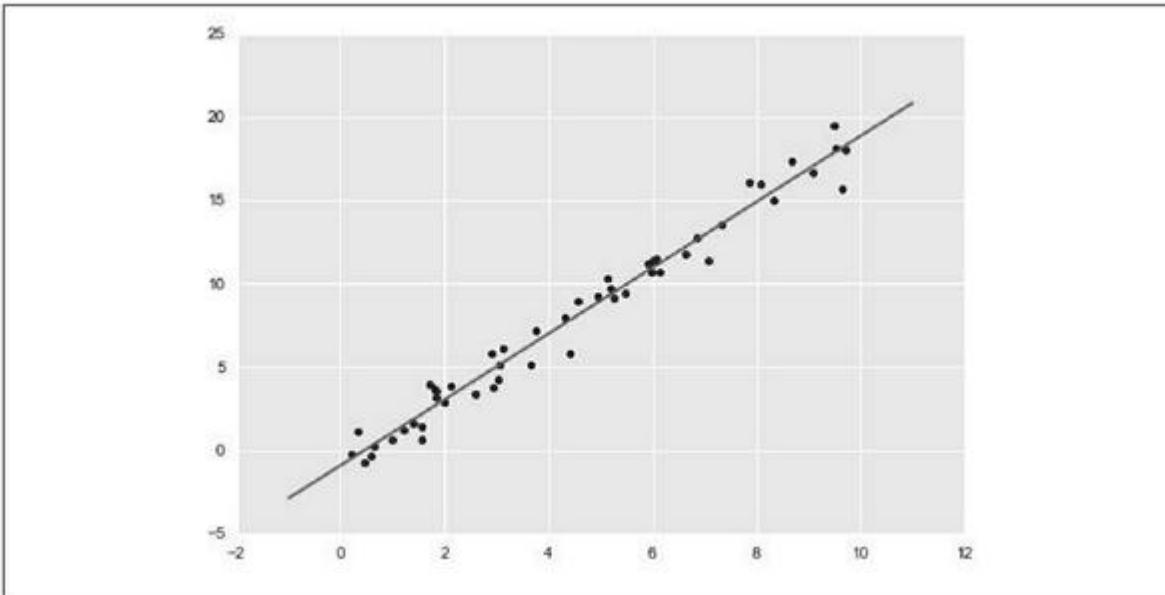
```
In[13]:  
xfit = xfit[:, np.newaxis]  
yfit = model.predict(xfit)
```

Il ne reste plus qu'à visualiser les résultats en affichant d'abord les données d'entrée brutes puis le résultat de l'ajustement au modèle ([Figure 5.15](#)) :

In[14]:

```
plt.scatter(x, y)
plt.plot(xfit, yfit);
```

Pour juger de l'efficacité d'un modèle, on procède généralement par comparaison des résultats à une situation de référence, ce que nous allons voir dans le prochain exemple.



[Figure 5.15](#) : Ajustement d'une régression linéaire simple à des données.

Apprentissage supervisé, exemple 2 : classification des iris

Découvrons maintenant la technique de classification en reprenant le jeu de données des Iris déjà rencontré. La question est la suivante : après avoir entraîné un modèle sur un sous-ensemble des données, avec quelle précision pouvons-nous prédire les noms des autres échantillons, c'est-à-dire leurs labels ?

Nous allons adopter un modèle générateur très simple, le modèle bayésien naïf gaussien. Dans ce modèle, il est supposé que chacune des classes est positionnée le long d'un axe d'une distribution gaussienne (une section de ce chapitre entre en détail dans la classification bayésienne

naïve). Ce modèle travaille très rapidement et ne demande de choisir aucun hyperparamètre, ce qui en fait un bon candidat pour obtenir une classification de référence initiale, à partir de laquelle nous pouvons tester des modèles plus complexes et voir s'ils apportent une augmentation de qualité du traitement.

Pour pouvoir vérifier le modèle, il nous faut l'alimenter avec des données que le modèle ne connaît pas encore. Nous allons donc créer deux jeux à partir des données d'entrée, un jeu d'entraînement et un jeu de test. Au lieu de procéder manuellement, nous nous servons de la fonction utilitaire `train_test_split()` :

```
In[15]:  
from sklearn.cross_validation import  
train_test_split  
Xtrain, Xtest, ytrain, ytest =  
train_test_split(X_iris,y_iris,  
random_state=1)
```

Les données étant prêtes, nous pouvons dérouler notre processus de prédiction des labels :

```
In[16]:  
from sklearn.naive_bayes import GaussianNB # 1.  
Choix classe modèle  
model = GaussianNB() # 2.
```

```
Instantiation  
model.fit(Xtrain, ytrain) # 3.  
Ajustement  
y_model = model.predict(Xtest) # 4.  
Prédiction sur inconnues
```

Nous nous servons de la fonction accuracy_score() pour voir le taux de véracité de nos labels prédits par rapport à leur vraie valeur que nous connaissons :

```
In[17]:  
from sklearn.metrics import accuracy_score  
accuracy_score(ytest, y_model)
```

```
Out[17]: 0.97368421052631582
```

Une valeur de 97 % est tout à fait honorable, ce qui fait de ce modèle pourtant simple une solution satisfaisante pour ce jeu de données en particulier !

Apprentissage non supervisé, exemple 1 : dimensionnalité des iris

Notre premier exemple non supervisé va consister à réduire d'abord la dimensionnalité des données des iris pour les rendre plus facilement visualisables. Souvenons-nous que ces données sont au départ à quatre dimensions : il y a quatre caractéristiques pour chaque échantillon.

Réduire le nombre de dimensions revient à chercher s'il est possible de diminuer le nombre de dimensions sans perdre les caractéristiques essentielles signifiantes. Cette réduction sert souvent à simplifier la visualisation, puisqu'il est évident qu'il est plus simple de visualiser deux dimensions que quatre ou plus !

Nous allons adopter ici la technique d'analyse par composantes principales *PCA* (*Principal Component Analysis*) à laquelle est dédiée une autre section du chapitre. C'est une technique de réduction dimensionnelle linéaire et rapide. Nous allons demander au modèle de produire deux composantes, c'est-à-dire une représentation des données en 2D.

Appliquons notre séquence de traitement standard :

In[18]:

```
#      1. Choix de la classe
from sklearn.decomposition import PCA

#      2. Instantiation avec hyperparamètres
model = PCA(n_components=2)

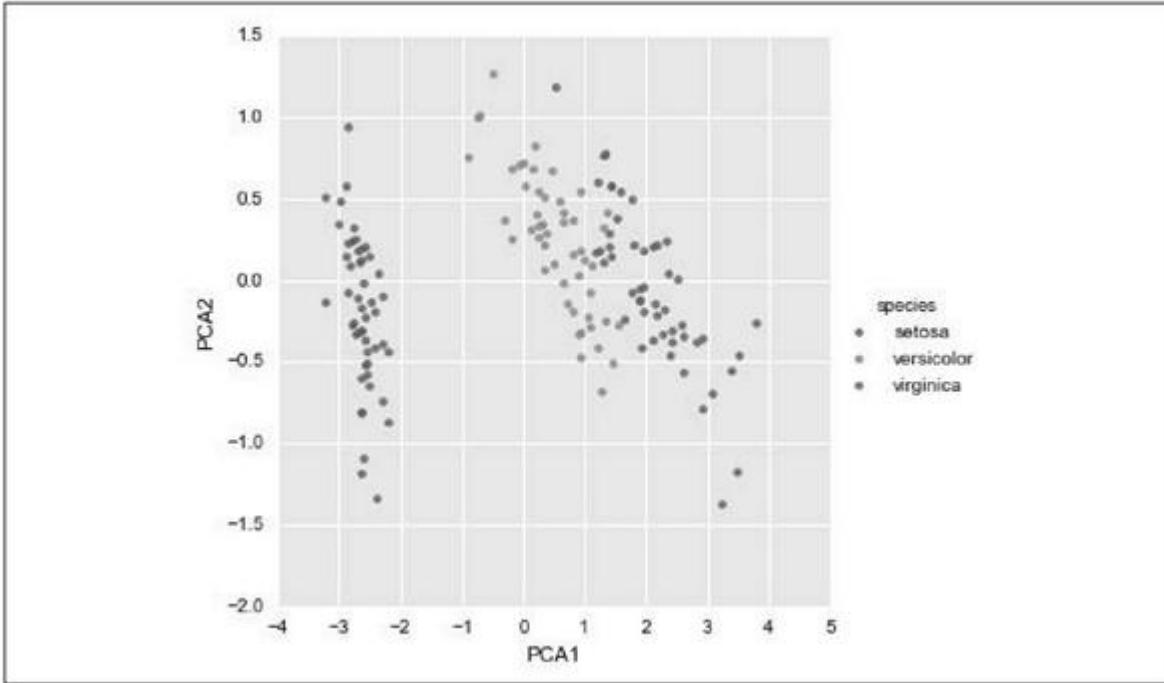
#      3. Ajustement. Notez que y n'est pas
#          mentionné !
model.fit(X_iris)
```

```
#      4. Transformation des données en 2D
X_2D = model.transform(X_iris)
```

Nous pouvons maintenant visualiser les résultats, ce qui se fait rapidement en insérant ceux-ci dans la structure DataFrame initiale des iris puis en utilisant lmplot() de Seaborn pour afficher les résultats ([Figure 5.16](#)) :

In[19]:

```
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot("PCA1", "PCA2", hue='species',
data=iris, fit_reg=False);
```



[Figure 5.16](#) : Les données des iris projetées en deux dimensions.

Dans cette vue en deux dimensions, nous constatons que les différentes espèces sont clairement distinctes, alors que l'algorithme PCA n'a même pas connaissance des labels des différentes espèces. Cela nous montre qu'une classification assez simple saura se montrer efficace avec ce jeu de données, ce que nous savions déjà.

Apprentissage non supervisé, exemple 2 : regroupement des iris

Tentons maintenant un regroupement (clustering) des données. Un algorithme de regroupement cherche à distinguer des agrégats dans les données sans s'appuyer sur les labels. Nous allons adopter une méthode de

regroupement très puissante qui est le modèle mixte gaussien *GMM* (*Gaussian Mixture Model*). Comme les autres techniques, une section de ce chapitre aborde ce modèle en détail. GMM essaye de modéliser les données sous la forme d'une collection de *condensats* ou *blobs* gaussiens.

Procédons à l'ajustement de notre modèle GMM :

In[20]:

```
# 1. Choix de la classe
from sklearn.mixture import GMM

# 2. Instanciation avec hyperparamètres
model = GMM(n_components=3,
covariance_type='full')

# 3. Ajustement. Notez que y n'est pas
mentionné !
model.fit(X_iris)

# 4. Prédiction des labels de groupes
y_gmm = model.predict(X_iris)
```

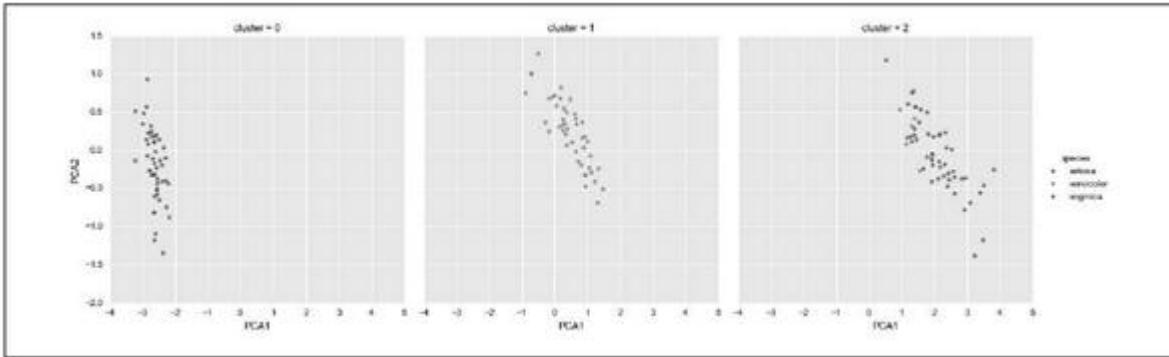
Comme précédemment, nous ajoutons les labels de groupes à la structure DataFrame de Iris puis utilisons Seaborn pour visualiser les résultats ([Figure 5.17](#)) :

In[21]:

```
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris,
```

```
hue='species',  
        col='cluster', fit_reg=False);
```

La répartition des données par numéro de groupe permet de bien juger comment l'algorithme GMM a trouvé les labels sous-jacents : l'espèce **setosa** est clairement regroupée dans le groupe 0, mais il reste encore un peu de mélange entre les espèces **versicolor** et **virginica**. Autrement dit, nous n'avons même pas besoin d'un expert pour différencier les labels des espèces car les mesures disponibles pour les échantillons de fleurs sont suffisamment parlantes pour identifier *automatiquement* l'existence des différents groupes avec ce simple algorithme de regroupement ! De plus, ce genre d'algorithme peut aider les experts du domaine à retrouver des indices expliquant les relations entre les différents échantillons observés.



[Figure 5.17](#) : Résultat d'un regroupement des données Iris.

Application pratique : reconnaissance de chiffres manuscrits

Voyons comment appliquer ces principes à un problème concret : reconnaître des chiffres manuscrits. En situation réelle, cela suppose d'abord de délimiter les caractères dans une image avant de les identifier. Pour aller plus vite, nous allons partir d'un jeu de chiffres manuscrits prédéfini dans Scikit-Learn et intégré à la librairie.

Chargement et visualisation des données d'entrée

Nous nous servons de l'interface d'accès aux données de Scikit-Learn pour voir à quoi ressemblent les données d'entrée :

In[22]:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
```

Out[22]:

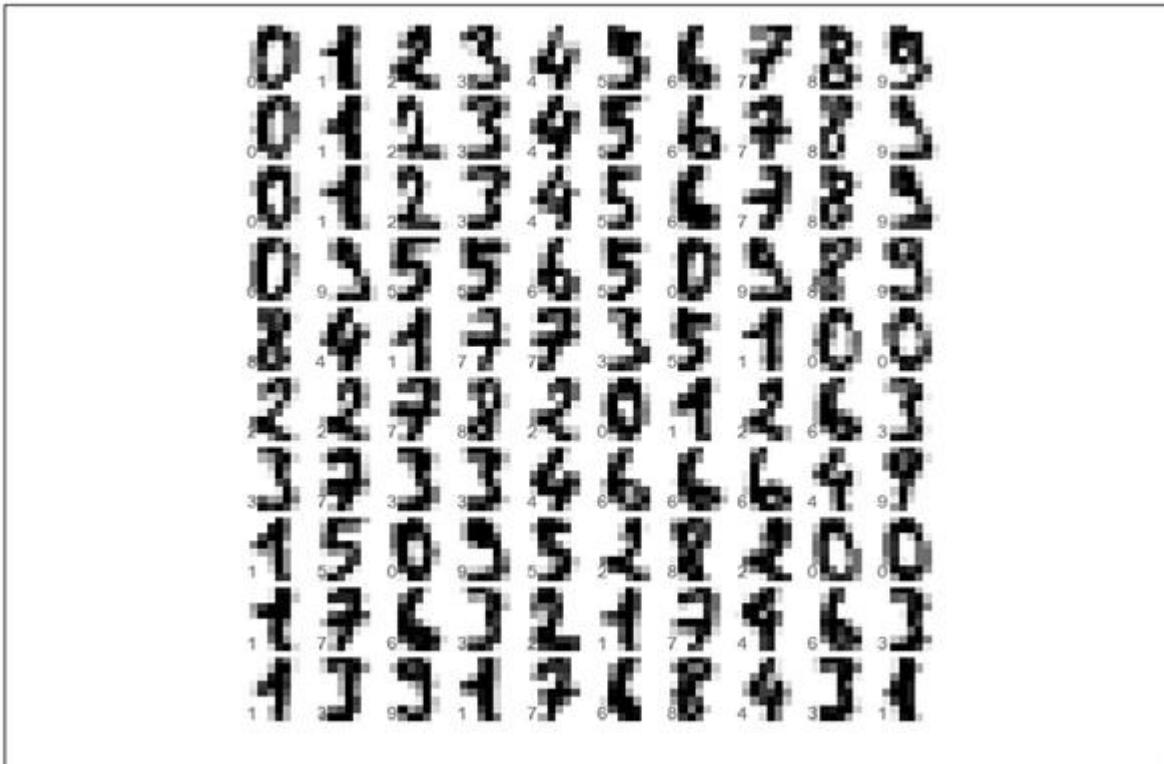
```
(1797, 8, 8)
```

Il s'agit d'un tableau à trois dimensions contenant 1797 échantillons, chacun étant incarné par une grille de 8 pixels sur 8. Affichons les cent premiers ([Figure 5.18](#)) :

In[23]:

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                       subplot_kw={'xticks':[],
'yticks':[]},
gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary',
interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes,
color='green')
```



[Figure 5.18](#) : Aperçu des données des chiffres manuscrits ; chaque échantillon correspond à une grille de 8 pixels sur 8.

Pour exploiter ces données avec Scikit-Learn, nous devons disposer d'une représentation en deux dimensions [n_samples, n_features]. Une solution consiste à considérer que chaque pixel d'image est une caractéristique, ce qui suppose donc d'aplatir le tableau pour obtenir un tableau à une dimension d'une longueur de 64 pour chaque chiffre. Il nous faut également préparer le tableau cible dans lequel sera stocké le label déterminé pour chaque chiffre. Ces deux nouvelles colonnes de données sont incorporées au jeu de chiffres au moyen des attributs nommés data et target, respectivement :

```
In[24]:
```

```
X = digits.data  
X.shape
```

```
Out[24]: (1797, 64)
```

```
In[25]:
```

```
y = digits.target  
y.shape
```

```
Out[25]: (1797, )
```

Nous confirmons qu'il y a bien 1 797 échantillons et 64 caractéristiques.

Apprentissage non supervisé : réduction dimensionnelle

Comme il est difficile de visualiser des points dans un espace paramétrique à 64 dimensions, nous allons ramener à 2 le nombre de dimensions en nous servant d'une méthode non supervisée. Nous adoptons l'algorithme d'apprentissage par variété (*manifold*) portant le nom **Isomap** (une section du chapitre aborde cette classe d'algorithmes). Nous obtenons ainsi deux dimensions :

```
In[26]:
```

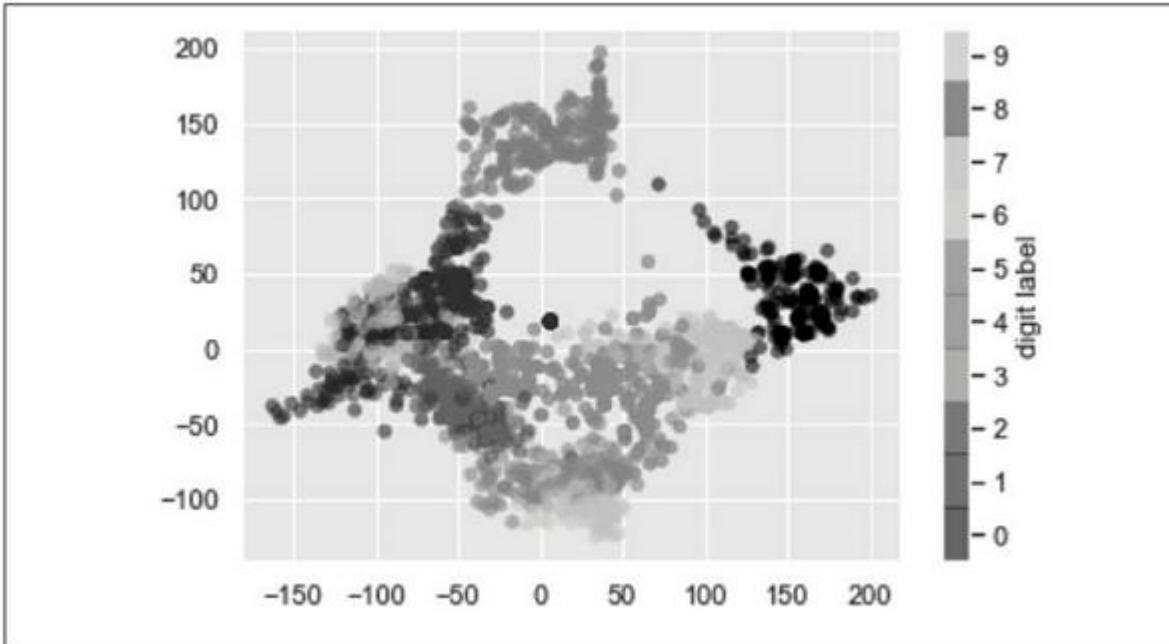
```
from sklearn.manifold import Isomap  
iso = Isomap(n_components=2)
```

```
iso.fit(digits.data)
data_projected = iso.transform(digits.data)
data_projected.shape
```

Out[26]: (1797, 2)

Voyons comment se présentent les deux dimensions que nous avons obtenues et cherchons à savoir si nous pouvons apprendre quelque chose de la structure ([Figure 5.19](#)) :

```
In[27]:
plt.scatter(data_projected[:, 0],
            data_projected[:, 1], c=digits.target,
            edgecolor='none', alpha=0.5,
            cmap=plt.cm.get_cmap('nipy_spectral',
10))
plt.colorbar(label='digit label',
ticks=range(10))
plt.ylim(-0.5, 9.5);
```



[Figure 5.19](#) : Traitement Isomap des chiffres manuscrits.

Ce tracé nous donne une première idée intéressante quant à la répartition des différents chiffres dans l'espace à 64 dimensions. (La version en couleurs du calepin est plus pratique pour suivre.) Les zéros qui sont en noir et les un qui sont en mauve sont largement distincts dans l'espace paramétrique. Ce n'est pas surprenant, puisqu'un zéro contient un vide au milieu alors qu'un un contient de l'encre dans sa partie centrale. En revanche, il n'en va pas de même entre les un et les quatre, ce qui se comprend du fait que certaines personnes ajoutent une coche en biais dans le haut des uns, ce qui les fait plus ressembler à des quatre.

Globalement, les différents groupes semblent néanmoins assez bien distincts, ce qui nous laisse croire qu'un

algorithme d'apprentissage supervisé assez simple devrait convenir à ces données. Vérifions cela.

Classification de chiffres manuscrits

Tentons d'utiliser un algorithme de classification. Comme avec les fleurs, nous allons d'abord répartir les données en un jeu de tests et un jeu d'entraînement puis appliquer un modèle bayésien naïf gaussien :

In[28]:

```
Xtrain, Xtest, ytrain, ytest =  
train_test_split(X, y, random_state=0)
```

In[29]:

```
from sklearn.naive_bayes import GaussianNB  
model = GaussianNB()  
model.fit(Xtrain, ytrain)  
y_model = model.predict(Xtest)
```

Une fois la prédiction réalisée, nous pouvons vérifier sa précision en comparant nos valeurs trouvées aux valeurs véritables du jeu de test :

In[30]:

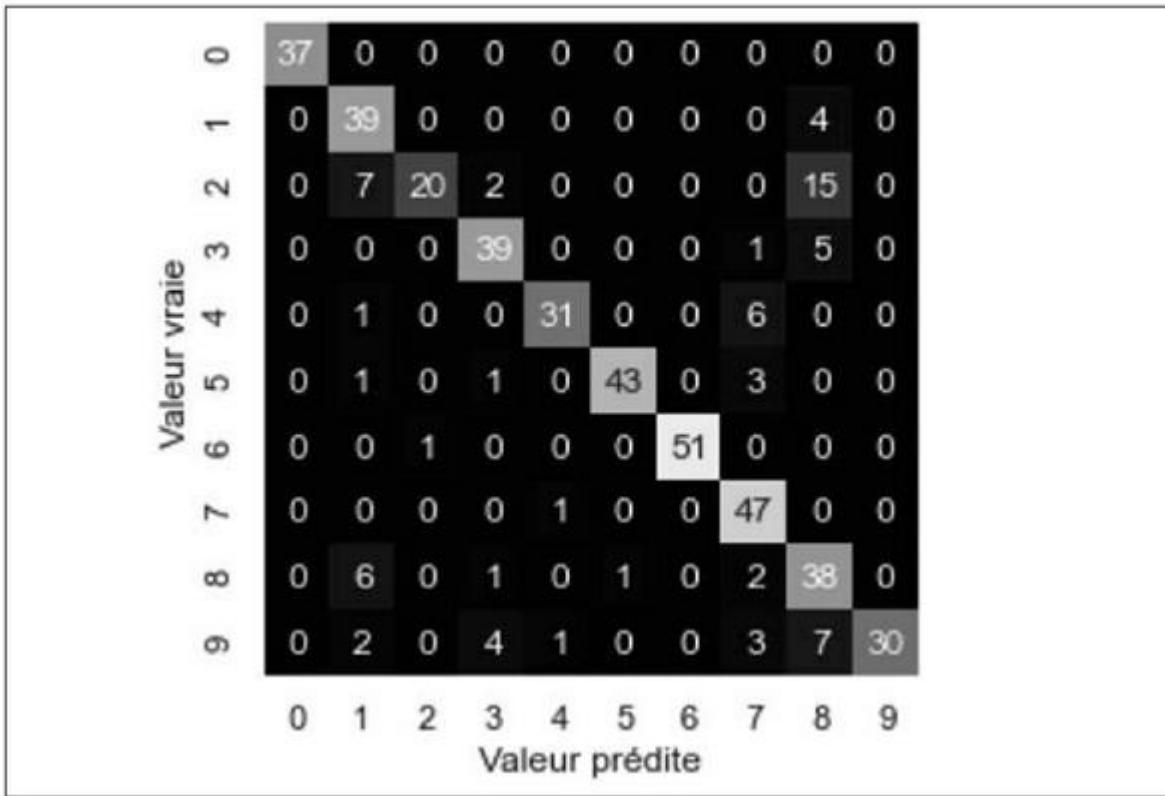
```
from sklearn.metrics import accuracy_score  
accuracy_score(ytest, y_model)
```

Out[30]: 0.8333333333333337

Ce modèle très simple permet néanmoins d'atteindre 80 % de précision pour la classification ! Mais cette valeur ne nous dit pas où il y a erreur. Une technique efficace pour en apprendre plus consiste à utiliser une *matrice de confusion* que nous allons faire calculer par Scikit-Learn puis visualiser par Seaborn ([Figure 5.20](#)) :

In[31]:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, y_model)
sns.heatmap(mat, square=True, annot=True,
cbar=False)
plt.xlabel('Valeur prédite')
plt.ylabel('Valeur vraie');
```



[Figure 5.20](#) : Matrice de confusion montrant la fréquence d'erreurs de classification.

Nous repérons les points litigieux : un grand nombre de deux sont confondus avec des un ou des huit. Pour en apprendre encore plus sur les caractéristiques du modèle, nous pouvons demander une nouvelle visualisation avec les labels prédits. Nous affichons en vert les labels corrects et en rouge ceux qui sont erronés ([Figure 5.21](#)) :

In[32]:

```
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                        subplot_kw={'xticks':[],
'yticks':[]},
```

```

gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary',
interpolation='nearest')
    ax.text(0.05, 0.05, str(y_model[i]),
            transform=ax.transAxes,
            color='green' if (ytest[i] ==
y_model[i]) else 'red')

```

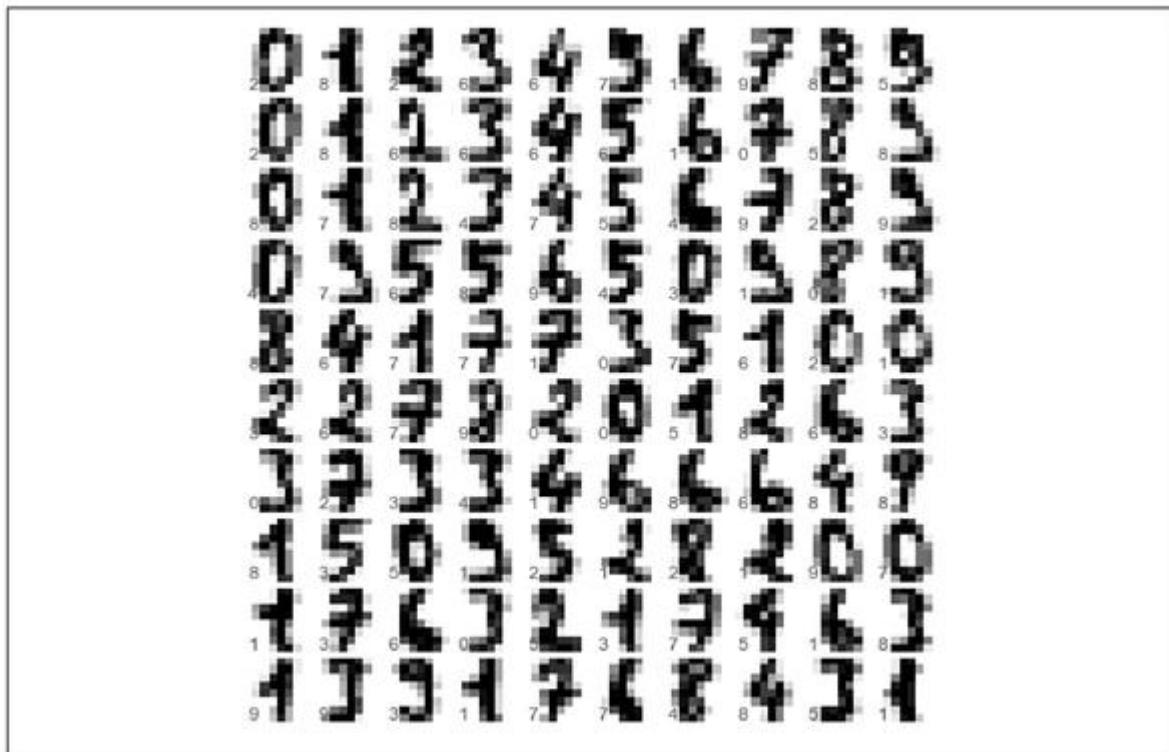


Figure 5.21 : Données correctes en vert et incorrectes en rouge pour les labels.



(N.d.T) : Une version en couleurs de toutes les visualisations que cela justifie est disponible sous forme d'un fichier PDF

faisant partie des éléments téléchargeables de la version française (sur le site www.editionsfirst.fr).

En étudiant cette sélection de données, nous pouvons déterminer quand l'algorithme fonctionne mal. Pour réussir à avoir plus de 80 % de réussite, nous pourrions utiliser un algorithme plus sophistiqué, par exemple une machine à vecteurs de support ou des forêts aléatoires, présentés en détail dans la suite du chapitre, ou bien une autre technique de classification.

Résumé

Nous venons de présenter les fonctions principales de représentation des données par Scikit-Learn et nous avons vu l'interface API de l'estimateur. Quel que soit le type d'estimation, il s'agit toujours d'utiliser la même séquence importation/instanciation/ajustement/prédiction. Le savoir-faire concernant l'estimateur permet d'assimiler la suite des informations concernant Scikit-Learn ; vous pouvez même commencer à appliquer différents modèles à vos données.

Abordons maintenant un sujet fondamental en apprentissage machine : les conditions de sélection et de validation du modèle.

5.3 : Hyperparamètres et validation du modèle

Nous avons présenté et utilisé la séquence principale d'exploitation d'un modèle d'apprentissage machine :

1. Choix d'une classe de modèles.
2. Choix des hyperparamètres du modèle.
3. Organisation des données en matrice de caractéristiques et vecteurs cibles.
4. Ajustement du modèle aux données d'entraînement.
5. Utilisation du modèle pour prédire les labels de données nouvelles.

L'utilisation efficace de ces outils et techniques suppose de travailler avec soin sur les deux premières étapes que sont le choix du modèle et celui des hyperparamètres. Pour bien choisir, il est nécessaire de pouvoir valider le modèle et les hyperparamètres face aux données. Cela semble simple, mais plusieurs pièges sont à éviter.

Le concept de validation de modèle

Le principe de validation d'un modèle est très simple : on choisit un modèle et ses hyperparamètres puis on estime

son efficacité en l'appliquant à des données d'entraînement et en comparant la prédiction ainsi faite aux valeurs réelles.

Commençons par une approche simpliste de cette validation et voyons pourquoi elle échoue. Nous verrons ensuite comment réaliser des évaluations de modèle plus fiables en utilisant les *jeux de rétention holdout* et la validation croisée.

Validation de modèle erronée

Repartons de notre jeu de données d'iris pour faire la démonstration de l'approche de validation trop naïve. Commençons par charger les données :

```
In[1]:  
from sklearn.datasets import load_iris  
iris = load_iris()  
X = iris.data  
y = iris.target
```

Nous choisissons ensuite le modèle et ses hyperparamètres. Ici, nous optons pour le modèle de classification à k-voisins avec une valeur n_neighbors=1. Ce modèle très intuitif et ultrasimple se fonde sur le principe que « le label d'un point inconnu doit être le même que le label de son point d'entraînement le plus proche » :

In[2] :

```
from sklearn.neighbors import  
KNeighborsClassifier  
model = KNeighborsClassifier(n_neighbors=1)
```

Nous entraînons le modèle puis lui demandons de prédire les labels des données que nous connaissons déjà :

In[3] :

```
model.fit(X, y)  
y_model = model.predict(X)
```

Nous demandons enfin de calculer le taux de réussite :

In[4] :

```
from sklearn.metrics import accuracy_score  
accuracy_score(y, y_model)
```

Out[4] : 1.0

Fantastique ! Nous aurions obtenu 100 % de prédictions correctes ! Mais est-ce que nous mesurons réellement la précision attendue ? Avons-nous réellement trouvé un modèle qui sera juste à 100 % tout le temps ?

Vous devinez que c'est absolument faux. Cette approche comporte un défaut fondamental : elle effectue l'entraînement puis la prédiction du modèle à partir des mêmes données. Et ceci sans compter sur le fait que ce

modèle basé sur les plus proches voisins utilise un estimateur basé sur les instances qui se contente de stocker les données d'entraînement pour prédire les labels en comparant les nouvelles données à ces données stockées. Sauf dans des cas très particuliers, il n'a aucun mal à obtenir 100 % de précision à tous les coups.

Validation de modèle avec jeu de rétention

Que peut-on faire ? Pour mieux juger des performances d'un modèle, on peut utiliser la technique du *jeu de rétention* qui consiste à soustraire un sous-ensemble des données d'entraînement puis à utiliser ce jeu de rétention pour effectuer le contrôle de performances. Nous nous servons de la fonction `train_test_split()` de Scikit-Learn pour répartir les données :

```
In[5]:  
from sklearn.cross_validation import  
train_test_split  
# Divise le jeu de données en 2 fois 50 %  
X1, X2, y1, y2 = train_test_split(X, y,  
random_state=0, train_size=0.5)  
  
# Ajuste le modèle sur un sous-jeu  
model.fit(X1, y1)  
  
# Évalue le modèle sur le second sous-jeu  
y2_model = model.predict(X2)
```

```
accuracy_score(y2, y2_model)
```

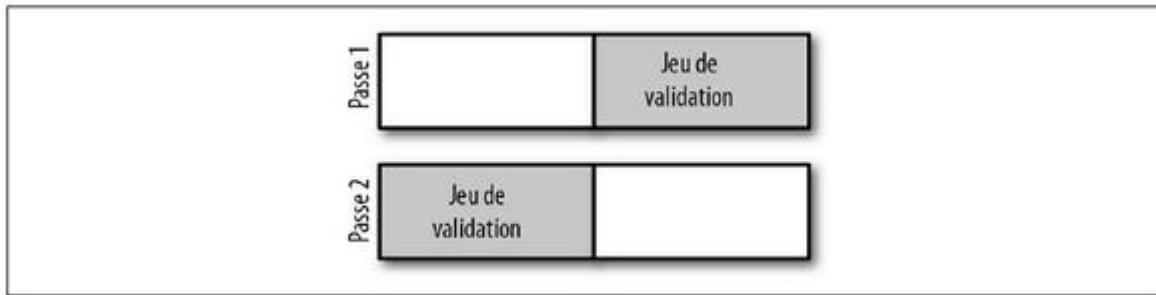
```
Out[5]: 0.9066666666666662
```

Nous obtenons un résultat beaucoup plus réaliste : ce classificateur à plus proches voisins offre une précision de 90 % sur le jeu de rétention. Ce jeu ressemble à des données inconnues, puisque le modèle ne l'a pas rencontré lors de l'entraînement.

Validation de modèle par validation croisée

La validation d'un modèle en utilisant l'astuce du jeu de rétention pose un problème : une partie des données n'est plus utilisée pour l'entraînement. Dans notre exemple, la moitié des données ne sert plus à entraîner le modèle, ce qui est loin d'être optimal. Le problème est particulièrement gênant lorsque le volume de données d'entrée est faible.

Une solution consiste à faire de la validation croisée, c'est-à-dire à utiliser alternativement une partie du jeu d'entraînement comme jeu de rétention puis l'autre, en travaillant en deux passes. Cela donne schématiquement ce que montre la [Figure 5.22](#).



[Figure 5.22](#) : Visualisation d'une validation croisée en deux passes.

Nous utilisons effectivement la moitié du jeu d'entraînement comme jeu de rétention. Voici comment nous pouvons exprimer cela dans le code à partir des données disponibles :

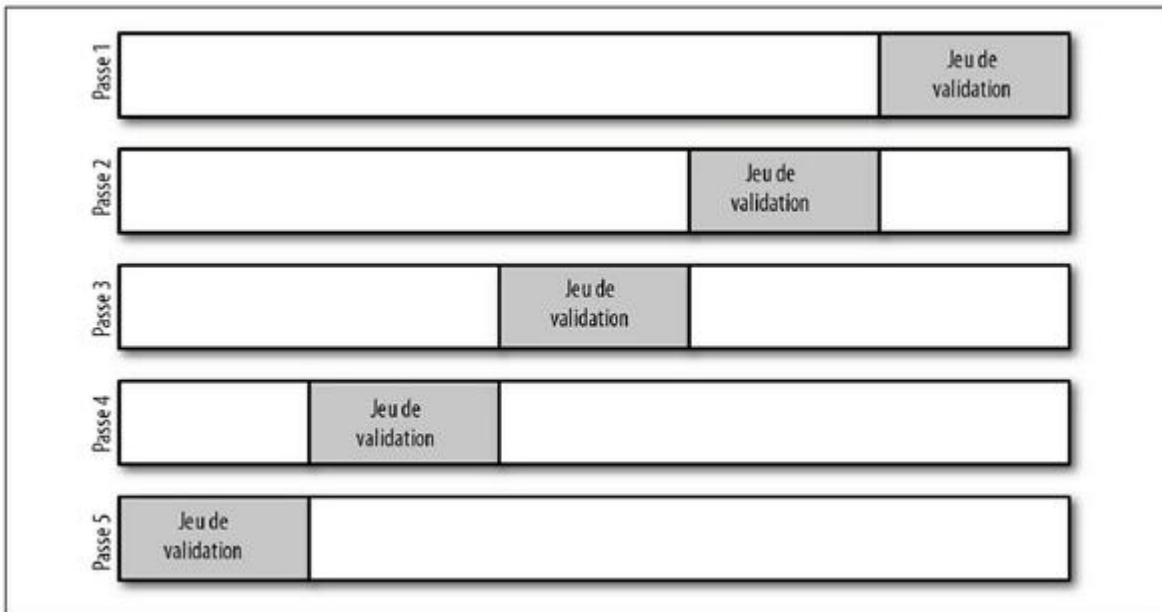
In[6] :

```
y2_model = model.fit(X1, y1).predict(X2)
y1_model = model.fit(X2, y2).predict(X1)
accuracy_score(y1, y1_model), accuracy_score(y2,
y2_model)
```

Out[6] : (0.9599999999999996,
0.9066666666666662)

Fort logiquement, nous obtenons deux scores de précision que nous pouvons combiner par exemple en demandant leur moyenne pour mieux évaluer la performance globale du modèle. Il s'agit ici d'une validation croisée en deux passes.

Nous pouvons augmenter le nombre de passes. La [Figure 5.23](#) montre une validation croisée en cinq passes.



[Figure 5.23](#) : Visualisation d'une validation croisée en cinq passes.

Dans cette approche, les 4/5 des données d'entraînement sont effectivement utilisées lors de chaque passe pour entraîner le modèle. La préparation des jeux serait fastidieuse à la main ; nous profitons de la fonction utilitaire `cross_val_score()` de Scikit-Learn pour y parvenir :

In[7] :

```
from sklearn.cross_validation import
cross_val_score
cross_val_score(model, X, y, cv=5)
```

Out[7]: array([0.96666667, 0.96666667,
0.93333333, 0.93333333, 1.])

En validant avec des sous-ensembles de données différents, nous obtenons une idée beaucoup plus fidèle des

performances de l'algorithme.

Scikit-Learn propose plusieurs schémas de validation croisée incarnés par des itérateurs dans le module `cross_validation`. On peut pousser le raisonnement à l'extrême en décidant qu'un seul point sert de validation, tous les autres servant à l'entraînement. Ce genre de validation croisée porte le nom de tout sauf un (*leave-one-out*). Voici comment s'en servir :

In[8]:

```
from sklearn.cross_validation import LeaveOneOut
scores = cross_val_score(model, X, y,
cv=LeaveOneOut(len(X)))
scores
```

Out[8]:

```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,  1.,
1.,  1.,  1.,
         1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,
```

```
1., 1., 1.,
    1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.,
    1., 0., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.,
    1., 0., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.,
    1., 1., 0., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1.,
    1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Nous disposons de 150 échantillons, et nous lançons donc 150 essais. Le score de chaque essai indique soit une réussite (1.0), soit un échec (0.0). Nous obtenons une estimation du taux d'erreurs en demandant la moyenne de ces valeurs :

In[9]: `scores.mean()`

Out[9]: 0.9599999999999996

Les autres schémas de validation croisée s'utilisent de la même façon. Pour en connaître plus à leur sujet, utilisez IPython pour explorer le sous-module `sklearn.cross_validation` ou bien renseignez-vous dans la section correspondante de la documentation de Scikit-Learn.

Sélection du meilleur modèle

Après la découverte des fondamentaux de la validation et la validation croisée, plongeons plus en détail dans la sélection du modèle et de ses hyperparamètres. Ce sont selon moi des aspects souvent négligés dans les présentations des techniques d'apprentissage machine.

La question cruciale est la suivante : si notre estimateur donne de mauvais résultats, comment devons-nous progresser ? Plusieurs pistes sont ouvertes :

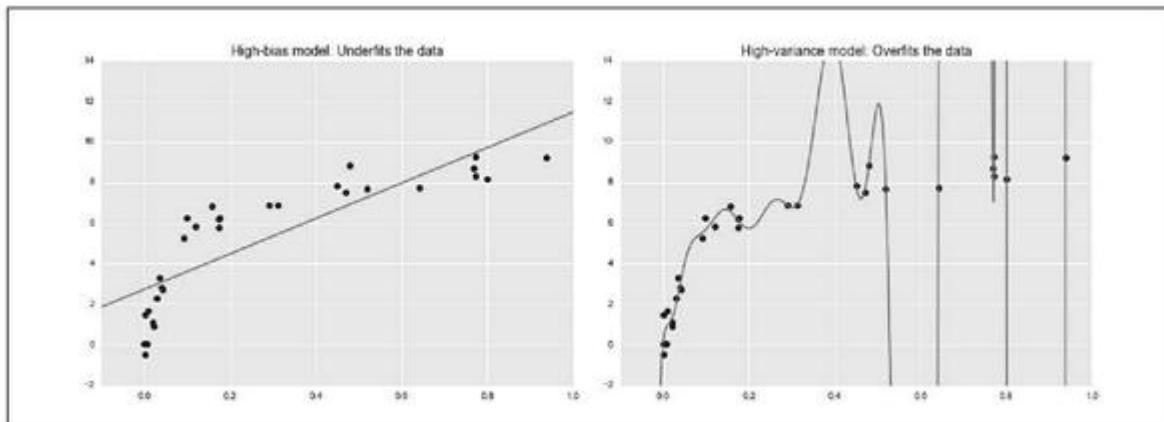
- opter pour un modèle plus complexe ou plus souple ;
- opter au contraire pour un modèle moins complexe et moins souple ;
- recueillir un plus grand nombre d'échantillons pour l'entraînement ;
- recueillir plus de données pour augmenter le nombre de caractéristiques de chaque échantillon.

Dans tous les cas, la réponse à cette question est contraire à l'intuition. Parfois, en adoptant un modèle plus complexe, on récolte des résultats encore pires, et le fait d'ajouter des échantillons au jeu d'entraînement n'améliore pas les résultats ! Savoir comment procéder pour améliorer un

modèle permet de distinguer le débutant en apprentissage machine de l'expert.

Arbitrage entre biais et variance

Trouver le modèle idéal revient à chercher un point d'équilibre entre biais et variance. Étudions la [Figure 5.24](#) qui propose deux ajustements par régression pour le même jeu de données.



[Figure 5.24](#) : Un modèle de régression avec fort biais ou forte variance.

Aucun des deux modèles n'est particulièrement bien ajusté aux données, mais leurs défauts ne sont pas identiques.

Le modèle de gauche cherche à tracer une ligne droite pour séparer les données. Comme ces données sont plus complexes, cette ligne droite ne permettra jamais de bien décrire ce jeu de données. Un tel modèle est dit sous-ajusté aux données : il n'offre pas une souplesse suffisante pour

prendre en compte toutes les caractéristiques. On parle d'un *modèle à biais fort*.

Le modèle de droite tente d'appliquer un polynôme à ordre élevé aux données. Il offre suffisamment de souplesse pour épouser les moindres caractéristiques des données, mais sa forme exacte semble trop tenir compte des propriétés du bruit qui accompagne les données, au lieu de se cantonner aux propriétés fondamentales qui reflètent le processus qui a servi à créer ces données. Ce modèle pêche par surajustement. Il est tellement souple qu'il finit par tenir compte d'erreurs aléatoires en plus de la distribution des données. Ce modèle est à *variance forte*.

Prenons un autre point de vue sur ces deux modèles en essayant de prédire les valeurs y pour des nouvelles données. La [Figure 5.25](#) comporte des points en gris plus clair qui correspondent aux données qui ont été retranchées du jeu d'entraînement.

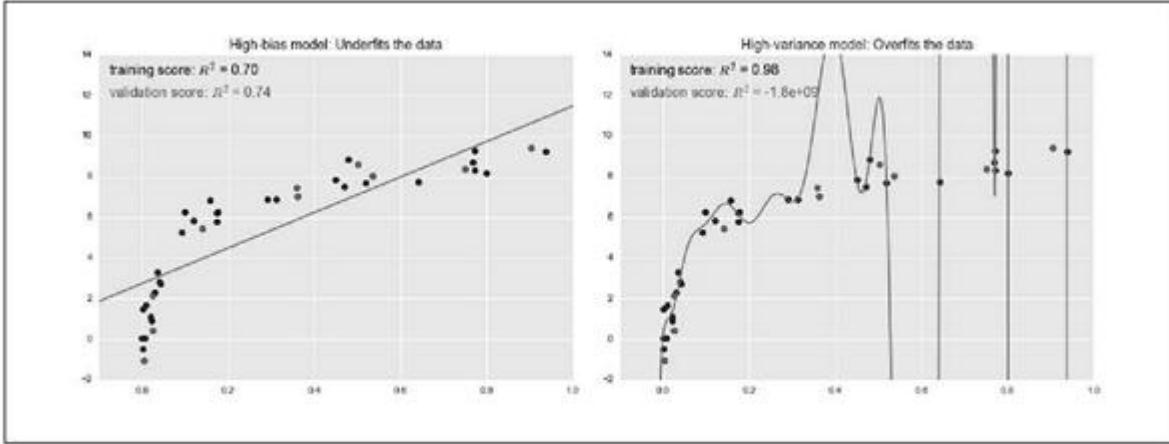


Figure 5.25 : Score d'entraînement et de validation pour un modèle à biais fort et à variance forte.

Le score mentionné ici est le coefficient de détermination ou score

R² (https://fr.wikipedia.org/wiki/Coefficient_de_d%C3%A9terminati on_n). Ce score permet de connaître l'efficacité du modèle en comparaison d'une simple moyenne des valeurs cibles. Une valeur de R² égale à 1 indique une correspondance parfaite et une valeur nulle signifie que le modèle ne fait pas mieux qu'une simple moyenne. Une valeur négative avoue que le modèle est encore pire qu'une moyenne. À la lecture des scores de ces deux modèles, nous pouvons émettre une observation plus générale :

- Dans le cas des modèles à biais fort, les performances sur le jeu de validation sont similaires à celles sur le jeu d'entraînement.

- Dans le cas des modèles à variance forte, les performances sur le jeu de validation sont beaucoup moins bonnes que celles sur le jeu d'entraînement.

Nous supposons qu'il est possible de régler la complexité du modèle d'une façon ou d'une autre, nous devrions pouvoir aboutir à ce que les scores d'entraînement et de validation se présentent comme montré dans la [Figure 5.26](#).

Le diagramme de la figure montre une *courbe de validation* dont voici les points remarquables :

- Le score d'entraînement est toujours supérieur au score de validation, ce qui est en général le cas : le modèle s'ajustera toujours mieux aux données qu'il connaît déjà qu'aux nouvelles données.
- Dans le cas d'un modèle à biais fort et à très faible complexité, il est sous-ajusté par rapport aux données d'entraînement. Ce modèle va mal prédire aussi bien les données d'entraînement que les nouvelles données.
- Dans le cas d'un modèle à variance forte et grande complexité, le modèle est surajusté aux données d'entraînement. Il les prédit très bien, mais en revanche, il prédit très mal les labels pour des données qu'il ne connaît pas encore.

- Le maximum de la courbe de validation se situe à peu près au milieu de cette courbe. C'est à cet endroit que l'on trouve le point d'équilibre entre biais et variance.

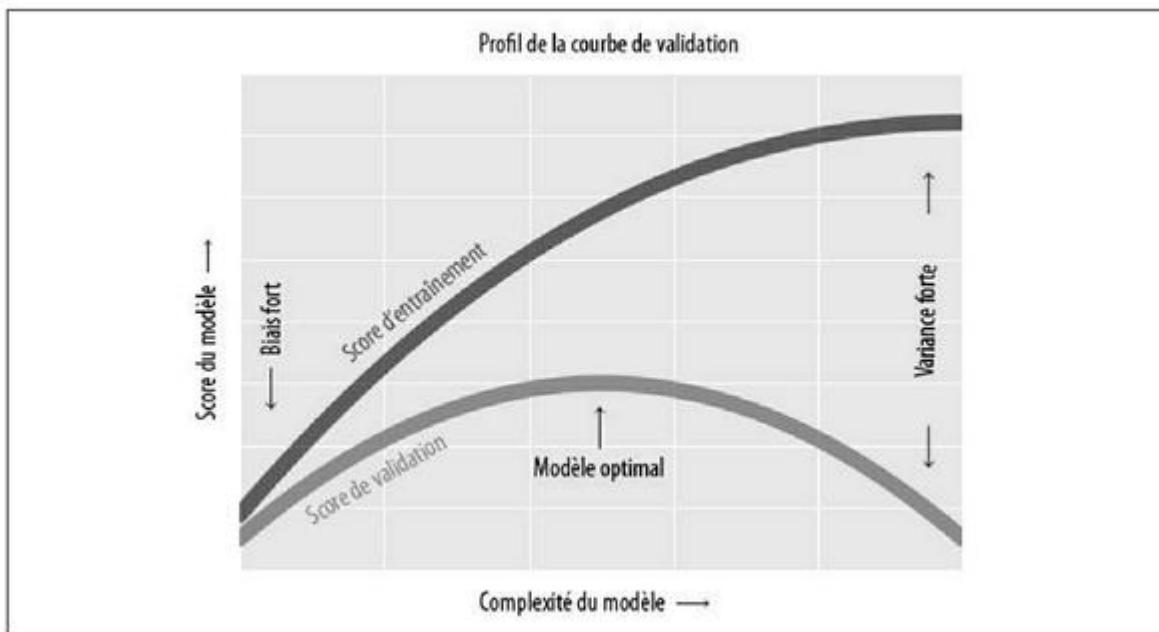


Figure 5.26 : Schéma des relations entre complexité de modèle, score d'entraînement et score de validation.

Les approches permettant de trouver la complexité idéale diffèrent d'un modèle à l'autre. Nous verrons comment réaliser ce genre d'optimisation dans chacune des sections dédiées à un modèle dans la suite du chapitre.

Courbe de validation dans Scikit-Learn

Découvrons un exemple dans lequel nous cherchons à calculer la courbe de validation d'une classe de modèles en utilisant une validation croisée. Nous nous servons d'un

modèle à régression polynomiale, c'est-à-dire un modèle linéaire généralisé dans lequel le degré de polynôme est ajustable. Par exemple, un polynôme de degré 1 correspond à une ligne droite par rapport aux données. En considérant deux paramètres de modèle nommés a et b :

$$y = ax + b$$

De même, un polynôme de degré 3 va ajuster une courbe cubique aux données. Les paramètres de modèle a , b , c et d sont exploités ainsi :

$$y = ax^3 + bx^2 + cx + d$$

Ce raisonnement peut être généralisé à un nombre quelconque de caractéristiques polynomiales. Nous pouvons, dans Scikit-Learn, utiliser une régression linéaire simple combinée à un préprocesseur de polynômes. Nous relions les opérations dans un pipeline de traitement (nous verrons les caractéristiques polynomiales et les pipelines dans la section suivante qui aborde l'ingénierie des caractéristiques) :

In[10]:

```
from sklearn.preprocessing import
PolynomialFeatures
from sklearn.linear_model import
LinearRegression
from sklearn.pipeline import make_pipeline

def PolynomialRegression(degree=2, **kwargs):
```

```
    return
make_pipeline(PolynomialFeatures(degree),
LinearRegression(**kwargs))
```

Nous générerons des données auxquelles le modèle devra s'ajuster :

In[11]:

```
import numpy as np

def make_data(N, err=1.0, rseed=1):
    # Sélection aléatoire des données
    rng = np.random.RandomState(rseed)
    X = rng.rand(N, 1) ** 2
    y = 10 - 1. / (X.ravel() + 0.1)
    if err > 0:
        y += err * rng.randn(N)
    return X, y

X, y = make_data(40)
```

Il ne reste plus qu'à visualiser les données avec des ajustements polynomiaux pour plusieurs degrés ([Figure 5.27](#)) :

In[12]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # plot formatting
```

```

X_test = np.linspace(-0.1, 1.1, 500)[:, None]

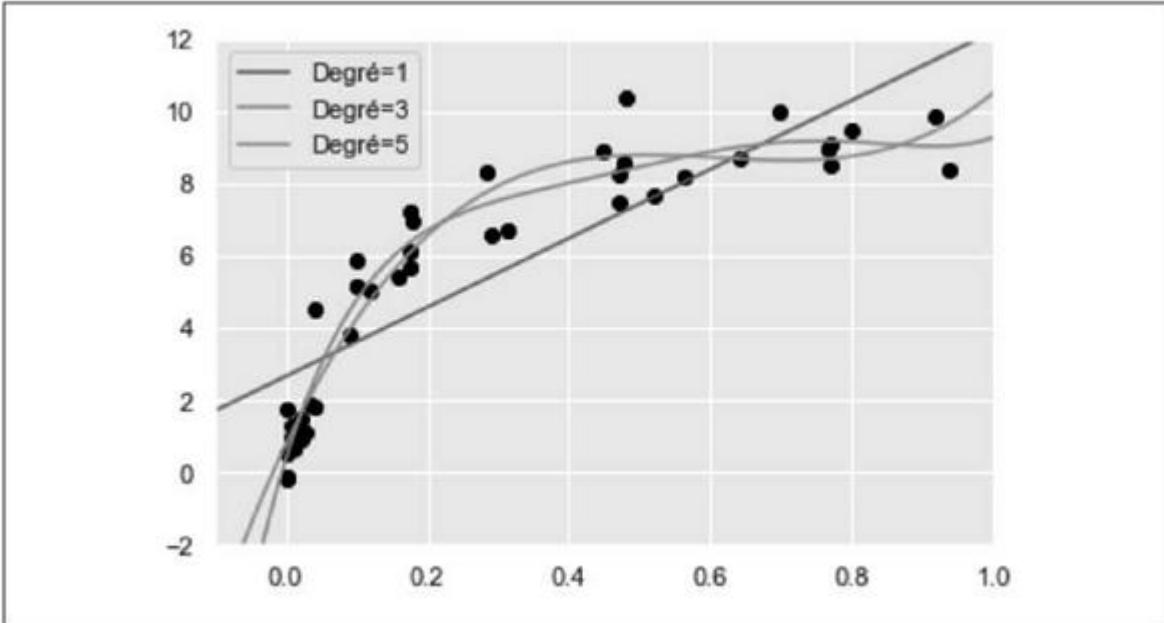
plt.scatter(X.ravel(), y, color='black')
axis = plt.axis()

for degree in [1, 3, 5]:
    y_test = PolynomialRegression(degree).fit(X,
y).predict(X_test)
    plt.plot(X_test.ravel(), y_test,
label='Degré={0}'.format(degree))

plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');

```

Le paramètre qui permet de jouer sur la complexité du modèle est le degré du polynôme, qui peut être toute valeur entière non négative. La question est donc simple : quel est le degré polynomial qui fournit le meilleur compromis entre biais fort (sous-ajustement) et variance forte (surajustement) ?



[Figure 5.27](#) : Trois modèles polynomiaux différents ajustés à un jeu de données.

Pour nous aider, nous pouvons visualiser la courbe de validation pour les données et le modèle concerné au moyen de la routine `validation_curve()` de Scikit-Learn. En lui fournissant un modèle, des données, un nom de paramètre et une plage d'exploration, nous pouvons demander à la fonction de calculer automatiquement le score d'entraînement et le score de validation pour toute une plage ([Figure 5.28](#)) :

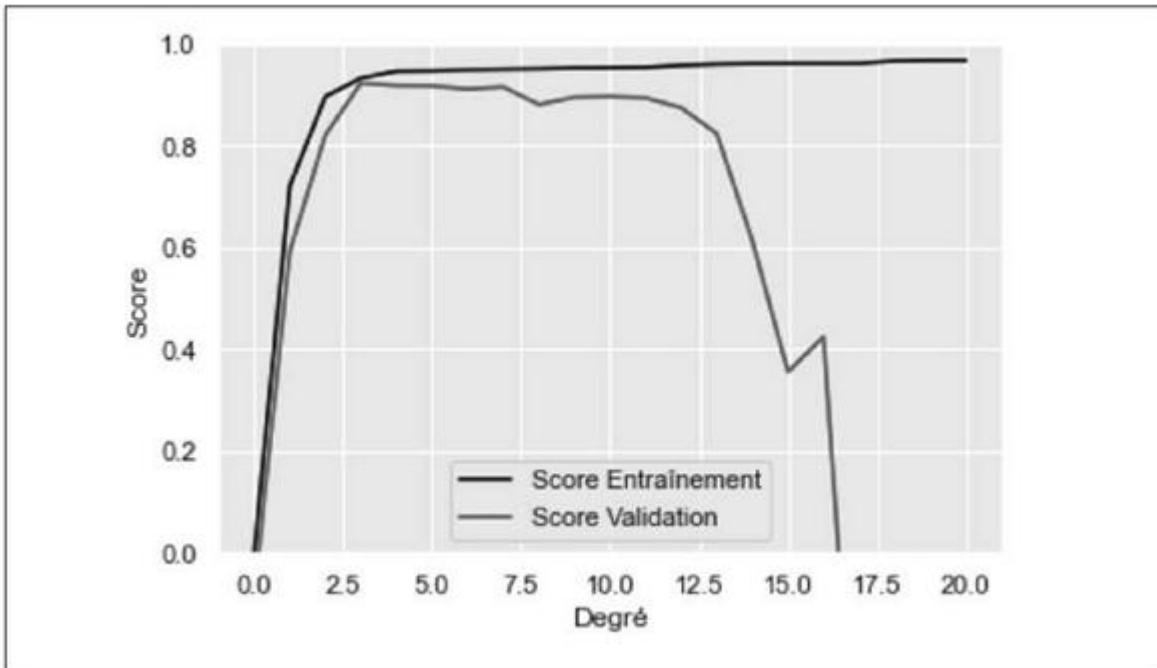
In[13]:

```
from sklearn.learning_curve import
validation_curve
degree = np.arange(0, 21)
train_score, val_score =
validation_curve(PolynomialRegression(), X, y,
```

```
'polynomialfeatures_degree',
degree,
cv=7)

plt.plot(degree, np.median(train_score, 1),
color='blue', label='Score Ent')
plt.plot(degree, np.median(val_score, 1),
color='red', label='Score Val')
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('Degré')
plt.ylabel('Score');
```

Le résultat correspond à nos attentes : le score d'entraînement est toujours supérieur au score de validation, et ce dernier atteint rapidement un maximum puis stagne avant de chuter, ce qui est le signe d'un surajustement sévère.

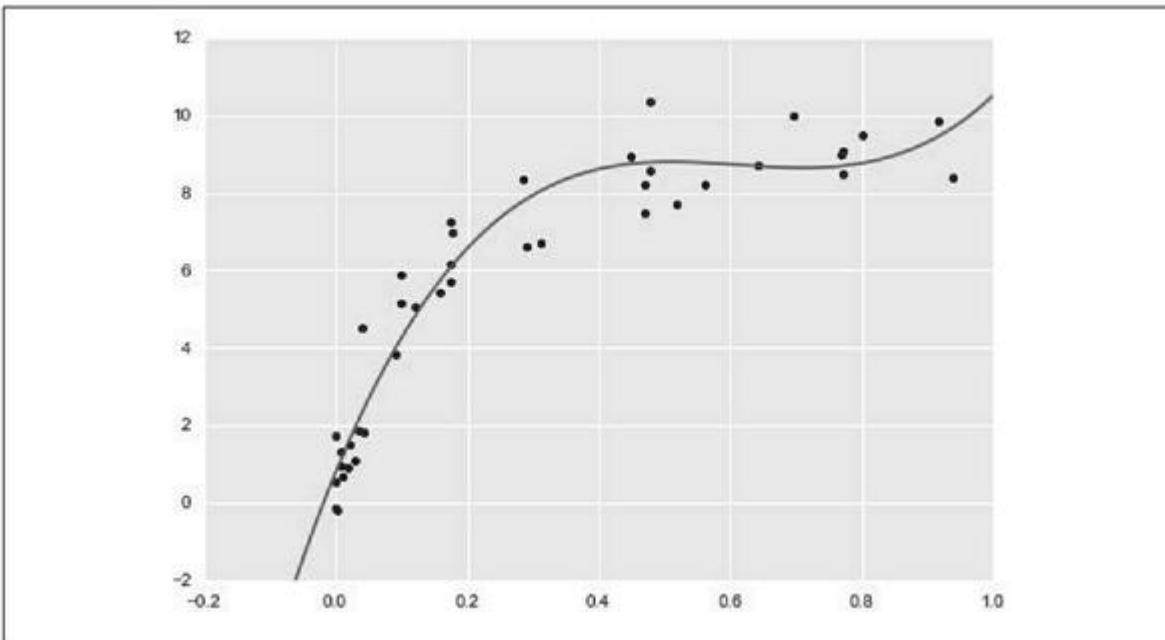


[Figure 5.28](#) : Courbes de validation pour les données de la Figure 5.27 (voir aussi Figure 5.26).

La lecture de la courbe nous apprend que le point d'équilibre entre biais et variance correspond à un polynôme de degré 3. Servons-nous de cette connaissance pour lancer un ajustement par rapport aux données d'origine ([Figure 5.29](#)) :

In[14]:

```
plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = PolynomialRegression(3).fit(X,
y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```



[Figure 5.29](#) : Le modèle de validation croisée optimisé pour les données de la Figure 5.27.

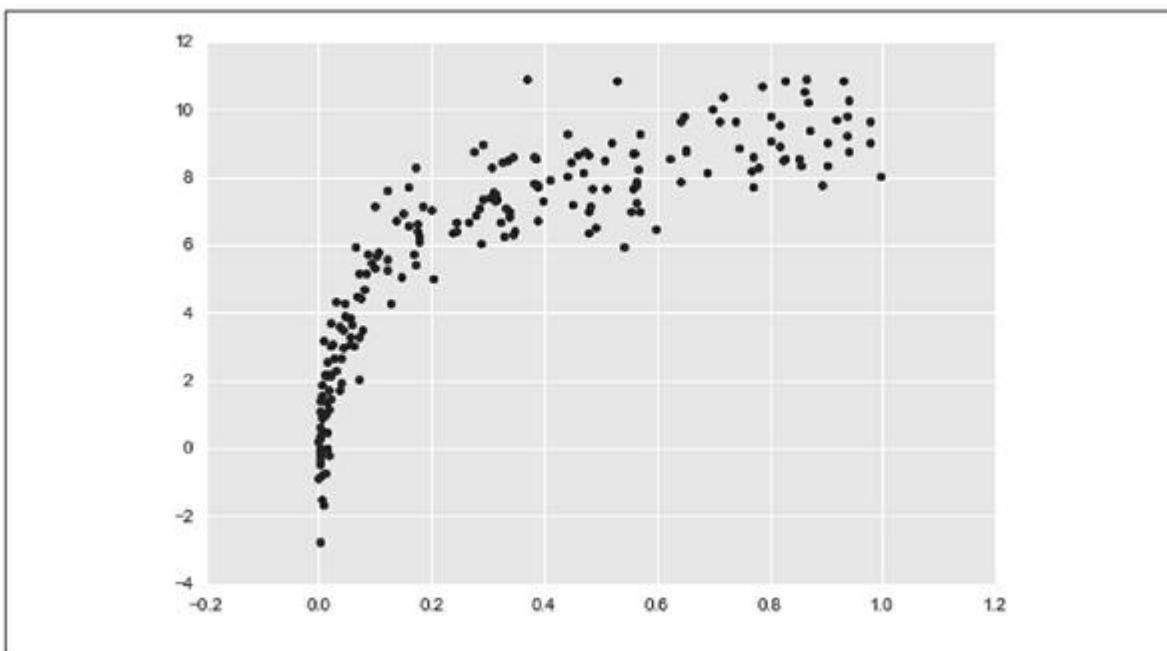
Vous remarquez que nous avons trouvé le modèle optimal sans avoir besoin de calculer le score d'entraînement. En examinant les relations entre score d'entraînement et score de validation, nous avons pu comprendre comment obtenir les meilleures performances avec ce modèle.

Courbes d'apprentissage

Pour bien estimer la complexité d'un modèle, il faut savoir que le modèle optimal diffère selon la taille des données d'entraînement. Générons par exemple un nouveau jeu de données comportant cinq fois plus de points ([Figure 5.30](#)) :

In[15]:

```
X2, y2 = make_data(200)  
plt.scatter(X2.ravel(), y2);
```



[Figure 5.30](#) : Jeu de données pour découvrir les courbes d'apprentissage.

Nous dupliquons le code précédent pour tracer la courbe de ce plus grand jeu de données. Nous en profitons pour afficher également les résultats précédents ([Figure 5.31](#)) :

In[16]:

```
degree = np.arange(21)  
train_score2, val_score2 =  
validation_curve(PolynomialRegression(), X2, y2,  
'polynomialfeatures_degree',
```

```
degree, cv=7)

plt.plot(degree, np.median(train_score2, 1),
color='blue',
      label='Score Ent.')
plt.plot(degree, np.median(val_score2, 1),
color='red',
      label='Score Val.')
plt.plot(degree, np.median(train_score, 1),
color='blue', alpha=0.3,
linestyle='dashed')
plt.plot(degree, np.median(val_score, 1),
color='red', alpha=0.3,
linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('Degré')
plt.ylabel('Score');
```

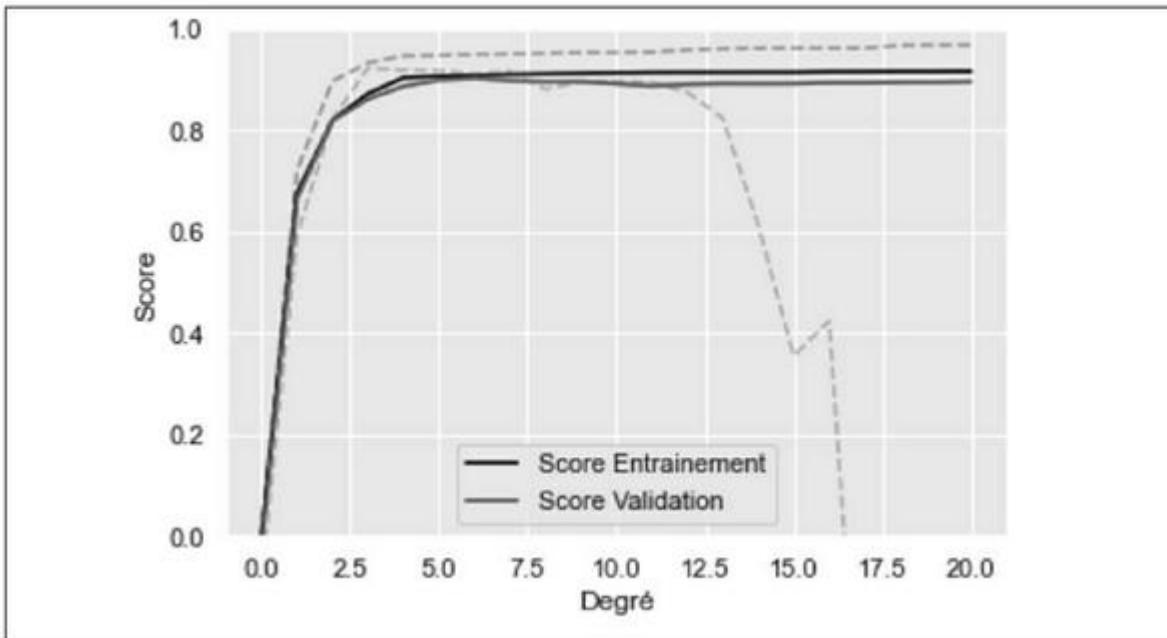


Figure 5.31 : Les courbes d'apprentissage pour l'ajustement du modèle polynomial avec les données de la Figure 5.30.

Les résultats pour le nouveau jeu de données sont tracés en lignes continues épaisses, les résultats antérieurs étant rappelés en lignes tiretées. On devine d'après la courbe de validation que le jeu de données plus volumineux permet un modèle bien plus complexe : le point haut correspond à peu près au degré 6, mais même un degré 20 ne provoque pas encore de surajustement sérieux. Les scores d'entraînement et de validation restent proches.

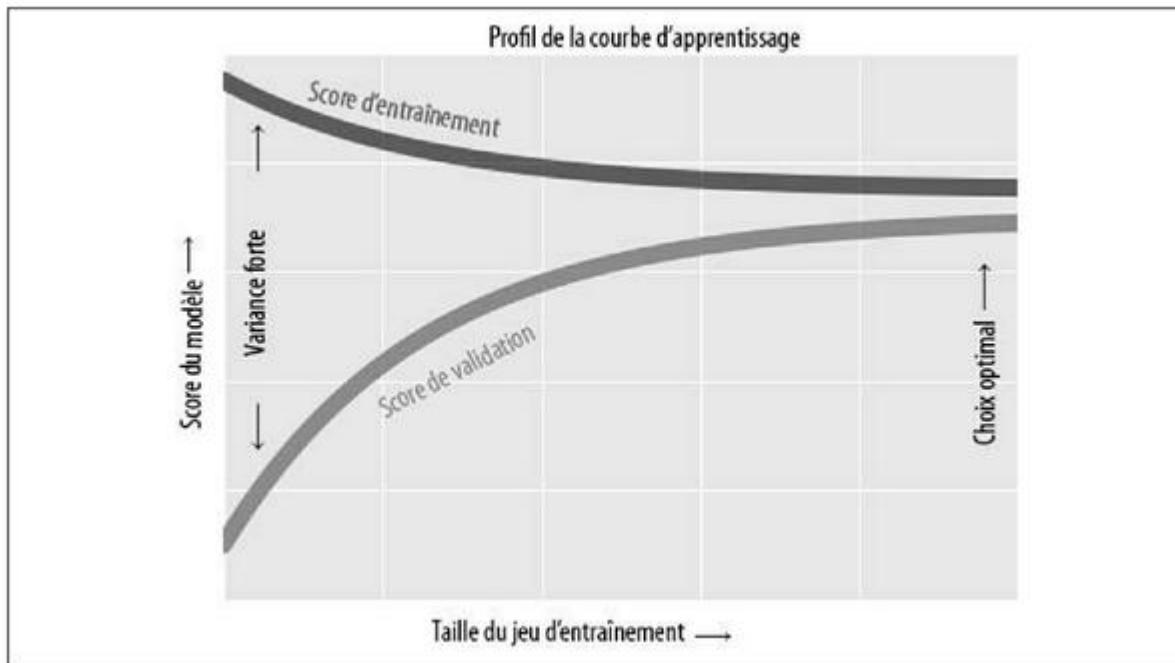
La courbe de validation dépend donc non pas d'un, mais de deux éléments d'entrée : la complexité du modèle et le nombre de points d'entraînement. Il est souvent utile d'explorer comment se comporte le modèle en fonction du

volume de points d'entraînement, ce qui se fait aisément en augmentant le sous-ensemble utilisé pour ajuster le modèle. Lorsque l'on trace la courbe en fonction des scores d'entraînement et de validation relativement à la taille du jeu de données d'entraînement, on trace une *courbe d'apprentissage*.

Voici les comportements que l'on devrait remarquer dans une courbe d'apprentissage :

- Pour une certaine complexité du modèle, celui-ci va surajuster un jeu de données peu volumineux. Le score d'entraînement sera assez élevé, mais le score de validation assez faible.
- Pour une certaine complexité du modèle, celui-ci va surajuster un jeu de données important : le score d'entraînement va diminuer alors que le score de validation va augmenter.
- Sauf pour une raison étrange, un modèle ne devrait jamais pouvoir donner un meilleur score de validation que son score d'entraînement ; les deux courbes doivent rester proches, mais ne pourront jamais se croiser.

Sous un aspect qualitatif, en tenant compte des remarques précédentes, une courbe d'apprentissage devrait se présenter comme dans la [Figure 5.32](#).



[Figure 5.32](#) : Schéma montrant l'interprétation typique d'une courbe d'apprentissage.

Ce schéma a ceci de remarquable que les courbes convergent au fur et à mesure de l'augmentation du nombre d'échantillons de l'entraînement. Une fois que vous avez atteint le point de convergence, il ne sert à rien d'ajouter des données d'entraînement ! Pour améliorer encore les performances du modèle, la seule solution qu'il reste est d'opter pour un autre modèle, en général plus complexe.

Les courbes d'apprentissage dans Scikit-Learn

Scikit-Learn propose un sous-module pour calculer les courbes d'apprentissage des modèles. Voyons comment

calculer celles de notre jeu de données de départ avec un modèle à polynôme de degré 2 ou 9 ([Figure 5.33](#)) :

```
In[17]:  
from sklearn.learning_curve import  
learning_curve  
  
fig, ax = plt.subplots(1, 2, figsize=(16, 6))  
fig.subplots_adjust(left=0.0625, right=0.95,  
wspace=0.1)  
  
for i, degree in enumerate([2, 9]):  
    N, train_lc, val_lc =  
    learning_curve(PolynomialRegression(degree),  
                   X, y,  
                   cv=7,  
  
                   train_sizes=np.linspace(0.3, 1, 25))  
    ax[i].plot(N, np.mean(train_lc, 1),  
               color='blue', label='Score Ent.')  
    ax[i].plot(N, np.mean(val_lc, 1), color='red',  
               label='Score Val.')  
    ax[i].hlines(np.mean([train_lc[-1],  
                         val_lc[-1]]), N[0], N[-1], color='gray',  
                linestyle='dashed')  
    ax[i].set_ylim(0, 1)  
    ax[i].set_xlim(N[0], N[-1])  
    ax[i].set_xlabel('Taille Entraînement')  
    ax[i].set_ylabel('Score')  
    ax[i].set_title('Degré = {}'.format(degree)),
```

```

size=14)
ax[i].legend(loc='best')

```

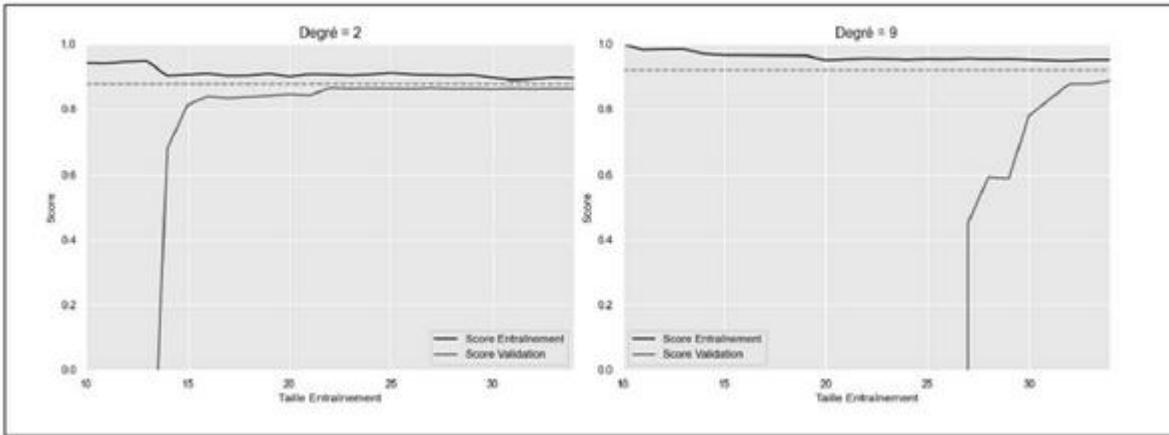


Figure 5.33 : Courbes d'apprentissage pour modèle peu (à gauche) et très complexe (à droite).

C'est un diagnostic fructueux que nous pouvons faire : nous pouvons vérifier visuellement comment le modèle répond à l'augmentation du volume de données d'entraînement. À l'endroit où les deux courbes ont convergé, c'est-à-dire qui sont déjà proches, il ne sert plus à rien d'ajouter des données d'entraînement ! Cela est clairement visible dans le panneau de gauche qui montre la courbe pour le modèle de degré 2.

Pour augmenter encore le score de convergence, il faut adopter un autre modèle, comme dans le panneau de droite. Nous augmentons ainsi le score de convergence, ce que montre la courbe en pointillé, mais au prix d'une variance plus élevée (ce que montre l'écart entre les deux scores). La

courbe d'apprentissage du modèle plus complexe pourrait finir par converger si nous ajoutions encore des points de données.

La production de la courbe d'apprentissage du modèle et du jeu de données concerné peut donc vous aider à vous orienter pour améliorer encore votre tactique d'analyse.

La validation en pratique : recherche en grille

Nous venons de voir l'importance de la recherche du meilleur compromis entre biais et variance, et de la dépendance à la complexité du modèle et à la taille du jeu d'entraînement. Dans la pratique, vous disposez en général d'autres paramètres pour régler votre modèle. Les tracés des courbes de validation et d'apprentissage peuvent passer du format d'une ligne au format d'une surface à plusieurs dimensions, mais dans ce cas, la visualisation devient difficile. Il serait intéressant de trouver directement le modèle qui procure le score de validation le plus élevé.

Scikit-Learn fournit justement un outil automatisé dans son module nommé `grid_search`. Voyons comment utiliser cette recherche pour trouver le modèle polynomial optimal. Nous allons partir d'une grille de caractéristiques de modèle à trois dimensions, le degré polynomial, un indicateur

déterminant s'il faut ajuster pour l'interception et un autre indicateur décidant s'il faut normaliser le problème. Nous mettons ces éléments en place au moyen du méta-estimateur de Scikit-Learn nommé GridSearchCV :

In[18]:

```
from sklearn.grid_search import GridSearchCV

param_grid = {'polynomialfeatures_degree':
np.arange(21),
              'linearregression_fit_intercept':
[True, False],
              'linearregression_normalize':
[True, False]}

grid = GridSearchCV(PolynomialRegression(),
param_grid, cv=7)
```

Comme dans le cas d'un estimateur classique, rien n'a encore été appliqué aux données. Nous allons faire ajuster le modèle en chaque point de la grille par appel à la méthode fit() tout en gardant une trace des scores pendant que nous progressons :

In[19]: `grid.fit(X, y);`

Une fois l'ajustement réalisé, nous demandons de connaître les meilleurs paramètres ainsi :

```
In[20]: grid.best_params_
```

```
Out[20]:
```

```
{'linearregression__fit_intercept': False,  
 'linearregression__normalize': True,  
 'polynomialfeatures__degree': 4}
```

Rien ne nous empêche enfin d'utiliser le meilleur modèle pour afficher l'ajustement aux données en réutilisant le code déjà vu ([Figure 5.34](#)) :

```
In[21]:
```

```
model = grid.best_estimator_
```

```
plt.scatter(X.ravel(), y)  
lim = plt.axis()  
y_test = model.fit(X, y).predict(X_test)  
plt.plot(X_test.ravel(), y_test, hold=True);  
plt.axis(lim);
```

La technique de recherche en grille offre bien d'autres options. Vous pouvez notamment décider d'une fonction de notation personnalisée, lancer les calculs en parallèle, effectuer des recherches aléatoires, etc. Nous en reparlons dans ce chapitre dans la section dédiée à l'algorithme d'estimation KDE ainsi que dans celle qui présente une application de détection de visages. Vous pouvez également voir la documentation de Scikit-Learn.

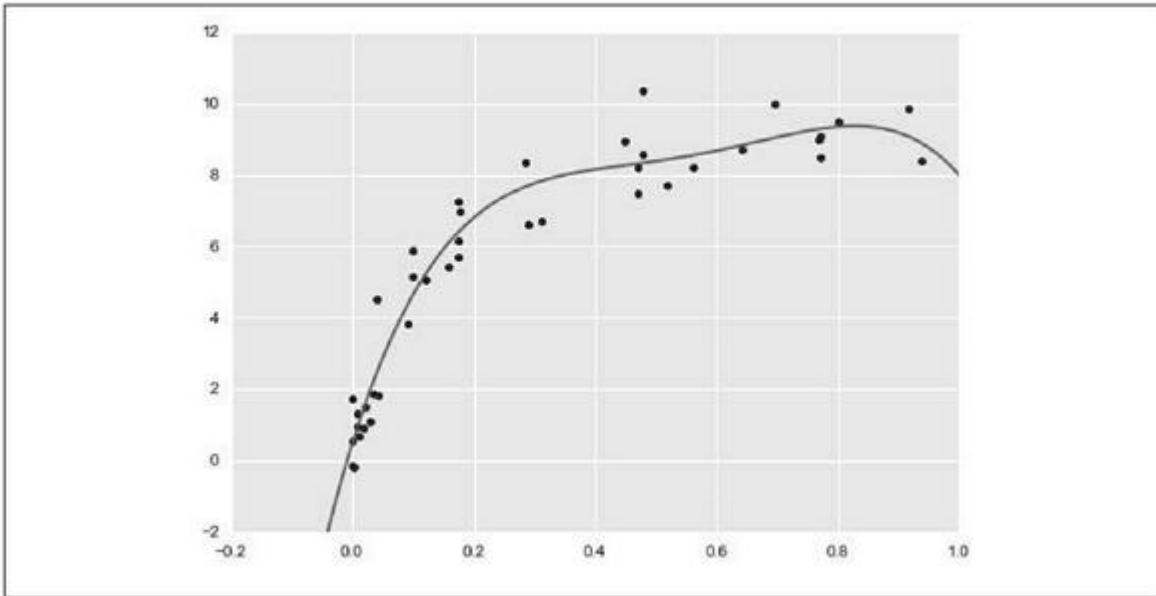


Figure 5.34 : Le modèle le mieux ajusté tel que déterminé par une recherche en grille automatique.

Résumé

Nous venons de lever le voile sur le concept de validation de modèle et d'optimisation des hyperparamètres. Nous nous sommes intéressés à l'aspect intuitif du compromis entre biais et variance, sans oublier son impact sur l'ajustement du modèle. Nous avons notamment remarqué qu'il est essentiel d'utiliser un jeu de validation ou une approche de validation croisée pour ajuster les paramètres afin d'éviter le surajustement des modèles plus complexes ou plus souples.

Nous verrons dans la suite du chapitre plus en détail certains modèles en montrant ce qu'il est possible d'ajuster en terme de paramètres libres et verrons comment ces

paramètres affectent la complexité du modèle à utiliser. Souvenez-vous de ce que vous avez découvert dans cette section tout en lisant la suite du chapitre et en découvrant les différents algorithmes d'apprentissage qui vont être passés en revue.

5.4 : Ingénierie des caractéristiques

Dans les sections précédentes, nous avons présenté les idées fondamentales de l'apprentissage machine, mais tous nos exemples supposaient que nous utilisions des données numériques dans un format simplifié, [n_samples, n_features]. Dans le monde réel, les données se présentent rarement sous cet aspect. Pour pouvoir bien exploiter l'apprentissage machine, une étape importante consiste à reconditionner les données d'entrée, c'est-à-dire à reformuler les caractéristiques, ce qui se nomme l'ingénierie des caractéristiques. Vous devez tirer profit de toutes les informations dont vous disposez au sujet de votre lot de données, et cherchez à en produire des valeurs numériques qui vont permettre de construire votre matrice de caractéristiques.

Nous allons découvrir dans cette section quelques exemples d'ingénierie de caractéristiques simples : nous verrons comment représenter des *catégories*, du texte et des images. Nous verrons enfin les *caractéristiques dérivées* qui permettent d'augmenter la complexité du modèle et verrons comment imputer des données manquantes. Ce processus est souvent

appelé vectorisation, car il s'agit de convertir des données arbitraires en vecteurs correctement conformés.

Caractéristiques catégorielles

Un type de données non numériques très répandu correspond aux données catégorielles. Une simple liste de maisons à vendre contiendra le « prix demandé » et le « nombre de pièces » sous forme numérique, mais également une indication du « quartier », sous forme de texte. Voici un exemple :

```
In[1]:  
data = [  
    {'prix': 850000, 'pièces': 4, 'quartier':  
     'Tour Eiffel'},  
    {'prix': 700000, 'pièces': 3, 'quartier':  
     'Panthéon'},  
    {'prix': 650000, 'pièces': 3, 'quartier':  
     'Vincennes'},  
    {'prix': 600000, 'pièces': 2, 'quartier':  
     'Panthéon'}  
]
```

Pour pouvoir traiter la caractéristique correspondant au quartier, vous pourriez être tenté de donner une valeur numérique à chacun des quartiers :

```
In[2]: {'Tour Eiffel': 1, 'Panthéon': 2,  
'Vincennes': 3};
```

L'expérience montre que l'approche n'est pas conseillée avec Scikit-Learn, car cette librairie suppose que les caractéristiques numériques incarnent bien des quantités algébriques. Dans l'exemple, on devrait pouvoir écrire Tour Eiffel < Panthéon < Vincennes ou même Vincennes - Tour Eiffel = Panthéon, ce qui est clairement risible, toute considération de lutte des classes mise à part.

Une technique plus efficace consiste à effectuer un encodage binaire appelé *one-hot*. Cela consiste à créer autant de nouvelles colonnes de caractéristiques que de valeurs différentes dans la catégorie, avec la valeur 1 ou 0. À partir du moment où les données d'entrée se présentent sous forme d'une liste de dictionnaires, vous pouvez profiter de la méthode DictVectorizer() de Scikit-Learn :

```
In[3]:
```

```
from sklearn.feature_extraction import  
DictVectorizer  
vec = DictVectorizer(sparse=False, dtype=int)  
vec.fit_transform(data)
```

```
Out[3]:
```

```
array([[ 4, 850000,      0,      1,      0],  
       [ 3, 700000,      1,      0,      0],
```

```
[ 3, 650000, 0, 0, 1],  
[ 2, 600000, 1, 0, 0]])
```

Vous constatez que l'ancienne colonne de quartier a été remplacée par trois colonnes, dans lesquelles un 1 indique l'appartenance au quartier considéré. Une fois ce recodage catégoriel réalisé, vous pouvez repartir dans un ajustement du modèle habituel.

Vous pouvez demander d'afficher les noms des caractéristiques pour savoir à quoi correspond chaque colonne :

In[4]:

```
vec.get_feature_names()
```

Out[4]:

```
['pièces',  
'prix',  
'quartier=Panthéon',  
'quartier=Tour Eiffel',  
'quartier=Vincennes']
```

L'inconvénient de cette technique apparaît lorsque le nombre de catégories devient énorme. La taille du jeu de données grossit en conséquence. Cependant, les nouvelles données contiennent essentiellement des zéros, et une bonne solution consiste à demander à produire une *matrice creuse* ou *éparse* :

In[5]:

```
vec = DictVectorizer(sparse=True, dtype=int)
vec.fit_transform(data)
```

Out[5]:

```
<4x5 sparse matrix of type '<class
'numpy.int64>''
with 12 stored elements in Compressed Sparse Row
format>
```

Quasiment tous les processus d'estimation de Scikit-Learn savent traiter des entrées éparses pour réaliser l'ajustement et l'évaluation. Scikit-Learn offre deux outils complémentaires capables de traiter cet encodage, `sklearn.preprocessing.OneHotEncoder` et `sklearn.feature_extraction.FeatureHasher`.

Caractéristiques de type texte

Un autre besoin fréquent dans le domaine de la préparation des données d'entrée est la conversion de données de type texte en valeurs numériques représentatives. La plupart des outils d'exploitation des données accessibles sur les réseaux sociaux utilisent une forme ou une autre de codage du texte. Une des méthodes les plus simples consiste à compter les mots : vous analysez chaque bloc de texte, vous comptez les

occurrences de chaque mot et vous stockez les résultats dans un tableau.

Considérons les trois segments de phrase suivants :

In[6] :

```
echanti = ['problème du diable',
            'diabe rose',
            'problème zénital']
```

Nous pouvons vectoriser ces données en comptant les mots et construire une colonne pour le mot problème, une autre pour le mot diable, une autre pour le mot horizon, etc. Faire cela à la main devient vite fastidieux, mais nous pouvons tirer profit de la méthode CountVectorizer de Scikit-Learn :

In[7] :

```
from sklearn.feature_extraction.text import
CountVectorizer
vec = CountVectorizer()
X = vec.fit_transform(echanti)
X
```

Out[7] :

```
<3x5 sparse matrix of type '<class
'numpy.int64'>'
      with 7 stored elements in Compressed Sparse
Row format>
```

Nous obtenons une matrice éparse qui contient le nombre d'apparitions de chacun des mots. Nous pouvons mieux la visualiser en la convertissant en un objet DataFrame avec des colonnes légendées :

In[8] :

```
import pandas as pd
pd.DataFrame(X.toarray(),
columns=vec.get_feature_names())
```

Out[8] :

	diable	du	problème	rose	zénital
0	1	1	1	0	0
1	1	0	0	1	0
2	0	0	1	0	1

Cette approche n'est pas sans défaut car le nombre de mots va donner plus de poids à ceux qui apparaissent le plus souvent, ce qui est loin d'être désiré dans certains algorithmes de classification. Une solution à ce problème est la technique *TF-IDF* qui cherche une proportion entre la fréquence d'un terme et l'inverse de sa fréquence dans les documents. Le poids relatif de chaque mot tient compte de son apparition globalement. La syntaxe permettant d'utiliser cette technique ressemble à celle de l'exemple précédent :

In[9] :

```
from sklearn.feature_extraction.text import
```

TfidfVectorizer

```
vec = TfidfVectorizer()  
X = vec.fit_transform(échanti)  
pd.DataFrame(X.toarray(),  
columns=vec.get_feature_names())
```

Out[9]:

```
        diable      du   problème     rose    zénital  
0  0.517856  0.680919  0.517856  0.000000  
0.000000  
1  0.605349  0.000000  0.000000  0.795961  
0.000000  
2  0.000000  0.000000  0.605349  0.000000  
0.795961
```

Nous verrons un exemple de classification utilisant TF-IDF lorsque nous aborderons la classification bayésienne naïve dans ce même chapitre.

Caractéristiques graphiques d'image

On a fréquemment besoin de préparer un encodage des images pour pouvoir faire un apprentissage machine. L'approche la plus simple dans ce domaine est celle que nous avons utilisée lorsque nous avons cherché à reconnaître des chiffres manuscrits en milieu de chapitre.

Cette approche simple n'est cependant pas satisfaisante dans un certain nombre d'applications.

Une description détaillée des techniques d'extraction de caractéristiques dans des images dépasse le cadre de ce livre. Vous pourrez en découvrir d'excellentes implémentations dans le projet nommé Scikit-Image (<http://scikit-image.org/>). Voyez également la dernière section de ce chapitre qui décrit une application de détection de visages.

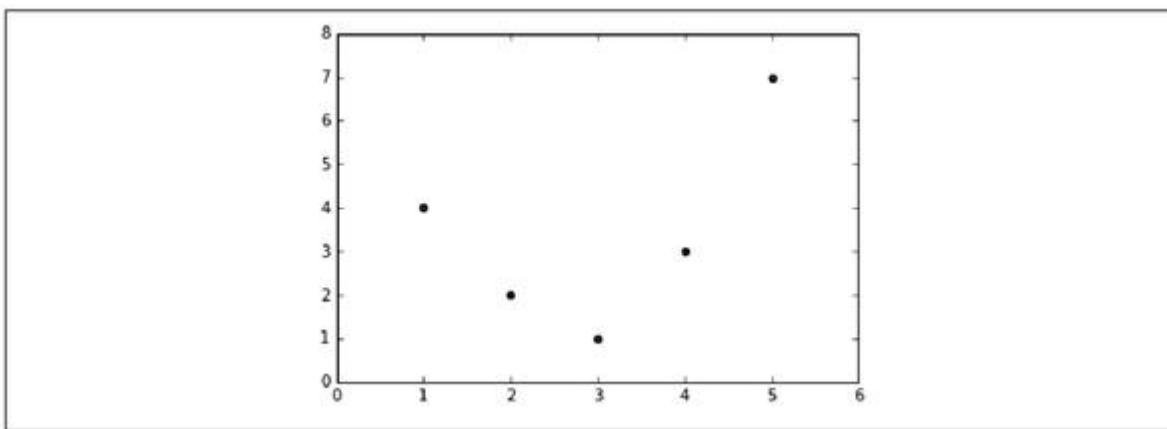
Caractéristiques dérivées

On peut enfin produire des caractéristiques par une opération mathématique à partir des données d'entrée. Vous vous souvenez de la section sur la validation du modèle et des hyperparamètres ; nous y avions construit des caractéristiques polynomiales. Nous avions transformé une régression linéaire en régression polynomiale non pas en changeant le modèle, mais en transformant les données d'entrée ! On parle parfois de régression de fonctions de base. Nous y reviendrons en détail dans la section sur la régression linéaire qui suit.

Commençons par un jeu de données qui ne peut clairement pas être décrit correctement par un segment de droite ([Figure 5.35](#)) :

```
In[10]:
```

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
x = np.array([1, 2, 3, 4, 5])
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
```



[Figure 5.35](#) : Données d'entrée ne convenant pas à une ligne droite.

Nous pouvons essayer d'ajuster une ligne droite optimisée au moyen de `LinearRegression()` ([Figure 5.36](#)) :

```
In[11]:
```

```
from sklearn.linear_model import
LinearRegression
X = x[:, np.newaxis]
model = LinearRegression().fit(X, y)
yfit = model.predict(X)
plt.scatter(x, y)
plt.plot(x, yfit);
```

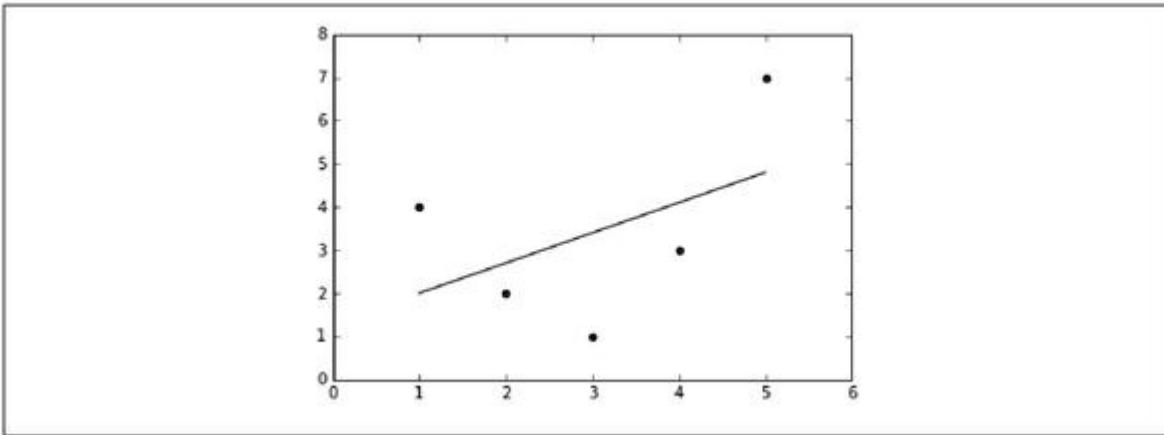


Figure 5.36 : Ajustement par ligne droite peu satisfaisant.

Il nous faut un modèle plus complexe pour visualiser les relations entre x et y. Nous pouvons créer de nouvelles colonnes de caractéristiques pour donner plus de souplesse au modèle. Voici par exemple comment générer des caractéristiques polynomiales :

In[12]:

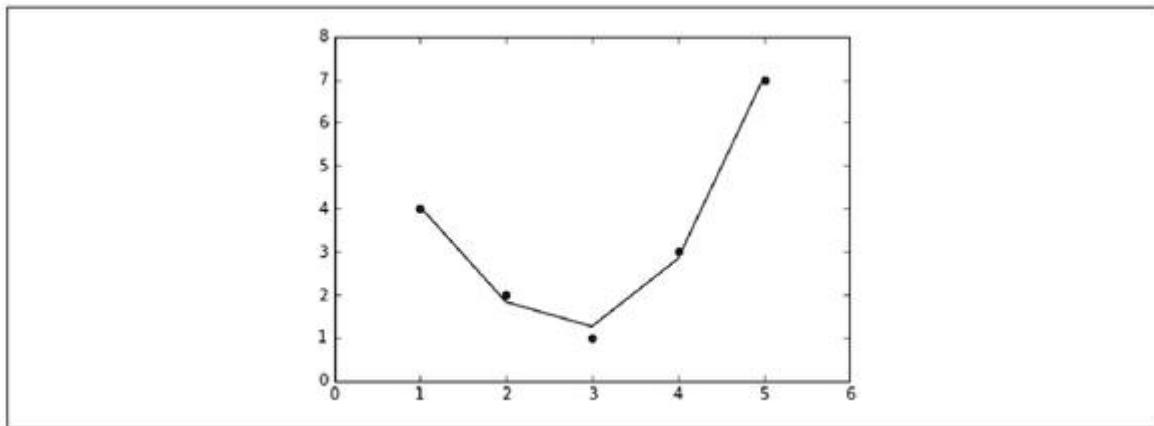
```
from sklearn.preprocessing import  
PolynomialFeatures  
poly = PolynomialFeatures(degree=3,  
include_bias=False)  
X2 = poly.fit_transform(X)  
print(X2)
```

```
[[ 1.  1.  1.]  
 [ 2.  4.  8.]  
 [ 3.  9.  27.]  
 [ 4. 16.  64.]  
 [ 5. 25. 125.]]
```

La matrice de caractéristiques produite contient une colonne pour x , une deuxième colonne pour x^2 et une troisième colonne pour x^3 . Si nous demandons une régression linéaire sur ces nouvelles entrées, nous obtenons un résultat mieux ajusté aux données réelles ([Figure 5.37](#)) :

In[13]:

```
model = LinearRegression().fit(X2, y)
yfit = model.predict(X2)
plt.scatter(x, y)
plt.plot(x, yfit);
```



[Figure 5.37](#) : Ajustement linéaire pour caractéristiques polynomiales dérivées des données.

L'idée consistant à améliorer les résultats non en changeant le modèle, mais en transformant les données d'entrée est une idée fondamentale dans la plupart des méthodes d'apprentissage machine sophistiqué. Nous y revenons dans la section suivante qui parle de régression linéaire. C'est

également une ligne directrice qui a guidé l'apparition des techniques relatives au noyau (kernel) que nous découvrirons dans la section sur les machines à vecteurs de support SVM.

Imputation des données manquantes

L'ingénierie des caractéristiques doit également résoudre le problème des données manquantes. Nous avons déjà vu dans le [Chapitre 2](#) que l'on utilisait souvent la pseudo-valeur NaN pour les valeurs absentes. Partons du jeu de données suivant :

```
In[14]:  
from numpy import nan  
X = np.array([[ nan,  0,   3 ],  
              [ 3,    7,   9 ],  
              [ 3,    5,   2 ],  
              [ 4,    nan,  6 ],  
              [ 8,    8,   1 ]])  
y = np.array([14, 16, -1,  8, -5])
```

Pour réaliser un apprentissage machine sur ces données, il faut d'abord remplacer les valeurs manquantes par des valeurs de remplissage, ce qui correspond à l'opération *d'imputation*. On peut opter pour une solution simpliste

consistant par exemple à remplacer toutes les valeurs par la moyenne de la colonne concernée ou une solution plus complexe, en s'appuyant sur une technique de remplissage de matrice. On peut aussi adopter un modèle mieux capable de gérer ces données manquantes.

Les approches les plus complexes sont en général très dépendantes du domaine d'application, et nous n'en décrirons pas les détails ici. Pour une approche d'imputation simple, Scikit-Learn propose la classe SimpleImputer qui permet de faire générer la moyenne, la médiane ou une autre valeur habituelle :

In[15]:

```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan,
strategy='mean')
X2 = imp.fit_transform(X)
X2
```

Out[15]:

```
array([[4.5, 0. , 3. ],
       [3. , 7. , 9. ],
       [3. , 5. , 2. ],
       [4. , 5. , 6. ],
       [8. , 8. , 1. ]])
```

Les données résultantes comportent à la place des valeurs manquantes la moyenne des autres valeurs de la colonne.

Nous pouvons directement injecter ces données dans un estimateur de type `LinearRegression()` :

In[16]:

```
model = LinearRegression().fit(X2, y)
model.predict(X2)
```

Out[16]:

```
array([ 13.14869292, 14.3784627 , -1.15539732,
       10.96606197, -5.33782027])
```

Enchaînement de traitements avec un pipeline

Tous les exemples précédents deviennent fastidieux à utiliser de façon répétée, car cela oblige à reformuler les mêmes instructions. Nous aimerais enchaîner automatiquement trois opérations successives dans une sorte de pipeline de traitement :

1. Imputation des valeurs manquantes en adoptant la moyenne.
2. Transformation de caractéristiques par la méthode quadratique.
3. Ajustement du modèle par régression linéaire.

Scikit-Learn propose un objet pipeline qui permet de sérier ces trois traitements :

```
In[17]:  
from sklearn.pipeline import make_pipeline  
  
model =  
make_pipeline(SimpleImputer(strategy='mean'),  
  
PolynomialFeatures(degree=2),  
              LinearRegression())
```

Le pipeline se comporte comme un objet Scikit-Learn standard ; il va appliquer les opérations spécifiées aux données d'entrée qui lui sont fournies :

```
In[18]:  
model.fit(X, y)          # X avec manquants  
comme au-dessus  
print(y)  
print(model.predict(X))  
  
[14 16 -1 8 -5]  
[ 14. 16. -1. 8. -5.]
```

Pour simplifier l'exposé, nous avons appliqué ici le modèle aux données d'entraînement, ce qui explique sa prédiction parfaite des résultats (revoyez si nécessaire la section sur la validation de modèle).

Les deux prochaines sections présentent des exemples de pipeline pour les régressions linéaires et pour les machines à vecteurs de support.

5.5 : Classification bayésienne naïve

Dans les sections précédentes du chapitre, nous avons fait le tour des concepts fondamentaux de l'apprentissage machine. Dans toutes les sections suivantes, nous allons entrer dans les détails des algorithmes d'apprentissage supervisé et non supervisé, en commençant par la classification bayésienne naïve.

Les modèles regroupés dans cette famille se distinguent par d'excellentes performances et une grande simplicité, ce qui les rend tout à fait aptes à traiter des jeux de données comportant un grand nombre de dimensions. Leur rapidité et le faible nombre de paramètres ajustables en font des modèles de choix pour produire une base de référence rapide d'un problème de classification. Découvrons une approche intuitive du fonctionnement des classificateurs bayésiens naïfs. Nous verrons ensuite quelques exemples d'application.

La classification bayésienne

Les classificateurs bayésiens naïfs se fondent sur des méthodes de classification bayésienne, elles-mêmes fondées sur le théorème de Bayes. Il s'agit d'une formule qui décrit les relations entre probabilité conditionnelle de quantités et

statistiques. Dans cette classification, nous cherchons à connaître la probabilité de vraisemblance d'un label à partir de caractéristiques observées. La formule peut s'écrire $P(L | \text{caractéristiques})$. Le théorème de Bayes nous indique comment poser l'équation sous la forme de quantités que nous pouvons calculer directement :

$$P(L | \text{caractéristiques}) = \frac{P(\text{caractéristiques} | L)P(L)}{P(\text{caractéristiques})}$$

Lorsque nous devons choisir entre deux labels nommés L_1 et L_2 , une solution consiste à faire calculer la proportion des probabilités *a posteriori* de chaque label :

$$\frac{P(L_1 | \text{caractéristiques})}{P(L_2 | \text{caractéristiques})} = \frac{P(\text{caractéristiques} | L_1)P(L_1)}{P(\text{caractéristiques} | L_2)P(L_2)}$$

Il nous suffit de trouver un modèle permettant de calculer $P(\text{caractéristiques} | L_i)$ pour chaque label. Il s'agit d'un *modèle génératif*, car il spécifie le processus aléatoire hypothétique permettant de générer les données. La séquence d'entraînement d'un classifieur bayésien consiste à spécifier ce modèle génératif pour chacun des labels. La version intégrale d'un tel entraînement suppose un travail lourd, mais nous pouvons adopter des hypothèses de simplification à propos de la forme du modèle.

C'est à ce point que le terme « naïf » justifie sa présence : nous pouvons trouver une approximation satisfaisante du

modèle génératif pour chaque classe en émettant des hypothèses très naïves à propos de chaque label. (N.d.T. : nous considérons naïvement que les caractéristiques ne sont pas liées les unes aux autres, même si c'est peut-être et souvent le cas.) Les différents styles de classifieurs bayésiens naïfs travaillent avec un jeu d'hypothèses naïves différentes à propos des données. Nous en découvrirons quelques-uns. Commençons par nos opérations traditionnelles d'import :

```
In[1]:  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()
```

Bayésien naïf gaussien

Le classifieur naïf le plus facile à comprendre est le gaussien. Par hypothèse, il considère que les données de chaque label proviennent d'une distribution gaussienne simple. Partons du jeu de données suivant ([Figure 5.38](#)) :

```
In[2]:  
from sklearn.datasets import make_blobs  
X, y = make_blobs(100, 2, centers=2,  
random_state=2, cluster_std=1.5)
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,  
cmap='RdBu');
```

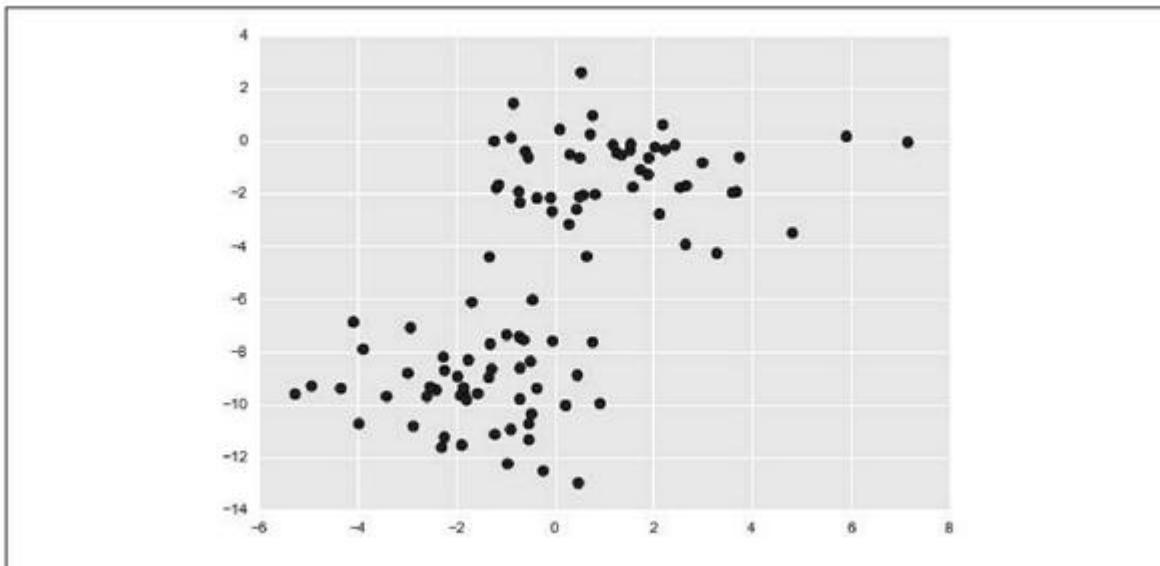
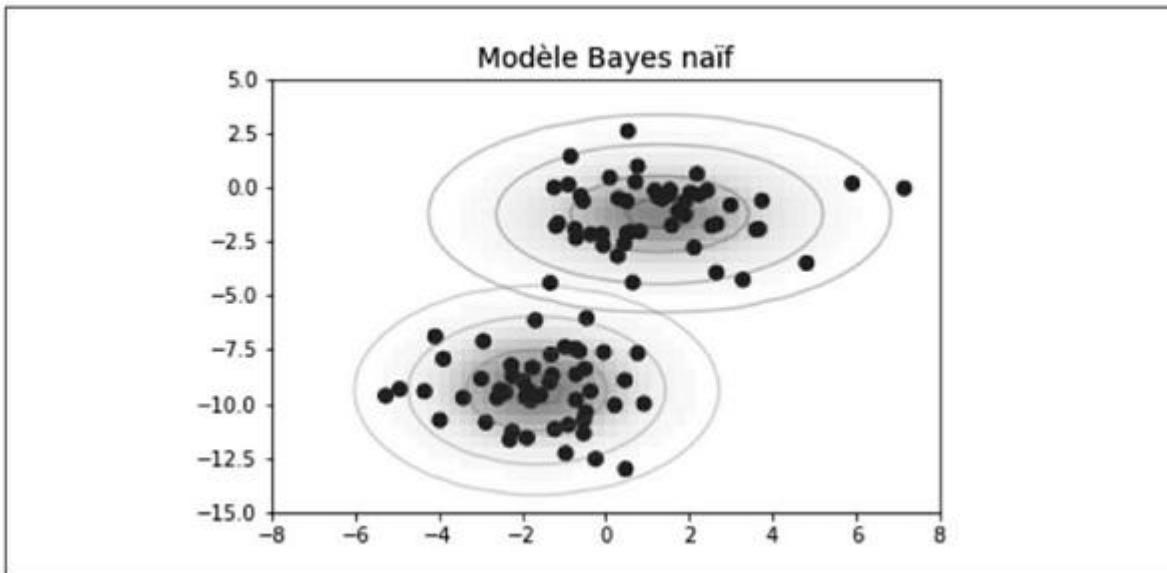


Figure 5.38 : Données d'entrée d'une classification bayésienne naïve gaussienne.

Nous pouvons créer un modèle simple très rapidement si nous supposons que les données correspondent à une distribution gaussienne sans aucune covariance entre les dimensions. Pour ajuster le modèle, nous cherchons la moyenne et l'écart-type des points de chaque label, ce qui suffit à définir la distribution. La [Figure 5.39](#) montre le résultat de cette hypothèse gaussienne naïve.



[Figure 5.39](#) : Visualisation d'un modèle bayésien naïf gaussien.

Les ellipses apparues correspondent au modèle génératif gaussien de chaque label. La probabilité augmente en allant vers le centre des ellipses. Une fois ce modèle génératif en place pour chacune des classes, nous pouvons facilement calculer la probabilité $P(\text{carac} \mid L_1)$ de chaque point de données puis obtenir la proportion *a posteriori* afin de savoir quel est le label le plus probable pour chaque point.

La procédure est implémentée dans l'estimateur de Scikit-Learn nommé `sklearn.naive_bayes`.

GaussianNB :

In[3] :

```
from sklearn.naive_bayes import GaussianNB
```

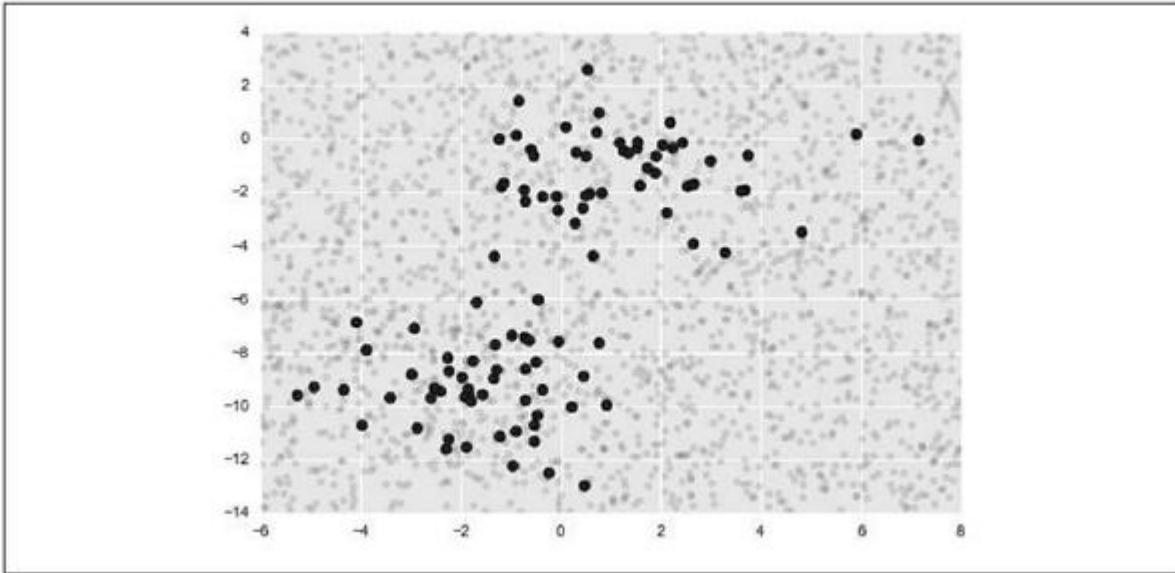
```
model = GaussianNB()  
model.fit(X, y);
```

Générons de nouvelles données aléatoires pour prédire les labels :

```
In[4]:  
rng = np.random.RandomState(0)  
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)  
ynew = model.predict(Xnew)
```

Traçons les données pour avoir une idée de la position des frontières de décision ([Figure 5.40](#)) :

```
In[5]:  
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,  
cmap='RdBu')  
lim = plt.axis()  
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew,  
s=20, cmap='RdBu', alpha=0.1)  
plt.axis(lim);
```



[Figure 5.40](#) : Visualisation d'une classification bayésienne naïve gaussienne.

Notez que dans ces modèles, les frontières sont en général quadratiques.

Cette approche bayésienne permet naturellement de faire des classifications probabilistes, que l'on peut calculer au moyen de la méthode `predict_proba()` :

In[6] :

```
yprob = model.predict_proba(Xnew)  
yprob[-8:].round(2)
```

Out[6] :

```
array([[ 0.89,  0.11],  
       [ 1. ,  0. ],  
       [ 1. ,  0. ],  
       [ 1. ,  0. ],  
       [ 1. ,  0. ],  
       [ 1. ,  0. ]])
```

```
[ 1. , 0. ],
[ 0. , 1. ],
[ 0.15, 0.85]])
```

Les colonnes contiennent les probabilités *a posteriori* pour le premier et le second label respectivement. Une approche bayésienne telle que celle-ci peut s'avérer très utile lorsque vous cherchez à estimer l'incertitude de votre classification.

En fin de compte, la classification ne pourra pas être meilleure que les hypothèses choisies, et c'est pourquoi le bayésien naïf gaussien ne produit généralement pas de très bons résultats. En revanche, et notamment lorsque le nombre de caractéristiques devient vraiment important, l'intérêt de ces classifications gaussiennes reste entier.

Bayésien naïf multinomial

L'approche gaussienne que nous venons de voir n'est pas la seule pouvant être utilisée pour spécifier la distribution génératrice des labels. Voyons l'approche bayésienne naïve multinomiale, dans laquelle les caractéristiques sont supposées être le produit d'une distribution multinomiale simple. Cette distribution décrit les probabilités d'observer des quantités dans plusieurs catégories. C'est l'approche la plus appropriée de ce fait pour toutes les caractéristiques qui correspondent à des dénombrements.

Le raisonnement est le même que pour l'approche précédente, à la seule différence qu'au lieu d'utiliser une modélisation gaussienne optimisée des données, nous choisissons une distribution multinomiale optimisée.

Exemple de classification de texte

La technique bayésienne naïve multinomiale est souvent utilisée pour classifier du texte, les caractéristiques correspondant à des fréquences de mots trouvés dans les documents. Nous avons vu comment extraire de telles caractéristiques à partir d'un texte dans la section précédente sur l'ingénierie des caractéristiques. Voyons comment classer des documents issus de plusieurs groupes de discussion (*newsgroups*) en plusieurs catégories en nous servant des fonctions de comptage de mots épars provenant de vingt groupes.

Nous récupérons d'abord les données et demandons de connaître les noms cibles :

In[7] :

```
from sklearn.datasets import fetch_20newsgroups  
data = fetch_20newsgroups()  
data.target_names
```

Out[7] :

```
['alt.atheism',  
 'comp.graphics',
```

```
'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware',
'comp.sys.mac.hardware',
'comp.windows.x',
'misc.forsale',
'rec.autos',
'rec.motorcycles',
'rec.sport.baseball',
'rec.sport.hockey',
'sci.crypt',
'sci.electronics',
'sci.med',
'sci.space',
'soc.religion.christian',
'talk.politics.guns',
'talk.politics.mideast',
'talk.politics.misc',
'talk.religion.misc']
```

Nous allons nous limiter à quelques catégories puis télécharger le jeu d'entraînement et de test :

In[8]:

```
categories = ['talk.religion.misc',
'soc.religion.christian', 'sci.space',
'comp.graphics']

train = fetch_20newsgroups(subset='train',
categories=categories)
test = fetch_20newsgroups(subset='test',
categories=categories)
```

Voici un exemple d'entrée telle que trouvée :

In[9]: `print(train.data[5])`

From: dmcmcgee@uluhe.soest.hawaii.edu (Don McGee)
Subject: Federal Hearing
Originator: dmcmcgee@uluhe
Organization: School of Ocean and Earth Science
and Technology
Distribution: usa
Lines: 10

Fact or rumor ? Madalyn Murray O'Hare an atheist who eliminated the use of the bible reading and prayer in public schools 15 years ago is now going to appear before the FCC with a petition to stop the reading of the Gospel on the airways of America. And she is also campaigning to remove Christmas programs, songs, etc from the public schools. If it is true then mail to Federal Communications Commission 1919 H Street Washington DC 20054 expressing your opposition to her request. Reference Petition number 2493.

(N.d.T.) : Ce courriel parle d'une pétition contre une athéiste rigoriste (assassinée en 1995) qui avait fait supprimer les



prières dans les écoles aux USA et demandait d'interdire la lecture des Évangiles dans les avions et les chants de Noël dans les écoles.

Pour que ces données puissent servir à un apprentissage machine, il faut convertir le contenu de chaque chaîne en un vecteur de nombres. Nous nous servons du vectoriseur TF-IDF présenté plus haut et créons un pipeline de traitement qui injecte son résultat dans un classifieur bayésien naïf multinomial :

In[10]:

```
from sklearn.feature_extraction.text import  
TfidfVectorizer  
from sklearn.naive_bayes import MultinomialNB  
from sklearn.pipeline import make_pipeline  
  
model = make_pipeline(TfidfVectorizer(),  
MultinomialNB())
```

Nous pouvons ensuite appliquer notre modèle aux données d'entraînement et prédire les labels des données de test :

In[11]:

```
model.fit(train.data, train.target)  
labels = model.predict(test.data)
```

Une fois les labels trouvés, nous pouvons estimer les performances. La [Figure 5.41](#) montre la matrice de confusion

entre labels réels et labels prédis pour les données de test :

In[12]:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True,
fmt='d', cbar=False,
xticklabels=train.target_names,
yticklabels=train.target_names)
plt.xlabel('Label vrai')
plt.ylabel('Label prédit');
```

	comp.graphics	sci.space	soc.religion.christian	talk.religion.misc
Label prédict	344	6	1	4
	comp.graphics	sci.space	soc.religion.christian	talk.religion.misc
Label vrai	344	364	392	48
comp.graphics	13	32	24	0
sci.space	6	5	0	0
soc.religion.christian	1	12	392	187
talk.religion.misc	4	12	0	48

Figure 5.41 : Matrice de confusion pour un classifieur de texte bayésien naïf multinomial.

Même un classifieur aussi simple distingue correctement les discussions à propos de l'espace de celles à propos de l'informatique. En revanche, il mélange les textes concernant la religion avec ceux à propos de la chrétienté, ce qui est après tout prévisible.

Une conséquence intéressante est la disponibilité des outils qui permettront de trouver la catégorie pour n'importe quelle chaîne de texte en utilisant la méthode predict() du pipeline. Définissons une fonction pour renvoyer la prédiction pour une chaîne :

```
In[13]:  
def predict_category(s, train=train,  
model=model):  
    pred = model.predict([s])  
    return train.target_names[pred[0]]
```

Testons cette nouvelle fonction :

```
In[14]:  
predict_category('sending a payload to the ISS')
```

```
Out[14]: 'sci.space'
```

```
In[15]: predict_category('discussing islam vs  
atheism')
```

```
Out[15]: 'soc.religion.christian'
```

```
In[16]: predict_category('determining the screen  
resolution')
```

```
Out[16]: 'comp.graphics'
```

Rappelons qu'il ne s'agit ici que d'un modèle probabiliste simple traitant la fréquence pondérée de chaque mot dans une chaîne. Le résultat est remarquable. Ainsi, un algorithme très naïf correctement entraîné sur un jeu de données comportant beaucoup de dimensions peut se montrer tout à fait efficace.

Domaine d'emploi du bayésien naïf

Les classifieurs bayésiens naïfs émettent des hypothèses lourdes à propos des données et ne peuvent donc pas rivaliser en terme de performances avec les modèles plus complexes. Ils ont néanmoins plusieurs avantages :

- Ils fonctionnent très vite pour l'entraînement et pour la prédiction.
- Ils fournissent des prédictions de probabilité compréhensibles.
- Ils sont souvent faciles à interpréter.
- Ils offrent pas ou peu de paramètres ajustables.

Ce sont les raisons pour lesquelles le classifieur bayésien naïf est souvent préconisé pour une première classification de référence. Si les résultats suffisent, vous avez trouvé une solution à votre problème avec un classifieur très rapide et très facile à interpréter. Si les résultats sont décevants, vous vous tournez vers des modèles plus sophistiqués, tout en profitant d'un minimum de connaissances vous permettant de savoir en quoi ces modèles doivent se montrer meilleurs.

Les classifieurs bayésiens naïfs sont particulièrement performants dans chacune des situations suivantes :

- lorsque les hypothèses naïves coïncident réellement avec les données (c'est très rare en pratique) ;
- lorsque les catégories sont bien distinctes et que la complexité du modèle a moins d'importance ;
- lorsqu'il y a beaucoup de dimensions dans les données et que la complexité du modèle a moins d'importance.

En fait, les deux derniers points sont liés : lorsque le nombre de dimensions d'un jeu de données augmente, les chances que deux points restent proches diminuent (ils doivent rester proches dans toutes les dimensions). Autrement dit, les groupes locaux tendent à être plus distincts en moyenne si le nombre de dimensions est important, plus que pour un nombre de dimensions inférieur (en supposant évidemment que les nouvelles dimensions apportent d'autres informations). C'est pour cette raison que les classifieurs bayésiens naïfs fonctionnent aussi bien ou mieux que des classifieurs plus complexes lorsque le nombre de dimensions croît : dès que vous avez assez de données, même un modèle simple peut se montrer efficace.

5.6 : Régression linéaire

Nous venons de voir la technique bayésienne naïve qui constitue un bon point de départ pour les tâches de classification. En ce qui concerne les tâches de régression, le modèle équivalent est celui de *régression linéaire*. Ce modèle est populaire parce qu'il est rapide à ajuster et que ses résultats sont faciles à interpréter. Vous connaissez certainement sa forme la plus simple qui consiste à chercher à ajuster une ligne droite à des données, mais il existe des variantes beaucoup plus complexes, permettant de s'ajuster à des données plus complexes.

Nous allons commencer par une courte présentation des éléments mathématiques qui supportent ce modèle puis verrons comment on peut généraliser les modèles linéaires pour les adapter à des structures de données plus sophistiquées. Nous commençons par demander les imports habituels :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()  
import numpy as np
```

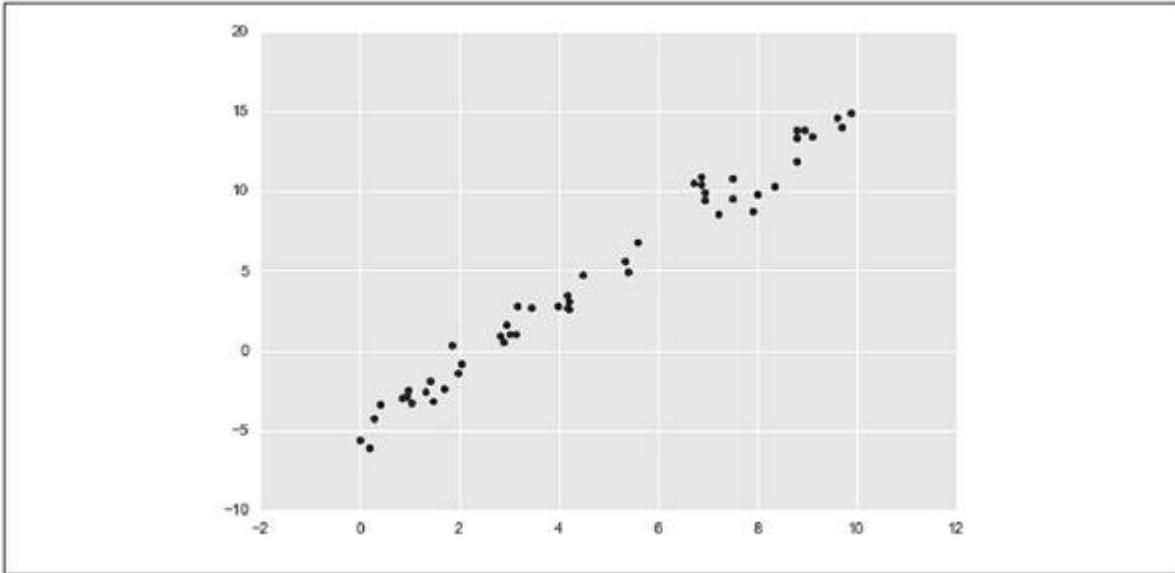
Régression linéaire simple

Nous commençons par l'ajustement d'une ligne droite qui est la régression linéaire la plus simple. Cette ligne équivaut à l'équation $y = ax + b$, avec a la pente et b l'*intercepteur*.

Commençons par générer le jeu de données. Nous choisissons de le répartir à peu près le long d'une ligne avec une pente de 2 et un intercepteur de -5 ([Figure 5.42](#)) :

In[2] :

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);
```

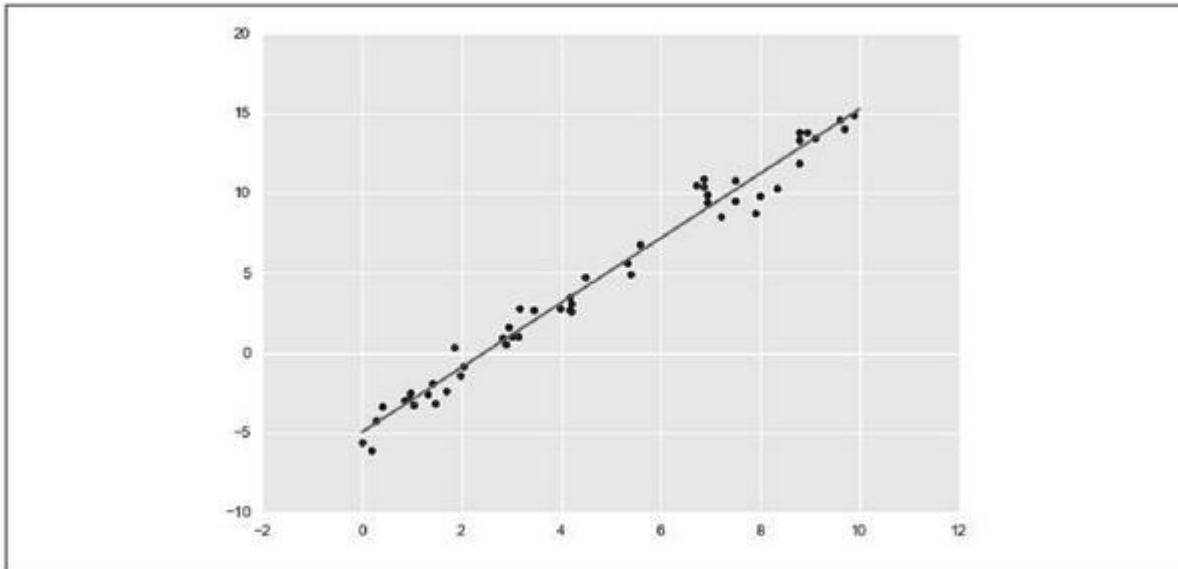


[Figure 5.42](#) : Données d'entrée de l'exemple de régression linéaire.

Nous profitons de l'estimateur de Scikit-Learn nommé LinearRegression pour demander l'ajustement de la ligne aux données et la construction de celle-ci ([Figure 5.43](#)) :

In[3] :

```
from sklearn.linear_model import  
LinearRegression  
model = LinearRegression(fit_intercept=True)  
  
model.fit(x[:, np.newaxis], y)  
  
xfit = np.linspace(0, 10, 1000)  
yfit = model.predict(xfit[:, np.newaxis])  
  
plt.scatter(x, y)  
plt.plot(xfit, yfit);
```



[Figure 5.43](#) : Un modèle de régression linéaire.

La pente et l'intercepteur des données ont été stockés dans les paramètres d'ajustement du modèle. Dans Scikit-Learn, les variables correspondantes se caractérisent par un soulignement en suffixe. Ils portent donc le nom `coef_` et `intercept_` :

In[4] :

```
print("Pente du modèle : ",  
model.coef_[0])  
print("Intercepteur du modèle : ",  
model.intercept_)
```

```
Pente du modèle : 2.02720881036  
Intercepteur du modèle : -4.99857708555
```

Nous constatons que le résultat calculé reste très proche des valeurs d'entrée, ce que nous espérions évidemment.

L'estimateur `LinearRegression` permet de réaliser bien plus que ce que nous venons de voir avec une simple ligne droite. Il permet de produire des modèles linéaires à plusieurs dimensions dans le format suivant :

$$y = a_0 + a_1 x_1 + a_2 x_2 + \dots$$

Les valeurs en `x` sont multiples. En termes géométriques, l'opération revient à ajuster un plan à un ensemble de points en trois dimensions, ou à ajuster un hyperplan à des points dans un nombre de dimensions encore supérieur.

Le résultat devient bien sûr moins facile à visualiser lorsqu'il y a plus de dimensions, mais nous pouvons néanmoins avoir un aperçu de ce genre d'ajustement en utilisant l'opérateur de multiplication matricielle de NumPy pour produire des données d'entrée :

In[5] :

```
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])

model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

0 . 5

[1 . 5 -2 . 1 .]

Dans l'exemple, la donnée y est construite à partir de trois valeurs en x aléatoires ; la régression linéaire récupère les coefficients utilisés pour construire les données d'entrée.

Nous pouvons donc ainsi nous servir du même estimateur LinearRegression pour demander un ajustement par rapport à une ligne, un plan ou un hyperplan. On pourrait redouter que l'approche soit limitée à des relations strictement linéaires entre les deux variables, mais nous allons voir que cette contrainte peut être allégée.

Régression par fonction de base

Une technique qui permet d'adapter la régression linéaire à des relations non linéaires entre variables consiste à transformer les données à partir d'une fonction de base. Nous en avons déjà vu un exemple lorsque nous avons construit un pipeline de traitement polynomial de régression dans les sections sur la validation du modèle et l'ingénierie des caractéristiques du chapitre précédent. Le principe est de partir du modèle linéaire à plusieurs dimensions :

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \dots$$

pour construire les variables x_1 , x_2 , x_3 , etc., à partir d'une seule x d'entrée à une dimension. Nous considérons en fait

que $x_n = f_n(x)$, avec $f_n()$ la fonction qui va transformer les données.

Si par exemple $x_n = f_n(x) = x^n$, le modèle devient une régression polynomiale :

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

Précisons que cela reste un modèle linéaire, le terme de linéarité désignant le fait que les coefficients en a_n ne sont jamais multipliés, ni divisés l'un par l'autre. L'opération consiste à partir des valeurs en x en une dimension pour les projeter en un nombre de dimensions supérieur, ce qui permet à un ajustement linéaire de prendre en compte des relations entre x et y plus complexes.

Fonctions de base polynomiales

Ce genre de projection polynomiale est suffisamment demandé pour avoir été intégré dans Scikit-Learn sous la forme du transformeur PolynomialFeatures :

In[6] :

```
from sklearn.preprocessing import  
PolynomialFeatures  
x = np.array([2, 3, 4])  
poly = PolynomialFeatures(3, include_bias=False)  
poly.fit_transform(x[:, None])
```

Out[6] :

```
array([[2., 4., 8.],  
       [3., 9., 27.],  
       [4., 16., 64.]])
```

Dans cet exemple, le transformeur a converti le tableau à une dimension en un tableau à trois dimensions en utilisant l'exposant de chaque valeur. Cette représentation plus riche en dimensions peut être fournie à la régression linéaire.

Comme déjà vu dans la section sur l'ingénierie des caractéristiques du chapitre précédent, la solution la plus efficace consiste à mettre en place un pipeline. Demandons un modèle polynomial à 7 degrés de cette façon :

In[7]:

```
from sklearn.pipeline import make_pipeline  
poly_model =  
make_pipeline(PolynomialFeatures(7),  
LinearRegression())
```

Avec une telle transformation, nous pouvons exploiter le modèle linéaire pour s'ajuster correctement à des relations entre x et y plus complexes. Voici l'exemple d'une sinusoïde bruitée ([Figure 5.44](#)):

In[8]:

```
rng = np.random.RandomState(1)  
x = 10 * rng.rand(50)  
y = np.sin(x) + 0.1 * rng.randn(50)
```

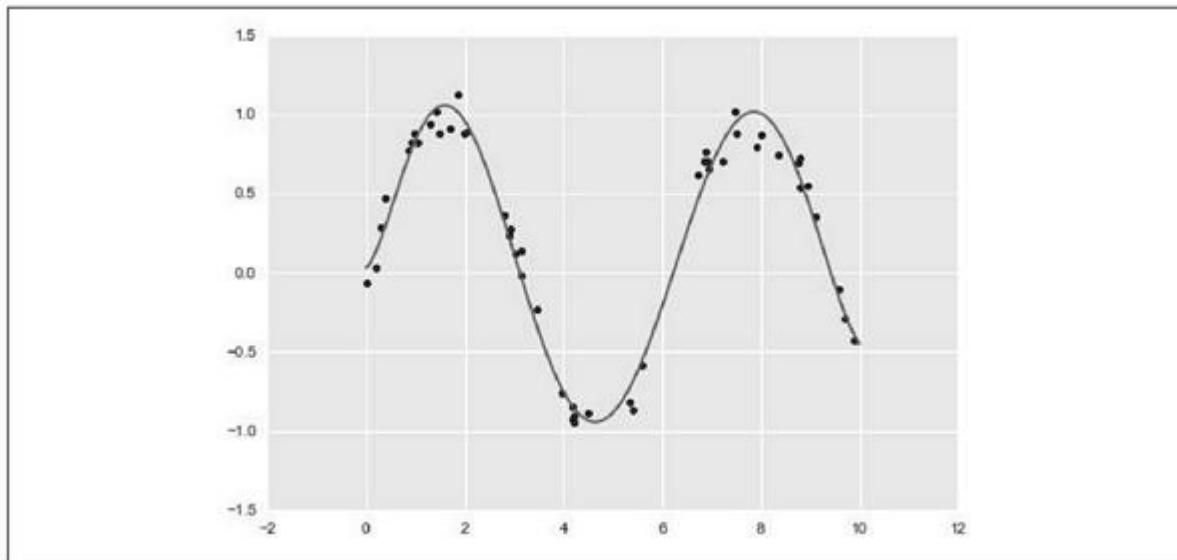
```

poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit);

```

En exploitant des fonctions de base polynomiales de degré 7, notre modèle linéaire réussit à proposer un excellent ajustement à des données non linéaires !

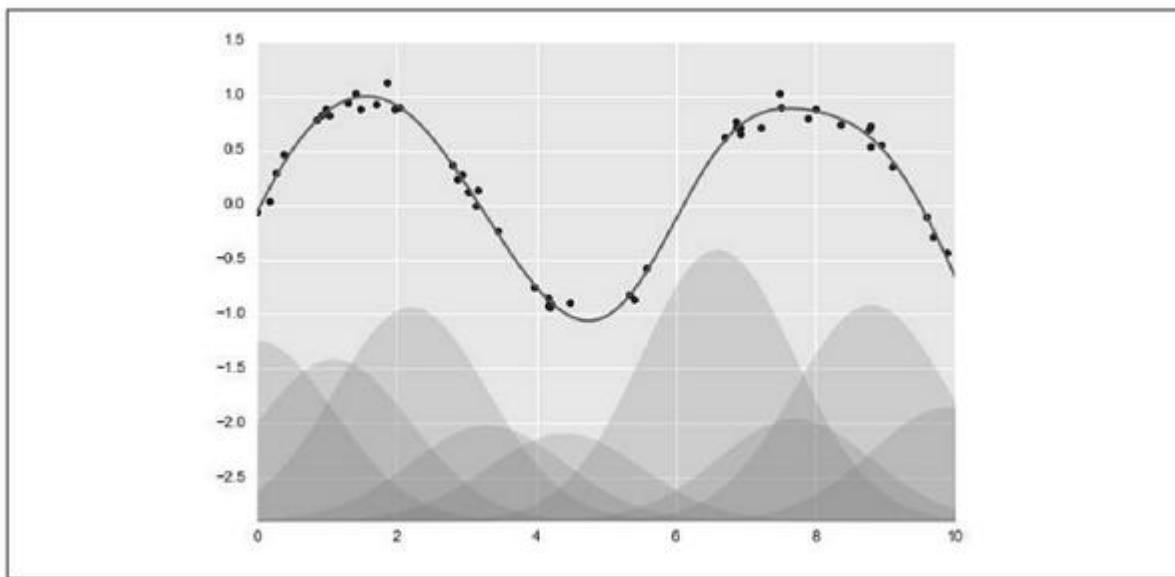


[Figure 5.44](#) : Ajustement d'un polynôme linéaire à des données d'entraînement non linéaires.

Fonctions de base gaussiennes

D'autres fonctions de base sont envisageables. Un motif intéressant est celui qui cherche à ajuster un modèle qui n'est pas la somme de base polynomiale, mais la somme de

base gaussienne. La [Figure 5.45](#) montre à quoi peut ressembler le résultat de ce traitement.



[Figure 5.45](#) : Ajustement par fonction de base gaussienne à des données non linéaires.

Les zones grisées dans le bas de la figure précédente correspondent aux fonctions de base élémentaires. Lorsqu'elles sont additionnées, elles épousent la courbe qui suit les données. Notez que ces fonctions de base gaussiennes ne sont pas intégrées à Scikit-Learn. Nous pouvons facilement rédiger un transformeur spécifique permettant de les créer, comme nous allons le voir ici avec le résultat en [Figure 5.46](#) (Les transformateurs Scikit-Learn sont écrits sous forme de classes Python ; pour apprendre comment en créer, consultez le code source de Scikit-Learn.) :

In[9]:

```
from sklearn.base import BaseEstimator,
TransformerMixin
class GaussianFeatures(BaseEstimator,
TransformerMixin):
    """Caract. gaussiennes espacées uniformément
pour entrées en 1D"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2,
axis))

    def fit(self, X, y=None):
        # Création N centres distribués le long
        # de la plage de données
        self.centers_ = np.linspace(X.min(),
X.max(), self.N)
        self.width_ = self.width_factor *
        (self.centers_[1] - self.centers_[0])
        return self

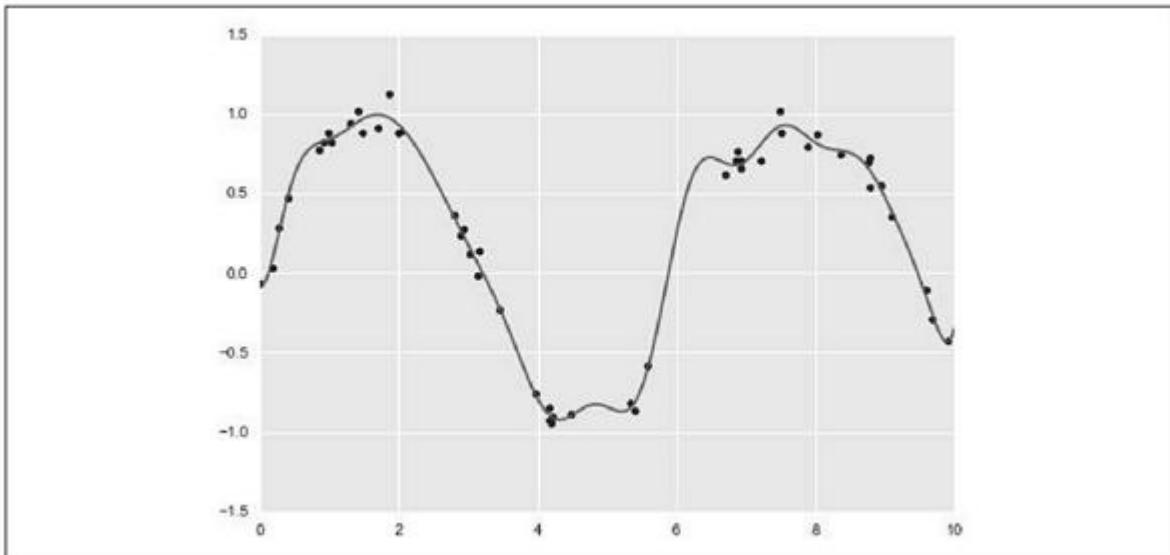
    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis],
self.centers_,
```

```

    self.width_, axis=1)

gauss_model =
make_pipeline(GaussianFeatures(20), LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])
plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);

```



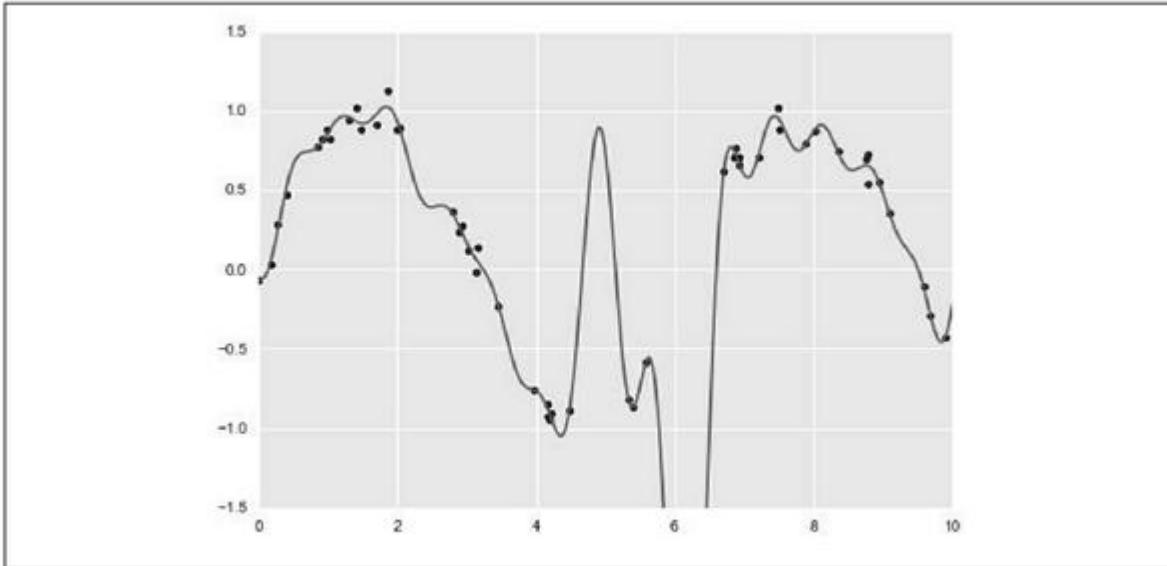
[Figure 5.46](#) : Ajustement d'une fonction de base gaussienne avec un transformeur spécifique.

Le but de l'exemple est de montrer qu'il n'y a rien de magique dans les fonctions de base polynomiales. Si vous avez une idée du processus de génération de vos données et que cela semble convenir à une base plutôt qu'à une autre, n'hésitez pas à l'adopter.

Régularisation

En utilisant des fonctions de base dans leur régression linéaire, le modèle devient beaucoup plus adaptable, mais on prend le risque d'arriver plus rapidement en surajustement (nous avons parlé de ce souci dans la section sur la validation des modèles du chapitre précédent). Si nous choisissons un trop grand nombre de fonctions de base gaussiennes, nous aboutissons à des résultats peu satisfaisants ([Figure 5.4.7](#)) :

```
In[10]:  
model = make_pipeline(GaussianFeatures(30),  
                      LinearRegression())  
model.fit(x[:, np.newaxis], y)  
  
plt.scatter(x, y)  
plt.plot(xfit, model.predict(xfit[:,  
                               np.newaxis]))  
plt.xlim(0, 10)  
plt.ylim(-1.5, 1.5);
```



[Figure 5.47](#) : Surajustement provoqué par un modèle à fonctions de base trop complexe.

Lorsque les données sont ainsi projetées sur une base à 30 dimensions, le modèle devient trop souple et va chercher les valeurs extrêmes entre les points de données auxquels il doit s'approcher. Pour connaître l'origine du souci, nous pouvons demander l'affichage des coefficients des bases gaussiennes en fonction de leur position ([Figure 5.48](#)):

In[11]:

```
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    if title is not None:
        ax[1].set_title(title)
    ax[1].plot(xfit, model.coef_.T)
    ax[1].set(xlabel='Position', ylabel='Coefficient value')
```

```

        ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5,
1.5))

    if title:
        ax[0].set_title(title)
    ax[1].plot(model.steps[0][1].centers_,
model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
ylabel='coefficient',
xlim=(0, 10))

model = make_pipeline(GaussianFeatures(30),
LinearRegression()) basis_plot(model)

```

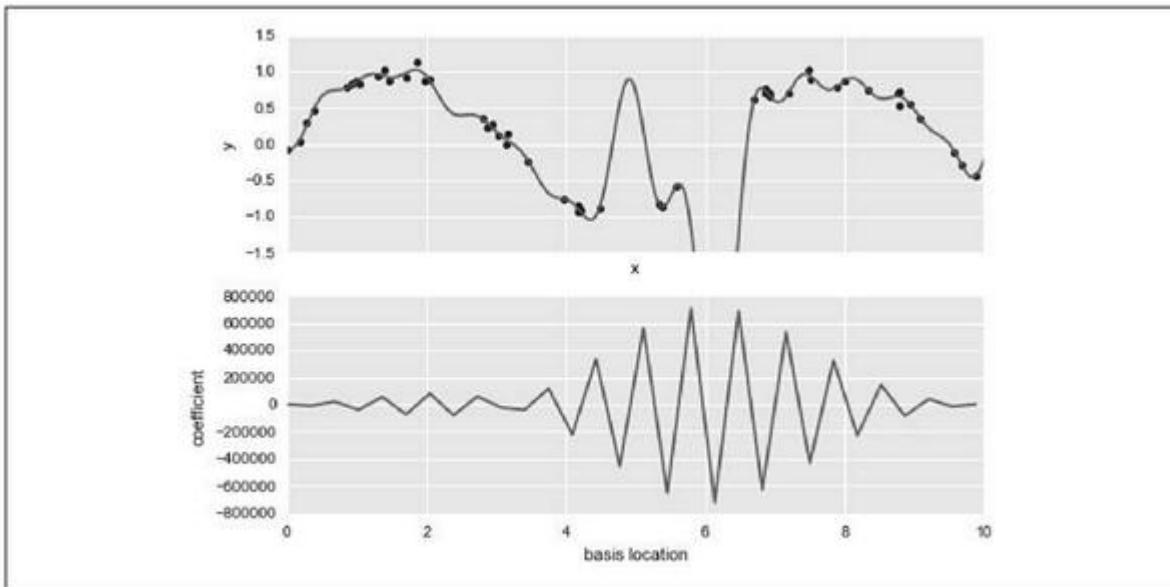


Figure 5.48 : Coefficients des bases gaussiennes dans un modèle trop complexe.

Le panneau inférieur de la [Figure 5.48](#) montre l'amplitude de la fonction de base dans chaque position. C'est le signe typique d'un surajustement dû au fait que les fonctions de

base se chevauchent : les coefficients des fonctions voisines explosent et s'annulent les uns les autres. Ce problème étant connu, il sera intéressant de pouvoir limiter ces pointes de façon explicite dans le modèle en pénalisant les trop fortes valeurs dans les paramètres. Cette pénalisation correspond à la régularisation, qui est disponible dans plusieurs formats.

Régression L₂ ou Ridge

La technique de régularisation sans doute la plus répandue est la régularisation L₂, également appelée régularisation Tikhonov, qui correspond à la régression *ridge*. Elle consiste à pénaliser la somme des carrés en norme 2 des coefficients des modèles. La pénalité d'ajustement du modèle s'écrirait ainsi :

$$P = \alpha \sum_{n=1}^N \theta_n^2$$

Dans l'équation, α est un paramètre libre qui détermine la force d'impact de la pénalité. Scikit-Learn dispose en interne de ce modèle pénalisant grâce à son estimateur Ridge ([Figure 5.49](#)) :

In[12]:

```
from sklearn.linear_model import Ridge
model = make_pipeline(GaussianFeatures(3@),
Ridge(alpha=@.1))
basis_plot(model, title='Régression Ridge')
```

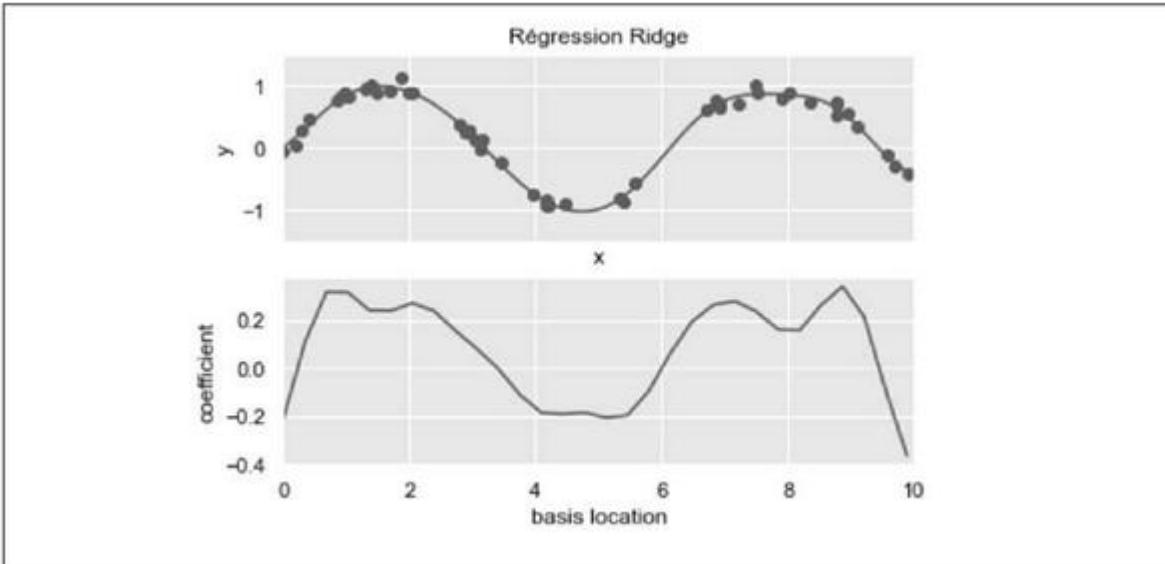


Figure 5.49 : Régularisation L_2 appliquée à un modèle trop complexe (comparez à la Figure 5.48).

Le paramètre α équivaut à un bouton pour contrôler précisément la complexité du modèle résultant. Dans le cas limite où $\alpha \rightarrow 0$, nous obtenons la régression linéaire standard. Dans le cas de la limite $\alpha \rightarrow \infty$, toutes les réponses seront supprimées. Un avantage de la régression Ridge est qu'elle se calcule de façon très efficace, en ne demandant quasiment pas plus de puissance de traitement que le modèle de régression linéaire originel.

Régularisation Lasso (L_1)

L'autre type de régularisation très répandu correspond au lasso. Il consiste à pénaliser la somme des valeurs absolues (en norme 1) des coefficients de régression :

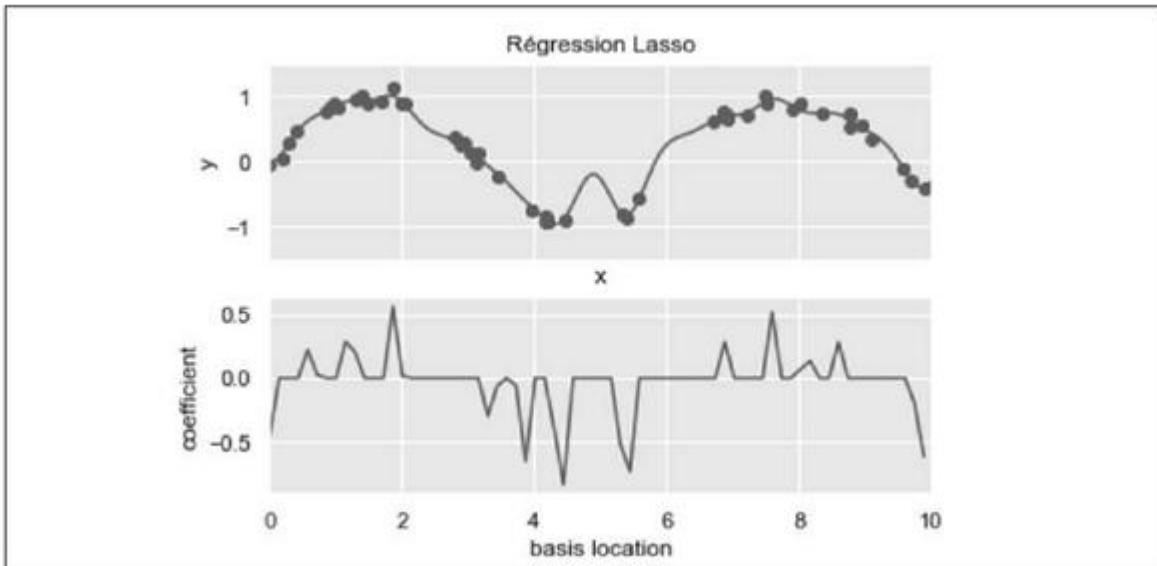
$$P = \alpha \sum_{n=1}^N |\theta_n|$$

Le concept reste très proche de celui de la régularisation L₂, mais les résultats peuvent diverger fortement. Pour des raisons géométriques, la régularisation Lasso a tendance à favoriser les modèles épars ou creux dès que possible. Autrement dit, le lasso cherche à ramener exactement à zéro les coefficients du modèle.

Nous pouvons en juger sur pièces en dupliquant le tracé de la [Figure 5.4.9](#), mais en utilisant cette fois-ci des coefficients normalisés L₁ ([Figure 5.50](#)) :

In[13]:

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(3@),
Lasso(alpha=@.@@1))
basis_plot(model, title='Régression Lasso')
```



[Figure 5.50](#) : Une régularisation Lasso L_1 appliquée à un modèle trop complexe (comparez à la Figure 5.48).

Le résultat de cette pénalisation Lasso est que la plupart des coefficients valent exactement zéro. Le comportement fonctionnel est modélisé par un petit sous-ensemble des fonctions de base disponibles. Comme dans le cas de la régularisation L_2 , c'est le paramètre alpha (α) qui contrôle la puissance de la pénalisation. Il doit être normalement déterminé par une validation croisée (revoyez la section sur la validation du modèle du [Chapitre 4](#) à ce sujet).

Exemple : prédiction d'un trafic de cyclistes

Pour un exemple de régression linéaire, essayons de prédire la fréquentation du pont Fremont Bridge de Seattle par des

cyclistes, en fonction de la météo, de la saison et d'autres critères. Nous avons déjà travaillé avec ce fichier lorsque nous avons vu les séries temporelles dans le [Chapitre 3](#).

Nous allons fusionner les données de fréquentation des cyclistes avec un jeu de données de météorologie afin de voir en quoi la température, les précipitations et l'ensoleillement influent sur le nombre de cyclistes empruntant le pont. Nous récupérons les données d'une station du service météo américain NOAA (précisément la station ID USW00024233). Pandas permet de combiner facilement les deux jeux de données. Nous allons tenter une régression linéaire simple et voir en quoi un réglage d'un des paramètres affecte la fréquentation du pont pour un jour donné.

Cet exemple va illustrer à quel point les outils de Scikit-Learn peuvent servir à créer une infrastructure de modélisation statistique, afin que les paramètres du modèle possèdent une signification restant interprétable. Ce n'est pas l'approche classique en apprentissage machine, mais certains modèles permettent ce genre d'interprétation.

Nous commençons par charger les deux jeux de données, qui sont indexés par dates :

In[14]:

```
import pandas as pd
counts = pd.read_csv('fremont_hourly.csv',
```

```
index_col='Date',
          parse_dates=True)
weather = pd.read_csv('599021.csv',
index_col='DATE', parse_dates=True)
```

Nous demandons ensuite de calculer la somme du trafic de cyclistes par jour et stockons le résultat dans un cadre DataFrame :

```
In[15]:
daily = counts.resample('d', how='sum')
daily['Total'] = daily.sum(axis=1)
daily = daily[['Total']] # Eliminer autres
colonnes
```

Nous avions vu lors du précédent exemple concernant les cyclistes que le trafic varie d'un jour à l'autre. Nous allons en tenir compte en ajoutant des colonnes binaires pour coder le jour de la semaine :

```
In[16]:
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri',
'Sat', 'Sun']
for i in range(7):
    daily[days[i]] = (daily.index.dayofweek ==
i).astype(float)
```

Nous prévoyons que les cyclistes se comportent différemment les jours fériés. Nous prévoyons un indicateur

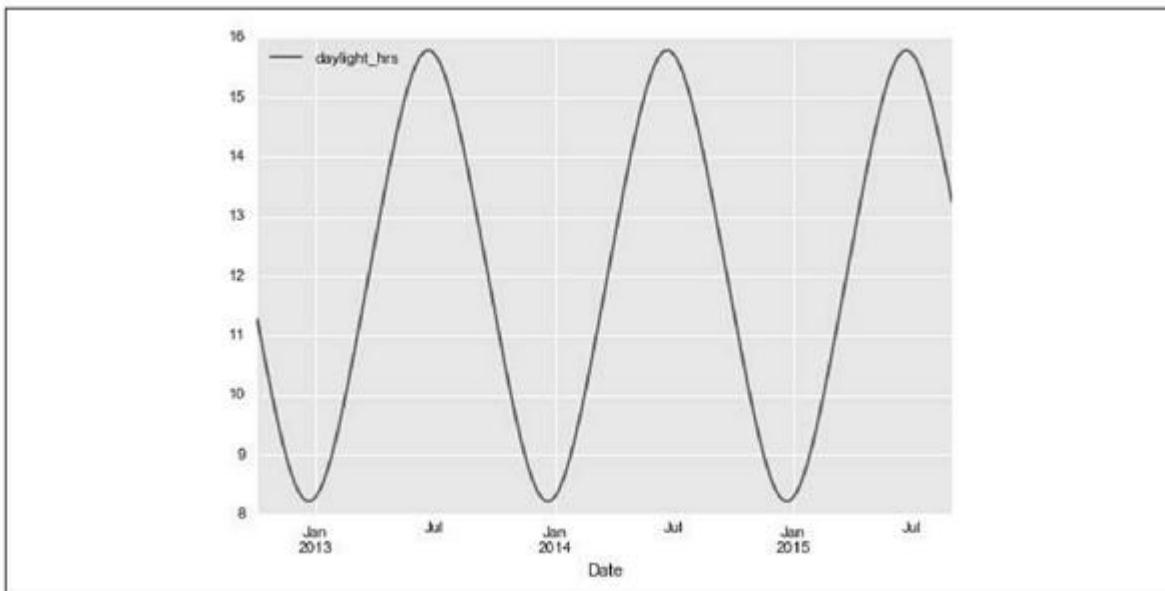
à cet effet :

```
In[17]:  
from pandas.tseries.holiday import  
USFederalHolidayCalendar  
cal = USFederalHolidayCalendar()  
holidays = cal.holidays('2012', '2016')  
daily = daily.join(pd.Series(1, index=holidays,  
name='holiday'))  
daily['holiday'].fillna(0, inplace=True)
```

Nous supposons enfin que le trafic varie selon l'heure, pour voir la différence entre plein jour et pénombre. Nous nous servons du calcul astronomique standard pour ajouter l'information ([Figure 5.51](#)) :

```
In[18]:  
def hours_of_daylight(date, axis=23.44,  
latitude=47.61):  
    """Ajuste heure été/hiver pour une date"""  
    days = (date - pd.datetime(2000, 12,  
21)).days  
    m = (1. - np.tan(np.radians(latitude))  
        * np.tan(np.radians(axis)) * np.cos(days  
* 2 * np.pi / 365.25)))  
    return 24. * np.degrees(np.arccos(1 -  
np.clip(m, 0, 2))) / 180.  
  
daily['daylight_hrs'] =
```

```
list(map(hours_of_daylight, daily.index))
daily[['daylight_hrs']].plot();
```



[Figure 5.51](#) : Visualisation du nombre d'heures d'ensoleillement à Seattle.

Nous allons également ajouter la température moyenne et le total des précipitations. Nous stockons la hauteur des pluies en pouces et prévoyons un indicateur pour les jours sans pluie :

```
In[19]: # Températures en 1/10 deg C; convertir
en Celsius
weather['TMIN'] /= 10
weather['TMAX'] /= 10
weather['Temp (C)'] = 0.5 * (weather['TMIN'] +
weather['TMAX'])

# precip en 1/10 mm; convertir en pouces
weather['PRCP'] /= 254
```


2012-10-06	2006	0	0	0	0	0	1	0	0
	11.103056								
2012-10-07	2142	0	0	0	0	0	0	1	0
	11.045208								

Date	PRCP	Temp (C)	dry day	annual
2012-10-03	0	13.35	1	0.000000
2012-10-04	0	13.60	1	0.002740
2012-10-05	0	15.30	1	0.005479
2012-10-06	0	15.85	1	0.008219
2012-10-07	0	15.85	1	0.010959

Nous pouvons maintenant choisir quelle colonne nous allons utiliser et demander un ajustement de la régression linéaire. Nous ajoutons en paramètre `fit_intercept = False`, car les indicateurs quotidiens constituent leur propre interception pour chaque journée :

In[22]:

```
column_names = ['Mon', 'Tue', 'Wed', 'Thu',
'Fri', 'Sat', 'Sun', 'holiday',
'daylight_hrs', 'PRCP', 'dry
day', 'Temp (C)', 'annual']
```

```
X = daily[column_names]
```

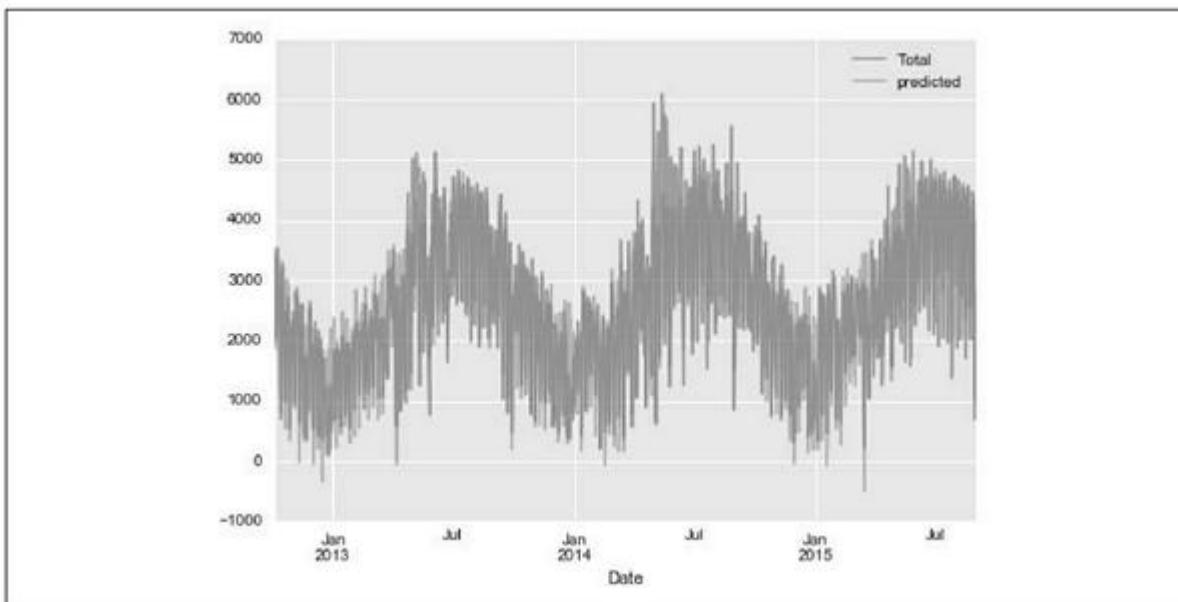
```
y = daily['Total']
```

```
model = LinearRegression(fit_intercept=False)
```

```
model.fit(x, y)
daily['predicted'] = model.predict(x)
```

Nous pouvons demander enfin un tracé du trafic de cyclistes total et prédit ([Figure 5.52](#)) :

```
In[23]: daily[['Total',
'predicted']].plot(alpha=0.5);
```



[Figure 5.52](#) : Les prédictions de notre modèle concernant le trafic de cyclistes.

Nous voyons clairement que nous avons oublié certaines caractéristiques, notamment pour les périodes d'été. Soit il nous manque des caractéristiques, par exemple parce que les gens choisissent de prendre leur vélo pour aller travailler en fonction d'autres critères, soit il y a une relation non linéaire que nous n'avons pas réussi à prendre en compte (peut-être que les gens prennent moins souvent leur vélo

lors des températures très basses ou très hautes). Cela dit, notre approximation nous donne déjà une idée globale et nous pouvons demander à voir les coefficients du modèle linéaire afin de juger de l'impact de chacune des caractéristiques sur le comptage quotidien de cyclistes :

In[24]:

```
params = pd.Series(model.coef_, index=x.columns)
params
```

Out[24]:

```
Mon           503.797330
Tue           612.088879
Wed           591.611292
Thu           481.250377
Fri           176.838999
Sat          -1104.321406
Sun          -1134.610322
holiday      -1187.212688
daylight_hrs 128.873251
PRCP          -665.185105
dry day       546.185613
Temp (C)      65.194390
annual        27.865349
dtype: float64
```

Pour bien interpréter ces valeurs, il faudrait disposer d'une mesure de leur incertitude. Ces incertitudes sont faciles à

obtenir en demandant un rééchantillonnage interne des données (*bootstrap*) :

In[25]:

```
from sklearn.utils import resample
np.random.seed(1)
err = np.std([model.fit(*resample(X, y)).coef_
             for i in range(1000)], 0)
```

Une fois cette estimation d'erreurs réalisée, nous pouvons réafficher les résultats :

In[26]:

```
print(pd.DataFrame({'effect': params.round(0),
                    'error': err.round(0)}))
```

	effect	error
Mon	504	85
Tue	612	82
Wed	592	82
Thu	481	85
Fri	177	81
Sat	-1104	79
Sun	-1135	82
holiday	-1187	164
daylight_hrs	129	9
PRCP	-665	62
dry day	546	33
Temp (C)	65	4
annual	28	18

Nous confirmons que la fréquentation reste assez stable sur la semaine travaillée : il y a beaucoup plus de cyclistes que le week-end et que les jours fériés. Nous voyons également que pour chaque heure d'ensoleillement de plus, il y a 129 ± 9 cyclistes de plus. De même, un degré de température Celsius en plus invite 65 ± 4 cyclistes de plus à prendre leur vélo. Un jour sans pluie voit 546 ± 33 cyclistes de plus. Chaque pouce de pluie laisse 665 ± 62 choisir un autre moyen d'aller travailler. Après la prise en compte de tous ces effets, nous constatons une légère augmentation de 28 ± 18 nouveaux cyclistes chaque année.

Notre modèle est certainement incomplet. Certains effets non linéaires tels qu'une combinaison de pluie et de froid ainsi que des tendances non linéaires au sein de chaque variable (comme la réticence à prendre son vélo lorsqu'il fait très chaud ou très froid) ne sont pas pris en compte. Nous avons également écarté des informations plus détaillées telles que la différence entre une pluie le matin et une pluie l'après-midi et avons ignoré les corrélations entre les jours (un mardi pluvieux peut impacter le mercredi suivant ou bien une journée particulièrement clémence après une série de jours pluvieux). Toutes ces conditions particulières peuvent dorénavant faire l'objet d'une investigation, car vous disposez des outils pour y parvenir !

5.7 : Machines à vecteurs de support (SVM)

Une classe d'algorithmes supervisés qui est efficace et très souple aussi bien pour la classification que pour la régression porte le nom de machine à vecteurs de support ou SVM (*Support Vector Machine*). Découvrons-en d'abord les principes puis son utilisation dans des problèmes de classification. Nous commençons par les opérations d'import habituelles :

```
In[1]:  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt  
from scipy import stats  
# Paramètres de tracé Seaborn par défaut  
import seaborn as sns; sns.set()
```

Avantages des machines à vecteurs de support

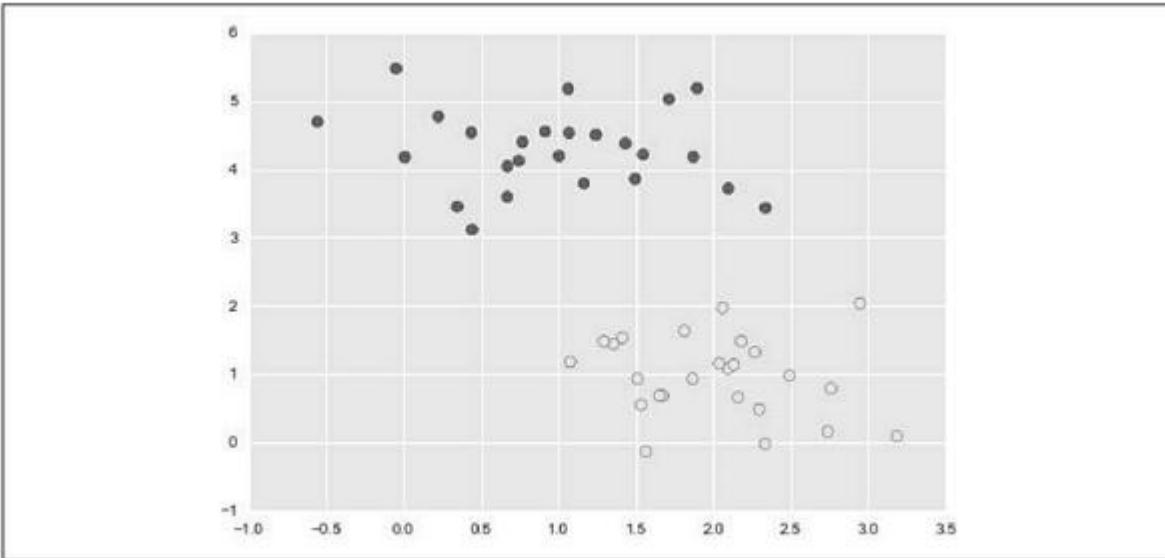
Dans la section précédente de ce chapitre qui présentait la classification bayésienne, nous avons présenté un modèle pour décrire la distribution de plusieurs classes, et nous nous en sommes servis pour déterminer de façon

probabiliste les labels de nouveaux points de données. Il s'agissait d'une *classification générative*. Nous allons maintenant voir une *classification discriminative* : au lieu de modéliser chaque classe, nous allons chercher une ligne droite ou une courbe en deux dimensions, et voir un distributeur en plus de dimensions appelé *manifold* pour distinguer les classes les unes des autres.

Lançons-nous dans une tâche de classification avec des données d'entrée qui se présentent sous forme de deux classes bien distinctes ([Figure 5.53](#)) :

In[2]:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=50, centers=2,
random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,
cmap='autumn');
```



[Figure 5.53](#) : Un jeu de données simple pour une classification.

Un classifieur linéaire chercherait à tirer une ligne droite pour séparer les deux sous-ensembles de données, ce qui produirait un modèle de classification. L'opération pourrait se faire à la main avec des données en deux dimensions comme ici. Mais il y a un vrai problème : plusieurs lignes permettent de discriminer parfaitement les deux classes. Laquelle choisir ?

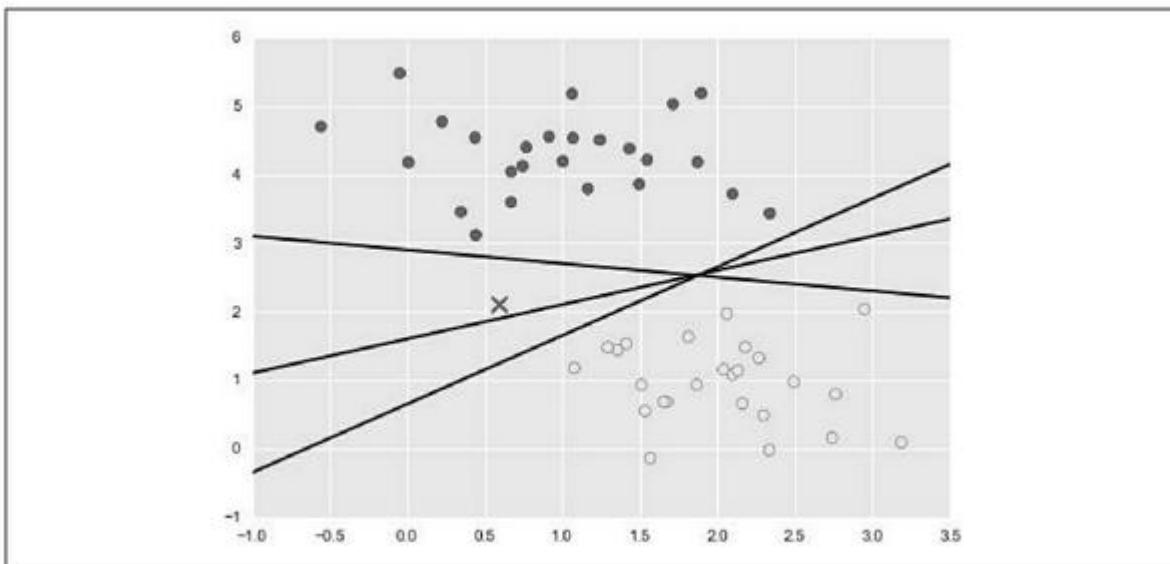
Nous pouvons même essayer de les tracer ([Figure 5.54](#)) :

In[3] :

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,
cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red',
markeredgecolor=2, markersize=10)
for m, b in [(1, 0.65), (0.5, 1.6), (-0.2,
```

2.9)]:

```
plt.plot(xfit, m * xfit + b, '-k')  
plt.xlim(-1, 3.5);
```



[Figure 5.54](#) : Trois classificateurs linéaires discriminants acceptables pour les données.

Les trois résultats sont franchement différents, mais ils séparent bien tous trois les jeux de données. En fonction de celui que vous choisissez, certains points de données seront rangés d'un côté ou de l'autre, comme c'est le cas de celui marqué « X » dans le graphique. Il semble évident que nous ne pouvons pas nous contenter de chercher à tirer un trait entre les classes. Il nous faut trouver quelque chose de plus sophistiqué.

Machines à vecteurs de support et recherche d'une marge maximale

Nous pouvons améliorer les résultats en adoptant une machine à vecteurs de support. Au lieu de tirer un trait d'épaisseur nulle entre les classes, nous créons une marge en instaurant une épaisseur qui va jusqu'au point le plus proche. Voici un premier exemple ([Figure 5.55](#)) :

```
In[4]:  
xfit = np.linspace(-1, 3.5)  
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,  
cmap='autumn')  
  
for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6,  
0.55), (-0.2, 2.9, 0.2)]:  
    yfit = m * xfit + b  
    plt.plot(xfit, yfit, '-k')  
    plt.fill_between(xfit, yfit - d, yfit + d,  
edgecolor='none',  
                     color='#AAAAAA', alpha=0.4)  
  
plt.xlim(-1, 3.5);
```

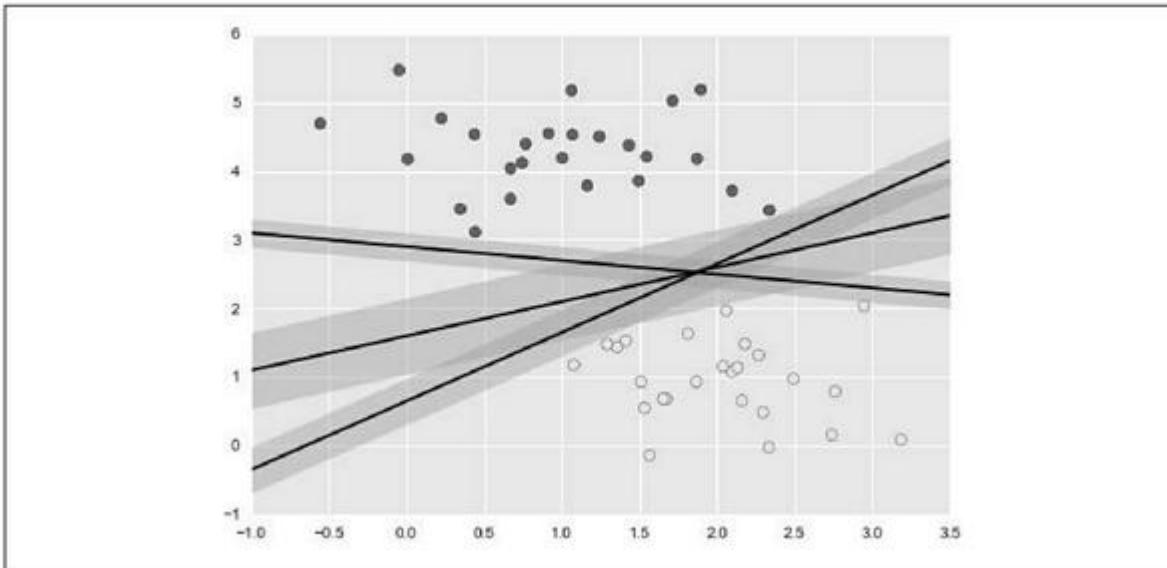


Figure 5.55 : Visualisation de « marges » avec des classifiateurs discriminants.

Dans une machine SVM, la ligne qui permet d'obtenir la plus large marge sera celle choisie comme modèle optimal. C'est pourquoi on parle d'estimateur de marge maximale.

Ajustement d'une machine SVM

Découvrons les résultats d'un essai d'ajustement avec les mêmes données. Nous nous servons du classifieur SVM de Scikit-Learn pour réaliser l'entraînement d'un modèle SVM. Dans une première étape, nous nous servons d'un noyau linéaire et donnons une valeur très grande au paramètre nommé C (nous en parlons en détail un peu plus loin) :

In[5] :

```
from sklearn.svm import SVC      # "Support vector classifier"
```

```
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

Out[5]:
SVC(C=100000000000.0)

Pour mieux rendre visibles les traitements réalisés, nous allons définir une fonction utilitaire qui va tracer les frontières de décision SVM ([Figure 5.56](#)) :

```
In[6]:
def plot_svc_decision_function(model, ax=None,
plot_support=True):
    """Trace la fonction de décision pour un SVC
en 2D"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # Création grille d'évaluation
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    X, Y = np.meshgrid(x, y)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P =
model.decision_function(xy).reshape(X.shape)

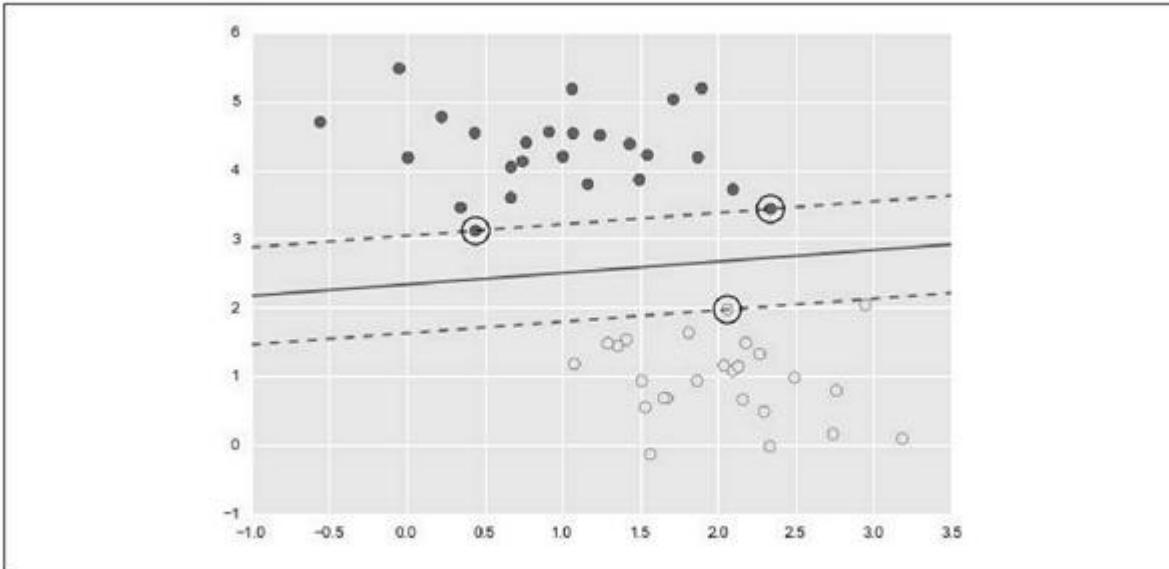
    # Tracé frontières de décision et marges
    ax.contour(X, Y, P, colors='k',
```

```
    levels=[-1, 0, 1], alpha=0.5,
linestyles=['--', '--', '-']))

# Tracé vecteurs de support
if plot_support:
    ax.scatter(model.support_vectors_[:, 0],
               model.support_vectors_[:, 1],
               s=300, linewidth=1,
               facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```

In[7]:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,
cmap='autumn')
plot_svc_decision_function(model);
```



[Figure 5.56](#) : Ajustement d'un classifieur SVM avec les marges en pointillé et les vecteurs de support en points.

Cette ligne propose la plus grande marge entre les deux ensembles de points. Vous remarquez que certains points de l'entraînement touchent la marge (ce sont ceux cerclés en noir dans la figure). Ce sont les pivots de l'ajustement et ils correspondent aux vecteurs de support d'où l'algorithme tire son nom. Dans Scikit-Learn, les paramètres de ces points sont stockés dans l'attribut du classifieur nommé `support_vectors_` :

```
In[8]:  
model.support_vectors_
```

```
Out[8]:  
array([[ 0.44359863,  3.11530945],
```

```
[ 2.33812285, 3.43116792],  
[ 2.06156753, 1.96918596]])
```

L'efficacité de ce classifieur tient au fait que ce n'est que la position des vecteurs de support qui compte : tous les points éloignés de la marge qui sont du bon côté n'influent pas sur l'ajustement. En termes techniques, c'est dû au fait que ces points ne contribuent pas à la fonction de perte utilisée pour l'ajustement du modèle. Ni leur position ni le nombre ne jouent tant qu'ils ne croisent pas la marge.

Nous pouvons le vérifier en demandant un tracé du modèle après un apprentissage à partir des 60 premiers points, puis à partir des 120 premiers points du jeu ([Figure 5.57](#)) :

```
In[9]:  
def plot_svm(N=10, ax=None):  
    X, y = make_blobs(n_samples=200, centers=2,  
                      random_state=0,  
                      cluster_std=0.60)  
    X = X[:N]  
    y = y[:N]  
    model = SVC(kernel='linear', C=1E10)  
    model.fit(X, y)  
  
    ax = ax or plt.gca()  
    ax.scatter(X[:, 0], X[:, 1], c=y, s=50,  
               cmap='autumn')  
    ax.set_xlim(-1, 4)  
    ax.set_ylim(-1, 6)
```

```

plot_svc_decision_function(model, ax)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95,
wspace=0.1)
for ax_i, N in zip(ax, [60, 120]):
    plot_svm(N, ax_i)
    ax_i.set_title('N = {}'.format(N))

```

Le graphique de gauche montre le modèle et les vecteurs de support pour 60 points d'entraînement. Le graphique de droite a doublé le nombre de points, mais le modèle n'a pas changé : les trois vecteurs de support sont au même endroit. C'est cette insensibilité à la valeur exacte des points distants qui constitue l'une des forces du modèle SVM.

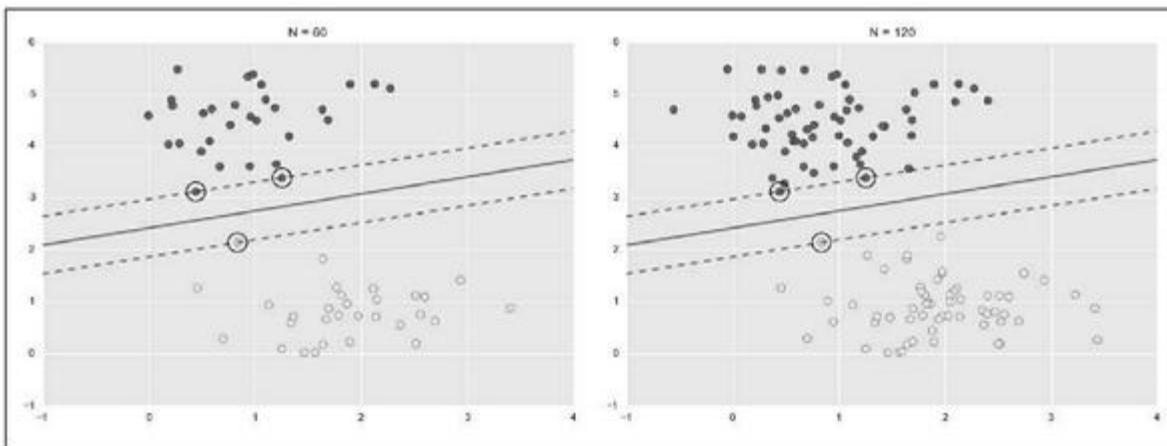


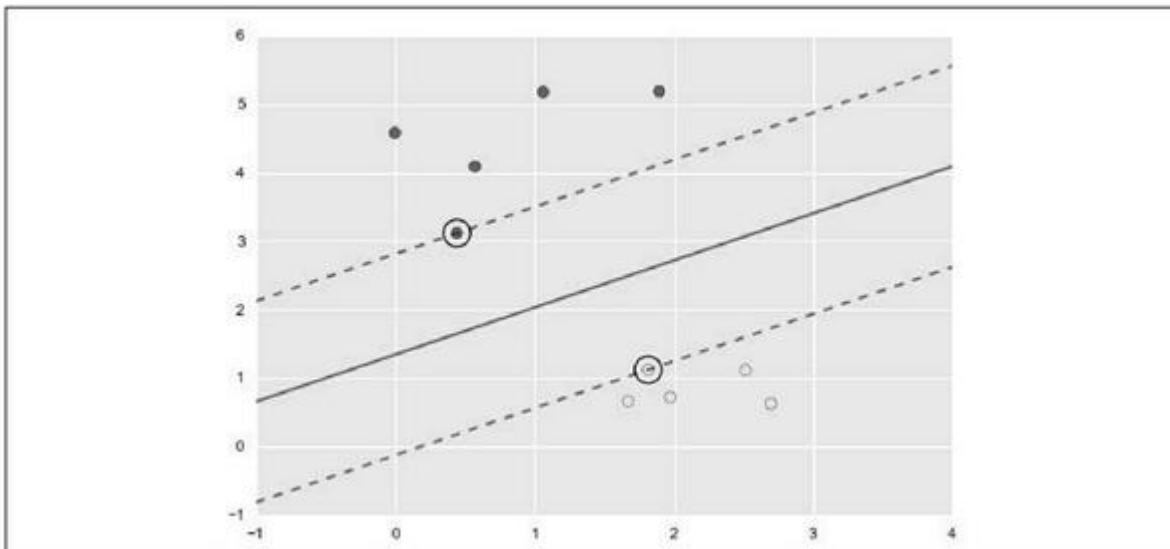
Figure 5.57 : Impact des nouveaux points d'entraînement sur le modèle SVM.

Si vous avez chargé le calepin d'exemples en mode interactif, vous pouvez manipuler les composants de

contrôle d'IPython pour vérifier cette insensibilité du modèle SVM de façon interactive ([Figure 5.58](#)) :

In[10]:

```
from ipywidgets import interact, fixed
interact(plot_svm, N=[10, 200], ax=fixed(None));
```



[Figure 5.58](#) : Exemple de visualisation interactive du modèle SVM (voyez le fichier d'exemples accompagnant le livre pour un test réel).

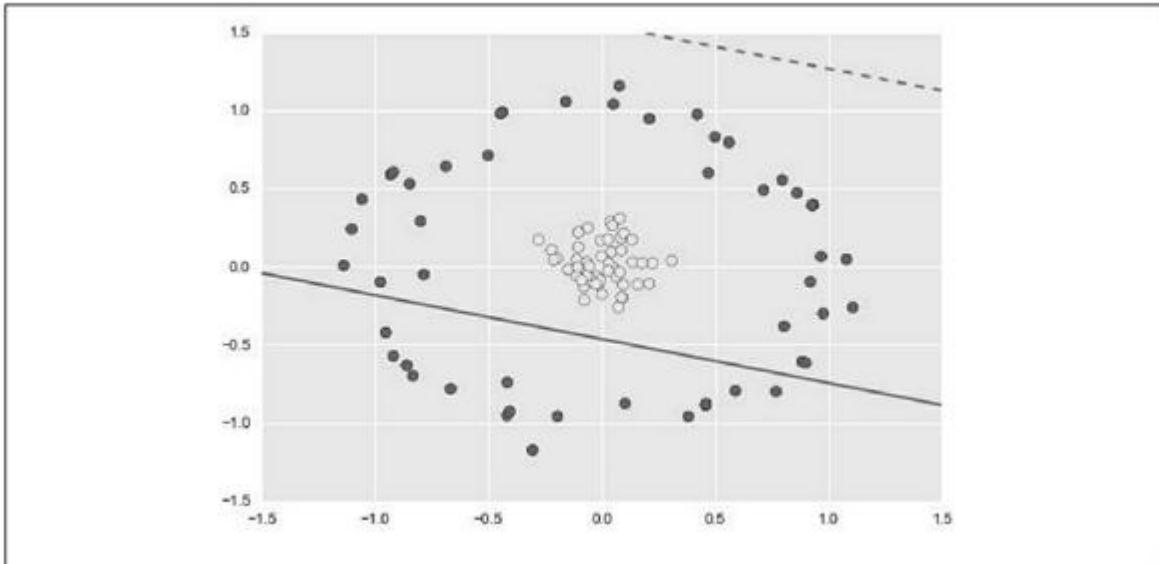
Au-delà des frontières linéaires avec le SVM à noyau

La puissance de l'approche SVM est décuplée lorsqu'elle est combinée avec les noyaux. Nous avons déjà rencontré les noyaux dans la description des régressions à fonctions de base de ce chapitre. Nous y avions projeté nos données dans un nombre supérieur de dimensions défini par des

polynômes et des fonctions de base gaussiennes. Cela nous avait permis de faire un ajustement avec un classifieur linéaire mais pour des relations non linéaires.

Nous pouvons réutiliser le même principe dans un modèle SVM. Commençons par mettre en place des données qui ne peuvent pas être distinguées de façon linéaire afin de motiver notre recours aux noyaux ([Figure 5.59](#)) :

```
In[11]:  
from sklearn.datasets import make_circles  
X, y = make_circles(100, factor=.1, noise=.1)  
  
clf = SVC(kernel='linear').fit(X, y)  
  
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,  
cmap='autumn')  
plot_svc_decision_function(clf,  
plot_support=False);
```



[Figure 5.59](#) : Un classifieur linéaire n'est pas satisfaisant pour des frontières non linéaires.

Il est évident qu'une discrimination linéaire ne pourra jamais distinguer correctement ce genre de données. Nous pouvons en revanche envisager de projeter les données en davantage de dimensions pour qu'un séparateur linéaire puisse se montrer satisfaisant. Essayons une projection simple qui consiste à calculer une *fonction de base radiale* centrée sur l'amas central :

```
In[12]: r = np.exp(-(X ** 2).sum(1))
```

Nous pouvons visualiser cette dimension supplémentaire avec un tracé en trois dimensions. Si vous utilisez le calepin en mode interactif, vous pouvez profiter des curseurs de réglage pour tourner autour du tracé ([Figure 5.60](#)) :

In[13]:

```
from mpl_toolkits import mplot3d

def plot_3D(elev=30, azim=30, X=X, y=y):
    ax = plt.subplot(projection='3d')
    ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50,
cmap='autumn')
    ax.view_init(elev=elev, azim=azim)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('r')

interact(plot_3D, elev=[-90, 90], azip=(-180,
180),
         x=fixed(X), y=fixed(y));
```

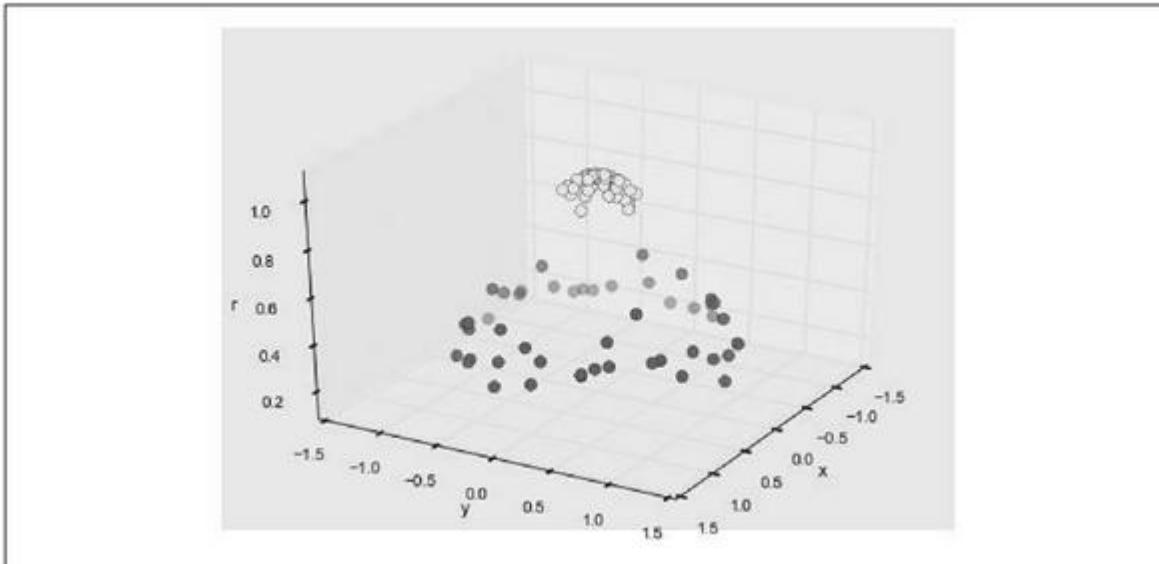


Figure 5.60 : L'ajout d'une troisième dimension va autoriser la séparation linéaire.

Nous constatons qu'avec cette dimension supplémentaire, il devient facile de distinguer les données en traçant un plan de séparation à par exemple $r=0.7$.

Cette réussite suppose néanmoins d'avoir bien optimisé la projection ; si nous n'avions pas correctement centré notre fonction de base radiale, nous n'aurions pas bénéficié d'un résultat aussi évident et linéaire. Cette obligation de faire le bon choix pose souvent problème. Nous aimeraisons beaucoup disposer d'un moyen de trouver automatiquement les fonctions de base les plus adaptées.

Une solution à ce problème consiste à faire calculer une fonction de base centrée pour chaque point du jeu de données, en laissant ensuite l'algorithme SVM analyser les résultats. Il s'agit d'une transformation de la fonction de

base qui porte le nom de *transformation de noyaux*, car elle est basée sur des relations de similarité (des noyaux) entre toutes les paires de points.

L'inconvénient de cette approche qui consiste à projeter N points dans N dimensions est qu'elle peut devenir très coûteuse en temps de calcul si N grandit beaucoup. Mais c'est sans compter une élégante astuce qui porte le nom d'*astuce du noyau* (https://fr.wikipedia.org/wiki/Astuce_du_noxyau). Il s'agit d'un ajustement réalisé implicitement sur les données transformées par les noyaux, c'est-à-dire sans jamais avoir besoin de construire la représentation complète en N dimensions de la projection de noyaux ! Cette procédure d'astuce du noyau est intégrée à la machine SVM, et c'est une des raisons qui explique la puissance de cette approche.

Dans Scikit-Learn, nous pouvons appliquer une machine SVM noyautée tout simplement en remplaçant le noyau linéaire par un noyau à fonction de base radiale RBF (*radial basis function*). Nous utilisons pour cela l'hyperparamètre de modèle portant le nom `kernel` ([Figure 5.61](#)) :

In[14]:

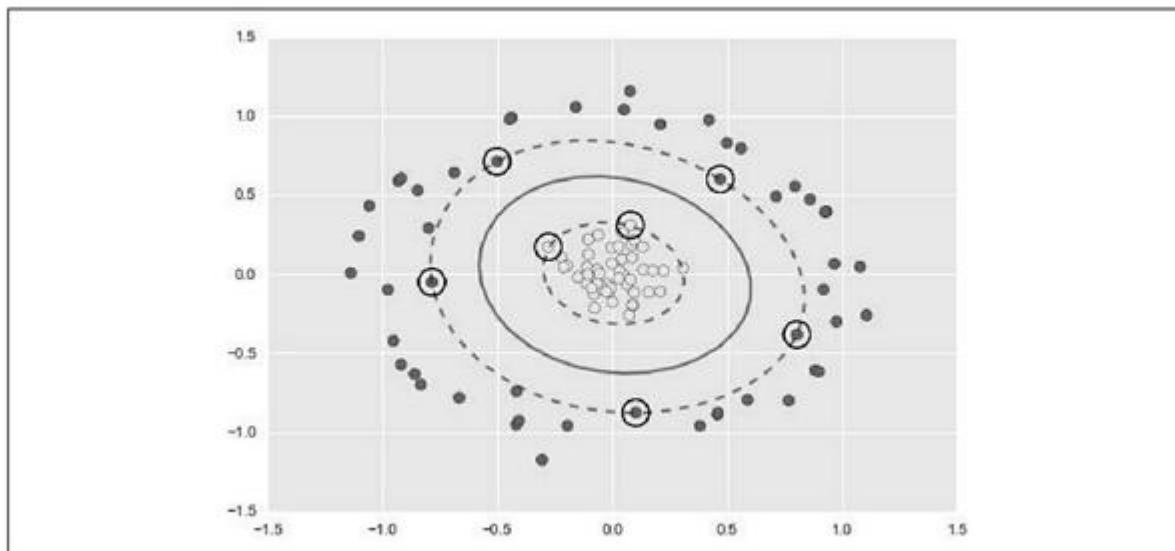
```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

Out[14]:

```
SVC(C=1000000.0)
```

```
In[15]:
```

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,
cmap='autumn')
plot_svc_decision_function(clf)
plt.scatter(clf.support_vectors_[:, 0],
clf.support_vectors_[:, 1],
s=300, lw=1, facecolors='none');
```



[Figure 5.61](#) : Ajustement de machine SVM noyautée.

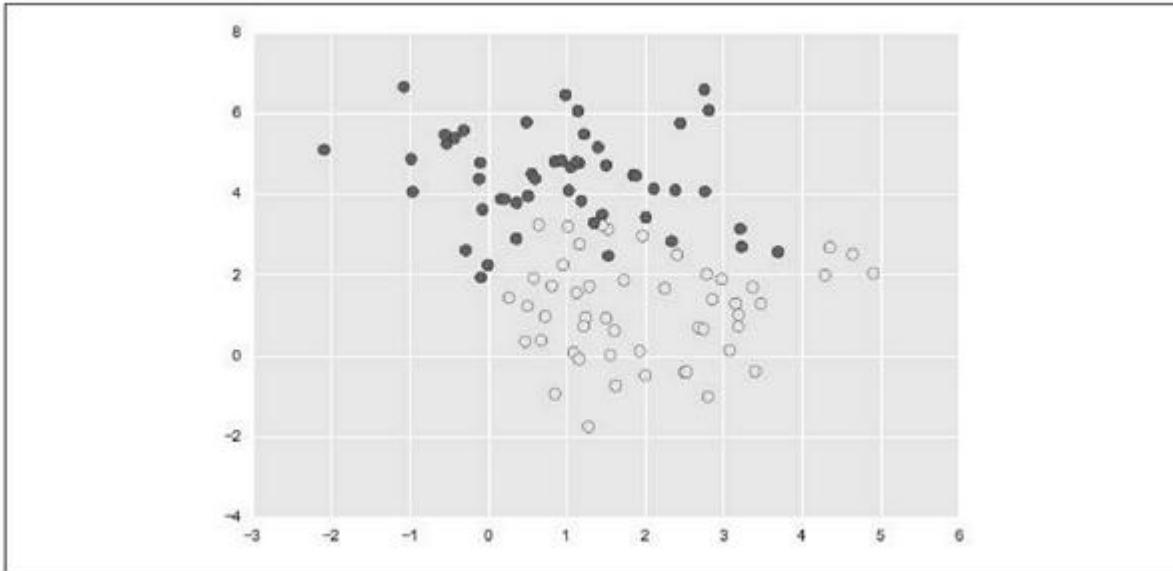
Grâce à cette SVM noyautée, il devient possible de trouver une frontière de décision non linéaire satisfaisante. Cette stratégie de transformation de noyaux est souvent utilisée en apprentissage machine pour transformer une méthode linéaire rapide en une méthode non linéaire rapide, en particulier avec les modèles qui permettent l'utilisation de l'astuce de noyau.

Optimisation de la SVM en adoucissant les marges

Jusqu'à présent, nous avons travaillé avec des jeux de données très propres, qui contiennent une frontière de décision évidente. Mais comment faire lorsqu'il y a des chevauchements dans les données ? Partons du jeu de données suivant ([Figure 5.62](#)) :

In[16]:

```
X, y = make_blobs(n_samples=100, centers=2,  
                   random_state=0,  
                   cluster_std=1.2)  
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,  
           cmap='autumn');
```



[Figure 5.62](#) : Un jeu de données avec des chevauchements.

La machine SVM est équipée d'un paramètre permettant d'augmenter la tolérance de la marge, de l'adoucir. Il devient ainsi possible à certains points d'entrer dans la zone de marge, si cela améliore l'ajustement. La souplesse de la marge est contrôlée par le paramètre d'ajustement qui porte normalement le nom C. Lorsque sa valeur est grande, la marge est rigide et aucun point ne peut y entrer. Lorsque la valeur de C est faible, la marge est adoucie et quelques points peuvent y pénétrer.

Le tracé de la [Figure 5.63](#) montre l'effet du paramètre C sur l'ajustement final, par adoucissement de marge :

In[17]:

```
X, y = make_blobs(n_samples=100, centers=2,  
random_state=0,
```

```

cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95,
wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50,
cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
axi.set_title('C = {:.1f}'.format(C), size=14)

```

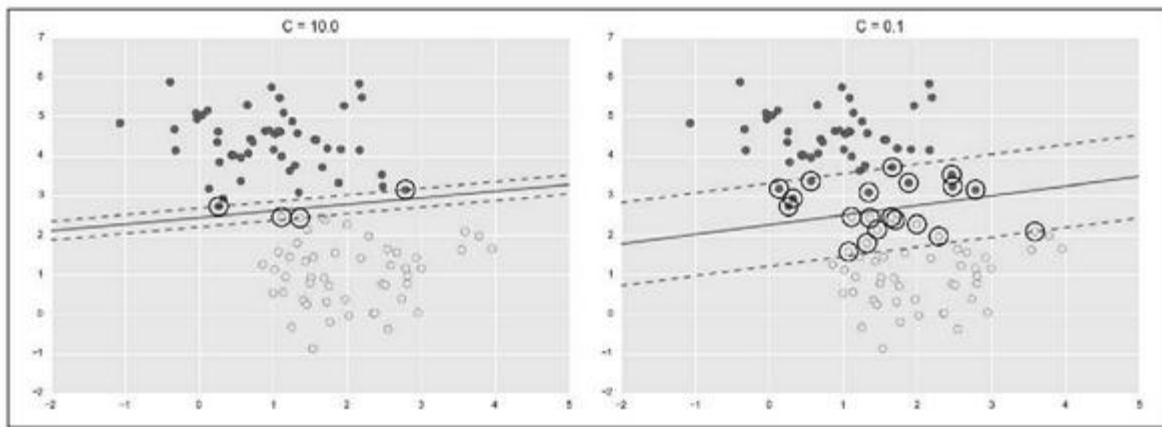


Figure 5.63 : Impact du paramètre C sur l'ajustement de la machine SVM.

La valeur idéale de C dépend bien sûr du jeu de données. Elle doit être recherchée par validation croisée ou autres

procédures similaires (revoyez la section sur les hyperparamètres et la validation de modèle si nécessaire).

Exemple : reconnaissance de visages

En guise d'exemple d'une machine SVM, attaquons-nous au problème de reconnaissance de visages. Nous allons partir du jeu de données prédéfini intitulé *Labeled Faces in the Wild* (LFW) une collection de plusieurs milliers de photos de personnalités politiques. Scikit-Learn dispose d'un outil d'extraction `fetch()` pour ce jeu de données :

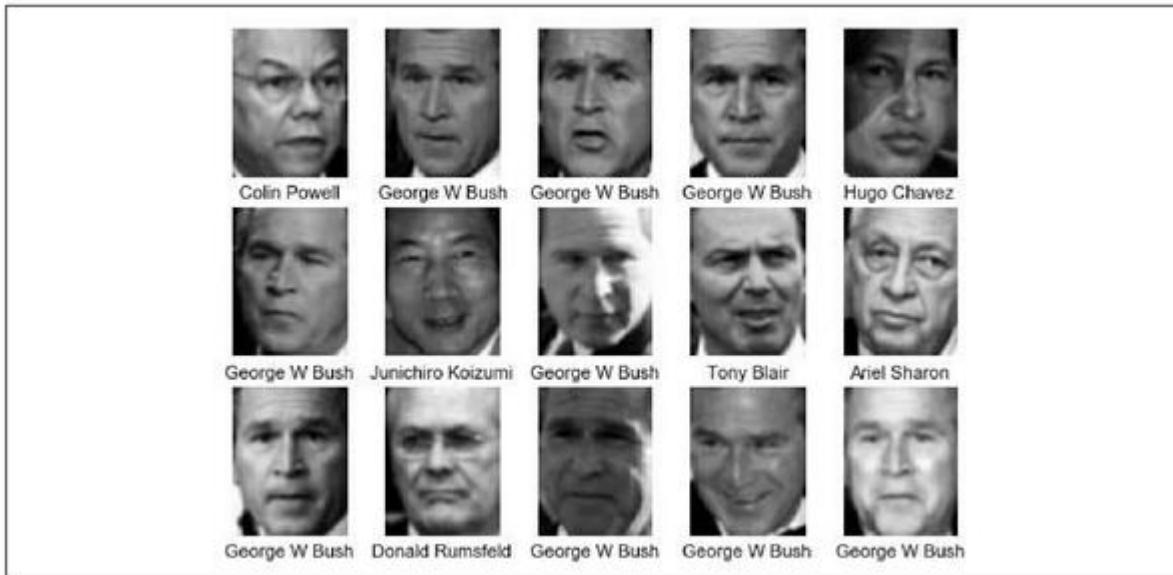
In[18]:

```
from sklearn.datasets import fetch_lfw_people
faces =
fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld'
'George W Bush'
'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro
Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Demandons l'affichage de quelques visages pour voir sur quoi nous allons travailler ([Figure 5.64](#)) :

In[19]:

```
fig, ax = plt.subplots(3, 5)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]]))
```



[Figure 5.64](#) : Un extrait du fichier des visages de personnalités politiques.

Chaque image est au format [62 x 47], soit environ 3 000 pixels. Nous pourrions travailler à la hussarde en décidant que chaque pixel correspond à une caractéristique, mais il est souvent plus efficace d'utiliser un préprocesseur pour n'extraire que quelques caractéristiques significatives. Ici, nous allons utiliser une analyse par composantes principales PCA (nous la décrivons dans la section ultérieure portant ce nom). Nous allons ainsi

extraire 150 composantes fondamentales que nous allons injecter dans le classifieur de notre machine SVM. Pour simplifier le travail, nous enchaînons l'exécution du préprocesseur et celle du classifieur dans un pipeline :

In[20]:

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA as RandomizedPCA
from sklearn.pipeline import make_pipeline

pca = RandomizedPCA(n_components=150,
whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')
model = make_pipeline(pca, svc)
```

Pour tester les résultats, nous répartissons les données en un jeu d'entraînement et un jeu de test :

In[21]:

```
from sklearn.model_selection import
train_test_split
Xtrain, Xtest, ytrain, ytest =
train_test_split(faces.data, faces.target,
random_state=42)
```

Nous appliquons alors une validation croisée à grille de recherche pour pouvoir explorer différentes valeurs de

paramètre. Nous allons ajuster le paramètre C qui contrôle la douceur de la marge ainsi que le paramètre gamma qui contrôle la taille du noyau de la fonction de base radiale. Nous déterminons ainsi le meilleur modèle :

```
In[22]:  
from sklearn.model_selection import  
GridSearchCV  
param_grid = {'svc__C': [1, 5, 10, 50],  
              'svc__gamma': [0.0001, 0.0005,  
0.001, 0.005]}  
grid = GridSearchCV(model, param_grid)  
  
%time grid.fit(Xtrain, ytrain)  
print(grid.best_params_)  
  
CPU times: user 47.8 s, sys: 4.08 s, total: 51.8  
s  
Wall time: 26 s  
{'svc__gamma': 0.001, 'svc__C': 10}
```

Les valeurs optimales se situent vers le milieu de la grille. Si elles étaient situées sur les bords, nous aurions étendu la grille pour être certains que nous avons vraiment trouvé le modèle optimal.

Une fois cette validation croisée réalisée, nous pouvons demander la prédiction des labels pour les données de test, celles que le modèle n'a pas encore rencontrées :

```
In[23]:
```

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

Affichons quelques visages avec les valeurs prédites pour vérifier ([Figure 5.65](#)) :

```
In[24]:
```

```
fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
    axi.imshow(Xtest[i].reshape(62, 47),
cmap='bone')
    axi.set(xticks=[], yticks=[])

    axi.set_ylabel(faces.target_names[yfit[i]].split(
)[-1],
                color='black' if yfit[i] ==
ytest[i] else 'red')

fig.suptitle('Noms prédits; les incorrects en
rouge', size=14);
```

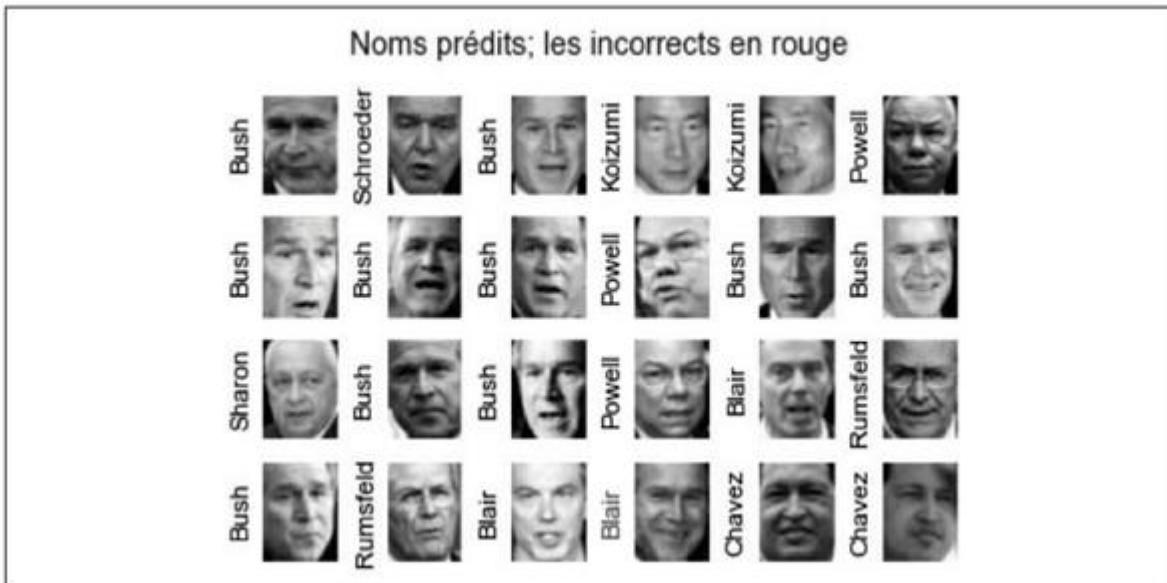


Figure 5.65 : Les labels tels que prédits par notre modèle.

Dans ce petit échantillon, l'estimateur optimisé ne s'est trompé qu'une fois (le second visage de Bush dans la rangée du bas a été confondu avec celui de Blair). Pour une idée plus précise des performances de notre estimateur, nous pouvons nous servir du rapport de classification qui fournit des statistiques label par label :

In[25]:

```
from sklearn.metrics import  
classification_report  
print(classification_report(ytest, yfit,
```

```
target_names=faces.target_names))
```

Ariel Sharon	0.65	0.73	0.69
15			
Colin Powell	0.81	0.87	0.84
68			
Donald Rumsfeld	0.75	0.87	0.81
31			
George W Bush	0.93	0.83	0.88
126			
Gerhard Schroeder	0.86	0.78	0.82
23			
Hugo Chavez	0.93	0.70	0.80
20			
Junichiro Koizumi	0.80	1.00	0.89
12			
Tony Blair	0.83	0.93	0.88
42			
avg / total	0.85	0.85	0.85
337			

Nous pouvons enfin demander l'affichage de la matrice de confusion entre ces classes ([Figure 5.66](#)) :

In[26]:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, yfit)
sns.heatmap(mat.T, square=True, annot=True,
fmt='d', cbar=False,
          xticklabels=faces.target_names,
          yticklabels=faces.target_names)
```

```

plt.xlabel('true label')
plt.ylabel('predicted label');

```

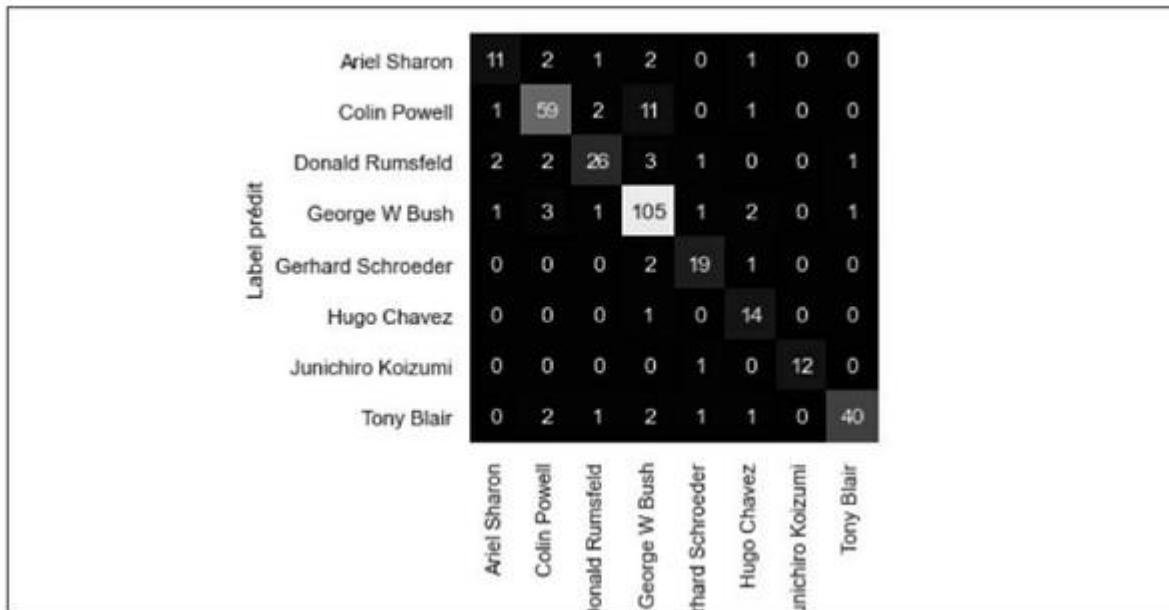


Figure 5.66 : Matrice de confusion pour les données des visages.

Cette matrice permet de se faire une idée des visages qui risquent d'être confondus par l'estimateur.

Dans un projet de reconnaissance de visages du monde réel, les photos ne sont généralement pas joliment préparées, mais la seule différence dans la classification correspond à la sélection des caractéristiques. Il faudrait recourir à un algorithme plus complexe pour trouver les visages puis extraire les caractéristiques indépendantes du bruit dans les pixels. Une option intéressante consiste à adopter OpenCV (<http://opencv.org>) qui parmi d'autres choses comporte des implémentations préentraînées de plusieurs outils

d'extraction de caractéristiques à la pointe du progrès destinés aux images en général, et aux visages en particulier.

Synthèse des machines SVM

Nous venons de traverser une introduction aux grands principes des machines SVM, que j'espère assez intuitive. Ce sont des méthodes de classification puissantes pour plusieurs raisons :

- Ces méthodes ont besoin d'un nombre de vecteurs de support relativement faible, ce qui en fait des modèles compacts qui occupent peu d'espace mémoire.
- Une fois que le modèle a été entraîné, la phase de prédiction est très rapide.
- Les modèles ne sont impactés que par les points proches de la zone de marge, et c'est pourquoi ils supportent bien des données à grand nombre de dimensions, même un nombre de dimensions supérieur au nombre d'échantillons, situation qui pose un vrai défi aux autres algorithmes.
- Leur intégration avec les méthodes de noyaux les rend très polyvalentes, et donc capables de s'adapter à de nombreux genres de données.

En revanche, les machines SVM souffrent de plusieurs inconvénients :

- Pour un nombre d'échantillons N , l'accroissement de la charge de traitement s'établit à $O[N^3]$ au pire, ou bien $O[N^2]$ si l'implémentation est efficace. La charge de traitement peut devenir décourageante pour un grand nombre d'échantillons d'entraînement.
- Les résultats dépendent fortement du choix adéquat du paramètre d'adoucissement C . Son choix doit être confirmé par une validation croisée, qui est elle-même longue pour un jeu de données volumineux.
- Les résultats ne peuvent pas être interprétés directement en termes de probabilités. Ils peuvent être estimés par une validation croisée interne (ce qui est en rapport avec le paramètre `probability` de SVC), mais cette estimation complémentaire est coûteuse elle-même.

En gardant ces différents critères à l'esprit, je n'opte généralement pour les machines SVM qu'après avoir vérifié que les méthodes moins lourdes et plus rapides se sont montrées insuffisantes pour répondre à mes besoins. Cela dit, si vous disposez d'une puissance de CPU suffisante pour lancer une machine SVM en entraînement et en validation

croisée sur vos données, vous pouvez espérer d'excellents résultats.

5.8 : Arbres de décision et forêts aléatoires

Nous venons d'explorer deux algorithmes : un classifieur génératif simple (le bayésien naïf) puis un classifieur discriminant (la machine à vecteurs de support). Découvrons maintenant un autre algorithme puissant, cette fois-ci non paramétrique, qui correspond à la *forêt aléatoire*. Il s'agit d'une méthode d'ensemble, qui se fonde donc sur l'agrégation des résultats de tout un ensemble d'estimateurs simples qui sont des *arbres de décision*. Certains sont étonnés lors de leur premier contact de constater que dans ce cas la somme est supérieure à ses parties. En effet, un vote majoritaire avec un certain nombre d'estimateurs est plus performant que les estimateurs pris individuellement ! Nous verrons cela par un exemple dans la suite. Commençons par les opérations d'import habituelles :

In[1] :

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Des forêts aléatoires d'arbres de décision

La forêt aléatoire est un exemple d'apprentissage par ensemble fondé sur des arbres de décision. Revenons donc d'abord sur ce qu'est un arbre de décision.

Un arbre de décision est une technique très intuitive qui permet de classer ou d'identifier des objets : il suffit de poser une série de questions de plus en plus précises jusqu'à avoir classé tous les éléments. Imaginez que vous faites une randonnée et que vous tombiez nez à nez avec un animal inconnu. Pour l'identifier, vous pourriez construire l'arbre montré dans la [Figure 5.67](#).

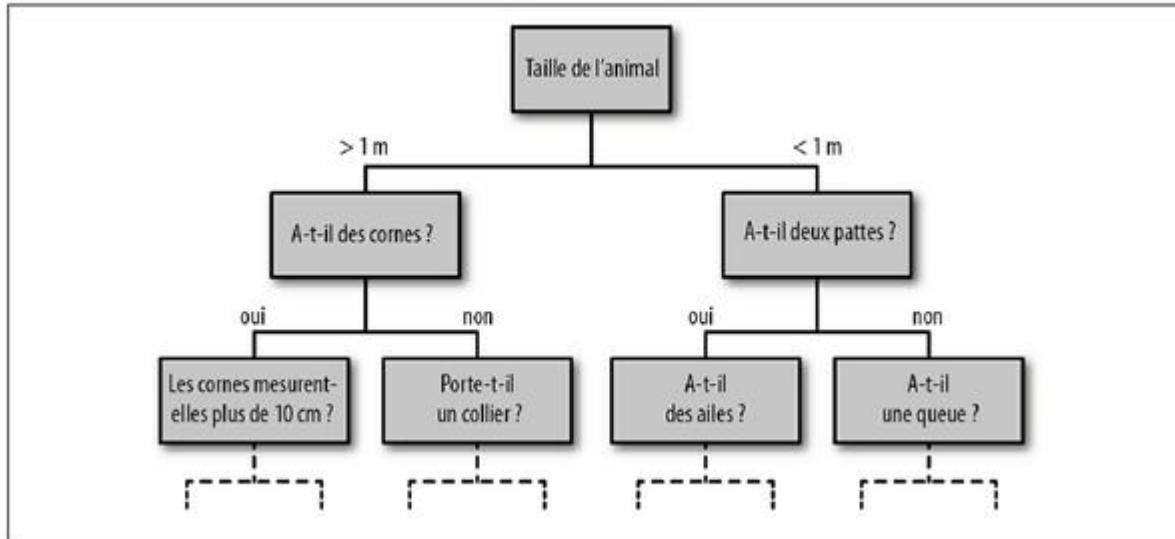


Figure 5.67 : Exemple d'arbre de décision binaire.

La répartition binaire rend l'opération très efficace. Si l'arbre a été bien construit, chaque réponse divise par deux le nombre de questions ouvertes, ce qui permet d'arriver vite à un classement complet, même en partant d'un grand nombre au départ. Toute l'astuce consiste à choisir quelle question poser à chaque étape. Dans les arbres de décision de l'apprentissage machine, les questions correspondent à des divisions alignées selon des axes parmi les données. Chaque nœud de l'arbre répartit les données en deux groupes en se basant sur une valeur critique de l'une des caractéristiques. Passons à un exemple.

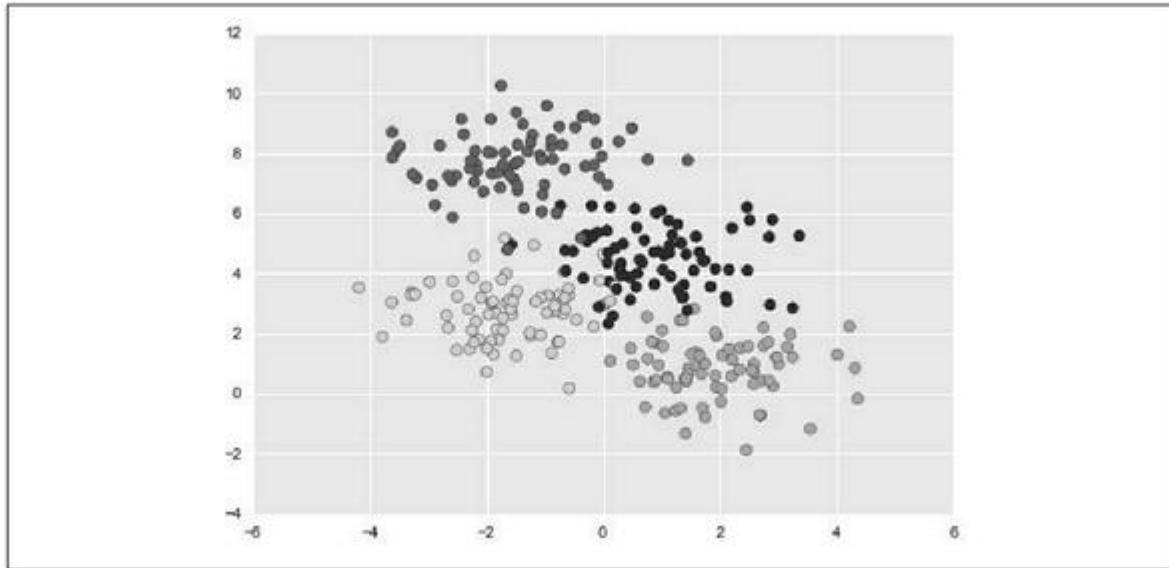
Création d'un arbre de décision

Préparons des données d'entrée en deux dimensions avec quatre labels de classes ([Figure 5.68](#)) :

In[2]:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4,
                   random_state=0,
                   cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50,
            cmap='rainbow');
```



[Figure 5.68](#) : Données d'entrée du classifieur à arbre de décision.

Si nous produisons un arbre de décision simple avec ces données, celles-ci vont être réparties selon un axe ou un autre en fonction d'un critère quantitatif. À chaque niveau dans l'arbre, la nouvelle région créée recevra comme label le résultat d'un vote majoritaire parmi les points qu'elle délimite. La [Figure 5.69](#) montre les quatre premières étapes d'un classifieur à arbre de décision avec ses données.

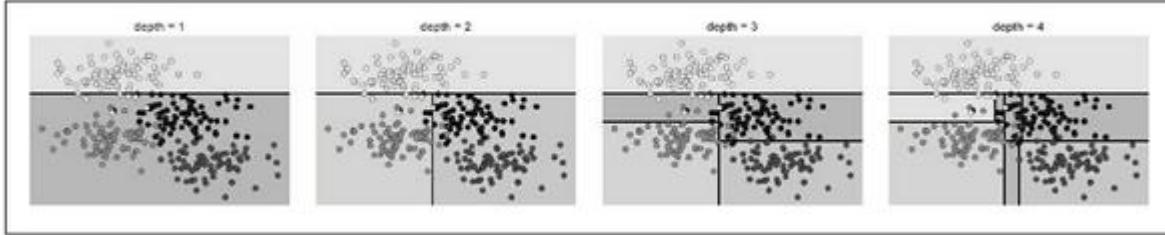


Figure 5.69 : Visualisation des étapes de répartition des données dans l'arbre de décision.

On remarque que la région supérieure est délimitée dès la première étape et ne change plus ensuite. En revanche, les régions qui comportent plusieurs catégories sont à nouveau divisées en deux à chaque étape selon l'une des deux caractéristiques.

Dans Scikit-Learn, nous demandons l'ajustement d'un arbre de décision à nos données au moyen de l'estimateur nommé `DecisionTreeClassifier` :

In[3] :

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier().fit(X, y)
```

Pour faciliter la visualisation des résultats du classifieur, nous définissons une petite fonction utilitaire :

In[4] :

```
def visualize_classifier(model, X, y, ax=None,
cmap='rainbow'):
    ax = ax or plt.gca()
```

```

# Trace les points d'entraînement
ax.scatter(X[:, 0], X[:, 1], c=y, s=30,
cmap=cmap,
           clim=(y.min(), y.max()), zorder=3)
ax.axis('tight')
ax.axis('off')
xlim = ax.get_xlim()
ylim = ax.get_ylim()

# Ajuste l'estimateur
model.fit(X, y)
xx, yy = np.meshgrid(np.linspace(*xlim,
num=200),
                     np.linspace(*ylim,
num=200))
Z = model.predict(np.c_[xx.ravel(),
yy.ravel()]).reshape(xx.shape)

n_classes = len(np.unique(y))
contours = ax.contourf(xx, yy, Z, alpha=0.3,
levels=np.arange(n_classes + 1) - 0.5,
cmap=cmap, clim=
(y.min(), y.max()), zorder=1)

ax.set(xlim=xlim, ylim=ylim)

```

Voyons maintenant ce que la classification par l'arbre de décision nous a apporté ([Figure 5.70](#)) :

In[5]:

```
visualize_classifier(DecisionTreeClassifier(),  
X, y)
```

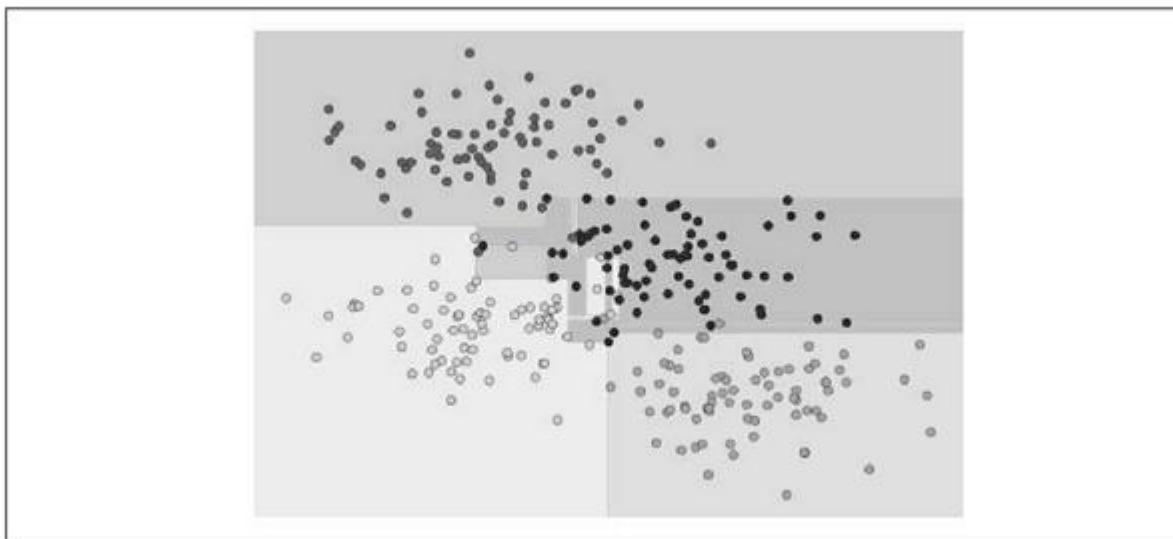


Figure 5.70 : Visualisation d'une classification par arbre de décision.

Si vous utilisez la version interactive du calepin, vous pouvez vous servir du script de soutien fourni avec le fichier d'archives (et sur le site <https://github.com/jakevdp/PythonDataScienceHandbook/notebooks>) pour obtenir une visualisation interactive du processus de construction de l'arbre (Figure 5.71) :

In[6]:

```
# helpers_05_08 se trouve dans l'annexe anglaise  
en ligne  
#  
(https://github.com/jakevdp/PythonDataScienceHandbook/notebooks)
```

```
import helpers_05_08  
helpers_@5_@8.plot_tree_interactive(X, y);
```

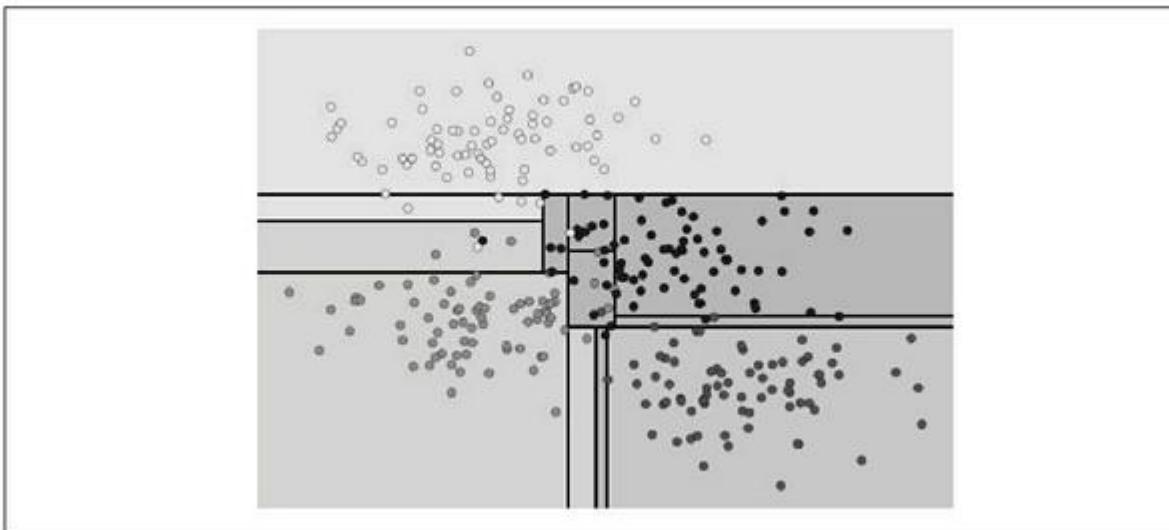


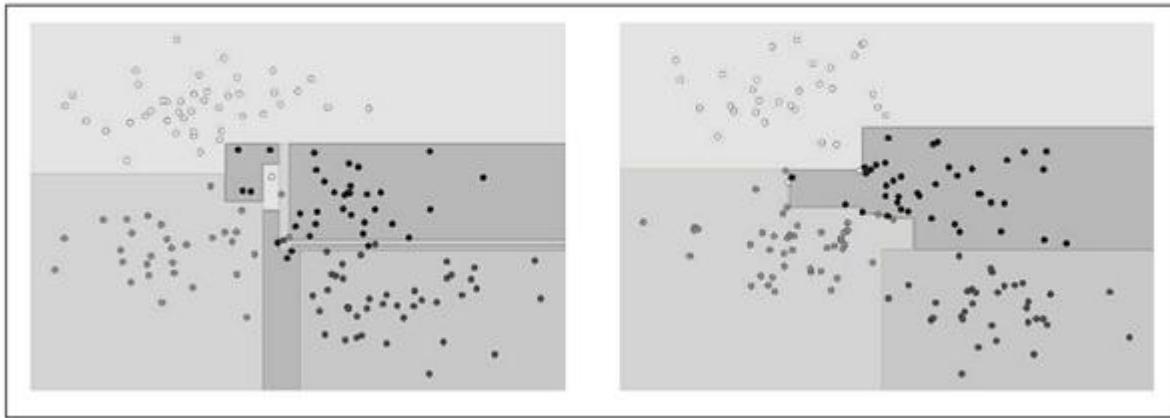
Figure 5.71: Première étape de l'outil de visualisation de l'arbre.

Vous remarquez que plus l'on progresse dans l'arbre, plus on découvre de régions étranges. À la cinquième étape apparaît une région très oblongue en bas au milieu. Il est probable que cela ne soit pas le résultat d'une bonne distribution de données, mais plus l'effet de l'échantillonnage ou de bruit dans les données. Autrement dit, au bout de cinq niveaux, notre arbre semble clairement surajusté.

Arbres de décision et surajustement

Le surajustement est un caractère habituel des arbres de décision ; on descend trop facilement dans les détails, ce qui provoque un ajustement à ces détails, en perdant de vue les

propriétés générales des distributions dont ils sont issus. Le phénomène devient encore plus évident lorsque nous demandons de traiter des sous-ensembles différents de données. Dans la [Figure 5.72](#), nous avons entraîné deux arbres différents, chacun avec la moitié des données de départ.



[Figure 5.72](#) : Exemple de deux arbres de décision sur des données aléatoires.

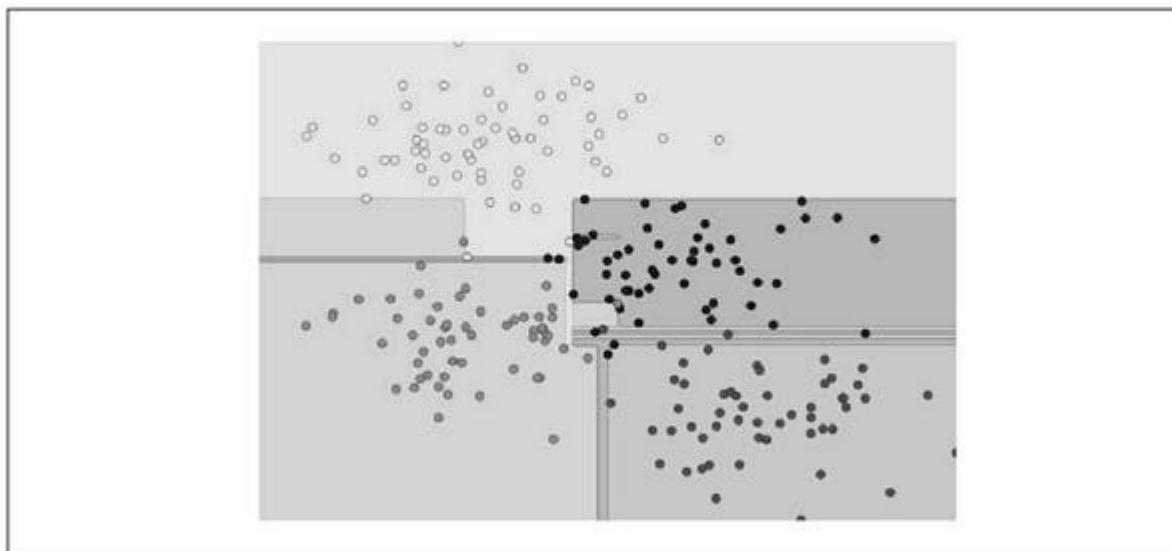
Dans certaines zones, les deux arbres aboutissent au même résultat, par exemple dans les quatre angles. Ailleurs, les classifications divergent fortement, par exemple dans toutes les régions entre deux grappes. On remarque que les incohérences se situent aux endroits où la classification est incertaine. C'est pourquoi on peut obtenir un meilleur résultat en combinant ces deux arbres.

Si vous utilisez la version interactive, la fonction suivante permet de reproduire ces opérations d'ajustement sur les sous-ensembles de données ([Figure 5.73](#)) :

In[7]:

```
# Voir la note plus haut pour helpers_05_08  
import helpers_05_08  
helpers_05_08.randomized_tree_interactive(x, y)
```

Puisque nous pouvons améliorer les résultats en combinant les données de deux arbres, nous supposons qu'il sera encore meilleur si nous combinons les informations d'un plus grand nombre d'arbres.



[Figure 5.73](#) : Première étape de l'outil d'arbres de décision aléatoires interactif.

Forêts aléatoires et classification

La technique consistant à combiner plusieurs estimateurs qui sont en surajustement pour diminuer les effets du surajustement est à la base d'une méthode d'ensemble qui porte l'étrange nom de *bagging*. Cette opération consiste à

agréger plusieurs estimateurs parallèles, qui ont tous subi un surajustement, afin de produire une moyenne qui va donner une meilleure classification. Lorsqu'un ensemble d'arbres de décision aléatoires est ainsi utilisé, il en résulte une *forêt aléatoire*.



(N.d.T.) *Bagging* n'a rien à voir avec des sacs, bags, car c'est la contraction de *bootstrap aggregating* ou agrégation autonome.

Pour obtenir dans Scikit-Learn ce genre de classification manuelle par bagging, nous nous servons du métal-estimateur nommé BaggingClassifier ([Figure 5.74](#)) :

In[8] :

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier()
bag = BaggingClassifier(tree, n_estimators=100,
max_samples=0.8,
random_state=1)

bag.fit(X, y)
visualize_classifier(bag, X, y)
```

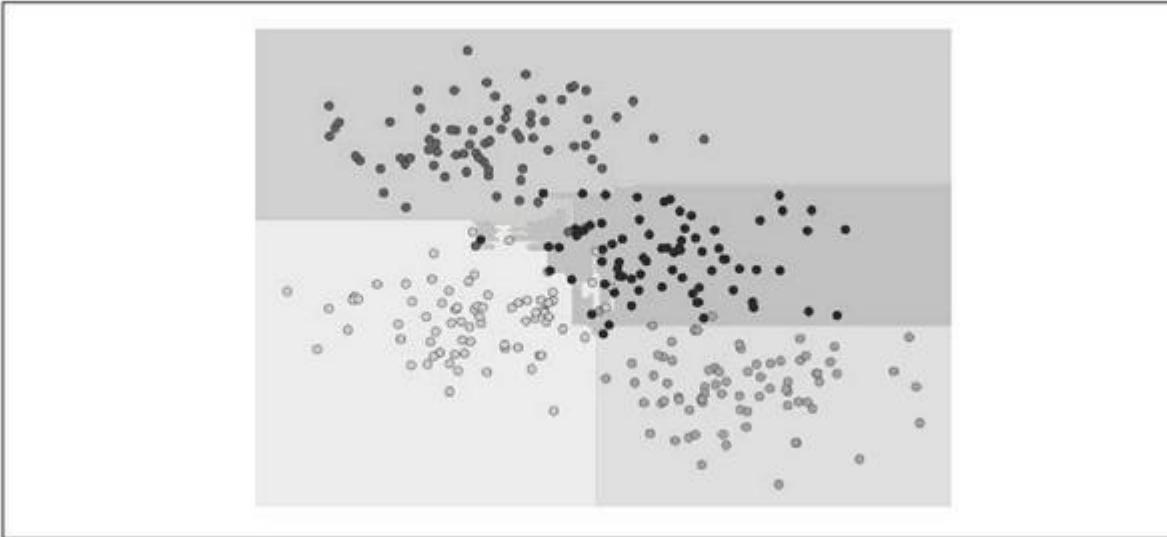


Figure 5.74 : Frontières de décision pour un ensemble d'arbres de décision aléatoires.

Pour rendre les données de l'exemple aléatoires, nous avons ajusté chaque estimateur avec 80 % des points d'entraînement choisis au hasard. La sélection aléatoire des données est plus efficace lorsqu'on injecte un peu de stochasticité (de hasard), afin que toutes les données contribuent à l'ajustement à chaque fois, sans que les résultats de l'ajustement perdent leur nature aléatoire recherchée. Par exemple, au moment de choisir quelle caractéristique utiliser pour les distributions, l'arbre aléatoire pourra choisir une caractéristique parmi les principales. La documentation de Scikit-Learn donne des détails techniques à propos de ces stratégies de sélection aléatoire (<http://scikit-learn.org/stable/modules/ensemble.html#forest>).

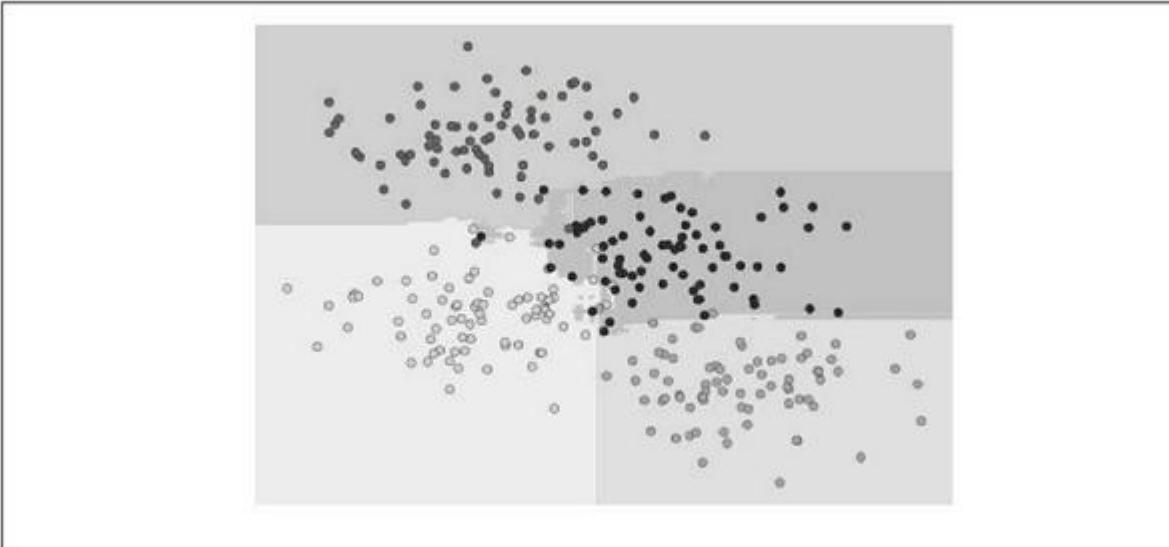
Dans Scikit-Learn, vous disposez d'un ensemble optimisé d'arbres de décision aléatoires avec l'estimateur nommé *RandomForestClassifier*. Il se charge automatiquement de tout le travail de préparation aléatoire. Il vous suffit de choisir un nombre d'estimateurs. Vous obtenez très vite, éventuellement en parallèle, l'ajustement sur l'ensemble des arbres ([Figure 5.75](#)) :

In[9] :

```
from sklearn.ensemble import
RandomForestClassifier

model = RandomForestClassifier(n_estimators=100,
random_state=0)
visualize_classifier(model, X, y);
```

Nous constatons qu'en demandant la moyenne à partir de 100 modèles bien mélangés, nous obtenons un modèle global bien plus proche de nos attentes concernant la distribution de l'espace paramétrique.



[Figure 5.75](#) : Frontières de décision pour une forêt aléatoire provenant d'un ensemble d'arbres optimisés.

Forêts aléatoires et régression

Nous venons de voir comment utiliser une forêt aléatoire en classification, mais elle est également utilisable en régression, c'est-à-dire avec des variables continues. Dans ce cas, l'estimateur porte le nom RandomForestRegressor, sa syntaxe restant très proche de celle de son collègue.

Partons des données suivantes qui sont le fruit de la combinaison d'une oscillation lente et d'une rapide ([Figure 5.76](#)) :

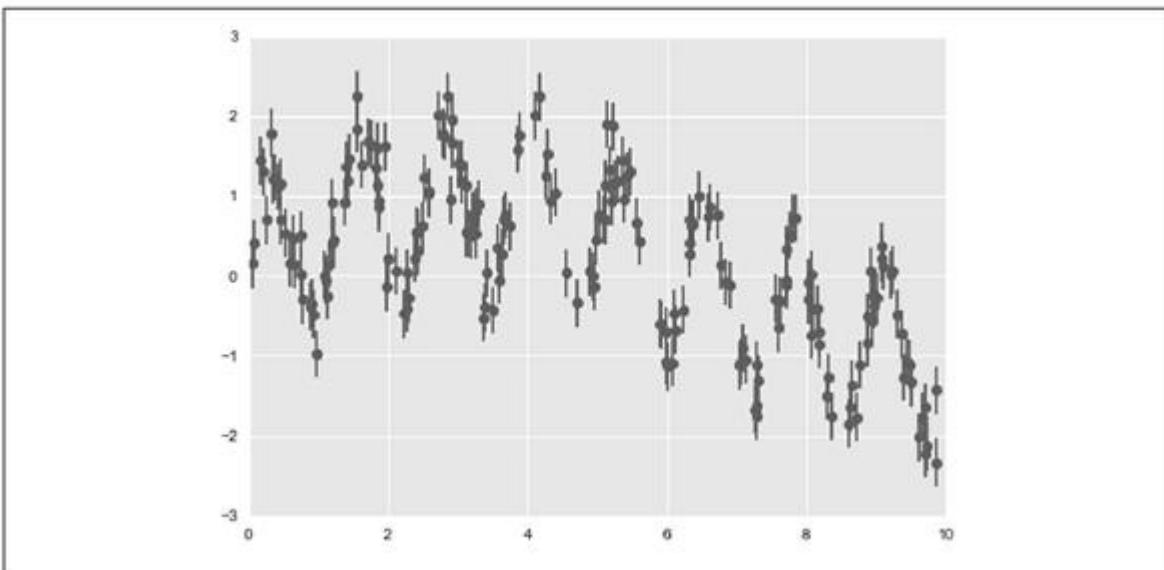
```
In[10]:  
rng = np.random.RandomState(42)  
x = 10 * rng.rand(200)
```

```

def model(x, sigma=0.3):
    fast_oscillation = np.sin(5 * x)
    slow_oscillation = np.sin(0.5 * x)
    noise = sigma * rng.randn(len(x))
    return slow_oscillation + fast_oscillation +
noise

y = model(x)
plt.errorbar(x, y, 0.3, fmt='o');

```



[Figure 5.76](#) : Données initiales d'une régression de forêts aléatoires.

Munis de notre régresseur, nous obtenons la courbe la mieux ajustée ainsi ([Figure 5.77](#)) :

```

In[11]:
from sklearn.ensemble import
RandomForestRegressor

```

```
forest = RandomForestRegressor(200)
forest.fit(x[:, None], y)

xfit = np.linspace(0, 10, 1000)
yfit = forest.predict(xfit[:, None])
ytrue = model(xfit, sigma=0)

plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
plt.plot(xfit, yfit, '-r');
plt.plot(xfit, ytrue, '-k', alpha=0.5);
```

Le modèle correspond à la courbe lissée alors que le modèle de forêts aléatoires correspond à la courbe hachurée. Vous constatez que le modèle de forêts aléatoires non paramétrique sait s'adapter à des données sur plusieurs périodes, sans que nous ayons besoin de préciser un modèle multipériode !

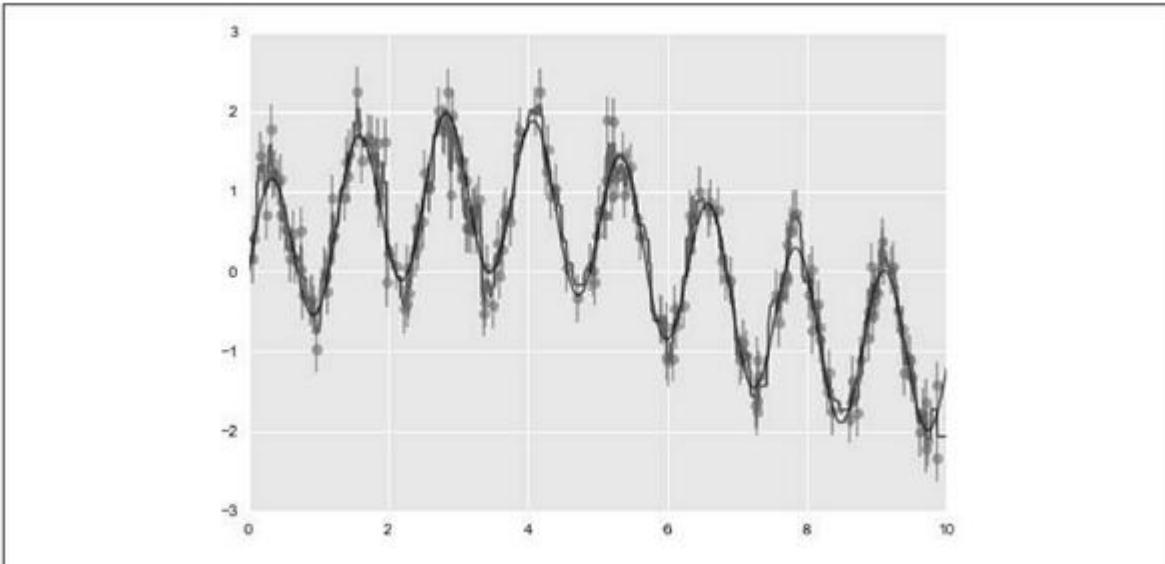


Figure 5.77 : Modèle de forêts aléatoires ajusté à des données.

Exemple : forêt aléatoire pour classer des chiffres

Nous avons déjà utilisé le jeu de données des chiffres manuscrits en début de chapitre. Voyons ce que peut en tirer un classifieur de style forêts aléatoires.

In[12]:

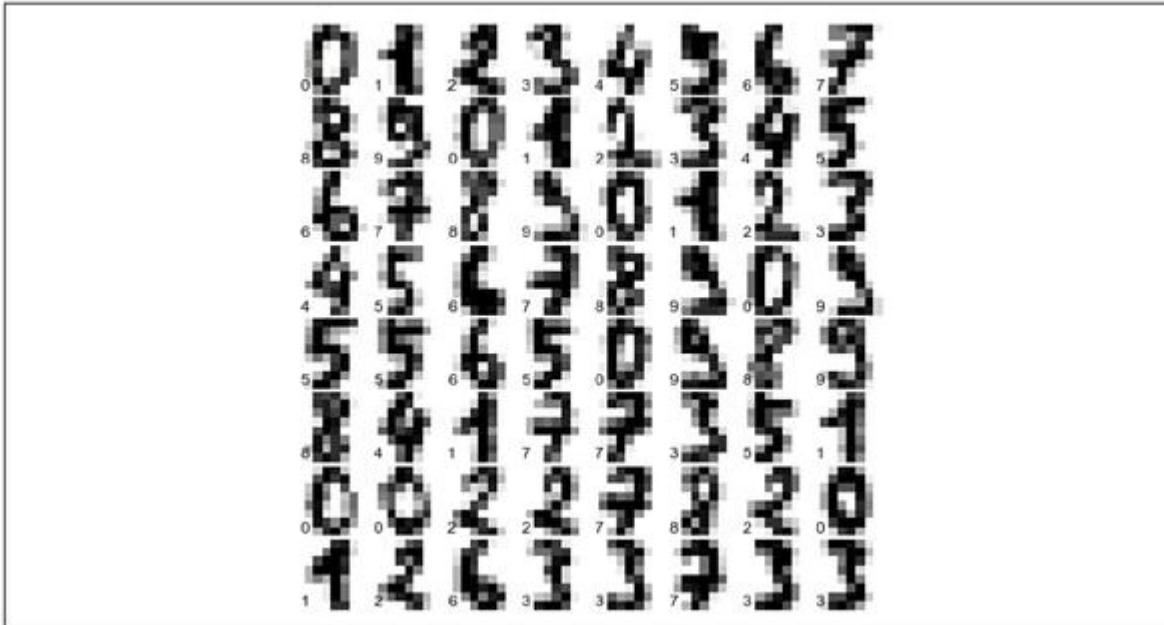
```
from sklearn.datasets import load_digits  
digits = load_digits()  
digits.keys()
```

Out[12]:

```
dict_keys(['target', 'data', 'target_names',  
'DESCR', 'images'])
```

Visualisons les premiers points de données en guise de rappel de ce sur quoi nous travaillons ([Figure 5.78](#)) :

```
In[13]:  
# Préparation de la figure  
fig = plt.figure(figsize=(6, 6)) # Taille de  
l'image en pouces  
fig.subplots_adjust(left=@, right=1, bottom=@,  
top=1,  
    hspace=@.@5, wspace=@.@5)  
  
# Tracé des chiffres : chaque image est sur 8 x  
8 pixels  
for i in range(64):  
    ax = fig.add_subplot(8, 8, i + 1, xticks=[],  
yticks=[])  
    ax.imshow(digits.images[i],  
cmap=plt.cm.binary, interpolation='nearest')  
  
    # Label d'image avec la valeur cible  
ax.text(@, 7, str(digits.target[i]))
```



[Figure 5.78](#) : Rappel des données de chiffres manuscrits.

L’application d’une forêt aléatoire sur ces chiffres est très rapide ([Figure 5.79](#)) :

In[14]:

```
from sklearn.model_selection import  
train_test_split  
  
Xtrain, Xtest, ytrain, ytest =  
train_test_split(digits.data, digits.target,  
random_state=0)  
  
model =  
RandomForestClassifier(n_estimators=1000)  
model.fit(Xtrain, ytrain)  
ypred = model.predict(Xtest)
```

Voyons le rapport de classification qui a été produit :

In[15]:

```
from sklearn import metrics
print(metrics.classification_report(ypred,
ytest))
```

	precision	recall	f1-score	support
0	1.00	0.97	0.99	38
1	1.00	0.98	0.99	44
2	0.95	1.00	0.98	42
3	0.98	0.96	0.97	46
4	0.97	1.00	0.99	37
5	0.98	0.96	0.97	49
6	1.00	1.00	1.00	52
7	1.00	0.96	0.98	50
8	0.94	0.98	0.96	46
9	0.96	0.98	0.97	46
avg / total	0.98	0.98	0.98	450

Demandons également de produire la matrice de confusion ([Figure 5.79](#)) :

In[16]:

```
from sklearn.metrics import confusion_matrix

mat = confusion_matrix(ytest, ypred)
sns.heatmap(mat.T, square=True, annot=True,
fmt='d', cbar=False)
```

```

plt.xlabel('true label')
plt.ylabel('predicted label');

```

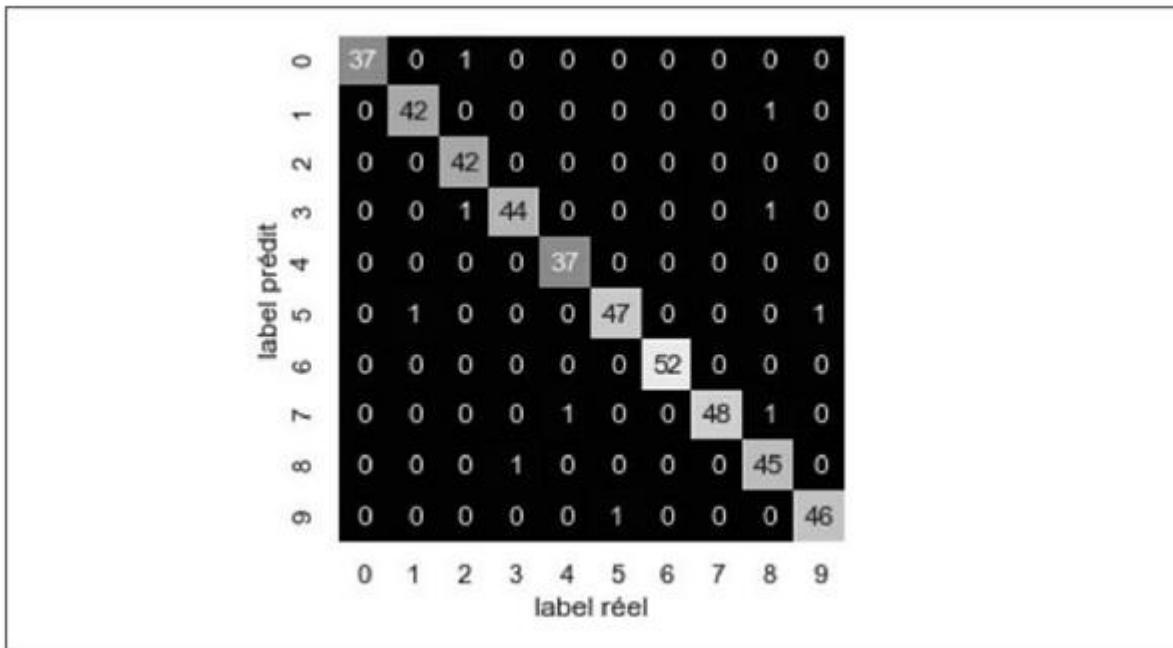


Figure 5.79 : Matrice de confusion pour classification de chiffres avec des forêts aléatoires.

Nous constatons que nous obtenons une classification vraiment précise des chiffres manuscrits avec une forêt aléatoire simple et même pas optimisée.

Conclusion sur les forêts aléatoires

Nous venons de découvrir et de pratiquer le concept d'estimateur d'ensembles, en particulier le modèle de forêts aléatoires qui fusionne des arbres de décision aléatoires.

Voici quelques avantages des forêts aléatoires qui expliquent leur puissance :

- L'entraînement et la prédiction se déroulent très vite parce que les arbres de décision sont simples. De plus, on peut aisément paralléliser les tâches parce que tous les arbres sont des entités indépendantes.
- En utilisant plusieurs arbres, on peut obtenir une classification probabiliste : le vote majoritaire parmi les différents estimateurs procure une estimation de la probabilité (vous y accédez dans Scikit-Learn avec la méthode `predict_proba()`).
- Les modèles non paramétriques sont très souples ; ils se comportent correctement dans des tâches provoquant un sous-ajustement avec d'autres estimateurs.

L'inconvénient principal des forêts aléatoires est la difficulté d'interprétation des résultats. Lorsque vous avez besoin de tirer des conclusions concernant la signification du modèle de classification, vous n'êtes pas dans les meilleures conditions avec une forêt aléatoire.

5.9 : Analyse par composantes principales (PCA)

Jusqu'ici, nous avons étudié les estimateurs d'apprentissage supervisés : ils prédisent des labels en se basant sur des données d'entraînement qui en possèdent. Voyons maintenant quelques estimateurs sans supervision. Ils permettent de mettre en lumière des aspects intéressants des données sans faire référence à aucun label connu.

Nous commençons par l'algorithme d'apprentissage non supervisé sans doute le plus répandu, l'analyse par composantes principales ou PCA (*Principal Component Analysis*). C'est d'abord un algorithme de réduction des dimensions, mais il peut également servir pour la visualisation, le débruitage, l'ingénierie et l'extraction des caractéristiques, parmi d'autres tâches. Après une brève présentation de l'algorithme PCA, nous allons enchaîner quelques exemples. Nous commençons par les imports habituels :

```
In[1]:  
%matplotlib inline  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()
```

Introduction à l'algorithme PCA

L'analyse par composantes principales est une méthode souple et rapide pour réduire le nombre de dimensions des données, comme nous l'avons dans l'introduction à Scikit-Learn en début de chapitre. Son fonctionnement est plus facile à visualiser en travaillant avec un jeu de données à deux dimensions. Partons des 200 points suivants ([Figure 5.80](#)) :

In[2] :

```
rng = np.random.RandomState(1)
X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal');
```

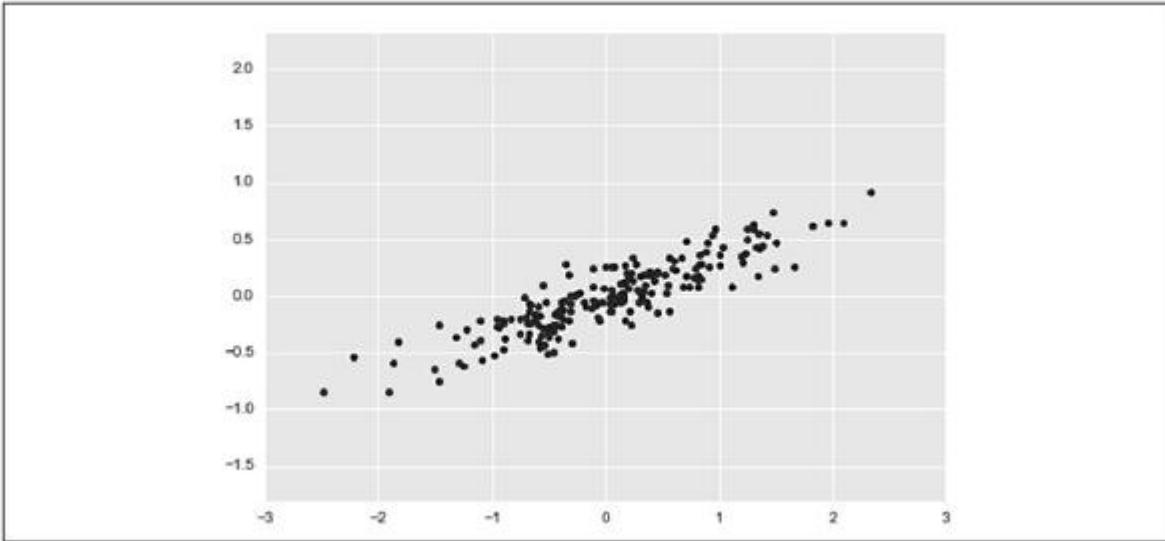


Figure 5.80 : Données d'entrée d'une démonstration de PCA.

La simple lecture des instructions permet de deviner qu'il y a une relation quasi linéaire entre les deux variables x et y . Cela rappelle les données de régression linéaire vues dans la section correspondante. Ici, ce que nous voulons faire est un peu différent : au lieu de chercher à prédire les valeurs y à partir des valeurs x , nous voulons découvrir quelles sont les *relations* entre x et y sans supervision.

L'analyse PCA cherche à quantifier la relation en cherchant la liste des axes principaux parmi les données puis en utilisant ces axes pour décrire le jeu de données. Pour réaliser l'opération, nous pouvons nous servir de l'estimateur nommé PCA de Scikit-Learn :

In[3] :

```
from sklearn.decomposition import PCA
```

```
pca = PCA(n_components=2)
pca.fit(X)
```

Out[3]:

```
PCA(copy=True, n_components=2, whiten=False)
```

L'opération d'ajustement récupère certaines quantités à partir des données, en particulier les *composantes* et la *variance expliquée* :

In[4]:

```
print(pca.components_)
```

```
[ [ 0.94446029  0.32862557]
  [ 0.32862557 -0.94446029] ]
```

In[5]:

```
print(pca.explained_variance_)
```

```
[ 0.75871884  0.01838551]
```

Pour comprendre ce que signifient ces valeurs, visualisons-les sous forme de vecteurs sur les données d'entrée. Nous utilisons les composantes pour définir la direction du vecteur et la variance expliquée pour définir la longueur du vecteur élevé au carré ([Figure 5.81](#)) :

In[6]:

```
def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca()
```

```

arrowprops=dict(arrowstyle='->',
               linewidth=2,
               shrinkA=0, shrinkB=0)
ax.annotate(' ', v1, v0,
            arrowprops=arrowprops)

plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in
zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');

```

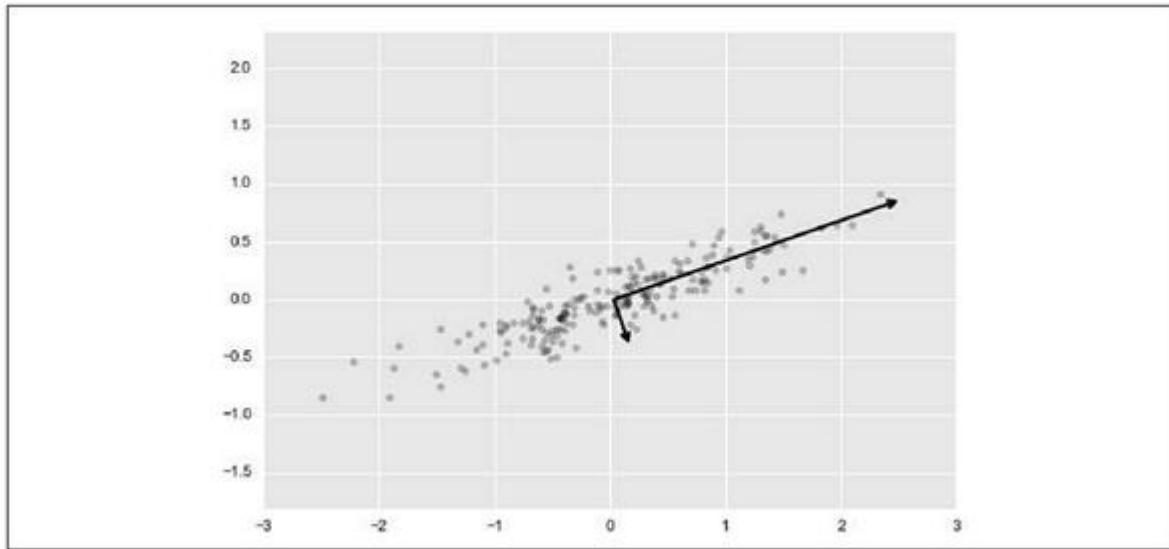
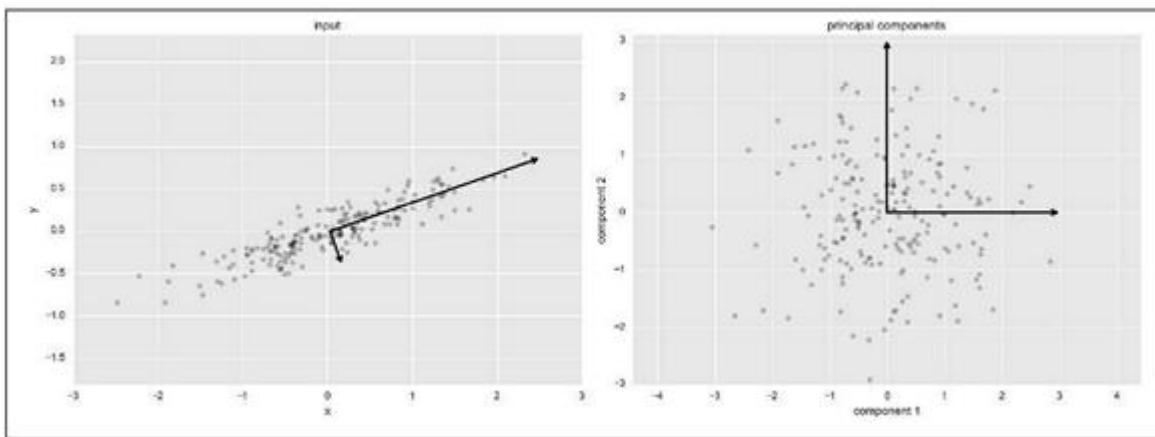


Figure 5.81 : Visualisation des axes principaux.

Ces deux vecteurs correspondent aux axes principaux. Leur longueur indique l'importance de l'axe dans la description de la distribution des données. C'est une mesure de la

variance des données dans la projection sur cet axe. La projection de chaque point de données sur les axes principaux constitue les « composantes principales » des données.

Nous pouvons afficher ces composantes principales pour obtenir les points montrés dans la [Figure 5.82](#).



[Figure 5.82](#) : Axes principaux transformés des données.

Cette transformation des axes de données en *axes principaux* est une transformation dite *affine*. Cela signifie une translation, une rotation et une remise à l'échelle uniforme.

Cet algorithme qui cherche des composantes principales pourrait ressembler à une simple curiosité mathématique. En réalité, son domaine d'application est très vaste en apprentissage machine et en exploration de données.

PCA pour la réduction dimensionnelle

Exploiter l'algorithme PCA pour réduire le nombre de dimensions suppose d'identifier une ou plusieurs des composantes principales les moins importantes pour les annuler. On obtient ainsi une projection avec moins de dimensions qui préserve la variance maximale des données.

Voici un exemple de transformation pour réduction dimensionnelle PCA :

In[7] :

```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X)
print("Forme originelle : ", X.shape)
print("Forme transformée: ", X_pca.shape)
```

Forme originelle : (200, 2)

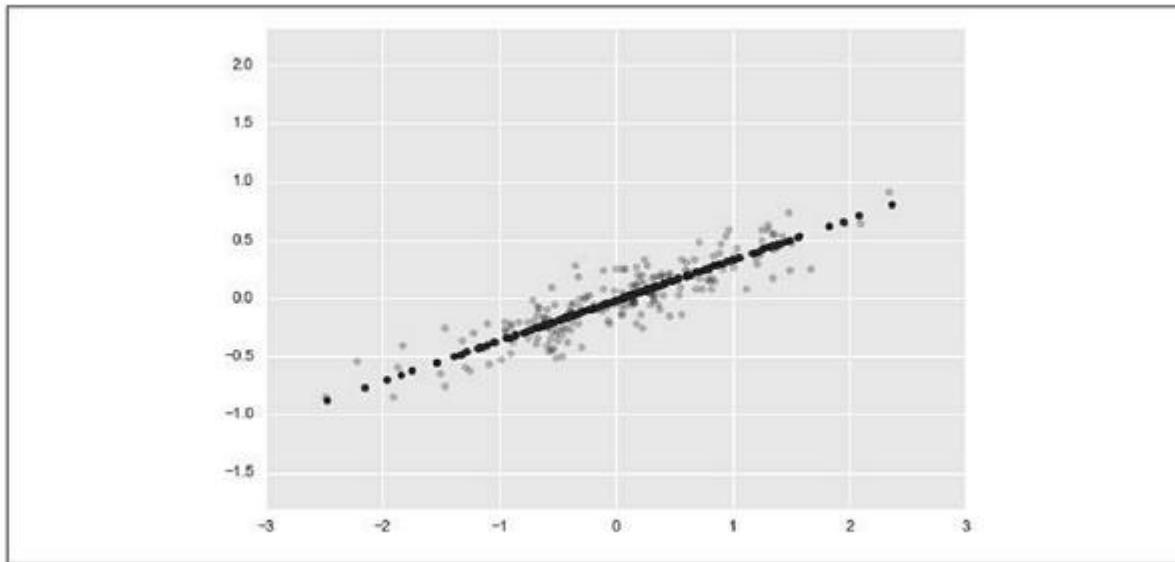
Forme transformée: (200, 1)

Le résultat ne conserve qu'une dimension. Pour juger de l'effet de cette réduction, nous demandons une transformation inverse des données réduites que nous visualisons en même temps que les données de départ ([Figure 5.83](#)) :

In[8] :

```
X_new = pca.inverse_transform(X_pca)
```

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
plt.axis('equal');
```



[Figure 5.83](#) : Visualisation d'une réduction dimensionnelle PCA.

Les points clairs sont les données de départ et les points foncés la version projetée. La figure permet immédiatement de comprendre ce que fait la réduction dimensionnelle PCA : les informations correspondant à l'axe principal le moins important ont été supprimées pour ne laisser que celles offrant la plus grande variance. La proportion de variance éliminée (proportionnelle à la dispersion des points par rapport à la ligne dans la [Figure 5.83](#)) revient grossièrement à mesurer le volume d'informations qui a été écarté dans l'opération.

Ce jeu de données réduit est d'une certaine façon suffisant pour contenir une description des relations essentielles entre les points : malgré une réduction de 50 % des dimensions, les relations globales entre les points de données sont pratiquement conservées.

PCA pour la visualisation : chiffres manuscrits

L'intérêt de la réduction dimensionnelle devient beaucoup plus évident lorsqu'il s'agit de travailler avec des données ayant un plus grand nombre de dimensions. Reprenons le jeu de données de chiffres manuscrits déjà rencontré et appliquons l'algorithme PCA.

Nous commençons par charger les données :

In[9] :

```
from sklearn.datasets import load_digits  
digits = load_digits()  
digits.data.shape
```

Out[9] :

```
(1797, 64)
```

Rappelons qu'il s'agit d'images de 8 pixels sur 8, donc avec 64 dimensions. Nous pouvons appliquer PCA pour projeter les données vers un nombre de dimensions inférieur afin de nous faire une première idée des relations entre les points de données. Nous choisissons deux dimensions :

```
In[10]:  
pca = PCA(2)          # Projection de 64 vers 2  
dimensions  
projected = pca.fit_transform(digits.data)  
print(digits.data.shape)  
print(projected.shape)  
  
(1797, 64)  
(1797, 2)
```

Nous pouvons immédiatement faire afficher les deux composantes principales pour chaque point afin d'en apprendre plus au sujet des données ([Figure 5.84](#)) :

```
In[11]:  
plt.scatter(projected[:, 0], projected[:, 1],  
           c=digits.target, edgecolor='none',  
           alpha=0.5,  
           cmap=plt.cm.get_cmap('nipy_spectral',  
           10))  
plt.xlabel('Composante 1')  
plt.ylabel('Composante 2')  
plt.colorbar();
```

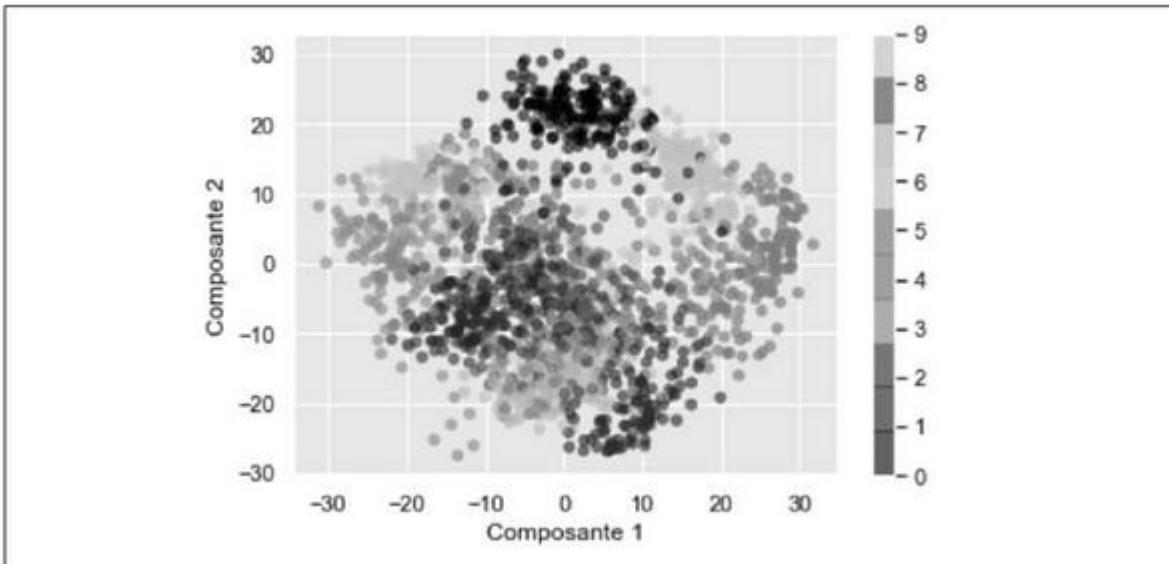


Figure 5.84 : La PCA appliquée aux chiffres manuscrits.

Rappelons ce que ces composantes nous disent : les données d'entrée forment un nuage de points à 64 dimensions, les points étant les projections des points de données orientés vers la plus grande variance. Nous avons ainsi trouvé l'étirement et la rotation les plus corrects dans un espace à 64 dimensions, ce qui permet de visualiser les chiffres en deux dimensions. L'opération a été réalisée sans supervision préalable, c'est-à-dire sans s'appuyer sur les labels.

Signification des composantes

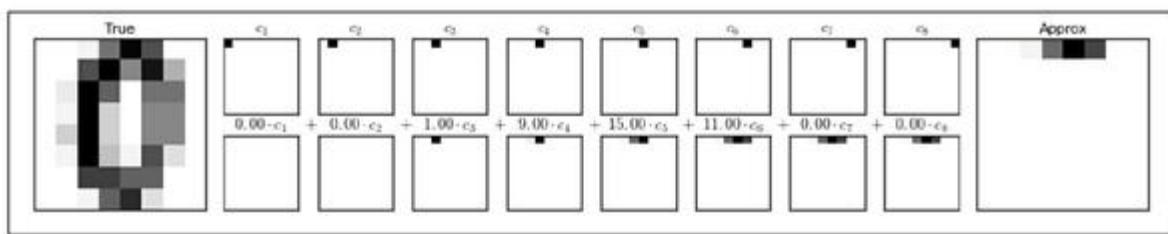
Allons plus loin et voyons ce que signifient les dimensions réduites. Nous pouvons les considérer comme des combinaisons de vecteurs de base. Chaque image du jeu d'entraînement est définie par un ensemble de 64 valeurs de pixels, que nous appelons le vecteur x :

$$x = [x_1, x_2, x_3, \dots, x_{64}]$$

On peut considérer cela comme une base de pixels. Pour construire l'image, nous multiplions chaque élément du vecteur par le pixel qu'il décrit puis additionnons les résultats pour obtenir l'image :

$$\text{image}(x) = x_1 \cdot (\text{pixel } 1) + x_2 \cdot (\text{pixel } 2) + x_3 \cdot (\text{pixel } 3) \dots x_{64} \cdot (\text{pixel } 64)$$

Une approche de réduction des dimensions consiste à éliminer la plupart des vecteurs de base sauf certains. Si nous ne conservons que les huit premiers pixels, il en résulte une projection à huit dimensions des données ([Figure 5.85](#)). On constate que le résultat rappelle très difficilement l'image de départ puisque nous avons jeté environ 90 % des pixels !



[Figure 5.85](#) : Une réduction dimensionnelle naïve par suppression de pixels.

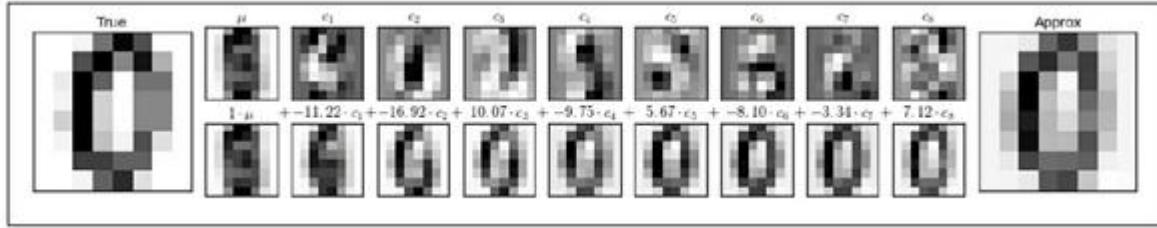
La rangée de cases supérieure montre les pixels individuels et la rangée inférieure la contribution cumulée à l'image finale. En n'utilisant que les huit premiers pixels, nous ne parvenons à construire qu'une petite portion de l'image

à 64 pixels. Il faudrait les utiliser tous pour retrouver l'image d'origine.

Mais ce n'est pas le seul choix de représentation possible. Avec d'autres fonctions de base, chacune contenant une contribution prédéfinie depuis chaque pixel, nous pouvons écrire ceci :

$$\text{image}(x) = \text{moyenne} + x_1 \cdot (\text{base } 1) + x_2 \cdot (\text{base } 2) + x_3 \cdot (\text{base } 3) \dots$$

L'approche PCA peut être vue comme une technique consistant à choisir la fonction de base optimale dans chaque cas, pour que l'addition de tous les résultats, en se limitant aux plus importants, permette de reconstruire grossièrement la plupart des éléments du jeu de données. Les composantes principales constituent la représentation des données dans un moins grand nombre de dimensions. Elles jouent le rôle de coefficients qui sont multipliés par chacun des éléments de la série. Dans la [Figure 5.86](#), nous reconstruisons un chiffre en nous limitant à la moyenne et aux huit premières des fonctions de base PCA.



[Figure 5.86](#) : Une réduction plus efficace possible en éliminant les composantes principales les moins importantes (comparez avec la Figure 5.85).

À la différence de l'approche par pixels, l'approche PCA permet de récupérer l'essentiel de la signification avec uniquement une moyenne et huit composantes ! La grandeur du pixel dans chaque composante est liée à l'orientation du vecteur dans l'exemple en deux dimensions. C'est de cette manière que PCA permet de construire une représentation des données réduite en dimensions : il s'agit de trouver un jeu de fonctions de base plus efficace que l'approche basée sur les pixels.

Choix du nombre de composantes

Pour bien tirer profit de PCA, il faut pouvoir estimer le nombre de composantes à prévoir pour bien décrire les données. Pour nous y aider, nous demandons à voir le ratio de *variance expliquée cumulative* en fonction du nombre de composantes ([Figure 5.87](#)) :

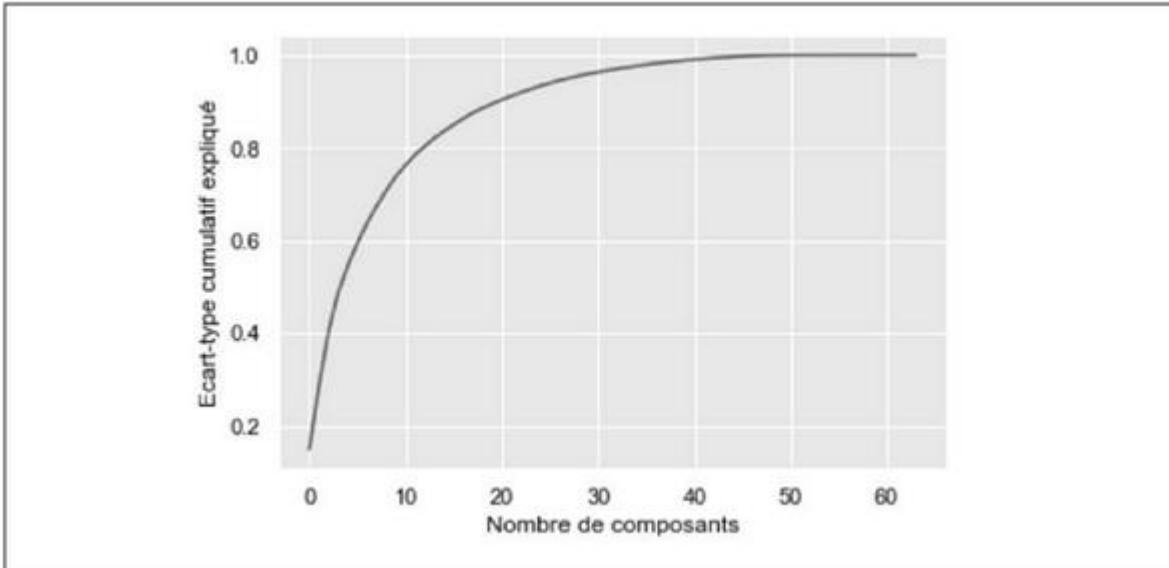
In[12]:

```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_)
```

```
)  
plt.xlabel('Nombre de composants')  
plt.ylabel('Variance cumulative expliquée');
```

Cette courbe montre quel pourcentage de la variance totale à 64 dimensions est préservé dans les N premières composantes. Nous voyons qu'avec les dix premières composantes en abscisse, nous préservons environ 75 % de la variance. Pour ne quasiment rien perdre de la signification, il faut monter jusqu'à 50 composantes.

Nous constatons que notre choix fait perdre beaucoup de signification et qu'il faudrait conserver au moins 20 composantes pour arriver à 90 % de variance. Ce genre de courbe établie pour un jeu de données à nombre de dimensions important vous donne une idée du niveau de redondance que l'on retrouve dans les observations multiples.



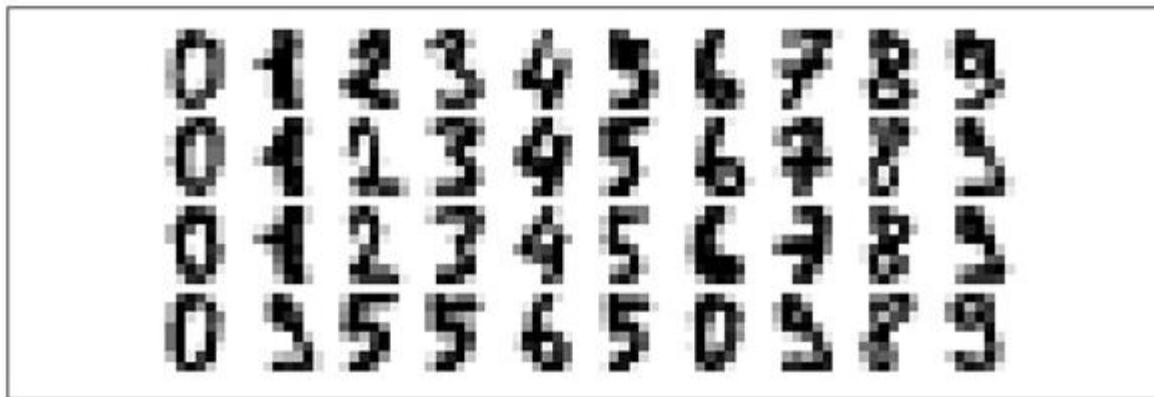
[Figure 5.87](#) : La variance expliquée cumulative permet de juger à quel point PCA préserve le contenu des données.

PCA et filtrage du bruit

L'algorithme PCA permet également d'assurer un filtrage pour éliminer du bruit dans les données. Le principe est le suivant : toute composante dont la variance est largement supérieure à l'impact du bruit doit normalement être peu concernée par ce bruit. Si l'on reconstruit les données en ne conservant qu'un bon sous-ensemble des composantes principales, on devrait préserver l'essentiel du signal et éliminer quasiment tout le bruit.

Appliquons ce raisonnement à nos chiffres manuscrits. Pour comparer, affichons une sélection des chiffres manuscrits non bruités ([Figure 5.88](#)) :

```
In[13]:  
def plot_digits(data):  
    fig, axes = plt.subplots(4, 10, figsize=(10,  
4),  
                           subplot_kw=  
                           {'xticks':[], 'yticks':[]},  
  
                           gridspec_kw=dict(hspace=0.1, wspace=0.1))  
    for i, ax in enumerate(axes.flat):  
        ax.imshow(data[i].reshape(8, 8),  
                   cmap='binary',  
                   interpolation='nearest',  
                   clim=(0, 16))  
plot_digits(digits.data)
```



[Figure 5.88](#) : Les chiffres manuscrits sans bruit.

Ajoutons maintenant un bruit aléatoire au jeu de données et réaffichons les chiffres ([Figure 5.89](#)) :

```
In[14]:  
np.random.seed(42)
```

```
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)
```



Figure 5.89 : Les mêmes chiffres après ajout d'un bruit aléatoire gaussien.

On constate clairement l'arrivée du bruit et des pixels parasites. Appliquons la technique PCA à ces données bruitées en demandant de préserver 50 % de la variance :

In[15]:
pca = PCA(0.50).fit(noisy)
pca.n_components_

Out[15]:
12

Dans notre cas, les 50 % correspondent à 12 composantes principales. Nous pouvons calculer les composantes puis demander la transformation inverse pour reconstruire les chiffres après filtrage ([Figure 5.90](#)) :

In[16]:

```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



[Figure 5.90](#) : Les chiffres débruités grâce à PCA.

Grâce à cette propriété de débruitage, PCA constitue une routine de sélection de caractéristiques très pratique. Vous pouvez ainsi éviter d'entraîner un classifieur sur des données à grand nombre de dimensions, en le faisant travailler sur une représentation après réduction dimensionnelle qui va automatiquement filtrer le bruit aléatoire des données d'entrée.

Exemple : visages propres (eigenfaces)

Dans la section précédente qui a présenté les machines à vecteurs de support, nous avions utilisé une projection PCA

pour sélectionner les caractéristiques dans une reconnaissance de visages. Revenons à cet exemple en entrant un peu plus dans les détails. Nous étions partis du jeu de données d'exemple des visages de personnalités politiques LFW (*Labeled Faces in the Wild*) qui est disponible dans Scikit-Learn :

In[17]:

```
from sklearn.datasets import fetch_lfw_people
faces =
fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)

['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld'
 'George W Bush' 'Gerhard Schroeder' 'Hugo
 Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Voyons quels sont les axes principaux de ces données. Le jeu étant volumineux, nous utilisons la méthode RandomizedPCA() qui possède une méthode aléatoire pour trouver les N composantes principales initiales automatiquement et bien plus vite que l'estimateur standard PCA. Cette méthode est particulièrement utile pour les données à grand nombre de dimensions ; dans l'exemple, il y en a presque 3 000. Voyons les 150 premières composantes :

In[18]:

```
from sklearn.decomposition import RandomizedPCA  
pca = RandomizedPCA(150)  
pca.fit(faces.data)
```

Out[18]:

```
RandomizedPCA(copy=True, iterated_power=3,  
n_components=150,  
random_state=None, whiten=False)
```

Demandons à visualiser les images qui correspondent aux sept premières composantes principales. Notez que ces composantes sont techniquement des vecteurs propres (eigenvectors) et c'est pourquoi on parle de visages propres (ce « propre » n'a rien à voir avec l'hygiène, mais avec la propriété). La [Figure 5.91](#) nous laisse avec une galerie de zombies :

In[19]:

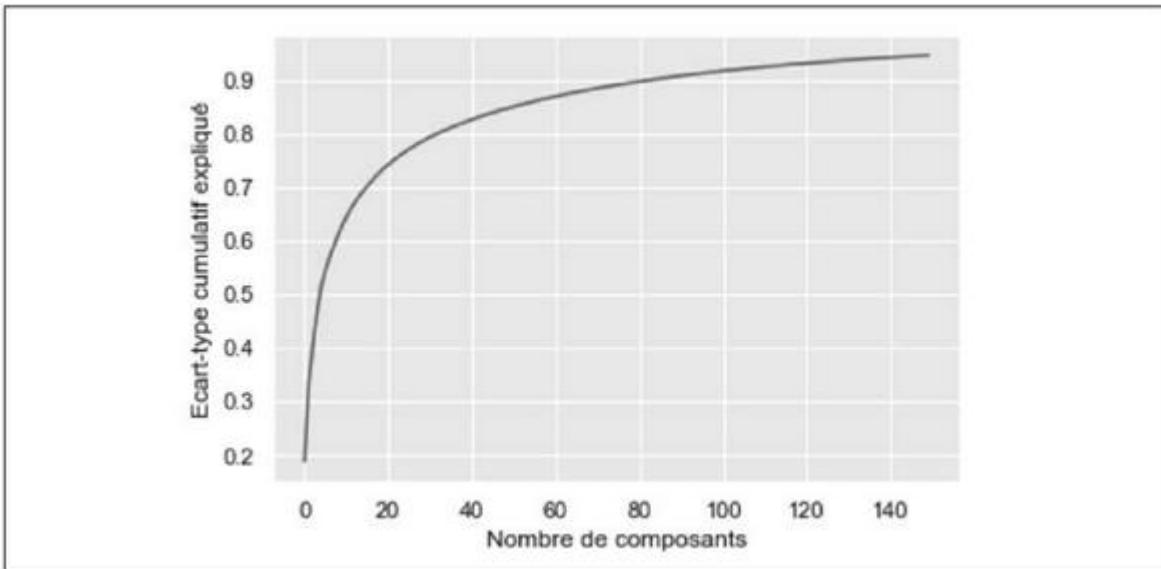
```
fig, axes = plt.subplots(3, 8, figsize=(9, 4),  
                        subplot_kw={'xticks':[],  
                                     'yticks':[]},  
  
gridspec_kw=dict(hspace=0.1, wspace=0.1))  
for i, ax in enumerate(axes.flat):  
    ax.imshow(pca.components_[i].reshape(62, 47),  
              cmap='bone')
```



[Figure 5.91](#) : Visualisation des images traitées à partir du jeu LFW.

Ces visages de zombies sont très intéressants néanmoins : ils nous aident à voir en quoi les images varient les unes par rapport aux autres : les huit premiers visages, en partant du haut à gauche, semblent avoir comme point commun l'angle d'éclairage, et les suivants s'intéressent chacun à une caractéristique en particulier (les yeux, les nez, les lèvres). Demandons à voir la courbe de variance cumulative de ces composantes pour estimer combien nous avons conservé d'information dans la projection ([Figure 5.92](#)) :

```
In[20]:  
plt.plot(np.cumsum(pca.explained_variance_ratio_))  
plt.xlabel('Nombre de composants')  
plt.ylabel('Variance expliquée cumulative');
```



[Figure 5.92](#) : Variance expliquée cumulative pour les données de visage.

Nous constatons que nous sommes à 90 % de la variance avec 150 composantes. Nous pouvons en déduire que nous devrions pouvoir récupérer l'essentiel des caractéristiques en nous servant de ces 150 composantes. Pour le vérifier, nous pouvons demander de comparer les images d'entrée avec les images reconstruites à partir des 150 composantes ([Figure 5.93](#)):

In[21]:

```
pca = RandomizedPCA(150).fit(faces.data)
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)
```

In[22]:

```
fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                      subplot_kw={'xticks':[],
```

```

'yticks':[]},

gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(10):
    ax[0, i].imshow(faces.data[i].reshape(62,
47), cmap='binary_r')
    ax[1, i].imshow(projected[i].reshape(62, 47),
cmap='binary_r')

ax[0, 0].set_ylabel('full-dim\nninput')
ax[1, 0].set_ylabel('150-dim\nreconstruction');

```

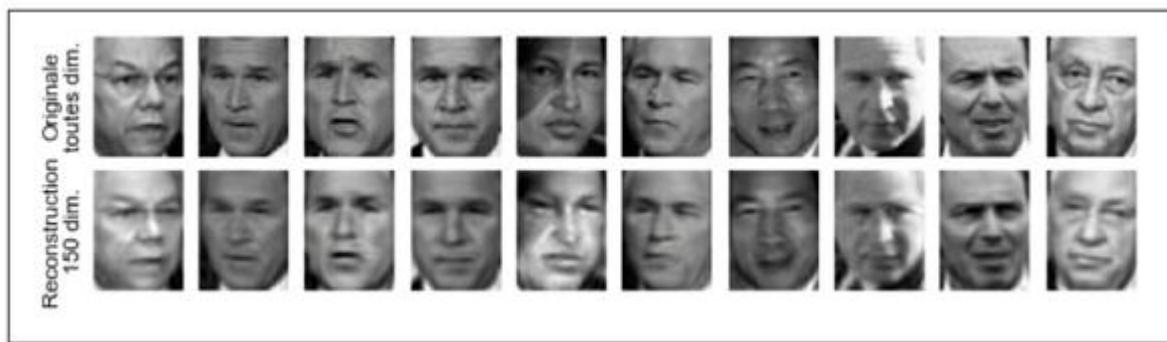


Figure 5.93 : Reconstruction des visages en PCA avec 150 dimensions.

La rangée du haut correspond aux images d'entrée et celle du bas aux traitements avec 150 dimensions parmi les 3 000 au départ. Vous comprenez pourquoi l'utilisation de PCA dans l'exemple sur les machines à vecteurs de support a si bien réussi : le nombre de dimensions est divisé par 20, pourtant les images résultantes ont préservé assez d'information pour que l'on puisse même à l'œil nu reconnaître les visages. Nous avons ainsi appris qu'il

suffisait entraîner l'algorithme de classification sur 150 dimensions et non 3 000. En choisissant le bon algorithme, nous obtenons une classification beaucoup plus efficace.

Synthèse de PCA

Nous venons de voir comment utiliser une analyse PCA pour réduire les dimensions, pour visualiser des données à dimensions nombreuses, pour filtrer du bruit et pour choisir des caractéristiques dans des données à dimensions nombreuses. La polyvalence et la lisibilité de PCA rendent cette technique très efficace dans de nombreux contextes et domaines. Lorsque je dois travailler avec un jeu de données à dimensions nombreuses, je commence en général par PCA pour me faire une idée des relations entre les points de données, ce que nous avons fait avec les chiffres, afin de comprendre où se situent les variances principales (ce que nous avons fait avec les visages) et enfin pour comprendre les dimensionnalités intrinsèques (en affichant la courbe de variance expliquée). PCA ne conviendra pas à tous les jeux de données à dimensions nombreuses, mais cela constitue une approche simple et efficace pour en apprendre plus au sujet de ces données.

La grande faiblesse de PCA est sa fragilité par rapport aux données aberrantes, c'est-à-dire les valeurs très différentes

des autres. Plusieurs variantes de PCA ont été produites afin de réduire ce défaut ; la plupart suppriment de façon itérative des points de données dès qu'ils sont mal décrits par les premières composantes principales. Scikit-Learn offre plusieurs variantes de PCA, et notamment RandomizedPCA et SparsePCA qui font partie du sous-module `sklearn.decomposition`. La première que nous avons utilisée propose une méthode non déterministe pour trouver rapidement les quelques premières composantes principales dans un jeu de données à dimensions nombreuses. Quant à SparsePCA, elle ajoute un terme de régularisation qui force la dissémination des composantes (*sparsity*). (Voyez aussi la description de la régularisation Lasso dans la section de ce chapitre qui présente la régression linéaire.)

Passons maintenant à d'autres méthodes d'apprentissage non supervisé qui réutilisent certaines des idées de PCA.

5.10 : Apprentissage par variété (manifold)

Nous avons vu comment réduire le nombre de caractéristiques d'un jeu de données tout en conservant l'essentiel des relations entre les points par la méthode PCA. C'est un outil souple, rapide et facile à interpréter, mais PCA montre ses limites lorsqu'il y a des relations *non linéaires* dans les données, et nous allons en voir quelques exemples.

Pour répondre à ce besoin, nous nous tournons vers une autre classe de méthodes, celle d'*apprentissage par variétés ou manifold*. Il s'agit d'une classe d'estimateurs non supervisés dont l'objectif est de décrire un jeu de données sous forme d'une variété au sens mathématique à faible nombre de dimensions incorporé dans un espace ayant plus de dimensions. Si le terme de variété ne vous dit rien, imaginez une feuille de papier : c'est un objet en deux dimensions (si on néglige son épaisseur), objet qui se situe dans notre espace habituel à trois dimensions. Vous pouvez plier, froisser ou rouler la feuille. Une fois ainsi transformée, elle s'inscrit dans l'espace ayant plus de dimensions, sous forme d'une variété du plan qu'elle est lorsqu'elle est bien plate.

Si l'on fait tourner, ou si l'on pouvait étirer la feuille de papier dans l'espace en trois dimensions tout en la

conservant bien plate, la géométrie de la feuille ne changerait pas : ces transformations sont dites linéaires. En revanche, si vous pliez, tordez ou froissez la feuille, bien qu'elle reste une variété à deux dimensions, elle s'inscrit dans l'espace à trois dimensions, donc de façon non linéaire. Un algorithme d'apprentissage par variétés cherche à découvrir la nature fondamentale à deux dimensions du papier, même s'il faut partir d'une structure pliée ou froissée dans un espace à trois dimensions.

Nous allons passer en revue plusieurs méthodes par variété, en nous intéressant plus en détail à certaines : le positionnement multidimensionnel *MDS* (*Multidimensional Scaling*), l'inscription linéaire locale *LLE* (*Locally Linear Embedding*) et le mappage isométrique (*Isomap*). Commençons par nos instructions d'import habituelles :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()  
import numpy as np
```

Apprentissage par variété : « SALUT »

Ces nouveaux concepts méritent d'être éclaircis. Commençons par préparer un nouveau jeu de données en deux dimensions qui va nous servir à définir notre variété. Voici la fonction qui va créer les données sous la forme du mot « SALUT » :

```
In[2]:  
def make_hello(N=1000, rseed=42):  
    # Trace un graphique avec le mot et le sauve  
    en PNG  
    fig, ax = plt.subplots(figsize=(4, 1))  
    fig.subplots_adjust(left=0, right=1,  
    bottom=0, top=1)  
    ax.axis('off')  
    ax.text(0.5, 0.4, 'SALUT', va='center',  
    ha='center',  
            weight='bold', size=85)  
    fig.savefig('salut.png')  
    plt.close(fig)  
  
    # Ouvre le PNG et affiche des points au  
    hasard  
    from matplotlib.image import imread  
    data = imread('salut.png')[::-1, :, 0].T  
    rng = np.random.RandomState(rseed)  
    X = rng.rand(4 * N, 2)
```

```

i, j = (X * data.shape).astype(int).T
mask = (data[i, j] < 1)
X = X[mask]
X[:, 0] *= (data.shape[0] / data.shape[1])
X = X[:N]
return X[np.argsort(X[:, 0])]

```

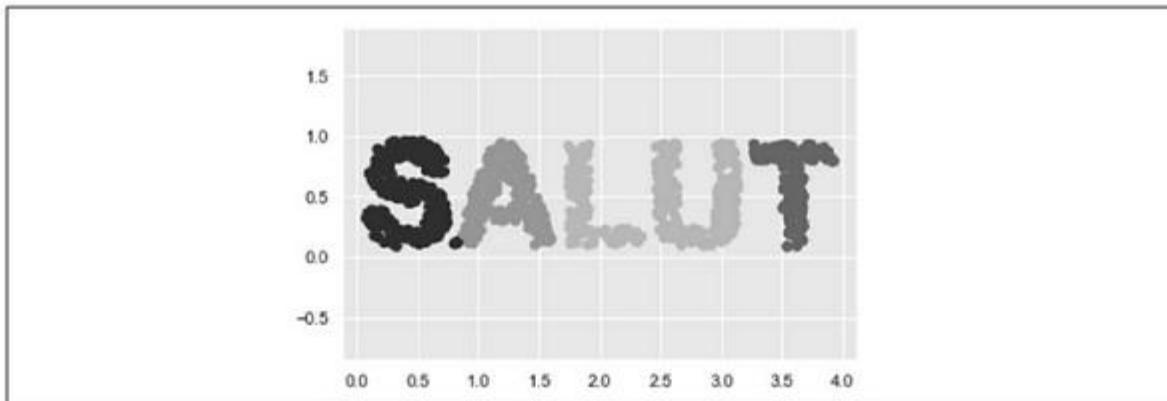
Essayons immédiatement notre fonction pour voir ce qu'il en ressort ([Figure 5.94](#)) :

In[3] :

```

X = make_hello(1000)
colorize = dict(c=X[:, 0],
cmap=plt.cm.get_cmap('rainbow', 5))
plt.scatter(X[:, 0], X[:, 1], **colorize)
plt.axis('equal');

```



[Figure 5.94](#) : Les données d'entrée de l'apprentissage par variété.



(N.d.T) Le terme anglais correspondant à *variété* est *manifold* qui signifie « nombreux plis », ce qui correspond bien à

l'idée d'une surface froissée donc d'un passage de deux à trois dimensions.

Notre résultat est en deux dimensions. Ce sont des points disséminés selon les formes des lettres du mot. Grâce à cette forme, nous allons pouvoir mieux juger de l'impact des algorithmes.

Partitionnement multidimensionnel (MDS)

En observant ces données d'entrée, nous devinons que le choix des valeurs pour x et y ne correspond pas à la description la plus fondamentale que l'on puisse trouver pour ces données. En effet, nous pouvons rééchelonner, déformer ou faire tourner les données, sans que le mot « SALUT » devienne illisible. Voyons par exemple comment appliquer une matrice de rotation, ce qui va modifier les valeurs x et y sans modifier essentiellement les données ([Figure 5.95](#)) :

In[4] :

```
def rotate(X, angle):
    theta = np.deg2rad(angle)
    R = [[np.cos(theta), np.sin(theta)],
          [-np.sin(theta), np.cos(theta)]]
    return np.dot(X, R)
```

```

x2 = rotate(X, 20) + 5
plt.scatter(x2[:, 0], x2[:, 1], **colorize)
plt.axis('equal');

```

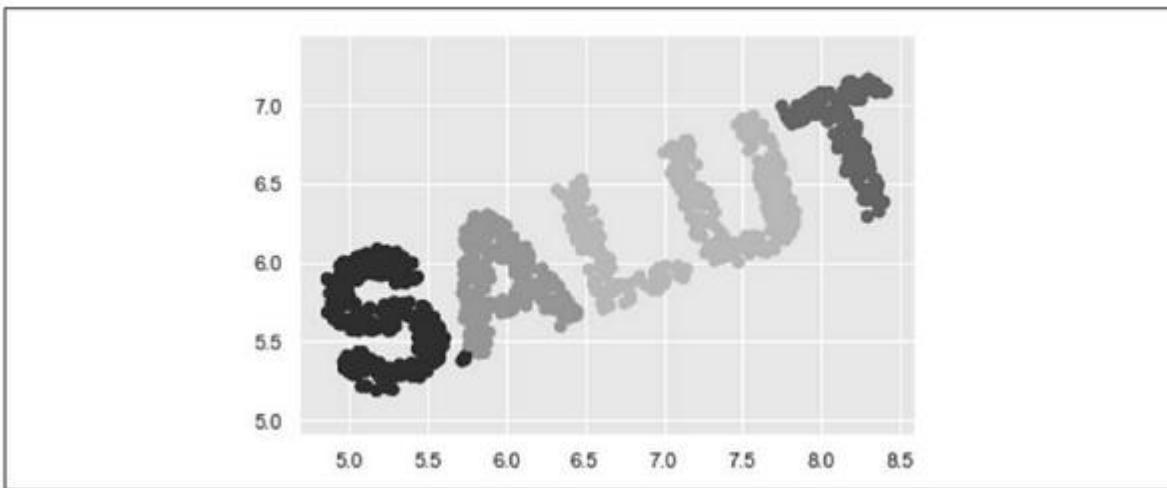


Figure 5.95 : Le jeu de données d'entrée après rotation.

Autrement dit, les valeurs x et y ne sont pas des éléments fondamentaux des relations parmi les données. Ce qui est fondamental dans l'exemple est la distance entre chaque point et tous les autres points du jeu. Pour le confirmer visuellement, nous pouvons demander une matrice de distances : avec N points, nous construisons un tableau de N par N dans lequel chaque entrée (i, j) contient la distance entre le point i et le point j. Profitons de la fonction efficace dont dispose Scikit-Learn qui se nomme `pairwise_distances()` pour traiter nos données de cette façon :

In[5]:

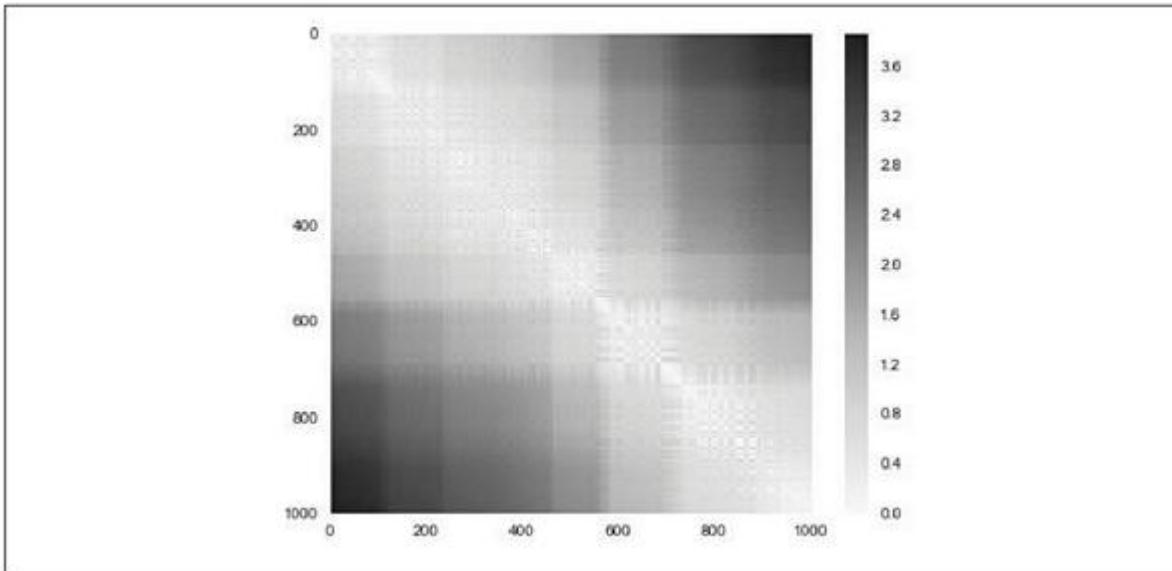
```
from sklearn.metrics import pairwise_distances
D = pairwise_distances(X)
D.shape
```

Out[5]: (1000, 1000)

Pour nos 1 000 points, nous obtenons bien une matrice de $1\ 000 \times 1\ 000$ que nous pouvons immédiatement visualiser ([Figure 5.96](#)) :

In[6]:

```
plt.imshow(D, zorder=2, cmap='Blues',
           interpolation='nearest')
plt.colorbar();
```



[Figure 5.96](#) : Visualisation des distances des paires de points.

Si nous construisons une autre matrice de distances pour le jeu de données après rotation et translation, nous arrivons au même résultat :

```
In[7]:  
D2 = pairwise_distances(X2)  
np.allclose(D, D2)
```

```
Out[7]: True
```

Nous obtenons ainsi une représentation des données qui est insensible aux rotations et translations. Il faut cependant avouer que la visualisation n'est pas facile à interpréter. Dans la [Figure 5.96](#), nous avons perdu tout indice des lettres du mot.

On peut facilement obtenir ce genre de matrice de distances à partir des coordonnées (x, y), mais l'opération inverse est assez difficile. Mais c'est exactement ce que permet notre algorithme de positionnement multidimensionnel : à partir d'une matrice de distances, il permet d'obtenir une représentation des coordonnées en D dimensions. Testons cet algorithme MDS dans notre exemple en spécifiant en paramètre la dissimilarité precomputed afin d'informer que nous transmettons une matrice de distances ([Figure 5.97](#)) :

In[8] :

```
from sklearn.manifold import MDS
model = MDS(n_components=2,
dissimilarity='precomputed', random_state=1)
out = model.fit_transform(D)
plt.scatter(out[:, 0], out[:, 1], **colorize)
plt.axis('equal');
```

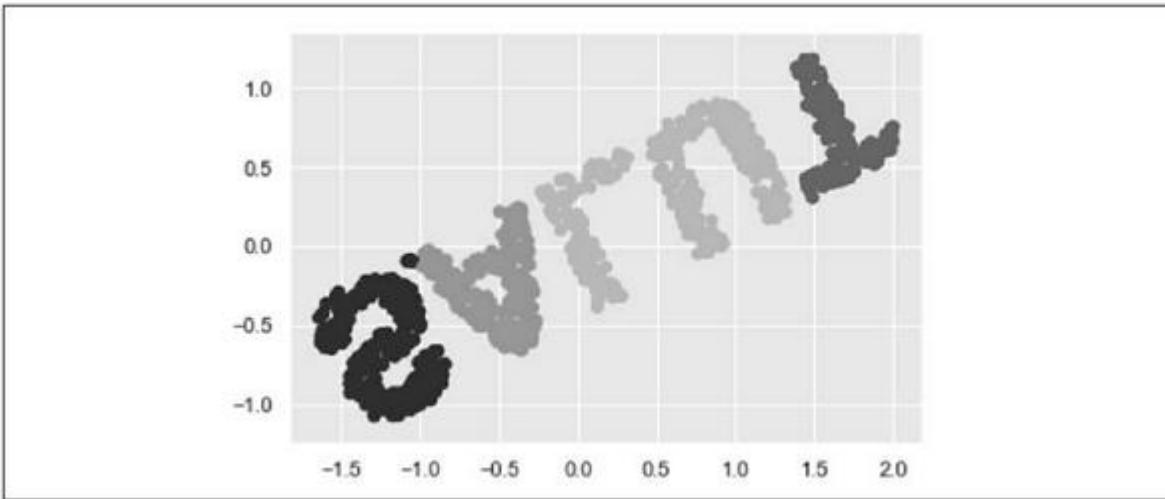


Figure 5.97 : Traitement MDS à partir des distances entre paires de points.

Cet algorithme permet donc de récupérer l'une des représentations de coordonnées en deux dimensions possibles, en utilisant seulement la matrice de distances de N par N qui sert à décrire les relations entre les points.

MDS pour l'apprentissage par variété (manifold learning)

L'intérêt de ce genre de traitement devient plus évident une fois que l'on a envisagé de faire calculer une matrice de distances pour des données dans n'importe quelle dimension. Au lieu de se contenter de faire tourner les données dans le plan, nous pouvons les projeter en trois dimensions au moyen de la fonction suivante qui est pour l'essentiel une généralisation en trois dimensions de la matrice de rotation précédente :

In[9]:

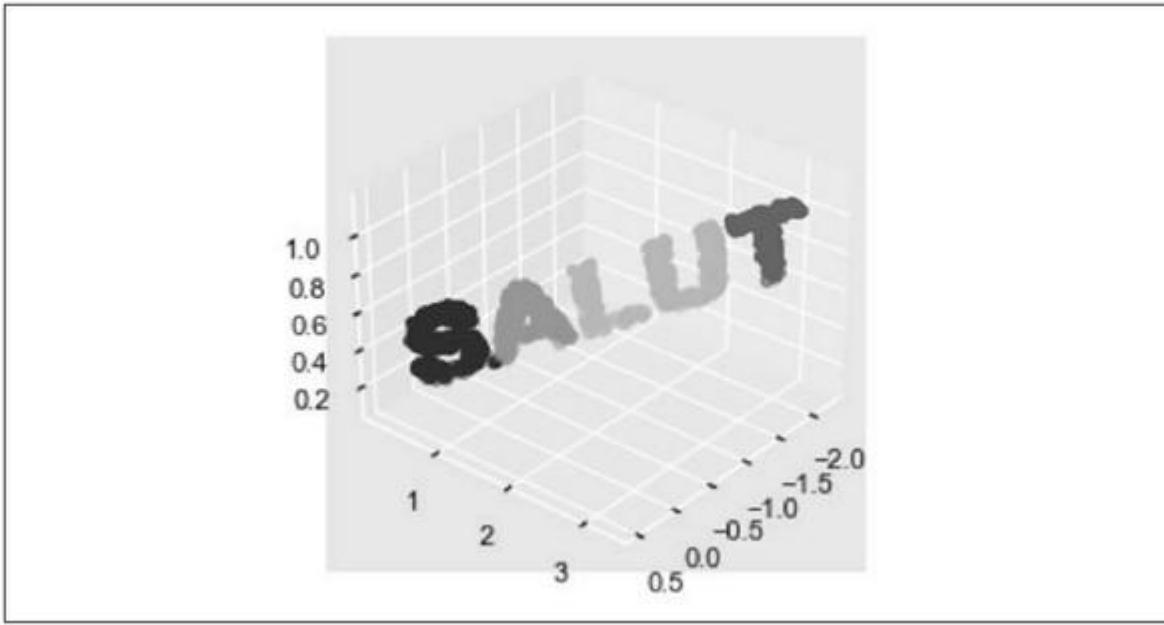
```
def random_projection(X, dimension=3, rseed=42):
    assert dimension >= X.shape[1]
    rng = np.random.RandomState(rseed)
    C = rng.randn(dimension, dimension)
    e, V = np.linalg.eigh(np.dot(C, C.T))
    return np.dot(X, V[:X.shape[1]])  
  
X3 = random_projection(X, 3)  
X3.shape
```

Out[9]: (1000, 3)

Nous demandons un tracé du résultat pour voir sur quoi nous travaillons ([Figure 5.98](#)) :

In[10]:

```
from mpl_toolkits import mplot3d
ax = plt.axes(projection='3d')
ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
             **colorize)
ax.view_init(azim=70, elev=50)
```



[Figure 5.98](#) : Données intégrées linéairement en trois dimensions.

Il ne reste plus qu'à utiliser l'estimateur MDS à partir de ces données en trois dimensions pour qu'il calcule la matrice de distances et trouve l'intégration en deux dimensions optimale de la matrice. Nous obtenons au final une représentation des données de départ ([Figure 5.99](#)) :

In[11]:

```
model = MDS(n_components=2, random_state=1)
out3 = model.fit_transform(X3)
plt.scatter(out3[:, 0], out3[:, 1], **colorize)
plt.axis('equal');
```

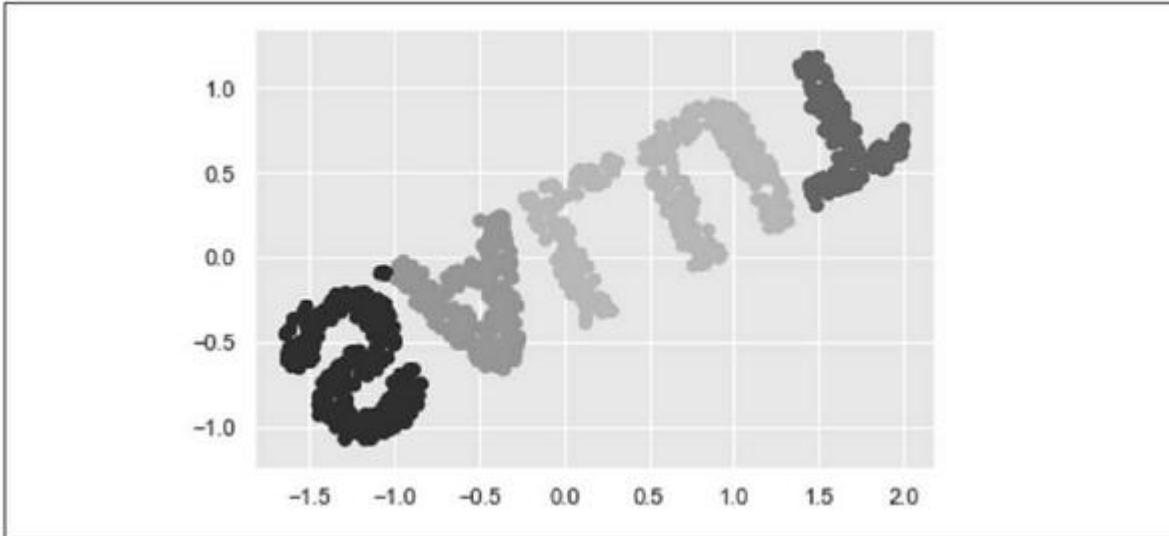


Figure 5.99 : Intégration MDS des données en trois dimensions. Les données sont récupérées hormis une rotation et une mise en miroir.

Nous venons de voir l'essentiel du traitement d'un estimateur d'apprentissage par variété : il part de données intégrées à grand nombre de dimensions et cherche une représentation à moins de dimensions qui préserve certaines relations. Dans le cas de MDS, c'est la distance entre chaque paire de points qui est préservée.

Intégration linéaire : échec de MDS

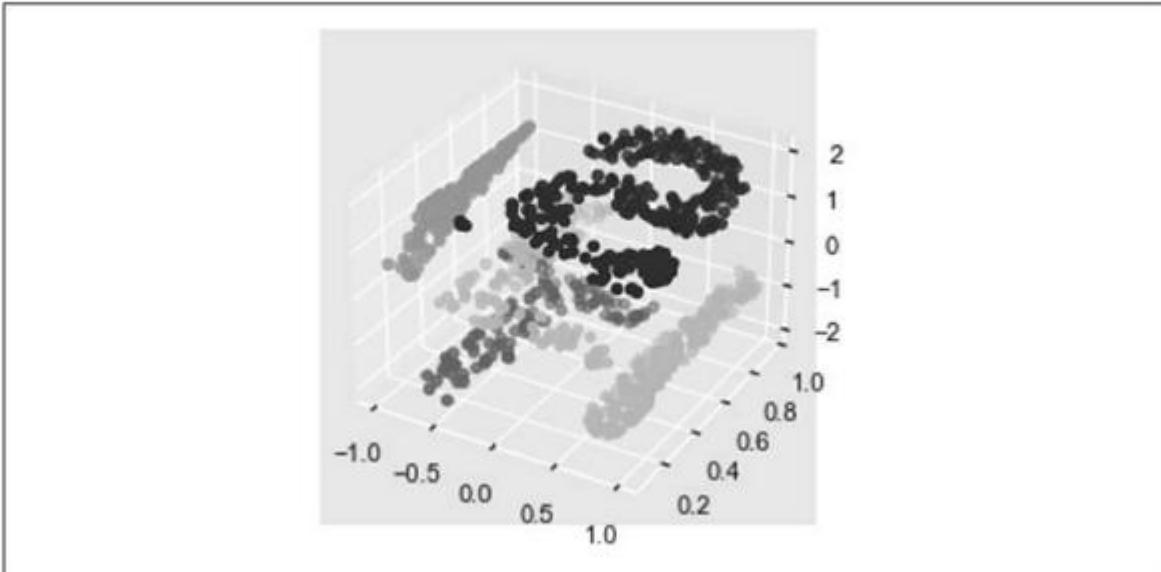
Nous n'avons rencontré pour l'instant que des intégrations linéaires, avec rotation, translation et mise à l'échelle vers des espaces à plus de dimensions. L'algorithme MDS est à genoux lorsque l'intégration n'est plus linéaire, c'est-à-dire avec des opérations plus complexes. Partons du cas suivant

qui consiste à déformer le mot pour qu'il forme un « S » en trois dimensions :

```
In[12]:  
def make_hello_s_curve(X):  
    t = (X[:, 0] - 2) * 0.75 * np.pi  
    x = np.sin(t)  
    y = X[:, 1]  
    z = np.sign(t) * (np.cos(t) - 1)  
    return np.vstack((x, y, z)).T  
  
XS = make_hello_s_curve(X)
```

Nous ne sommes qu'en trois dimensions, mais l'intégration est devenue beaucoup plus compliquée ([Figure 5.100](#)) :

```
In[13]:  
from mpl_toolkits import mplot3d  
ax = plt.axes(projection='3d')  
ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],  
**colorize);
```



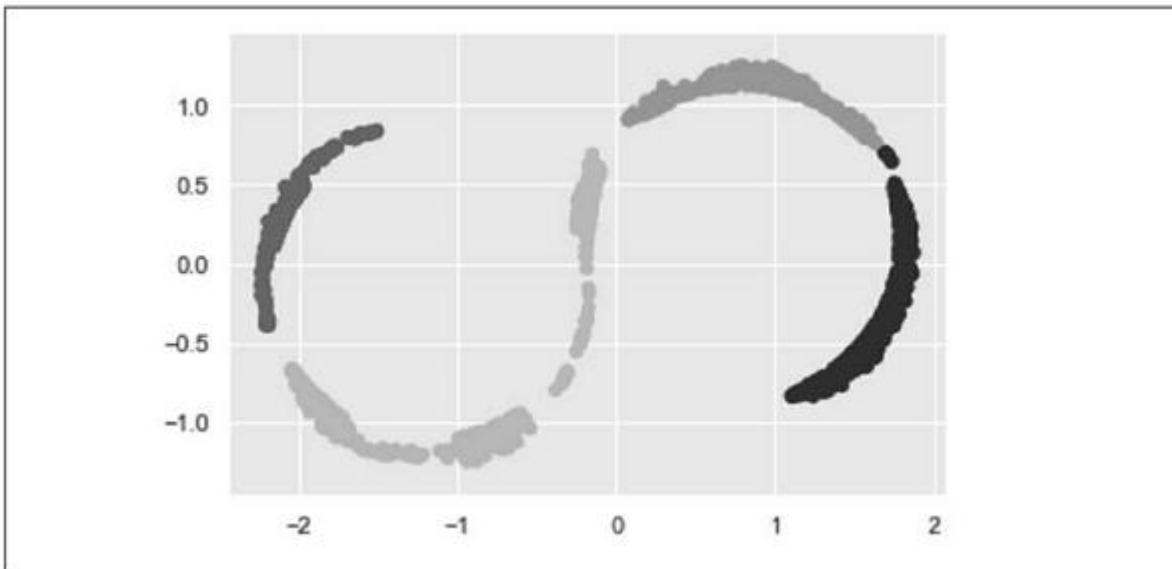
[Figure 5.100](#) : Intégration de données de façon non linéaire en trois dimensions.

Les relations essentielles entre les points existent toujours, mais l'ensemble a été transformé de façon non linéaire pour former un « S ».

Si nous appliquons un algorithme MDS simple sur ces données, celui-ci ne parviendra pas à « déplier » la figure et nous perdrons les relations fondamentales de la variété intégrée ([Figure 5.101](#)) :

In[14]:

```
from sklearn.manifold import MDS
model = MDS(n_components=2, random_state=2)
outS = model.fit_transform(XS)
plt.scatter(outS[:, 0], outS[:, 1], **colorize)
plt.axis('equal');
```



[Figure 5.101](#) : Application de l'algorithme MDS à des données non linéaires ; la structure sous-jacente n'est pas retrouvée.

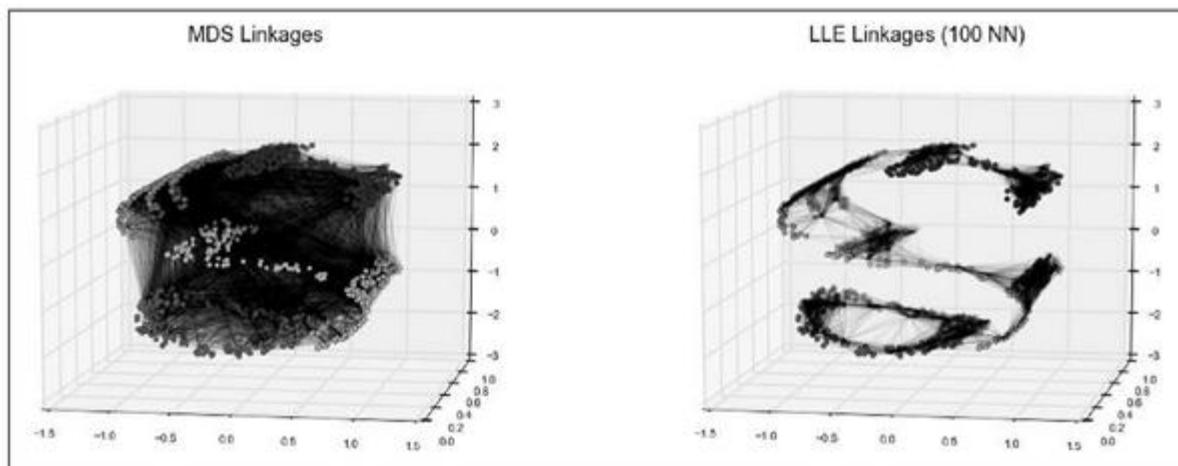
La meilleure intégration linéaire en deux dimensions ne réussit pas à déplier la courbe en S, et fait perdre les données de l'axe y.

Variétés non linéaires : intégration à linéarité locale (LLE)

Comment nous sortir de cette impasse ? Nous constatons que le problème est lié au fait que MDS cherche à préserver les distances entre des points très éloignés. Ne pourrions-nous pas modifier l'algorithme pour qu'il ne cherche à préserver que les distances entre points proches ? L'intégration qui en résulterait se rapprocherait plus de nos attentes.

La différence est visible dans les deux traitements de la [Figure 5.102](#).

Chaque ligne estompée correspond à une distance à préserver dans l'intégration. Le panneau de gauche montre le modèle qui est utilisé par MDS. Il cherche à préserver les distances entre toutes les paires de points. Dans le panneau droit, nous voyons le modèle qu'utilise un algorithme d'apprentissage par variété portant le nom d'intégration à linéarité locale LLE (*Locally Linear Embedding*). L'algorithme ne cherche à préserver les distances que pour les *points voisins*, dans l'exemple les 100 voisins les plus proches de chaque point.



[Figure 5.102](#) : Comparaison des recherches de distances entre points avec MDS et avec LLE.

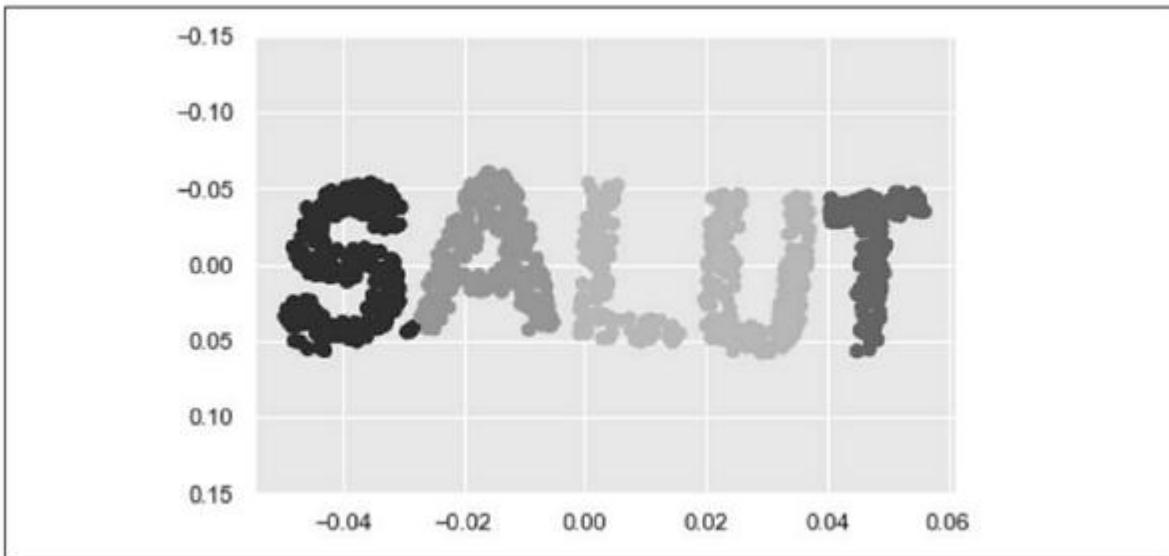
Nous voyons dans le panneau gauche pourquoi MDS échoue : il n'y a aucune possibilité d'aplatir les données tout en

maintenant les longueurs de toutes les lignes tracées entre chaque couple de points. Les choses deviennent envisageables dans le panneau droit. Nous devrions pouvoir trouver une solution pour dérouler les données tout en maintenant les longueurs des lignes tracées à peu près les mêmes. C'est ce que permet l'algorithme LLE qui utilise une optimisation globale d'une *fonction de coût*.

Parmi les variantes existantes de LLE, nous allons adopter l'algorithme LLE modifié afin de retrouver la variété en deux dimensions qui a été intégrée. Cette variante LLE modifiée s'en sort généralement mieux que les autres lorsqu'il s'agit de récupérer une variété bien définie en ajoutant peu de distorsion ([Figure 5.103](#)) :

In[15]:

```
from sklearn.manifold import  
LocallyLinearEmbedding  
model = LocallyLinearEmbedding(n_neighbors=100,  
n_components=2,  
method='modified',  
eigen_solver='dense')  
out = model.fit_transform(XS)  
  
fig, ax = plt.subplots()  
ax.scatter(out[:, 0], out[:, 1], **colorize)  
ax.set_xlim(0.15, -0.15);
```



[Figure 5.103](#) : Une intégration localement linéaire permet de retrouver les données sous-jacentes à partir d'une entrée intégrée non linéairement.

Le résultat retrouve les relations essentielles des données de départ, même si il reste un peu de distorsion.

Considération à propos des méthodes par variété

Bien que leurs résultats soient captivants, les techniques d'apprentissage par variété s'avèrent en pratique assez délicates et ne sont que peu utilisées, sauf pour une visualisation qualitative de données à dimensions nombreuses.

Voici quelques-uns des défis qui se posent à ce genre d'apprentissage ; tous rendent l'approche par PCA plus intéressante :

- Dans l'apprentissage par variété, il n'existe pas d'infrastructure adéquate pour gérer les données manquantes. À l'inverse, vous disposez de plusieurs approches itératives simples pour gérer les données manquantes en PCA.
- Dans l'apprentissage par variété, le bruit dans les données peut « court-circuiter » la variété et altérer fortement l'intégration. Tout au contraire, PCA filtre naturellement le bruit des composantes principales.
- Le résultat d'une intégration par variété dépend en général énormément du nombre de voisins choisis. Il n'y a en général aucun moyen fiable de choisir le nombre de voisins idéal. Ce dilemme n'existe pas en PCA.
- Le nombre optimal de dimensions de sortie est difficile à estimer en apprentissage par variété. Au contraire, vous pouvez trouver le nombre de dimensions de sortie PCA en vous basant sur la variance expliquée.
- Dans l'apprentissage par variété, ce que signifient les dimensions intégrées n'est pas toujours évident. En PCA, les composantes principales sont toujours faciles à comprendre.

- Dans l'apprentissage par variété, l'augmentation de la charge de calculs varie en raison de $O[N^2]$ ou $O[N^3]$. En PCA, on dispose d'approches randomisées en général bien plus rapides. Citons néanmoins le paquetage nommé megaman (<https://github.com/mmp2/megaman>) qui propose des implémentations de l'apprentissage par variété capables de monter en charge.

Une fois tous ces points faibles connus, nous pensons que le seul vrai avantage de l'apprentissage par variété en comparaison de PCA est sa capacité à préserver les relations non linéaires. C'est pourquoi je ne me lance dans une exploration des données par variété qu'après une première exploration avec PCA.

Scikit-Learn contient plusieurs des variantes les plus répandues de l'apprentissage par variété, en plus d'Isomap et LLE. Vous trouverez une discussion et un comparatif dans la documentation de Scikit-Learn, à la page décrivant les manifolds. D'après mon expérience personnelle, voici quelques recommandations :

- Pour les problèmes d'étude tels que le mot en S que nous venons de voir, la méthode la plus efficace est LLE ainsi que ses variantes, notamment LLE modifié. Cela correspond à `sklearn.manifold.LocallyLinearEmbedding`.

- Pour les données à dimensions nombreuses du monde réel, LLE est souvent décevant, mais le mappage Isomap procure généralement des intégrations plus parlantes. Cela correspond à `sklearn.manifold.Isomap`.
- Enfin, lorsque les données sont fortement agrégées, c'est l'algorithme t-SNE qui semble le plus efficace, même s'il peut être vraiment long. Cela correspond à `sklearn.manifold.TSNE`.

Si vous avez envie de voir comment fonctionnent ces autres algorithmes, n'hésitez pas à appliquer les méthodes correspondantes sur les données de l'exemple.

Exemple : Isomap en reconnaissance de visages

Un domaine d'emploi fréquent de l'apprentissage par variété est l'analyse des relations entre les points de données à nombreuses dimensions. Nous avons vu que c'est toujours le cas des images. Un jeu d'images de 1 000 pixels chacune peut être considéré comme une collection de points dans 1 000 dimensions, la luminosité de chaque pixel définissant la coordonnée dans chacune de ces dimensions.

Essayons de traiter les visages des personnalités politiques avec l'algorithme Isomap. Ces visages ont déjà été utilisés

avec les machines à vecteurs de support et l'analyse PCA. Nous commençons par télécharger les données et les stocker dans le répertoire de travail pour pouvoir y accéder ensuite :

In[16]:

```
from sklearn.datasets import fetch_lfw_people
faces =
fetch_lfw_people(min_faces_per_person=30)
faces.data.shape
```

Out[16]: (2370, 2914)

Nous obtenons 2 370 images, chacune étant constituée de 2 914 pixels. Il s'agit effectivement de points de données dans un espace à 2 914 dimensions !

Demandons l'affichage de quelques images pour voir sur quoi nous travaillons ([Figure 5.104](#)):

In[17]:

```
fig, ax = plt.subplots(4, 8,
subplot_kw=dict(xticks=[], yticks=[]))
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='gray')
```



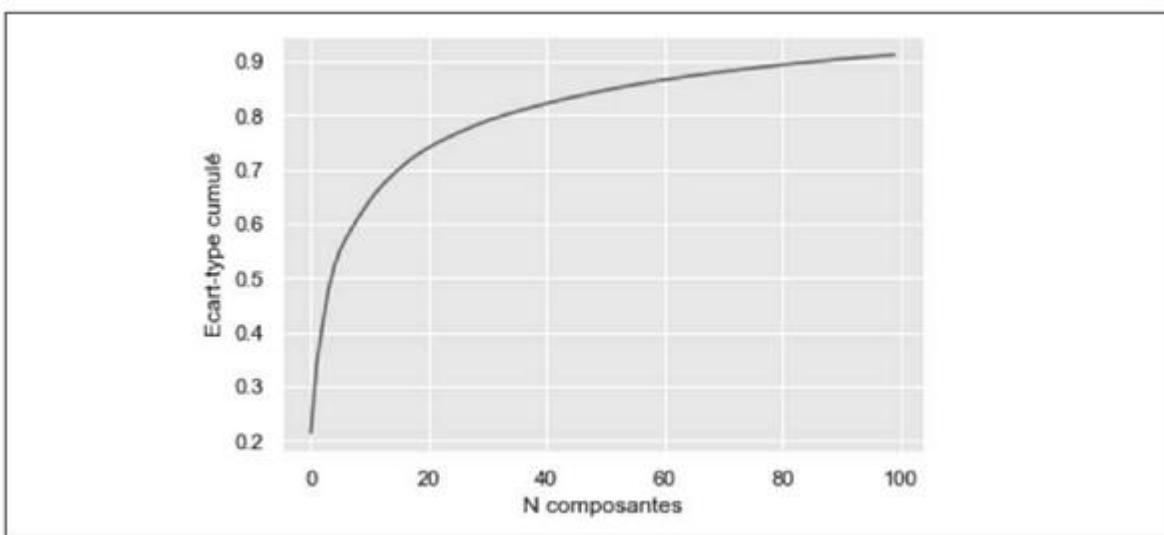
[Figure 5.104](#) : Un aperçu des visages des personnalités politiques.

Pour découvrir les relations fondamentales entre les images, il nous faudrait afficher une intégration après réduction des dimensions. Nous pouvons commencer par calculer une PCA pour pouvoir utiliser le ratio de variance expliquée. Nous aurons ainsi une idée du nombre de caractéristiques linéaires qu'il faut pour pouvoir continuer à décrire les données ([Figure 5.105](#)):

In[18]:

```
from sklearn.decomposition import PCA as  
RandomizedPCA  
model = RandomizedPCA(100).fit(faces.data)  
plt.plot(np.cumsum(model.explained_variance_ratio_))  
plt.xlabel('n composantes')  
plt.ylabel('Variance cumulative');
```

Nous constatons qu'il faut à peu près 100 composantes pour maintenir 90 % de variance. Cela confirme que les données ont un grand nombre de dimensions par nature, et qu'elles ne peuvent pas être décrites de façon linéaire avec une poignée de composantes.



[Figure 5.105](#) : Variance cumulée de la projection PCA.

Dans cette situation, nous pourrions profiter d'une intégration par variété non linéaire telle que LLE ou Isomap. Demandons une intégration Isomap à partir des visages en réutilisant une technique déjà vue plus haut :

In[19] :

```
from sklearn.manifold import Isomap
model = Isomap(n_components=2)
proj = model.fit_transform(faces.data)
proj.shape
```

```
Out[19]: (2370, 2)
```

Nous obtenons une projection de toutes les images en deux dimensions. Pour visualiser plus aisément le travail fait par la projection, nous définissons une fonction qui va afficher des vignettes des images aux positions projetées :

```
In[20]:
```

```
from matplotlib import offsetbox

def plot_components(data, model, images=None,
ax=None,
                     thumb_frac=0.05,
cmap='gray'):
    ax = ax or plt.gca()
    proj = model.fit_transform(data)
    ax.plot(proj[:, 0], proj[:, 1], '.k')

    if images is not None:
        min_dist_2 = (thumb_frac * max(proj.max(0) -
proj.min(0))) ** 2
        shown_images = np.array([2 * proj.max(0)])
        for i in range(data.shape[0]):
            dist = np.sum((proj[i] - shown_images) ** 2, 1)
            if np.min(dist) < min_dist_2:
                # Masquer les points trop proches
                continue
```

```
    shown_images = np.vstack([shown_images,
proj[i]])
    imagebox = offsetbox.AnnotationBbox(
offsetbox.OffsetImage(images[i], cmap=cmap),
proj[i])
    ax.add_artist(imagebox)
```

Utilisons notre fonction pour obtenir le résultat ([Figure 5.106](#)) :

```
In[21]:
fig, ax = plt.subplots(figsize=(10, 10))
plot_components(faces.data,
model=Isomap(n_components=2),
images=faces.images[:, ::2, ::2])
```

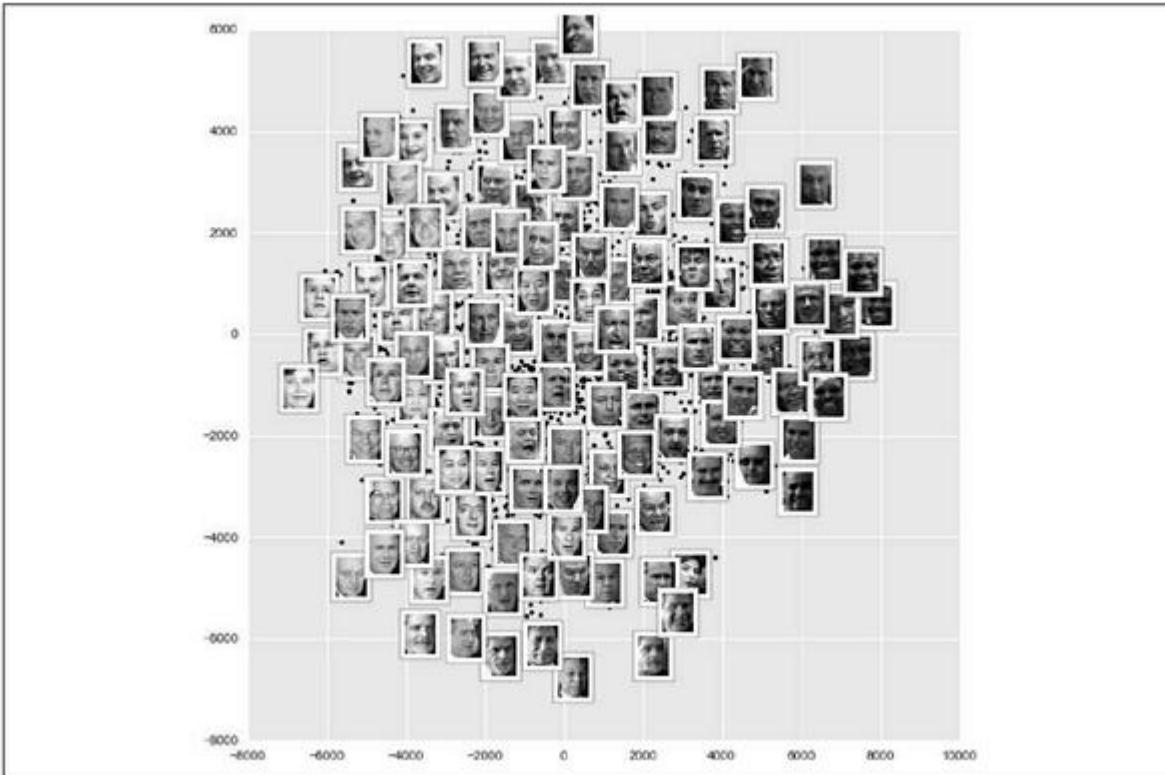


Figure 5.106 : Intégration des données des visages par Isomap.

Le résultat est intéressant, puisque les deux premières dimensions résultantes d'Isomap semblent effectivement décrire des caractéristiques fondamentales. Le degré de luminosité de chaque image va de gauche à droite et l'orientation de chaque visage du haut vers le bas. Nous avons donc bien obtenu une représentation visuelle des caractéristiques fondamentales des données.

Nous pourrions poursuivre en demandant une classification, en utilisant en entrée de l'algorithme de classification des caractéristiques par variété, comme nous l'avons fait dans la section sur les machines à vecteurs de support.

Exemple : affichage de la structure dans des chiffres manuscrits

Voyons un autre exemple de visualisation au moyen de l'apprentissage par variété. Nous allons utiliser un autre jeu de chiffres manuscrits, celui du MNIST. Il est proche de celui déjà utilisé lorsque nous avons vu les arbres de décision. On peut utiliser l'outil de Scikit-Learn approprié pour télécharger les données depuis l'adresse <http://mldata.org/> :

In[22]:

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
mnist.data.shape
```

Out[22]: (70000, 784)



(N.d.T.) Ayez un peu de patience pour l'exécution du bloc précédent. Par ailleurs, il est possible que cet exemple rencontre des soucis lorsque vous tenterez de l'exécuter, car il se fonde sur des éléments en pleine évolution au moment d'écrire ces lignes. Les archives d'accompagnement du livre (voir l'introduction) contiendront si nécessaire un complément d'information.

Nous obtenons 70 000 images de 784 pixels (chaque image mesure 28×28). Demandons à voir un aperçu de la collection ([Figure 5.107](#)):

In[23]:

```
fig, ax = plt.subplots(6, 8,
subplot_kw=dict(xticks=[], yticks=[]))
for i, axi in enumerate(ax.flat):
    axi.imshow(mnist.data[1250 * i].reshape(28,
28), cmap='gray_r')
```



[Figure 5.107](#) : Un aperçu des chiffres du jeu MNIST.

On constate que le style de tracé des chiffres est assez variable.

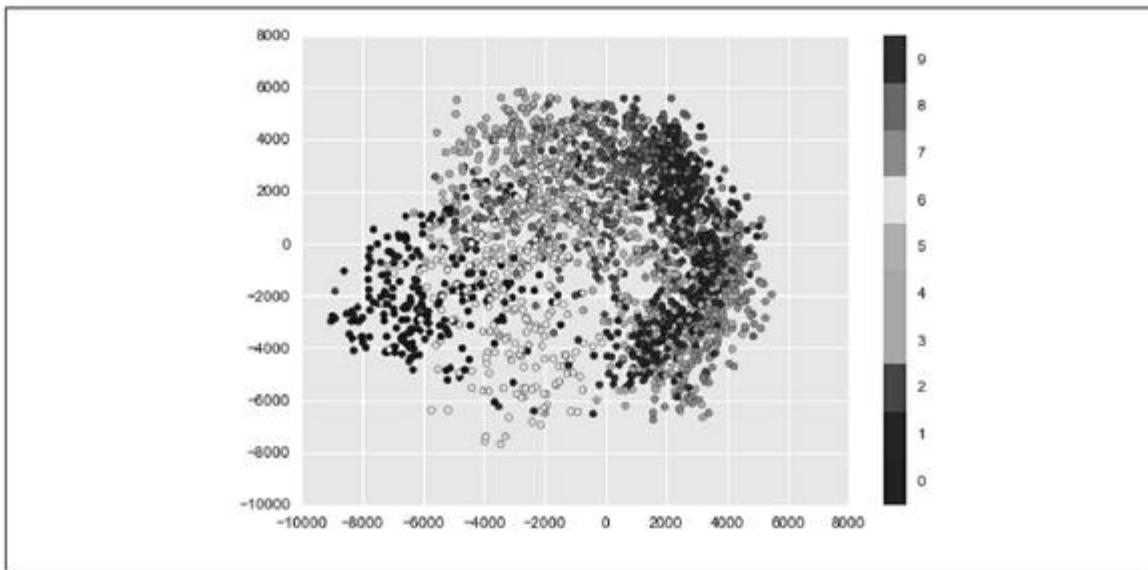
Nous demandons une projection de type apprentissage par variété ([Figure 5.108](#)). Pour écourter le traitement, nous n'utilisons que 1/30 du jeu complet, soit environ 2 000 points. L'algorithme utilisé ne réussit pas très bien sa montée en charge ; j'ai constaté qu'il était astucieux de démarrer avec quelques milliers d'échantillons pour une

première exploration d'ensemble avant d'envisager un traitement du lot complet :

In[24]:

```
# Avec 1/30 du lot complet pour abréger !
data = mnist.data[:30]
target = mnist.target[:30]

model = Isomap(n_components=2)
proj = model.fit_transform(data)
plt.scatter(proj[:, 0], proj[:, 1], c=target,
            cmap=plt.cm.get_cmap('jet', 10))
plt.colorbar(ticks=range(10))
plt.ylim(-10000, 8000)
```



[Figure 5.108](#) : Intégration Isomap des chiffres manuscrits MNIST.

Le nuage de points affiché permet de distinguer certaines relations entre les points, mais la lecture n'est pas aisée.

Rendons les choses plus claires en ne traitant qu'un seul chiffre à la fois ([Figure 5.109](#)):

In[25]:

```
from sklearn.manifold import Isomap
```

```
# Limiter à 1/4 des chiffres "1"
```

```
data = mnist.data[mnist.target == 1][::4]
```

```
fig, ax = plt.subplots(figsize=(10, 10))
```

```
model = Isomap(n_neighbors=5, n_components=2,
```

```
eigen_solver='dense')
```

```
plot_components(data, model,
```

```
images=data.reshape((-1, 28, 28)),
```

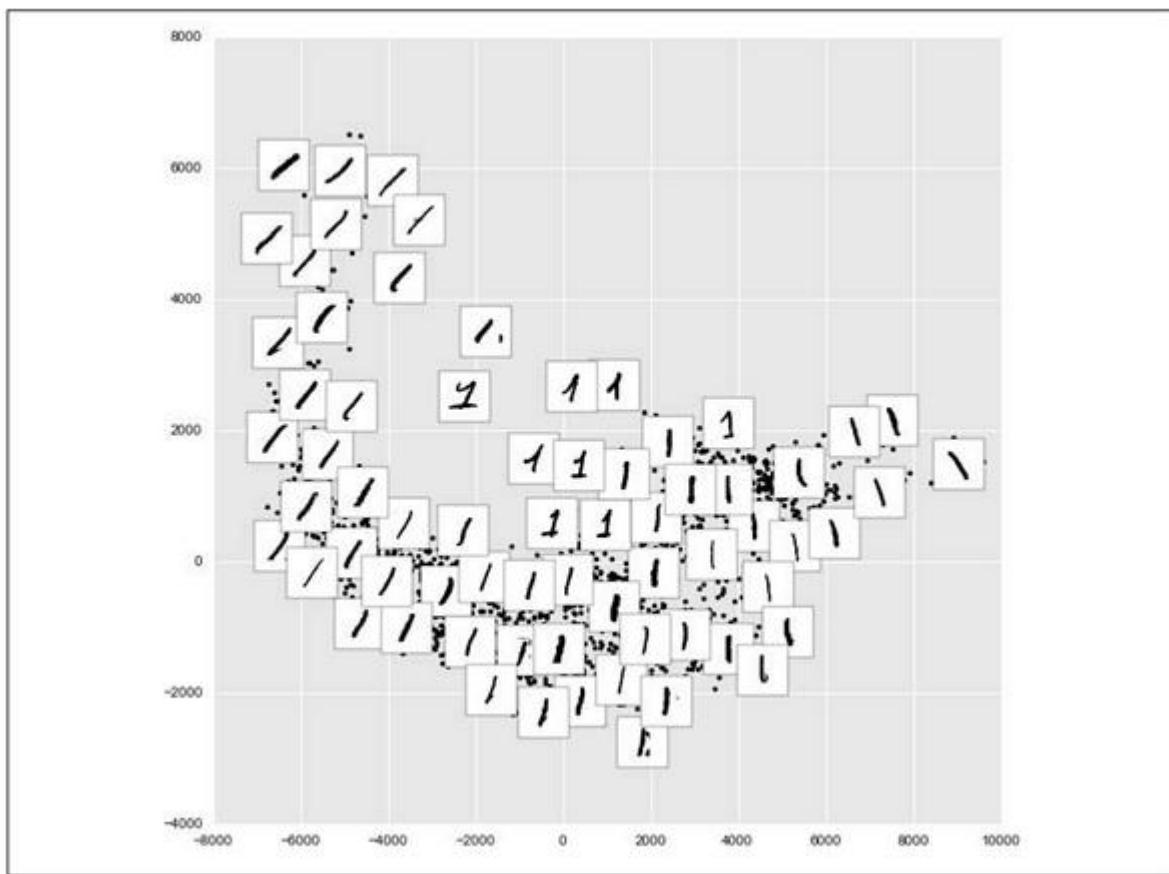
```
ax=ax, thumb_frac=0.05,
```

```
cmap='gray_r')
```

Nous obtenons une bonne idée de la grande variété de tracés du seul chiffre « 1 ». Les données sont réparties le long d'une courbe dans l'espace de projection, et il semble que la courbe épouse l'orientation du tracé. Vers le haut du graphique, les chiffres possèdent un chapeau ou une base, mais ces occurrences sont assez rares. Nous remarquons également des valeurs aberrantes, par exemple des portions de chiffres voisins qui ont été emportées dans les images extraites.

Ces opérations ne nous permettent pas de faire une classification des chiffres, mais elles nous aident à mieux

comprendre la nature des données, et nous donnent des indices sur ce qu'il faut réaliser ensuite. Nous pouvons ainsi plus facilement décider comment effectuer un prétraitement des données avant de les injecter dans un pipeline de classification.



[Figure 5.109](#) : Intégration Isomap des chiffres manuscrits 1.

5.11 : Groupements en k-moyennes

Les précédentes sections nous ont permis d'explorer une première catégorie de modèles d'apprentissage non supervisé, correspondant à la réduction dimensionnelle. Découvrons une autre catégorie de modèles sans supervision : les algorithmes de groupement ou partitionnement. Leur but est de déduire à partir des propriétés des données comment diviser de façon optimale ou agréger en plusieurs parties les labels des groupes de points de données.

Scikit-Learn, comme d'autres outils, offre tout un choix d'algorithmes de partitionnement. Celui sans doute le plus simple à maîtriser correspond au partitionnement en k-moyennes qui correspond à la classe `sklearn.cluster.KMeans`. Nous commençons par nos opérations d'import habituelles :

In[1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set() # for plot
styling
import numpy as np
```

Présentation des k-moyennes

L'algorithme en k-moyennes cherche un nombre de parties prédéterminé dans un jeu de données à plusieurs dimensions sans labels. Les décisions sont prises en se fondant sur des critères de partitionnement :

- Le centre de chaque partie ou groupe est la moyenne arithmétique de tous les points du groupe.
- Chaque point d'un groupe est plus proche de son centre de groupe que des autres centres.

Ces deux critères forment la base du modèle. Nous allons voir comment l'algorithme travaille pour atteindre cet objectif. Voyons d'abord un jeu de données simplifié et le résultat produit en k-moyennes.

Nous générerons d'abord un jeu à deux dimensions constitué de quatre groupes distincts. En guise de rappel du fait que l'algorithme est non supervisé, nous ne montrons pas les labels dans la visualisation ([Figure 5.110](#)) :

In[2] :

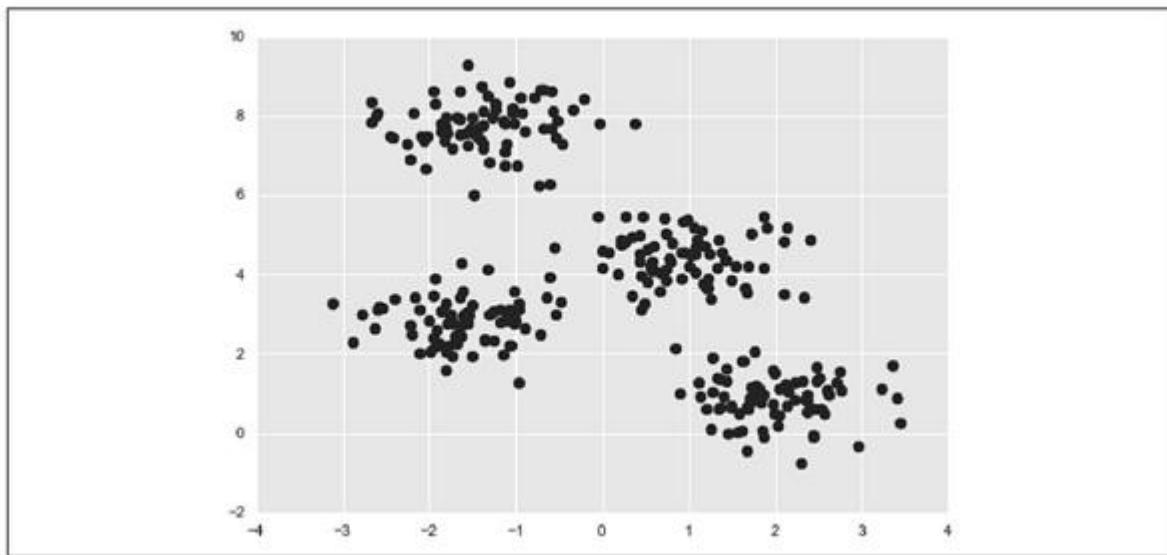
```
from sklearn.datasets import make_blobs
X, y_true = make_blobs(n_samples=300, centers=4,
                      cluster_std=0.60,
                      random_state=0)
```

```
plt.scatter(X[:, 0], X[:,  
1], s=50);
```

Les quatre groupes se détectent à l'œil nu. C'est ce que fait l'algorithme en k-moyennes de façon automatique. Dans Scikit-Learn, nous utilisons l'API d'estimateur habituel :

In[3]:

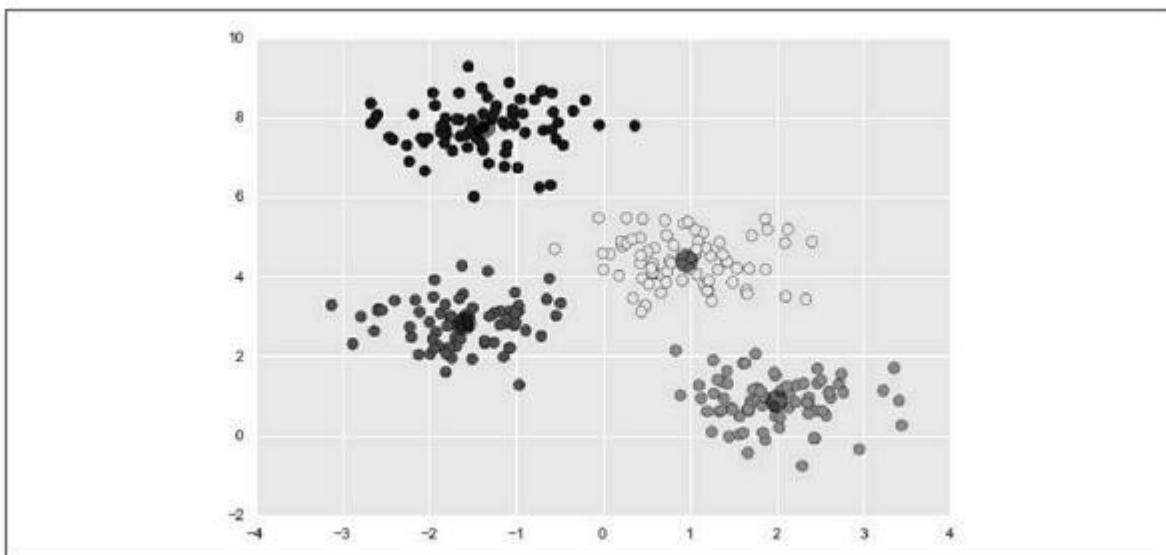
```
from sklearn.cluster import KMeans  
kmeans = KMeans(n_clusters=4)  
kmeans.fit(X)  
y_kmeans = kmeans.predict(X)
```



[Figure 5.110](#) : Données de démonstration d'un partitionnement.

Affichons les résultats en distinguant les couleurs selon les labels. Nous ajoutons également un point pour marquer le centre de chaque partie telle que l'estimateur en k-moyennes l'a trouvé ([Figure 5.111](#)) :

```
In[4]:  
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50,  
cmap='viridis')  
  
centers = kmeans.cluster_centers_  
plt.scatter(centers[:, 0], centers[:, 1],  
c='black', s=200, alpha=0.5);
```



[Figure 5.111](#) : Centre des parties trouvées en k -moyennes et parties distinguées par des couleurs.

Nous sommes heureux de découvrir que cet algorithme trouve les points et les parties comme nous le ferions à l'œil nu, au moins dans ce cas simplifié. Vous vous demandez peut-être comment l'algorithme a pu travailler si vite. Le nombre de combinaisons de regroupement augmente de façon exponentielle, en fonction du nombre de points de données. Une recherche consistant à essayer toutes les

combinaisons deviendrait rapidement beaucoup trop longue. Cette recherche exhaustive n'est pas requise. L'approche habituelle avec les k-moyennes consiste à utiliser une technique itérative qui porte le nom *espérance-maximisation*.

Algorithme en k-moyennes et Espérance-Maximisation

Plusieurs domaines scientifiques font appel au puissant algorithme qui porte le nom *Espérance-Maximisation*, que nous abrégerons en E-M. Les k-moyennes constituent une application simple et lisible de cet algorithme, et nous allons donc le détailler. L'approche Espérance-Maximisation réalise le raisonnement suivant :

1. Deviner quelques centres de groupe.
2. Répéter l'opération jusqu'à convergence.
 - a. Étape E : affecter des points au centre de groupe le plus proche.
 - b. Étape M : positionner le centre de groupe sur la moyenne.

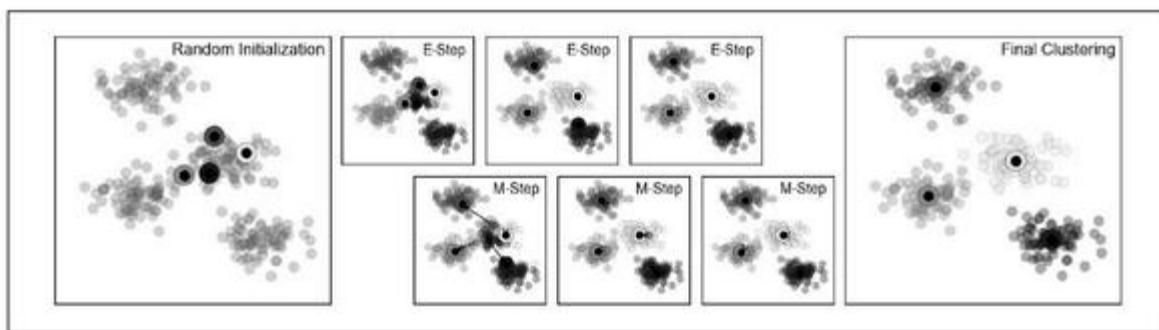
« L'étape E » suppose de mettre à jour l'Espérance d'appartenance de chaque point à un groupe en particulier. « L'étape M » consiste à maximiser une fonction d'ajustement dont le but est de définir la position des

centres de groupes. Dans notre cas, l'opération se résume à prendre la moyenne des données dans chaque groupe.

La théorie de cet algorithme a été largement documentée par ailleurs. Pour résumer, dans les circonstances normales, chaque répétition de l'étape E et de l'étape M donne une estimation améliorée des caractéristiques des groupes.

La [Figure 5.112](#) permet de visualiser le travail de l'algorithme.

Dans cet exemple, les groupes convergent en trois tours ou itérations. Le fichier interactif accompagnant le livre contient une version interactive de cette figure.



[Figure 5.112](#) : Visualisation de l'algorithme E-M en k-moyennes.

L'algorithme en k-moyennes est tellement simple qu'il est possible de le reconstituer en quelques lignes de code. En voici une implémentation volontairement minimalist ([Figure 5.113](#)) :

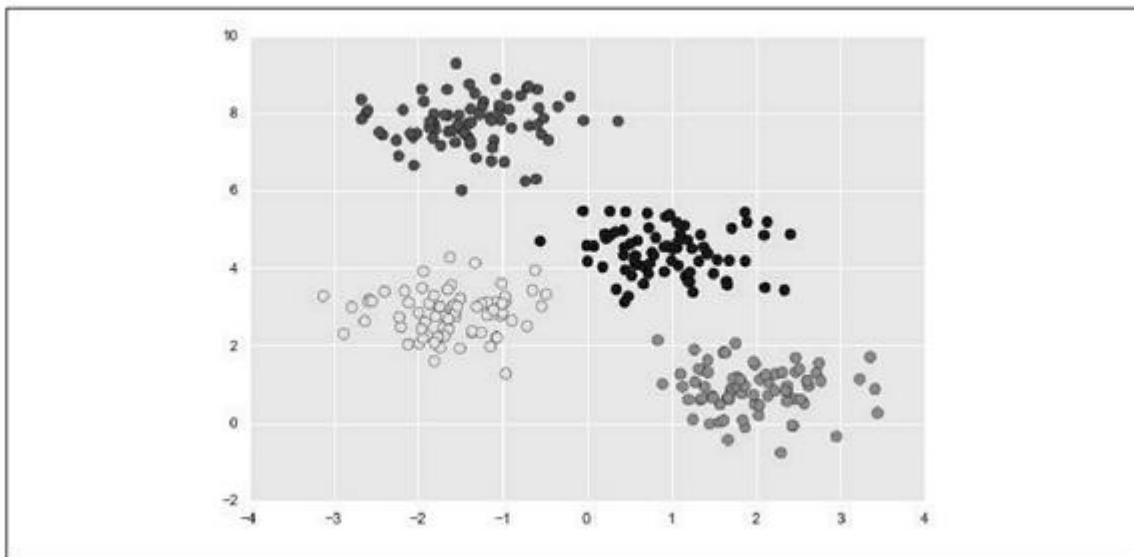
```
In[5]:  
from sklearn.metrics import  
pairwise_distances_argmin  
  
def find_clusters(X, n_clusters, rseed=2):  
    # 1. Choix aléatoire des clusters (ou groupes)  
    rng = np.random.RandomState(rseed)  
    i = rng.permutation(X.shape[0])[:n_clusters]  
    centers = X[i]  
    while True:  
        # 2a. Affecte des labels en fonction du centre le plus proche  
        center_labels =  
pairwise_distances_argmin(X, centers)  
  
        # 2b. Trouve nouveaux centres selon moyenne des points  
        new_centers = np.array([X[labels ==  
i].mean(0)  
                           for i in  
range(n_clusters)])  
  
        # 2c. Teste la convergence  
        if np.all(centers == new_centers):  
            break  
        centers = new_centers  
  
    return centers, labels
```

```

centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50,
cmap='viridis');

```

Les implémentations disponibles sont un peu plus sophistiquées, mais le code précédent donne une idée de la technique d'Espérance-Maximisation.



[Figure 5.113](#) : Différenciation des groupes en k -moyennes.

Points faibles d'Espérance-Maximisation

Quelques points sont à surveiller lorsque l'on utilise l'algorithme Espérance-Maximisation.

E-M n'aboutit pas nécessairement au meilleur résultat global

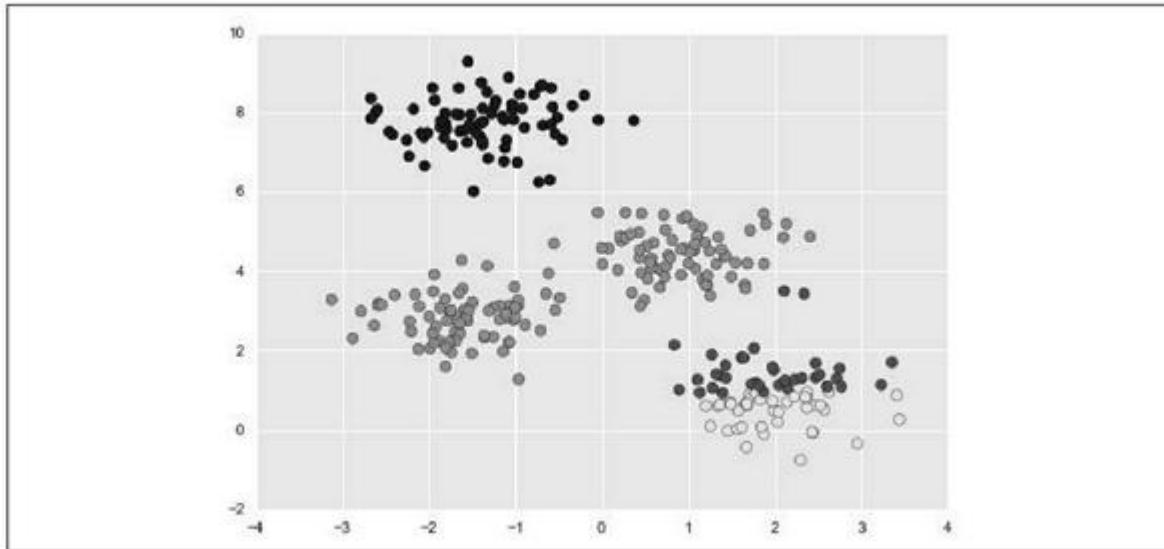
L'approche E-M assure améliorer le résultat lors de chaque étape, mais cela ne signifie pas qu'elle se dirige

progressivement vers la meilleure solution globale. Si nous utilisons par exemple une autre graine de génération aléatoire, les premières étapes d'évaluation ne sont pas très heureuses et le résultat est mauvais ([Figure 5.114](#)) :

```
In[6]:
```

```
centers, labels = find_clusters(X, 4, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50,
cmap='viridis');
```

En effet, l'approche E-M a bien convergé, mais pas en direction d'une configuration globalement optimale. C'est pour cette raison que l'on utilise l'algorithme plusieurs fois avec des points de départ différents, ce que fait d'ailleurs Scikit-Learn par défaut (en se basant sur la valeur du paramètre `n_init` qui vaut 10 par défaut).



[Figure 5.114](#) : Exemple de mauvaise convergence en k -moyennes.

Le nombre de groupes doit être choisi au départ

L'approche en k -moyennes comporte un autre défi habituel : c'est à vous de dire combien de groupes ou parties vous voulez obtenir car il ne peut pas le découvrir à partir des données. Si nous demandons par exemple de trouver six groupes, il cherche les six meilleurs groupes selon lui ([Figure 5.115](#)) :

In[7]:

```
labels = KMeans(6,
random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50,
cmap='viridis');
```

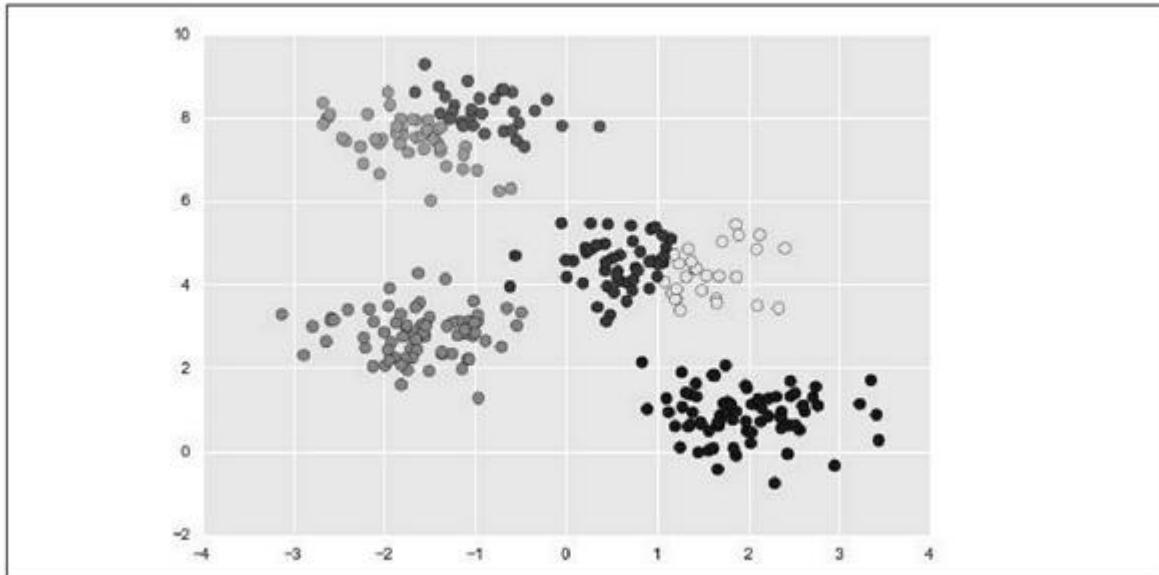


Figure 5.115 : Exemple d'un mauvais choix de groupes cibles.

Il n'est pas facile de juger de la pertinence des résultats. Il existe une approche assez intuitive que nous ne décrirons pas ici qui consiste à chercher le coefficient de silhouette (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html et [https://fr.wikipedia.org/wiki/Silhouette_\(clustering\)](https://fr.wikipedia.org/wiki/Silhouette_(clustering))).

Une autre solution consiste à adopter un algorithme de partitionnement plus complexe, de sorte qu'il offre une meilleure mesure quantitative de l'ajustement par rapport au nombre de groupes (les modèles de mélange gaussien) ou qui soit capable de trouver le bon nombre de groupes (par exemple DBSCAN, ou propagation d'affinité, tous disponibles dans le sous-module `sklearn.cluster`).

k-moyennes est limité à des frontières de groupe linéaires

Le principe fondamental de l'approche en k-moyennes qui consiste à choisir les points lorsqu'ils sont plus proches de leur centre de groupe que des autres centres rend cet algorithme fréquemment inefficace dès que les groupes montrent une géométrie complexe.

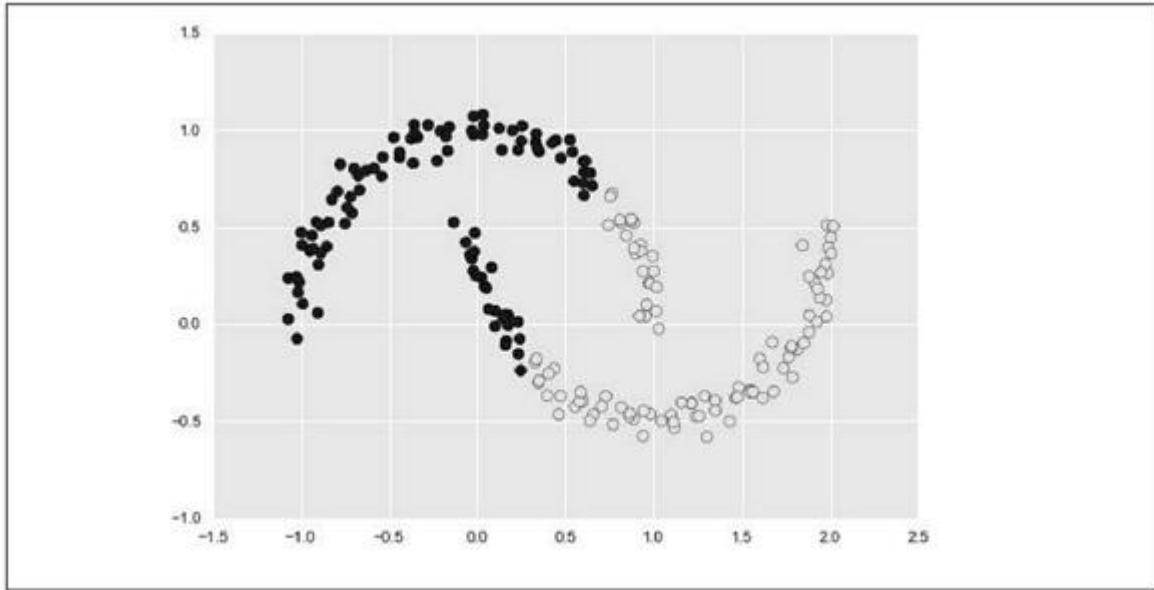
En effet, les frontières que va délimiter k-moyennes seront toujours linéaires, et seront donc fausses si les frontières réelles sont plus complexes. Prenons l'exemple suivant avec des labels de groupes tels que l'approche typique k-moyennes les trouvent ([Figure 5.116](#)) :

In[8] :

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05,
random_state=0)
```

In[9] :

```
labels = KMeans(2,
random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50,
cmap='viridis');
```



[Figure 5.116](#) : Échec de k-moyennes avec les frontières non linéaires.

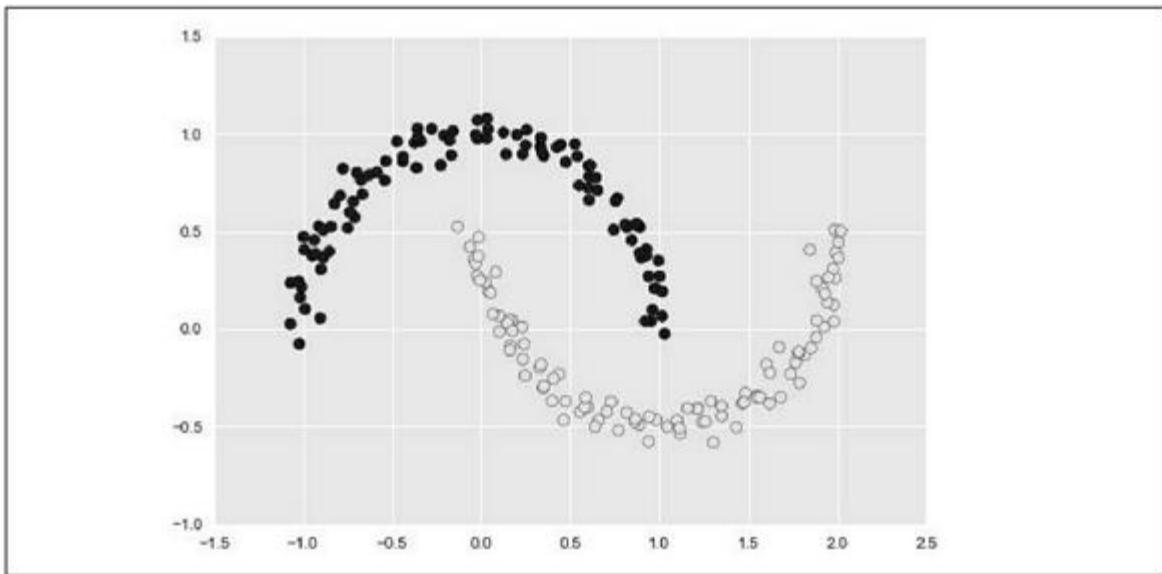
La situation est proche de celle rencontrée lorsque nous avons décrit les machines à vecteurs de support et utilisé une transformation de noyaux pour projeter les données dans un plus grand nombre de dimensions, afin de pouvoir faire une séparation linéaire. Nous pourrions utiliser le même genre d'astuce pour que k-moyennes détecte des frontières non linéaires.

D'ailleurs, Scikit-Learn propose une version de k-moyennes avec noyaux par son estimateur SpectralClustering. Il se fonde sur un graphe des plus proches voisins pour produire une représentation des données à plus grandes dimensions, puis affecte les labels grâce à un algorithme en k-moyennes ([Figure 5.117](#)) :

```
In[10]:
```

```
from sklearn.cluster import SpectralClustering
model = SpectralClustering(n_clusters=2,
                            affinity='nearest_neighbors',
                            assign_labels='kmeans'
)
labels = model.fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=50,
            cmap= viridis );
```

Nous constatons que dans cette approche, les k-moyennes avec noyaux réussissent à trouver des frontières plus complexes et non linéaires entre les groupes.



[Figure 5.117](#) : Frontières non linéaires découvertes par SpectralClustering.

K-moyennes peut être lent si le nombre d'échantillons est important

Lors de chaque itération des traitements de k-moyennes, il faut accéder à chacun des points du jeu de données. L'algorithme peut donc être assez lent, d'autant plus si le nombre d'échantillons est important. Vous vous demandez s'il est possible d'éviter cette exhaustivité ; vous pourriez par exemple tenter d'utiliser un sous-ensemble des données pour mettre à jour les centres de groupes à chaque étape. C'est justement l'idée de l'algorithme en k-moyennes en traitement par lots dont il existe une version nommée `sklearn.cluster.MiniBatchKMeans`. L'interface de l'algorithme reste la même que celle du KMeans standard et nous en verrons un exemple dans la suite de notre exposé.

Exemples

Si nous tenons bien compte des points faibles de l'algorithme, nous pouvons tirer profit de k-moyennes dans des situations diverses. Voyons deux exemples.

Exemple 1 : k-moyennes et chiffres

Essayons d'appliquer l'algorithme k-moyennes sur les chiffres isolés que nous avons déjà utilisé en découvrant les forêts aléatoires et l'analyse PCA. Nous allons tenter de détecter les chiffres similaires sans utiliser les labels. Cela équivaut à une première étape d'extraction de signification

en partant d'un jeu pour lequel vous n'avez aucune information de label.

Nous chargeons d'abord le fichier des chiffres puis demandons de trouver les parties ou groupes par kmeans. Souvenez-vous que ce jeu de chiffres contient 1 797 échantillons possédant chacun 64 caractéristiques. Chacune des caractéristiques correspond à la luminosité d'un pixel de chaque image de 8 sur 8 :

In[11]:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

Out[11]: (1797, 64)

Nous réalisons de façon classique le partitionnement :

In[12]:

```
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

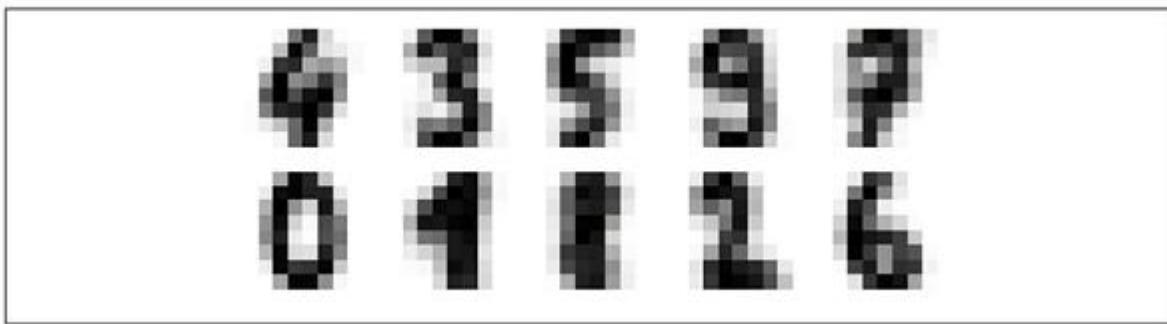
Out[12]: (10, 64)

Nous obtenons 10 parties en 64 dimensions. Les centres des groupes sont eux-mêmes des points en 64 dimensions et

peuvent être considérés comme les modèles typiques de chaque chiffre. Voyons justement à quoi ressemblent ces centres de groupes ([Figure 5.118](#)) :

In[13]:

```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8,
8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks[])
    axi.imshow(center, interpolation='nearest',
cmap=plt.cm.binary)
```



[Figure 5.118](#) : Les centres de groupes tels qu'appris avec k -moyennes.

Sans aucun label, kmeans est capable de trouver les groupes qui correspondent aux différents chiffres, à l'exception peut-être du 1 et du 8.

L'algorithme n'a aucun indice de la nature exacte de chaque groupe et les labels de 0 à 9 pourraient avoir été intervertis. Pour corriger cela, nous faisons correspondre chaque label

des groupes appris avec les vrais labels qui sont dans le fichier :

```
In[14]:  
from scipy.stats import mode  
  
labels = np.zeros_like(clusters)  
for i in range(10):  
    mask = (clusters == i)  
    labels[mask] = mode(digits.target[mask])[0]
```

Nous pouvons demander à connaître la précision de notre partitionnement non supervisé afin de savoir si les chiffres ont été bien distingués :

```
In[15]:  
from sklearn.metrics import accuracy_score  
accuracy_score(digits.target, labels)
```

Out[15]: 0.79354479688369506

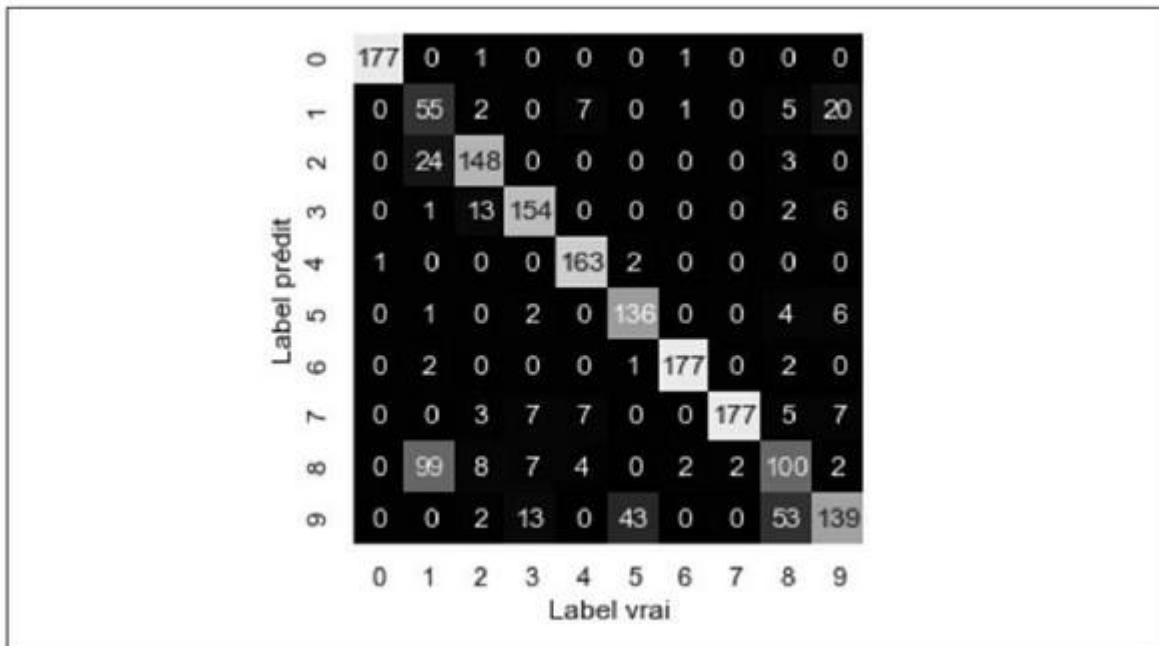
Autrement dit, ce simple algorithme nous a permis de distinguer correctement 80 % des chiffres ! Demandons la matrice de confusion correspondante ([Figure 5.119](#)) :

```
In[16]:  
from sklearn.metrics import confusion_matrix  
mat = confusion_matrix(digits.target, labels)  
sns.heatmap(mat.T, square=True, annot=True,
```

```

fmt='d', cbar=False,
        xticklabels=digits.target_names,
        yticklabels=digits.target_names)
plt.xlabel('Label vrai')
plt.ylabel('Label prédit');

```



[Figure 5.119](#) : Une matrice de confusion pour un classifieur en k -moyennes.

Comme nous l'avions déjà remarqué, c'est surtout entre les chiffres 1 et 8 que l'algorithme ne s'en sort pas bien. Néanmoins, nous venons de montrer que l'on pouvait créer un classifieur de chiffres en k -moyennes sans disposer d'aucun label !

Essayons de pousser un peu plus loin, pour le plaisir. Nous pouvons exploiter l'algorithme t-SNE (*t-distributed stochastic neighbor embedding*) dont nous avons cité le nom dans la

présentation de l'apprentissage par variété. Il peut nous permettre de prétraiter les données avant d'appliquer k-moyennes. t-SNE est un algorithme d'intégration non linéaire qui réussit assez bien à préserver les points dans les groupes. Mettons-le à l'ouvrage :

In[17]:

```
from sklearn.manifold import TSNE

# Projection des données (peut demander
# plusieurs secondes)
tsne = TSNE(n_components=2, init='pca',
random_state=0)
digits_proj = tsne.fit_transform(digits.data)

# Calcul des groupes (clusters)
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits_proj)

# Permutation des labels
labels = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    labels[mask] = mode(digits.target[mask])[0]

# Calcul de la précision
accuracy_score(digits.target, labels)
```

Out[17]: 0.93356149137451305

Nous obtenons une précision de classification de presque 94 % sans aucun label. Cela illustre les pouvoirs de l'apprentissage non supervisé lorsqu'il est utilisé à bon escient : vous pouvez ainsi extraire la signification d'un jeu de données lorsque cela devient difficile à réaliser manuellement.

Exemple 2 : k-moyennes et compression de couleurs

Les opérations de partitionnement ou clustering sont particulièrement appréciées lorsqu'il s'agit de comprimer les données de couleurs dans les images. Partons d'une image codée en million de couleurs. Très souvent, la plus grande partie de la gamme de couleurs disponible n'est pas utilisée, car de nombreux pixels de l'image vont utiliser des couleurs identiques ou similaires.

Nous allons travailler sur l'image montrée dans la [Figure 5.120](#) qui est disponible dans le module datasets de Scikit-Learn (pour tester l'exemple, vous devez installer le paquetage de Python nommé pillow) :

```
In[18]: # Note : le paquetage pillow doit être installé
from sklearn.datasets import load_sample_image

china = load_sample_image("china.jpg")
```

```
ax = plt.axes(xticks=[], yticks[])
ax.imshow(china);
```

Les données de l'image sont stockées sous forme d'un tableau à trois dimensions pour la hauteur, la largeur, et le codage RGB. Ce codage correspond aux trois composantes des couleurs primaires rouge, vert et bleu (G = green) sous forme de valeurs entières entre 0 et 255 :

```
In[19]:
china.shape
```

```
Out[19]:
(427, 640, 3)
```

Nous pouvons considérer cette série de pixels comme un nuage de points dans un espace de couleurs en trois dimensions. Nous allons reformer les données vers le format [n_échantillons * n_caractéristiques] puis renormaliser les valeurs de couleurs pour qu'elles soient toutes entre 0 et 1 :

```
In[20]:
data = china / 255.0          # Echelle 0...1
data = data.reshape(427 * 640, 3)
data.shape
```

```
Out[20]: (273280, 3)
```



[Figure 5.120](#) : L'image à traiter.

Nous pouvons visualiser les pixels dans l'espace de couleurs, en nous limitant à 10 000 pixels pour cet essai ([Figure 5.121](#)) :

```
In[21]:  
def plot_pixels(data, title, colors=None,  
N=10000):  
    if colors is None:  
        colors = data  
  
    # Choix d'un sous-ensemble au hasard  
    rng = np.random.RandomState(0)  
    i = rng.permutation(data.shape[0])[:N]  
    colors = colors[i]  
    R, G, B = data[i].T  
  
    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
```

```

    ax[0].scatter(R, G, color=colors, marker='.')
    ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

    ax[1].scatter(R, B, color=colors, marker='.')
    ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

fig.suptitle(title, size=20);

```

In[22]:

```
plot_pixels(data, title='Espace de couleurs en entrée : 16 millions possibles')
```

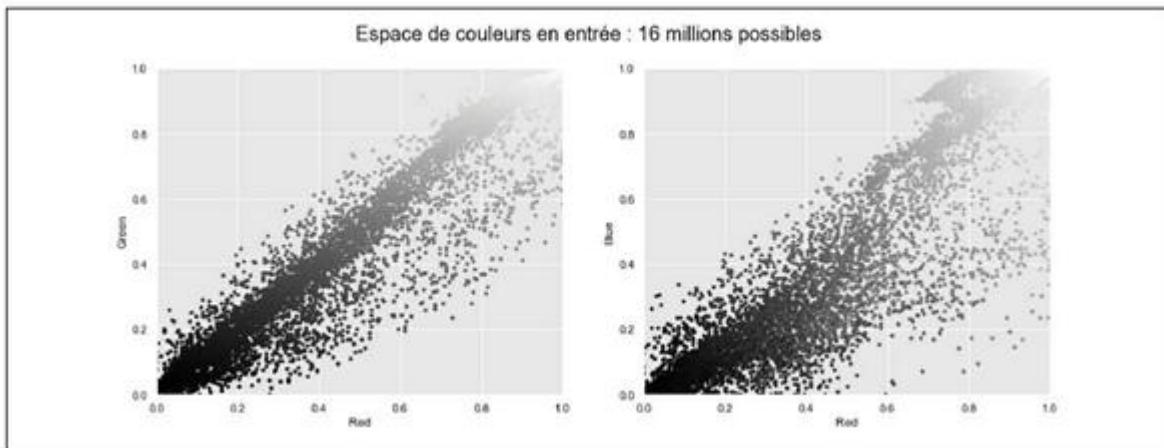


Figure 5.121 : Distribution des pixels dans l'espace de couleurs RGB.

Nous pouvons maintenant compresser ces 16 millions de couleurs en 16 couleurs en nous servant d'un partitionnement en k-moyennes de l'espace des pixels. Le volume de données d'entrée étant énorme, nous allons nous servir de la variante de k-moyennes qui travaille par

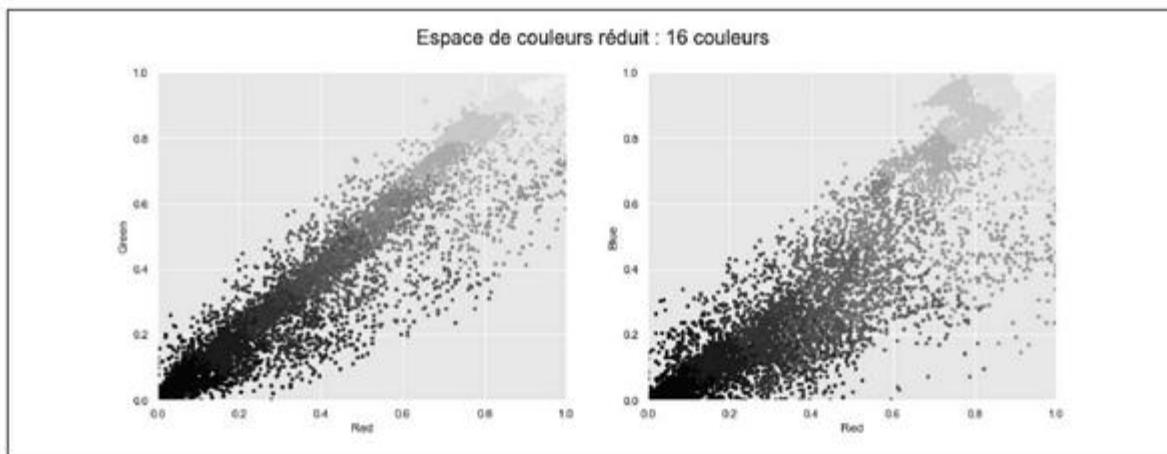
minilots (MiniBatchKMeans). Il traite des sélections de données pour obtenir le résultat bien plus vite que l'algorithme en k-moyennes standard ([Figure 5.122](#)) :

In[23]:

```
from sklearn.cluster import MiniBatchKMeans

kmeans = MiniBatchKMeans(16)
kmeans.fit(data)
new_colors =
kmeans.cluster_centers_[kmeans.predict(data)]

plot_pixels(data, colors=new_colors,
            title="Espace de couleurs réduit : 16
couleurs")
```

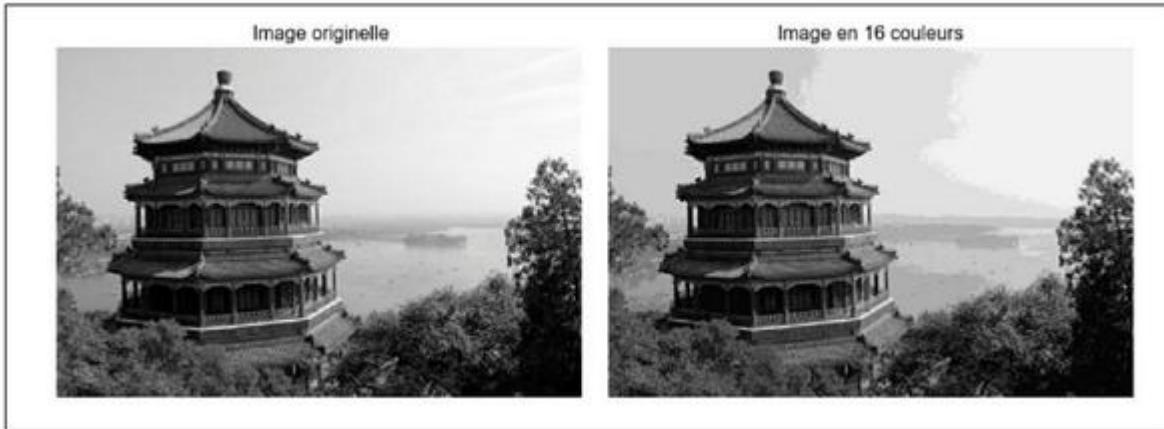


[Figure 5.122](#) : 16 groupes dans l'espace de couleurs RGB.

Le résultat est une image simplifiée au niveau de la richesse des nuances. Chaque pixel reçoit la couleur du centre de groupe le plus proche. Nous pouvons juger du résultat en

faisant tracer l'image avec les nouvelles couleurs dans l'espace d'image et non dans celui des pixels ([Figure 5.123](#)) :

```
In[24]:  
china_recolored =  
new_colors.reshape(china.shape)  
  
fig, ax = plt.subplots(1, 2, figsize=(16, 6),  
subplot_kw=dict(xticks=[], yticks=[]))  
fig.subplots_adjust(wspace=0.05)  
  
ax[0].imshow(china)  
ax[0].set_title('Image originelle', size=16)  
ax[1].imshow(china_recolored)  
ax[1].set_title('Image en 16 couleurs',  
size=16);
```



[Figure 5.123](#) : Comparaison de l'image de départ à gauche et de l'image en 16 couleurs à droite.

En observant de près, on constate que certains détails sont perdus, mais l'image reste tout à fait reconnaissable. Il faut savoir que l'image de droite est le résultat d'une compression d'environ un million ! Il s'agit d'une application intéressante des k-moyennes. Il existe des techniques plus appropriées pour compresser des images, mais cet exemple suffit à montrer comment on peut trouver de nouveaux domaines d'application pour une méthode non supervisée telle que celle des k-moyennes.

5.12 : Modèles de mélange gaussien (GMM)

Le modèle de partitionnement en k-moyennes que nous venons de voir est assez simple à comprendre et à utiliser, mais cette simplicité se paie en défi à relever pour l'utiliser. Sa nature non probabiliste, et son utilisation de la distance par rapport au centre de chaque groupe pour répartir les membres a pour conséquence de mauvaises performances dans de nombreuses situations réelles. Découvrions donc les modèles de mélange gaussien (GMM) qui sont d'une certaine façon une extension du principe des k-moyennes, tout en permettant de puissantes opérations d'estimation allant au-delà du partitionnement. Nous commençons par nos imports :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()  
import numpy as np
```

GMM contre les faiblesses des k-moyennes

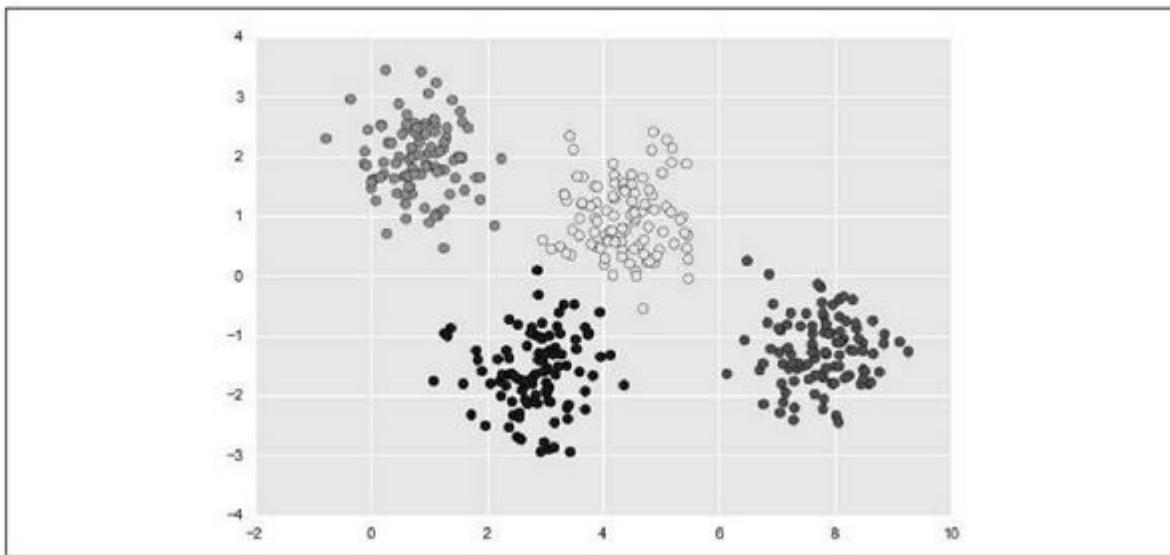
Revenons sur certaines faiblesses de l'algorithme en k-moyennes afin d'envisager des pistes d'amélioration de ce partitionnement. Nous avons vu que les résultats étaient corrects si les données d'entrée étaient bien séparées.

Si nous devons par exemple traiter des agrégats de données, l'algorithme en k-moyennes trouve rapidement des labels pour les différents groupes, avec un résultat assez fidèle à notre propre vérification visuelle ([Figure 5.124](#)) :

```
In[2]: # Génération de données
from sklearn.datasets import make_blobs
X, y_true = make_blobs(n_samples=400, centers=4,
                      cluster_std=0.60,
                      random_state=0)
# Inversion des axes pour aider au tracé
X = X[:, ::-1]
```

```
In[3]: # Tracé avec labels de k-moyennes
from sklearn.cluster import KMeans
kmeans = KMeans(4, random_state=0)
labels = kmeans.fit(X).predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40,
cmap='viridis');
```

Le simple bon sens nous fait supposer que l'attribution des groupes sera plus efficace pour certains points que pour d'autres. Dans l'exemple, il y a un léger chevauchement entre les deux groupes du milieu, ce qui nous fait douter de la qualité de l'affectation dans cette zone. Le souci est que le modèle en k-moyennes n'offre aucune possibilité de mesurer la probabilité ou l'incertitude des affectations qu'il réalise (il reste possible d'utiliser une approche autorebouclante – *bootstrap*, pour juger de l'incertitude). Il nous faut donc généraliser le modèle.



[Figure 5.124](#) : Labels de k-moyennes pour des données simples.

Le mécanisme du modèle en k-moyennes peut s'imaginer comme suit : il positionne un cercle (ou une hypersphère dans un nombre de dimensions supérieur) au centre de chaque groupe, le rayon du cercle correspondant au point le

plus éloigné du centre tout en appartenant à ce groupe. Le rayon constitue donc une limite franche pour appartenir ou pas à un groupe dans le jeu d'entraînement : un point extérieur au cercle n'est pas membre du groupe. Nous pouvons visualiser le modèle correspondant avec la fonction que nous définissons ci-dessous ([Figure 5.125](#)) :

In[4] :

```
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

def plot_kmeans(kmeans, X, n_clusters=4,
rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # Trace les données d'entrée
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40,
cmap='viridis', zorder=2)

    # Trace les représentations du modèle k-
    # moyennes
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i],
[center]).max()
        for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r,
```

```
fc='#CCCCCC', lw=3,  
alpha=0.5, zorder=1))
```

```
In[5] :  
kmeans = KMeans(n_clusters=4, random_state=0)  
plot_kmeans(kmeans, X)
```

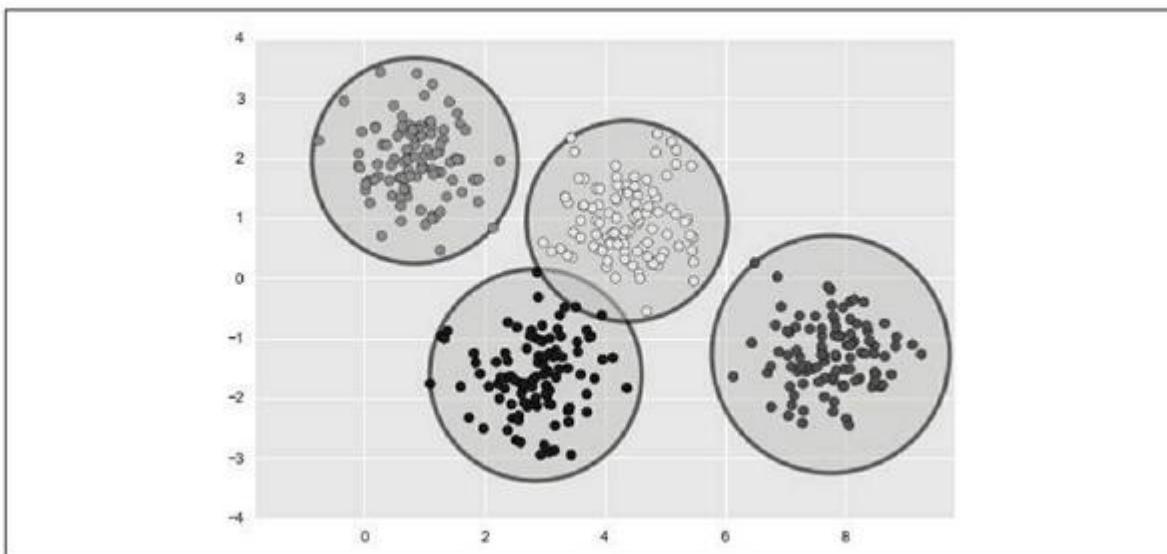


Figure 5.125 : Les cercles délimitant les groupes du modèle en k-moyennes.

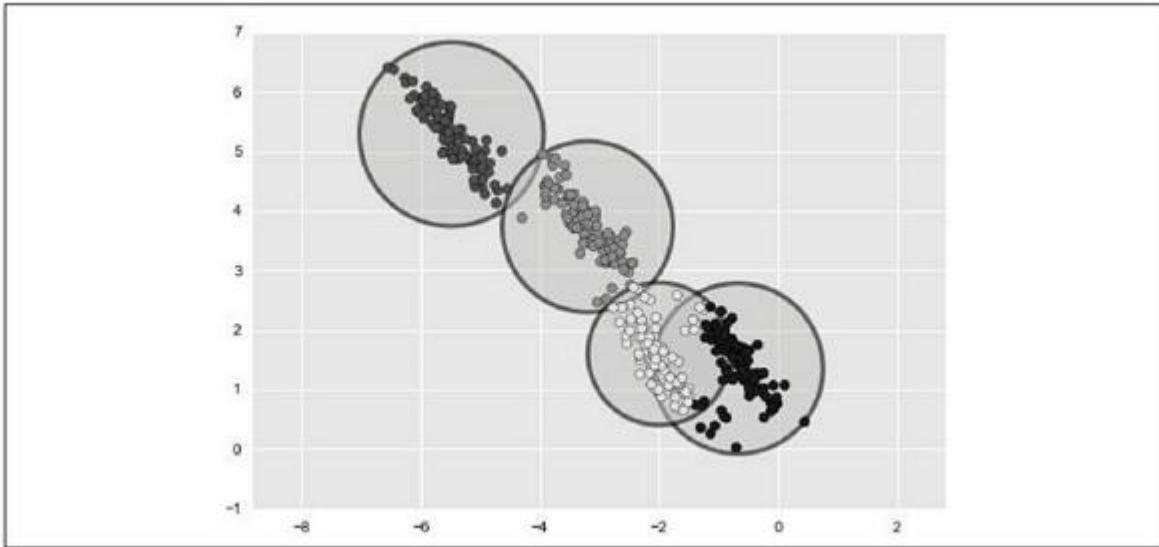
Un point essentiel est que k-moyennes ne peut utiliser que des cercles pour les groupes. Il est impossible d'imaginer un groupe avec une zone elliptique. Si nous nous amusons à reformer les données d'entrée, les affectations de groupe deviennent presque inexploitables ([Figure 5.126](#)) :

```
In[6]:  
rng = np.random.RandomState(13)  
X_stretched = np.dot(X, rng.randn(2, 2))
```

```
kmeans = KMeans(n_clusters=4, random_state=0)
plot_kmeans(kmeans, X_stretched)
```

L'image montre que les groupes ne sont pas circulaires, mais plutôt alignés selon un axe, et ne conviennent donc pas à cet algorithme. Pourtant, k-moyennes va tenter de forcer l'appartenance à un groupe plutôt qu'à un autre pour chaque point. C'est à cause de cela que les cercles se chevauchent, notamment du côté droit en bas. On pourrait être tenté de prétraiter les données par exemple avec une analyse PCA, mais rien ne garantit qu'un tel formatage global rendra les données individuelles mieux réparties en cercles, tout en restant assez distinctes.

L'algorithme en k-moyennes souffre donc de deux vrais points faibles qui peuvent vous empêcher de bien en profiter : son manque de souplesse quant à la forme des régions et l'absence de possibilité d'affecter les groupes de façon probabiliste.



[Figure 5.126](#) : Mauvaise performance de k-moyennes pour des groupes non circulaires.

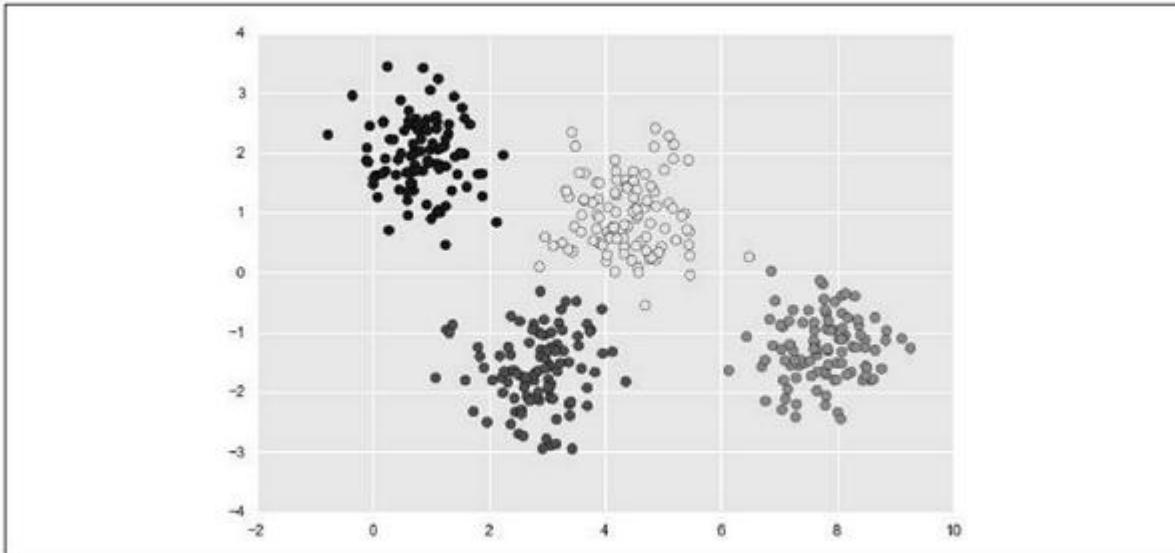
Pour pallier ces faiblesses, on pourrait imaginer de généraliser le modèle en k-moyennes en mesurant l'incertitude des affectations au groupe, en comparant les distances de chaque point à tous les centres de groupes, et pas seulement aux plus proches. On peut également imaginer de faire des frontières de groupes en ellipse pour pouvoir gérer des groupes de données non circulaires. Il se trouve que ces deux stratégies sont deux composants fondamentaux d'un autre modèle de partitionnement : le modèle de mélange gaussien.

Généralisation E-M : modèles de mélange gaussien

Le modèle de mélange gaussien GMM (*Gaussian Mixture Model*) cherche la combinaison ou le mélange de distribution de probabilités gaussiennes à plusieurs dimensions qui offre le meilleur ajustement du modèle aux données d'entrée. Une utilisation simplifiée de GMM permet par exemple, comme l'approche par k-moyennes, de trouver des groupes (ou clusters) ([Figure 5.127](#)) :

In[7] :

```
from sklearn.mixture import GMM
gmm = GMM(n_components=4).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40,
cmap='viridis');
```



[Figure 5.127](#) : Labels de données trouvés par le modèle GMM.

Mais GMM peut faire mieux : du fait qu'il contient un modèle probabiliste, il permet de trouver des groupes de façon probabiliste, dans Scikit-Learn au moyen de la méthode `predict_proba()`. Cette méthode renvoie une matrice sous le format `[n_samples, n_clusters]` qui permet de connaître la probabilité pour chaque point d'appartenir au groupe concerné :

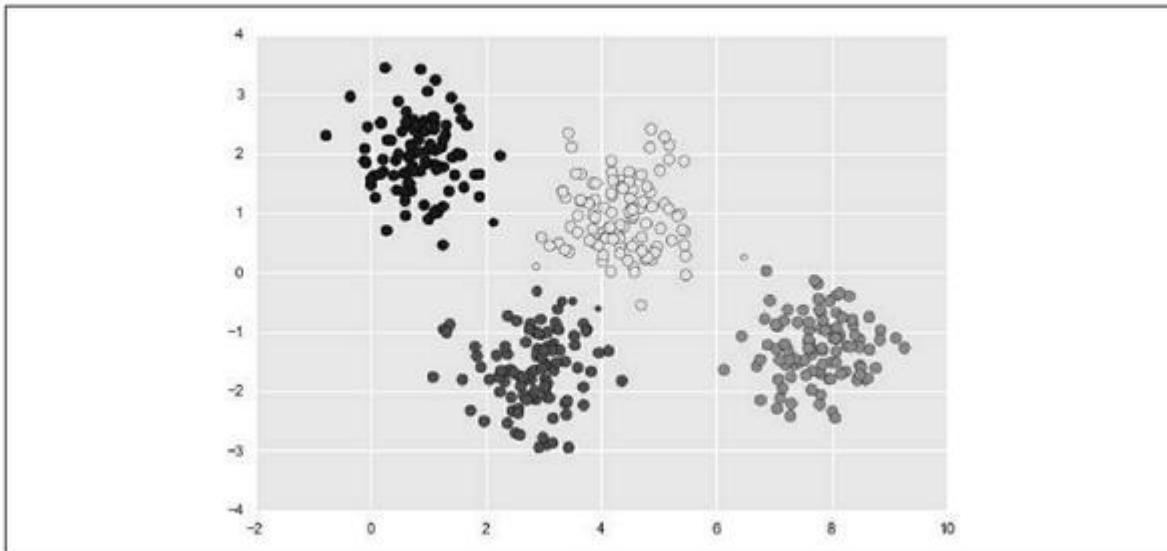
```
In[0]:  
probs = gmm.predict_proba(X)  
print(probs[:5].round(3))
```

```
[[0.469 0.    0.    0.531]  
 [0.    0.    1.    0.    ]  
 [0.    0.    1.    0.    ]
```

```
[0.    0.    0.    1.    ]  
[0.    0.    1.    0.    ]]
```

On peut rendre visible le degré d'incertitude en faisant jouer sur le diamètre des points. La [Figure 5.128](#) confirme que ce sont les points sur les frontières entre groupes qui ont le plus haut degré d'incertitude d'affectation :

```
In[9]: # L'élévation au carré amplifie les  
différences  
size = 50 * probs.max(1) ** 2  
plt.scatter(X[:, 0], X[:, 1], c=labels,  
cmap='viridis', s=size);
```



[Figure 5.128](#) : Labels probabilistes de GMM ; le diamètre du point est proportionnel à sa certitude.

Le modèle GMM est techniquement proche du modèle en k-moyennes puisqu'il utilise la technique d'Espérance-

Maximisation :

1. Estimation initiale de position et de forme.
2. Répétition de deux étapes jusqu'à convergence.
 - a. Étape E : pour chaque point, recherche du poids correspondant à la probabilité d'appartenance à chaque groupe.
 - b. Étape M : pour chaque groupe, mise à jour de la position, normalisation et formatage à partir de tous les points de données en exploitant les poids calculés.

Grâce à cette approche, les frontières de chaque groupe ne sont plus des cercles aux limites abruptes, mais suivent un modèle gaussien adouci. Parfois, l'algorithme peut ne pas aboutir à la solution optimale au niveau global (comme les k-moyennes), ce qui oblige à réaliser plusieurs initialisations avec un départ aléatoire.

Nous pouvons définir une fonction pour nous aider à visualiser les positions et les formes des groupes GMM en ajoutant des ellipses basées sur la sortie de GMM :

In[10]:

```
from matplotlib.patches import Ellipse  
  
def draw_ellipse(position, covariance, ax=None,
```

```

**kwargs):
    """Trace une ellipse selon position et
covariance"""
    ax = ax or plt.gca()

    # Convertit la covariance en axes principaux
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0],
U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Trace l'ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig *
width, nsig * height,
                           angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels,
s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40,

```

```
zorder=2)
ax.axis('equal')

w_factor = 0.2 / gmm.weights_.max()
for pos, covar, w in zip(gmm.means_,
gmm.covariances_, gmm.weights_):
    draw_ellipse(pos, covar, alpha=w *
w_factor)
```

Nous pouvons maintenant voir à quoi ressemble le traitement de notre GMM à quatre composantes sur nos données initiales ([Figure 5.129](#)) :

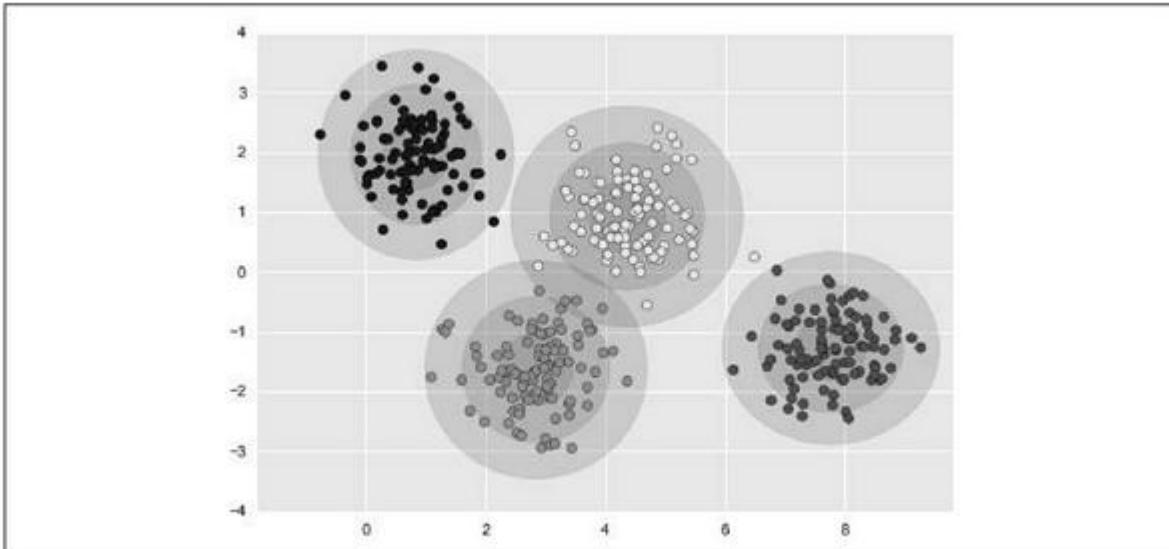
In[11]:

```
gmm = GMM(n_components=4, random_state=42)
plot_gmm(gmm, X)
```

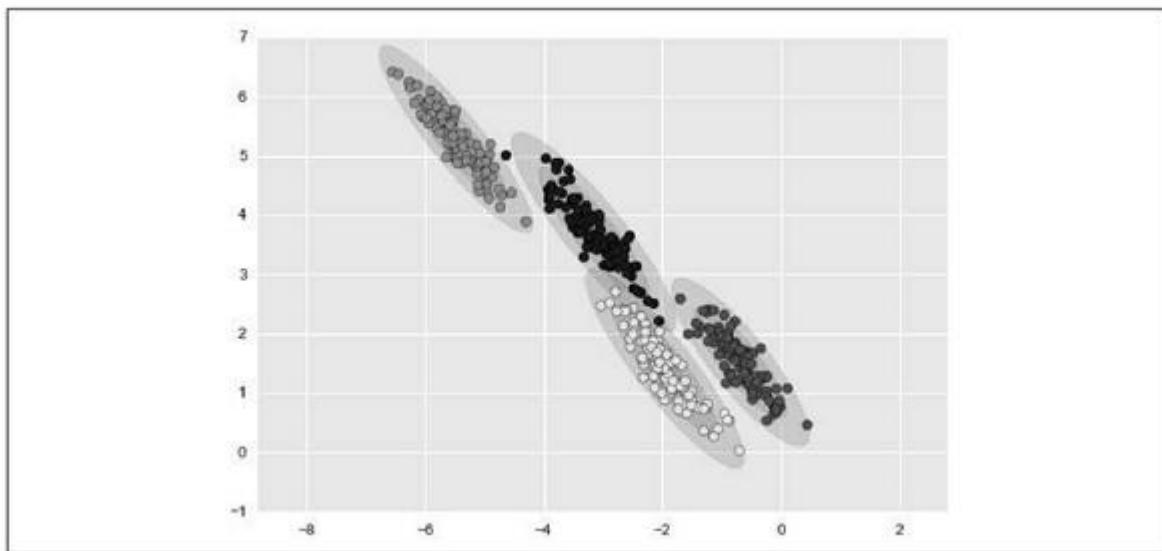
L'approche GMM saura traiter correctement le jeu de données en élongation. En fournissant la valeur full pour le paramètre covariance, le modèle va pouvoir s'ajuster à des groupes tout en longueur ([Figure 5.130](#)) :

In[12]:

```
gmm = GMM(n_components=4,
covariance_type='full', random_state=42)
plot_gmm(gmm, X_
stretched)
```



[Figure 5.129](#) : Visualisation du modèle GMM à quatre composantes, avec groupes circulaires.



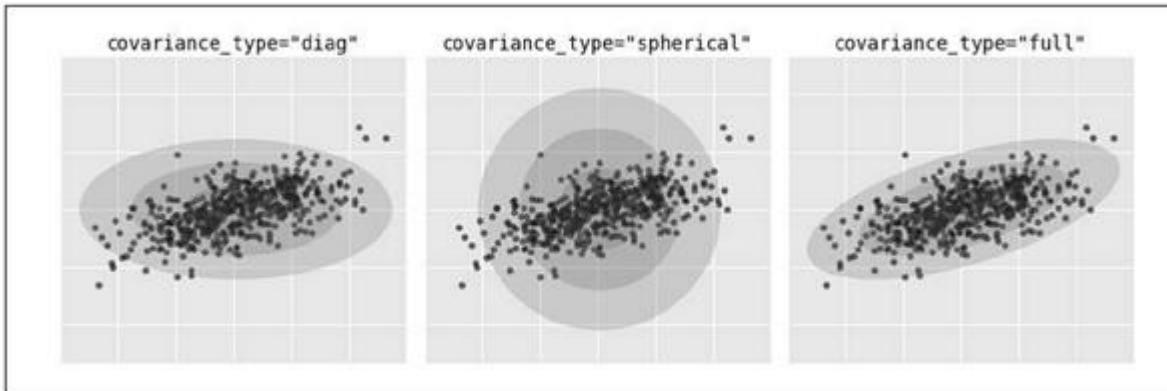
[Figure 5.130](#) : Résultat du modèle GMM à quatre composantes avec des groupes non circulaires.

Le modèle GMM résout donc correctement les deux points faibles principaux mentionnés à propos des k-moyennes.

Choix du type de covariance

L'hyperparamètre de covariance détermine le nombre de degrés de liberté de la forme de chaque groupe. Il correspond à l'option `covariance_type` qui varie selon le type de problème à résoudre et doit être choisi avec soin. La valeur par défaut, donc utilisée dans le premier exemple, correspond à `covariance_type='diag'`. Dans ce cas, vous pouvez choisir indépendamment la taille de chaque groupe dans chaque dimension, l'ellipse résultante devant s'aligner selon les axes. Le choix `covariance_type="spherical"` demande d'utiliser un modèle plus simple dans lequel la forme du groupe est de type circulaire (toutes les dimensions sont égales). Le résultat est proche des k-moyennes, mais pas exactement équivalent. Enfin, la covariance complète, `covariance_type="full"` permet de modéliser chaque groupe sous forme d'une ellipse d'orientation arbitraire, mais le modèle demande des calculs intensifs, surtout si le nombre de dimensions est important.

La figure suivante permet de comparer les trois choix de covariance pour un même groupe ([Figure 5.131](#)).



[Figure 5.131](#) : Visualisation des types de covariance GMM.

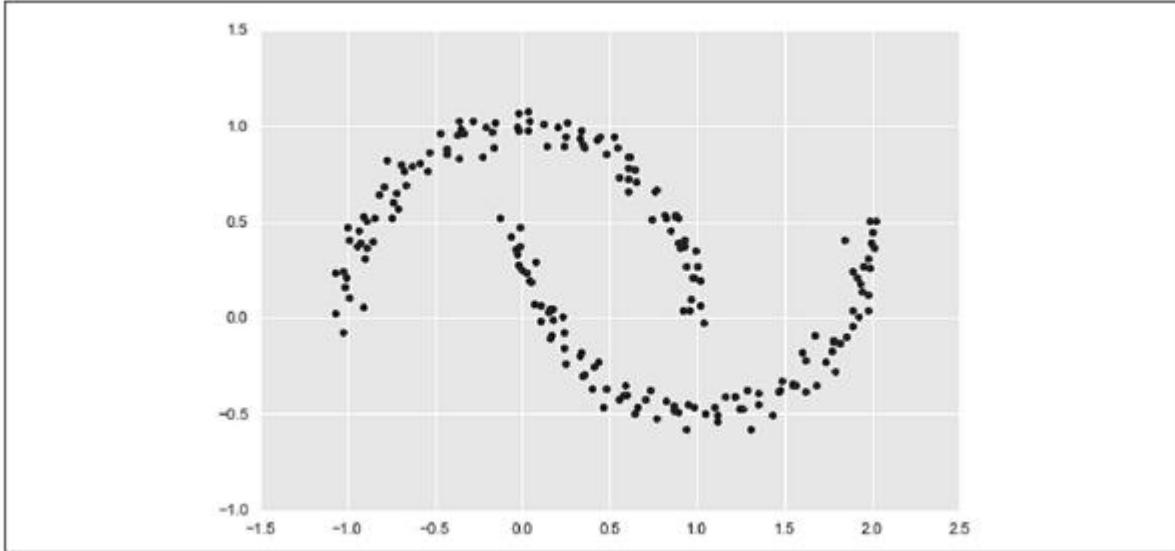
GMM et estimation de densité

On présente généralement GMM en tant qu'algorithme de partitionnement, mais c'est d'abord un algorithme d'*estimation de densité*. Le résultat produit par GMM dans son ajustement ne correspond pas techniquement à un modèle de partitionnement, mais plutôt à un modèle probabiliste générateur qui sert à décrire la distribution des données.

Partons de quelques données générées à partir de la fonction de Scikit-Learn nommée `make_moons()` ([Figure 5.132](#)). Nous l'avons déjà rencontrée dans la section dédiée aux k-moyennes :

In[13]:

```
from sklearn.datasets import make_moons
Xmoon, ymoon = make_moons(200, noise=.05,
                           random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

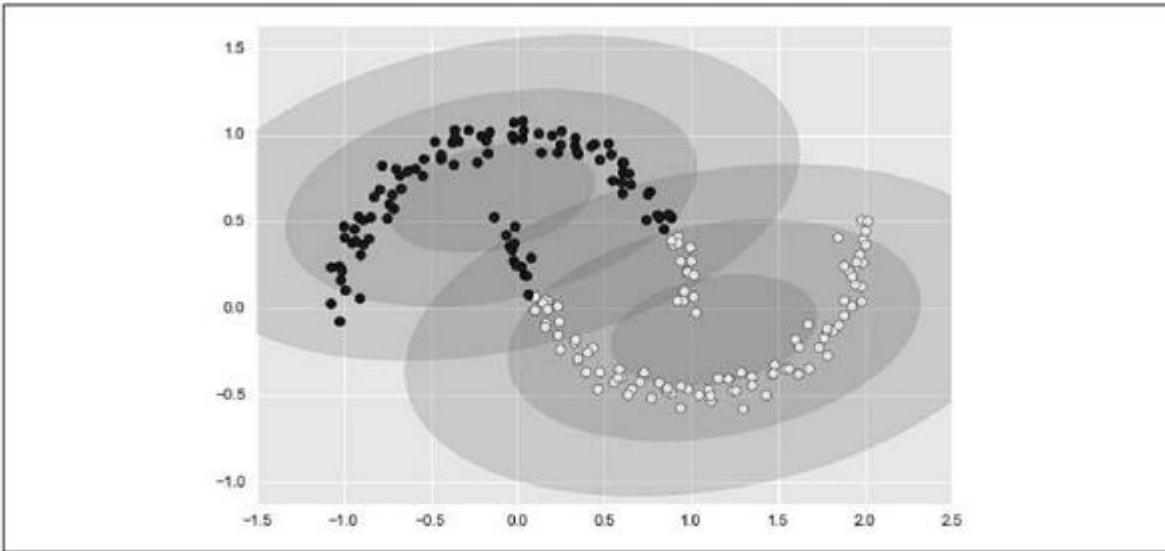


[Figure 5.132](#) : GMM sur des groupes à frontières non linéaires.

Si nous tentons d'ajuster ce résultat à un GMM à deux composantes considéré comme un modèle de partitionnement, nous n'allons pas obtenir de résultats très intéressants ([Figure 5.133](#)) :

In[14]:

```
gmm2 = GMM(n_components=2,  
covariance_type='full', random_state=@)  
plot_gmm(gmm2, Xmoon)
```

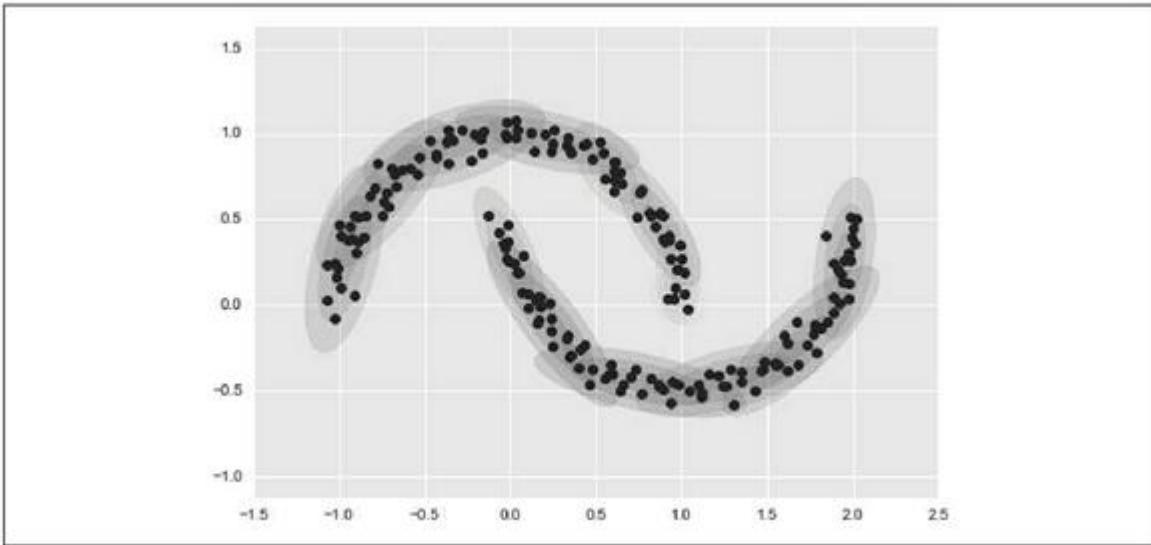


[Figure 5.133](#) : Ajustement d'un GMM à deux composantes à ces groupes non linéaires.

Nous obtenons un ajustement qui épouse mieux les données d'entrée si nous utilisons un nombre de composantes plus grand et ignorons les labels des groupes ([Figure 5.134](#)) :

In[15] :

```
gmm16 = GMM(n_components=16,  
covariance_type='full', random_state=@)  
plot_gmm(gmm16,  
Xmoon, label=False)
```



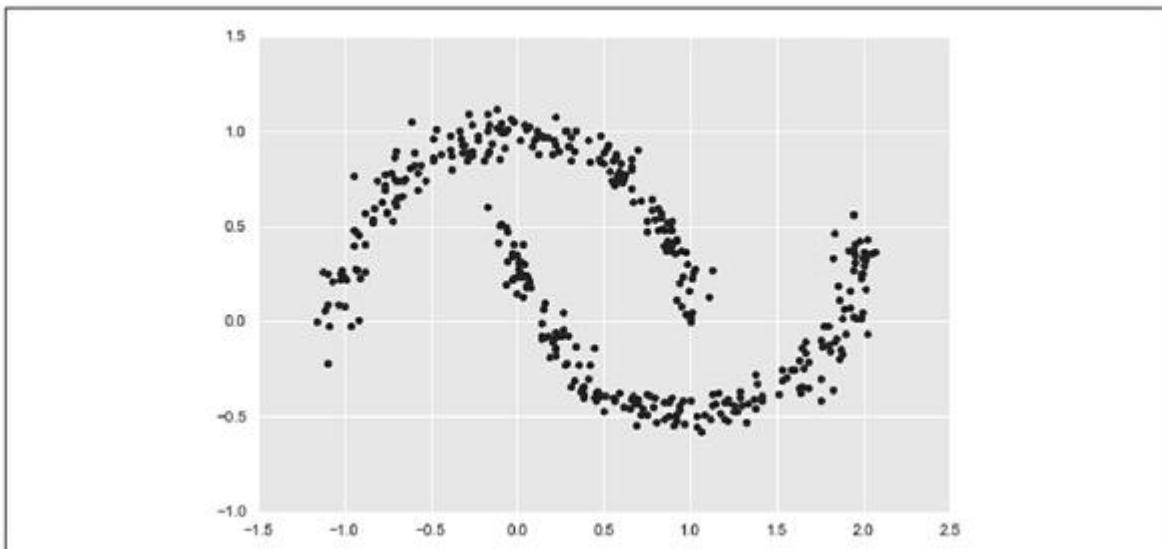
[Figure 5.134](#) : Modélisation efficace de la distribution des points avec GMM à grand nombre de composantes.

Dans cet exemple, le mélange de 16 gaussiennes ne nous sert pas à trouver les délimitations des groupes de données, mais à modéliser la distribution générale des données d'entrée. Il s'agit d'un modèle générateur de la distribution ; GMM nous procure une recette pour générer de nouvelles données au hasard, mais selon une distribution similaire aux données fournies. Voici par exemple 400 nouveaux points produits par ce modèle GMM à 16 composantes ajusté par rapport à nos données ([Figure 5.135](#)) :

In[16]:

```
Xnew = gmm16.sample(400, random_state=42)
plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```

GMM constitue ainsi une technique adaptable pour modéliser une distribution de données arbitraire à plusieurs dimensions.



[Figure 5.135](#) : Nouvelles données générées par le GMM à 16 composantes.

Combien de composantes ?

Puisque le modèle GMM est générateur, il nous fournit naturellement une solution pour connaître le nombre optimal de composantes en fonction du jeu de données. Puisqu'il s'agit d'une distribution de probabilités du jeu d'entrée, nous pouvons évaluer la vraisemblance des données avec ce modèle en prenant soin d'éviter tout surajustement par une validation croisée. Une autre façon de corriger le surajustement consiste à retoucher la vraisemblance en utilisant un outil analytique tel que le critère d'information Akaike (AIC,

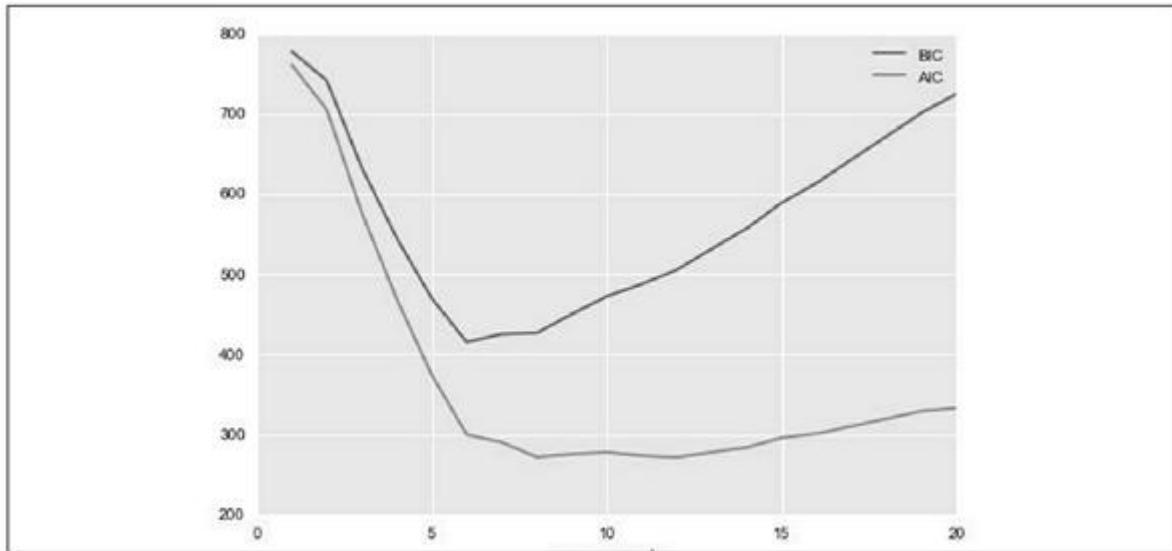
https://fr.wikipedia.org/wiki/Critère_d'information_d'Akaike) ou le critère d'information bayésien (BIC, https://fr.wikipedia.org/wiki/Critère_d'information_bayésien). Ces deux techniques sont faciles à utiliser, puisque l'estimateur GMM de Scikit-Learn définit des méthodes pour calculer ces deux critères.

Demandons à voir les critères AIC et BIC en fonction du nombre de composantes GMM pour notre jeu de données de lune ([Figure 5.136](#)) :

```
In[17]:  
n_components = np.arange(1, 21)  
models = [GMM(n, covariance_type='full',  
random_state=0).fit(Xmoon)  
          for n in n_components]  
plt.plot(n_components, [m.bic(Xmoon) for m in  
models], label='BIC')  
plt.plot(n_components, [m.aic(Xmoon) for m in  
models], label='AIC')  
plt.legend(loc='best')  
plt.xlabel('n_components');
```

Il suffit alors d'utiliser la valeur minimale de AIC ou de BIC selon l'approximation à utiliser pour trouver le nombre de groupes optimal. Selon AIC, la valeur de 16 composantes était trop élevée ; il aurait mieux valu choisir entre 8 et 12 composantes. Le critère BIC suggère d'adopter

un modèle plus simple, ce qui est typique dans ce genre de contexte.



[Figure 5.136](#) : Affichage des critères AIC et BIC afin de choisir le nombre de composantes de GMM.

Un point important : le choix du nombre de composantes détermine la qualité du traitement de GMM en tant qu'estimateur de densité pas en tant qu'algorithme de partitionnement (*clustering*). Je vous conseille de toujours considérer GMM comme un estimateur de densité, et de ne vous en servir pour des partitionnements qu'une fois vos tests de pertinence réalisés avec des jeux de données plus légers.

Exemple : GMM pour générer des données

Nous venons de voir comment utiliser GMM comme modèle générateur, afin de créer de nouveaux échantillons fondés sur la distribution détectée dans les données d'entrée. Mettons cette idée en pratique en faisant générer de nouveaux chiffres manuscrits, inspirés du lot de chiffres déjà utilisé.

Nous commençons bien sûr par charger les chiffres manuscrits au moyen des outils de Scikit-Learn :

In[18]:

```
from sklearn.datasets import load_digits  
digits = load_digits()  
digits.data.shape
```

Out[18]: (1797, 64)

Demandons de visualiser les 100 premiers chiffres pour nous souvenir de ce que nous manipulons ([Figure 5.137](#)) :

In[19]:

```
def plot_digits(data):  
    fig, ax = plt.subplots(10, 10, figsize=(8,  
8),  
    subplot_kw=dict(xticks=[], yticks=[]))
```

```
fig.subplots_adjust(hspace=0.05, wspace=0.05)
for i, axi in enumerate(ax.flat):
    im = axi.imshow(data[i].reshape(8, 8),
cmap='binary')
    im.set_clim(0, 16)

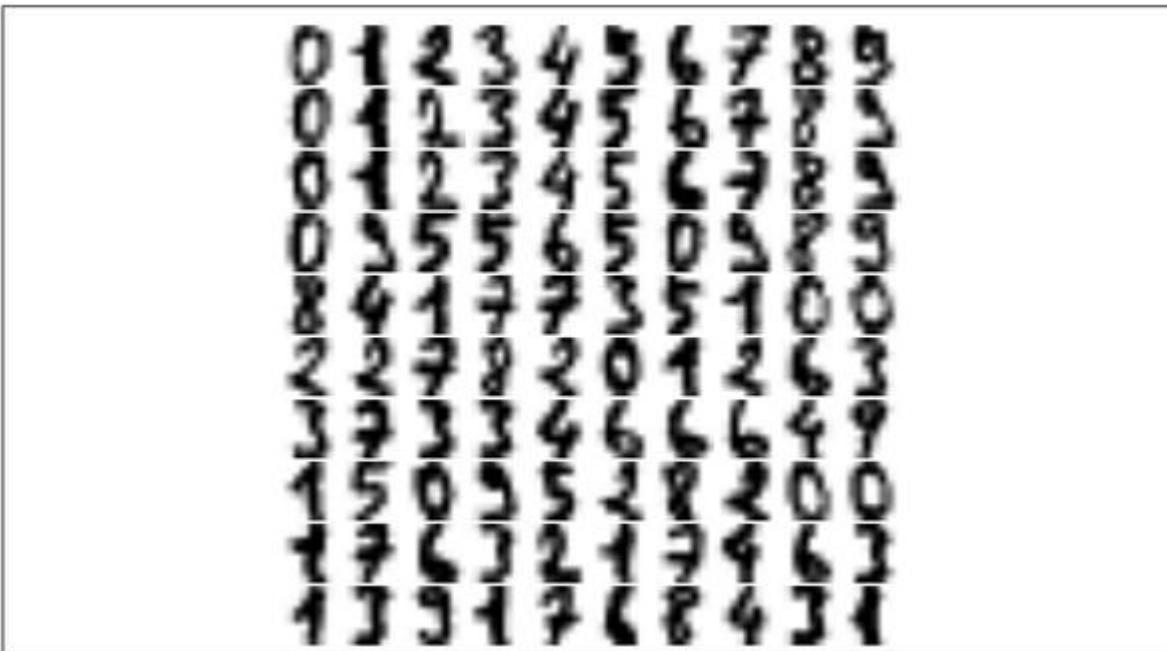
plot_digits(digits.data)
```

Nous partons d'environ 1 800 chiffres en 64 dimensions. Nous pouvons ajouter un modèle GMM pour en générer d'autres. Du fait que GMM pourra avoir du mal à réussir la convergence lorsqu'il y a un grand nombre de dimensions, nous commençons par appliquer un algorithme de réduction dimensionnelle inversible. Nous optons pour un PCA en lui demandant de préserver 99 % de la variance dans les données projetées :

In[20]:

```
from sklearn.decomposition import PCA
pca = PCA(0.99, whiten=True)
data = pca.fit_transform(digits.data)
data.shape
```

Out[20]: (1797, 41)



[Figure 5.137](#) : Chiffres manuscrits des données d'entrée.

Il nous reste 41 dimensions, ce qui est une réduction d'un tiers, quasiment sans perte d'information. Nous exploitons ces données projetées avec le critère AIC pour obtenir une idée du nombre de composantes GMM à utiliser ([Figure 5.138](#)) :

In[21]:

```
n_components = np.arange(50, 210, 10)
models = [GMM(n, covariance_type='full',
random_state=0)
          for n in n_components]
aics = [model.fit(data).aic(data) for model in
models]
plt.plot(n_components, aics);
```

Nous apprenons ainsi que nous pouvons réduire l'AIC avec 110 composantes. Nous utiliserons le modèle correspondant. Demandons un ajustement aux données pour pouvoir confirmer que la convergence a eu lieu :

In[22]:

```
gmm = GMM(110, covariance_type='full',
random_state=0)
gmm.fit(data)
print(gmm.converged_)
```

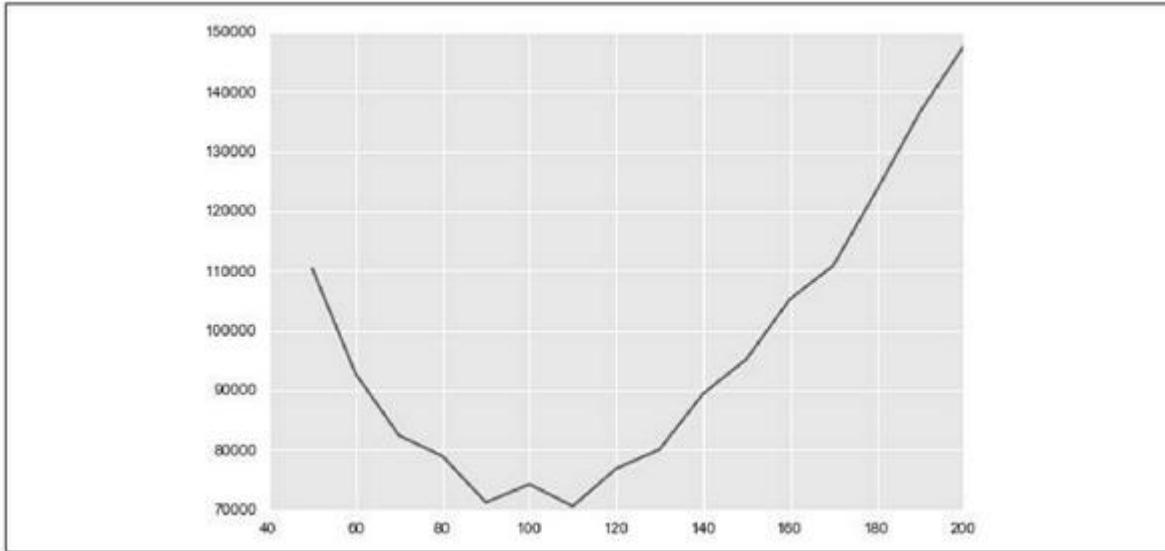
True

Nous pouvons maintenant afficher les échantillons de 100 nouveaux points dans l'espace projeté à 41 dimensions, GMM servant de modèle générateur :

In[23]:

```
data_new = gmm.sample(100, random_state=0)
data_new.shape
```

Out[23]: (100, 41)



[Figure 5.138](#) : Courbe AIC pour décider du nombre idéal de composantes GMM.

Il ne reste plus qu'à nous servir de la transformation inverse de l'objet PCA pour faire construire les nouveaux chiffres ([Figure 5.139](#)) :

In[24]:

```
digits_new = pca.inverse_transform(data_new)  
plot_digits(digits_new)
```

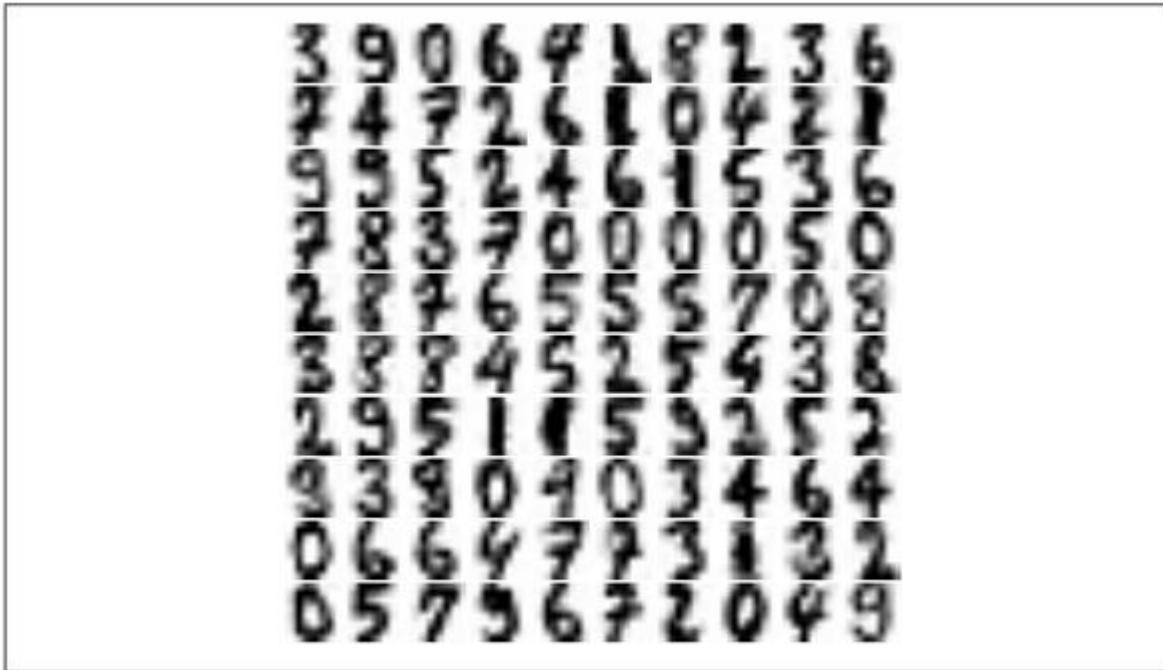


Figure 5.139 : « Nouveaux » chiffres produits à partir du modèle de l'estimateur GMM.

Le résultat montre des chiffres de synthèse qui ressemblent à des vrais !

Reformulons les étapes du traitement : en partant d'une sélection de chiffres manuscrits, nous avons produit le modèle de distribution des données de telle manière qu'il devient ensuite possible de créer de nouveaux échantillons à partir des données. La plupart de ces chiffres « de synthèse » peuvent être confondus avec des vrais chiffres manuscrits !

L'algorithme utilisé permet donc de capturer les caractéristiques générales des données d'entrée telles que modélisées par le modèle de mélange. Ce modèle générateur

peut se montrer très utile dans le cadre d'un classifieur génératif bayésien, ce que nous allons voir dans la prochaine section.

5.13 : Estimation par noyau

Les modèles de mélange gaussien (GMM) que nous venons de voir sont une sorte de combinaison d'un estimateur de partitionnement et d'un estimateur de densité. Rappelons qu'un estimateur de densité part d'un jeu de données d'entrée à D -dimensions pour produire une estimation de la distribution de probabilités en D -dimensions qui semble avoir produit ces données. Cette estimation est obtenue en représentant la densité sous forme d'une somme pondérée de distributions gaussiennes. L'algorithme d'estimation par noyau KDE (*Kernel density estimation*) est en quelque sorte une application systématique de la technique du mélange de gaussiennes : le mélange qu'il utilise comporte une composante gaussienne par point, ce qui produit un estimateur de densité pour l'essentiel non paramétrique. Découvrons la raison d'être et les domaines d'emploi de KDE. Nous commençons bien sûr par les imports indispensables :

```
In[1]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn as sns; sns.set()  
import numpy as np
```

Raison d'être de KDE : histogrammes

Nous savons qu'un estimateur de densité est un algorithme dont le but est de modéliser la distribution des probabilités ayant généré un jeu de données. Vous connaissez déjà un estimateur de densité très simple dans le cas des données en une dimension : il s'agit de l'histogramme. L'histogramme répartit les données en compartiments ou bacs (appelés aussi maladroitement « classes »). Il dénombre les points dans chaque bac et visualise le résultat de façon très intuitive.

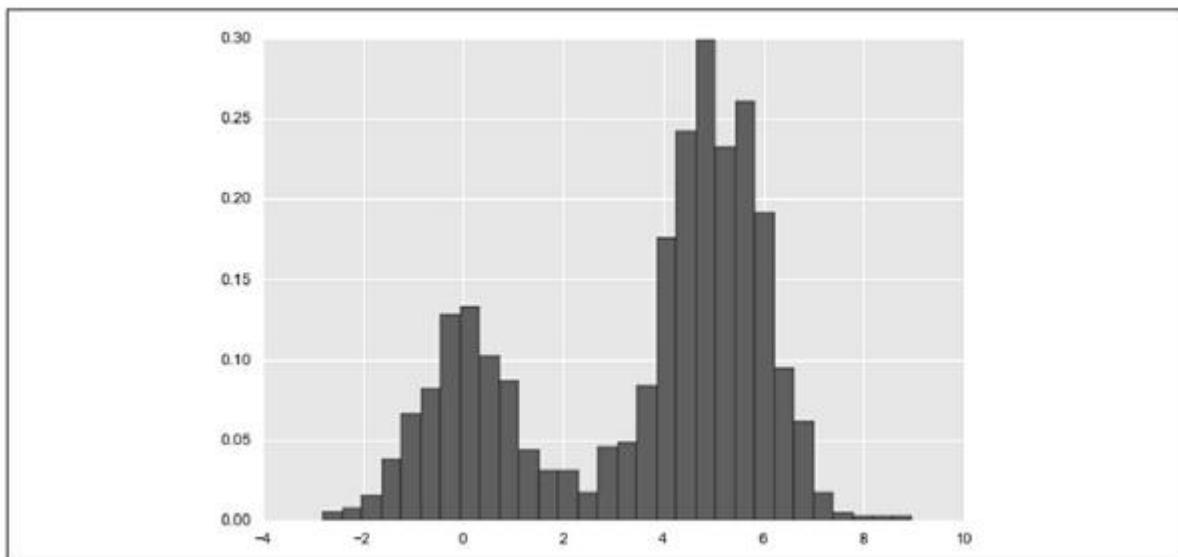
Générons quelques données à partir de deux distributions normales :

```
In[2]:  
def make_data(N, f=0.3, rseed=1):  
    rand = np.random.RandomState(rseed)  
    x = rand.randn(N)  
    x[int(f * N):] += 5  
    return x  
  
x = make_data(1000)
```

Nous avons vu que les histogrammes à dénombrement standard peuvent être créés au moyen de la fonction plt.hist(). Nous activons l'option de densité au moyen du

paramètre `density` pour obtenir un histogramme normalisé dans lequel les hauteurs des barres ne correspondent pas aux quantités mais aux densités de probabilité ([Figure 5.140](#)) :

```
In[3]: hist = plt.hist(x, bins=30, density=True)
```



[Figure 5.140](#) : Données générées par combinaison de deux distributions normales.

Sachez que pour égaliser la taille des classes ou bacs, la normalisation qui est utilisée se contente d'ajuster l'échelle de l'axe des ordonnées y. Les hauteurs relatives sont de ce fait les mêmes que dans un histogramme construit à partir des quantités. Cette normalisation a été choisie de sorte que la surface totale sous l'histogramme soit égale à 1, ce que nous pouvons confirmer en demandant d'obtenir la sortie de la fonction histogramme :

```
In[4]:
```

```
density, bins, patches = hist  
widths = bins[1:] - bins[:-1]  
(density * widths).sum()
```

```
Out[4]: 1.0
```

Un point à surveiller lorsqu'on utilise un histogramme pour estimer une densité est le suivant : le choix de la taille des classes et de leur position entraîne des représentations dont les résultats sont qualitativement très différents. Si nous travaillons par exemple avec les mêmes données mais avec seulement 20 points, les choix faits pour tracer les bacs vont donner une interprétation tout à fait différente ! Voyons cela par un exemple ([voir la Figure 5.141](#)) :

```
In[5]:
```

```
x = make_data(20)  
bins = np.linspace(-5, 10, 10)
```

```
In[6]:
```

```
fig, ax = plt.subplots(1, 2, figsize=(12, 4),  
                      sharex=True, sharey=True,  
                      subplot_kw={'xlim':(-4,  
9),  
                      'ylim':(-0.02,  
0.3)})  
fig.subplots_adjust(wspace=0.05)  
for i, offset in enumerate([0.0, 0.6]):
```

```

    ax[i].hist(x, bins=bins + offset,
normed=True)
    ax[i].plot(x, np.full_like(x, -0.01), '|k',
markeredgecolor='black')

```

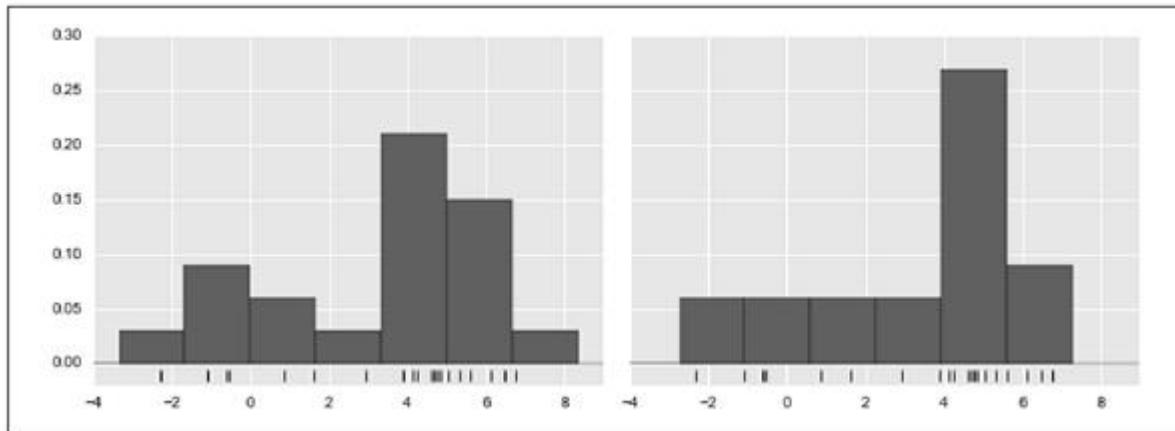


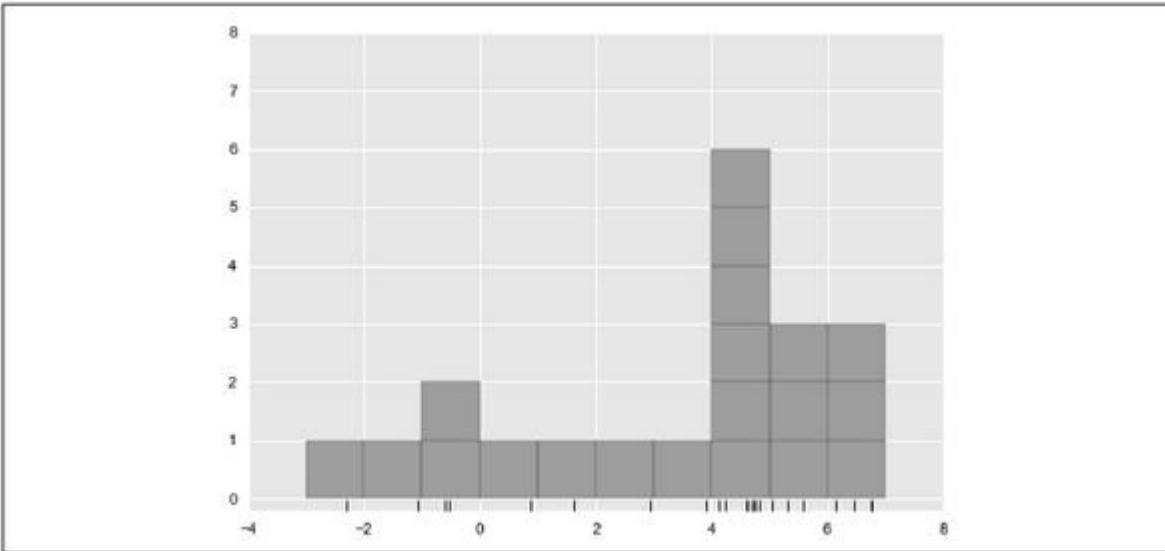
Figure 5.141 : La position des classes peut affecter l'interprétation d'un histogramme.

Dans le graphique de gauche, on voit clairement qu'il s'agit d'une distribution bimodale. Dans la figure de droite, la distribution semble unimodale avec une longue queue. Si vous n'avez pas connaissance du code qui les génère, vous ne pouvez pas deviner que les deux histogrammes représentent les mêmes données. Comment peut-on faire confiance à l'intuition que procure un histogramme ? Et comment pallier cette situation ?

On peut se représenter un histogramme sous forme d'un empilement de blocs, un bloc étant posé à la position de

chaque point du jeu, dans chaque classe. Visualisons cet empilement ([Figure 5.14.2](#)) :

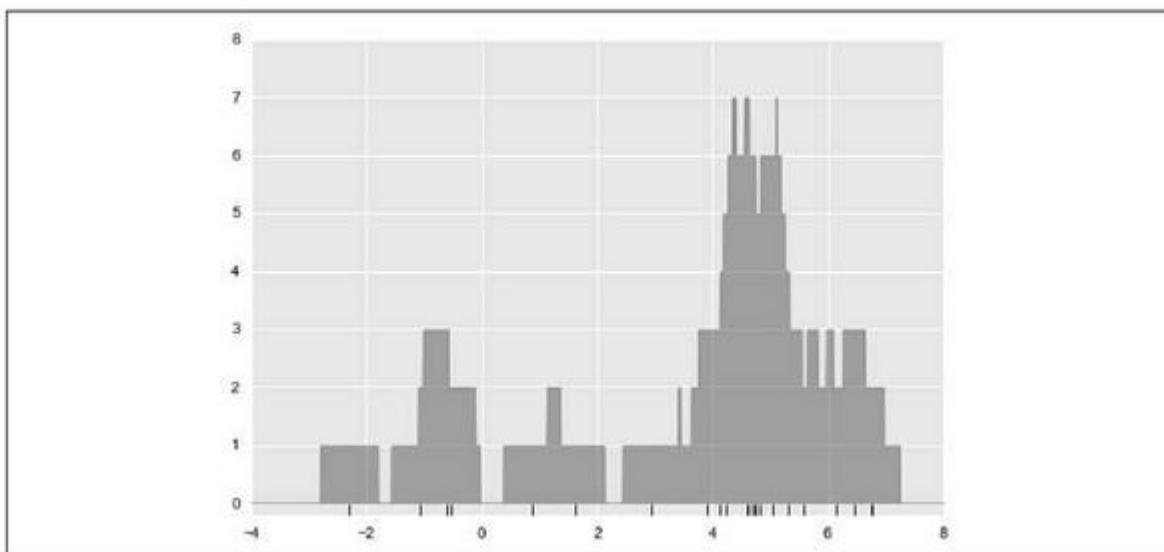
```
In[7]:  
fig, ax = plt.subplots()  
bins = np.arange(-3, 8)  
ax.plot(x, np.full_like(x, -0.1), '|k',  
markeredgecolor=1)  
for count, edge in zip(*np.histogram(x, bins)):  
    for i in range(count):  
        ax.add_patch(plt.Rectangle((edge, i), 1,  
1, alpha=0.5))  
  
ax.set_xlim(-4, 8)  
ax.set_ylim(-0.2, 8)  
  
Out[7]: (-0.2, 8)
```



[Figure 5.142](#) : Un histogramme en tant qu'empilement de blocs.

Le problème de ces deux distributions dans des bacs est lié au fait que la hauteur d'une pile de blocs ne découle pas obligatoirement de la densité réelle des points du voisinage, mais des coïncidences d'alignement des bacs par rapport aux points de données. L'histogramme ne peut pas être une représentation fidèle s'il y a un mauvais alignement entre les points et les blocs. Au lieu d'empiler les blocs en alignement par rapport aux bacs, pourquoi ne pas les empiler en les alignant par rapport aux points représentés ? Dans ce cas, les blocs ne seront plus alignés, mais rien ne nous empêche d'additionner leur contribution à chaque position le long de l'axe x pour trouver le résultat. Tentons cette approche ([Figure 5.143](#)) :

```
In[8]:  
x_d = np.linspace(-4, 8, 2000)  
density = sum((abs(xi - x_d) < 0.5) for xi in x)  
  
plt.fill_between(x_d, density, alpha=0.5)  
plt.plot(x, np.full_like(x, -0.1), '|k',  
markeredgecolor='black', markeredgewidth=1)  
  
plt.axis([-4, 8, -0.2, 8]);
```



[Figure 5.143](#) : Un pseudo-histogramme, les blocs étant centrés sur les points individuels. C'est un exemple d'estimation de densité de noyau.

Le résultat est moins net, mais il offre une représentation beaucoup plus fiable des caractéristiques réelles des données. Les crénelages sont non seulement inesthétiques, mais ils ne représentent aucune vraie propriété. Pour les lisser, nous pourrions remplacer les blocs par une fonction

de lissage telle qu'une gaussienne. Voyons ce que nous apporte le remplacement à chaque point d'un bloc par une courbe standard normale ([Figure 5.144](#)) :

In[9] :

```
from scipy.stats import norm

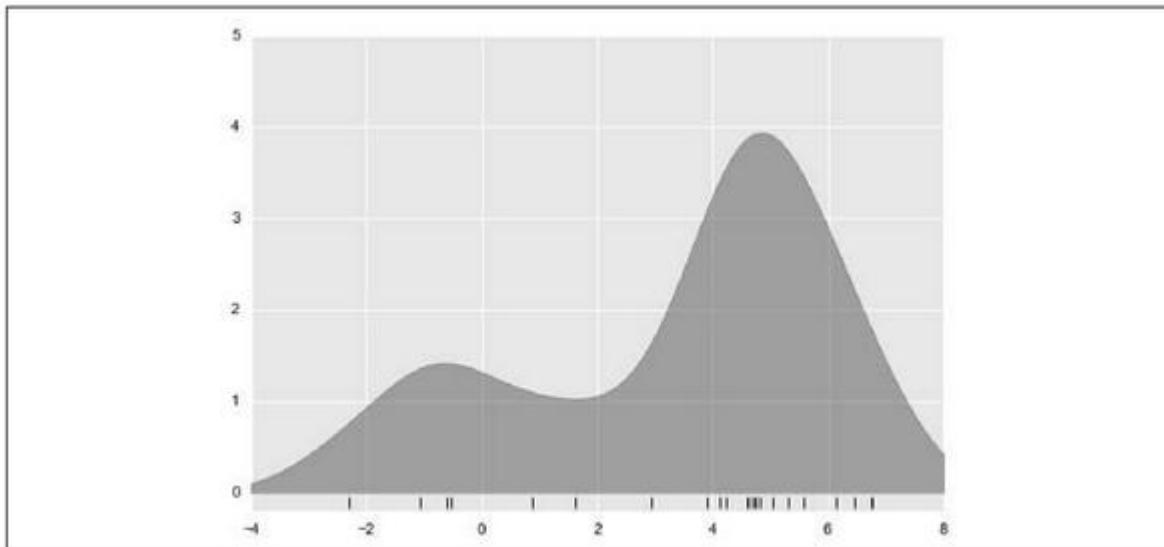
x_d = np.linspace(-4, 8, 1000)
density = sum(norm(xi).pdf(x_d) for xi in x)

plt.fill_between(x_d, density, alpha=0.5)
plt.plot(x, np.full_like(x, -0.1), '|k',
markeredgecolor='black', markeredgewidth=1)

plt.axis([-4, 8, -0.2, 5]);
```

Ce tracé lissé grâce à l'effet d'une distribution gaussienne à la position de chaque point d'entrée nous donne une idée beaucoup plus claire du format de la distribution des données, avec bien moins de variance (les différences dans l'échantillonnage entraînent beaucoup moins de changement).

Nos deux derniers tracés constituent des exemples d'estimation de densité de noyau en une dimension. Le premier utilise un noyau appelé « tophat » et le second un noyau gaussien. Plongeons maintenant plus dans les détails de l'estimation de densité de noyau.



[Figure 5.144](#) : Estimation de densité de noyau avec noyau gaussien.

L'estimation de densité de noyau en pratique

L'estimation de densité de noyau possède deux paramètres libres : tout d'abord le noyau qui détermine la forme de la distribution à la position de chaque point ; ensuite, la bande passante de noyau qui contrôle la taille du noyau à chaque point. Un certain nombre de noyaux sont disponibles pour faire une estimation de densité. L'implémentation Scikit-Learn de KDE en propose six. Pour en savoir plus, voyez la documentation de l'estimation de densité de Scikit-Learn (page titrée *density* dans la section des modules de la documentation de Scikit-Learn).

Python offre le choix parmi plusieurs modules d'estimation de densité de noyau, notamment dans les paquetages SciPy et StatsModels. Je préfère cependant la version de Scikit-Learn car elle est efficace et souple. Elle est incarnée par l'estimateur nommé `sklearn.neighbors.KernelDensity`. Il se charge de l'opération KDE en plusieurs dimensions en utilisant un noyau choisi parmi six et une métrique de distance parmi douze ou plus. L'estimateur de Scikit-Learn se fonde sur un algorithme en arbre pour limiter la charge de traitement. Il possède deux paramètres appelés `atol` (tolérance absolue) et `rtol` (tolérance relative) pour réaliser un arbitrage entre durée du traitement et précision. La fenêtre (*bandwidth*) de noyau étant un paramètre libre, nous pouvons la déterminer en nous servant des outils de validation croisée standard de Scikit-Learn, ce que nous allons voir bientôt.

Commençons par un exemple de réPLICATION simple qui reproduit le dernier tracé réalisé au moyen de l'estimateur de Scikit-Learn nommé `KernelDensity` ([Figure 5.145](#)) :

In[10]:

```
from sklearn.neighbors import KernelDensity
```

```
# Instanciation et ajustement du modèle KDE
```

```
kde = KernelDensity(bandwidth=1.0,
```

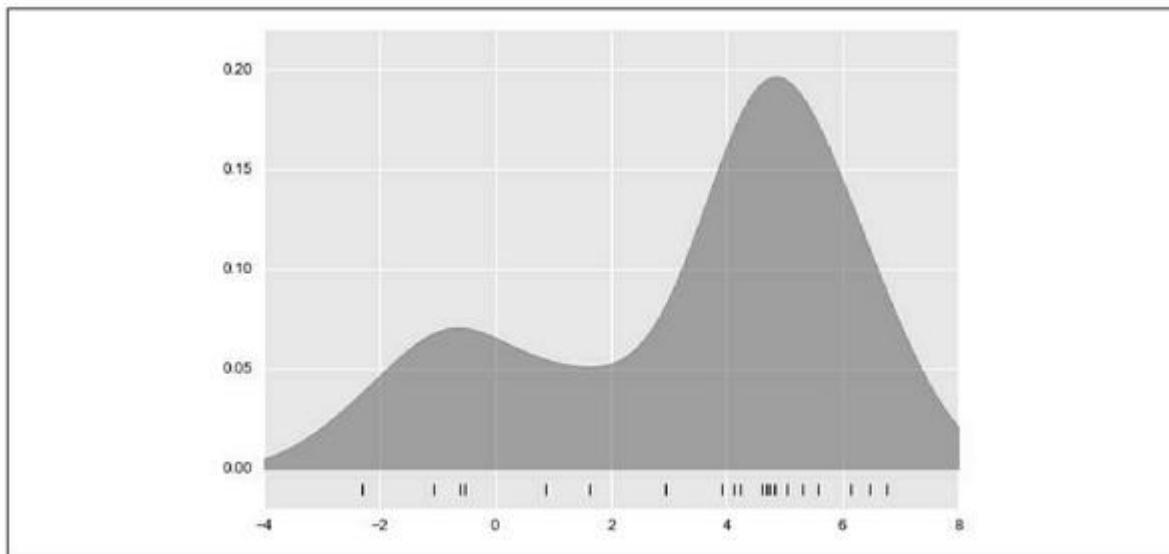
```
kernel='gaussian')
```

```
kde.fit(x[:, None])
```

```
# score_samples renvoie le log de densité de proba
logprob = kde.score_samples(x_d[:, None])

plt.fill_between(x_d, np.exp(logprob),
alpha=0.5)
plt.plot(x, np.full_like(x, -0.01), '|k',
markeredgewidth=1)
plt.ylim(-0.02, 0.22)
```

Out[10]: (-0.02, 0.22)



[Figure 5.145](#) : Estimation de densité de noyau produit par Scikit-Learn.

Le résultat a été normalisé pour que la surface sous la courbe soit égale à 1.

Sélection de bandwidth par validation croisée

Le choix du paramètre de fenêtre bandwidth dans KDE est crucial pour pouvoir trouver une estimation de densité correcte. C'est ce paramètre qui permet d'arbitrer entre biais et variance dans l'estimation. Si la valeur est trop faible, on obtient une estimation à forte variance, donc un surajustement ; la présence ou l'absence d'un seul point peut avoir un impact important. Inversement, si la fenêtre est trop large, on obtient une estimation à biais fort, donc un sous-ajustement ; la structure des données est alors effacée à cause d'un noyau trop vaste.

La science des statistiques a défini depuis longtemps un certain nombre de méthodes pour estimer rapidement la meilleure fenêtre de lissage en fonction de contraintes concernant les données : vous pouvez étudier l'utilisation de certaines de ces règles en allant voir l'implémentation de KDE dans les deux paquetages SciPy et StatsModels.

On a pu constater dans les projets d'apprentissage machine que le réglage de ce genre d'hyperparamètre est souvent fait de façon empirique par une validation croisée. D'ailleurs, l'estimateur `KernelDensity` de Scikit-Learn a été conçu pour pouvoir être exploité directement avec les outils de recherche *grid* standard de Scikit-Learn. Voyons comment optimiser la fenêtre de lissage du jeu de données précédent au moyen de `GridSearchCV`. Notre jeu de données étant

d'une taille limitée, nous allons opter pour la validation croisée un-contre-tous (*leave-one-out*), car elle limite la réduction de la taille du jeu d'entraînement lors de chaque essai de validation croisée :

```
In[11]:  
from sklearn.model_selection import GridSearchCV  
from sklearn.model_selection import LeaveOneOut  
  
bandwidths = 10 ** np.linspace(-1, 1, 100)  
grid =  
GridSearchCV(KernelDensity(kernel='gaussian'),  
             {'bandwidth':  
              bandwidths},  
  
cv=LeaveOneOut() )  
grid.fit(x[:, None]);
```

Nous pouvons maintenant demander la valeur de lissage qui va donner le score maximal (dans l'exemple, c'est par défaut la probabilité *log*) :

```
In[12]:  
grid.best_params_  
  
Out[12]:  
{'bandwidth': 1.1233240329780276}
```

La valeur optimale de fenêtre s'avère très proche de celle que nous avions utilisée dans le tracé d'exemple, avec la valeur 1.0 (c'est d'ailleurs la valeur par défaut pour `scipy.stats.norm`).

Exemple 1 : KDE sur une sphère

Un des domaines d'emploi principaux de KDE consiste à représenter graphiquement des distributions de points. D'ailleurs, KDE a été intégré au paquetage Seaborn et il est utilisé automatiquement pour aider l'affichage des points en une et deux dimensions. (Nous avons présenté « Seaborn » à la fin du [Chapitre 4](#).)

Passons à un exemple plus sophistiqué utilisant KDE pour visualiser une distribution. Nous allons profiter d'un exemple intégré à Scikit-Learn avec des données géographiques. Il s'agit de l'inventaire de deux espèces de petits mammifères d'Amérique du Sud, *Bradypus variegatus* (le Paresseux à gorge brune) et *Microryzomys minutus* (le petit rat du riz forestier).

Commençons par charger les données avec Scikit-Learn :

```
In[13]:  
from sklearn.datasets import  
fetch_species_distributions
```

```
data = fetch_species_distributions()

# Obtention matrices/tableaux des ID d'espèces
# et lieux
latlon = np.vstack([data.train['dd lat'],
                    data.train['dd long']]).T
species =
np.array([d.decode('ascii').startswith('micro')
           for d in
data.train['species']], dtype='int')
```



(N.d.T.) Ayez un peu de patience pour l'exécution du bloc précédent. Par ailleurs, il est possible que cet exemple rencontre des soucis lorsque vous tenterez de l'exécuter, car il se fonde sur des éléments en pleine évolution au moment d'écrire ces lignes. Les archives d'accompagnement du livre (voir l'introduction) contiendront si nécessaire un complément d'information.

Nous pouvons maintenant profiter de la boîte à outils Basemap (déjà utilisée à la fin du [Chapitre 4](#)) afin d'afficher les positions correspondant aux deux espèces inventoriées en Amérique du Sud ([Figure 5.14.6](#)) :

In[14]:

```
from mpl_toolkits.basemap import Basemap
from sklearn.datasets.species_distributions
import construct_grids

xgrid, ygrid = construct_grids(data)
```

```
# Trace côtes avec basemap
m = Basemap(projection='cyl', resolution='c',
            llcrnrlat=ygrid.min(),
            urcrnrlat=ygrid.max(),
            llcrnrlon=xgrid.min(),
            urcrnrlon=xgrid.max())
m.drawmapboundary(fill_color='#DDEEFF')
m.fillcontinents(color='#FFEEDD')
m.drawcoastlines(color='gray', zorder=2)
m.drawcountries(color='gray', zorder=2)

# Trace lieux
m.scatter(latlon[:, 1], latlon[:, 0], zorder=3,
          c=species,
          cmap='rainbow', latlon=True);
```



Figure 5.146 : Lieu de vie de deux espèces dans les données d'entraînement.

Cette première approche ne nous donne pas une idée précise de la densité de chacune des espèces parce que les points se recouvrent les uns les autres. Cela ne se voit pas sur la figure, mais il y a plus de 1 600 points au total !

Adoptons une estimation de densité par noyau pour obtenir quelque chose de plus interprétable. Nous voulons une indication lissée de la densité. Le système de coordonnées des points correspond à une surface sphérique et non à un plan. Pour représenter correctement les distances exprimées sur une surface courbe, nous nous servons de la métrique nommée haversine.

L'extrait qui suit comporte un certain nombre de lignes de configuration générique (c'est un des inconvénients de

Basemap). La signification de chaque sous-bloc devrait être claire néanmoins ([Figure 5.14.7](#)) :

```
In[15]: # Préparation de la grille de données
pour les contours
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()
xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = np.radians(xy[land_mask])

# Création de deux tracés juxtaposés
fig, ax = plt.subplots(1, 2)
fig.subplots_adjust(left=0.05, right=0.95,
wspace=0.05)
species_names = ['Bradypus Variegatus',
'Microryzomys Minutus']
cmaps = ['Purples', 'Reds']

for i, axi in enumerate(ax):
    axi.set_title(species_names[i])

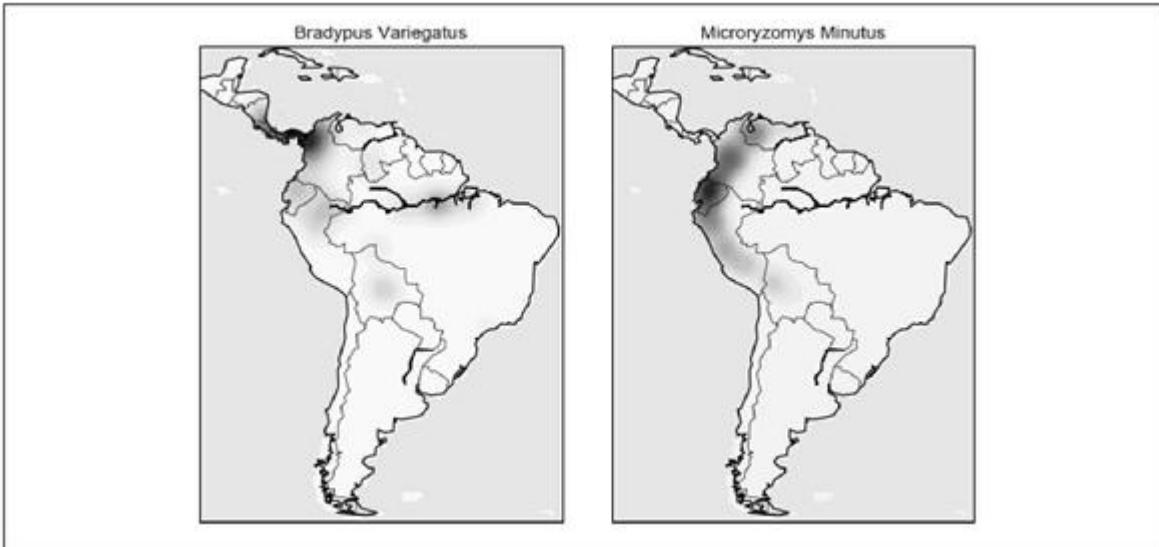
    # Ajout des côtes avec Basemap
    m = Basemap(projection='cyl',
llcrnrlat=Y.min(),
                urcrnrlat=Y.max(),
llcrnrlon=X.min(),
                urcrnrlon=X.max(),
resolution='c', ax=axi)
    m.drawmapboundary(fill_color='#DDEEFF')
```

```
m.drawcoastlines()
m.drawcountries()

# Construction d'une estimation de densité
# sphérique de la distribution
kde = KernelDensity(bandwidth=0.03,
metric='haversine')
kde.fit(np.radians(latlon[species == i]))

# Évaluation juste sur les terres. -9999
# signifie sur les mers
Z = np.full(land_mask.shape[0], -9999.0)
Z[land_mask] = np.exp(kde.score_samples(xy))
Z = Z.reshape(X.shape)

# Tracé des contours de la densité
levels = np.linspace(0, Z.max(), 25)
axi.contourf(X, Y, Z, levels=levels,
cmap=cmaps[i])
```



[Figure 5.147](#) : Représentation KDE de la distribution de deux espèces.

Cette visualisation donne une image beaucoup plus compréhensible de la distribution géographique des deux espèces.

Exemple 2 : un bayésien pas si naïf

Voyons maintenant comment réaliser avec KDE une classification générative bayésienne. Nous verrons ainsi comment utiliser l'architecture de Scikit-Learn pour créer un estimateur personnalisé.

Dans la section de début de chapitre dédiée à la classification bayésienne naïve, nous avions découvert la classification bayésienne en créant un modèle génératif simple pour chaque classe. Nous avions ensuite utilisé des modèles pour construire un classifieur rapide. Dans l'approche bayésienne

naïve, le modèle générateur se résume à une gaussienne alignée selon les axes. Nous pouvons lui faire perdre toute naïveté en adoptant un algorithme à estimation de densité tel que KDE. Nous pourrons ainsi réaliser la même classification avec pour chaque classe un modèle génératif plus sophistiqué. Il s'agit toujours d'une classification bayésienne, mais elle n'est plus naïve.

Voici la procédure générale pour une classification générative :

1. Répartition des données d'entraînement par labels.
2. Pour chaque jeu, ajustement KDE pour produire un modèle génératif des données. Pour chaque observation x et label y , il devient ainsi possible de calculer la probabilité $P(x | y)$.
3. À partir du nombre d'exemples dans chaque classe du jeu d'entraînement, calcul de la classe a priori, $P(y)$.
4. La probabilité a posteriori de chaque classe pour un point inconnu x vaut $P(y | x) \propto P(x | y)P(y)$. La classe qui maximise ce a posteriori correspond aux labels affectés au point.

Le fonctionnement de cet algorithme est facile à comprendre. Ce qui est moins facile, c'est de le combiner avec l'infrastructure Scikit-Learn pour pouvoir utiliser la recherche *grid* et la validation croisée.

Voici d'abord le code qui implémente l'algorithme dans Scikit-Learn. Nous étudierons chaque sous-bloc un à un dans la suite :

In[16]:

```
from sklearn.base import BaseEstimator,
ClassifierMixin

class KDEClassifier(BaseEstimator,
ClassifierMixin):
    """Classification générative bayésienne selon
KDE

    Paramètres
    -----
    bandwidth : float
        la fenêtre noyau dans chaque classe
    kernel : str
        le nom du noyau transmis à KernelDensity
    """

    def __init__(self, bandwidth=1.0,
kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel

    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in
self.classes_]
        self.models_ =
```

```

[KernelDensity(bandwidth=self.bandwidth,
kernel=self.kernel).fit(Xi)
    for Xi in training_sets]
self.logpriors_ = [np.log(Xi.shape[0] /
X.shape[0])
    for Xi in training_sets]
return self

def predict_proba(self, X):
    logprobs = np.array([model.score_samples(X)
        for model in
self.models_]).T
    result = np.exp(logprobs + self.logpriors_)
    return result / result.sum(1, keepdims=True)

def predict(self, X):
    return
self.classes_[np.argmax(self.predict_proba(X),
1)]

```

Anatomie d'un estimateur spécifique

Passons le code en revue pour en discuter les points essentiels :

```

from sklearn.base import BaseEstimator,
ClassifierMixin

class KDEClassifier(BaseEstimator,

```

```
ClassifierMixin):
    """Classification générative bayésienne selon
KDE

    Paramètres
    -----
    bandwidth : float
        la fenêtre noyau dans chaque classe
    kernel : str
        le nom du noyau transmis à KernelDensity
    """
```

Dans Scikit-Learn, tous les estimateurs sont des classes. Elles ont intérêt à hériter de la classe `BaseEstimator` ainsi que de la classe *mixin* appropriée pour disposer des fonctionnalités standard. Parmi d'autres choses, la classe `BaseEstimator` définit la logique permettant de cloner ou copier un estimateur dans le cadre d'une procédure de validation croisée. Par ailleurs `ClassifierMixin` définit une méthode par défaut `score()` que les routines vont utiliser. Notez que nous avons prévu une chaîne de documentation qui sera être récupérée par la fonction d'aide d'IPython décrite en début de livre.

Nous arrivons ensuite au niveau de la méthode d'initialisation de classe :

```
def __init__(self, bandwidth=1.0,
kernel='gaussian'):
```

```
    self.bandwidth = bandwidth
    self.kernel = kernel
```

C'est ce code qui est exécuté lors de la création de l'instance par `KDEClassifier()`. Lorsque vous utilisez Scikit-Learn, il est important de n'ajouter aucune opération dans le bloc d'initialisation, sauf l'affectation des valeurs à transmettre par nom à `self`. Cette contrainte est due à la logique que contient `BaseEstimator` pour pouvoir cloner et modifier les estimateurs en vue des validations croisées, des recherches *grid* et autres. Par ailleurs, tous les arguments de `__init__` doivent être explicites. N'utilisez pas `*args` ou `**kwargs` car ces paramètres ne sont pas correctement pris en compte dans les routines de validation croisée.

Nous arrivons ensuite à la définition de la méthode `fit()` qui gère les données d'entraînement :

```
def fit(self, X, y):
    self.classes_ = np.sort(np.unique(y))
    training_sets = [X[y == yi] for yi in
                     self.classes_]
    self.models_ =
        [KernelDensity(bandwidth=self.bandwidth,
                       kernel=self.kernel).fit(Xi)
                     for Xi in training_sets]
    self.logpriors_ = [np.log(Xi.shape[0] /
                             X.shape[0])]
```

```
        for xi in training_sets]
    return self
```

C'est dans ce bloc que nous trouvons les classes uniques des données d'entraînement, que nous entraînons le modèle KernelDensity pour chaque classe puis calculons les a priori en fonction du nombre d'échantillons en entrée. La méthode fit() doit toujours renvoyer self, afin que l'on puisse enchaîner sur d'autres traitements. Voici un exemple :

```
label = model.fit(X, y).predict(X)
```

Chaque résultat persistant de l'ajustement est stocké avec un nom se terminant par le caractère de soulignement (par exemple, self.logpriors__). Cette convention de Scikit-Learn est très pratique : vous pouvez ainsi rapidement balayer les différents membres d'un estimateur avec la touche Tabulation dans IPython pour voir rapidement quels membres conviennent aux données d'entraînement.

Nous arrivons enfin à la logique qui va prédire les labels sur de nouvelles données :

```
def predict_proba(self, X):
    logprobs = np.vstack([model.score_samples(X)
                          for model in
                          self.models_]).T
    result = np.exp(logprobs + self.logpriors_)
    return result / result.sum(1, keepdims=True)
```

```
def predict(self, X):
    return
    self.classes_[np.argmax(self.predict_proba(X),
1)]
```

Il s'agit ici d'un classifieur probabiliste. Nous définissons donc d'abord `predict_proba()`, méthode qui va renvoyer un tableau de probabilités de classe de la forme `[n_samples, n_classes]`. Dans ce tableau, l'entrée désignée par `[i, j]` est la probabilité *a posteriori* que l'échantillon *i* est membre de la classe *j*.

Nous terminons par la méthode `predict()` qui exploite les probabilités pour renvoyer la classe qui a la plus forte.

Utilisation de l'estimateur spécifique

Testons notre estimateur sur un problème connu : la classification des chiffres manuscrits. Nous chargeons les chiffres puis faisons calculer le score de validation croisée pour toute une série de valeurs de `bandwidth` (fenêtres de lissage) candidates grâce au méta-estimateur `GridSearchCV` (revoyez si nécessaire la section sur les hyperparamètres et la validation de modèle en début de chapitre) :

```
In[17]:
from sklearn.datasets import load_digits
from sklearn.model_selection import GridSearchCV
```

```

digits = load_digits()

bandwidths = 10 ** np.linspace(0, 2, 100)
grid = GridSearchCV(KDEClassifier(),
{'bandwidth': bandwidths})
grid.fit(digits.data, digits.target)

scores =
grid.cv_results_.get('mean_test_score').tolist()

```

Nous pouvons tracer le score de validation croisée en fonction de la valeur de bandwidth ([Figure 5.148](#)) :

```

In[18]:
plt.semilogx(bandwidths, scores)
plt.xlabel('bandwidth')
plt.ylabel('accuracy')
plt.title('Performances du modèle KDE')
print(grid.best_params_)
print('accuracy =', grid.best_score_)

{'bandwidth': 7.0548023107186433}
accuracy = 0.966611018364

```

Notre classifieur bayésien qui n'est plus tellement naïf atteint une précision de validation croisée d'un peu plus de 96 %. Rappelons que la classification bayésienne naïve avait du mal à dépasser les 80 % :

In[19]:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import
cross_val_score

cross_val_score(GaussianNB(), digits.data,
digits.target).mean()
```

Out[19]: 0.81860038035501381

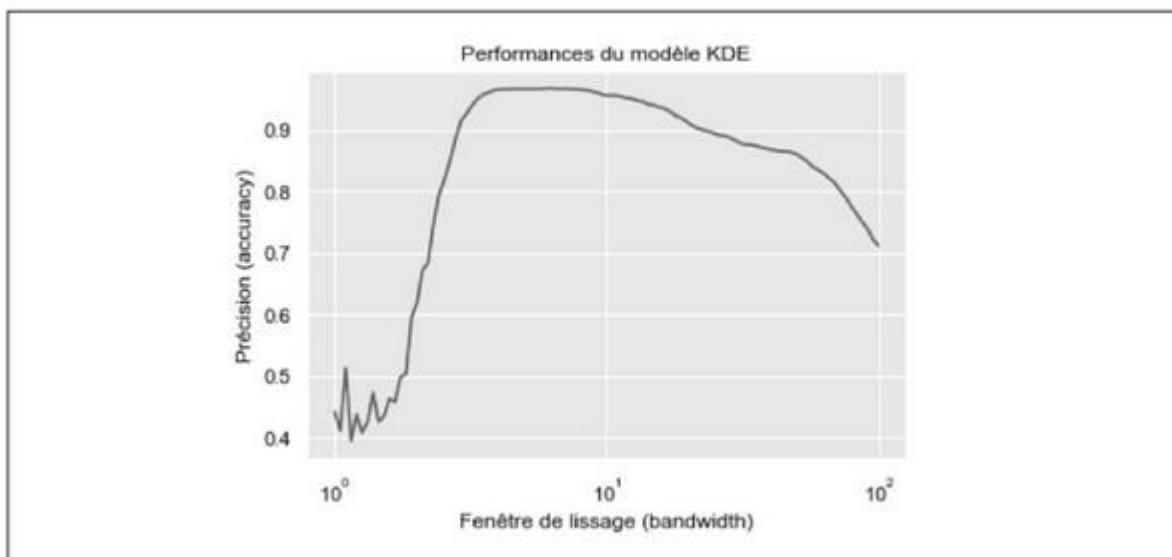


Figure 5.148 : Courbe de validation du classifieur bayésien basé sur KDE.

Un des points forts de ce genre de classifieur génératif est que les résultats sont faciles à interpréter : pour chaque échantillon inconnu, nous n'obtenons pas qu'une classification probabiliste, mais un modèle complet de la distribution de tous les points auxquels nous le comparons ! Vous disposez ainsi de pistes intuitives pour trouver les

raisons d'une classification particulière, raisons que d'autres algorithmes tels que SVM ou les forêts aléatoires ont tendance à masquer.

Si vous voulez aller plus loin, voici quelques améliorations qui peuvent être apportées à ce modèle de classifieur KDE :

- Vous pouvez faire varier indépendamment la fenêtre de lissage de chaque classe.
- Vous pouvez optimiser les fenêtres non à partir du score de prédiction, mais à partir de la probabilité du jeu d'entraînement sous le modèle génératif au sein de chaque classe (par exemple en utilisant les scores directement de KernelDensity plutôt que la précision de prédiction globale).

Enfin, si vous voulez essayer de créer votre propre estimateur, commencez par construire un classifieur bayésien en utilisant un modèle de mélange gaussien plutôt que KDE.

5.14 : Histogrammes de gradients orientés (HOG)

Ce grand chapitre nous a permis de découvrir un certain nombre de concepts fondamentaux d'apprentissage machine ainsi qu'une dizaine d'algorithmes. La mise en pratique de ces concepts dans des applications réelles n'est pas nécessairement simple. En effet, les jeux de données du monde réel sont souvent bruts et non homogènes ; il leur manque des caractéristiques, ils peuvent présenter les données dans un format qui n'est pas facile à reformuler dans une forme matricielle simple de type [n_samples, n_features]. Pour pouvoir exploiter les méthodes décrites ici, vous devez d'abord extraire ces caractéristiques de vos données. Il n'y a pas de formule magique pouvant s'appliquer à tous les domaines. C'est à vous en tant que spécialiste et datologue de contribuer par votre expertise et votre intuition à certains choix.

Un domaine d'application de l'apprentissage machine à la fois intéressant et exigeant est celui de la reconnaissance d'images. Plusieurs exemples du chapitre nous ont permis de voir comment exploiter les caractéristiques au niveau des pixels pour des classifications. Dans un contexte réel, les données ne sont que rarement aussi bien préparées, et on ne

peut pas utiliser directement des pixels. Un domaine de recherche foisonnant s'intéresse justement aux méthodes d'extraction des caractéristiques pour les données de type image (nous en avons parlé dans la section sur l'ingénierie des caractéristiques en début de chapitre).

Dans cette dernière section du chapitre, nous allons découvrir une technique d'extraction de caractéristiques qui porte le nom d'*Histogramme de gradients orientés* (HOG).



(N.d.T.) Nous utiliserons l'acronyme anglais HOG, bien que cette technique ait été mise au point en France, à l'INRIA de Grenoble.

Cette technique HOG convertit les pixels d'une image en une représentation vectorielle qui tient compte globalement des caractéristiques déterminantes de l'image, après application d'un filtrage pour éliminer les effets négatifs, en particulier les changements de luminosité. Nous allons nous servir de cette technique pour créer un pipeline de traitement pour détecter des visages, en réutilisant des algorithmes et des concepts présentés au cours du chapitre. Nous commençons par nos opérations d'import habituelles :

```
In[1]: %
matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Principe des histogrammes de gradients orientés

HOG désigne une procédure d'extraction de caractéristiques efficace, qui cherche au départ à identifier des piétons dans des images. Le traitement HOG comprend les cinq étapes suivantes :

1. Prénormalisation éventuelle des images. Cette opération permet de rendre les caractéristiques moins sensibles aux variations d'éclairage.
2. Application de filtres de convolution d'image, filtres qui sont sensibles au gradient de luminosité horizontal et vertical. Cela permet de capturer les bords, les contours et les textures.
3. Division de l'image en cellules de taille prédéfinie, puis calcul de l'histogramme des gradients orientés dans chaque cellule.
4. Dans chaque cellule, normalisation des histogrammes par comparaison aux blocs contenant les cellules voisines afin d'atténuer encore les effets de luminosité dans l'image.
5. Construction d'un vecteur de caractéristiques à une dimension à partir des informations de chaque cellule.

Scikit-Image embarque un extracteur HOG rapide. Nous pouvons facilement le tester afin de visualiser les gradients orientés dans chaque cellule ([Figure 5.14.9](#)) :

In[2]:

```
from skimage import data, color, feature
import skimage.data

image = color.rgb2gray(data.chelsea())
hog_vec, hog_vis = feature.hog(image,
visualise=True)
fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                      subplot_kw=dict(xticks=[],
ytics=[])) ax[0].imshow(image, cmap='gray')
ax[0].set_title('input image')
ax[1].imshow(hog_vis)
ax[1].set_title('Visualisation des HOG
features');
```



Figure 5.149 : Visualisation des caractéristiques HOG d'une image.

Un détecteur de visages simplifié avec HOG

Les possibilités de HOG permettent de créer un algorithme de détection de visages simplifié avec n'importe quel estimateur de Scikit-Learn. Ici, nous allons choisir une machine à vecteurs de support linéaire (revoyez la section correspondante du chapitre si nécessaire). Voici les étapes de notre progression :

1. Collecte des vignettes d'un ensemble d'images de visages qui va constituer le jeu d'entraînement positif.
2. Collecte des vignettes d'un ensemble d'images sans visage pour constituer le jeu d'entraînement négatif.
3. Extraction des caractéristiques de gradients orientés à partir de ces deux jeux d'entraînement.

4. Entraînement d'un classifieur SVM linéaire sur ces échantillons.
5. Analyse d'une image inconnue au moyen d'une fenêtre mobile, en se servant du modèle pour estimer si la fenêtre contient un visage ou non.

(En cas de chevauchement des détections, recombinaison en une seule fenêtre.)

Mettons en pratique cette procédure en six étapes :

1. Collecte du jeu d'entraînement positif

Il nous faut d'abord un jeu d'échantillons positifs, montrant donc des visages. Nous en avons sous la main dans le fichier des personnalités politiques déjà utilisé (LFW). Nous le téléchargeons avec Scikit-Learn :

In[3] :

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people()
positive_patches = faces.images
positive_patches.shape
```

Out[3] : (13233, 62, 47)

Nous disposons maintenant de 13 000 images de visages pour notre entraînement.

2. Collecte du jeu d'entraînement négatif

Il nous faut maintenant les vignettes ne contenant aucun visage. Une solution rapide consiste à choisir un jeu d'images d'entrée pour en extraire des vignettes à différentes échelles. Nous puisions dans les images qui sont livrées avec Scikit-Image grâce à PatchExtractor :

In[4]:

```
from skimage import data, transform
imgs_to_use = ['camera', 'text', 'coins',
'moon',
        'page', 'clock',
'immunohistochemistry',
        'chelsea', 'coffee',
'hubble_deep_field']
images = [color.rgb2gray(getattr(data, name)())
for name in imgs_to_use]
```

In[5]:

```
from sklearn.feature_extraction.image import
PatchExtractor

def extract_patches(img, N, scale=1.0,
patch_size=positive_patches[0].shape):
    extracted_patch_size = \
        tuple((scale *
np.array(patch_size)).astype(int))
    extractor =
```

```

PatchExtractor(patch_size=extracted_patch_size,
              max_patches=N,
              random_state=0)
    patches =
extractor.transform(img[np.newaxis])
    if scale != 1:
        patches =
np.array([transform.resize(patch, patch_size)
          for patch in
patches])
    return patches

negative_patches =
np.vstack([extract_patches(im, 1000, scale)
          for im in images for scale in
[0.5, 1.0, 2.0]])
negative_patches.shape

```

Out[5]: (30000, 62, 47)

Nous disposons maintenant de 30 000 vignettes sans visages. Affichons-en quelques-unes pour voir à quoi elles ressemblent ([Figure 5.150](#)) :

```

In[6]:
fig, ax = plt.subplots(6, 10)
for i, axi in enumerate(ax.flat):
    axi.imshow(negative_patches[500 * i],
cmap='gray')
    axi.axis('off')

```



[Figure 5.150](#) : Aperçu des vignettes d'images négatives, sans visages.

Nous espérons bien sûr que ces échantillons couvriront suffisamment l'espace graphique de « non-visages » auquel sera confronté notre algorithme.

3. Combinaison des jeux et extraction des caractéristiques HOG

Nous disposons de nos échantillons positifs et négatifs ; nous pouvons les combiner pour faire calculer les caractéristiques de gradients HOG. Soyez patients, car le traitement HOG effectue des calculs sophistiqués sur chaque image :

```
In[7]:  
from itertools import chain  
X_train = np.array([feature.hog(im)  
                   for im in
```

```
chain(positive_patches, negative_patches)))
y_train = np.zeros(X_train.shape[0])
y_train[:positive_patches.shape[0]] = 1
```

```
In[8]: X_train.shape
Out[8]: (43233, 1215)
```

Nous aboutissons à 43 000 échantillons d'entraînement en 1 215 dimensions. Nos données sont dorénavant dans un format qui peut être injecté dans Scikit-Learn.

4. Entraînement d'une machine à vecteurs de support

Nous pouvons maintenant nous doter d'un classifieur pour traiter nos vignettes en puisant dans les outils découverts tout au long de ce chapitre. Une classification binaire avec un tel nombre de dimensions incite à adopter une machine à vecteurs de support linéaire. Nous nous servons de LinearSVC de Scikit-Learn qui supporte mieux que SVC le traitement d'un grand nombre d'échantillons.

Pour un premier aperçu rapide, lançons d'abord une bayésienne naïve gaussienne très simple :

```
In[9]:
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import
cross_val_score

cross_val_score(GaussianNB(), X_train, y_train)
```

```
Out[9]:
```

```
array([ 0.9408785 ,  0.8752342 ,  0.93976823])
```

À partir de nos données d'entraînement, ce simple algorithme bayésien nous apporte déjà plus de 90 % de précision. Essayons maintenant notre machine SVM avec une recherche *grid* pour avoir plusieurs choix pour le paramètre C :

```
In[10]:
```

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import
GridSearchCV

grid = GridSearchCV(LinearSVC(), { c : [1.0,
2.0, 4.0, 8.0]})
grid.fit(X_train, y_train)
grid.best_score_
```

```
Out[10]:
```

```
0.98667684407744083
```

```
In[11]:
```

```
grid.best_params_
```

```
Out[11]:
```

```
{'C': 4.0}
```

Lançons un entraînement sur le jeu de données complet en lui fournissant le meilleur estimateur :

```
In[12]:  
model = grid.best_estimator_  
model.fit(X_train, y_train)
```

```
Out[12]: LinearSVC()
```

5. Trouver les visages dans une image inconnue

Une fois le modèle en place, nous pouvons vérifier comment il se comporte avec une image encore inconnue. Pour simplifier, nous allons travailler sur une partie de l'image d'astronaute et appliquer une fenêtre mobile pour évaluer chaque échantillon ([Figure 5.151](#)) :

```
In[13]:  
test_image = skimage.data.astronaut()  
test_image = skimage.color.rgb2gray(test_image)  
test_image =  
skimage.transform.rescale(test_image, 0.5)  
test_image = test_image[:160, 40:180]  
  
plt.imshow(test_image, cmap='gray')  
plt.axis('off');
```



Figure 5.151 : Image dans laquelle nous tentons de trouver un visage.

Nous définissons une fonction de fenêtre mobile qui va balayer les différentes cellules de l'image pour calculer les caractéristiques HOG de chacune :

```
In[14]:  
def sliding_window(img,  
patch_size=positive_patches[0].shape,  
                  istep=2, jstep=2, scale=1.0):  
    Ni, Nj = (int(scale * s) for s in patch_size)  
    for i in range(0, img.shape[0] - Ni, istep):  
        for j in range(0, img.shape[1] - Ni,  
jstep):  
            patch = img[i:i + Ni, j:j + Nj]  
  
            if scale != 1:  
                patch = transform.resize(patch,
```

```
patch_size)
        yield (i, j), patch

indices, patches =
zip(*sliding_window(test_image))
patches_hog = np.array([feature.hog(patch) for
patch in patches])
patches_hog.shape
```

Out[14]: (1911, 1215)

Nous récupérons ces différentes cellules (ou *patches*) qualifiées par les gradients HOG et utilisons le modèle pour évaluer si chacune des cellules contient ou non un visage :

In[15]:

```
labels = model.predict(patches_hog)
labels.sum()
```

Out[15]: 33.0

Sur 2 000 cellules, nous avons 30 détections. Servons-nous des informations récoltées au sujet des cellules pour visualiser leur position dans l'image de test, sous forme de cadres rectangulaires ([Figure 5.152](#)) :

In[16]:

```
fig, ax = plt.subplots()
ax.imshow(test_image, cmap='gray')
ax.axis('off')
```

```
Ni, Nj = positive_patches[0].shape
indices = np.array(indices)

for i, j in indices[labels == 1]:
    ax.add_patch(plt.Rectangle((j, i), Nj, Ni,
edgecolor='red',
alpha=0.3, lw=2,
facecolor='none'))
```

Les blocs ont effectivement trouvé le visage, mais ils se chevauchent. Ce n'est tout de même pas un mauvais résultat pour ces quelques lignes de Python !



[Figure 5.152](#) : Affichage des fenêtres correspondant aux visages détectés.

Mises en garde et pistes d'amélioration

Si vous étudiez en détail le code source de notre exemple, vous conviendrez qu'il reste un peu de travail avant d'en faire un détecteur de visages de niveau production. Le modèle possède encore quelques faiblesses, et plusieurs pistes d'amélioration sont possibles.

Le jeu d'entraînement est un peu limité, notamment pour les caractéristiques négatives

Le premier souci est que de nombreuses textures qui ressemblent à des visages ne sont pas disponibles dans le jeu d'entraînement, ce qui entraîne de nombreux faux

positifs par le modèle. Vous pouvez le vérifier en appliquant l'algorithme sur l'image complète de l'astronaute : de nombreuses détections erronées surviennent dans d'autres régions de l'image complète.

Nous pouvons améliorer la situation en ajoutant un plus vaste ensemble d'images pour la partie négative du jeu d'entraînement. Une approche plus simple est corrective : elle correspond à une collecte manuelle de négatifs (*hard negative mining*). Il s'agit de sélectionner un jeu d'images encore inconnues par le classifieur, et d'y trouver toutes les cellules correspondant à des faux positifs, pour les ajouter en tant qu'instances négatives dans le jeu d'entraînement, puis de relancer l'entraînement.

Le pipeline actuel ne recherche que dans une taille

Dans sa version actuelle, l'algorithme ne trouvera pas les visages qui n'ont pas une taille de 64 sur 47 pixels environ. Une solution immédiate consiste à appliquer des fenêtres mobiles de taille variée, en redimensionnant chaque cellule avec `skimage.transform.resize` avant de l'injecter dans le modèle. D'ailleurs, la fonction utilitaire `sliding_window()` que nous avons définie a été écrite de sorte de permettre cet enrichissement.

Les cellules détectées en chevauchement devraient être combinées

Pour rendre l'outil apte à la production, il est préférable de ne pas aboutir à 30 détections du même visage. Il s'agit donc de réduire les chevauchements pour ne garder qu'une seule détection. On peut adopter une approche de regroupement ou de partitionnement non supervisé telle que le partitionnement *MeanShift* ou utiliser une approche procédurale, par exemple celle de *non-maximum suppression*, souvent utilisée en vision artificielle.

Le pipeline devrait être plus ergonomique

Une fois les problèmes précédents résolus, il serait appréciable de créer un pipeline de traitement plus intuitif, de l'injection des images d'entraînement à la prédiction des sorties de la fenêtre mobile. Le langage Python est un excellent outil dans ce domaine. Moyennant quelques efforts, le code de prototypage et le paquetage peuvent être réunis pour proposer une interface de programmation API orientée objet qui permettra aux utilisateurs de s'en servir facilement. Je laisse les lecteurs motivés répondre à ce besoin en guise d'exercice.

De nouvelles techniques sont apparues, et notamment l'apprentissage profond

Je dois avouer pour conclure que les méthodes d'extraction de caractéristiques procédurales telles que HOG ne sont plus les techniques de pointe pour traiter des images. D'autres

processus de sélection d'objets se fondent sur les réseaux neuronaux. On peut considérer qu'un réseau neuronal représente un estimateur permettant de connaître les stratégies optimales d'extraction de caractéristiques des données. Il n'y a ainsi plus de dépendance par rapport à la capacité d'intuition de l'utilisateur. Tant en termes théoriques qu'en termes de puissance de traitement, les méthodes des réseaux neuronaux profonds dépassent largement le cadre de cette section.

Notons néanmoins l'existence d'outils tels que TensorFlow de Google (<https://www.tensorflow.org/>) grâce auxquels les stratégies d'apprentissage profond deviennent plus accessibles que jamais. Au moment de terminer ce livre, les disciplines d'apprentissage profond sont encore récentes en Python. Vous trouverez quelques pistes dans les références de la prochaine section.

Autres ressources d'apprentissage machine

Malgré sa longueur, ce chapitre n'a proposé qu'une visite rapide de la pratique de l'apprentissage machine avec Python. Nous avons principalement utilisé les outils de la librairie Scikit-Learn. Parmi les nombreux sujets qui n'ont pas pu être abordés, certains sont présentés dans les ressources suivantes.



(N.d.T) S'il existe une version en langue française, nous le mentionnerons.

L'apprentissage machine avec Python

Voici les ressources que je préconise pour en savoir plus sur l'apprentissage machine en Python :

Le Machine learning avec Python (*First*)

De Andreas C. Mueller et Sarah Guido, ce livre va plus loin dans le vaste choix d'outils d'apprentissage machine avec Python. Andreas est l'un des développeurs les plus fertiles de l'équipe Scikit-Learn. Le contenu vous permettra d'aller plus loin dans l'exploitation de la boîte à outils de Scikit-Learn.

Le site Web de Scikit-Learn

Ce site regorge de documentations et d'exemples abordant certains des modèles présentés dans ce chapitre, et bien d'autres choses.

Tutoriels vidéo de SciPy, PyCon et PyData

La librairie Scikit-Learn est souvent sélectionnée, parmi d'autres sujets, dans les tutoriels présentés au cours des conférences Python, notamment PyCon, SciPy et PyData. Une simple recherche Web permet de les trouver.

Python Machine Learning (Packt)

Le livre de Sebastian Raschka s'intéresse moins à la librairie Scikit-Learn qu'à la gamme d'outils d'apprentissage machine disponibles sous Python. Vous y trouverez une discussion détaillée des possibilités de support de la montée en charge de Python pour traiter des jeux de données volumineux et complexes.

L'apprentissage machine en général

L'apprentissage machine n'est pas limité au langage Python. De nombreuses ressources sont disponibles. En voici quelques-unes qui m'ont été très utiles.

Machine Learning

Ce cours en ligne gratuit de Andrew Ng chez Coursera aborde les principes de l'apprentissage machine d'un point de vue algorithmique. Il suppose un niveau Licence en mathématiques et en programmation. Vous êtes invité à implémenter certains des modèles de chez vous sous forme de devoirs.

Pattern Recognition and Machine Learning (*Springer*)

Ce livre de Christopher Bishop couvre de façon détaillée les concepts techniques présentés dans le présent chapitre. Une lecture incontournable pour qui veut se spécialiser dans ce domaine.

Machine learning: A Probabilistic Perspective (*MIT Press*)

Ce livre de niveau supérieur écrit par Kevin Murphy explore d'un point de vue probabiliste unifié les principaux algorithmes d'apprentissage machine.

Ces ressources offrent un contenu plus technique que ce livre. Pour en tirer profit, il est préférable de disposer d'un bagage conséquent en mathématiques. N'hésitez pas à relever le défi si vous vous sentez prêt à renforcer vos compétences en science des données !

Sommaire

Couverture

Python pour la Data Science

Copyright

Introduction

À qui s'adresse ce livre

Pourquoi Python ?

Organisation de ce livre

Contenu additionnel

Prérequis d'installation

Mise en pratique des exemples

Conventions utilisées dans ce livre

Terminologie française

Chapitre 1. IPython, plus loin que python

Interpréteur shell ou calepin ?

[Aide et documentation d'IPython](#)

[Raccourcis clavier du shell IPython](#)

[Commandes magiques d'IPython](#)

[Historique d'entrées et de sorties](#)

[IPython et les commandes shell](#)

[Commandes magiques liées au shell](#)

[Erreurs et débogage](#)

[Profilage et chronométrage d'exécution](#)

[Autres ressources IPython](#)

[Chapitre 2. Introduction à NumPy](#)

[2.1 : Les types de données Python](#)

[2.2 : Fondamentaux des tableaux NumPy](#)

[2.3 : Les fonctions universelles ufuncs](#)

[2.4 : Agrégats min, max et intermédiaire](#)

[2.5 : Calculs sur les tableaux : diffusion \(broadcast\)](#)

[2.6 : Comparaisons, masques et logique booléenne](#)

[2.7 : Indexation fancy](#)

[2.8 : Tri de tableaux](#)

[2.9 : Données structurées : les tableaux structurés de NumPy](#)

[2.10 : En route vers Pandas](#)

[Chapitre 3. Manipulation de données avec Pandas](#)

[3.1 : Installer et utiliser Pandas](#)

[3.2 : Présentation des objets de Pandas](#)

[3.3 : Indexation et sélection de données](#)

[3.4 : Opérations sur les données Pandas](#)

[3.5 : Gestion des données manquantes](#)

[3.6 : Indexation hiérarchique](#)

[3.7 : Combinaison de jeux de données avec concat et append](#)

[3.8 : Combinaison de jeux de données avec merge et join](#)

[3.9 : Agrégations et groupements](#)

[3.10 : Tableaux croisés dynamiques ou tableaux pivots](#)

[3.11 : Opérations vectorisées sur les chaînes](#)

[3.12 : Traitement des données temporelles](#)

[3.13 : Hautes performances Pandas avec eval\(\) et query\(\)](#)

[3.14 : Autres ressources](#)

[Chapitre 4. Visualisation avec Matplotlib](#)

[4.1 : Techniques Matplotlib fondamentales](#)

[4.2 : Deux interfaces pour le prix d'une](#)

[4.3 : Tracé de lignes simples](#)

[4.4 : Tracé en nuage de points](#)

[4.5 : Visualisation des erreurs](#)

[4.6 : Tracé de densités et de contours](#)

[4.7 : Histogrammes, bins et densité](#)

[4.8 : Personnalisation des légendes de tracé](#)

[4.9 : Personnalisation des barres de couleurs](#)

[4.10 : Sous-tracés multiples](#)

[4.11 : Texte et annotations](#)

[4.12 : Personnalisation des graduations](#)

[4.13 : Configuration et feuilles de styles de Matplotlib](#)

[4.14 : Tracés Matplotlib en 3D](#)

[4.15 : Données géographiques avec Basemap](#)

[4.16 : Visualisation avec Seaborn](#)

[4.17 : Autres ressources](#)

[Chapitre 5. Apprentissage machine](#)

[5.1 : L'apprentissage machine, c'est quoi ?](#)

[5.2 : Présentation de Scikit-Learn](#)

[5.3 : Hyperparamètres et validation du modèle](#)

[5.4 : Ingénierie des caractéristiques](#)

[5.5 : Classification bayésienne naïve](#)

[5.6 : Régression linéaire](#)

[5.7 : Machines à vecteurs de support \(SVM\)](#)

[5.8 : Arbres de décision et forêts aléatoires](#)

[5.9 : Analyse par composantes principales \(PCA\)](#)

[5.10 : Apprentissage par variété \(manifold\)](#)

[5.11 : Groupements en k-moyennes](#)

[5.12 : Modèles de mélange gaussien \(GMM\).](#)

[5.13 : Estimation par noyau](#)

[5.14 : Histogrammes de gradients orientés \(HOG\).](#)

[Autres ressources d'apprentissage machine](#)