



Avec les Nuls, tout devient facile !

Python

POUR

LES NULS

- ✓ Affichage des données
- ✓ Interagir avec des modules
- ✓ Gérer des listes et utiliser les classes
- ✓ Enregistrer des données dans des fichiers
- ✓ Installer de nouvelles bibliothèques



John Paul Mueller



Avec les Nuls, tout devient facile!

Python

POUR

LES NULS

- ✓ Affichage des données
- ✓ Interagir avec des modules
- ✓ Gérer des listes et utiliser les classes
- ✓ Enregistrer des données dans des fichiers
- ✓ Installer de nouvelles bibliothèques



John Paul Mueller

Python *POUR* ***LES NULS***

John Paul Mueller

FIRST
» Interactive

Python pour les Nuls

Titre de l'édition originale : *Python For Dummies*

Pour les Nuls est une marque déposée de Wiley Publishing, Inc

For Dummies est une marque déposée de Wiley Publishing, Inc

Collection dirigée par Jean-Pierre Cano

Mise à jour et révision : Daniel Rougé

Maquette : Marie Housseau

© Éditions First, un département d'Édi8, 2015

Éditions First, un département d'Édi8

12 avenue d'Italie 75013 Paris

Tél. : 01 44 16 09 00

Fax : 01 44 16 09 01

e-mail : firstinfo@efirst.com

Internet : www.editionsfirst.fr

ISBN : 978-2-7540-8321-8

ISBN numérique : 9782754085939

Dépôt légal : 1^{er} trimestre 2016

Cette œuvre est protégée par le droit d'auteur et strictement réservée à l'usage privé du client. Toute reproduction ou diffusion au profit de tiers, à titre gratuit ou onéreux, de tout ou partie de cette œuvre est strictement interdite et constitue une contrefaçon prévue par les articles L 335-2 et suivants du Code de la propriété intellectuelle. L'éditeur se réserve le droit de poursuivre toute atteinte à ses droits de propriété intellectuelle devant les juridictions civiles ou pénales.

Sommaire

[Page de titre](#)

[Page de copyright](#)

[Introduction](#)

[À propos de ce livre](#)

[Quelques suppositions un peu folles](#)

[Icônes utilisées dans ce livre](#)

[Et maintenant...](#)

[Première partie - Débuter avec Python](#)

[Chapitre 1 - Parler à votre ordinateur](#)

[Comprendre pourquoi vous voulez parler à votre ordinateur](#)

[Oui, une application est une forme de communication](#)

[Des procédures pour tous les jours](#)

[Écrire des procédures](#)

[Les applications sont des procédures comme les autres](#)

[Comprendre que les ordinateurs font les choses à la lettre](#)

[Définir ce qu'est une application](#)

[Comprendre que les ordinateurs utilisent un langage spécial](#)

[Aider les humains à parler à l'ordinateur](#)

[Comprendre pourquoi Python est si cool](#)

[Quelques bonnes raisons de choisir Python](#)

[Décider comment tirer un bénéfice personnel de Python](#)

[Quelles organisations utilisent Python ?](#)

[Trouver des applications Python utiles](#)

[Comparer Python avec d'autres langages](#)

[Chapitre 2 - Obtenir votre propre copie de Python](#)

[**Télécharger la version dont vous avez besoin**](#)

[**Installer Python**](#)

[**Travailler avec Windows**](#)

[**Travailler avec le Mac**](#)

[**Travailler avec Linux**](#)

[**Accéder à Python sur votre machine**](#)

[**Utiliser Windows**](#)

[**Utiliser le Mac**](#)

[**Utiliser Linux**](#)

[**Tester votre installation**](#)

[**Chapitre 3 - Interagir avec Python**](#)

[**Ouvrir la ligne de commandes**](#)

[**Lancer Python**](#)

[**Maîtriser la ligne de commandes**](#)

[**Tirer profit des variables d'environnement
Python**](#)

[**Taper une commande**](#)

[**Dire à l'ordinateur ce qu'il doit faire**](#)

[**Dire à l'ordinateur que vous avez terminé**](#)

[**Voir les résultats**](#)

[**Obtenir de l'aide de Python**](#)

[**Utiliser l'aide directe**](#)

[**Demander de l'aide**](#)

[**Quitter l'aide**](#)

[**Obtenir de l'aide directement**](#)

[**Refermer la ligne de commandes**](#)

[**Chapitre 4 - Écrire votre première application**](#)

[**Comprendre l'environnement de développement
intégré \(IDLE\)**](#)

[**Lancer IDLE**](#)

[**Utiliser des commandes standard**](#)

[**Comprendre le codage des couleurs**](#)

[**Obtenir de l'aide d'IDLE**](#)

[**Configurer IDLE**](#)

[**Créer l'application**](#)

[**Ouvrir une nouvelle fenêtre**](#)

[**Tapez les commandes**](#)

- Sauvegarder le fichier
- Exécuter l'application
- Comprendre l'utilisation des indentations
- Ajouter des commentaires
 - Comprendre les commentaires
 - Utiliser des commentaires comme aide-mémoire
 - Utiliser des commentaires pour empêcher du code de s'exécuter
- Charger et exécuter des applications existantes
 - Utiliser l'Invite de commandes ou une fenêtre de terminal
 - Utiliser la fenêtre d'édition
 - Utiliser Python en mode Shell ou Ligne de commandes
- Refermer IDLE

Deuxième partie - Apprendre la langue de Python

Chapitre 5 - Enregistrer et modifier des informations

- Enregistrer des informations
 - Considérer les variables comme des boîtes de rangement
 - Utiliser la bonne boîte pour enregistrer les bonnes données
- Python et ses principaux types de données
 - Placer des informations dans des variables
 - Comprendre les types numériques
 - Comprendre les valeurs booléennes
 - Comprendre les chaînes de caractères
- Travailler avec des dates et des heures

Chapitre 6 - Gérer l'information

- Contrôler la manière dont Python voit les données

Faire des comparaisons

Comprendre comment les ordinateurs effectuent des comparaisons

Travailler avec les opérateurs

Définir les opérateurs

Comprendre l'ordre de priorité des opérateurs

Créer et utiliser des fonctions

Voir les fonctions comme des boîtes de rangement pour le code

Comprendre la réutilisabilité du code

Définir une fonction

Accéder aux fonctions

Transmettre des informations aux fonctions

Renvoyer des informations depuis une fonction

Comparer les sorties de fonctions

Interagir avec l'utilisateur

Chapitre 7 - Prendre des décisions

Prendre des décisions simples avec l'instruction if

Comprendre l'instruction if

Utiliser l'instruction if dans une application

Choisir entre plusieurs alternatives avec l'instruction if...else

Comprendre l'instruction if...else

Utiliser l'instruction if...else dans une application

Utiliser l'instruction if...elif dans une application

Utiliser des décisions imbriquées

Utiliser des instructions if ou if...else multiples

Combiner d'autres types de décisions

Chapitre 8 - Effectuer des tâches répétitives

Traiter des données en utilisant l'instruction for

Comprendre l'instruction for

Créer une boucle for simple

Contrôler l'exécution avec l'instruction break

Contrôler l'exécution avec l'instruction continue
Contrôler l'exécution avec la clause pass
Contrôler l'exécution avec la clause else
Traiter des données avec l'instruction while
Comprendre l'instruction while
Utiliser l'instruction while dans une application
Imbriquer des boucles
Chapitre 9 - Les erreurs ? Quelles erreurs ?
Savoir pourquoi Python ne vous comprend pas
Prendre en considération les sources d'erreurs
Erreurs surgissant à un moment spécifique
Distinguer les types d'erreurs
Intercepter les exceptions
Gérer les exceptions de base
Gérer des exceptions en allant du plus spécifique au moins spécifique
Imbriquer des exceptions
Lever des exceptions
Lever des exceptions lors de conditions exceptionnelles
Passer des informations sur une erreur à l'appelant
Créer et utiliser des exceptions personnalisées
Utiliser la clause finally

Troisième partie - Effectuer des tâches courantes

Chapitre 10 - Interagir avec les modules
Créer des groupes de code
Importer des modules
Utiliser l'instruction import
Utiliser l'instruction from...import
Trouver des modules sur le disque
Voir le contenu d'un module

Utiliser la documentation des modules de Python

Ouvrir l'application pydoc

Utiliser les liens d'accès rapide

Taper un terme à rechercher

Voir les résultats

Chapitre 11 - Travailler avec les chaînes de caractères

Comprendre que les chaînes sont différentes

Définir une chaîne de caractères en utilisant des nombres

Utiliser des caractères pour créer des chaînes

Créer des chaînes comportant des caractères spéciaux

Sélectionner des caractères individuels

Trancher et couper les chaînes de caractères

Localiser une valeur dans une chaîne

Formater les chaînes de caractères

Chapitre 12 - Gérer des listes

Organiser les informations dans une application

Définir une organisation à l'aide de listes

Comprendre comment les ordinateurs voient les listes

Créer des listes

Accéder aux listes

Parcourir les listes

Modifier des listes

Faire des recherches dans les listes

Trier des listes

Travailler avec l'objet Counter

Chapitre 13 - Collecter toutes sortes de données

Comprendre les collections

Travailler avec les tuples

Travailler avec les dictionnaires

Créer et utiliser un dictionnaire

Remplacer l'instruction switch par un dictionnaire

Créer des piles en utilisant des listes

Travailler avec les files

Travailler avec des deque

Chapitre 14 - Créer et utiliser des classes

Comprendre les classes

Les classes et leurs composants

Créer la définition d'une classe

Les classes et leurs attributs intégrés

Travailler avec les méthodes

Travailler avec les constructeurs

Travailler avec les variables

Utiliser des méthodes avec des listes
d'arguments variables

Surcharger les opérateurs

Créer une classe

Utiliser la classe dans une application

Étendre des classes pour en créer de nouvelles

Construire la classe enfant

Tester la classe dans une application

Quatrième partie - Effectuer des tâches avancées

Chapitre 15 - Enregistrer des données dans des fichiers

Comprendre le fonctionnement des supports de
stockage permanents

Créer du contenu

Créer un fichier

Lire le contenu d'un fichier

Mettre à jour le contenu d'un fichier

Supprimer un fichier

Chapitre 16 - Envoyer un e-mail

Comprendre ce qui se passe lorsque vous
envoyez un e-mail

Un e-mail, c'est comme du courrier

Définir les parties de l'enveloppe

[Définir les parties du courrier](#)
[Créer un e-mail](#)
[Travailler avec un message texte](#)
[Travailler avec un message HTML](#)
[Consulter ses messages](#)

[Cinquième partie - Les Dix Commandements](#)

[Chapitre 17 - Dix ressources de programmation à découvrir](#)

[Travailler avec la documentation de Python en ligne](#)
[Utiliser le tutoriel LearnPython.org](#)
[Programmer pour le Web avec Python](#)
[Obtenir des bibliothèques supplémentaires](#)
[Créer des applications plus rapidement avec un environnement de développement interactif](#)
[Vérifier votre syntaxe avec plus de facilité](#)
[Utiliser XML à votre avantage](#)
[Éviter les erreurs courantes des débutants](#)
[Comprendre Unicode](#)
[Rendre vos applications Python plus rapides](#)

[Chapitre 18 - Dix domaines où faire fortune avec Python](#)

[Utiliser Python dans l'assurance qualité](#)
[Devenir responsable informatique dans une petite organisation](#)
[Applications et scripts Python](#)
[Administrer un réseau](#)
[Apprendre aux autres](#)
[Aider les gens à choisir un emplacement](#)
[Explorer des données](#)
[Interagir avec des systèmes embarqués](#)
[Travailler avec des données scientifiques](#)
[Effectuer des analyses de données en temps réel](#)

Chapitre 19 - Dix outils intéressants

Pister les bogues avec Roundup Issue Tracker

Créer un environnement virtuel avec VirtualEnv

Installer votre application avec PyInstaller

Construire une documentation de développement avec pdoc

Développer le code de l'application avec Komodo Edit

Déboguer votre application avec pydbgr

Entrer dans un environnement interactif avec IPython

Tester les applications Python avec PyUnit

Améliorer votre code avec Isort

Contrôler les versions en utilisant Mercurial

Chapitre 20 - Dix bibliothèques à connaître

Développer un environnement sécurisé avec PyCrypto

Interagir avec des bases de données grâce à SQLAlchemy

Voir le monde avec Google Maps

Ajouter une interface utilisateur graphique avec TkInter

Présenter des données tabulées avec PrettyTable

Sonoriser votre application avec PyAudio

Manipuler des images avec PyQtGraph

Localiser vos informations avec IRLib

Créer des liens avec Java en utilisant JPype

Accéder à des ressources réseau locales avec Twisted Matrix

Accéder à des ressources Internet en utilisant des bibliothèques

Index

Introduction

Vite ! Quel est le langage de programmation qui vous permet d'écrire rapidement des applications fonctionnelles et utilisables sur pratiquement n'importe quelle plate-forme ? Vous donnez votre langue au chat ? Oui, c'est Python. Ce qu'il y a de merveilleux avec Python, c'est effectivement que vous pouvez réellement écrire une application sur une plate-forme, et l'utiliser sur n'importe quelle autre plate-forme (à condition bien sûr qu'elle ne soit pas totalement exotique). Contrairement à d'autres langages de programmation, Python ne fait pas que faire des promesses : il les tient. Et, dans ce cas, le résultat vaut la promesse.

Python propose un code lisible et une syntaxe concise qui vous permet d'écrire des applications qui nécessitent moins de code que d'autres langages. De plus, il est utile dans toutes sortes de domaines. Certains le voient uniquement comme un langage de script, mais il vaut en réalité bien plus que cela (voyez par exemple à ce sujet le Chapitre 18).

À propos de ce livre

Ce livre aborde tout ce qui concerne l'installation, la mise en œuvre et surtout la mise en action rapide de Python. Vous voulez apprendre rapidement les bases de ce langage de manière à devenir productif dans votre travail, qui peut être à peu près quelconque, ou

encore dans la formation que vous suivez. Contrairement à nombres d'autres livres consacrés à ce sujet, celui-ci commence par le commencement en vous montrant ce qui rend Python différent des autres langages, et comment il peut vous aider dans vos activités quelles qu'elles soient. Ceci vous aidera à mieux comprendre comment atteindre vos objectifs personnels et à réaliser les tâches que vous vous fixerez, et ce en particulier grâce aux exemples (simples) proposés. Vous découvrirez même comment installer efficacement Python sur votre propre système.

Une fois Python correctement installé, vous pourrez commencer à travailler avec lui. Quand vous aurez étudié les exemples de ce livre, vous devriez être capable d'écrire des programmes simples et d'effectuer certaines tâches avancées, par exemple envoyer des e-mails. Certes, vous ne deviendrez pas un expert, mais vous serez capable d'utiliser Python pour des besoins spécifiques dans votre environnement professionnel. Pour rendre tout cela plus digeste, ce livre utilise quelques conventions :

- Le texte que vous êtes supposé taper au clavier est imprimé en caractères gras. D'accord, lorsque vous devez suivre une liste d'étapes, dont le texte est déjà mis en caractères gras, le texte à taper restera tel quel.
- Si vous voyez des caractères en *italiques* dans une séquence que vous devez saisir, cela signifie que vous devez les remplacer par quelque chose de plus personnel. Exemple : « Tapez **Votre nom** et appuyez sur Entrée ». Remplacez alors *Votre nom* par votre propre nom (ou par un autre, d'ailleurs).

➤ Les adresses Web apparaissent dans une `police particulière`. C'est juste pour ne pas les confondre avec autre chose.

En fait, c'est à peu près tout, et donc vous pouvez constater qu'il n'y a rien de particulièrement compliqué dans ces conventions.

Quelques suppositions un peu folles

Même si vous avez du mal à le croire, je ne suppose rien de particulier sur vous. Après tout, on ne se connaît pas ! Même si les suppositions sont généralement un peu vides de sens, il en faut tout de même quelques-unes pour se mettre bien d'accord dès le début.

Il est important que vous soyez familiarisé avec la plate-forme que vous utilisez, car vous ne trouverez rien de particulier dans ce livre à ce sujet (même si le Chapitre 2 vous explique comment installer Python sur les principales plates-formes). Dans le même esprit, nous ne traiterons pas non plus des problèmes potentiellement liés à telle ou telle plate-forme. En bref, vous devez savoir comment installer des applications en général, comment les utiliser, et plus généralement comment utiliser votre plate-forme préférée avant de commencer à travailler avec ce livre.

Nous allons aussi supposer que vous savez un certain nombre de choses sur l'Internet. En fait, vous trouverez dans ce livre de nombreuses références en ligne qui vous aideront à développer vos connaissances. Mais, comme il se doit, ces sources

d'information ne vous seront utiles que si vous êtes capable de les trouver. De plus, ces sources sont essentiellement en anglais. Une bonne pratique de la langue de nos amis anglo-saxons est donc un prérequis auquel vous ne pourrez pas échapper.

Icônes utilisées dans ce livre

Cette section décrit les icônes intéressantes (ou pas) que vous allez rencontrer dans ce livre.



Ces trucs et astuces sont utiles pour vous faire gagner du temps ou pour effectuer rapidement certaines tâches. Certaines pointent également vers des ressources que vous devriez essayer pour tirer le maximum de profit de Python.



Je ne veux pas donner l'impression d'être un vieux grincheux, mais vous devriez éviter de faire ce qui est signalé par cette icône. Sinon, il se pourrait bien que votre application n'entraîne que confusion chez vos utilisateurs, qui ne seront plus vos utilisateurs puisqu'ils refuseront de se servir de votre application...



Cette icône signale des informations ou des techniques avancées. Si cela vous ennuie, passez à la suite. Mais vous pouvez peut-être aussi prendre le temps de les lire, pour le cas où cela pourrait vous servir plus tard.



Cette icône est un rappel de ce que vous devriez

retenir. Il s'agit généralement d'un processus important, ou encore d'une information que vous devez connaître pour pouvoir écrire des applications Python fonctionnelles et efficaces.

Et maintenant...

Il est temps de débiter votre aventure avec Python. Si vous êtes totalement novice, vous devriez partir du Chapitre 1 et progresser à une vitesse vous permettant d'absorber autant de matériaux que possible. Mais sachez prendre votre temps, et revenir en arrière si c'est nécessaire. Python ne va pas mettre le feu au lac !

Si vous débutez, mais que vous êtes du genre pressé, vous pourriez partir du Chapitre 2 tout en comprenant que certaines notions pourraient vous sembler un peu confuses plus tard. Si Python est déjà installé sur votre système, vous pourriez être tenté de sauter directement au Chapitre 3, mais, dans ce cas, revenez très vite au Chapitre 2 pour ne pas risquer par la suite une sortie de route...

Si vous avez déjà un peu l'expérience de Python, vous pouvez gagner du temps en vous rendant directement au Chapitre 5. Vous pourrez toujours reprendre ultérieurement les chapitres précédents si vous avez des questions. Cependant, il est important de comprendre comment chaque exemple fonctionne avant de passer au suivant. Chaque exemple représente une leçon importante, et vous pourriez bien manquer des informations vitales si vous voulez aller trop vite.



Bien entendu, vous pouvez ressaisir tous les exemples manuellement. Mais vous aurez certainement envie de les avoir à portée de main, sans taper une par une toutes les lignes de tous les exemples, avec tout ce que cela comporte de risque d'erreur. C'est pourquoi vous pouvez télécharger les exemples de ce livre depuis l'adresse suivante :

<http://www.pourlesnuls.fr/>

Cliquez tout en bas de la page sur le lien Téléchargement. Dans la page qui va s'afficher, déroulez la liste et choisissez l'option Pour les Nuls Vie numérique. Il ne vous reste plus qu'à localiser l'image de couverture de ce livre, puis à cliquer dessus. Le fichier à télécharger est petit. Tout devrait donc aller très vite.

Et maintenant, bonne programmation avec Python !

Première partie

Débuter avec Python

Dans cette partie...

- ▶ La programmation, ses tenants et ses aboutissants, et ses rapports avec Python.
- ▶ Télécharger votre copie de Python et l'installer sur votre système.
- ▶ Travailler avec l'environnement interactif de Python.
- ▶ Créer votre première application avec Python.
- ▶ Comprendre l'importance des commentaires dans votre application.

Chapitre 1

Parler à votre ordinateur

Dans ce chapitre :

- ▶ Comprendre pourquoi vous voulez parler à votre ordinateur.
- ▶ Comprendre qu'une application est une forme de communication.
- ▶ Comprendre ce que font les applications, et pourquoi vous voulez en créer.
- ▶ Pourquoi utiliser Python comme langage de programmation ?

Avoir une conversation avec son ordinateur peut encore de nos jours paraître relever de la science-fiction. Après tout, les membres de l'équipage du vaisseau Enterprise dans *Star Trek* parlaient régulièrement à leurs ordinateurs. Mais, avec l'avènement de logiciels interactifs, comme Siri (Apple), Now (Google) ou encore Cortana (Microsoft), finalement rien de tout cela n'est plus inconcevable.



Demander vocalement à un ordinateur ou à un appareil portable comme un smartphone ou une tablette de donner certaines informations est une

chose. Mais fournir des instructions à la machine en est une autre. Ce chapitre (comme évidemment tout ce livre) considère que vous voulez donner des instructions à votre ordinateur et en obtenir des résultats. Vous y découvrirez également qu'il est nécessaire de faire appel à un langage spécial pour effectuer ce type de communication, et pourquoi vous devez utiliser le langage Python pour y arriver. Mais l'idée principale à retirer de ce chapitre, c'est que la programmation est simplement une forme de communication, qui vient prendre sa place parmi les autres formes de communication dont vous vous servez déjà avec votre ordinateur.

Comprendre pourquoi vous voulez parler à votre ordinateur

Parler (j'emploie ce verbe au sens large) à une machine peut sembler un peu bizarre au premier abord, mais c'est nécessaire, car l'ordinateur ne peut pas lire dans votre cerveau. Du moins, pas encore. Et même s'il était capable de déchiffrer vos pensées, il ne ferait que continuer à communiquer avec vous. Rien ne peut se produire sans cet échange d'informations entre la machine et vous. Des activités telles que celles-ci :

- Lire vos messages électroniques
- Écrire quelque chose à propos de vos vacances
- Trouver le plus beau cadeau du monde

... sont toutes des exemples de communication entre vous et l'ordinateur. Et le fait que, pour atteindre les buts que vous lui assignez, il doive pousser plus loin

l'interaction avec d'autres machines, ou d'autres personnes, pour leur transmettre vos requêtes n'y change rien. Il s'agit tout simplement d'une extension de l'idée de base, à savoir que communiquer est une condition nécessaire pour obtenir n'importe quel résultat.

Dans la plupart des cas, tout cela se déroule de manière pratiquement invisible pour vous, à moins que vous n'y réfléchissiez d'un peu plus près. Supposons par exemple que vous dialoguiez avec une autre personne en ligne (disons par l'intermédiaire d'un *messenger* quelconque). En réalité, vous communiquez avec votre ordinateur (ou votre smartphone), qui lui-même communique avec l'ordinateur (ou le smartphone) de la personne qui est à l'autre bout via un « salon où l'on cause » quelconque, et cet ordinateur lointain communique avec votre correspondant ou votre correspondante. Évidemment, le même principe s'applique dans l'autre sens... La [Figure 1.1](#) illustre ce concept.

Figure 1.1 : Les communications entre personnes et ordinateurs s'effectuent de manière invisible, transparente.



Remarquez l'espèce de nuage au centre de la [Figure 1.1](#). Il pourrait contenir à peu près n'importe quoi, mais vous savez qu'on peut au moins y trouver d'autres ordinateurs qui exécutent certaines applications. C'est grâce à eux que vous et votre correspondant(e) pouvez dialoguer. De votre point de vue, c'est quelque chose de très facile, voire même d'évident. Toute cette chaîne travaille à votre service

en arrière-plan, de façon invisible, et vous avez simplement l'impression de converser comme si vous vous trouviez dans la même pièce (enfin presque).

Oui, une application est une forme de communication

La communication avec les ordinateurs s'établit dès lors que vous vous servez d'applications. Par exemple, vous utilisez une application pour répondre à vos messages, une autre pour acheter quelque chose, et une autre encore pour créer un diaporama. Une *application* (certains l'appellent maintenant une *app*) fournit le moyen d'exprimer des idées humaines devant un ordinateur, tout en permettant à celui-ci de les comprendre et en définissant les outils grâce auxquels ce dernier est capable de gérer ces processus de communication. Les données utilisées pour exprimer le contenu d'une présentation sont évidemment différentes de celles qui concernent un cadeau pour votre maman. Par conséquent, les manières dont vous les visualisez, les utilisez et les comprenez sont différentes pour chaque tâche. C'est pourquoi vous devez utiliser des applications diverses de manière à interagir avec toutes ces données pour que l'ordinateur et vous arriviez à vous comprendre.

Il est possible de trouver des applications qui répondent à vos besoins quotidiens. En fait, il en existe vraisemblablement des tas qui sont capables de le faire, sans que vous le sachiez. Les programmeurs ont travaillé dur pour créer des millions et des millions d'applications de tous types depuis de nombreuses années. Mais cela ne répond pas forcément à votre question. La réponse tient dans

une réflexion sur vos données et sur la manière dont vous voulez interagir avec elles. Certaines de ces données peuvent être tellement spécifiques qu'aucun programmeur ne s'est penché dessus. À moins que vous n'ayez besoin d'un format de données totalement nouveau. Dans ce cas, le dialogue avec l'ordinateur ne peut pas s'établir à moins que vous ne définissiez vous-même une application capable de traiter vos données.

Les sections qui suivent décrivent cette notion d'application du point de vue de *vos* données, avec les traitements dont *vous* avez besoin. Vous pourriez par exemple vouloir accéder à une bibliothèque d'enregistrements vidéo, mais sans pouvoir y accéder d'une manière qui réponde à vos souhaits. Ces données sont donc uniques, et vos besoins d'accès très spécifiques. Vous allez donc devoir créer une application qui réponde à ces deux problèmes : quelles sont vos données, et comment pouvez-vous y accéder ?

Des procédures pour tous les jours

Une *procédure*, c'est simplement une série d'étapes à exécuter pour effectuer une certaine tâche. Pour griller du pain, par exemple, vous devez suivre une procédure comme celle-ci :

- Vous sortez le pain et le beurre du réfrigérateur.
- Vous ouvrez le sachet du pain et vous sortez deux tartines.
- Vous ouvrez le couvercle du grille-pain (s'il en

a un).

- Vous insérez vos deux tranches de pain dans l'appareil.
- Vous appuyez sur le bouton *ad-hoc* du grille-pain pour lancer la cuisson.
- Vous attendez que le grille-pain vous rende les tartines.
- Vous retirez les tartines du grille-pain.
- Vous placez les tartines sur un plateau.
- Vous beurrez les tartines.

D'accord, il peut y avoir des variations, et chacun prépare son petit-déjeuner à sa manière. Mais il y a un point qui met tout le monde d'accord : vous ne beurrez pas vos tartines *avant* de les mettre dans le grille-pain. Si ? Excusez-moi. À chacun ses préférences. Mais, dans tous les cas, reconnaissons que la plupart des gens effectuent tous ces gestes mécaniquement, sans y réfléchir. Pourtant, ils répondent bel et bien à une *procédure* qui ne varie pas d'une journée à l'autre (enfin, sauf si vous êtes en retard pour aller travailler).



Les ordinateurs sont incapables d'exécuter des tâches en l'absence de procédures. Autrement dit, vous devez lui expliquer quelles étapes il doit effectuer, dans quel ordre, et quels sont les cas particuliers (les *exceptions*) auxquels il doit répondre. Toutes ces informations (et plus encore) sont définies à l'intérieur d'une application. Pour faire court, une application est simplement une procédure écrite que vous utilisez pour expliquer à l'ordinateur ce qu'il doit faire, quand il doit le faire, et comment il doit le faire. Du fait que vous utilisez des procédures dans pratiquement chaque instant de votre vie, tout ce que vous avez à faire, c'est d'appliquer des connaissances que vous

possédez déjà pour expliquer à un ordinateur ce qu'il doit réaliser. D'accord, c'est un peu simpliste, mais vous voyez l'idée.

Écrire des procédures

Lorsque j'étais écolier, notre institutrice nous demandait régulièrement d'écrire sur un papier une série d'étapes, par exemple pour faire griller du pain comme dans l'exemple ci-dessus. Chaque papier était lu, et on expérimentait ce qu'il décrivait. D'accord, on faisait parfois des erreurs. Je me souviens d'une fois où j'avais oublié de mentionner qu'il fallait enlever les tranches du grille-pain avant de les tartiner. Je n'ose même pas vous raconter le résultat... Mais ces leçons m'ont été très utiles. Il est vrai qu'écrire des procédures n'est pas un long fleuve tranquille, et que souvent nous oublions de préciser telle ou telle étape en supposant que les autres vont comprendre ce que nous voulons faire. Mais c'est une mauvaise hypothèse.

Dans la vraie vie, de nombreuses expériences reposent sur la mise en œuvre de procédures. Songez un instant à la « check-list » que doivent respecter les pilotes avant un décollage. Sans une procédure bien rodée, l'avion risquerait bien de s'écraser. Apprendre à écrire une bonne procédure prend du temps, mais cela en vaut la peine. Cela ne marche généralement pas du premier coup, mais c'est quand même une possibilité. Et écrire une procédure ne suffit pas. Il faut aussi la tester, notamment en faisant appel à une ou plusieurs personnes qui ne connaissent pas la règle du jeu et ne sont pas familiarisées avec les tâches à mettre en œuvre. Lorsque vous travaillez avec un

ordinateur, c'est lui qui est votre premier et meilleur cobaye. Et, croyez-moi, les enfants en sont encore de bien meilleurs pour trouver les failles dans vos procédures...

Les applications sont des procédures comme les autres

Lorsque vous écrivez une application, vous écrivez en réalité une procédure qui définit une série d'étapes que l'ordinateur devrait exécuter pour accomplir certaines tâches, ce que vous voulez lui voir faire. Si vous manquez une étape, vous n'obtiendrez pas les résultats escomptés. L'ordinateur ne sait pas ce que vous voulez dire, ou ce que vous avez l'intention de lui faire réaliser. La seule chose qu'il comprenne, c'est que vous lui avez fourni une procédure spécifique, et qu'il doit suivre les étapes de celle-ci.

Comprendre que les ordinateurs font les choses à la lettre

Il y a sans doute des personnes qui suivent des procédures que vous avez créées. Elles compensent automatiquement les déficiences de votre procédure en fonction de leur propre expérience, ou prennent des notes pour signaler ce qui ne va pas. En d'autres termes, les êtres humains sont capables d'interpréter vos intentions pour en combler les lacunes.



Lorsque vous commencez à écrire des programmes pour ordinateur, vous ressentez généralement de la

frustration, car la machine effectue les tâches qui lui sont dévolues d'une manière totalement précise, et elle prend vos instructions au pied de la lettre. Par exemple, si vous dites à l'ordinateur qu'une certaine valeur devrait être égale à 5, l'ordinateur recherche une valeur strictement égale à 5. Point. Un humain pourrait voir 4,95, et comprendre que cette valeur répond en fait à votre demande. Pour l'ordinateur, 4,95 et 5 sont deux choses bien différentes. En bref, les ordinateurs sont inflexibles, pas du tout intuitifs et manquent totalement d'imagination. Lorsque vous écrivez une procédure informatique, l'ordinateur doit faire ce que vous demandez avec une précision totale, sans jamais varier dans le temps, et sans jamais modifier ou interpréter votre procédure.

Définir ce qu'est une application

Comme je l'ai déjà mentionné, les applications fournissent des moyens pour définir des idées humaines avec précision, et d'une manière telle qu'elle soit compréhensible par un ordinateur. Pour atteindre cet objectif, l'application s'appuie sur une ou plusieurs procédures qui décrivent comment manipuler certaines données, et comment les présenter. Par exemple, vous pouvez voir sur votre écran quelque chose qui est affiché par votre traitement de texte. Mais, pour cela, l'ordinateur a besoin de procédures permettant de retrouver les données qui forment ce texte sur le disque, leur donner une mise en forme compréhensible par vous, puis vous présenter le résultat. Les sections qui suivent définissent les spécificités d'une application plus en détail.

Comprendre que les ordinateurs utilisent un langage spécial

Les langues humaines sont complexes et difficiles à comprendre. Même des applications telles que Siri, Google Now ou Cortana ont de sérieuses limites quant à la compréhension de ce que vous dites. Au fil des ans, les ordinateurs (ou plutôt ceux qui développent des applications pour les ordinateurs) ont acquis la possibilité de transformer la voix en données et de comprendre que certains mots sont des commandes qu'on veut les voir exécuter. Pour autant, cette compréhension ne dépasse pas un certain niveau de complexité. Prenons l'exemple d'un texte juridique. Si vous lisez un compte-rendu de jugement, vous allez penser qu'il s'agit d'un langage extraterrestre. Cependant, le but est de transcrire par des mots des idées et des concepts d'une manière qui ne soit pas sujette à l'interprétation. Cet objectif est difficile à atteindre, justement parce que le langage humain est imprécis.

De leur côté, les ordinateurs sont, par essence, incapables de s'appuyer uniquement sur votre langage pour comprendre les procédures que vous écrivez. Comme je l'ai dit plus haut, ils prennent celles-ci au pied de la lettre. Si vous écriviez des applications comme vous parlez, les résultats seraient totalement imprévisibles. C'est pourquoi vous devez utiliser un langage spécial, appelé un *langage de programmation*, pour communiquer avec les ordinateurs. Ce, ou plutôt ces langages spéciaux rendent possible l'écriture de procédures qui sont à la fois spécifiques et totalement compréhensibles aussi bien par les humains que par les ordinateurs.



En fait, les ordinateurs ne parlent aucun langage. Ils utilisent des codes binaires pour modifier en interne des sortes de bascules et effectuer des tas de calculs mathématiques. Ils ne comprennent même pas les lettres, rien que les nombres. Une application particulière transforme le langage spécifique que vous utilisez pour communiquer avec l'ordinateur en codes binaires (des 0 et des 1, et c'est tout). Évidemment, c'est là un sujet que vous n'avez en aucun cas besoin d'approfondir pour lire ce livre. Mais il est intéressant de savoir que les ordinateurs ne connaissent que les chiffres, et donc ne parlent aucune langue.

Aider les humains à parler à l'ordinateur

Il est important de ne jamais perdre de vue l'objectif que vous poursuivez lorsque vous écrivez une application. Celle-ci sert à aider les humains à dialoguer avec l'ordinateur d'une certaine manière. Toute application travaille avec certains types de données qui sont entrées, stockées, manipulées et « recrachées » pour fournir les résultats désirés. Que cette application soit un jeu ou un tableur, l'idée de base est exactement la même. Les ordinateurs travaillent avec des données fournies par des êtres humains pour délivrer un certain résultat.

Lorsque vous créez une application, vous fournissez une nouvelle méthode pour le dialogue entre l'homme et la machine. La nouvelle approche que vous construisez permettra à d'autres humains de voir des données d'une nouvelle façon. La communication

homme/machine devrait être suffisamment simple pour que l'application semble disparaître de la vue. Réfléchissez à votre propre expérience d'utilisateur. Les meilleures applications sont bien celles qui vous permettent de vous concentrer uniquement sur les données avec lesquelles vous interagissez. Par exemple, un jeu ne sera véritablement « immersif » que si vous n'avez à vous occuper que de la planète que vous essayez de sauver, ou des ennemis que vous devez combattre, et pas du tout de l'application qui vous permet de faire tout cela.



Une des meilleures façons de commencer à réfléchir à une application que vous voudriez créer, c'est de regarder comme d'autres gens procèdent. Écrire ce que vous aimez ou pas dans d'autres applications est un bon point départ pour définir ce que vous voulez faire, comment vous voulez le faire, et comment tout cela devrait se présenter aux yeux du monde. Voici quelques questions que vous pouvez vous poser au cours de ce processus mental :

- Qu'est-ce qui m'a gêné dans l'application ?
- Quelles fonctions étaient faciles à utiliser ?
- Quelles fonctions étaient difficiles à utiliser ?
- Est-ce qu'il était facile ou pas d'interagir avec mes données ?
- Comment serait-il possible d'améliorer ces interactions ?
- Qu'est-ce que je pourrais faire pour améliorer une telle application ?

Les développeurs professionnels se posent bien d'autres questions lorsqu'ils développent des applications. Mais celles-ci constituent un bon point de départ, car elles vous aident à mieux percevoir les

applications comme des moyens d'aider les êtres humains à communiquer avec les ordinateurs. Si vous vous êtes déjà senti frustré en utilisant une application, vous pouvez comprendre ce que vos utilisateurs pourront ressentir si vous ne vous êtes pas posé les bonnes questions au cours de votre travail de développement. La communication est l'élément le plus important de toute application que vous êtes appelé à créer.

Vous pouvez aussi commencer à réfléchir à la manière dont vous travaillez. Commencez à écrire des procédures pour des choses que *vous* faites. Une bonne idée consiste à prendre le processus étape par étape, en écrivant à chaque fois toutes vos réflexions. Lorsque vous avez terminé, demandez à quelqu'un d'autre de tester votre procédure afin de voir si elle marche effectivement. Vous pourriez être surpris de découvrir que, même en fournissant de gros efforts, vous pouvez facilement oublier d'inclure certaines étapes.



La pire application du monde débute généralement par le fait que le programmeur ne sait pas exactement ce que cette application est censée faire, en quoi elle est spéciale, ce qu'elle doit apporter, et à qui elle est destinée. Lorsque vous décidez de créer une application, assurez-vous que vous savez pourquoi vous le faites, et quel est l'objectif que vous voulez atteindre. Mettre un plan de bataille en place aide beaucoup à bien programmer, et à y prendre du plaisir (ce qui a son importance). Vous pourrez ainsi travailler posément sur votre nouvelle application et voir les étapes se réaliser les unes après les autres jusqu'à ce que tout soit opérationnel. Après quoi, vos amis à qui vous demanderez de tester votre

application trouveront qu'elle est vraiment *cool*, et vous aussi.

Comprendre pourquoi Python est si cool

Il existe de nombreux langages de programmation. En fait, un étudiant pourrait passer tout un semestre à étudier des langages de programmation, et ne même pas entendre parler de certains d'entre eux. Vous pourriez penser que les développeurs sont parfaitement à l'aise avec tout cela, et qu'ils en choisissent simplement un pour communiquer avec l'ordinateur. Mais ce n'est que le petit bout de la lorgnette.



Il y a de bonnes raisons qui justifient la création de nouveaux langages. Chaque langage a quelque chose de particulier à offrir, quelque chose qu'il fait particulièrement bien. De plus, la technologie évolue sans cesse, et donc les langages de programmation doivent eux aussi progresser. Du fait que créer une application est une affaire de communication efficace, de nombreux programmeurs connaissent plusieurs langages de manière à pouvoir choisir celui qui convient le mieux pour réaliser une tâche particulière. Par exemple, un certain langage sera meilleur pour gérer une base de données, tandis qu'un autre sera particulièrement adapté pour créer une interface utilisateur.

Comme tous les langages de programmation, Python fait certaines choses exceptionnellement bien ; vous devez donc savoir quels sont ses principaux champs de compétence avant de commencer à l'utiliser. Vous pourriez bien être étonné par les choses réellement

cool que vous pouvez réaliser avec Python. Connaître les forces et les faiblesses d'un langage de programmation vous aide non seulement à en tirer le meilleur parti, mais aussi à éviter les frustrations dues à l'emploi d'un langage pour des choses qu'il ne fait pas très bien. Les sections qui suivent devraient vous aider à prendre ce genre de décision.

Quelques bonnes raisons de choisir Python

La plupart des langages de programmation sont créés pour répondre à des besoins spécifiques. Ces besoins aident à définir les caractéristiques du langage, et à déterminer ce que vous pouvez faire avec lui. En fait, il ne peut pas exister de langage « universel », car les gens ont des buts et des besoins extrêmement variés lorsqu'ils créent des applications. Dans le cas de Python, l'objectif principal était de produire un langage de programmation qui aide les programmeurs à être efficaces et productifs. En gardant cela présent à l'esprit, voici quelques (bonnes) raisons d'utiliser Python pour créer vos applications :

✓ **Moins de temps de développement** : Le code Python est généralement deux à trois fois plus court que l'équivalent écrit en C/C++ ou en Java. Vous passez donc moins de temps à écrire votre application, et plus de temps à l'utiliser.

✓ **Facilité de lecture** : Un langage de programmation est comme tout autre langage, vous devez être capable de le lire pour comprendre ce qu'il fait. Le code Python tend à être plus facile à lire que d'autres, ce qui

signifie que vous passez moins de temps à l'interpréter, et plus de temps à apporter les modifications essentielles.

✓ **Temps d'apprentissage réduit** : Les créateurs de Python ont voulu fabriquer un langage de programmation avec le moins de règles complexes possible pouvant rendre difficile son apprentissage. Après tout, les programmeurs veulent créer des applications, pas apprendre de nouvelles langues obscures et tarabiscotées.



Vous devez comprendre que si Python est un langage populaire, ce n'est pas le *plus* populaire de tous. Ainsi, si vous envisagiez de chercher du travail dans le milieu de la programmation, il faut reconnaître que Python serait un bon choix, mais que C/C++, Java, C# ou même Visual Basic seraient de meilleurs choix. Assurez-vous que vous choisissiez un langage que vous aimez, et qui soit en même temps bien adapté aux besoins du développement de votre application, de même qu'aux objectifs que vous poursuivez. Python a été le langage de l'année en 2007 et en 2010, et faisait partie du quatuor de tête en 2011. Dans le même temps, Python est aussi un langage de choix pour l'apprentissage de la programmation dans nombre d'établissements scolaires.

Décider comment tirer un bénéfice personnel de Python

En dernier ressort, vous pouvez utiliser n'importe quel langage de programmation pour écrire toutes sortes

d'applications. Mais si vous vous trompez dans votre choix, le processus risque d'être lent, avec de gros risques d'erreurs ou de *bogues* (de fautes, si vous voulez), et vous allez détester ce langage. Mais vous pouvez quand même arriver au bout de vos peines. Bien entendu, personne n'a envie de vivre une telle expérience, et il est donc important de savoir quels types d'applications sont généralement créés avec Python. En voici quelques exemples (évidemment non exhaustifs) :

➤ **Créer des exemples de « maquettes » d'applications :**

Les développeurs ont souvent besoin de produire un *prototype*, disons comme une maquette d'application, avant d'obtenir les ressources qui leur permettront de créer l'application elle-même. Python est un outil adapté à la productivité, ce qui permet de fabriquer rapidement de tels prototypes.

➤ **Créer des scripts pour des applications Web :**

Même si JavaScript est probablement le langage de programmation le plus populaire dans ce domaine, Python est un second tout proche de la première place. Il offre en particulier des fonctionnalités que ne propose pas JavaScript, et sa grande efficacité accélère la création de ce type d'applications (ce qui est un plus dans notre monde en accélération permanente).

➤ **Concevoir des applications mathématiques, scientifiques ou d'ingénierie :**

Il est intéressant de savoir que Python donne accès à des bibliothèques qui facilitent grandement la création de telles applications. Les deux bibliothèques les plus populaires sont NumPy (www.numpy.org/) et SciPy (www.scipy.org/). Elles permettent de

réduire considérablement le temps passé à écrire des applications à caractère scientifique.

✓ **Travailler avec XML :** XML (pour eXtensible Markup Language, ou langage de balises étendu) est la base utilisée pour le stockage de la plupart des données sur l'Internet, ainsi que par de nombreuses applications du bureau. Contrairement à la plupart des langages, pour qui XML est bien souvent comme une pièce rapportée, Python en fait un citoyen de première classe. Si vous avez besoin de travailler avec un service Web, ce qui est la principale méthode pour échanger des informations sur l'Internet (ou toute autre application faisant un usage intensif du XML), Python est un excellent choix.

✓ **Interagir avec des bases de données :** L'industrie et les services reposent très largement de nos jours sur la gestion de bases de données. Python n'est pas un langage de requête, comme SQL, mais il est très bon dans l'interaction avec les bases de données. Il rend la création de connexions et de manipulation des données relativement aisée.

✓ **Développer des interfaces utilisateur :** Python ne ressemble pas à d'autres langages comme C#, où vous disposez d'un outil de conception intégré pour réaliser une interface utilisateur et où vous pouvez simplement déposer des éléments par glisser-déposer à partir d'une boîte à outils pour créer une interface utilisateur. Cependant, il possède de multiples extensions qui rendent ce genre d'interface facile à fabriquer (voyez l'adresse wiki.python.org/GuiProgramming pour plus de détails). En fait, Python n'est pas lié à tel ou tel type d'interface. Vous pouvez utiliser la

méthode qui convient le mieux à vos besoins.

Quelles organisations utilisent Python ?

Python est très bien adapté aux tâches pour lesquelles il a été conçu. Cela va sans dire, mais encore mieux en le disant... En fait, c'est la raison pour laquelle de nombreuses organisations se servent de Python pour des prototypes de développement de leurs propres applications (et bien entendu plus si affinité). Vous avez donc besoin de connaître un tel langage de programmation, car ces grandes organisations ont tendance à investir dans son développement pour le rendre meilleur.

Citons donc quelques noms connus :

- **Go.com** (go.com) : Application Web.
- **Google** (www.google.com) : Moteur de recherche.
- **NASA** (www.nasa.gov) : Applications scientifiques.
- **Nyse - New York Stock Exchange** (nyse.nyx.com) : Applications Web.
- **Redhat** (www.redhat.com) : Outils d'installation Linux.
- **Yahoo !** (www.yahoo.com) : Des parties de la messagerie Yahoo !
- **YouTube** (www.youtube.com) : Moteur graphique.



Cette liste ne constitue évidemment qu'une toute

petite partie des organisations qui utilisent Python d'une manière extensive. Pour en savoir plus à ce sujet, voyez l'adresse www.python.org/about/success/. Le nombre de ces « success stories » est devenu si important que cette liste est probablement très incomplète et qu'il faudrait créer de nouvelles catégories pour mieux prendre conscience de cette réussite.

Trouver des applications Python utiles

Vous pouvez très bien avoir une application Python en pleine activité sur votre ordinateur sans que vous en ayez même conscience. Python est en effet utilisé de nos jours dans de multiples types d'applications. Cela va des utilitaires qui fonctionnent dans une console (une fenêtre de commandes, si vous préférez) à des suites dédiées à la Conception Assistée par Ordinateur (la CAO). Certaines applications sont dédiées aux appareils portables, tandis que d'autres sont réservées à des services pour de grandes entreprises. En résumé, aucun domaine n'échappe aux applications écrites en Python. Pour autant, rien ne remplace l'expérience que vous pouvez retirer de celle d'autres utilisateurs. Vous pouvez trouver de multiples références en ligne, mais le meilleur point de départ se trouve à l'adresse wiki.python.org/moin/Applications.



Désolé, mais vous aurez besoin de réviser votre anglais pour accéder à toutes ces ressources !

En tant que programmeur Python, vous avez aussi

envie de savoir que nombre d'outils de développement sont facilement accessibles afin de vous faciliter l'existence. Un *outil de développement* fournit certains niveaux d'automatisation lorsque vous écrivez des procédures qui indiquent à l'ordinateur ce qu'il doit faire. Plus il y a d'outils de développement, et moins vous avez à travailler pour créer une application parfaitement fonctionnelle. Les développeurs adorent partager leurs listes d'outils préférés. Mais, en tout état de cause, vous devriez pouvoir trouver votre bonheur (ou une bonne partie de celui-ci) en consultant la page www.python.org/about/apps/.



Des outils (ou bibliothèques) tels que NumPy ou SciPy ont déjà été abordés dans ce chapitre. La suite de ce livre liste d'autres outils. En fait, vous devez simplement vous assurer que vous téléchargez et installez ce dont vous avez besoin pour continuer à progresser.

Comparer Python avec d'autres langages

Comparer un langage avec un autre est un exercice assez dangereux, car il s'agit à la fois d'une question de préférences personnelles et de besoins à assurer, comme d'ailleurs dans toute démarche structurée et scientifique. Il est donc important de bien comprendre que Python n'est *pas* le langage universel dont tout le monde peut rêver en vain. Il n'y a pas de *meilleur* langage de programmation. Il y a simplement des langages qui sont mieux adaptés pour tel ou tel type d'application. Cela étant posé, vous pouvez

également vous référer aux comparaisons consultables sur la page wiki.python.org/moin/LanguageComparisons.

C#

Des tas de gens clament que Microsoft a simplement copié Java pour créer C#. Cela étant dit, il est vrai que C# a certains avantages (ou désavantages) comparativement à Java. La première remarque (incontestable) est que C# est plus facile à apprendre et à utiliser. Par contre, et puisque nous parlons ici de Python, celui-ci a plusieurs avantages :

- ✓ Il est incontestablement plus facile à apprendre.
- ✓ Son code est plus concis.
- ✓ Il est totalement gratuit (ou *open source* comme on dit).
- ✓ Il supporte mieux les plates-formes multiples.
- ✓ Il permet d'utiliser facilement des environnements de développement variés.
- ✓ Il est facile à étendre via des composants Java ou C/C++.
- ✓ Il supporte mieux les développements scientifiques.

Java

Depuis bien longtemps, les développeurs recherchent un langage qui leur permettrait d'écrire une application une fois, et de la faire exécuter en tout temps et partout. Java est conçu pour être utilisé sur de multiples plates-formes. Il repose pour cela sur certains « trucs » que vous découvrirez plus tard dans ce livre afin d'accomplir cette « magie ». Pour l'instant, et pour ce qui nous concerne ici, il vous suffit de savoir que Java a rencontré un tel succès qu'il a

été largement copié (avec un niveau de succès très inégal). Mais Python conserve un certain nombre d'avantages :

- Il est nettement plus facile à apprendre.
- Son code est bien plus concis.
- Les variables de Python (ces choses qui sont stockées dans la mémoire de votre ordinateur) peuvent contenir des données non seulement extrêmement variées, mais aussi totalement dynamiques.
- Les développements sont bien plus rapides.

Pas mal, non ?

Perl

Perl est l'acronyme de Practical Extraction and Report Language (« langage d'extraction et de rapport pratique », ce qui n'est pas très joli, non ?). Le point fort de Perl, c'est l'extraction d'informations depuis des bases de données et la présentation de celles-ci. Bien entendu, Perl s'est développé bien au-delà de ses sources. Vous pouvez donc l'utiliser pour écrire toutes sortes d'applications. Mais, comparativement à Perl, Python conserve là encore un certain nombre d'avantages :

- Il est plus simple à apprendre.
- Il est plus simple à lire.
- Il protège mieux les données.
- Il s'intègre mieux avec Java.
- Il est moins dépendant des diverses plateformes.

Chapitre 2

Obtenir votre propre copie de Python

Dans ce chapitre :

- ▶ Obtenir une copie de Python pour votre système.
- ▶ Effectuer l'installation de Python.
- ▶ Trouver et utiliser Python sur votre système.
- ▶ S'assurer du bon fonctionnement de l'installation.

Créer des applications nécessite que vous possédiez une autre application (à moins de vouloir communiquer avec l'ordinateur en code binaire, une expérience que même les programmeurs chevronnés évitent). Si vous voulez écrire une application dans le langage Python, vous avez donc besoin des applications qui vous permettent de le faire. Ces applications vous aideront à créer votre code Python, vous aideront dans ce travail, et vous permettront ensuite d'exécuter ce code.

Dans ce chapitre, nous allons voir comment obtenir votre copie de l'application Python, comment

l'installer sur votre disque dur, comment localiser cette installation pour pouvoir l'utiliser, et enfin comment la tester afin de vérifier que tout fonctionne correctement.

Télécharger la version dont vous avez besoin

Chaque *plate-forme* (la combinaison entre le matériel dont est composé l'ordinateur et le système d'exploitation de celui-ci) a des règles spéciales qu'elle suit lorsqu'elle exécute des applications. Mais Python vous cache fort heureusement tous ces détails. Vous tapez du code qui s'exécute sur n'importe quelle plate-forme supportée par Python, et celui-ci convertit votre code dans quelque chose de compréhensible par cette plate-forme.

Pour que cette traduction puisse se faire, vous devez donc posséder une version de Python fonctionnant sur votre plate-forme particulière. La liste des plates-formes supportées par Python est copieuse :

- Advanced IBM Unix (AIX)
- Amiga Research OS (AROS)
- Application System 400 (AS/400)
- BeOS
- Hewlett-Packard Unix (HP-UX)
- Linux
- Mac OS X (préinstallé avec le système)
- Microsoft Disk Operating System (MS-DOS)
- MorphOS
- Operating System 2 (OS/2)
- Operating System 390 (OS/390) et z/OS
- PalmOS

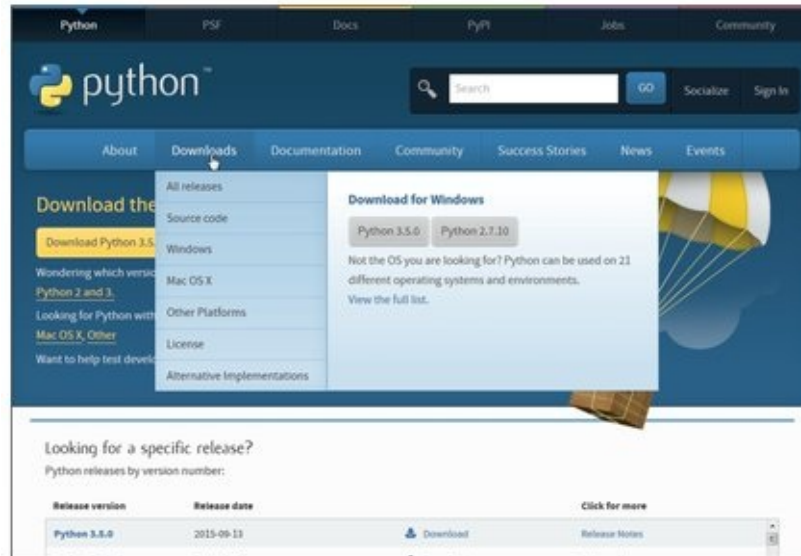
- ✓ PlayStation
- ✓ Psion
- ✓ QNX
- ✓ RISC OS (auparavant Acorn)
- ✓ Series 60
- ✓ Solaris
- ✓ Virtual Memory System (VMS)
- ✓ Windows 32-bit (XP et suivants)
- ✓ Windows 64-bit
- ✓ Windows CE/Pocket PC



Très impressionnant, non ? Les exemples de ce livre ont été testés sous Windows, Mac OS X et Linux. Mais ils pourraient parfaitement fonctionner sur toutes les autres plates-formes, puisque le code n'est lié à aucune d'entre elles, uniquement à Python. Au moment où ce livre est écrit, Python en est à sa version 3.5.0.

Pour obtenir la version de Python adaptée à votre propre plate-forme, vous devez accéder dans votre navigateur Web au site www.python.org. Dans la page d'accueil, cliquez sur le bouton Download (voir la [Figure 2.1](#)). Le site vous propose en priorité Windows et Mac OS X (ce sont les plus répandus), mais vous pouvez aussi cliquer sur le lien Other Platforms pour accéder à la liste des plates-formes supportées par Python (voir la [Figure 2.2](#)).

Figure 2.1 : La page d'accueil de Python.

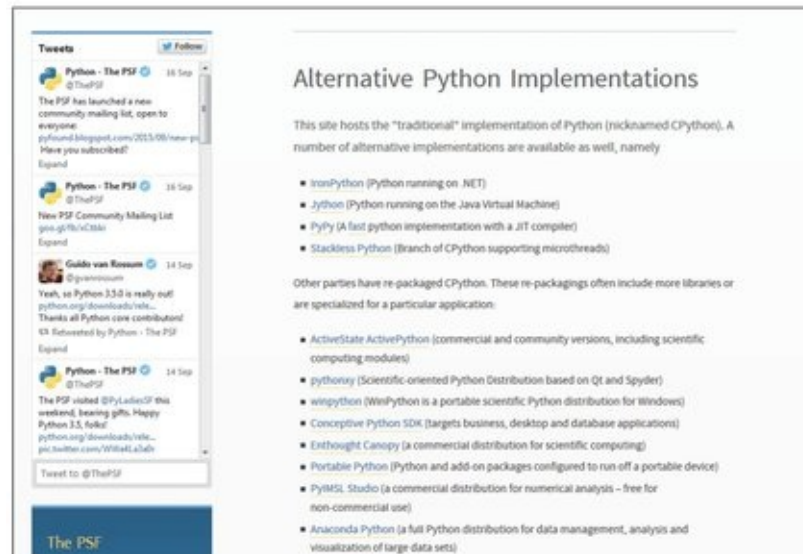


Nous nous en tiendrons dans ce livre à l'implémentation « traditionnelle » de Python (appelée CPython). Mais le lien Alternative Implementations vous permet si vous le souhaitez d'accéder à toute une série de variantes proposant par exemple un éditeur plus avancé, ou une implémentation plus spécifiquement dédiée à certains types d'applications, notamment scientifiques (voir la [Figure 2.3](#)). En règle générale, ces variantes (de même que les implémentations sur d'autres plates-formes) sont réalisées et gérées par des passionnés, et non par les membres de l'équipe de Python. Si vous avez des questions, vous devrez donc vous intéresser à leurs publications et leurs forums de discussion.

Figure 2.2 :
Choisir une autre
plate-forme.



Figure 2.3 :
Choisir une autre
implémentation de
Python.



Si la page de Web de Python n'a pas été refondue depuis l'écriture de ce livre, le mieux est sans doute de choisir le lien qui convient dans la zone d'en-tête (intitulée Download the latest version). Vous devrez ensuite si nécessaire choisir un numéro de version (la dernière en date est la version 3.5.0), ainsi qu'une implémentation 32 bits ou 64 bits.



Le site propose divers types de fichiers à télécharger :

exécutable, compressé ou encore installation à partir du Web. Tant qu'à se lancer, autant choisir la version exécutable de l'installation. Mais, dans tous les cas, vous obtiendrez à l'arrivée l'installation de Python sur votre plate-forme.

Installer Python

Une fois votre copie de Python téléchargée, il est temps de l'installer sur votre système. Le fichier que vous avez obtenu contient tout ce qu'il vous faut pour débiter :

- L'interpréteur Python
- Une documentation (fichiers d'aide)
- L'accès en mode Ligne de commandes
- L'environnement de développement intégré (IDLE)
- Le désinstallateur (uniquement sur certaines plates-formes)

Voyons d'un peu plus près comment les choses se passent sous Windows, Mac OS X et Linux.

Travailler avec Windows

L'installation de Python sous Windows s'effectue comme avec n'importe quelle autre application. En fait, il vous suffit de retrouver le fichier que vous avez téléchargé pour lancer la procédure. Suivez ces étapes :

1. **Localisez la copie de Python que vous avez téléchargée.**

Le nom du fichier peut varier selon le mode choisi. Normalement, il devrait se présenter sous la forme python-3.5.0.exe (32 bits) ou encore python-3.5.0-amd64.exe (64 bits). L'exécutable contient le numéro de la version actuelle de Python, en l'occurrence 3.5.0.

2. **Faites un double clic sur le fichier d'installation.**



Windows devrait afficher une boîte de dialogue vous demandant si vous voulez exécuter ce fichier. Comme la réponse est évidemment oui, cliquez sur le bouton Exécuter. L'installateur de Python va apparaître (voir la [Figure 2.4](#)). En l'occurrence, il s'agit d'une version 32 bits donc plus « universelle », mais la démarche serait la même en 64 bits (le choix entre les deux n'a pas d'importance particulière tant qu'il ne s'agit pas de développer des projets considérables).

Figure 2.4 :
Python vous propose de s'installer pour tous les utilisateurs et d'ajouter son chemin d'accès à Windows.



3. **Cochez la case qui indique Add Python 3.5 to PATH.**
Ceci facilitera les relations entre Windows et Python.



En cochant cette option, vous gagnerez du temps par la suite. Elle permet en effet d'accéder à Python depuis l'Invite

de commandes de Windows. Ne vous souciez pas de tout cela pour l'instant, mais c'est réellement une bonne chose que de le faire. Dans tout ce qui suit, je supposerais donc que vous avez coché cette option.

4. **Cliquez sur Install Now pour accepter l'installation complète de Python dans le dossier proposé par défaut, ou sur Customize installation pour effectuer des choix plus personnels.**

Ici, nous allons retenir la seconde proposition de manière à sélectionner une destination plus conventionnelle.

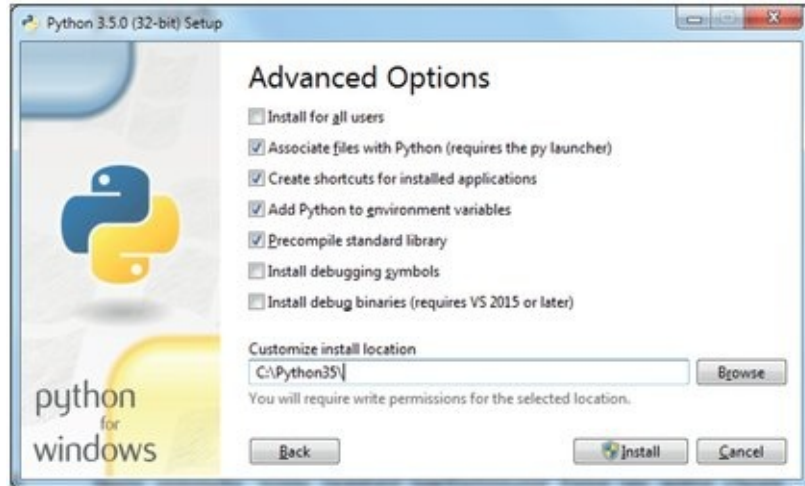
5. **Après avoir cliqué sur Customize installation, il vous est demandé quels composants optionnels vous voulez copier. Laissez toutes les cases cochées et cliquez sur Next (Suivant).**
6. **Dans la fenêtre qui suit, indiquez le dossier de destination.**

Sur la [Figure 2.5](#), vous pouvez constater que j'ai choisi comme nom de dossier C :\Python35. C'est une bonne solution de facilité.



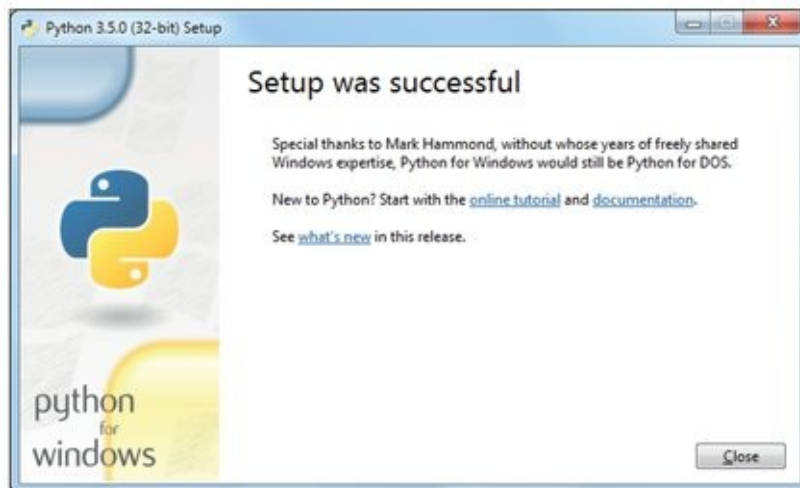
Utiliser comme emplacement C :\Program Files (x86)\Python 3.5 est problématique, et ce pour deux raisons. La première est que ce nom comporte un espace, ce qui compliquerait l'accès à ce dossier depuis l'application. En second lieu, ce dossier nécessite des privilèges d'administrateur, et vous risquez donc d'avoir constamment à vous battre avec le Contrôle de compte utilisateur de Windows.

Figure 2.5 : Vous devez décider de l'emplacement où votre copie de Python sera installée.



7. **Une fois votre dossier de destination défini, cliquez sur le bouton Install.**
Vous n'avez plus qu'à laisser l'installation se dérouler normalement.
8. **Une fois l'installation terminée, cliquez sur le bouton Close (voir la [Figure 2.6](#)).**

Figure 2.6 : L'installation de Python s'est déroulée avec succès.



Travailler avec le Mac

Normalement, Python devrait être installé par défaut sur votre Mac. Cependant, il est possible que votre version date depuis quelque temps. Pour ce qui nous

concerne dans ce livre, ce n'est pas un problème. Vous n'allez pas tester ici les derniers développements scientifiques ou technologiques de Python, mais plus simplement commencer à utiliser celui-ci.



La version Leopard d'OS X (10.5) utilise une version réellement ancienne de Python (la 2.5.1). Celle-ci manque en particulier d'un accès direct à l'application IDLE. De ce fait, certains exercices de ce livre pourraient ne pas fonctionner correctement. Voyez à ce sujet l'article <https://wiki.python.org/moin/MacPython/Leopard>. Les dernières versions d'OS X sont livrées avec la version 2.7 de Python, qui est parfaite pour les exemples de ce livre.

Selon l'usage que vous voulez faire de Python, vous aurez peut-être besoin (ou envie) de mettre à jour votre installation. Une partie de ce processus implique l'installation de GCC (GNU Compiler Collection) de manière à ce que Python puisse accéder aux ressources de bas niveau dont il a besoin. Dans ce cas, suivez ces étapes :

1. **Accédez au site Web de Python (www.python.org).**
2. **Cliquez sur le lien Mac OS X.**
3. **Dans la page qui s'affiche, cliquez sur le lien qui correspond à l'installation que vous voulez réaliser.**

Vous disposez de deux choix principaux pour la version 3.5.0 de python :

- Mac OS X 64-bit/32-bit installer (pour processeur Intel).
- Mac OS X 32-bit i386/PPC installer (pour processeur Power PC).

L'image disque va se télécharger (cela demande un peu de patience). Votre Mac va ensuite ouvrir automatiquement cette image disque.

Celle-ci ressemble à un dossier qui devrait en particulier contenir un fichier dont le nom indique quelque chose comme `python.mpkg`. Il s'agit de celui qui contient l'application Python. Vous devriez également trouver là quelques fichiers de texte classiques concernant le numéro de version, la licence ou encore diverses notes.

4. **Faites un double clic sur le fichier** `python.mpkg`.

Une boîte de dialogue de bienvenue va apparaître.

5. **Cliquez trois fois sur le bouton Continue.**

Après avoir passé diverses informations, vous allez pouvoir choisir la destination.

6. **Sélectionnez le Volume (disque dur ou autre support) que vous voulez utiliser pour installer Python, puis cliquez sur Continue.**

Vous pouvez alors décider de personnaliser l'installation afin de choisir ce que vous souhaitez copier, ou encore redéfinir l'emplacement de destination.

Ce livre suppose que vous effectuez une installation standard, et donc que vous ne modifiez pas la destination.

7. **Cliquez sur Install.**

Si le système vous demande votre nom et votre mot de passe d'administrateur, saisissez ces informations puis cliquez sur OK. L'installation se lance, et vous pouvez suivre son déroulement.

Finalement, une boîte de dialogue vous informe que tout s'est déroulé avec succès.

8. **Cliquez sur Close.**

Python est prêt à être utilisé. Vous pouvez refermer l'image disque et la supprimer de votre système.

Travailler avec Linux

Python est installé avec certaines versions de Linux. C'est par exemple le cas avec les distributions RPM (Red Hat Package Manager) comme SUSE, Red Hat, Yellow Dog, Fedora Core et CentOS. Vous n'avez pas besoin de faire quoi que ce soit.



Selon votre version de Linux, la version de Python varie, et certains systèmes ne contiennent pas l'environnement de développement interactif (IDLE). Ceci pourrait poser problème avec certains des exemples de ce livre. Il est donc vivement conseillé de procéder à une mise à niveau de Python.

Il y a en fait deux techniques possibles pour installer Python sous Linux. La première fonctionne avec une distribution Linux quelconque, tandis que la seconde demande à ce que certains critères soient remplis.

Utiliser l'installation Linux standard

Cette technique fonctionne sur n'importe quel système. Cependant, elle nécessite le passage par l'application Terminal et la saisie d'un certain nombre de commandes. Voyez en particulier l'adresse <http://docs.python.org/3/install/> pour des informations plus poussées à ce sujet. Sinon :

1. **Accédez au site Web de Python (www.python.org).**
2. **Cliquez sur le lien Linux/UNIX.**
3. **Dans la page qui s'affiche, cliquez sur le lien approprié à votre version de Linux :**
 - XZ compressed source tarball (toute version de Linux)
 - Gzipped source tarball (meilleure compression et téléchargement plus rapide).
4. **En réponse au message qui vous demande si vous voulez ouvrir ou enregistrer le fichier, choisissez la**

seconde proposition.

Patientez pendant que le téléchargement s'effectue.

5. Faites un double clic sur le fichier téléchargé.

La fenêtre du gestionnaire d'archives va s'ouvrir. Une fois les fichiers extraits, vous devriez voir dans la fenêtre du gestionnaire un dossier appelé normalement `Python 3.5.0`.

6. Faites un double clic sur le dossier `Python 3.5.0`.

Le contenu de l'archive est extrait dans le sous-dossier `Python 3.5.0` de votre dossier principal.

7. Ouvrez une copie de Terminal.

La fenêtre de Terminal apparaît. Si vous n'avez jamais suivi ce genre de procédure auparavant, sachez que vous devez installer SQLite et bzip2, à défaut de quoi l'installation de Python échouera. Sinon, vous pouvez passer directement à l'Étape 11 pour commencer à utiliser Python tout de suite.

8. Tapez la commande `sudo apt-get install build-essential` et appuyez sur Entrée.

Linux installe le support Build Essential nécessaire pour construire des packages (voyez l'adresse <https://packages.debian.org/squeeze/build-essential> pour plus de détails).

9. Tapez la commande `sudo apt-get install libsqlite3-dev` et appuyez sur Entrée.

Linux installe le support SQLite dont Python a besoin pour la manipulation des bases de données (voyez l'adresse <https://packages.debian.org/squeeze/libsqlite3-dev> pour plus de détails).

10. Tapez la commande `sudo apt-get install libbz2-dev` et appuyez sur Entrée.

Linux installe le support bzip2 nécessaire à Python pour le traitement des archives (voyez l'adresse <https://packages.debian.org/sid/libbz2-dev> pour plus de détails).

11. Tapez `CD Python 3.5.0` dans la fenêtre de Terminal et appuyez sur Entrée.

Vous accédez ainsi au dossier `Python 3.5.0` sur votre système.

12. Tapez `./configure` et appuyez sur Entrée.

Le script commence par vérifier le type de construction du système avant d'effectuer une série de tâches basées sur ces informations. Ce processus peut prendre une minute ou deux, car il y a de multiples éléments à contrôler.

13. **Tapez la commande `make` et appuyez sur Entrée.**

Linux exécute le script `make` afin de créer l'application Python. Cette phase peut prendre environ une minute (tout dépend bien sûr de la rapidité de votre machine).

14. **Tapez la commande `sudo make altinstall` et appuyez sur Entrée.**

Le système peut vous demander votre mot de passe d'administrateur. Saisissez-le et appuyez sur Entrée. Vous n'avez plus qu'à attendre que l'installation se termine.

Installation Linux en mode graphique

Toutes les versions de Linux supportent la procédure d'installation décrite dans la section précédente. Cependant, quelques-unes d'entre elles, basées sur la distribution Debian, comme Ubuntu 12.x et suivants, fournissent également une méthode d'installation graphique. Vous avez besoin pour cela du mot de passe du groupe des administrateurs (`sudo`). Les étapes ci-dessus concernent la mise en œuvre de cette technique sous Ubuntu, mais la démarche est tout à fait semblable avec d'autres installations Linux :

1. **Ouvrez le dossier `Ubuntu Software Center` (il peut être appelé **Synaptics** sur d'autres plates-formes).**

Vous voyez une liste d'applications classiques.

2. **Sélectionnez les outils de développement dans la liste qui propose d'accéder à tout.**

Python devrait figurer dans la liste des outils pour développeurs.

3. **Faites un double clic sur l'entrée `Python 3.5.0..`**

Ubuntu fournit des détails sur l'entrée Python 3.5.0 et vous propose de l'installer pour vous.

4. **Acceptez l'installation.**

Ubuntu lance le processus d'installation. Vous pouvez suivre le déroulement de celle-ci grâce à une barre de progression. Une fois le travail terminé, le bouton d'installation devient un bouton de suppression.

5. **Refermez le dossier** Ubuntu Software Center.

L'icône de Python devrait maintenant apparaître sur votre bureau. Python est prêt à être utilisé.

Accéder à Python sur votre machine

Une fois Python installé sur votre système, vous avez le cas échéant besoin de savoir où le retrouver. Sur certains aspects, il vous simplifie l'existence en vous proposant d'ajouter son chemin d'accès au système au cours de l'installation. Mais voyons cela de plus près.

Utiliser Windows

Une installation Windows crée un nouveau dossier dans le menu Démarrer. Ouvrez celui-ci, puis choisissez Tous les programmes (ou Toutes les applications sous Windows 10). Localisez alors le dossier Python 3.5. En l'ouvrant, vous allez y trouver en particulier les deux points d'entrée dans le monde de Python : le mode Ligne de commandes (appelé Python 3.5) et le mode interactif (IDLE Python 3.5).

**Un mot sur les copies
d'écran**

En travaillant avec ce livre, vous aurez à utiliser Python aussi bien en mode Ligne de commandes que l'environnement IDLE. Le nom de l'interface graphique (IDLE) est le même sur les trois plates-formes, et vous ne verrez aucune différence significative dans sa présentation. Ces variations sont mineures, et vous pouvez simplement les ignorer.

Cela étant posé, les copies d'écran que vous trouverez dans ce livre ont été réalisées sous Windows (et même plus spécifiquement Windows 7, mais cela ne change rien à l'affaire).

De la même manière, le mode Ligne de commandes fonctionne précisément de la même manière sur les trois plates-formes. Si la présentation peut varier légèrement plus que pour l'interface graphique, cela est dû au système lui-même. Pour autant, les commandes que vous aurez à saisir sont strictement les mêmes dans tous les cas. Et il en va de même pour les sorties que vous obtiendrez. Dans les copies d'écran, seul compte donc le contenu.

D'ailleurs, cette indépendance de la plate-forme utilisée (du moins pour ce qui vous concerne) est une des forces du langage Python.

Cliquer sur l'option IDLE provoque l'apparition de la fenêtre illustrée sur la [Figure 2.7](#). IDLE affiche automatiquement certaines informations à son sujet, comme son numéro de version (ici 3.5.0), ou encore le

type de système que vous utilisez pour exécuter Python (en version 32 bits sur la figure, mais une version 64 bits ne changerait rien pour ce qui nous concerne dans ce livre).

L'option Python (la ligne de commandes) ouvre elle aussi une fenêtre qui rappelle nettement plus l'Invite de commandes de Windows (voir la [Figure 2.8](#)). Les présentations des deux modes se ressemblent en fait, et les informations affichées sont les mêmes.

Figure 2.7 :
Utilisez IDLE si vous préférez un environnement graphique.

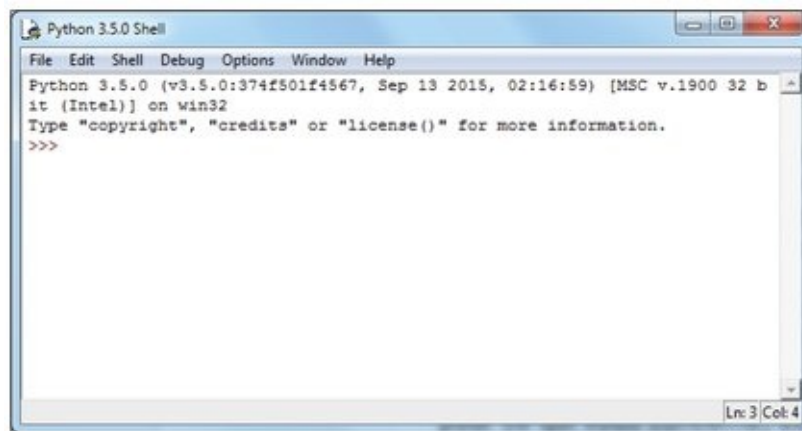


Figure 2.8 :
Utilisez la ligne de commandes si vous préférez la rapidité et la souplesse de ce type d'interface.



Une troisième méthode consiste à ouvrir l'Invite de commandes de Windows. Tapez alors **python** puis appuyez sur Entrée. Vous pouvez utiliser cette approche si vous avez besoin d'une souplesse supplémentaire, pour charger automatiquement des

éléments, ou encore pour exécuter Python dans un environnement doté de privilèges supplémentaires, notamment en ce qui concerne la sécurité (pour cela, cliquez droit sur l'Invite de commandes et choisissez l'option Exécuter en tant qu'administrateur).

Python fournit une impressionnante liste d'options. Pour les voir, tapez **python / ?** au lieu de python dans la fenêtre de l'Invite de commandes. Ne vous souciez pas particulièrement de toutes ces options. Vous n'en aurez pas besoin dans ce livre. Néanmoins, il est utile de savoir qu'elles existent (voir la [Figure 2.9](#)).

Figure 2.9 :
Exécuter Python via l'Invite de commandes Windows offre un grand nombre d'options pour en personnaliser le comportement.



```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Windows\system32>python /?
usage: python [option] ... [-c cmd | -n mod | file | -l [arg] ...]
Options and arguments (and corresponding environment variables):
-b      : issue warnings about str(bytes_instance), str(bytearray_instance)
        and comparing bytes/bytearray with str. (-bb: issue errors)
-B      : don't write .pyc() files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd  : program passed in as string (terminates option list)
-d      : debug output from parser; also PYTHONDEBUG=x
-E      : ignore PYTHON* environment variables (such as PYTHONPATH)
-h      : print this help message and exit (also --help)
-i      : inspect interactively after running script; forces a prompt even
        if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-l      : isolate Python from the user's environment (implies -E and -s)
-n mod  : run library module as a script (terminates option list)
-O      : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO     : remove doc-strings in addition to the -O optimizations
-q      : don't print version and copyright messages on interactive startup
-z      : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-s      : don't imply 'import site' on initialization
-u      : unbuffered binary stdout and stderr, stdin always buffered;
        also PYTHONUNBUFFERED=x
        see man page for details on internal buffering relating to '-u'
-v      : verbose (trace import statements); also PYTHONVERBOSE=x
        can be supplied multiple times to increase verbosity
-U      : print the Python version number and exit (also --version)
-W arg  : warning control; arg is action:message:category:module:lineno
        also PYTHONWARNINGS=arg
-x      : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt  : set implementation-specific option
file    : program read from script file
-       : program read from stdin (default; interactive mode if a tty)
arg ... : arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP : file executed on interactive startup (no default)
PYTHONPATH    : ':'-separated list of directories prefixed to the
        default module search path. The result is sys.path.
PYTHONHOME    : alternate <prefix> directory (or <prefix>:\local\prefix).
        The default module search path uses <prefix>\lib.
PYTHONCASEOK  : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
        to seed the hashes of str, bytes and datetime objects. It can also be
        set to an integer in the range [0,4294967295] to get hash values with a
        predictable seed.

C:\Windows\system32>
```



Cette troisième méthode nécessite l'inclusion de Python dans la chaîne des chemins d'accès Windows. C'est pourquoi je vous ai demandé de valider cette option lors de l'installation de Python. Sinon, vous devrez le faire manuellement par la suite, ce qui est beaucoup moins commode.



Pour définir un chemin permanent vers Python, ouvrez le Panneau de configuration de Windows. Choisissez alors Système et sécurité, puis Système. Dans le volet de gauche, cliquez sur le lien Paramètres système avancés. Dans la boîte de dialogue qui apparaît, cliquez sous l'onglet des paramètres système sur le bouton Variables d'environnement. Dans la liste du haut, sélectionnez PATH, puis cliquez sur le bouton Modifier (si cette variable n'existe pas, cliquez sur Nouvelle). Complétez l'entrée Valeur de la variable comme illustré sur la [Figure 2.10](#), qui suppose ici l'installation de Python dans un dossier appelé Python35. Si nécessaire, placez un point-virgule à la fin de la chaîne existante, et séparez également deux entrées par un point-virgule. Cliquez ensuite plusieurs fois sur OK. Vous pouvez ensuite refermer le Panneau de configuration.

Figure 2.10 :
Ajouter le chemin
vers Python à la
variable
d'environnement
PATH de Windows.



La même technique vaut également pour ajouter des variables d'environnement spécifiques à Python, comme :

- ✓ PYTHONSTARTUP
- ✓ PYTHONPATH
- ✓ PYTHONHOME
- ✓ PYTHONCASEOK
- ✓ PYTHONIOENCODING

- PYTHONFAULTHANDLER
- PYTHONHASHSEED

Aucune de ces variables n'est utilisée dans ce livre. Si le sujet vous intéresse, vous pouvez consulter la page <http://docs.python.org/3.5/using/cmdline.html#environ-variables> pour en apprendre plus à leur sujet.

Utiliser le Mac

Dans le cas du Mac, Python est probablement déjà installé. Par contre, vous devez là encore savoir où il se trouve. La réponse dépend de votre type d'installation.

Localiser l'installation par défaut

Dans la plupart des cas, l'installation par défaut sous OS X ne comprend pas un dossier Python spécifique. Dans ce cas, vous devez ouvrir la fenêtre Terminal à partir du menu des utilitaires dans la liste des applications. Tapez ensuite **Python** et appuyez sur Entrée pour accéder au mode Ligne de commandes de Python. Le résultat ressemble à ce que vous obtenez sous Windows (reportez-vous à la [Figure 2.8](#)). Toujours comme sous Windows, cette méthode offre l'avantage de donner accès aux options de Python pour modifier la manière dont celui-ci fonctionne (reportez-vous à la [Figure 2.9](#)).

Localiser la version actualisée de Python que vous avez installée

Si vous avez effectué vous-même la mise à jour de Python vers une version plus récente, ouvrez le

dossier `Applications`. À l'intérieur de celui-ci, vous devriez trouver un sous-dossier `Python 3.5` contenant divers éléments :

- Un dossier `Extras`
- L'application IDLE (l'interface utilisateur)
- Le « Launcher » Python (l'outil de développement en mode Ligne de commandes)
- Une commande Update Sh...

Un double clic sur l'application IDLE ouvre un environnement graphique interactif très semblable à celui qui est illustré sur la [Figure 2.7](#). Il y a quelques différences cosmétiques, mais le contenu de la fenêtre est le même. Un double clic sur Python active le mode Ligne de commandes, exactement selon le même principe que sur la [Figure 2.8](#). Cet environnement utilise toutes les valeurs par défaut de Python pour fournir un environnement d'exécution standard.



Même si vous avez installé une nouvelle version de Python sur votre Mac, vous n'avez rien à faire pour utiliser l'environnement par défaut. Il est toujours possible d'ouvrir Terminal pour accéder aux options de lancement de Python. Dans ce cas, vous devez vous assurer que vous n'accédez pas à l'installation d'origine. N'oubliez pas d'ajouter `/usr/local/bin/Python3.5` au chemin de recherche du noyau.

Utiliser Linux

Une fois l'installation terminée, vous allez trouver un sous-dossier `Python 3.5` sous votre dossier principal.

L'emplacement physique de Python 3.5 sur votre système Linux est normalement le dossier `/usr/local/bin/Python3.5`. Cette information est importante, car vous pouvez avoir besoin de modifier le chemin d'accès vers ce dossier manuellement. Les développeurs Linux doivent taper **Python3.5**, et non seulement **Python**, dans la fenêtre de Terminal pour accéder à l'installation de Python 3.5.0.

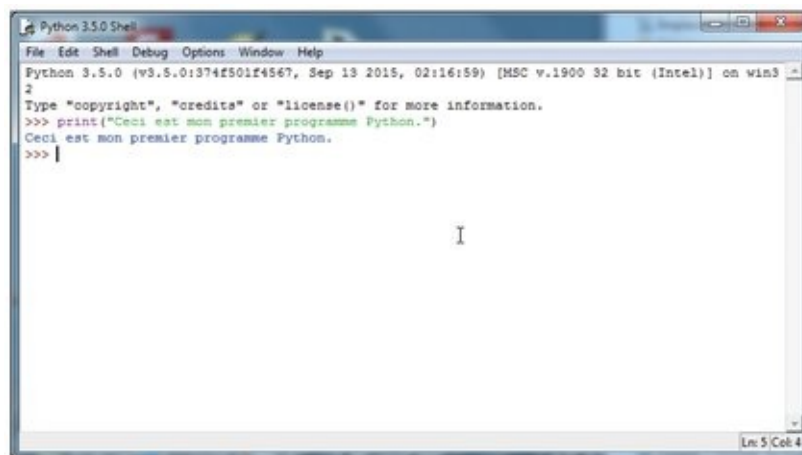
Tester votre installation

Pour vous assurer que votre installation est opérationnelle, vous devez la tester. Il est important de savoir que tout fonctionne normalement, et à tout moment. Bien entendu, ce test implique d'écrire votre première application Python. Pour cela, ouvrez une copie de l'interface IDLE. Comme cela a déjà été mentionné, IDLE affiche automatiquement des informations sur la version de Python et sur le système hôte (reportez-vous à la [Figure 2.7](#)).

Pour voir Python au travail, tapez ceci :

```
print("Ceci est mon premier programme Python.")
```

Figure 2.11 : La commande `print()` affiche toutes les informations qui lui sont passées.



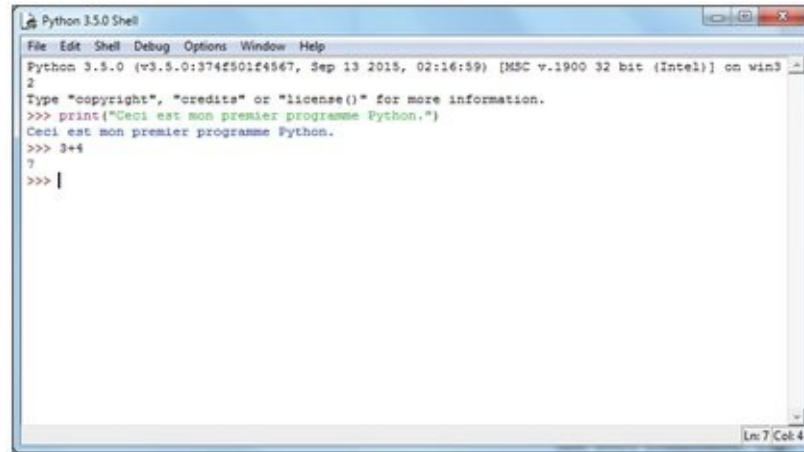
Appuyez ensuite sur Entrée. Python va afficher le message que vous venez de saisir (voir la [Figure 2.11](#)). La commande `print ()` affiche à l'écran tout ce que vous placez entre les guillemets. Vous la retrouverez assez souvent dans ce livre afin d'afficher les résultats des tâches que vous demandez à Python d'accomplir. Il est donc nécessaire de bien se familiariser avec elle.

Remarquez qu'IDLE colore les divers types d'entrées de manière à faciliter la lecture et la compréhension. Ces codes de couleur vous indiquent que (ou si) vous avez fait quelque chose de correct. À ce stade, vous devriez en voir quatre :

- **Pourpre** : Indique que vous avez tapé une commande.
- **Vert** : Spécifie le contenu transmis à une commande.
- **Bleu** : Montre la sortie provoquée par une commande.
- **Noir** : Définit des entrées qui ne sont pas des commandes.

Vous savez maintenant que Python fonctionne, car vous avez pu saisir une commande à laquelle il a réagi. Il pourrait être intéressant de voir une autre commande. Tapez **3+4** et appuyez sur Entrée. Python répond en affichant la bonne réponse, 7 (voir la [Figure 2.12](#)). Vous pourrez noter que la ligne **3+4** est affichée en noir. Pourquoi ? Parce que cette opération n'est en réalité pas une commande, à la différence de `print ()`. Par contre, la réponse apparaît bien en bleu, puisqu'il s'agit d'une sortie.

Figure 2.12 :
L'environnement
interactif de Python
supporte
directement les
opérations
mathématiques.



Il est temps de quitter votre première session IDLE. Tapez **quit()** et appuyez sur Entrée. IDLE devrait alors afficher le message illustré sur la [Figure 2.13](#). Vous n'avez généralement pas envie de tuer quelqu'un (*to kill*), mais comme il ne s'agit ici que d'un tir virtuel, cliquez sur OK pour refermer l'environnement IDLE.

Figure 2.13 : IDLE
semble beaucoup
dramatiser la
fermeture d'une
session !



La commande `quit ()` se termine par des parenthèses, exactement comme dans le cas de `print ()`. Toutes les commandes sont construites sur ce même schéma. Cela vous permet en particulier de savoir qu'il s'agit bien de commandes et pas d'autre chose. Comme vous n'avez rien à dire de particulier à la commande `quit ()`, vous ouvrez puis refermez tout de suite les parenthèses. Tout simplement.

Chapitre 3

Interagir avec Python

Dans ce chapitre :

- ▶ Accéder à la ligne de commandes.
- ▶ Utiliser des commandes pour effectuer des tâches.
- ▶ Obtenir de l'aide sur Python.
- ▶ Terminer la session.

En dernier ressort, toute application que vous créez interagit avec l'ordinateur et les données qu'il contient. Le point central, ce sont les données. Sans données, il n'y a effectivement aucune raison valable pour avoir une application. Et toute application que vous utilisez (même un petit jeu de solitaire) manipule des données d'une certaine manière. En fait, tout cela pourrait se résumer en quelques mots :

- ✓ Créer
- ✓ Lire
- ✓ Actualiser
- ✓ Supprimer

Si vous vous rappelez de l'acronyme CLAS, vous aurez présent à l'esprit le résumé de ce que la plupart des

applications font avec les données contenues dans l'ordinateur. Cependant, avant que votre application puisse accéder à l'ordinateur, vous devez vous-même interagir avec un langage de programmation qui va créer une liste de tâches à effectuer dans un autre langage, celui que l'ordinateur comprend. Tel est l'objet de ce chapitre. Ici, vous allez commencer à interagir avec Python. C'est lui qui va être chargé de prendre la liste des étapes que vous voulez réaliser avec les données de l'ordinateur, et les traduire dans la langue extrêmement binaire de la machine.

Comprendre l'importance du fichier README

De nombreuses applications sont accompagnées d'un fichier appelé README (« Lisez-moi »). Celui-ci fournit généralement des informations mises à jour sur une nouvelle version, et qui n'ont pas pu être intégrées à la documentation ou à l'aide de l'application. Malheureusement, la plupart des gens ignorent ce fichier README, et d'autres ne savent même pas qu'il existe. Conséquence : des tas d'utilisateurs passent à côté de renseignements intéressants sur leur nouveau produit. Python possède évidemment un fichier README.txt que vous trouverez dans son dossier d'installation. En ouvrant ce fichier (en fait, il a aussi un pendant appelé NEWS.txt), vous devriez trouver toutes sortes d'informations intéressantes :

✓ Comment construire une copie de Python

pour les systèmes Linux.

- ✓ Où trouver la description des nouvelles fonctionnalités de cette version de Python.
- ✓ Où trouver la documentation de la dernière version de Python.
- ✓ Comment convertir d'anciennes applications pour les rendre compatibles avec Python 3.5.x.
- ✓ Ce que vous devez faire pour tester des modifications personnalisées de Python.
- ✓ Comment installer de multiples versions de Python sur un même système.
- ✓ Comment accéder aux correctifs de Python.
- ✓ Comment obtenir les mises à jour de Python.
- ✓ Savoir quelles sont les évolutions prévues.

Ouvrir et lire les fichiers README et NEWS vous aidera à devenir un génie de Python. Vous épaterez votre entourage, qui viendra vous poser des tas de questions. Et cela vous permettra peut-être aussi d'obtenir une ascension fulgurante dans votre carrière (quoique je refuse totalement de m'engager sur ce point). Mais, bien entendu, vous pouvez aussi tout simplement passer à autre chose en vous disant que la lecture de ces fichiers, c'est juste trop d'efforts.

Ouvrir la ligne de commandes

Python offre plusieurs méthodes permettant d'interagir avec le langage sous-jacent. Par exemple, vous avez fait connaissance dans le Chapitre 2 avec

l'environnement de développement intégré (IDLE). Celui-ci facilite effectivement la création d'applications. Cependant, dans certaines circonstances, vous voulez uniquement faire des expériences, ou encore exécuter une application existante. Dans ce cas, la version en mode Ligne de commandes de Python est souvent mieux adaptée, car elle offre un contrôle plus poussé sur l'environnement de Python grâce aux options de celui-ci. De plus, elle utilise moins de ressources et présente une interface minimaliste vous permettant de vous concentrer sur le code, plutôt que sur l'interface.

Lancer Python

Selon votre plate-forme, vous pouvez disposer de plusieurs méthodes pour lancer Python en mode Ligne de commandes. Voyons quelques rappels du Chapitre 2 :

- Sélectionnez l'option Python (accompagné ou non de la précision Command line ou quelque chose d'équivalent) que vous trouvez dans le dossier `Python 3.5`. Cette option démarre une session Python avec ses réglages par défaut.
- Ouvrez l'Invite de commandes ou le Terminal, tapez ensuite **Python** et appuyez sur Entrée. Utilisez cette option si vous avez besoin de plus de souplesse dans la configuration de l'environnement Python en utilisant les options (ou les *commutateurs* si vous voulez) de la ligne de commandes.
- Localisez le dossier qui contient Python (par exemple `C :\Python35` sous Windows) et lancez

directement le fichier `python.exe`. Dans ce cas, vous ouvrez également une session en mode Ligne de commandes qui utilise la configuration par défaut, mais avec cette fois des privilèges plus évolués (notamment en termes de sécurité) ou la possibilité de personnaliser les propriétés du fichier exécutable (afin d'ajouter des commutateurs à la ligne de commandes).

Figure 3.1 : Le mode Ligne de commandes de Python vous dit (presque) tout sur celui-ci.



Dans tous les cas, vous devriez voir apparaître une fenêtre semblable à celle de la [Figure 3.1](#). Bien entendu, l'apparence de celle-ci varie légèrement d'un système à un autre, mais le contenu devrait être le même. Sinon, cela signifie que vous utilisez IDLE au lieu de la ligne de commandes, ou bien que votre système a une configuration différente de la mienne, ou encore que vous avez une autre version de Python. Le message que vous devriez voir apparaître affiche le numéro de version de Python ainsi que des informations sur le système d'exploitation hôte.

Maîtriser la ligne de commandes

Si cette section peut paraître un peu compliquée à première vue, et même si vous n'aurez pas

normalement à en passer par là en lisant ce livre, elle est tout de même utile, voire importante. Pour l'instant, il vous suffit de parcourir ces informations afin de savoir qu'elles sont disponibles et que vous pourrez y revenir en cas de besoin.

Pour lancer Python en mode Ligne de commandes, tapez **Python** et appuyez sur Entrée. Mais ce n'est pas tout ce que vous pouvez faire. Il est également possible de fournir des informations supplémentaires afin de définir la manière dont Python fonctionne :

➤ **Options** : Une option, ou encore un commutateur de la ligne de commandes, débute par un signe moins (« - »), suivi d'une ou plusieurs lettres. Par exemple, si vous voulez obtenir de l'aide sur Python, vous tapez **Python - h** et vous appuyez sur Entrée. Vous pouvez également voir d'autres options sur l'art et la manière de démarrer Python. Nous y reviendrons un peu plus loin.

➤ **Nom de fichier** : Fournir un nom de **fichier** demande à Python de charger celui-ci et de l'exécuter. Vous pouvez par exemple lancer l'un des exemples de ce livre en spécifiant le nom du fichier correspondant sur la ligne de commandes. Supposons par exemple que vous ayez à disposition un fichier de script nommé `Bonjour.py`. Pour l'exécuter, il vous suffirait alors de taper **Python Bonjour.py** et d'appuyer sur Entrée.

➤ **Arguments** : Une application peut accepter des informations supplémentaires afin de préciser la manière dont elle doit fonctionner. Ce type d'information est appelé un *argument*. Ne vous faites pas de soucis à ce sujet pour l'instant. Nous y reviendrons plus loin dans ce

livre.



Pour l'instant, la plupart de ces options n'ont aucun sens pour vous. Elles sont là pour que vous puissiez les retrouver plus tard lorsque vous en aurez besoin. Mais les découvrir vous aidera à mieux comprendre ce dont vous disposez. Évidemment, vous pouvez aussi laisser tout cela de côté jusqu'à ce que vous y trouviez une utilité particulière.



Python utilise des options qui sont sensibles à la casse, autrement dit aux majuscules et aux minuscules. Ainsi, les options `-s` et `-S` sont complètement différentes.

Voyons d'un peu plus près ce que sont ces options Python :

✓ `-b` : Ajoute des avertissements à la sortie lorsque Python utilise certaines fonctionnalités, en particulier `str(bytes_instance)`, `str(bytearray_instance)`, la comparaison d'octets `(bytes)` ou `bytearray` avec `str()`.

✓ `-bb` : Ajoute des erreurs à la sortie lorsque Python utilise certaines fonctionnalités, en particulier `str(bytes_instance)`, `str(bytearray_instance)`, la comparaison d'octets `(bytes)` ou `bytearray` avec `str()`.

✓ `-B` : Empêche d'écrire une extension de **fichier** `.py` ou `.pyco` lors de l'importation d'un module.

✓ `-c cmd` : Utilise l'information fournie par `cmd` pour lancer un programme. Cette option indique également à Python de cesser de traiter le reste de l'information de la ligne de

commandes en tant qu'options.

✓ `-d` : Lance le débogueur (il sert à localiser les erreurs dans votre application).

✓ `-E` : Ignore toutes les variables d'environnement de Python, comme `PYTHONPATH`, qui sont utilisées pour configurer son environnement.

✓ `-h` : Affiche une aide sur les options et les variables d'environnement de base. Python se termine après avoir exécuté cette requête sans rien faire d'autre.

✓ `-i` : Force Python à vous permettre d'inspecter le code de manière interactive après avoir lancé un script.

✓ `-m mod` : Lance le module de bibliothèque spécifié par *mod*. Cette option indique également à Python de cesser de traiter le reste de l'information de la ligne de commandes en tant qu'options.

✓ `-o` : Optimise légèrement le code binaire généré par Python.

✓ `-OO` : Optimise davantage le code binaire généré par Python.

✓ `-q` : Demande à Python de ne pas afficher le message de version et de copyright lors de son lancement.

✓ `-s` : Force Python à ne pas ajouter le chemin du site de l'utilisateur à `sys.path` (une variable qui indique à Python où trouver des modules).

✓ `-S` : Demande à Python de ne pas rechercher lors de son initialisation des chemins d'accès qui pourraient contenir des modules dont il a besoin.

✓ `-v` : Place Python en mode dit « verbose » de manière à pouvoir visualiser toutes les instructions importées.

✓ `-V` : Affiche le numéro de version de Python et

quitte.

✓ `--version` : Agit comme `-V`.

✓ `-W args` : Modifie le niveau des avertissements émis par Python. Les valeurs valides pour *args* sont :

- `action`
- `message`
- `category`
- `module`
- `lineno`

✓ `-x` : Saute la première ligne d'un fichier contenant du code source, ce qui permet d'utiliser des formes non Unix de `# ! cmd`.

✓ `-X opt` : Définit une option spécifique d'implémentation (voyez si nécessaire avec votre documentation Python).

Tirer profit des variables d'environnement Python

Les *variables d'environnement* sont des réglages spéciaux qui sont placés sur la ligne de commandes, ou qui sont définis dans votre système d'exploitation. Elles servent à configurer Python d'une manière particulière. Elles effectuent des tâches qui correspondent à des options fournies lors du lancement de Python, mais elles peuvent être rendues permanentes. Ceci permet donc d'obtenir une configuration de Python qui sera identique à chaque démarrage, sans avoir besoin de spécifier ces options à chaque fois sur la ligne de commandes.



Comme pour les options, la plupart de ces variables

d'environnement ne devraient pour l'instant pas avoir un grand sens pour vous. Mais vous pouvez lire ce qui suit pour savoir ce dont vous pouvez disposer. Certaines de ces variables seront utilisées ultérieurement dans ce livre. Si vous le souhaitez, vous pouvez pour l'instant laisser cette section de côté. Vous y reviendrez plus tard en fonction de vos besoins.

La plupart des systèmes d'exploitation permettent de définir des variables d'environnement de manière temporaire, en les configurant au cours d'une certaine session, ou bien de manière permanente, en les enregistrant dans les options de configuration du système lui-même. La manière de procéder dépend de votre système d'exploitation. Dans le cas de Windows, par exemple, nous avons vu dans le Chapitre 2 comme définir une variable d'environnement permanente depuis le Panneau de configuration.



Utiliser des variables d'environnement est utile si vous avez besoin de configurer Python d'une certaine manière, et ce, de façon régulière.

Voici un aperçu de ces variables d'environnement :

- ✓ `PYTHONCASEOK=x` : Force Python à ignorer la casse lorsqu'il analyse des instructions importées. Cette variable ne s'utilise que sous Windows.
- ✓ `PYTHONDEBUG=i` : Joue le même rôle que l'option `-d`.
- ✓ `PYTHONDONTWRITEBYTECODE=x` : Joue le même rôle que l'option `-B`.
- ✓ `PYTHONFAULTHANDLER= x` : Force Python à lister tous les appels qui ont conduit à des erreurs fatales.
- ✓ `PYTHONHASHSEED=arg` : Définit une valeur source (ou semence) servant à générer des valeurs de

hachage pour différents types de données. Lorsque cette variable est définie avec la valeur `random`, Python utilise un nombre aléatoire pour générer les valeurs de hachage des objets `str` (chaîne), `bytes` (octets) et `datetime` (temps). L'intervalle entier valide va de 0 à 4294967295. Utilisez une valeur de semence spécifique afin d'obtenir des valeurs de hachage prévisibles lors de phases de test.

✓ `PYTHONHOME=arg` : Définit le chemin d'accès par défaut que Python utilise pour rechercher des modules.

✓ `PYTHONINSPECT=x` : Joue le même rôle que l'option `-i`.

✓ `PYTHONIOENCODING=arg` : Spécifie la valeur `encoding[:errors]` (comme `utf-8`) utilisée pour les dispositifs `stdin`, `stdout` et `stderr`.

✓ `PYTHONNOUSERSITE` : Joue le même rôle que l'option `-S`.

✓ `PYTHONOPTIMIZE=x` : Joue le même rôle que l'option `-O`.

✓ `PYTHONPATH=arg` : Fournit une liste de chemins d'accès séparés par un point-virgule pour la recherche des modules. Cette valeur est enregistrée dans la variable `sys.path` de Python.

✓ `PYTHONSTARTUP=arg` : Définit le nom d'un fichier à exécuter au lancement de Python. Il n'y a pas de valeur par défaut pour cette variable d'environnement.

✓ `PYTHONUNBUFFERED=x` : Joue le même rôle que l'option `-u`.

✓ `PYTHONVERBOSE=x` : Joue le même rôle que l'option `-v`.

✓ `PYTHONWARNINGS=arg` : Joue le même rôle que l'option `-W`.

Taper une commande

Une fois que vous avez lancé Python en mode Ligne de commandes, vous pouvez commencer à saisir quelque chose. Utiliser des commandes permet d'effectuer certaines tâches, de tester vos idées, ou encore d'améliorer vos connaissances sur Python. La ligne de commandes vous permet donc d'améliorer votre expérience en vous confrontant directement à Python (alors que des détails pourraient être masqués dans le cadre d'un environnement de développement interactif tel qu'IDLE). Les sections qui suivent vous permettent d'approfondir cela.

Dire à l'ordinateur ce qu'il doit faire

Python, comme tout autre langage de programmation, est basé sur des commandes. Une *commande* est simplement une étape dans une procédure. Dans le Chapitre 1, par exemple, j'ai décrit les étapes d'une procédure permettant d'obtenir du pain grillé et beurré. Lorsque vous travaillez avec Python, une commande telle que `print ()` répond exactement au même principe : une étape dans une procédure.

Pour dire à l'ordinateur ce qu'il doit faire, vous exécutez une ou plusieurs commandes dans un langage que Python comprend. Celui-ci traduit ensuite ces commandes dans des instructions que l'ordinateur comprend, et vous voyez ensuite le résultat de cette chaîne de traduction. Une commande telle que `print ()` peut afficher de tels résultats à l'écran. Cependant, Python supporte toutes sortes de commandes, dont la plupart réalisent des tâches importantes sans afficher pour autant quoi que ce soit.

Au fur et à mesure de votre progression dans ce livre, vous utilisez des commandes pour effectuer toutes sortes d'actions. Chacune d'entre elles vous permettra d'atteindre un certain but, exactement comme les étapes d'une procédure. Si vous avez par moment la sensation que toutes ces commandes deviennent bien trop complexes, souvenez-vous qu'il faut les prendre comme des étapes dans une procédure. Même les procédures purement humaines sont parfois d'une extrême complexité. Mais, si vous les prenez étape par étape, vous comprenez mieux de quoi il s'agit et comment tout cela fonctionne. Il en va exactement de même pour les commandes de Python. Ne vous laissez pas déborder par elles, mais concentrez-vous plutôt sur la succession des étapes qui conduisent à l'objectif fixé.

Dire à l'ordinateur que vous avez terminé

À un certain moment, la procédure que vous avez créée se termine (sinon, il y a un sérieux problème...). Dans notre exemple du Chapitre 1, la procédure est finie une fois le pain grillé et beurré. Il y a un point de départ et un point d'arrivée. Lorsque vous tapez des commandes, une étape particulière se termine quand vous appuyez sur la touche Entrée. Dans ce livre, vous allez constater que Python fournit diverses manières de signifier qu'une étape, un groupe d'étapes, ou même une application tout entière se termine. Quelles que soient les tâches réalisées, les programmes d'ordinateur ont toujours un début et une fin distincts.

Voir les résultats

Vous savez maintenant qu'une commande est une étape dans une procédure, et que chaque commande a un point de départ et un point d'arrivée. Et il en va de même pour des groupes de commandes ainsi que pour les applications.

Pour passer de la théorie à la pratique, suivez ces étapes :

1. **Lancez une copie de Python en mode Ligne de commandes.**

Les trois chevrons indiquent que Python attend vos ordres.

2. **Tapez la commande suivante :** `print (« Ceci est une ligne de texte. »)`.

Remarquez qu'il ne se passe rien pour l'instant. Vous avez bien tapé une commande, mais vous n'avez pas signifié que cette commande était terminée.

3. **Appuyez sur Entrée.**

Maintenant, la commande est terminée, et vous pouvez en voir le résultat, comme sur la [Figure 3.2](#).

Ce petit exercice vous montre comment les choses fonctionnent à l'intérieur de Python. Chaque commande que vous saisissez effectue une certaine tâche, mais uniquement après que vous ayez indiqué à Python que la commande est terminée. Dans cet exemple, vous avez utilisé la commande `print ()` en fournissant un certain texte à afficher. Notez que le résultat apparaît immédiatement après l'appui sur la touche Entrée, car il s'agit d'un environnement interactif, où les commandes sont donc exécutées immédiatement. Plus tard, vous commencerez à créer des applications et vous constaterez alors que les résultats n'apparaissent pas forcément tout de suite. Une application, en effet, n'est pas un environnement interactif... Mais chaque commande n'en est pas

moins exécutée immédiatement, une fois que l'application (et non plus vous) a indiqué à Python qu'elle était terminée.

Figure 3.2 :

Terminer une commande indique à Python ce qu'il doit demander à l'ordinateur de faire.



```
C:\Python35\python.exe
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Ceci est une ligne de texte.")
Ceci est une ligne de texte.
>>>
```

Obtenir de l'aide de Python

Python est un langage informatique, pas un langage humain. Vous ne pouvez donc pas discuter facilement avec lui (du moins pas encore). Vous devez donc découvrir les commandes de Python pas à pas, exactement comme si vous appreniez une langue étrangère. Si votre langue maternelle est le français et que vous essayez de dire quelque chose en allemand, vous avez besoin d'une sorte de guide pour vous y aider. Sinon, vous ne sortirez qu'un charabia incompréhensible, et les gens vous regarderont bizarrement. Et même si ce que vous dites a un sens, ce n'est peut-être pas celui que vous vouliez donner à votre phrase.

C'est exactement le même principe avec Python. Vous avez besoin d'un guide pour vous aider à parler correctement dans sa langue. Heureusement, Python est assez accommodant de ce point de vue, et il est toujours prêt à vous aider.



Vous l'avez compris depuis longtemps. Le dialogue avec Python passe par une troisième langue : l'anglais. Désolé, mais il n'est pas possible d'y échapper !

L'aide fournie par Python se situe à deux niveaux :

- **Le mode Aide**, dans lequel vous pouvez parcourir la liste des commandes disponibles.
- **L'aide directe**, qui vous permet de demander des informations sur une commande spécifique.

Il n'y a pas une méthode meilleure que l'autre. Tout dépend de ce dont vous avez besoin à un moment donné. Les sections suivantes détaillent tout cela.

Utiliser l'aide directe

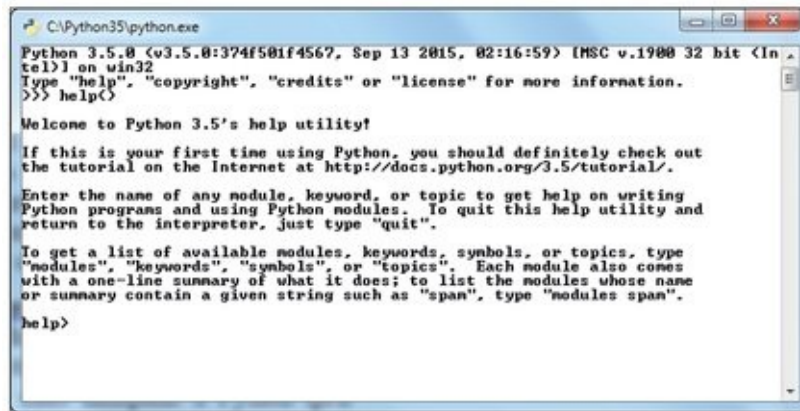
Lorsque vous lancez Python, vous obtenez un affichage semblable à celui de la [Figure 3.1](#). Remarquez en particulier que la fenêtre met en exergue quatre commandes (qui constituent en fait le point de départ des informations d'aide) :

- help
- copyright
- credits
- license

Ces quatre commandes vous apportent des informations d'une certaine sorte concernant Python. Les trois dernières sont classiques et concernent vos

droits, ou encore qui est en charge du développement de Python. Mais celle qui nous intéresse, c'est la première, `help`, et donc celle dont vous avez besoin pour demander de l'aide.

Figure 3.3 : Dans ce mode, vous demandez de l'aide sur d'autres commandes à Python.



```
C:\Python35\python.exe
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help<

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "span", type "modules span".

help>
```

Pour rentrer dans ce mode, tapez **help()** et appuyez sur Entrée. Notez bien que vous devez taper les parenthèses à la fin de la commande, même si cela ne vous semble pas servir à grand-chose. En réalité, ces parenthèses sont indispensables, car elles indiquent à Python qu'il s'agit bien d'une commande. Une fois que vous avez appuyé sur Entrée, Python bascule en mode Aide, et vous voyez s'afficher une page d'informations semblable à celle qui est illustrée sur la [Figure 3.3](#).



Ce mode se signale par la présence de l'indicatif `help>` au début de la ligne de commandes.

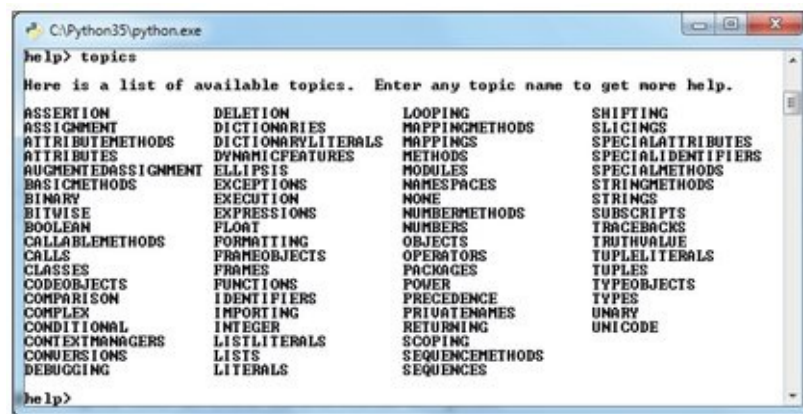
Demander de l'aide

Pour obtenir de l'aide, vous devez savoir quelle question vous devez poser. L'affichage illustré sur la [Figure 3.3](#) vous fournit quelques pistes. Pour explorer Python, les trois sujets de base sont :

- ✓ modules
- ✓ keywords (mots-clés)
- ✓ topics (sujets)

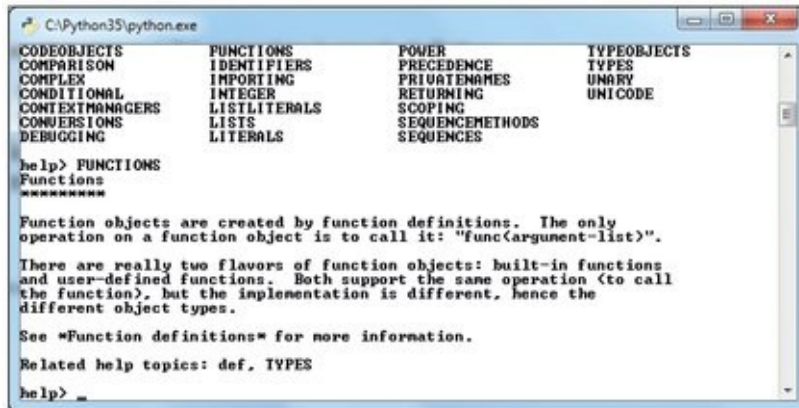
Pour l'instant, les deux premiers ne devraient pas vous parler. En fait, vous n'aurez pas à vous soucier des modules avant le Chapitre 10. L'entrée `keywords` commencera à être utile à partir du Chapitre 4. Par contre, l'option `topics` est intéressante dès maintenant, car elle vous aide à comprendre où commencer votre voyage avec Python. Pour en savoir plus, tapez **topics** et appuyez sur Entrée. Vous voyez apparaître une liste de sujets, comme sur la [Figure 3.4](#).

Figure 3.4 : Le mot-clé `topics` vous fait entrer dans le monde de Python.



Si vous voyez un sujet que vous souhaitez approfondir, par exemple `FUNCTIONS`, tapez simplement le mot correspondant et appuyez sur Entrée. Faites-le, en utilisant bien des majuscules. Vous allez voir des informations concernant la nature des fonctions de Python, comme sur la [Figure 3.5](#).

Figure 3.5 : Vous devez utiliser des majuscules pour demander des informations sur un sujet.



```
C:\Python35\python.exe

CODEOBJECTS      FUNCTIONS      POWER      TYPEOBJECTS
COMPARISON      IDENTIFIERS    PRECEDENCE  TYPES
COMPLEX         IMPORTING      PRIORITIES  UNARY
CONDITIONAL     INTEGER        RETURNING   UNICODE
CONTEXTMANAGERS LISTLITERALS    SCOPING
CONVERSIONS     LISTS          SEQUENCEMETHODS
DEBUGGING       LITERALS      SEQUENCES

help> FUNCTIONS
Functions
*****

Function objects are created by function definitions. The only
operation on a function object is to call it: "func(argument-list)".

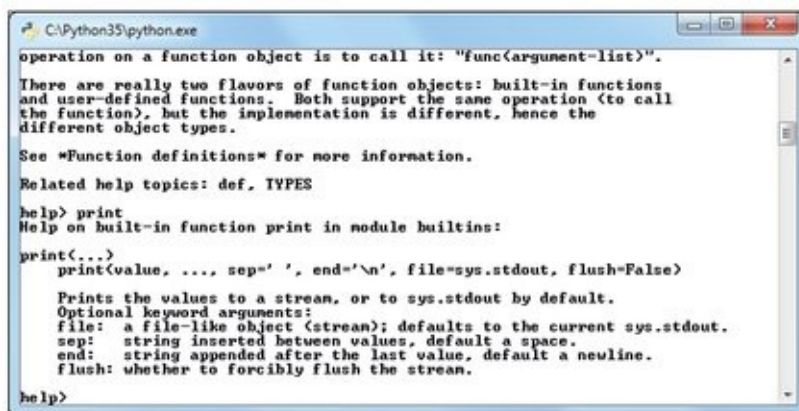
There are really two flavors of function objects: built-in functions
and user-defined functions. Both support the same operation (to call
the function), but the implementation is different, hence the
different object types.

See *Function definitions* for more information.

Related help topics: def, TYPES

help> _
```

Figure 3.6 : Pour obtenir de l'aide sur une commande, tapez son nom sans parenthèses et en respectant sa casse.



```
C:\Python35\python.exe

operation on a function object is to call it: "func(argument-list)".

There are really two flavors of function objects: built-in functions
and user-defined functions. Both support the same operation (to call
the function), but the implementation is different, hence the
different object types.

See *Function definitions* for more information.

Related help topics: def, TYPES

help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

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

help>
```

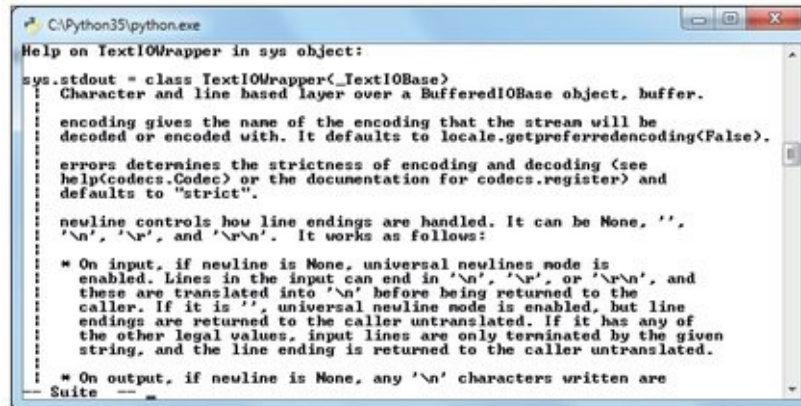
En travaillant avec ce livre, vous allez utiliser des commandes sur lesquelles vous voudrez en savoir plus. Ainsi, nous avons déjà eu à faire à la commande `print ()` plusieurs fois. Pour afficher des informations sur cette commande, tapez **print** et appuyez sur Entrée. Vous ne devez pas ajouter ici de parenthèses, puisque vous demandez simplement de l'aide, et donc vous n'exécutez pas la commande. La [Figure 3.6](#) illustre ce que renvoie alors Python.



À ce stade, il est vraisemblable que lire ces informations ne vous aidera pas beaucoup, car vous avez besoin d'en apprendre plus sur Python (et peut-être de faire quelques révisions pour parfaire votre anglais). Mais supposons par exemple que vous vouliez en apprendre plus sur la signification du

`sys.stdout` qui apparaît dans la page d'aide de la commande `print ()` (en l'occurrence, elle concerne la sortie standard du système). Dans ce cas, tapez **`sys.stdout`** et appuyez sur Entrée. Vous obtenez alors les informations illustrées sur la [Figure 3.7](#).

Figure 3.7 : Vous pouvez demander de l'aide sur l'aide affichée.



```
C:\Python35\python.exe
Help on TextIOWrapper in sys object:

sys.stdout = class TextIOWrapper(<TextIOBase>
    Character and line based layer over a BufferedIOBase object, buffer.

    encoding gives the name of the encoding that the stream will be
    decoded or encoded with. It defaults to locale.getpreferredencoding(False).

    errors determines the strictness of encoding and decoding (see
    help(codecs.Codec) or the documentation for codecs.register) and
    defaults to "strict".

    newline controls how line endings are handled. It can be None, '',
    '\n', '\r', and '\r\n'. It works as follows:

    * On input, if newline is None, universal newlines mode is
      enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
      these are translated into '\n' before being returned to the
      caller. If it is '', universal newline mode is enabled, but line
      endings are returned to the caller untranslated. If it has any of
      the other legal values, input lines are only terminated by the given
      string, and the line ending is returned to the caller untranslated.

    * On output, if newline is None, any '\n' characters written are
      Suite --
```

Vous pouvez parfaitement trouver que ces renseignements ne vous sont pas d'un grand secours, mais au moins vous en savez un peu plus. Remarquez en bas de la [Figure 3.7](#) la ligne qui indique - Suite - . Elle indique que la rubrique n'est pas terminée. Appuyez sur la barre d'espace pour afficher la suite de l'aide. Recommencez pour consulter la rubrique jusqu'au bout. Vous revenez alors à l'indicatif `help >`. Mais rien n'est perdu : il vous suffit de faire glisser vers le haut la barre de défilement vertical de la fenêtre pour remonter vers les pages précédentes.

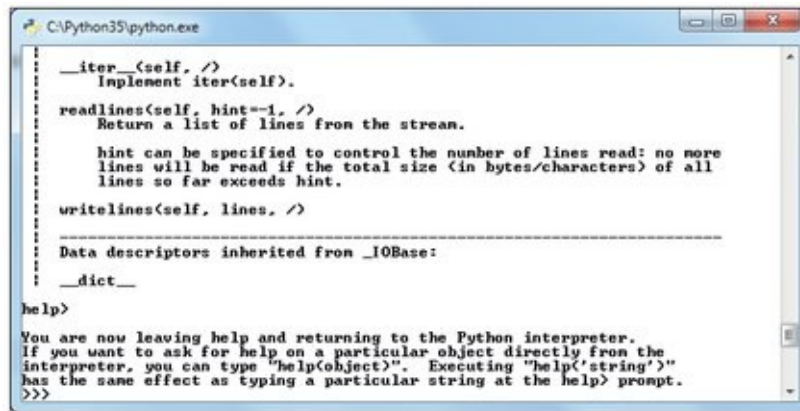
Quitter l'aide

Une fois vos recherches terminées, vous devez sortir de l'aide pour vous mettre à travailler. Tout ce que vous avez à faire pour cela, c'est d'appuyer sur la touche Entrée sans rien saisir. Vous voyez alors apparaître un message qui vous indique que vous

quittez le système d'aide et que vous retourniez à Python lui-même (voir la [Figure 3.8](#)).

Figure 3.8 :

Quittez l'aide en appuyant simplement sur la touche Entrée.



```
C:\Python35\python.exe

__iter__(self, />
    Implement iter(self).
readlines(self, hint=-1, />
    Return a list of lines from the stream.

    hint can be specified to control the number of lines read: no more
    lines will be read if the total size (in bytes/characters) of all
    lines so far exceeds hint.

writelines(self, lines, />

-----
Data descriptors inherited from _IOBase:
._dict_

help>
You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

Obtenir de l'aide directement

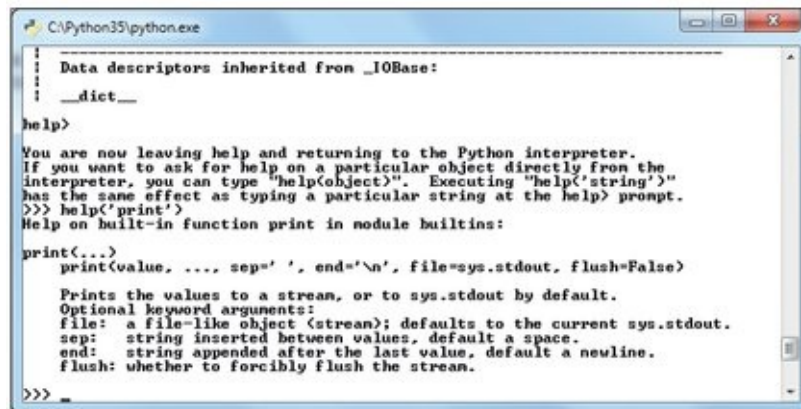
Le mode d'aide présenté ci-dessus n'est pas toujours indispensable, sauf si vous savez quel sujet vous voulez parcourir, ce qui est toujours une bonne idée, ou si vous n'avez pas une idée précise de la question à poser. Si vous savez où vous voulez aller, par contre, il vous suffit de demander de l'aide sur ce qui vous pose question. Pour cela, vous tapez le mot **help**, suivi d'une parenthèse ouvrante, d'une apostrophe et de ce que vous voulez retrouver, après quoi il ne vous reste plus qu'à refermer l'apostrophe et la parenthèse et à appuyer sur Entrée.

À titre d'exemple, essayez ceci : tapez **help('print')** et appuyez sur Entrée. Le résultat est illustré sur la [Figure 3.9](#).

Vous pouvez aussi parcourir l'aide générale de Python. Si vous tapez par exemple **help('topics')** et que vous appuyez sur Entrée, vous allez voir s'afficher une liste de sujets, comme l'illustre la [Figure 3.10](#). Vous noterez

au passage que le résultat est le même que celui de la [Figure 3.4](#).

Figure 3.9 : Vous pouvez obtenir de l'aide sur un point précis sans quitter l'indicatif de Python.



```
C:\Python35\python.exe

Data descriptors inherited from _IOBase:
  __dict__

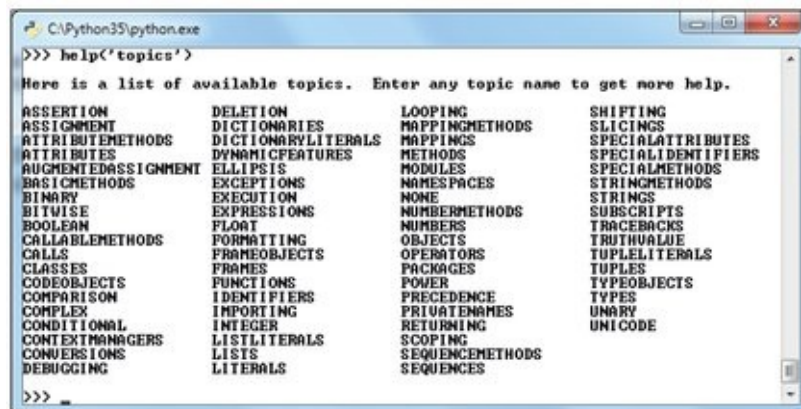
help>
You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)". Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>> help('print')
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

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

>>> _
```

Figure 3.10 : Il est possible de parcourir directement l'aide de Python en utilisant le bon mot-clé.



```
C:\Python35\python.exe

>>> help('topics')
Here is a list of available topics. Enter any topic name to get more help.

ASSERTION      DELETION        LOOPING          SHIFTING
ASSIGNMENT      DICTIONARIES    MAPPINGMETHODS  SLICINGS
ATTRIBUTEMETHODS  DICTONARYLITERALS  MAPPINGS        SPECIALATTRIBUTES
ATTRIBUTES      DYNAMICFEATURES  METHODS          SPECIALIDENTIFIERS
AUGMENTEDASSIGNMENT  ELLIPSIS         MODULES          SPECIALMETHODS
BASICMETHODS     EXCEPTIONS       NAMESPACES       STRINGMETHODS
BINARY          EXECUTION        NONE             STRINGS
BITWISE         EXPRESSIONS      NUMBERMETHODS    SUBSCRIPTS
BOOLEAN         FLOAT            OBJECTS          TRACEBACKS
CALLABLEMETHODS  FORMATTING       OPERATORS        TRUTHVALUE
CALLS           FRAMEOBJECTS     PACKAGES         TUPLELITERALS
CLASSES         FRAMES           POWER            TUPLES
CODEOBJECTS     FUNCTIONS        PRECEDENCE       TYPEOBJECTS
COMPARISON      IDENTIFIERS      PRIVATENAMES     TYPES
COMPLEX         IMPORTING         RETURNING        UNARY
CONDITIONAL     LISTLITERALS     SCOPING          UNICODE
CONTEXTMANAGERS  LISTS            SEQUENCEMETHODS
CONVERSIONS     LITERALS
DEBUGGING
```



Si Python propose deux modes d'aide, c'est pour mieux... vous aider. Cependant, le premier mode est plus pratique à utiliser, puisqu'il se concentre uniquement sur la question de l'aide. De surcroît, il vous fait également gagner un peu de saisie, et il permet aussi d'obtenir davantage d'informations (reportez-vous par exemple à la [Figure 3.3](#)). Il y a donc plein de bonnes raisons pour choisir ce mode si vous avez de nombreuses questions à poser à Python.



Vous devez veiller à respecter la syntaxe de Python, et

en particulier la capitalisation des lettres. Si vous voulez obtenir des informations générales sur les fonctions, vous devez taper **help('FUNCTIONS')**, et non `help(Functions')` ou `help(functions')`. Si vous vous trompez, Python vous répondra qu'il ne comprend pas ce que vous voulez dire, ou encore qu'il ne trouve pas le sujet d'aide correspondant. Mais ne m'en demandez pas les raisons...

Refermer la ligne de commandes

Un jour, vous allez vouloir quitter Python. Certes, c'est difficile à croire, mais il y a des gens qui ont parfois autre chose à faire que de jouer toute la journée avec Python. Vous avez pour cela deux méthodes standard, et des tas de méthodes non standard. En règle générale, il vaut mieux s'en tenir aux méthodes officielles pour s'assurer d'une clôture en bonne et due forme de Python, même si les autres fonctionnent également très bien si vous voulez simplement pratiquer quelque exploration sans vous lancer dans un travail productif.

Les deux méthodes standard sont :

```
✍ quit()
```

```
✍ exit()
```

Les deux referment la version interactive de Python. C'est donc au choix.

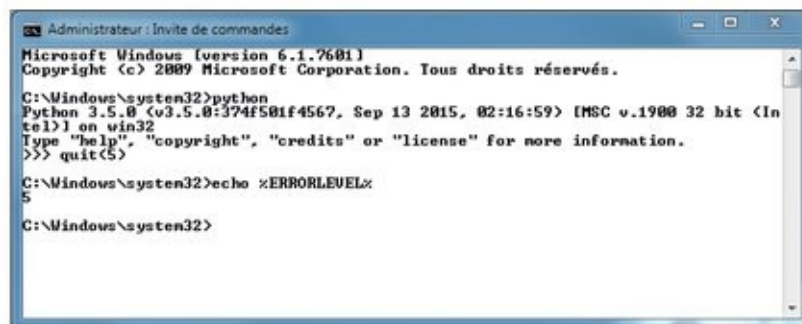
Ces commandes peuvent recevoir un argument optionnel. Vous pouvez par exemple taper **quit(5)** ou **exit(5)** et appuyer sur Entrée pour terminer votre session. L'argument numérique est passé à la variable

d'environnement `ERRORLEVEL`, qu'il est ensuite possible d'intercepter depuis la ligne de commandes ou à l'intérieur d'un fichier de script batch. La pratique courante est de ne pas utiliser cette possibilité lorsque l'application s'est terminée correctement.

Pour voir comment cela fonctionne, suivez ces étapes :

1. **Ouvrez l'Invite de commandes ou un terminal.**
Vous voyez un message s'afficher.
2. **Tapez `Python` et appuyez sur Entrée.**
Python va se réveiller.
3. **Tapez `quit (5)` et appuyez sur Entrée.**
Vous revenez au message d'origine.
4. **Tapez `echo %ERRORLEVEL %` et appuyez sur Entrée.**
Vous voyez alors s'afficher le code d'erreur, comme sur la [Figure 3.11](#). Si vous travaillez avec d'autres plates-formes que Windows, la syntaxe peut varier. Par exemple, avec un script bash, vous devez à la place taper **`echo $`**.

Figure 3.11 :
Ajoutez un code d'erreur si vous avez besoin de signaler à d'autres l'état de Python lorsqu'il se termine.



```
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Windows\system32>python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> quit(5)

C:\Windows\system32>echo %ERRORLEVEL%
5

C:\Windows\system32>
```

Pour ce qui concerne les méthodes non standard, le plus simple consiste à cliquer sur la case de fermeture de la fenêtre dans laquelle s'exécute Python. Dans ce cas, votre application ne pourra pas procéder à un nettoyage du système, ce qui peut se traduire par des problèmes de comportement. Il vaut toujours mieux suivre les règles jusqu'au bout, du moins si vous ne vous contentez pas de naviguer dans Python.



En fait, il existe plusieurs autres commandes permettant de quitter la ligne de commandes interactive de Python, mais il est peu probable que vous en ayez besoin. Je les mentionne à titre d'information, mais vous n'êtes pas obligé de lire ce qui suit.

Lorsque vous utilisez `quit ()` ou `exit ()`, Python effectue un certain nombre de travaux ménagers pour s'assurer que tout est propre et bien rangé avant de clore la session. Si vous suspectez que celle-ci pourrait quand même ne pas se terminer normalement, vous pouvez toujours faire appel à l'une de ces deux commandes pour quitter Python :

- ✓ `sys.exit()`
- ✓ `os._exit()`

Toutes deux ne sont utilisées que dans des situations d'urgence. La première, `sys.exit()`, fournit des fonctions de gestion d'erreur particulières que vous découvrirez dans le Chapitre 9. La seconde, `os._exit()`, referme Python sans effectuer les travaux ménagers mentionnés ci-dessus. Dans les deux cas, vous devez au préalable importer le module voulu, autrement dit `sys` ou `os`, avant de pouvoir utiliser ces commandes. Dans le premier cas, les choses se présentent donc ainsi :

```
import sys
...
sys.exit()
```

Dans le second cas, vous devez en plus fournir un code d'erreur, car cette commande n'est employée que dans des circonstances extrêmes. Autrement dit,

l'appel à cette commande échouera si vous ne lui passez pas un code d'erreur. Par exemple :

```
import os
...
os._exit(5)
```

Le Chapitre 10 discute en détail de l'importation des modules. Pour l'instant, retenez simplement que ces deux commandes ne servent que dans des cas spéciaux et que vous ne devriez pas en avoir besoin dans vos applications.

Chapitre 4

Écrire votre première application

Dans ce chapitre :

- ▶ Travailler avec l'environnement de développement intégré (IDLE).
- ▶ Débuter avec IDLE.
- ▶ Écrire une première application.
- ▶ Vérifier le fonctionnement de l'application.
- ▶ Formater votre code.
- ▶ Comprendre les commentaires.
- ▶ Travailler avec des applications existantes.
- ▶ Terminer une session avec IDLE.

Nombreux sont les gens qui voient le développement d'applications comme une sorte de magie pratiquée par des sorciers appelés des *geeks* qui envoient des ondes bizarres à leurs claviers pour produire toutes sortes de programmes. En fait, la vérité est bien plus prosaïque.

Le développement d'une application suit un certain nombre de processus très stricts, qui n'ont absolument rien à voir avec quelque magie que ce

soit. Ce chapitre a pour but d'effacer toute part d'imaginaire pour rentrer dans le concret, la technologie. Lorsque vous l'aurez terminé, vous devriez être à même de développer une application, certes simple, sans avoir besoin d'invoquer je ne sais quel sort.

Comme pour n'importe quel autre type de tâche, il faut des outils pour écrire des applications. Dans le cas de Python, vous n'avez pas à prendre un véritable outil en main. Mais utiliser un outil, même comme ici en apparence immatériel, rend le travail bien plus facile. Dans ce chapitre, vous allez donc vous servir d'un outil fourni avec Python, l'environnement de développement intégré appelé IDLE. Dans le chapitre précédent, vous vous êtes un peu exercé avec le mode Ligne de commandes de Python. Mais IDLE permet d'aller plus loin et d'écrire plus facilement des applications.



Il existe en fait un grand nombre d'outils dédiés au travail avec Python. Ici, nous allons nous en tenir à ce que vous possédez déjà, autrement dit IDLE. Mais, au fur et à mesure que vous progresserez, vous trouverez que d'autres outils sont plus simples et plus performants. Vous pouvez par exemple regarder du côté de Komodo Edit (<http://komodoide.com/download/#edit>). Vous trouverez toute une liste de ces outils en visitant l'adresse

<https://wiki.python.org/moin/IntegratedDevelopmentEr>

Comprendre l'environnement de développement intégré (IDLE)

Vous pouvez littéralement créer n'importe quelle application Python en utilisant un simple éditeur de texte. Dès lors que vous êtes capable de produire du texte pur, sans aucun formatage, vous pouvez écrire du code Python. Cependant, cette technique n'est en aucun cas efficace ou évidente. Pour faciliter le processus de création d'applications, les développeurs ont écrit ce que l'on appelle des environnements de développement intégrés, ou IDE. Python est accompagné de ce type d'outils, appelé IDLE. Mais bien d'autres environnements de ce type sont disponibles.



D'un environnement de développement à un autre, les fonctionnalités et les besoins couverts peuvent beaucoup varier. En fait, c'est la principale raison pour laquelle il en existe autant. IDLE, de son côté, fournit certaines fonctionnalités de base qui sont communes à pratiquement tout le monde. Vous pouvez vous en servir pour effectuer diverses tâches :

- Écrire du code Python.
- Reconnaître et mettre en évidence les mots-clés ainsi que certains types de textes particuliers.
- Effectuer des tâches d'édition courantes (comme copier, couper et coller) et d'autres plus spécifiques à la programmation (comme montrer les parenthèses qui encadrent une expression).
- Sauvegarder et ouvrir des fichiers Python.
- Naviguer dans le chemin d'accès à Python pour localiser plus facilement des fichiers.
- Localiser des classes Python.
- Effectuer des tâches de débogage simples (supprimer des erreurs dans le code, si vous

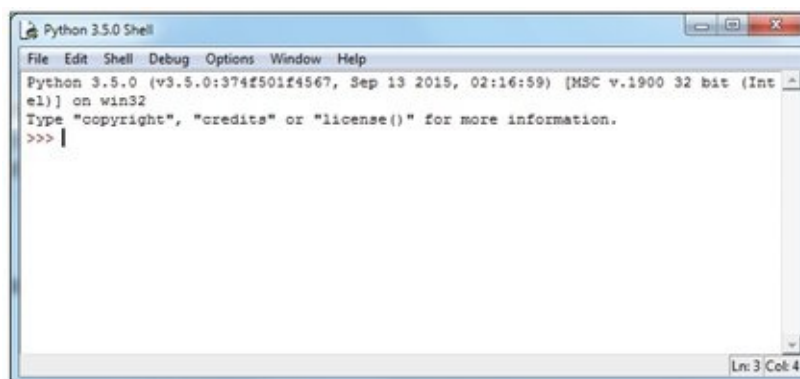
préférez).

IDLE diffère du mode Ligne de commandes en ce qu'il vous permet de disposer d'une interface bien plus complète et d'accomplir de multiples tâches bien plus facilement. D'autre part, la mise au point des applications (le *débogage*) est nettement moins compliquée avec IDLE, même si celui-ci a ses limites.

Lancer IDLE

Vous devriez trouver IDLE dans le dossier des applications de votre système, sous Python 3.5. Lorsque vous cliquez ou double-cliquez (selon votre plate-forme) sur cette entrée, l'éditeur illustré sur la [Figure 4.1](#) apparaît. Vous retrouvez deux lignes affichant des informations sur la version courante de Python, ainsi que sur la plate-forme utilisée, de même des suggestions sur des commandes utiles. Bien entendu, ce que vous voyez sur votre ordinateur peut légèrement différer, selon votre système, sa configuration ou celle d'IDLE. Mais l'essentiel demeure.

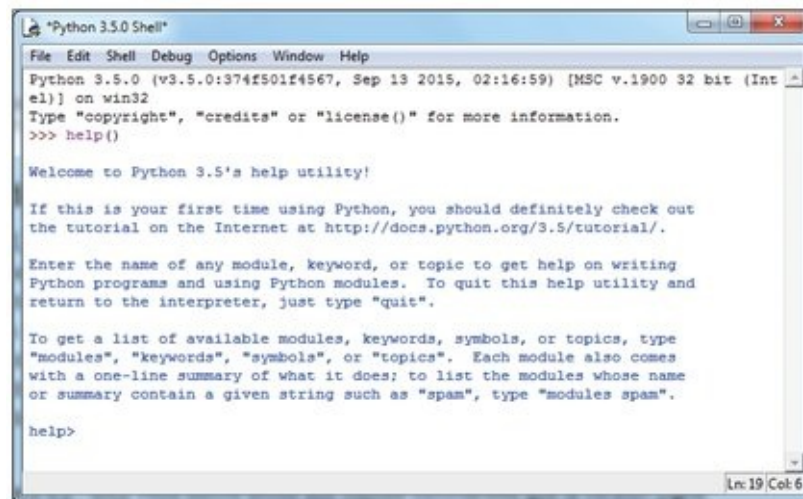
Figure 4.1 : IDLE vous offre une interface de travail interactive pour développer vos applications.



Utiliser des commandes standard

IDLE reconnaît exactement les mêmes commandes que la version de base de Python. Remarquez simplement, en comparant cette fenêtre avec ce que nous avons vu dans le Chapitre 3, qu'elle ne vous propose pas tout de suite d'accéder à l'aide de Python. Après, IDLE est là pour vous aider ! Vous pouvez évidemment taper **help()** et appuyer sur Entrée pour activer le mode Aide. La [Figure 4.2](#) illustre le résultat obtenu.

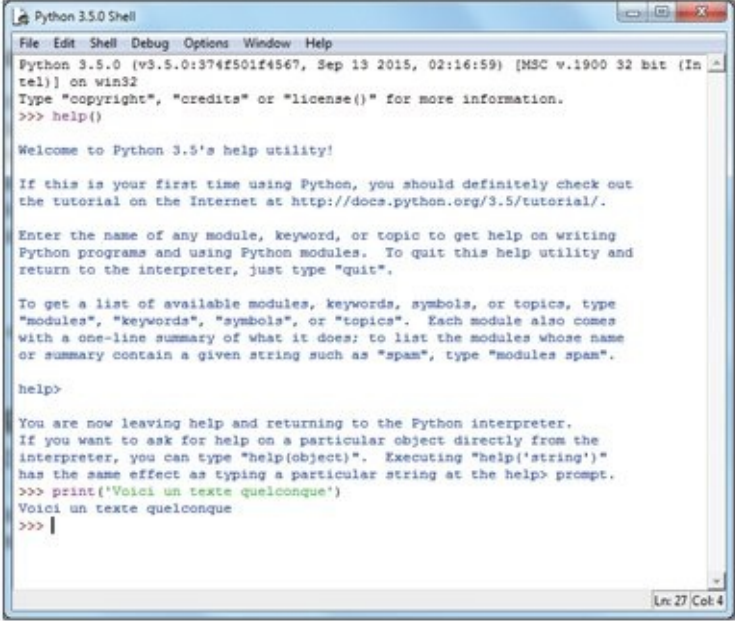
Figure 4.2 : IDLE vous donne accès aux mêmes commandes que la version Ligne de commandes de Python.



Comprendre le codage des couleurs

L'emploi de couleurs dans IDLE vous permet de distinguer plus facilement les commandes et de les différencier d'autres sortes de textes. Si nécessaire, commencez par sortir de l'aide en appuyant simplement sur Entrée. Tapez maintenant **print('Voici un texte quelconque')**. Appuyez ensuite sur Entrée. Vous voyez bien la sortie attendue, comme sur la [Figure 4.3](#).

Figure 4.3 : Grâce au codage des couleurs, vous pouvez facilement déterminer le rôle de chaque type de texte dans une application.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> help()

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>> print('Voici un texte quelconque')
Voici un texte quelconque
>>> |
```

Notez maintenant comme sont utilisées les couleurs. Par exemple, `print ()` est affiché en pourpre pour signaler qu'il s'agit d'une commande. Le texte placé à l'intérieur des parenthèses est mis en vert. Il s'agit de données, et non d'une quelconque commande. La sortie apparaît en bleu. Ces codes rendent les choses plus faciles à lire et à comprendre, ce qui est juste une des nombreuses raisons de travailler avec IDLE plutôt qu'en mode Ligne de commandes.

Obtenir de l'aide d'IDLE

IDLE propose très classiquement un menu d'aide (Help) dans lequel vous pouvez voir quatre entrées :

- **About IDLE** : Donne les dernières informations disponibles sur IDLE.
- **IDLE Help** : Affiche un fichier de texte contenant des informations spécifiques au travail avec IDLE. Vous y trouvez par exemple

une description des commandes disponibles dans les menus.

✓ **Python Docs** : Donne accès à des aides sur Python.

✓ **Turtle Demo** : Propose toute une série d'exemples d'applications Python afin de vous aider à progresser, et même à reprendre des idées déjà bien en place.

Par exemple, l'option About IDLE ouvre une fenêtre qui fournit des adresses utiles, ainsi que des boutons donnant directement accès aux importants fichiers que sont README et NEWS (voir la [Figure 4.4](#)). Cliquez simplement sur le bouton Close pour refermer cette boîte de dialogue.

Figure 4.4 : La boîte de dialogue About IDLE contient certaines informations utiles.



Ce que vous voyez en choisissant dans le menu Help l'option Python Docs dépend de la plate-forme que vous utilisez. La [Figure 4.5](#) montre la version Windows de cette documentation, par ailleurs très complète. Elle propose même une section Tutorial dans laquelle vous découvrirez des tas de conseils, d'astuces et d'exemples qui pourront vous être très utiles une fois

ce livre refermé.

Figure 4.5 : Le système d'aide de Python accessible depuis IDLE.



Configurer IDLE

À la base, IDLE est un éditeur de texte amélioré, et il n'est donc pas surprenant que vous puissiez le configurer pour mieux travailler avec vos textes. Pour cela, ouvrez le menu Options et choisissez l'option Configurer IDLE. Vous voyez s'afficher la boîte de dialogue illustrée sur la [Figure 4.6](#). C'est ici que vous choisissez des choses comme la police de caractères utilisée par IDLE pour afficher les textes. Sur la figure, l'onglet Fonts/Tabs vous permet de définir la taille et le style des caractères, ainsi que le nombre d'espaces utilisés pour l'indentation (voyez plus loin dans ce chapitre la section « Comprendre l'utilisation des indentations » pour plus de détails).



Comme cela a déjà été mentionné, IDLE utilise des codes de couleurs pour faciliter la lecture et la

compréhension des commandes et applications Python. Cette boîte de dialogue vous permet également de personnaliser ces codes depuis l'onglet Highlighting (voir la [Figure 4.7](#)). Remarquez qu'il est possible de sauvegarder différents thèmes en cliquant sur le bouton Save as New Custom Theme. Vous pourriez par exemple créer un thème pour l'utiliser lorsque vous travaillez avec votre ordinateur portable dans de bonnes conditions d'éclairage, et un autre pour travailler sous la lampe.

Figure 4.6 :
Configurez IDLE en fonction de vos besoins particuliers, ou de vos préférences personnelles.



Même si vous n'entendrez pas souvent parler dans ce livre de l'emploi de raccourcis clavier, du fait des différences qui existent d'une plate-forme à une autre, il est utile de savoir qu'IDLE les supporte. IDLE est fourni avec des jeux de raccourcis adaptés aux environnements Mac OS X, Unix et Windows (voir la [Figure 4.8](#)). Pour choisir celui qui vous convient, cliquez sur la liste intitulée sur la figure IDLE Classic Windows. Il est également possible de créer ses propres schémas de raccourcis afin d'adapter IDLE à ses habitudes de travail.

L'onglet General contrôle la manière dont IDLE

fonctionne (voir la [Figure 4.9](#)). Par exemple, vous pouvez demander à IDLE d'ouvrir une fenêtre Python en mode « Shell » (pour pouvoir faire des expériences) ou en mode « Edit » (pour écrire une application). Par défaut, c'est la première option qui est active. Il est aussi possible d'exiger une confirmation pour sauvegarder une application lors de son lancement (ce qui est la prudence même), ou encore de définir la taille par défaut des nouvelles fenêtres que vous êtes amené à créer. En fait, la configuration standard fonctionne très bien, et il n'y a donc aucune bonne raison de la modifier.

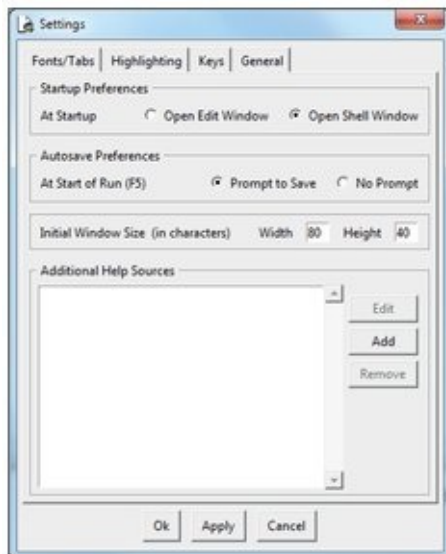
Figure 4.7 :
Changez les
couleurs d'IDLE
pour les adapter à
différentes
situations.



Figure 4.8 :
Utilisez les raccourcis clavier qui ont le plus de sens pour vous en tant que développeur.



Figure 4.9 :
L'onglet General contrôle le fonctionnement d'IDLE.



Notez enfin la section intitulée Additional Help Sources. Elle vous permet d'ajouter de nouvelles sources d'aide qui deviendront donc disponibles dans IDLE. Par exemple, vous pourriez créer un lien vers une ressource en ligne, comme la documentation Web de Python, à l'adresse <https://docs.python.org/3/>. Pour cela, cliquez sur le bouton Add. Saisissez alors le texte que vous voulez afficher dans le menu d'aide, ainsi que l'emplacement de la source sur le disque dur ou en ligne (voir la [Figure 4.10](#)). Cliquez ensuite sur OK et

vérifiez dans le menu Help que tout se passe comme prévu.

Figure 4.10 :
Enregistrer de nouvelles sources d'aide ne peut qu'améliorer votre expérience de développeur.



Une fois que vous avez défini une source d'aide, vous pouvez à tout moment revenir à la boîte de dialogue de configuration d'IDLE pour modifier ou supprimer cette référence.

Créer l'application

Il est maintenant temps de créer votre première application Python. La fenêtre actuelle ne convient pas pour cela (elle est là pour fonctionner en mode interactif). Vous devez donc ouvrir une nouvelle fenêtre d'édition. Vous tapez ensuite vos commandes et vous sauvegarderez le fichier sur votre disque dur.

Ouvrir une nouvelle fenêtre

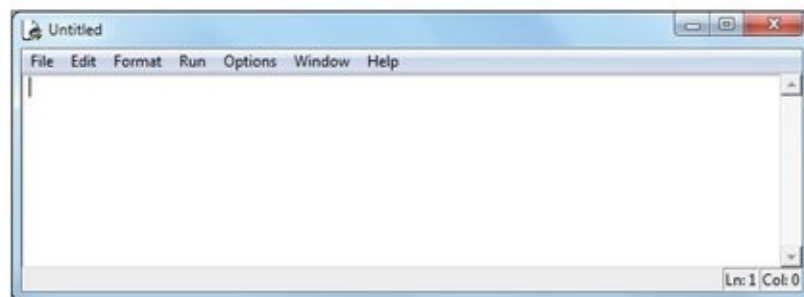
La fenêtre d'IDLE convient parfaitement pour effectuer des expérimentations, pas pour écrire des applications. Vous avez besoin pour cela d'une autre fenêtre, servant, elle, à l'édition. La première vous permet de taper des commandes dans un environnement interactif, et donc qui réagit

immédiatement. La seconde fournit un environnement statique, où vous pouvez entrer des commandes, les sauvegarder puis les exécuter une fois votre application développée. Les deux modes jouent donc des rôles différents.



Pour créer une nouvelle fenêtre, ouvrez le menu File puis choisissez l'option New File. Une nouvelle fenêtre (dite Edit), semblable à celle qui est illustrée sur la [Figure 4.11](#), va s'ouvrir. Notez que la barre de titre indique maintenant simplement Untitled (donc sans titre). Les deux fenêtres présentent presque les mêmes menus, à ceci près que la nouvelle y remplace les entrées Shell et Debug par Format et Run. C'est cette dernière qui va vous servir ensuite pour tester votre application.

Figure 4.11 :
Utilisez la fenêtre
d'édition pour créer
vos applications.



Travailler avec la fenêtre Edit est tout à fait semblable à ce qui se passe dans n'importe quel éditeur de texte. Vous y avez accès à des commandes de base, comme Copier, Couper ou Coller. Appuyer sur Entrée vous renvoie à la ligne au lieu d'exécuter simplement la commande courante. Tout ceci provient du fait que cette fenêtre crée un environnement statique, autrement dit dans lequel vous pouvez saisir une série de commandes et les sauvegarder pour les exécuter plus tard.

La fenêtre Edit propose également des commandes

spéciales pour formater le texte. Les sections « Comprendre l'utilisation des indentations » et « Ajouter des commentaires » de ce chapitre vous en disent plus sur ces fonctionnalités. Tout ce que vous avez besoin de savoir à ce stade, c'est que ces commandes agissent différemment de ce que l'on rencontre dans un éditeur de texte standard, car elles vous aident à contrôler l'apparence de votre code, et non pas celle de votre futur best-seller. Du fait que la plupart de ces options fonctionnent d'une manière automatique, vous n'avez pas vraiment à vous en soucier pour l'instant.

Finalement, la fenêtre Edit donne aussi accès à des commandes qui demandent à Python d'exécuter les étapes de la procédure une par une. Voyez la section « Exécuter l'application », plus loin dans ce chapitre, pour plus de détails à ce sujet.

Tapez les commandes

Comme dans l'environnement interactif de Python, vous pouvez simplement commencer à saisir une commande dans la fenêtre Edit. Pour mieux voir comment tout cela fonctionne, tapez **print()**. Remarquez alors que des indications sur les options (les *arguments*) de la commande s'affichent immédiatement (voir la [Figure 4.12](#)). Certes, à ce stade, tout cela peut vous paraître un peu ésotérique, mais vous en apprendrez plus sur tout cela au fil de ce livre. Vous noterez simplement pour l'instant que la commande `print ()` a besoin d'une valeur (*value*) avant de pouvoir afficher quoi que ce soit. Et vous rencontrerez bien d'autres valeurs différentes dans la suite des événements...

Terminez la commande en tapant « **Ceci est une application Python très simple.** ») et en appuyant sur Entrée. Comparez votre résultat avec l'illustration de la [Figure 4.13](#). Il s'agit évidemment d'une des applications les plus simples que vous puissiez créer avec Python.

Figure 4.12 : La fenêtre d'édition vous fournit des informations utiles sur la commande que vous saisissez.

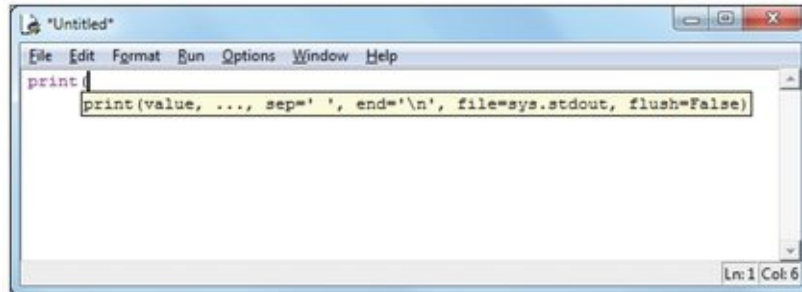
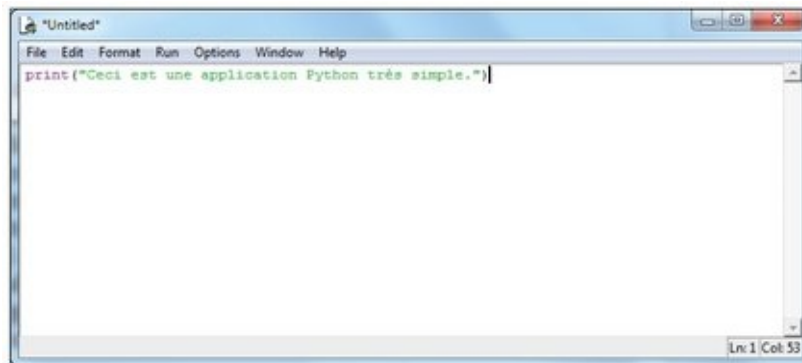


Figure 4.13 : Une application complète peut être très courte.



Sauvegarder le fichier

Vous pourriez exécuter cette application dès maintenant. Cependant, sauvegarder votre travail avant de le tester est toujours une bonne idée. Si vous avez fait une erreur qui pourrait provoquer le plantage de Python, voire même du système, votre code sera quand même mis à l'abri. Il sera donc plus facile d'y revenir par la suite pour déterminer la cause du problème, apporter les corrections nécessaires, et essayer d'exécuter l'application à nouveau.

Ouvrez le menu File et choisissez la commande Save. Une traditionnelle boîte de dialogue pour permettre de choisir le dossier de destination.



Utiliser le dossier de Python pour y enregistrer vos applications est une mauvaise idée. Évitez de mélanger Python lui-même et vos créations personnelles. Je vous conseille donc de créer un dossier spécifique et autant de sous-dossiers qu'il est nécessaire pour y ranger vos fichiers.



Si vous avez téléchargé les exemples de ce livre à l'adresse indiquée dans l'introduction, vous trouverez les exemples de ce chapitre dans un dossier appelé BP4D. Vous pouvez le renommer à votre convenance, et l'enregistrer dans un emplacement facile à retrouver.

Dans le champ Nom du fichier, entrez **FirstApp.py** et cliquez sur Enregistrer. Votre nouvelle application est maintenant stockée sur votre disque dur et vous pouvez y accéder quand vous voulez.

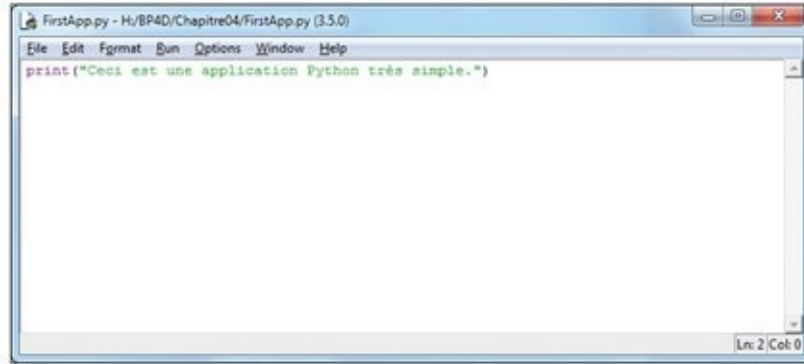


NdT : Pour éviter des risques d'erreur, les noms des fichiers d'application sont ceux du livre original.

Lorsque vous revenez à la fenêtre d'édition, vous constatez que la barre de titre affiche maintenant le chemin d'accès complet à votre application (voir la [Figure 4.14](#)).

Figure 4.14 :

Lorsque l'application est enregistrée sur le disque, la barre de titre affiche son nom et son chemin d'accès.

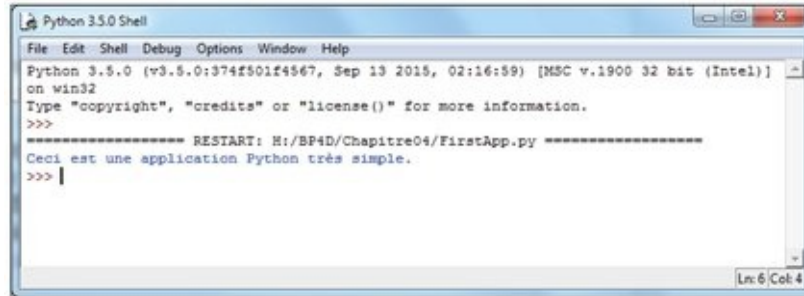


Exécuter l'application

Les applications ne servent pas à grand-chose si vous ne les exécutez pas. Python fournit pour cela diverses méthodes. Dans cette section, nous allons voir la manière la plus simple de s'y prendre. Vous en découvrirez d'autres dans la section « Charger et exécuter des applications existantes », plus loin dans ce chapitre. La chose importante à se rappeler, c'est que Python offre un environnement extrêmement souple. Si une méthode servant à réaliser une certaine tâche ne marche pas, il y en a certainement une autre qui fonctionnera presque certainement.

Pour exécuter votre application, ouvrez le menu Run et choisissez l'option Run Module. Vous allez voir s'ouvrir une nouvelle fenêtre de Python en mode « Shell ». Celle-ci affiche le résultat produit par l'application (voir la [Figure 4.15](#)).

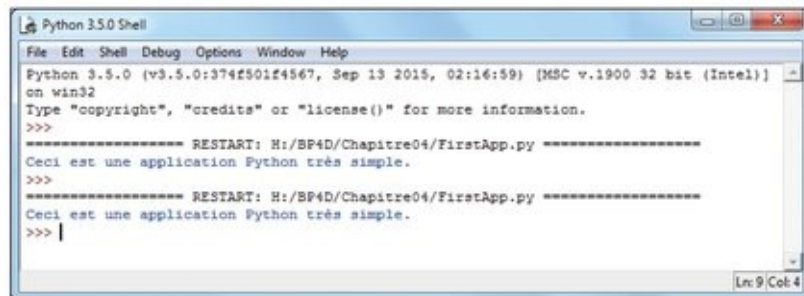
Figure 4.15 :
Python exécute
notre exemple
d'application.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:/BP4D/Chapitre04/FirstApp.py =====
Ceci est une application Python très simple.
>>>
```

Les deux premières lignes vous sont maintenant familières. Il s'agit des informations qui apparaissent lorsque vous lancez Python.

Figure 4.16 : Le
message RESTART
apparaît chaque
fois que vous
relancez
l'application.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:/BP4D/Chapitre04/FirstApp.py =====
Ceci est une application Python très simple.
>>>
===== RESTART: H:/BP4D/Chapitre04/FirstApp.py =====
Ceci est une application Python très simple.
>>>
```

Il vient ensuite une ligne qui commence par RESTART suivi du chemin d'accès complet à l'application. Vous verrez ce message chaque fois que vous allez exécuter l'application. Pour vous en rendre compte, revenez à la fenêtre d'édition et lancez à nouveau la commande Run Module. La fenêtre d'exécution va afficher une seconde fois le même message (voir la [Figure 4.16](#)).

Comprendre l'utilisation des indentations

En suivant les exemples de ce livre, vous allez constater que certaines lignes sont indentées, autrement dit décalées par rapport à la marge gauche. En fait, ces exemples comprennent aussi

d'autres espaces blancs, comme des lignes supplémentaires vides à l'intérieur du code. Python ignore toutes les indentations dans vos applications. Ces indentations ont pour but d'aider à mieux voir les relations qu'entretiennent les différents éléments du code.

Les diverses utilisations de ces indentations vous deviendront plus familières au fur et à mesure que vous progresserez dans ce livre. Cependant, il est important d'en comprendre dès maintenant l'utilité et l'importance. Pour cela, nous allons créer un nouvel exemple qui se servira de l'indentation afin de rendre plus apparente et plus compréhensible la structuration du code.

1. **Créer un nouveau fichier en choisissant la commande New File du menu File d'IDLE.**

Une nouvelle fenêtre sans titre apparaît.

2. **Tapez `print(« Ceci est une ligne de texte réellement longue qui va « +`.**

Ce texte s'affiche tout à fait normalement dans la fenêtre. Mais il y a quelque chose de nouveau ici : la ligne se termine par un signe plus (+). Ce signe indique à Python qu'il y a encore du texte à afficher. Cette sorte d'addition (un bout de texte *plus* un autre bout de texte) est appelée *concaténation*. Nous aurons bien sûr à y revenir, et vous n'avez donc pas trop à vous en soucier pour l'instant.

3. **Appuyez sur Entrée.**

Le point d'insertion ne se déplace pas au début de la ligne suivante. En fait, il se trouve maintenant exactement à la verticale des premiers guillemets (voir la [Figure 4.17](#)). On appelle cela une indentation automatique, et c'est l'une des fonctionnalités qui différencient un éditeur de texte quelconque d'un outil dédié à l'écriture de code, comme IDLE.

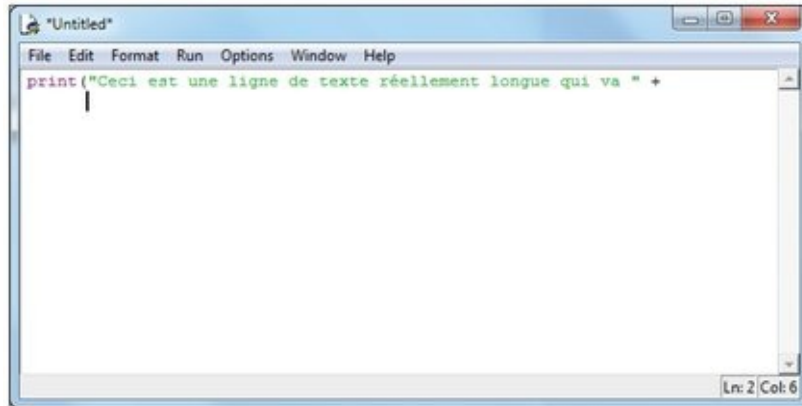
4. **Tapez maintenant apparaître sur plusieurs lignes dans le fichier du code source. »). Vous pouvez**

ensuite appuyer sur Entrée.

Vous devriez constater que le point d'insertion revient en début de ligne. IDLE a compris que vous aviez terminé la saisie d'une commande (voire même de l'application).

5. **Ouvrez le menu Fichier et choisissez Save.**
6. **Donnez à votre application comme nom LongLine.py et validez.**

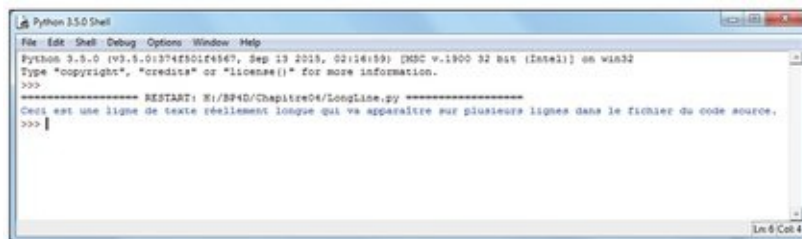
Figure 4.17 : La fenêtre d'édition indente automatiquement certains types de texte.



7. **Dans le menu Run, choisissez Run Module.**

Comme dans l'exemple précédent, une nouvelle fenêtre apparaît. Vous pouvez alors vérifier que les deux parties du texte ont bien été concaténées de manière à ne plus former qu'une seule chaîne de caractères (voir la [Figure 4.18](#)).

Figure 4.18 : Avec la concaténation, plusieurs éléments de texte ne forment plus qu'une seule ligne lors de l'exécution de l'application.



Ajouter des commentaires

Les gens prennent tout le temps des notes pour se rappeler de choses diverses et variées. Par exemple,

vous faites une liste lorsque vous allez faire des courses (non ?). Vous regardez cette liste lorsque vous êtes dans le magasin, et vous rayez les articles au fur et à mesure pour être sûr de ne rien oublier. Et je pourrais multiplier les exemples (notes de réunion, notes de lecture, notes pour des rendez-vous, pour aider votre mémoire dans des tas de circonstances, ou encore avant d'écrire un livre, et ainsi de suite). Commenter le code source d'une application est juste une autre forme de note. Vous ajoutez ces commentaires pour pouvoir vous souvenir plus tard du rôle de chaque tâche (et, croyez-moi, on oublie très vite ce genre de choses). Mais les commentaires sont aussi très importants pour les autres personnes qui pourraient avoir à lire ou reprendre votre code pour qu'elles comprennent ce que vous avez voulu faire. Les sections qui suivent décrivent les commentaires plus en détail.

Comprendre les commentaires

Les ordinateurs ont besoin d'un signal particulier pour déterminer que le texte que vous écrivez est un commentaire, et non du code à exécuter. Python fournit pour cela deux méthodes. La première consiste à placer un commentaire sur une ligne, en le faisant précéder par le caractère dièse (`#`), comme ceci :

```
# Ceci est un commentaire.  
print("Bonjour de Python !") #Ceci est aussi un commentaire.
```



Ce genre de commentaire pour être placé sur sa propre ligne, ou à la suite d'un code exécutable. Il est en règle générale assez court et sert à apporter une précision sur tel ou tel morceau de code, ou sur telle

ou telle commande.

Si vous avez besoin de créer un commentaire plus long, vous devez faire appel à la seconde méthode, qui est multiligne. Ce type de commentaire débute et se termine par trois guillemets (pas des apostrophes), comme ceci :

```
Application: Comments.py  
Auteur: John  
Objet: Montre comment utiliser des commentaires.
```



Tout ce que vous placez entre ces deux jeux de guillemets triples est considéré par Python comme formant un seul et unique commentaire. L'exemple ci-dessus est assez typique de ce que l'on trouve au début d'un fichier de code source : le nom de l'application, son auteur et le genre de tâche qui va être réalisée. Bien entendu, il n'existe aucune règle concernant l'art et la manière d'utiliser ces commentaires. Le but premier, hormis votre propre compréhension et celle des autres, est d'indiquer précisément à l'ordinateur ce qui est un commentaire, et ce qui ne l'est pas. Évidemment, toute erreur, comme l'oubli d'un guillemet, provoquera une erreur d'exécution.



Même si les deux méthodes exposées ci-dessus permettent de placer des commentaires, l'éditeur IDLE permet facilement de les distinguer. Si vous utilisez le schéma de couleur par défaut, les commentaires multilignes apparaissent en vert, les autres étant affichés en rouge. Python n'a que faire de cette coloration : elle est uniquement destinée à vous aider en tant que développeur.

Utiliser des commentaires comme aide-mémoire

Trop de gens ne savent pas quoi faire des commentaires dans une application. Mais songez que vous pouvez écrire un certain code aujourd'hui, et ne plus y revenir pendant plusieurs années. Ce jour-là, vous allez plus que certainement vous demander de quoi il s'agit, à quoi ce code peut bien servir, et comment il fonctionne.

En fait, il y a plusieurs bonnes raisons pour commenter votre code :

- ✓ Vous rappeler vous-même ce que fait le code et pourquoi vous l'avez écrit.
- ✓ Expliquer aux autres comment assurer la maintenance de votre code.
- ✓ Rendre votre code accessible à d'autres développeurs.
- ✓ Fournir des idées pour des mises à jour ultérieures.
- ✓ Donner une liste des sources de documentation que vous avez utilisées pour écrire le code.
- ✓ Gérer une liste des évolutions que vous avez apportées au code.

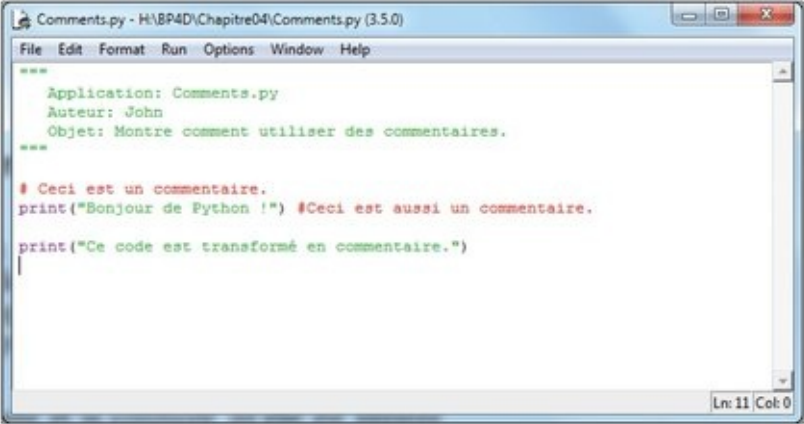
Vous pouvez trouver d'autres utilisations possibles, mais celles-ci sont les plus courantes. Regardez les exemples de ce livre. En fait, vous constaterez très vite que plus votre code devient complexe, plus vous avez besoin d'ajouter des commentaires, et plus vous avez besoin aussi de les noter avec précision de manière à ce qu'ils soient réellement utiles pour plus tard.

Utiliser des commentaires pour empêcher du code de s'exécuter

Les développeurs ne se servent pas de commentaires pour transformer certaines lignes de code afin qu'elles ne s'exécutent pas. Le cas typique est celui d'une ligne qui provoque une erreur dont on a du mal à voir la cause, ce qui peut permettre de tester le fonctionnement du reste de l'application.

Cette technique est si courante et si commode qu'elle est intégrée directement dans IDLE. Prenons un exemple pour comprendre comment cela fonctionne. Supposons que vous ayez une application qui se présente comme sur l'illustration de la [Figure 4.19](#) (vous pouvez le trouver sous le nom `Comments.py` dans le fichier téléchargeable associé à ce livre).

Figure 4.19 : Un code à commenter.



```
Comments.py - H:\BP4D\Chapitre04\Comments.py (3.5.0)
File Edit Format Run Options Window Help
'''
Application: Comments.py
Auteur: John
Objet: Montre comment utiliser des commentaires.
'''

# Ceci est un commentaire.
print("Bonjour de Python !") #Ceci est aussi un commentaire.

print("Ce code est transformé en commentaire.")
|
```

Pour une raison quelconque, vous voudriez transformer la dernière ligne en commentaire. Placez alors le point d'insertion au début de celle-ci (vous pourriez également sélectionner la ligne entière). Ouvrez ensuite le menu Format, et choisissez l'option Comment Out Region. IDLE change le code en commentaire, comme l'illustre la [Figure 4.20](#).



Lorsque vous procédez ainsi, vous pouvez remarquer que l'éditeur place deux signes dièse (##) en début de ligne. Ceci permet de différencier visuellement un commentaire que vous avez saisi manuellement, d'une ligne de code transformée provisoirement en commentaire, mais qui a vraisemblablement vocation à reprendre un jour son rang.

Figure 4.20 :
Commentez le code
que Python ne doit
pas exécuter.

```
***
Application: Comments.py
Auteur: John
Objet: Montre comment utiliser des commentaires.
***

# Ceci est un commentaire.
print("Bonjour de Python !") #Ceci est aussi un commentaire.

##print("Ce code est transformé en commentaire.")
```

Bien entendu, vous voulez vérifier par vous-même le résultat de cette opération. Sauvegardez le fichier sur votre disque (éventuellement sous un nouveau nom), puis lancez la commande Run Module dans le menu Run. La fenêtre d'exécution apparaît, et vous pouvez constater qu'une seule ligne de texte est affichée (voir la [Figure 4.21](#)). La seconde commande `print ()` a bien été ignorée par Python.

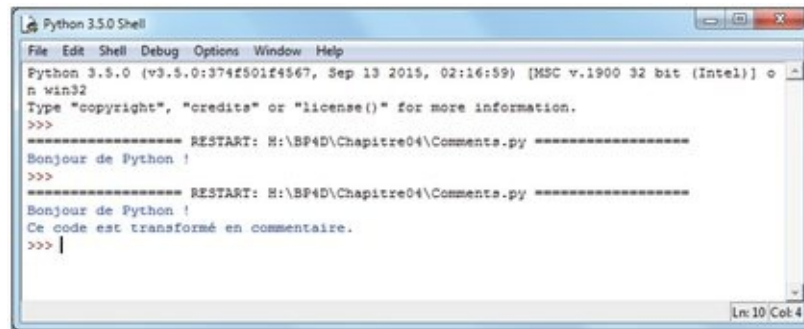
Figure 4.21 : Les
lignes de code
transformées en
commentaires ne
sont pas exécutées.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: Hi\BP4D\Chapitre04\Comments.py =====
Bonjour de Python !
>>>
```

Pour revenir à la normale, placez le point d'insertion

dans la ligne de code concernée, puis choisissez dans le menu Format la commande Uncomment Region. Il ne vous reste plus qu'à enregistrer à nouveau le fichier et à le lancer pour juger du résultat (voir la [Figure 4.22](#)).

Figure 4.22 : Les deux commandes `print()` sont à nouveau exécutées.



```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] o
n win32
Type "copyright", "credits" or "license()" for more information.
>>>
***** RESTART: H:\BP4D\Chapitre04\Comments.py *****
Bonjour de Python !
>>>
***** RESTART: H:\BP4D\Chapitre04\Comments.py *****
Bonjour de Python !
Ce code est transformé en commentaire.
>>> |
```



Vous pouvez commenter (ou l'inverse) toute une série de lignes de code en les sélectionnant d'un coup.



Si une seule ligne est transformée en commentaire, vous pouvez aussi lui rendre son statut de code en supprimant manuellement les deux signes dièse (`##`).

Charger et exécuter des applications existantes

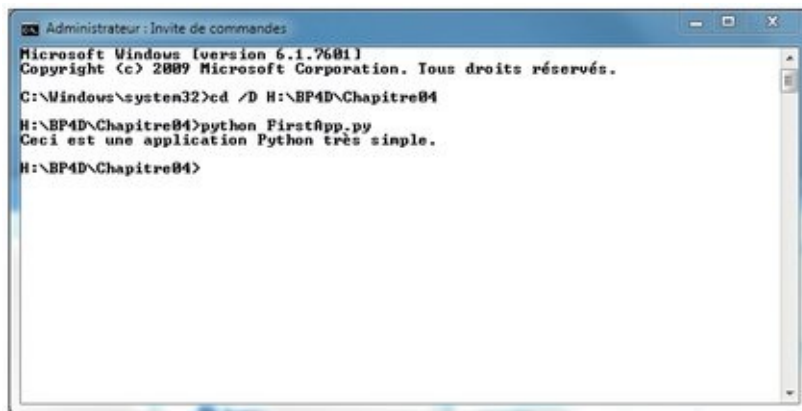
Exécuter des applications immédiatement après les avoir tapées est amusant et intéressant. Mais, à un certain moment, vous allez quitter IDLE sans avoir oublié d'enregistrer votre fichier sur le disque. Ce fichier contient votre application, mais vous devez savoir comment l'utiliser pour l'exécuter à nouveau. En fait, Python fournit pour cela un nombre considérable de méthodes. Les sections qui suivent décrivent les trois principales.

Utiliser l'Invite de commandes ou une fenêtre de terminal

Nous savons déjà comment lancer Python lui-même dans l'Invite de commandes Windows, ou encore depuis une fenêtre de terminal. Nous avons aussi appris que la ligne de commande de Python pouvait recevoir de multiples commutateurs. Ici, tout le problème consiste à se placer d'abord dans le dossier qui contient votre application, puis à taper son nom à la suite de la commande **python**.

La [Figure 4.23](#) illustre cette méthode dans le cas de notre première application, `FirstApp.py`. Mais vous pourriez évidemment lancer n'importe quelle autre application de la même manière.

Figure 4.23 : Il est possible d'exécuter une application directement depuis l'Invite de commandes ou une fenêtre de terminal.



```
Administrateur : Invite de commandes
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Windows\system32>cd /D H:\BP4D\Chapitre04
H:\BP4D\Chapitre04>python FirstApp.py
Ceci est une application Python très simple.
H:\BP4D\Chapitre04>
```

Utiliser la fenêtre d'édition

Chaque fois que vous vous servez d'IDLE, vous pouvez ouvrir une application existante et l'exécuter. Servez-vous pour cela de la commande Open dans le menu File afin de localiser et d'ouvrir le bon fichier, puis de la commande Run Module dans le menu Run. Et c'est tout.



Sous Windows, vous pouvez aussi ouvrir le dossier voulu dans l'Explorateur de fichiers. Faites ensuite un clic droit sur le nom d'un fichier d'extension `.py` (la marque de Python). Dans le menu qui s'affiche, choisissez Edit with IDLE, puis Edit with IDLE 3.5. Immédiatement, la fenêtre d'édition va apparaître avec le contenu de votre application.

Utiliser Python en mode Shell ou Ligne de commandes

Ces deux variations sur le thème de Python servent en premier lieu à taper des commandes d'une manière interactive. Cependant, vous devez connaître les commandes qui permettent de réaliser telle ou telle tâche. Dans ce cas, l'exécution d'une application contenue dans un fichier est un peu plus complexe que la simple commande `print ()` utilisée jusqu'ici. Pour exécuter par exemple l'application `FirstApp.py`, vous avez besoin de saisir l'une ou l'autre des deux commandes suivantes :

```
exec(open("C:\\BP4D\\Chapitre04\\FirstApp.py").read())  
exec(open("C:/BP4D/Chapitre04/FirstApp.py").read())
```



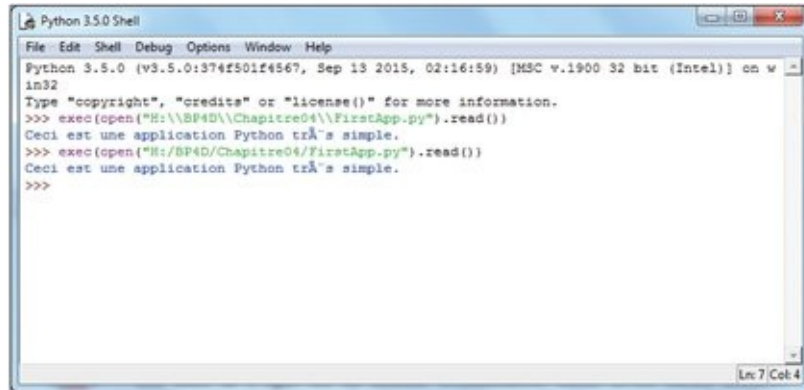
En fait, il s'agit de la même commande, mais avec deux formes différentes d'écriture des barres obliques. Python supporte les deux styles.

Voyons ce que dit cette commande :

1. **Ouvrir le fichier `FirstApp.py` qui se trouve dans le dossier `\\BP4D\\ Chapitre04` du disque `C` (commande**

- open()).**
2. **Lire le contenu de ce fichier dans l'environnement Python (commande read()).**
 3. **Exécuter ensuite les instructions contenues dans ce fichier (commande exec()).**

Figure 4.24 :
Définir une
commande pour
ouvrir, lire et
exécuter un fichier
d'application.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on w
in32
Type "copyright", "credits" or "license()" for more information.
>>> exec(open("H:\\BP4D\\Chapitre04\\FirstApp.py").read())
Ceci est une application Python tr  a simple.
>>> exec(open("H:\\BP4D\\Chapitre04\\FirstApp.py").read())
Ceci est une application Python tr  a simple.
>>>
```

Il est encore un peu t  t pour s'int  resser de tr  s pr  t    une telle ligne de commande, mais vous d  couvrirez dans la suite de ce livre que vous pouvez cr  er toutes sortes de commandes combin  es. Pour l'instant, essayez simplement cette commande pour voir si elle fonctionne. Le r  sultat est illustr   sur la [Figure 4.24](#).



Vous pourrez remarquer sur la [Figure 4.24](#) que les caract  res accentu  s sont mal trait  s. Ceci vient d'un d  faut de la commande elle-m  me, et non de Python. De ce fait, cette m  thode, par ailleurs d  j   plut  t complexe, n'est pas adapt  e au fran  ais, et sans doute plus largement aux langues comportant des caract  res accentu  s.

Refermer IDLE

Lorsque vous avez termin   votre travail avec IDLE, il ne reste plus qu'   le refermer. Vous ouvrez alors tout

simplement le menu File (Fichier), et vous constatez alors qu'il y a deux commandes de fin, ce qui peut être source de confusion :

✓ **Close** : Referme juste la fenêtre active. Autrement dit, en supposant que vous ayez ouvert à la fois une fenêtre Python en mode Shell, et une autre en mode Édition avec IDLE, la commande Close fermera la première fenêtre, mais pas la seconde.

✓ **Exit** : Referme la fenêtre courante et toutes les autres fenêtres associées. Supposons par exemple que vous avez ouvert une application dans IDLE, puis lancé son exécution dans une fenêtre Shell. Vous avez donc maintenant deux fenêtres. En choisissant Exit dans l'une ou l'autre, elles vont se fermer toutes les deux. Vous venez de dire « Bonne nuit » à Python !



N'oubliez pas de sauvegarder votre travail avant de quitter. Si vous ne l'avez pas fait, IDLE vous enverra un message de rappel.



En réalité, vous ne quittez pas réellement Python, mais uniquement votre session courante. Si vous avez ouvert par exemple deux sessions avec deux fichiers différents, vous devrez utiliser Close ou Exit pour la première session, et recommencer pour la seconde.

Deuxième partie

Apprendre la langue de Python

Dans cette partie...

- ▶ Apprendre à créer des variables pour traiter des données.
- ▶ Créer des fonctions pour rendre le code plus facile à lire.
- ▶ Dire à votre application Python de prendre une décision.
- ▶ Effectuer des tâches répétitives.
- ▶ S'assurer que votre application ne comporte pas d'erreurs.

Chapitre 5

Enregistrer et modifier des informations

Dans ce chapitre :

- ▶ Comprendre le stockage des données.
- ▶ Comprendre les différents types de stockage des données.
- ▶ Ajouter des dates et des heures aux applications.

Souvenez-vous que le début du Chapitre 3 vous proposait de retenir l'acronyme CLAS (Créer, Lire, Actualiser, Supprimer). C'est une méthode simple pour se souvenir des tâches que l'ordinateur effectue à partir des informations que vous voulez gérer. Bien entendu, les *geeks* se servent d'un autre mot pour désigner ces informations. Ils les appellent des *données*, mais informations ou données, le sens est le même.



Pour que vos informations soient utiles, vous devez disposer de certaines méthodes pour qu'elles soient conservées de manière permanente. Sinon, vous

devriez tout reprendre à zéro chaque fois que l'ordinateur se mettrait en marche. De plus, Python doit fournir des règles pour traiter toutes ces informations. Ce chapitre est donc consacré au contrôle des informations : comment les stocker de manière permanente, et comment les manipuler dans vos applications.

Enregistrer des informations

Une application doit pouvoir accéder rapidement aux informations. Sinon, elles pourraient mettre pas mal de temps à réaliser les tâches qui leur sont dévolues. Elles stockent donc ces informations (ou données) en mémoire. Mais cette mémoire n'est que temporaire. Lorsque vous éteignez la machine, il est indispensable que vos données soient enregistrées sous une forme permanente, qu'il s'agisse de votre disque dur, d'un disque ou d'une clé USB, d'une carte mémoire, d'un disque réseau, ou encore d'un nuage Internet. De plus, vous devez tenir compte de la nature de ces données, par exemple des nombres ou du texte. Les sections qui suivent discutent de ces sujets plus en détail.

Considérer les variables comme des boîtes de rangement

Lorsque vous travaillez avec des applications, vous enregistrez des informations (ou des données) dans des variables. Une *variable*, c'est comme une sorte de boîte de rangement. Lorsque vous voulez accéder à une certaine information, vous utilisez la variable qui la contient. Si vous avez de nouvelles informations à

enregistrer, vous les placez dans une variable. Changer des informations implique d'accéder d'abord à la bonne variable (la bonne boîte de rangement), puis à y enregistrer les nouvelles valeurs. Et vous pouvez bien sûr aussi vider une boîte de rangement, et donc également une variable. Bref, les variables, c'est comme dans la vraie vie.



Les ordinateurs sont des machines plutôt soigneuses quant au rangement des affaires. Chaque variable contient uniquement un objet, ou plutôt une pièce d'information. Ceci permet de retrouver beaucoup plus facilement cette pièce d'information particulière, sans avoir à fouiller dans les tréfonds d'une gigantesque malle en osier. Vous n'avez pas eu à rencontrer de variables dans les précédents chapitres, mais sachez que la plupart des applications reposent sur une utilisation intensive de boîtes de rangement électroniques.

Utiliser la bonne boîte pour enregistrer les bonnes données

Trop de gens ont tendance à ranger les choses dans le mauvais type de boîte. Il n'est pas si rare par exemple de retrouver des chaussures dans un sac à vêtements, et un foulard dans une boîte à chaussures. Python, lui, aime que tout soit au bon endroit. C'est pourquoi vous trouvez des nombres qui sont enregistrés dans une sorte de variable, et des textes qui sont enregistrés dans une autre sorte de variables. Bien sûr, il s'agit dans tous les cas de variables, mais il y en a différents types pour stocker des informations de nature différente. Cela s'appelle ne pas mélanger les

torchons avec les serviettes.

Utiliser des variables spécialisées permet de travailler avec leur contenu en utilisant des outils spécifiques. Pour l'instant, vous n'avez pas à vous soucier des détails. Retenez simplement que chaque type d'information est enregistré dans un type spécial de variable.



Si Python sait tenir sa maison, ce n'est en fait pas le cas des ordinateurs qui n'ont aucune notion sur la nature des informations (donc des données). Tout ce qu'un ordinateur connaît, ce sont des suites gigantesques de 0 et de 1, qui correspondent à la présence ou à l'absence d'un certain voltage. À un niveau plus élevé, les ordinateurs sont certes capables de travailler avec des nombres, mais ce n'est au final qu'une extension de cette affaire de 0 et de 1. Les nombres, les lettres, les dates, les heures, et tout ce à quoi vous pouvez penser en termes d'informations, se traduisent en définitive par des 0 et des 1 dans un système informatique. On est bien peu de choses, tout de même. Ainsi, la lettre A majuscule est en réalité enregistrée sous la forme 01000001, qui est numérique et se traduit dans le système décimal par la valeur 65. En réalité, l'ordinateur n'a aucune idée en général, et en particulier de ce que peut bien être la lettre A, pas plus de ce qu'est une date comme le 31/12/2016.

Python et ses principaux types de données

Chaque langage de programmation définit des

variables destinées à contenir différents types d'informations, et Python ne fait pas exception à la règle. Une variété particulière de variable est appelée un *type de donnée*. Connaître le type de donnée d'une variable est important, puisque c'est lui qui vous indique quel genre d'information celle-ci contient. De plus, pour enregistrer une information dans une variable, il faut en choisir une qui possède le bon type de donnée. Ainsi, Python ne vous permet pas d'enregistrer du texte dans une variable conçue pour contenir des données numériques. D'abord, votre texte risquerait d'être transformé en une bouillie de chiffres, et ensuite votre application risquerait tout bonnement de ne pas fonctionner. Vous pouvez en gros classer les types de données de Python en trois catégories principales : les nombres, les textes et les valeurs logiques (ou *booléennes*). Mais il n'y a pas vraiment de limites quant à la manière dont tout cela peut se présenter du point de vue d'un humain. Les sections qui suivent décrivent ces différents types de données standards.

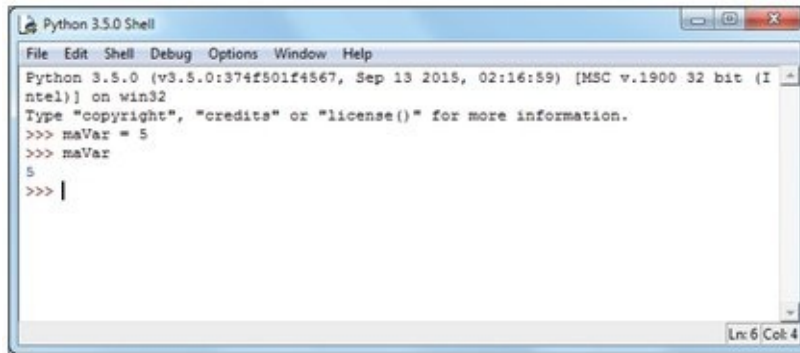
Placer des informations dans des variables

Pour placer une valeur dans une variable, vous vous servez du signe égal (=), que l'on appelle l'*opérateur d'affectation* (autrement dit, vous *affectez* une valeur à une variable). Vous retrouverez la cohorte des opérateurs Python de base dans le Chapitre 6. Mais vous avez besoin dès maintenant de vous familiariser avec celui-ci.

Supposons donc simplement que vous vouliez stocker le nombre 5 en mémoire, c'est-à-dire dans une boîte,

et donc dans une variable. Il faut que vous donniez un nom à celle-ci, sans quoi personne ne pourrait savoir de quoi on parle. Disons que ce nom sera `maVar`. En faisant appel à l'opérateur d'affectation, vous tapez donc **`maVar = 5`** dans la fenêtre de Python, et vous appuyez sur Entrée. Même si Python engloutit cette ligne sans rien dire, il a compris de quoi il s'agissait. Pour le voir, tapez le nom de votre variable et appuyez sur Entrée. Python va afficher sa valeur, donc le contenu de cette boîte particulière (voir la [Figure 5.1](#)).

Figure 5.1 :
Utilisez l'opérateur d'affectation pour placer une information dans une variable.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> maVar = 5
>>> maVar
5
>>> |
```



Les noms des variables sont sensibles à la capitalisation. Si vous avez nommé votre variable `maVar`, taper **`mavar`**, qui est une autre variable, totalement différente, du point de vue de Python, provoquera une erreur puisque cette dernière n'a jamais été définie.

Comprendre les types numériques

Les humains ont tendance à penser aux nombres en termes généraux (d'ailleurs, beaucoup ne font même pas la différence entre chiffres et nombres). Nous

voyons par exemple 1 et 1.0, ou 1,0, comme étant le même nombre. C'est la même valeur unitaire, mais la seconde forme contient en plus un séparateur décimal. Pour ce qui nous concerne, nous, humains, ces deux nombres sont égaux et parfaitement interchangeables. Mais Python a un avis très différent. Pour lui, il s'agit de deux types de nombres différents, et chaque format nécessite un mode de traitement spécifique. Les sections qui suivent décrivent les types numériques entiers, en virgule flottante, ainsi que les nombres complexes supportés par Python.

Entiers

Un *entier*, c'est... comment dire... un nombre entier. Par exemple, 1 est un entier. D'un autre côté, 1.0 n'est pas un entier. C'est un nombre décimal. Point. Dans Python, les entiers sont représentés par un type de donnée appelé `int` (pour *integer* en anglais).



Comme les boîtes de rangement, les variables ont des capacités de stockage plus ou moins limitées. Essayez de placer une valeur trop grande dans une variable, et vous obtiendrez une erreur en retour. Sur la plupart des plates-formes, les entiers peuvent appartenir à un intervalle compris entre - 9,223,372,036,854,775,808 et 9,223,372,036,854,775,807 (en fait, ce sont les limites qu'est capable d'atteindre une variable sur 64 bits). Même si cette plage est énorme, elle n'est pas infinie.

Lorsque vous travaillez avec le type `int`, vous avez accès à nombre de fonctionnalités intéressantes. Nous retrouverons la plupart d'entre elles le moment venu, mais une de ces fonctionnalités est la capacité de ce type d'utiliser différentes bases numériques :

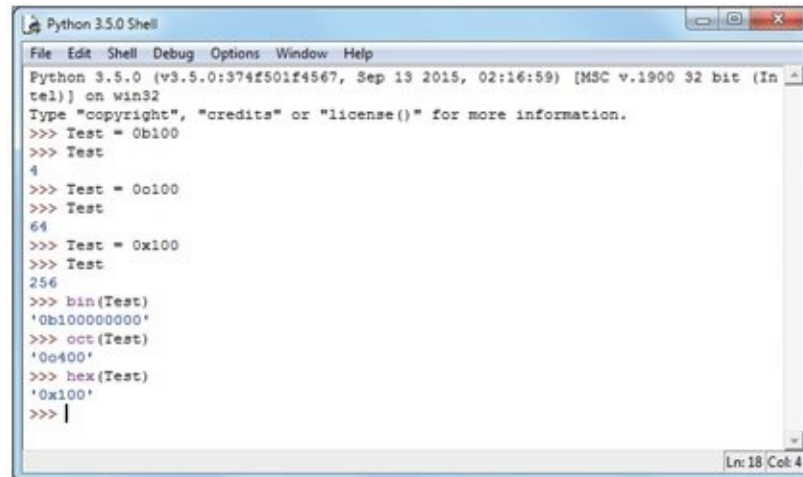
- ✓ **Base 2** : Utilise uniquement des 0 et des 1.
- ✓ **Base 8** : Utilise des chiffres de 0 à 7.
- ✓ **Base 10** : C'est notre système décimal de tous les jours.
- ✓ **Base 16** : C'est le système *hexadécimal*, qui utilise les chiffres de 0 à 9 ainsi que les lettres de A à F pour obtenir seize « chiffres » différents.

Pour indiquer à Python quelle base vous voulez utiliser (en dehors de la base 10), vous ajoutez au nombre un 0 suivi d'une lettre spéciale. Par exemple, la syntaxe `0b100` est la valeur un-zéro-zéro en base 2, soit 4 dans le système décimal. Ces lettres spéciales sont les suivantes :

- ✓ **b** : Base 2
- ✓ **o** : Base 8
- ✓ **x** : Base 16

Il est aussi possible de convertir des valeurs numériques d'une base vers une autre en utilisant les commandes `bin()`, `oct()` et `hex()`. La [Figure 5.2](#) vous en propose quelques exemples. Essayez-les vous-même en variant les valeurs pour mieux voir comment tout cela fonctionne. Utiliser une base différente rend en fait les choses plus faciles dans de nombreuses situations. Vous en trouverez quelques-unes plus loin dans ce livre. Pour l'instant, retenez simplement que le type `int` est capable de supporter des valeurs entières dans plusieurs bases numériques.

Figure 5.2 : Le type `int` peut contenir des valeurs dans plusieurs bases numériques.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Test = 0b100
>>> Test
4
>>> Test = 0o100
>>> Test
64
>>> Test = 0x100
>>> Test
256
>>> bin(Test)
'0b100000000'
>>> oct(Test)
'0o400'
>>> hex(Test)
'0x100'
>>> |
```

Valeurs en virgule flottante

Tout nombre qui comprend une partie décimale est dit en *virgule flottante*. C'est par exemple le cas de 1.0. La différence entre un entier et un nombre en virgule flottante est donc très facile à repérer : il y a un point ou pas. Python enregistre les nombres en virgule flottante dans le type de donnée `float`.

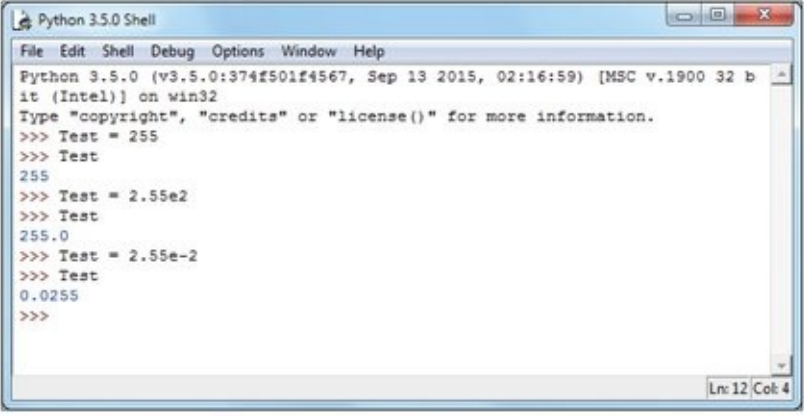


Les valeurs en virgule flottante ont un avantage sur les entiers purs et durs : vous pouvez y enregistrer des nombres immensément grands, ou incroyablement petits. Bien entendu, il y a aussi des limites. Dans leur cas, la plus grande valeur possible est $\pm 1.7976931348623157 \times 10^{308}$ et la plus petite est $\pm 2.2250738585072014 \times 10^{-308}$, du moins sur la plupart des plates-formes.

Lorsque vous travaillez avec des nombres en virgule flottante, vous pouvez affecter leur valeur à des variables de différentes manières. Les deux méthodes les plus courantes consistent à saisir directement le nombre, ou à utiliser la notation scientifique. Dans ce dernier cas, un `e` sépare le nombre de son exposant. La [Figure 5.3](#) illustre ces deux techniques. Remarquez

qu'un exposant négatif se traduit par une valeur fractionnaire.

Figure 5.3 : Les valeurs en virgule flottante peuvent être affectées de plusieurs manières.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 b
it (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Test = 255
>>> Test
255
>>> Test = 2.55e2
>>> Test
255.0
>>> Test = 2.55e-2
>>> Test
0.0255
>>>
```

Nombres complexes

Peut-être avez-vous étudié les nombres complexes sur les bancs du lycée ou de l'université. Un *nombre complexe* est formé par l'association entre un nombre réel et un nombre imaginaire. Ces nombres étranges sont utilisés dans différents domaines :

- Ingénierie électrique
- Dynamique des fluides
- Mécanique quantique
- Graphismes sur ordinateur
- Systèmes dynamiques

Ils ont encore d'autres usages, mais cette liste vous donne une idée générale. Si vous n'êtes pas concerné par une de ces disciplines, il est vraiment peu probable que vous ayez un jour à les rencontrer. Cependant, Python est l'un des rares langages qui supportent en natif ces nombres complexes. Comme vous le découvrirez au fil de ce livre, Python est très doué pour tout ce qui est scientifique et technique.

La partie imaginaire d'un nombre complexe est suivie

de la lettre *j*. Si vous voulez par exemple créer un nombre complexe dont la partie réelle est 3 et la partie imaginaire 4, vous pourrez saisir une affectation telle que celle-ci :

```
monComplexe = 3 + 4j
```

Pour voir uniquement la partie réelle, tapez alors simplement **monComplexe.real** à la suite de l'indicatif de Python et appuyez sur Entrée. De même, pour afficher la partie imaginaire, vous tapez **monComplexe.imag** et vous appuyez sur Entrée.

Mais pourquoi autant de types de données numériques ?

De nombreux programmeurs, y compris parmi les plus anciens, ont du mal à comprendre pourquoi il faut plusieurs types de données numériques. Après tout, les êtres humains n'ont pas besoin de tout cela. Pour mieux appréhender cette notion, vous avez besoin d'en savoir un peu plus sur la manière dont un ordinateur traite les nombres.

Un entier est enregistré simplement sous la forme de bits que l'ordinateur est capable d'interpréter directement. Par exemple, une valeur binaire égale à 100 se traduit par le chiffre 4 dans le système décimal. Mais, d'un autre côté, les nombres qui contiennent un point décimal (dans le langage des

ordinateurs, ou une virgule décimale dans notre propre système) ont besoin d'être enregistrés d'une manière totalement différente. Souvenez-vous un peu de vos années d'école, et vous comprendrez mieux ce qui se passe. Un nombre décimal (ou en virgule flottante) doit être enregistré avec un signe plus ou moins (ce qui ne demande en fait qu'un bit), suivi de sa *mantisse* (la partie fractionnaire du nombre) et d'un *exposant* (une puissance de 2). Pour obtenir une valeur en virgule flottante, vous utilisez donc une équation du style :

$$\text{Valeur} = \text{Mantisse} * 2^{\text{exposant}}$$

Bon. Il est vrai que le sujet déborde largement du cadre de ce livre, mais vous pouvez réviser vos anciens cours de mathématiques (ou faire une recherche à ce sujet sur Internet). Dans tous les cas, rien ne vaut le fait de jouer avec toutes ces valeurs. Mais, pour ce qui nous concerne ici, reprenez seulement que les nombres en virgule flottante consomment davantage d'espace en mémoire que les nombres entiers. D'ailleurs, ils utilisent une partie différente du microprocesseur (bien sûr moins rapide que tout ce qui concerne uniquement des entiers).

Ces derniers offrent en plus une précision bien supérieure à celle des nombres en virgule flottante, qui n'offrent qu'une approximation de leurs valeurs (aussi fines soient-elles). Par contre, les nombres en virgule flottante permettent de stocker des valeurs bien supérieures à celle des entiers.

La vérité dans tout cela, c'est que les valeurs décimales (ou en virgule flottante) sont indispensables pour se confronter au monde réel, mais qu'utiliser des entiers chaque fois que vous le pouvez réduit la quantité de mémoire utilisée par votre application, tout en accélérant le fonctionnement de celle-ci.

Comprendre les valeurs booléennes

Cela peut paraître un peu étrange, mais les ordinateurs vous donnent toujours une réponse franche et directe. Un ordinateur ne vous dira jamais « Peut-être que oui, peut-être que non, va savoir ». Toute réponse que vous obtenez de sa part se décline en deux catégories : c'est vrai (*true* en anglais) ou c'est faux (*false* en anglais). *True* ou *false*, là est la question...

En fait, il existe toute une branche des mathématiques appelée algèbre booléenne. Elle doit son origine au mathématicien anglais George Boole (un super geek de cette époque) grâce auquel (même s'il vivait au 19^e siècle) les ordinateurs sont capables de prendre des décisions. Et, pour dire vrai, l'algèbre booléenne existe depuis 1854, donc bien avant l'invention des ordinateurs. Et si l'on dit souvent que l'invention de ces extraordinaires machines doit tout à Alan Turing (1912 - 1954), elles ne seraient rien sans George Boole. Comme quoi...

Lorsque vous utilisez une valeur booléenne dans Python, vous faites appel au type `bool`. Une variable de

ce type ne peut contenir que deux valeurs : vrai (`true`) ou faux (`false`). Vous pouvez définir une valeur en utilisant un de ces deux mots-clés, ou créer une expression qui définit une idée logique valant ou bien *true*, ou bien *false*.

Par exemple, vous pourriez dire `monBooleen = 1 > 2`. Mais vous comprenez alors que 1 étant plus petit que 2, cette expression sera toujours fausse. Vous trouverez dans ce livre nombre d'exemples utilisant le type `bool`. Retenez donc pour l'instant uniquement ce concept sans chercher à aller plus loin.

Déterminer le type d'une variable

Vous aurez parfois besoin de déterminer le type d'une variable. Il peut s'agir d'un cas où vous n'êtes pas certain de la réponse, ou d'une situation dans laquelle vous avez récupéré le code de quelqu'un d'autre dont le code source n'est pas accessible ou insuffisamment documenté.

Dans tous les cas, vous pouvez utiliser la commande `type ()`. Si vous placez par exemple la valeur 5 dans la variable `monInt` (en tapant **monInt = 5** suivi d'un appui sur la touche Entrée), et que vous oubliez plus tard de quoi il s'agit, vous pourrez retrouver cette information en tapant **type(monInt)** puis en appuyant sur Entrée). La réponse sera `<class 'int'>`, ce qui vous informera que la variable `monInt` contient une

valeur entière.

Comprendre les chaînes de caractères

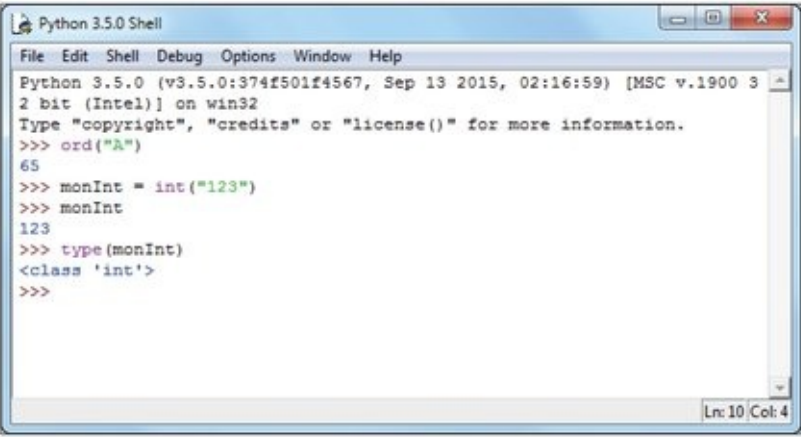
De tous les types de données, les chaînes de caractères sont les plus faciles à comprendre par des êtres humains comme vous et moi, et parmi les plus difficiles à comprendre pour les ordinateurs. Si vous avez lu les précédents chapitres de ce livre, vous avez déjà fait connaissance avec ces chaînes de caractères (un texte quelconque, si vous préférez). Par exemple, tout le code proposé dans le Chapitre 4 repose sur des chaînes de caractères. Ainsi, l'affectation `maChaine = « Python est un grand langage »` affecte une chaîne de caractères (*Python est un grand langage*) à une variable appelée *maChaine*.

De son côté, l'ordinateur n'a aucune notion de ce que sont des lettres (ou des chiffres, ou des signes de ponctuation, et ainsi de suite). Pour lui, seuls comptent les nombres en mémoire. Par exemple, le caractère A majuscule a un code numérique, soit 65 en valeur décimale. Pour en avoir le cœur net, tapez **`ord(« A »)`** (suivi bien sûr d'un appui sur la touche Entrée) et voyez la réponse qui est renvoyée. Elle va bien indiquer 65. Il est donc possible, au passage, de convertir une lettre quelconque en une valeur numérique grâce à cette commande `ord ()`.

L'ordinateur ne comprend donc rien aux chaînes de caractères, mais ces dernières sont très utiles dans pratiquement toutes les applications. Vous aurez donc aussi besoin d'en convertir certaines en valeurs

numériques. Vous aurez pour cela besoin des commandes `int ()` et `float ()`. Comme vous vous en doutez certainement, la première effectue une conversion vers une valeur entière, et la seconde vers une valeur en virgule flottante. Si vous tapez par exemple **`monInt = int(« 123 »)`** suivi d'un appui sur Entrée, vous allez créer une variable de type `int` qui contient la valeur `123`. La [Figure 5.4](#) illustre ce mécanisme.

Figure 5.4 :
Convertir une chaîne en nombre est facile grâce aux commandes `int()` et `float()`.

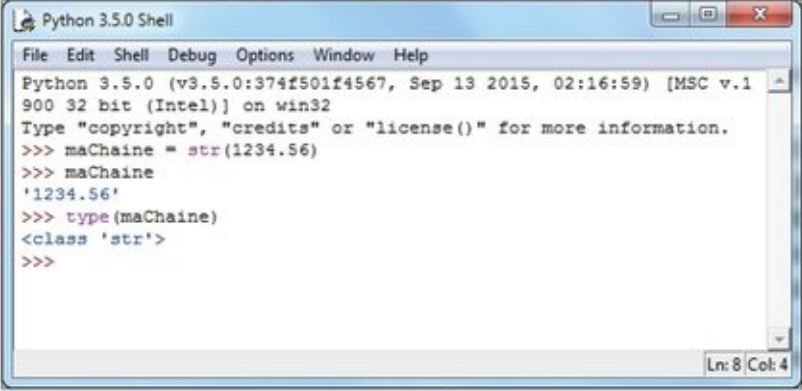
A screenshot of a Python 3.5.0 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following code and output:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ord("A")
65
>>> monInt = int("123")
>>> monInt
123
>>> type(monInt)
<class 'int'>
>>>
```

The status bar at the bottom right indicates 'Ln: 10 / Col: 4'.

Vous pouvez effectuer la conversion inverse en vous servant de la commande `str()`. Si vous tapez par exemple **`maChaine = str(1234.56)`** (suivi d'un appui sur Entrée), vous allez créer une chaîne de caractères contenant la valeur « `1234.56` » que vous assignez à la variable chaîne `maChaine`. La [Figure 5.5](#) illustre cette technique. L'important est que vous pouvez alterner ainsi entre chaînes et nombres avec une grande facilité. Les chapitres ultérieurs vous en diront plus à ce sujet.

Figure 5.5 :
Convertir des
nombres en
chaînes, c'est aussi
possible.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1
900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> maChaine = str(1234.56)
>>> maChaine
'1234.56'
>>> type(maChaine)
<class 'str'>
>>>
```

Travailler avec des dates et des heures

Les dates et les heures sont des données couramment utilisées. Qu'il s'agisse d'un rendez-vous, d'un événement quelconque, d'un anniversaire ou tout simplement de se réveiller le matin, ce type de donnée rythme notre existence. Du fait de cette familiarité avec notre vie, il est utile, et même nécessaire, de savoir comment Python traite tout cela. Et, comme les ordinateurs ne connaissent que des nombres, les dates et les heures n'ont pas plus de sens pour eux que les chaînes de caractères.

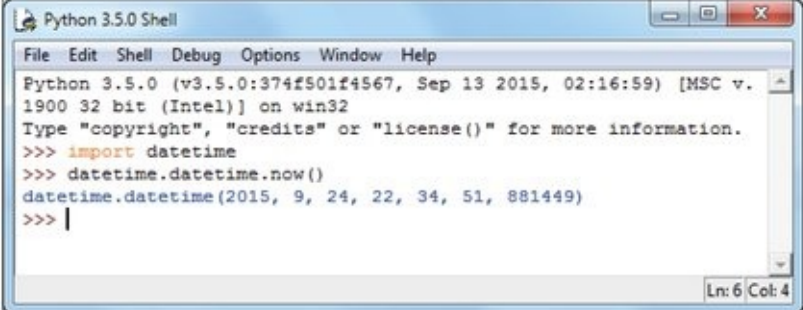


Pour travailler avec des dates et des heures, vous devez opérer une tâche particulière dans Python. Celui-ci est un peu comme un ordinateur, et il ne sait pas de quoi on parle tant que vous n'avez pas importé (`import`) un module particulier appelé `datetime`. Mais voilà un sujet sur lequel nous reviendrons dans le Chapitre 10. Ne vous souciez donc pas pour l'instant des tenants et des aboutissants de cette commande. Tout ce que vous voulez dans l'immédiat, c'est pouvoir faire quelque chose avec les dates et les heures.

Les ordinateurs contiennent des horloges. Mais celles-

ci servent surtout aux humains qui utilisent l'ordinateur. Tournez le problème dans tous les sens, et vous arriverez à la même conclusion... Pour obtenir l'heure courante, vous pouvez taper **datetime.datetime.now()** suivi d'un appui sur Entrée. Vous allez alors voir s'afficher des informations extrêmement précises sur le jour, l'heure et même des fractions de seconde (voir la [Figure 5.6](#)).

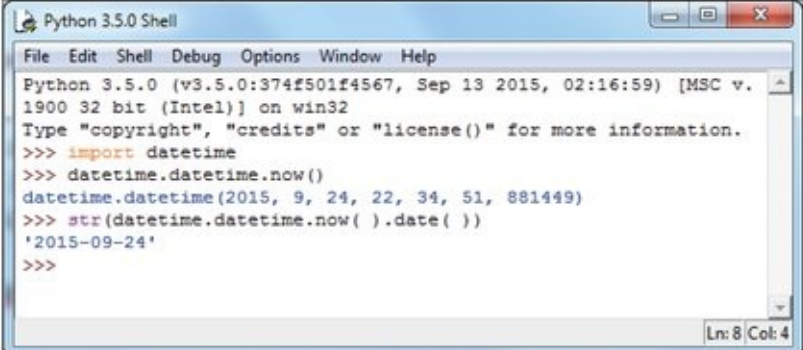
Figure 5.6 : Pour obtenir la date et l'heure courantes, utilisez la commande `now()`.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.
1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2015, 9, 24, 22, 34, 51, 881449)
>>> |
```

Certes, ces informations sont un peu dures à lire dans le format tel qu'il est affiché. Supposons donc que vous vouliez uniquement récupérer la date du jour courant dans un format lisible. Pour accomplir cette tâche, vous allez devoir combiner plusieurs choses qui ont été vues dans les sections précédentes. Tapez **str(datetime.datetime.now().date())** et appuyez sur Entrée. La [Figure 5.7](#) illustre le résultat obtenu, tout en vous montrant que la réponse est affichée au format anglo-saxon (autrement dit, dans le sens année-mois-jour).

Figure 5.7 : La commande `str()` permet d'afficher le jour courant dans un format plus lisible.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.
1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2015, 9, 24, 22, 34, 51, 881449)
>>> str(datetime.datetime.now().date())
'2015-09-24'
>>>
```

De la même manière, Python dispose aussi d'une commande `time ()` qui permet d'obtenir l'heure courante. Il est aussi possible de séparer toutes ces informations en faisant appel aux commandes `day` (jour), `month` (mois), `year` (année), `hour` (heure), `minute`, `second` et `microsecond`. Les chapitres qui suivent vous aideront à mieux comprendre comment utiliser ces fonctions de date et d'heure afin de permettre aux utilisateurs de vos applications de rester informés à chaque instant.

Chapitre 6

Gérer l'information

Dans ce chapitre :

- ▶ Comprendre la manière dont Python voit les données.
- ▶ Utiliser des opérateurs pour affecter, modifier et comparer des données.
- ▶ Organiser le code à l'aide de fonctions.
- ▶ Interagir avec l'utilisateur.

Que vous utilisiez l'un des termes *information* ou *données* pour faire référence aux contenus gérés par votre application, cela ne fait aucune différence. En fait, j'utilise ces mots de manière interchangeable dans ce livre. Mais, dans tous les cas, vous devez fournir certaines procédures pour travailler avec ces informations, ou ces données. Sinon, votre application ne servirait strictement à rien. Vous avez donc besoin de méthodes qui vous permettent d'affecter des valeurs à des variables, de modifier le contenu de ces variables afin d'atteindre des buts spécifiques, et de comparer ces contenus avec des résultats attendus. Ce chapitre traite donc de ces trois sujets.

Il est aussi essentiel de commencer à travailler avec

des méthodes vous permettant d'écrire un code compréhensible. Certes, vous pourriez vous contenter d'écrire ce code comme une (très) longue procédure, mais essayer alors de s'y retrouver serait une tâche incroyablement ardue, et vous devriez en plus répéter certaines étapes devant être exécutées plusieurs fois. Les fonctions sont un des outils dont vous disposez pour « emballer » votre code afin qu'il soit plus facile à lire, à comprendre et à réutiliser.

Les applications ont également besoin d'interagir avec l'utilisateur. Bien sûr, ce n'est pas une totale obligation. Mais le cas contraire est non seulement extrêmement rare, mais aussi excessivement limité. Pour être à même d'offrir à l'utilisateur un certain service, les applications doivent donc être capables de communiquer avec l'utilisateur pour savoir comment il veut gérer les données. Cette question sera également abordée dans ce chapitre. Mais, à l'évidence, nous aurons à y revenir tout au long de ce livre.

Contrôler la manière dont Python voit les données

Comme vous l'explique le Chapitre 5, toutes les données contenues dans votre ordinateur sont enregistrées sous la forme de 0 et de 1. L'ordinateur ne comprend pas les concepts de lettres, de données booléennes, de dates, d'heures, ou tout autre type d'information à l'exception des nombres. De plus, la capacité que possède l'ordinateur à travailler avec des nombres est à la fois inflexible et relativement simpliste. Lorsque vous manipulez une chaîne dans Python, vous dépendez de la manière dont Python

traduit le concept de chaîne de caractères dans une forme compréhensible par l'ordinateur. Il est donc important de comprendre que la manière dont Python « voit » les données est différente de votre propre perception ou du mode de fonctionnement de l'ordinateur. Python fonctionne comme un intermédiaire chargé de rendre vos applications fonctionnelles.



Pour gérer les données dans une application, celle-ci doit contrôler la manière dont Python voit les données. Pour atteindre cet objectif, vous avez notamment à votre disposition les opérateurs, des méthodes telles que les fonctions, ainsi que des outils permettant d'interagir avec l'utilisateur. Toutes ces techniques reposent en partie sur la notion de comparaison. Déterminer ce qu'il faut faire ensuite signifie comprendre l'état actuel des données par comparaison avec un autre état. Si, par exemple, une variable contient en ce moment le prénom Jean, alors que vous voudriez qu'elle contienne le prénom Pierre, vous devez disposer d'un moyen vous permettant de connaître son contenu actuel. Ce n'est qu'après cela que vous pouvez décider de modifier sa valeur.

Faire des comparaisons

La méthode principale de Python pour effectuer des comparaisons repose sur l'emploi d'opérateurs. Plus généralement, les opérateurs jouent un rôle majeur dans la manipulation des données. Pour en savoir plus à leur sujet, voyez plus loin dans ce chapitre la section « Travailler avec les opérateurs ». Et les chapitres qui suivent feront un usage extensif de ces opérateurs afin de créer des applications capables de prendre des

décisions, d'effectuer des tâches répétitives et d'interagir avec l'utilisateur. Cependant, l'idée de base est que les opérateurs aident les applications à réaliser différents types de comparaisons.

Dans certains cas, vous utiliserez des voies un peu détournées pour effectuer des comparaisons dans une application. Par exemple, vous pouvez comparer la sortie produite par deux fonctions (voyez la section « Comparer les sorties de fonctions », plus loin dans ce chapitre). Avec Python, il est possible d'effectuer des comparaisons selon différents niveaux afin de gérer vos données sans problème. Ces techniques masquent les détails de ces processus de manière à ce que vous puissiez vous concentrer sur l'essentiel : effectuer des comparaisons et définir les réactions qui en découlent.

Les choix que vous effectuez parmi ces techniques affectent la manière dont Python voit les données, et déterminent également les sortes de choses que vous pouvez faire afin de gérer les données une fois ces comparaisons effectuées. Tout cela peut paraître pour l'instant absurdement complexe, mais le point important à mémoriser est que les applications ont besoin de comparaisons pour interagir correctement avec les données.

Comprendre comment les ordinateurs effectuent des comparaisons

Les ordinateurs sont incapables de comprendre ce que sont par exemple les fonctions, ou toute autre structure que vous créez avec Python. Toutes ces

choses n'existent que pour votre propre bénéfice, pas celui de l'ordinateur. Cependant, les ordinateurs supportent directement le concept d'opérateur. De ce fait, la plupart des opérateurs de Python ont un corollaire direct avec une commande compréhensible par l'ordinateur. Par exemple, lorsque vous avez besoin de savoir si un nombre est plus grand qu'un autre, l'ordinateur est capable d'effectuer cette comparaison.



Si vous demandez à Python de comparer deux chaînes de caractères, ce qu'il fait en réalité, c'est de comparer la valeur numérique interne associée à chaque caractère de la chaîne. Par exemple, la lettre A est stockée dans la mémoire de l'ordinateur sous la forme d'une valeur numérique égale à 65 dans le système décimal. Par contre, le a minuscule a une autre valeur numérique, en l'occurrence 97. De ce fait, même si vous considérez que ABC et abc sont identiques dans le sens où elles sont formées des trois premières lettres de l'alphabet, l'ordinateur a un point de vue radicalement différent. Il vous dira que abc est plus grand que ABC, puisque 97 est plus grand que 65.

Travailler avec les opérateurs

Les opérateurs sont à la base du contrôle et de la gestion des données dans les applications. Vous utilisez des opérateurs pour définir comment un morceau de donnée est comparé avec un autre, ainsi que pour modifier l'information contenue dans une variable. Les opérateurs jouent donc un rôle essentiel dans tout calcul ainsi que dans l'affectation de données.



Lorsque vous utilisez un opérateur, vous devez fournir soit une variable, soit une expression. Vous savez déjà qu'une variable est une sorte de boîte de rangement servant à contenir des données. Une *expression* est une équation ou une formule qui fournit une description d'un concept mathématique. Dans la plupart des cas, le résultat renvoyé par l'évaluation d'une expression est une valeur booléenne (vraie ou fausse). Les sections qui suivent décrivent en détail les opérateurs.

Comprendre l'opérateur ternaire de Python

Un opérateur ternaire nécessite trois éléments. Python supporte un seul opérateur de ce type que vous pouvez utiliser pour déterminer la valeur vraie d'une expression. Cet opérateur prend la forme suivante :

➤ ValeurVrai if Expression else ValeurFaux

Si Expression est vraie, l'opérateur renvoie la valeur ValeurVrai. Sinon, il retourne ValeurFaux. Par exemple, si vous tapez :

➤ "Bonjour" if True else "Au revoir"

l'opérateur va retourner Bonjour. Et si vous tapez :

➤ "Bonjour" if False else "Au revoir"

il retournera Au revoir.

Cet opérateur est commode dans des situations où vous avez besoin de prendre rapidement une décision et que vous ne voulez pas écrire un tas de code pour y arriver.

L'un des avantages de Python, c'est qu'il offre normalement plusieurs manières d'arriver à un certain résultat. En l'occurrence, il offre une version plus courte de cet opérateur, sous la forme :

➤ (ValeurFaux, ValeurVrai) [Expression]

L'exemple ci-dessus pourrait donc s'écrire :

➤ ("Au revoir", "Bonjour") [True]

ou encore

➤ ("Au revoir", "Bonjour") [False]

avec les mêmes résultats.

Simplement, la première forme est plus claire, et la seconde plus courte.

Définir les opérateurs

Un *opérateur* accepte une ou plusieurs entrées sous la forme de variables ou d'expressions, il effectue une certaine tâche (comme une comparaison ou une addition), et il renvoie une sortie en fonction du résultat obtenu. La classification des opérateurs se fait pour partie selon leur effet, et pour partie selon le nombre d'éléments qu'ils réclament. Par exemple, un opérateur *unaire* n'a besoin que d'une seule variable ou expression, tandis qu'un opérateur *binaire* en a besoin de deux.



Les éléments passés à un opérateur sont appelés des *opérandes*. Dans le cas d'un opérateur binaire, vous avez donc un opérande gauche (à gauche), et un opérande droit (à droite). Python offre plusieurs catégories d'opérateurs :

- Unaire
- Arithmétique
- Relationnel
- Logique
- Au niveau du bit
- Affectation
- Membre
- Identité

Chacune de ces catégories réalise une tâche spécifique. Par exemple, les opérateurs arithmétiques effectuent des calculs mathématiques, tandis que les opérateurs relationnels servent aux comparaisons. Voyons donc cela plus en détail.

Unaire

Un opérateur unaire n'a besoin que d'une seule variable ou expression. Ils servent souvent dans des processus de décision (par exemple pour déterminer si une chose n'est pas semblable à une autre). Le [Tableau 6.1](#) décrit les opérateurs unaires.

[Tableau 6.1](#) : Les opérateurs unaires de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>

~	Inverse les bits d'un nombre (les 0 deviennent 1 et vice-versa)	~4 renvoie commande valeur -5.
	Inverse la valeur positive ou négative d'un nombre	-(-4) donne 4, et -4 donne -4.
+	Renvoie simplement la valeur en entrée	+4 donne 4.

Arithmétique

Les ordinateurs sont capables de réaliser des calculs mathématiques complexes, mais ceux-ci sont souvent basés sur des opérations plus simples, comme l'addition. Python dispose de bibliothèques spécialisées dans les calculs de haut vol, mais il offre comme il se doit des opérateurs beaucoup plus simples. Ceux-ci sont décrits dans le [Tableau 6.2](#).

Tableau 6.2 : Les opérateurs arithmétiques de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>
+	Ajoute deux valeurs	$5 + 2 = 7$
-	Soustrait l'opérande droit de l'opérande gauche	$5 - 2 = 3$
*	Multiplie deux valeurs	$5 * 2 = 10$
/	Divise l'opérande gauche par l'opérande droit	$5 / 2 = 2.5$
%	Divise l'opérande gauche par l'opérande droit et retourne le	$5 \% 2 = 1$

reste
(opérateur
modulo)

* *	Calcule la valeur exponentielle de l'opérande gauche par l'opérande droit	$5 * * 2 = 25$
//	Divise l'opérande gauche par l'opérande droit et retourne la partie entière	$5 // 2 = 2$

Relationnel

Les opérateurs relationnels comparent une valeur avec une autre et vous disent si la relation fournie est vraie. Par exemple, 1 est plus petit que 2, mais 1 n'est jamais plus grand que 2. La valeur vraie des relations

est souvent utilisée pour prendre des décisions en s’assurant que les conditions sont réunies pour effectuer une certaine tâche. Le [Tableau 6.3](#) décrit les opérateurs relationnels.

Tableau 6.3 : Les opérateurs relationnels de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>
<code>==</code>	Détermine si deux valeurs sont égales. Notez que cet opérateur contient deux signes d’égalité. L’oubli d’un de ces signes est une erreur courante, qui se traduit par une simple affectation.	<code>1 == 2</code> renvoie <code>False</code> (faux).

!=	Détermine si deux valeurs sont différentes. L'emploi par inadvertance d'un opérateur tel que <> provoque une erreur dans les versions actuelles de Python.	1 != 2 renvoie True (vrai).
>	Vérifie que l'opérande gauche est plus grand que l'opérande droit	1 > 2 renvoie False (faux).
<	Vérifie que l'opérande gauche est plus petit	1 < 2 renvoie True (vrai).

	que l'opérande droit	
$\geq$	Vérifie que l'opérande gauche est supérieur ou égal à l'opérande droit	$1 \geq 2$ renvoie False (faux).
$\leq$	Vérifie que l'opérande gauche est inférieur ou égal à l'opérande droit	$1 \leq 2$ renvoie True (vrai).

Logique

Les opérateurs logiques combinent la valeur vraie ou fausse de variables ou d'expressions pour pouvoir déterminer la valeur vraie résultante. Ils servent à créer des expressions booléennes qui vous aident à savoir si une certaine tâche doit ou non être réalisée. Ces opérateurs sont décrits dans le [Tableau 6.4](#).

Tableau 6.4 : Les opérateurs logiques de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>
and (et)	Détermine si les deux opérandes sont vrais.	True and True renvoie True (vrai) True and False renvoie False (faux) False and True renvoie False (faux) False and False renvoie False (faux)

or (ou)	Détermine si l'un des deux opérandes est vrai.	True and True renvoie True (vrai)

		True and False renvoie True (vrai)

		False and True renvoie True (vrai)

		False and False renvoie False (faux)

not (non)	Inverse la valeur logique de	not True renvoie False

l'opérande. (faux)

not False
renvoie
True
(vrai)

Opérateurs au niveau du bit

Les opérateurs au niveau du bit interagissent avec les bits (0 ou 1) individuels d'un nombre. Par exemple, le chiffre 6 s'écrit 0b0110 dans le système binaire. Avec ce type d'opérateur, une valeur de 0 est comptée comme fausse (False), et une valeur de 1 comme vraie (True). Le [Tableau 6.5](#) précise tout cela.

[Tableau 6.5](#) : Les opérateurs au niveau de bit de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>
& (et)	Détermine si les bits individuels des deux opérateurs sont vrais, et renvoie 1 dans ce cas	0b1100 & 0b0110 = 0b0100

ou 0 sinon.

(ou)	Détermine si les bits individuels des deux opérateurs sont ou non vrais, et renvoie 1 dans ce cas ou 0 sinon.	0b1100 & 0b0110 = 0b1110

^ (ou exclusif)	Détermine si un seul des bits individuels des deux opérateurs est vrai, et renvoie 1 dans ce cas. Si les deux bits sont vrais ou tous les deux faux, le	0b1100 & 0b0110 = 0b1010

résultat est
faux.

~ (complément de un)	Calcule le complément à un de la valeur.	~0b1100 = - 0b11 ~0b0110 = - 0b01
<< (décalage à gauche)	Décale vers la gauche les bits de l'opérande gauche selon le rang déterminé par l'opérande droit. Les bits entrants sont mis à zéro, et les bits sortants sont perdus.	0b00110 << 2 = 0b11001
>> (décalage à droite)	Décale vers la droite les bits de	0b00110 >> 2 = 0b00001

l'opérande
gauche
selon le rang
déterminé
par
l'opérande
droit. Les
bits entrants
sont mis à
zéro, et les
bits sortants
sont perdus.

Affectation

Les opérateurs d'affectation placent une valeur dans une variable. Nous avons déjà vu le cas le plus simple (`=`), mais Python offre bien d'autres opérateurs de ce type, afin notamment de réaliser une opération arithmétique en même temps que l'affectation. Ces opérateurs sont décrits dans le [Tableau 6.6](#). Dans ce cas particulier, la valeur initiale de la variable `MaVar` dans la colonne Exemple est 5.

[Tableau 6.6](#) : Les opérateurs d'affectation de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>

=	Affecte la valeur de l'opérande droit à l'opérande gauche.	MaVar = 2 affecte la valeur 2 à la variable MaVar.
+=	Ajoute la valeur de l'opérande droit à l'opérande gauche et place le résultat dans l'opérande gauche.	MaVar += 2 affecte la valeur 7 à MaVar.
-=	Soustrait la valeur de l'opérande droit de l'opérande gauche et place le résultat dans l'opérande	MaVar -= ; affecte la valeur 3 à MaVar.

gauche.

$\ast =$	Multiplie la valeur de l'opérande droit par celle de l'opérande gauche et place le résultat dans l'opérande gauche.	MaVar $\ast =$ 2 affecte la valeur 7 à MaVar.
$/ =$	Divise la valeur de l'opérande gauche par celle de l'opérande droit et place le résultat dans l'opérande gauche.	MaVar $/ =$ 2 affecte la valeur 2.5 à MaVar.
$\% =$	Divise la valeur de	MaVar $\% =$ 2 affecte

l'opérande gauche par celle de l'opérande droit et place le reste dans l'opérande gauche. la valeur 1 à MaVar.

* *=

Calcule la valeur exponentielle de l'opérande gauche par celle de l'opérande droit et place le résultat dans l'opérande gauche.

MaVar * *= 2 affecte la valeur 25 à MaVar.

//=

Divise la valeur de l'opérande gauche par celle de

MaVar//= : affecte la valeur 2 à MaVar.

l'opérande
droit et place
le résultat
entier dans
l'opérande
gauche.

Membre

Ces opérateurs détectent l'apparence d'une valeur dans une liste ou une séquence, et retournent la valeur vraie de cette apparence. Par exemple, vous entrez une valeur dont vous pensez qu'elle devrait se trouver dans une base de données, et la routine de recherche vous indique si cette valeur existe ou non dans la base. Les deux opérateurs disponibles sont décrits dans le [Tableau 6.7](#).

Tableau 6.7 : Les opérateurs « membres de » de Python.

Opérateur	Description	Exemple

in	Détermine si la valeur de l'opérande gauche apparaît dans la	« Bonjour » in « Bonjour et au revoir » est vrai (True)

séquence
fournie par
l'opérande
droit.

not in	Détermine si la valeur de l'opérande gauche manque dans la séquence fournie par l'opérande droit.	« Bonjour » not in « Bonjour et au revoir » est faux (False)

Identité

Ces opérateurs déterminent si une valeur ou une expression est d'un certain type ou classe. Ils permettent de s'assurer qu'une information est bien du type que vous pensez être. Ceci peut vous aider à éviter des erreurs, ou à déterminer quel genre de traitement appliquer à une valeur. Le [Tableau 6.8](#) décrit les opérateurs d'identité.

[Tableau 6.8](#) : Les opérateurs d'identité de Python.

<i>Opérateur</i>	<i>Description</i>	<i>Exemple</i>
is	Renvoie vrai ou faux selon que les deux opérandes sont ou non de même type	type(2) is int renvoie vrai (True)
is not	Renvoie vrai ou faux selon que les deux opérandes ne sont pas ou sont de même type	type(2) is not int renvoie faux (False)

Comprendre l'ordre de priorité des opérateurs

Lorsque vous créez des instructions simples ne contenant qu'un seul opérateur, la structure de cet opérateur détermine directement la sortie. Par contre, lorsque vous commencez à travailler avec de

multiples opérateurs, il devient nécessaire de savoir lequel est évalué en premier. Par exemple, l'expression $1 + 2 * 3$ peut renvoyer la valeur 7 (si la multiplication est effectuée en premier) ou 9 (si l'addition est effectuée en premier). L'ordre de priorité des opérateurs vous indique que la bonne réponse est 7, à moins d'utiliser des parenthèses pour modifier l'ordre par défaut. Ainsi, $(1 + 2) * 3$ donnera comme résultat 9, car les parenthèses ont un ordre de priorité plus élevé que celui de la multiplication. Le [Tableau 6.9](#) décrit l'ordre de priorité des opérateurs sous Python.

Tableau 6.9 : Ordre de priorité des opérateurs de Python.

<i>Opérateur</i>	<i>Description</i>
$()$	Les parenthèses regroupent des expressions afin de modifier l'ordre des priorités des opérateurs qu'elles contiennent, afin par exemple d'évaluer une addition avant une multiplication.
$**$	Exponentiation (opérande gauche

puissance opérande droit).

~ + -	Les opérateurs unaires interagissent avec une seule variable ou expression.
* / % //	Multiplication, division, modulo et reste de la division.
>> <<	Décalage droit et gauche des bits.
&	Opérateur logique ET au niveau des bits.
^	Opérateur logique OU au niveau des bits ou standard.
<= < > >=	Opérateurs de comparaison.
= %= /= //= -= += *= * *=	Opérateurs d'affectation.

is, is not	Opérateurs d'identité.

in, not in	Opérateurs de membre.

Not or and	Opérateurs logiques.

Créer et utiliser des fonctions

Afin de gérer correctement les informations, vous avez besoin d'organiser les outils utilisés pour effectuer les tâches demandées. Chacune des lignes de code que vous écrivez réalise une certaine tâche, et vous combinez ces lignes de code afin d'atteindre le résultat escompté. Parfois, vous devez répéter des instructions avec différentes données, ce qui risque au bout du compte de rendre votre code si long qu'il devient presque impossible de déterminer ce que fait chaque partie. Les fonctions servent précisément à organiser les choses pour que votre code reste propre et rangé à la perfection. De plus, les fonctions permettent de réutiliser les instructions qu'elles contiennent autant de fois qu'il est nécessaire avec des données différentes. Ces notions sont développées dans les sections qui suivent, et vous allez y créer vos premières applications sérieuses.

Voir les fonctions comme des boîtes de rangement pour le code

Vous vous dirigez vers votre placard, vous ouvrez la porte, et tout dégringole. C'est une véritable avalanche qui manque de vous emporter. La boule de bowling qui se trouvait sur l'étagère du haut aurait pu provoquer de sérieux dommages ! Cependant, vous disposez d'une batterie de boîtes de rangement, et vous avez tôt fait d'y ranger les objets de l'armoire dans un ordre précis. Les chaussures vont dans une boîte, les jeux dans une autre, les anciennes photos dans une troisième, et ainsi de suite. Quand vous avez terminé, vous pouvez ensuite retrouver tout ce dont vous avez besoin rapidement et sans prendre des risques inutiles. Les fonctions jouent exactement le même rôle que ces boîtes : elles permettent de regrouper proprement des éléments de code, et d'en faciliter ensuite la lecture, la compréhension ainsi que l'exécution.



Pour éviter les problèmes, faites en sorte que chacune de vos fonctions réponde à un seul objectif, comme cela doit être pour les boîtes de rangement.

Comprendre la réutilisabilité du code

Revenons à notre armoire. Vous y rangez par exemple vos vêtements. Quand vous en avez besoin, vous les sortez et vous les mettez. Quand ils sont sales, vous les lavez, puis vous les remettez en place lorsqu'ils sont secs. Bref, vos vêtements sont *réutilisables*. Et il en va de même pour les fonctions. Personne n'a envie de répéter encore et encore la même tâche. Cela deviendrait vite fastidieux et ennuyeux. Lorsque vous créez une fonction, vous définissez quelque chose

comme une « boîte » de code que vous pouvez ensuite utiliser chaque fois que vous devez répéter la même tâche. Il suffit pour cela d'indiquer qu'elle doit faire appel à votre fonction, et la manière dont cet appel doit être réalisé. L'ordinateur exécutera fidèlement chaque instruction de la fonction lors de ces appels.



Le code qui a besoin des services d'une fonction est souvent décrit comme l'*appelant*. L'appelant doit fournir des informations à la fonction (qui agit comme une sorte de boîte noire), et celle-ci retourne également des informations à l'appelant.

Vous pouvez créer des fonctions pour faire tout ce que vous voulez. La réutilisabilité du code est pratiquement toujours une nécessité, notamment pour :

- ✓ Réduire le temps de développement
- ✓ Réduire les erreurs de programmation
- ✓ Améliorer la fiabilité des applications
- ✓ Permettre à d'autres personnes de profiter du travail d'un programmeur
- ✓ Rendre le code plus facile à comprendre
- ✓ Améliorer l'efficacité des applications

En travaillant avec les exemples de ce livre, vous comprendrez mieux pourquoi la réutilisabilité du code rendra votre vie de développeur plus facile et plus agréable. Sinon, vous risqueriez d'en être réduit à classer manuellement des hordes de 0 et de 1 dans le grand placard de l'ordinateur.

Définir une fonction

Créer une fonction ne demande pas beaucoup de travail. Python rend les choses rapides et faciles. En voici la preuve :

1. **Ouvrez une fenêtre Python en mode Shell.**

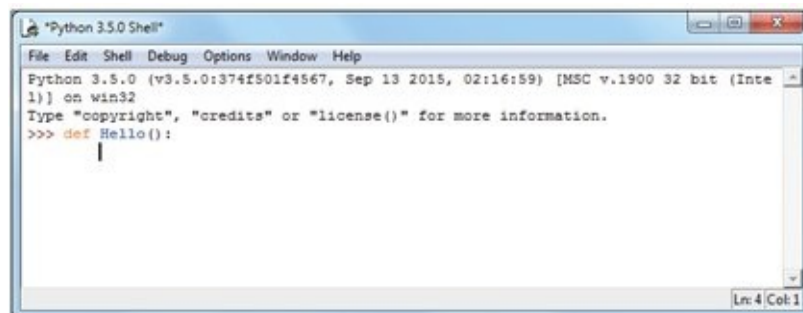
L'indicatif de Python apparaît.

2. **Tapez `def Hello()` : et appuyez sur Entrée.**

Cette étape demande à Python de définir (*def*) une fonction appelée Hello. Les parenthèses sont importantes, car elles servent à définir les informations dont la fonction a besoin (même si elles restent vides dans ce cas). Les deux-points qui suivent le nom de la fonction indiquent à Python que vous avez terminé la définition de la manière dont il est possible d'accéder à la fonction. Remarquez que le point d'insertion est maintenant indenté, comme sur la [Figure 6.1](#). Cette indentation est là pour vous rappeler que vous devez maintenant expliquer à la fonction ce qu'elle a à faire.

Figure 6.1 :

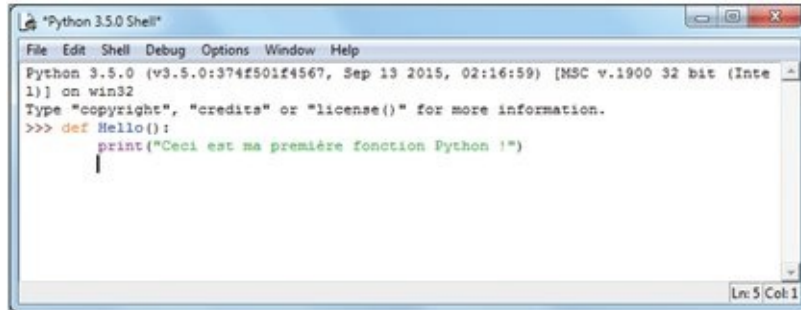
Définir le nom d'une fonction.



3. **Tapez `print(« Ceci est ma première fonction Python ! »)` et appuyez sur Entrée.**

Vous devriez remarquer deux choses. Comme l'illustre la [Figure 6.2](#), le point d'insertion reste indenté, car IDLE attend la prochaine étape de la fonction. Ensuite, Python n'a pas exécuté la commande `print ()`, puisque celle-ci fait partie de la fonction en cours de définition.

Figure 6.2 : IDLE attend votre prochaine instruction.

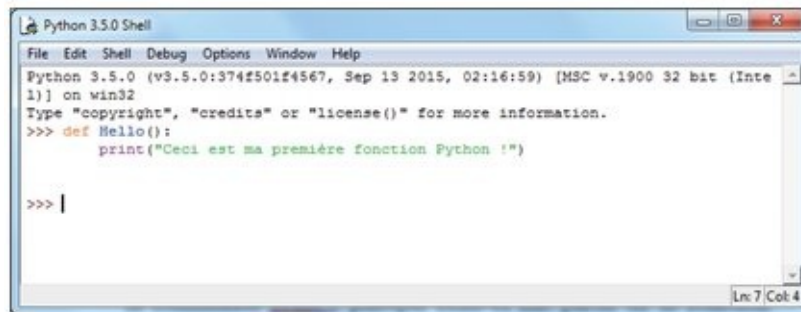


```
Python 3.5.0 (v3.5.0:374f50124567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def Hello():
    print("Ceci est ma première fonction Python !")
    |
```

4. Appuyez sur Entrée.

La fonction est maintenant complète. Le point d'insertion revient prendre place sur la gauche de la fenêtre, et vous retrouvez l'indicatif (`>>>`) de Python (voir la [Figure 6.3](#)).

Figure 6.3 : La fonction est terminée, et IDLE attend que vous fournissiez une autre instruction.



```
Python 3.5.0 (v3.5.0:374f50124567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def Hello():
    print("Ceci est ma première fonction Python !")
>>> |
```

Même si cet exemple est très simple, il illustre la trame générale de création d'une fonction sous Python. Vous commencez par le mot-clé `def`, vous définissez ensuite le nom de la fonction, vous fournissez entre parenthèses les éléments nécessaires à son exécution (aucun dans ce cas), et vous spécifiez ensuite les étapes à accomplir. Une fonction se termine lorsqu'une ligne supplémentaire est ajoutée (vous appuyez deux fois sur Entrée).



Travailler avec les fonctions s'effectue de la même manière en mode Shell et en mode Édition, si ce n'est que, dans le second cas, vous devez sauvegarder le contenu de la fenêtre sur le disque. Vous retrouverez cet exemple sous le nom `FirstFunction.py` dans le sous-

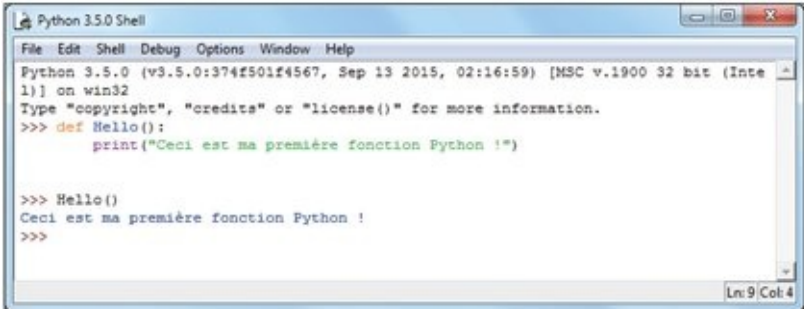
dossier \BP4D\Chapitre06 des sources téléchargeables. Pour l'ouvrir dans la fenêtre d'IDLE, revoyez si nécessaire les explications données dans la section « Utiliser la fenêtre d'édition » du Chapitre 4.

Accéder aux fonctions

Une fois que vous avez défini une fonction, vous allez bien entendu vouloir l'utiliser, et donc y accéder. Dans la précédente section, vous avez créé une fonction appelée `Hello()`. Pour y faire appel, il vous suffit de taper **Hello()** et d'appuyer sur Entrée. La [Figure 6.4](#) illustre le résultat de l'exécution de cette fonction.

Figure 6.4 :

Chaque fois que vous tapez le nom d'une fonction, elle retourne le résultat pour lequel elle est programmée.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def Hello():
    print("Ceci est ma première fonction Python !")
>>> Hello()
Ceci est ma première fonction Python !
>>>
```

Toute fonction que vous pouvez être amené à créer s'utilise de la même manière : vous tapez son nom, puis l'entrée à lui passer entre parenthèses, et vous appuyez sur Entrée. Dans la suite de ce chapitre, nous allons voir des exemples dans lesquels vous devrez passer certaines valeurs à des fonctions.

Transmettre des informations aux fonctions

L'exemple de la section précédente est sympathique, puisque vous n'avez rien à spécifier de particulier

pour appeler la fonction. Mais, bien entendu, il est extrêmement limité puisqu'il ne fait que bégayer toujours la même chose. Les fonctions ont évidemment une tout autre étendue que vous ne pouvez révéler qu'en leur passant des arguments.

Comprendre les arguments

Le mot *argument* ne veut pas dire que vous deviez engager le débat avec la fonction. Il signifie que vous devez fournir des informations à la fonction pour que celle-ci puisse les traiter. Même si ce nom, argument, n'est pas nécessairement idéal, ce concept est relativement évident. Un argument rend possible la transmission de données à la fonction afin qu'elle puisse réaliser certaines tâches avec elles. Ceci augmente donc la souplesse et la puissance de vos fonctions.

En l'état, la fonction `Hello()` est totalement rigide, puisqu'elle ne fait qu'afficher encore et toujours la même chaîne de caractères. Ajouter un argument à cette fonction la rendra plus flexible, puisqu'elle pourra alors afficher un message quelconque. Pour cela, ouvrez une nouvelle fenêtre de Python en mode Shell (ou ouvrez le fichier `Arguments01.py` dans le dossier des exemples de ce chapitre). Définissez ainsi une variante appelée `Hello2 ()` :

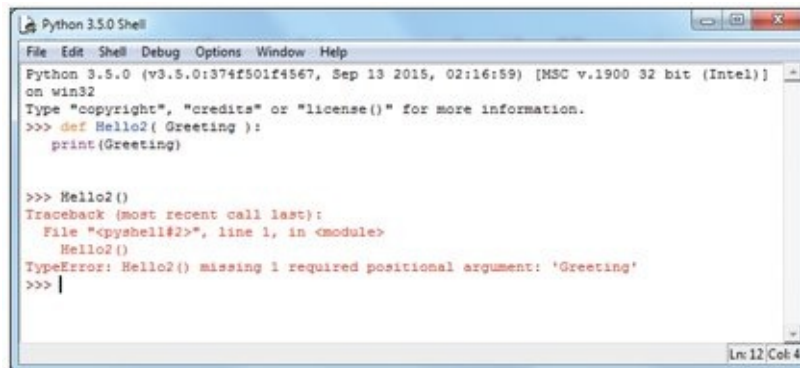
```
def Hello2( Greeting ):  
    print(Greeting)
```

Remarquez que les parenthèses ne sont plus vides. Elles contiennent un mot, `Greeting`, qui est l'argument de la fonction. Il s'agit en fait d'une variable que vous pouvez passer à la commande `print ()` de manière à afficher un message à l'écran.

Transmettre les arguments

Vous avez maintenant une nouvelle fonction appelée `Hello2()`. Elle nécessite cette fois le passage d'un argument. Pour le vérifier, tapez simplement **Hello2()** et appuyez sur Entrée. Vous allez voir s'afficher un message d'erreur qui vous rappelle que la fonction réclame un argument (voir la [Figure 6.5](#)).

Figure 6.5 : Si vous ne fournissez pas d'argument, un message d'erreur s'affiche.



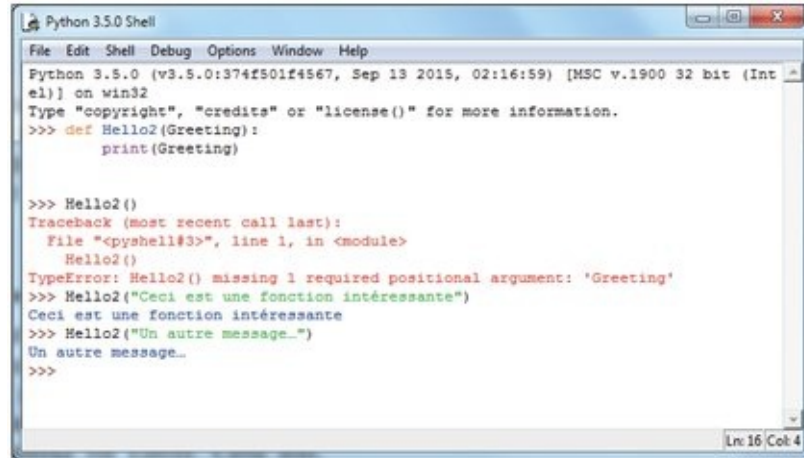
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> def Hello2( Greeting ):
    print(Greeting)

>>> Hello2()
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    Hello2()
TypeError: Hello2() missing 1 required positional argument: 'Greeting'
>>> |
```

Les termes du message d'erreur sont un peu compliqués, mais remarquez surtout que Python vous donne le nom de l'argument en prime. En créant la fonction de cette manière, vous avez forcé le passage d'un argument. Tapez maintenant par exemple **Hello2(« Ceci est une fonction intéressante »)** et appuyez sur Entrée. Cette fois, vous voyez bien le texte spécifié. Essayez ensuite avec **Hello2(« Un autre message... »)**. Les résultats, illustrés sur la [Figure 6.6](#), démontrent bien que `Hello2()` constitue un progrès sensible par rapport à la version `Hello()`.

Mais la souplesse de la fonction `Hello2()` va au-delà de l'affichage de chaînes de caractères. Tapez par exemple **Hello2(1234)** et appuyez sur Entrée. Vous obtenez comme réponse 1234. Plus fort encore : tapez **Hello2(5+4)** et appuyez sur Entrée. La fonction vous donne alors le résultat de l'opération, 9.

Figure 6.6 : La fonction Hello2() vous permet d'afficher un message quelconque.



Transmettre des arguments par nom

Progressivement, vos fonctions vont devenir de plus en plus complexes, de même que les méthodes permettant de les utiliser. Vous aurez donc besoin de davantage de contrôle pour préciser comment appeler une fonction et comment lui transmettre des arguments. Jusqu'ici, vous n'avez utilisé que des *arguments par position*, c'est-à-dire dans l'ordre où ils apparaissent dans la définition de la fonction. Cependant, Python vous permet aussi de passer des *arguments par nom*, ou encore à l'aide de mots-clés. Pour cela, vous fournissez le nom de l'argument, suivi d'un signe égal, et de la valeur à transmettre.

Pour voir cela en action, ouvrez le fichier d'exemple Arguments02.py ou ouvrez une nouvelle fenêtre en mode Shell et tapez la fonction suivante :

```
def AddIt(Value1, Value2):
    print(Value1, " + ", Value2, " = ", (Value1 + Value2))
```

Notez que la fonction `print ()` reçoit une liste d'arguments séparés par une virgule. De plus, ces arguments peuvent être de différents types, ce qui est une autre des facilités offertes par Python.

Pour tester cette fonction, commencez par taper

AddIt(2, 3) et appuyez sur Entrée. C'est un passage d'arguments par position. La réponse $2 + 3 = 5$ va s'afficher. Tapez ensuite **AddIt(Value2 = 3, Value1 = 2)** et appuyez sur Entrée. Cette fois, il s'agit d'un passage d'arguments par nom. Vous recevez encore la même réponse, même si les deux arguments ont été inversés.

Donner aux arguments une valeur par défaut

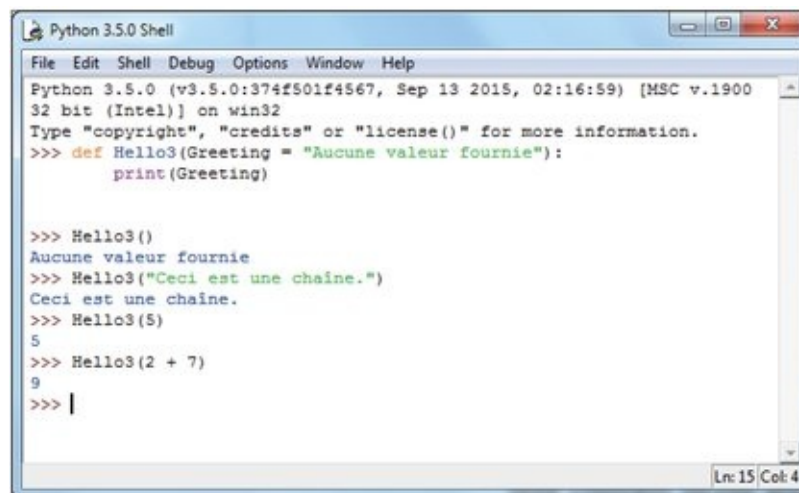
Quelle que soit la méthode de passage utilisée (par position ou par nom), les fonctions précédentes nécessitent la fourniture d'une ou plusieurs valeurs. Cependant, une fonction peut parfois utiliser une valeur par défaut lorsqu'une valeur simple ou courante est disponible. Ceci permet de simplifier l'utilisation des fonctions et de limiter les risques d'erreurs. Pour définir une valeur par défaut, il suffit de faire suivre le nom de l'argument d'un signe égal et de ladite valeur.

Pour voir cela en action, ouvrez le fichier d'exemple Arguments03.py ou ouvrez une nouvelle fenêtre en mode Shell et tapez la fonction suivante :

```
def Hello3(Greeting = "Aucune valeur fournie"):  
    print(Greeting)
```

Figure 6.7 :

Fournir des arguments par défaut lorsque c'est possible rend votre fonction plus facile à utiliser.



```
Python 3.5.0 Shell  
File Edit Shell Debug Options Window Help  
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900  
32 bit (Intel)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> def Hello3(Greeting = "Aucune valeur fournie"):  
    print(Greeting)  
  
>>> Hello3()  
Aucune valeur fournie  
>>> Hello3("Ceci est une chaîne.")  
Ceci est une chaîne.  
>>> Hello3(5)  
5  
>>> Hello3(2 + 7)  
9  
>>> |
```

C'est une autre version des fonctions `Hello ()` et `Hello2()`, mais qui compense cette fois automatiquement les oublis qui pourraient se produire. Si quelqu'un essaie d'exécuter `Hello3()` sans passer d'argument, cette fonction ne provoque pas d'erreur. Au lieu de cela, elle affiche le message par défaut. Pour le vérifier, tapez **Hello3()** et appuyez sur Entrée. Tapez ensuite quelque chose comme **Hello3(« Ceci est une chaîne. »)** suivi d'un appui sur Entrée pour vérifier le comportement normal de la fonction. Vous pouvez également essayer les variations **Hello3(5)** ou encore **Hello3(2+7)** pour constater qu'elle affiche bien la valeur ou le résultat du calcul. La [Figure 6.7](#) illustre ces différents tests.

Créer des fonctions avec un nombre variable d'arguments

Dans la plupart des cas, vous connaissez exactement le nombre d'arguments dont votre fonction a besoin. En fait, c'est l'objectif que vous devriez vous fixer, car les fonctions ayant un nombre déterminé d'arguments sont plus faciles à corriger plus tard. Cependant, cela n'est quand même pas toujours possible. Supposons par exemple que vous deviez créer une application qui fonctionne en mode Ligne de commandes. Dans ce cas, l'utilisateur pourrait fort bien ne fournir aucun argument, le nombre maximum des arguments possibles, ou encore toute combinaison intermédiaire d'arguments.

Heureusement, Python fournit une technique permettant de transmettre un nombre variable d'arguments à une fonction. Il vous suffit pour cela de placer un astérisque (`*`) devant le nom de l'argument, comme dans `*VarArgs`. La méthode habituelle consiste à spécifier un second argument qui contient le nombre

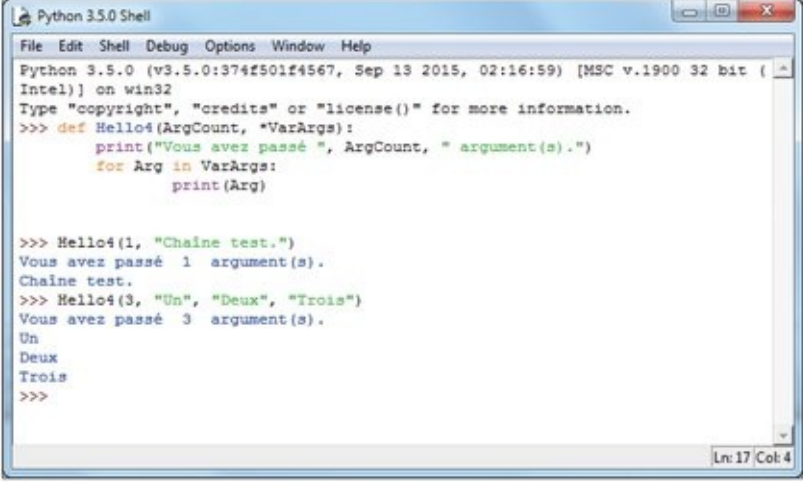
d'arguments passés à la fonction. Le fichier `VarArgs.py` du dossier \ BP4\Chapitre06 contient une fonction de ce type. Elle se définit ainsi :

```
def Hello4(ArgCount, *VarArgs):  
    print("Vous avez passé ", ArgCount, " argument(s).")  
    for Arg in VarArgs:  
        print(Arg)
```

Ne paniquez pas si vous avez du mal à suivre le fil... Cet exemple utilise quelque chose que l'on appelle une boucle `for`. Nous retrouverons cette structure dans le Chapitre 8. Pour l'instant, il vous suffit de savoir que cette fonction récupère un argument à la fois depuis `VarArgs`, en plaçant chaque élément dans la variable `Arg`, et en affichant tout simplement la valeur de ces arguments via la commande `print ()`. L'intérêt est ici de voir comment définir et utiliser un nombre variable d'arguments.

Une fois cette fonction saisie (ou le fichier correspondant ouvert), tapez par exemple **Hello4(1, « Chaîne test. »)** et appuyez sur Entrée. Vous devriez voir s'afficher le nombre d'arguments ainsi que la chaîne qui suit. Essayez maintenant avec **Hello4(3, « Un », « Deux », « Trois »)**. Comme l'illustre la [Figure 6.8](#), la fonction gère le nombre variable d'arguments sans aucun problème.

Figure 6.8 : Les fonctions avec un nombre variable d'arguments peuvent rendre vos applications plus souples.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def Hello4(ArgCount, *VarArgs):
    print("Vous avez passé ", ArgCount, " argument(s).")
    for Arg in VarArgs:
        print(Arg)

>>> Hello4(1, "Chaine test.")
Vous avez passé 1 argument(s).
Chaine test.
>>> Hello4(3, "Un", "Deux", "Trois")
Vous avez passé 3 argument(s).
Un
Deux
Trois
>>>
```

Renvoyer des informations depuis une fonction

Les fonctions peuvent afficher directement des résultats, ou bien encore retourner des données à l'application de manière à ce que la procédure appelante puisse les traiter d'une façon ou d'une autre. En fait, les deux sont possibles simultanément, mais c'est une situation moins courante.

La manière dont ces fonctions font leur travail dépend du type de tâche qu'elles sont supposées effectuer. Par exemple, une fonction chargée de réaliser des calculs mathématiques est plus susceptible de renvoyer des données à la procédure appelante qu'une autre.

Pour renvoyer des données, une fonction doit contenir le mot-clé `return`, suivi des informations à retourner. Il n'y a pas de limites quant aux données retournées à la procédure appelante. Voici certains types courants dans de tels cas :

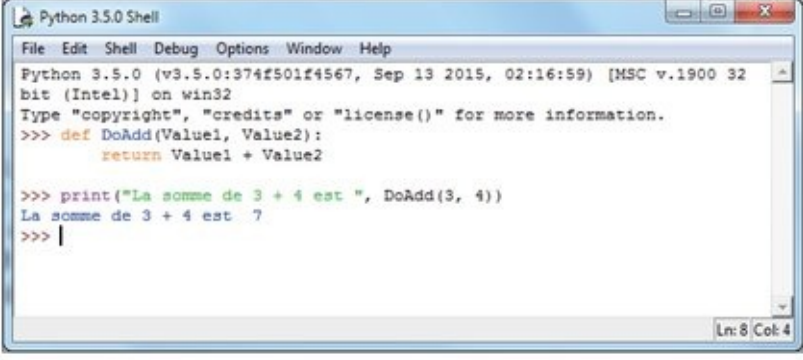
- **Valeurs** : Toute valeur est acceptable. Il peut s'agir de nombres, de chaînes de caractères, ou encore de valeurs booléennes.
- **Variables** : Une variable quelconque se comporte comme une valeur directe. L'appelant reçoit ce que contient cette variable.
- **Expressions** : De nombreux développeurs utilisent des expressions en tant que raccourcis. Par exemple, vous pouvez parfaitement renvoyer $A + B$ plutôt que d'effectuer l'opération, placer le résultat dans une variable, puis retourner cette dernière à la procédure appelante. Utiliser une expression est plus rapide et accomplit le même travail.
- **Résultats d'autres fonctions** : Vous pouvez aussi renvoyer le résultat produit par d'autres fonctions.

Il est temps de voir d'un peu plus près comment les choses se déroulent. Ouvrez une fenêtre Python en mode Shell, ou ouvrez le fichier `ReturnValue.py`, et saisissez ce code :

```
def DoAdd(Value1, Value2):  
    return Value1 + Value2
```

Cette fonction prend deux valeurs en entrée, et retourne leur somme. Certes, cette tâche pourrait être effectuée sans faire appel à une fonction, mais c'est aussi le point de départ de nombreuses fonctions. Pour la tester, tapez **`print(« La somme de 3 + 4 est « , DoAdd(3, 4))`** et appuyez sur Entrée. La sortie qui en résulte est illustrée sur la [Figure 6.9](#).

Figure 6.9 : Les valeurs de retour peuvent rendre les fonctions encore plus utiles.



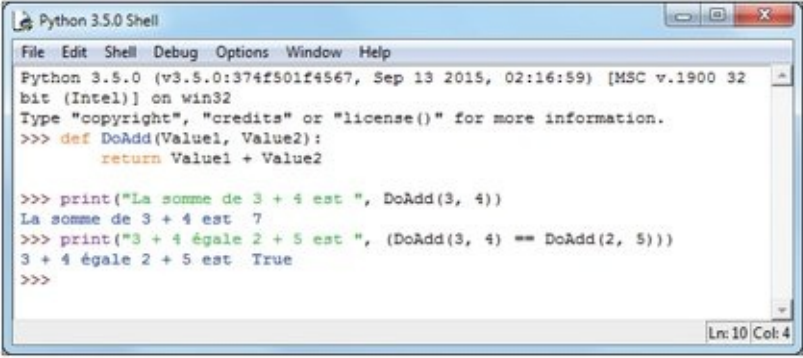
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def DoAdd(Value1, Value2):
>>>     return Value1 + Value2
>>>
>>> print("La somme de 3 + 4 est ", DoAdd(3, 4))
La somme de 3 + 4 est 7
>>> |
```

Comparer les sorties de fonctions

Vous pouvez utiliser des fonctions qui retournent des données de multiples manières. Par exemple, la section précédente de ce chapitre vous montre l'emploi d'une fonction pour fournir une entrée dans une autre fonction. L'une des applications possibles de ce mécanisme est la comparaison entre données, afin notamment de créer des expressions définissant une valeur logique.

Pour mieux voir comment tout cela s'articule, vous allez utiliser la fonction `DoAdd()` de la section précédente. Tapez **`print(« 3 + 4 égale 2 + 5 est « , (DoAdd(3, 4) == DoAdd(2, 5))`** et appuyez sur Entrée. Vous devriez voir s'afficher la valeur `True` (vrai), comme l'illustre la [Figure 6.10](#). La notion à retenir ici est que les fonctions n'ont pas à agir d'une seule façon, ou que vous n'avez pas à les voir d'une seule façon. Ces fonctions peuvent rendre votre code plus souple et plus versatile.

Figure 6.10 :
Utiliser des
fonctions pour
effectuer toutes
sortes de tâches.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def DoAdd(Value1, Value2):
    return Value1 + Value2

>>> print("La somme de 3 + 4 est ", DoAdd(3, 4))
La somme de 3 + 4 est 7
>>> print("3 + 4 égale 2 + 5 est ", (DoAdd(3, 4) == DoAdd(2, 5)))
3 + 4 égale 2 + 5 est True
>>>
```

Interagir avec l'utilisateur

Très peu d'applications vivent uniquement dans leur propre monde, autrement dit sans interaction avec l'utilisateur. En fait, c'est cette interaction qui est essentielle, notamment du fait que les ordinateurs (ou les smartphones et les tablettes qui en sont les descendants) sont conçus pour servir les besoins des utilisateurs. Pour cela, une application doit fournir à l'utilisateur des moyens lui permettant de communiquer avec elle. En fait, la technique la plus courante est relativement facile à mettre en œuvre. Vous avez simplement besoin de faire appel à la fonction `input ()` pour obtenir ce résultat.



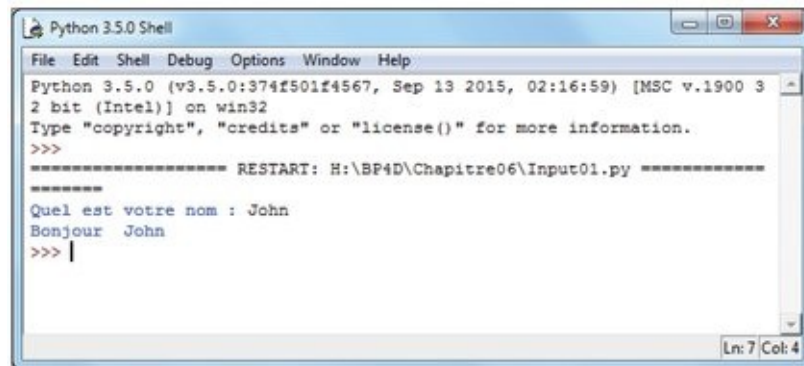
La fonction `input ()` retourne toujours une chaîne de caractères, même si l'utilisateur tape un nombre. Ceci signifie que, dans le cas où vous attendez effectivement une valeur numérique, vous devrez convertir la chaîne renvoyée pour obtenir le résultat attendu. Cette fonction vous permet également de définir un message demandant une réponse à l'utilisateur, ce qui permet de préciser ce que l'application attend de lui.

Le fichier `Input01.py` contient un exemple d'utilisation de la fonction `input ()`. En voici le code :

```
Name = input("Quel est votre nom : ")
print("Bonjour ", Name)
```

Dans ce cas, la fonction demande un nom d'utilisateur. Une fois que l'utilisateur a saisi cette information et appuyé sur Entrée, le code affiche un message de bienvenue. Essayez de l'exécuter en ouvrant Python en mode Shell ou depuis IDLE. La [Figure 6.11](#) illustre ce comportement.

Figure 6.11 : Vous donnez un nom d'utilisateur, et l'application vous salue.

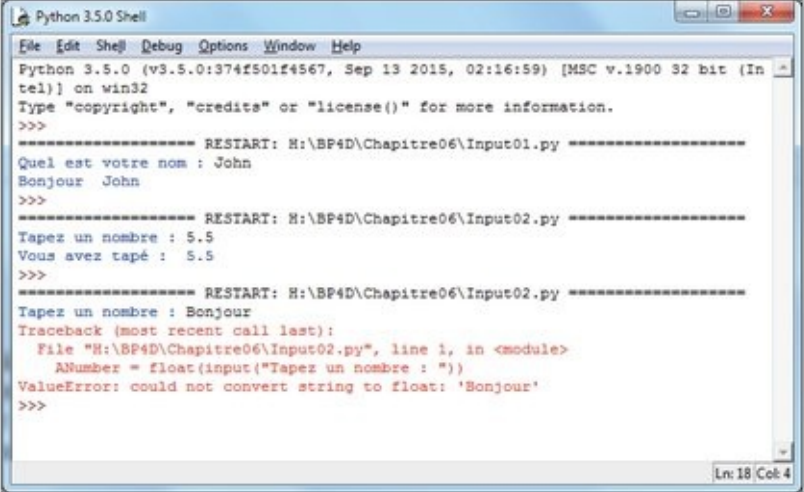


Vous pouvez utiliser `input ()` pour obtenir d'autres types de données. Tout ce dont vous avez ensuite besoin, c'est de la bonne fonction de conversion. Le code du fichier `Input02.py` propose un exemple de cette technique afin de demander une entrée numérique. Voici ce code :

```
ANumber = float(input("Tapez un nombre : "))
print("Vous avez tapé : ", ANumber)
```

Quand vous exécutez cet exemple, l'application vous demande une entrée numérique. L'appel à la fonction `float ()` convertit la saisie en nombre. Il suffit ensuite d'appeler `print ()` pour afficher le résultat. La [Figure 6.12](#) illustre ce fonctionnement.

Figure 6.12 : Une conversion de donnée change le type de la saisie utilisateur en fonction de vos besoins, mais elle peut aussi provoquer des erreurs.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre06\Input01.py =====
Quel est votre nom : John
Bonjour John
>>>
===== RESTART: H:\BP4D\Chapitre06\Input02.py =====
Tapez un nombre : 5.5
Vous avez tapé : 5.5
>>>
===== RESTART: H:\BP4D\Chapitre06\Input02.py =====
Tapez un nombre : Bonjour
Traceback (most recent call last):
  File "H:\BP4D\Chapitre06\Input02.py", line 1, in <module>
    ANumber = float(input("Tapez un nombre : "))
ValueError: could not convert string to float: 'Bonjour'
>>>
```



Il est important de comprendre que la conversion de données n'est pas sans risques. Si vous essayez de taper autre chose qu'un nombre dans cet exemple, Python va renvoyer un message d'erreur, comme le montre la [Figure 6.12](#). Le Chapitre 9 vous aidera à comprendre comment détecter et réparer de telles erreurs avant qu'elles ne bloquent le système.

Chapitre 7

Prendre des décisions

Dans ce chapitre :

- ▶ Utiliser l'instruction `if` pour prendre une décision simple.
- ▶ Prendre des décisions plus complexes avec l'instruction `if...else`.
- ▶ Créer des niveaux de décision multiples en imbriquant les instructions.

La possibilité de prendre des décisions pour suivre un chemin ou bien un autre est quelque chose d'essentiel dans toute activité. Les mathématiques donnent à l'ordinateur la capacité d'obtenir des informations utiles. Et ces informations permettent ensuite d'effectuer telle ou telle action. Sans cela, un ordinateur ne servirait strictement à rien. Tout langage de programmation doit offrir des fonctionnalités pour pouvoir prendre des décisions d'une certaine manière. Ce chapitre explore les techniques que Python utilise dans ce domaine.



Réfléchissez au processus que vous suivez lorsque vous prenez des décisions. Vous obtenez la valeur de

quelque chose, vous la comparez avec une valeur attendue, et vous agissez en conséquence. Prenons un exemple : vous voyez un signal lumineux et vous voyez qu'il est rouge. Vous le comparez au signal que vous attendez, le vert. Vous constatez donc que le signal n'est pas au vert. Action : vous vous arrêtez au feu. La plupart des gens ne prennent pas le temps de réfléchir à ce genre de processus, tout simplement, car c'est devenu un automatisme. La plupart des décisions quotidiennes se prennent naturellement, mais les ordinateurs, eux, doivent effectuer à chaque fois les tâches suivantes :

1. **Obtenir la valeur réelle ou actuelle de quelque chose.**
2. **Comparer cette valeur réelle ou actuelle avec une valeur attendue.**
3. **Effectuer l'action voulue en fonction du résultat de la comparaison.**

Prendre des décisions simples avec l'instruction if

Dans Python, l'instruction `if` fournit la méthode la plus simple pour prendre des décisions. Si cette instruction constate que quelque chose est vrai, cela signifie que Python devrait exécuter les étapes qui suivent. Les prochaines sections vous expliquent comment utiliser l'instruction `if` pour prendre différentes décisions. Vous pourriez être surpris de découvrir tout ce que ces deux petites lettres peuvent faire pour vous.

Comprendre l'instruction if

Sans le savoir, vous utilisez couramment des

instructions `if` dans la vie de tous les jours. Vous vous dites par exemple : « Si c'est vendredi, je vais aller nager à la piscine. » L'instruction `if` de Python est moins bavarde, mais elle suit le même schéma. Supposons que vous créez une variable appelée `TestMe` et que vous y placez la valeur 6, comme ceci :

```
TestMe = 6
```

Vous demandez ensuite à l'ordinateur de comparer la valeur de `TestMe` à 6 :

```
if TestMe == 6:  
    print("TestMe est égal à 6 !")
```

Cette séquence débute par le mot `if` (si). Lorsque Python voit ce mot, il sait que vous voulez prendre une décision. À sa suite vient une condition. Une *condition* indique tout simplement le type de comparaison que vous voulez demander à Python d'effectuer. Dans ce cas, il s'agit de déterminer si `TestMe` contient bien la valeur 6.



Remarquez que la condition utilise l'opérateur relationnel d'égalité, `==`, et non l'opérateur `=`, qui définit une affectation. Oublier un des deux signes d'égalité est une erreur courante. Les opérateurs relationnels (et les autres) sont décrits dans le Chapitre 6.

La condition se termine toujours par un deux-points (`:`). Si vous l'oubliez, Python ne saura pas que la condition se termine, et il continuera à rechercher sa suite avant de prendre une décision. Une fois ce caractère entré, vous placez les tâches que Python devrait effectuer si la condition est vérifiée (vraie). Ici, vous demandez simplement à afficher un message qui confirme que `TestMe` est bien égal à 6.

Utiliser l'instruction `if` dans une application

Dans Python, il est possible d'utiliser l'instruction `if` de diverses manières. Cependant, vous avez immédiatement besoin d'en connaître trois :

- Utiliser une condition unique pour exécuter une seule instruction quand cette condition est vraie.
- Utiliser une condition unique pour exécuter plusieurs instructions quand cette condition est vraie.
- Combiner plusieurs instructions en une seule et exécuter une ou plusieurs instructions que la condition combinée est vraie.

Les sections qui suivent explorent ces trois possibilités et fournissent des exemples d'utilisation. Vous en retrouverez bien d'autres dans ce livre, car `if` est une méthode importante dès lors qu'il s'agit de prendre des décisions.

Travailler avec les opérateurs relationnels

Un opérateur *relationnel* détermine la manière dont une valeur placée sur le côté gauche de cet opérateur se compare avec celle qui est placée du côté droit de l'expression. Une fois le résultat déterminé, il produit une réponse `True` ou `False` qui reflète la valeur vraie de l'expression. Par exemple, `6 == 6` est vrai, tandis que `5 == 6` est faux. Le [Tableau 6.3](#) décrit les opérateurs relationnels. Les étapes qui suivent vous montrent comment créer et utiliser une instruction `if`. Vous le retrouvez dans le fichier téléchargeable `SimpleIf1.py`.

1. Ouvrez une fenêtre Python en mode Shell.

Vous voyez l'indicatif bien connu de Python.

2. **Tapez `TestMe = 6` et appuyez sur Entrée.**

Cette étape assigne la valeur 6 à la variable `TestMe`. Remarquez qu'elle utilise l'opérateur d'affectation (`=`), et non l'opérateur d'égalité (`==`).

3. **Tapez `if TestMe == 6 :` et appuyez sur Entrée.**

Cette étape crée une instruction `if` qui teste la valeur de `TestMe` en utilisant l'opérateur d'égalité. À ce stade, vous pouvez noter deux choses dans la fenêtre de Python :

- Le mot *if* est affiché dans une couleur différente de celle du reste de l'instruction.
- La ligne suivante est automatiquement indentée.

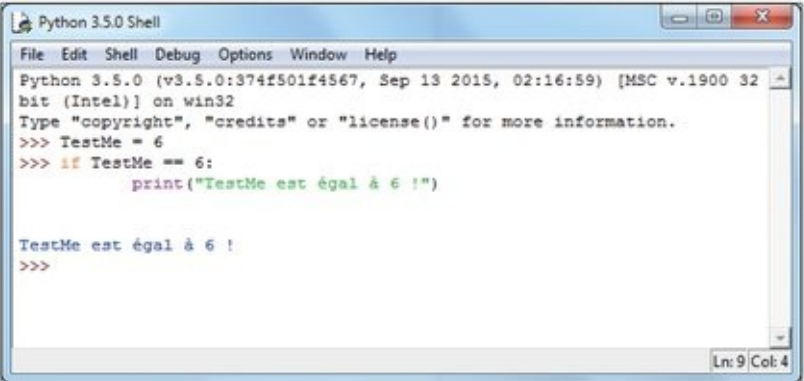
4. **Tapez `print(« TestMe est égal à 6 ! »)` et appuyez sur Entrée.**

Notez que Python n'exécute pas tout de suite l'instruction `if`. Il se contente d'indenter la ligne suivante. Le mot *print* apparaît dans une couleur particulière, car il s'agit d'un nom de fonction. La chaîne de caractères possède également son propre code de couleur. Ce mode d'affichage rend plus facile la lecture du code Python.

5. **Appuyez sur Entrée.**

Python annule l'indentation, et il exécute l'instruction `if` (voir la [Figure 7.1](#)). Là encore, la sortie possède sa propre couleur. Du fait que la variable `TestMe` contient bien la valeur 6, l'instruction `if` fonctionne comme prévu.

Figure 7.1 : Des instructions `if` simples peuvent aider vos applications à savoir ce qu'elles doivent faire dans certaines conditions.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32
bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> TestMe = 6
>>> if TestMe == 6:
    print("TestMe est égal à 6 !")

TestMe est égal à 6 !
>>>
```

Effectuer des tâches multiples

Après avoir pris une décision, vous avez très souvent non pas une, mais plusieurs tâches à accomplir. Python se base sur l'indentation pour déterminer le moment où il doit arrêter d'exécuter les tâches qui sont associées à l'instruction `if`. Tant que le texte est indenté, cela signifie qu'il fait partie de l'instruction `if`. Dès que cette indentation se termine, la ligne qui suit devient la première qui est extérieure à l'instruction `if`. Un *bloc de code* consiste en une instruction plus les tâches associées à celle-ci. Cette notion est extrêmement générale, mais, en l'occurrence, elle s'applique à une instruction `if` qui fait partie d'un bloc de code. Vous pouvez retrouver l'exemple qui suit dans le fichier téléchargeable `SimpleIf2.py`.

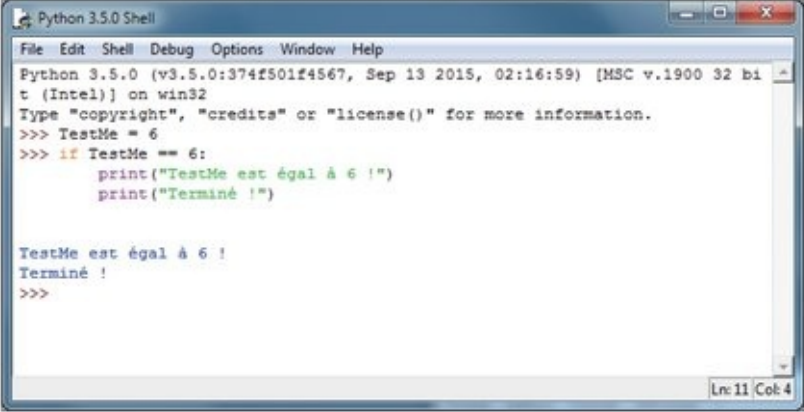
1. **Ouvrez une fenêtre Python en mode Shell.**
Vous voyez l'indicatif bien connu de Python.
2. **Tapez le code ci-dessous en appuyant sur Entrée à la fin de chaque ligne :**

```
TestMe = 6
if TestMe == 6:
    print("TestMe est égal à 6 !")
    print("Terminé !")
```

Notez que Python continue à indenter les lignes tant que vous continuez à saisir du code. Toutes les lignes que vous tapez ainsi font partie du bloc de code associé à l'instruction `if` courante.

3. **Appuyez sur Entrée.**
Python va exécuter tout le bloc de code. La sortie est illustrée sur la [Figure 7.2](#).

Figure 7.2 : Un bloc de code peut contenir de multiples lignes, une pour chaque tâche.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bi
t (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> TestMe = 6
>>> if TestMe == 6:
    print("TestMe est égal à 6 !")
    print("Terminé !")

TestMe est égal à 6 !
Terminé !
>>>
```

Effectuer des comparaisons multiples en utilisant des opérateurs logiques

Jusqu'ici, les exemples proposés n'utilisaient qu'une seule comparaison. Mais, dans la vraie vie, il est fréquent d'avoir à traiter une série de comparaisons pour répondre à divers besoins. Si vous faites par exemple cuire des sablés, si le minuteur est tombé à zéro et que les bords des gâteaux ont brunis, *alors* il est temps de les sortir du four.



Pour effectuer des comparaisons multiples, vous utilisez des opérateurs relationnels que vous combinez ensuite à l'aide d'opérateurs logiques (reportez-vous au [Tableau 6.4](#)). Un *opérateur logique* décrit la manière de combiner des conditions. Par exemple, vous pourriez dire que $x == 6$ et $y == 7$ sont deux conditions permettant si elles sont vérifiées d'effectuer une ou plusieurs tâches. Le mot-clé *et* (*and* dans le langage de Python) est un opérateur logique qui demande à ce que les conditions soient vraies pour être validé.

L'un des usages les plus courants des comparaisons multiples consiste à déterminer si une valeur appartient à un certain intervalle, autrement dit se trouve entre deux valeurs. En fait, il s'agit de quelque

chose d'important pour rendre vos applications à la fois plus sûres et mieux adaptées aux besoins des utilisateurs. Prenons un exemple. Il s'agit ici de créer un fichier vous permettant d'exécuter la même application de multiples fois. Vous en trouverez le code dans le fichier téléchargeable `SimpleIf3.py`.

1. **Ouvrez une fenêtre de fichier de Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Value = int(input("Tapez un nombre entre 1 et 10: "))  
  
if (Value > 0) and (Value <= 10):  
    print("Vous avez tapé : ", Value)
```

La première ligne demande à l'utilisateur de taper un nombre entre 1 ET 10. Vous n'avez alors aucune idée de ce que l'utilisateur a réellement saisi, ou même s'il s'agit d'une valeur quelconque. L'emploi de la fonction `int ()` signifie que l'utilisateur doit saisir un nombre entier (donc sans point décimal). Sinon, l'application va générer une *exception* (autrement dit, une indication d'erreur, sujet sur lequel nous reviendrons dans le Chapitre 9). Ce premier contrôle permet de vérifier si l'entrée possède le type demandé.

L'instruction `if` contient deux conditions. La première vérifie que la valeur est strictement supérieure à 0 (et donc supérieure ou égale à 1, s'agissant d'un entier). La seconde fait la même de l'autre côté de l'intervalle pour s'assurer que la valeur est inférieure ou égale à 10. Ce n'est que lorsque ces deux conditions sont réunies que `if` peut autoriser l'instruction suivante à afficher la valeur saisie par l'utilisateur.

3. **Choisissez la commande Run Module dans le menu Run.**

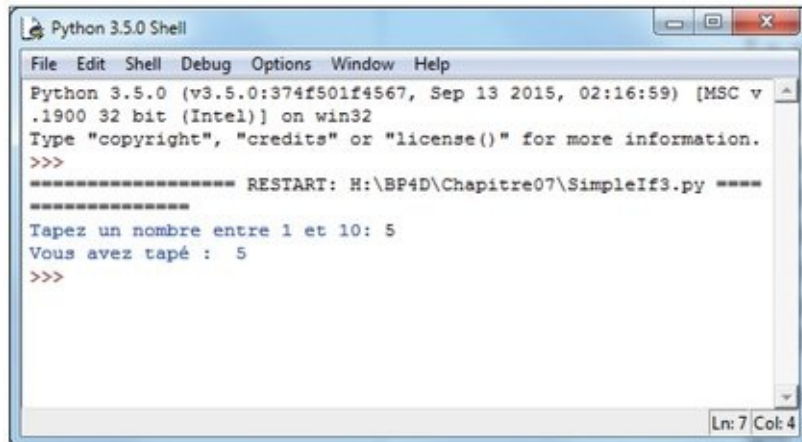
Une fenêtre Python en mode Shell va s'ouvrir. Un message vous demande de taper un nombre entre 1 et 10.

4. **Tapez 5 et appuyez sur Entrée.**

L'application détermine que cette valeur appartient à l'intervalle demandé, et elle affiche le message illustré sur la [Figure 7.3](#).

Figure 7.3 :

L'application vérifie que la valeur saisie appartient au bon intervalle, et elle affiche dans ce cas un message.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v
.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\SimpleIf3.py =====
Tapez un nombre entre 1 et 10: 5
Vous avez tapé : 5
>>>
```

5. **Reprenez les Étapes 3 et 4, mais tapez cette fois 22 au lieu de 5.**

L'application n'affiche rien, puisque le nombre n'est pas compris entre 1 et 10. Dans ce cas, la ou les instructions qui font partie du bloc `if` ne sont pas exécutées.

6. **Reprenez les Étapes 3 et 4, et tapez cette fois 5.5.**

Python va afficher le message d'erreur illustré sur la [Figure 7.4](#). Bien entendu, vous pensez que la valeur 5.5 est comprise entre 1 et 10, et que 5.5 est bien une valeur numérique. Mais Python de son côté, sait que 5 est un nombre entier, et 5.5 un nombre décimal, et par conséquent non entier.

7. **Répétez encore une fois les Étapes 3 et 4, en tapant cette fois le mot Bonjour.**

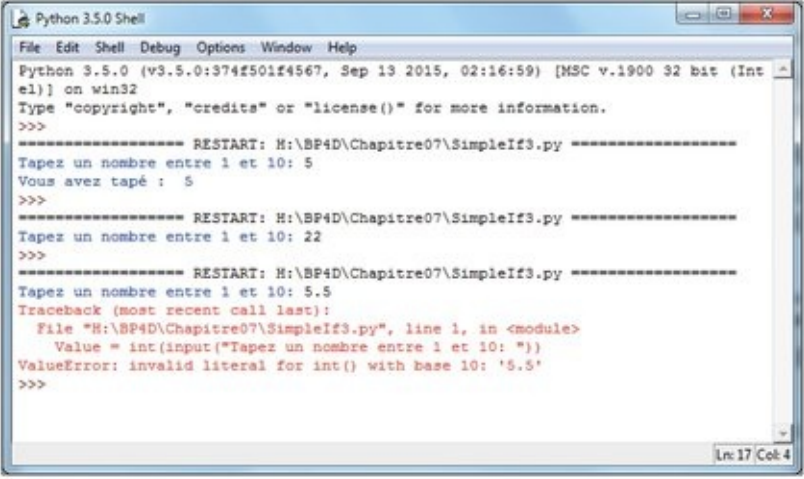
Python affiche le même message d'erreur que ci-dessus. Il ne veut même pas déterminer quel est le type erroné de l'entrée. Il sait seulement que ce type est incorrect, et donc inutilisable.



Les meilleures applications utilisent diverses sortes de

tests d'intervalles pour s'assurer qu'elles se comportent d'une manière prévisible. Plus une application devient prévisible, moins l'utilisateur a besoin de penser à ce qu'on lui demande, et plus celui-ci peut passer de temps à travailler de manière réellement efficace. Essayez toujours de vous placer du point de vue de l'utilisateur, et non du point de vue du programmeur !

Figure 7.4 : Un mauvais type d'information se traduit par l'affichage d'un message d'erreur.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Int
el)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\SimpleIf3.py =====
Tapez un nombre entre 1 et 10: 5
Vous avez tapé : 5
>>>
===== RESTART: H:\BP4D\Chapitre07\SimpleIf3.py =====
Tapez un nombre entre 1 et 10: 22
>>>
===== RESTART: H:\BP4D\Chapitre07\SimpleIf3.py =====
Tapez un nombre entre 1 et 10: 5.5
Traceback (most recent call last):
  File "H:\BP4D\Chapitre07\SimpleIf3.py", line 1, in <module>
    Value = int(input("Tapez un nombre entre 1 et 10: "))
ValueError: invalid literal for int() with base 10: '5.5'
>>>
```

Choisir entre plusieurs alternatives avec l'instruction if...else

La plupart des décisions que vous avez à prendre dans une application consistent à choisir une ou deux options basées sur certaines conditions. Par exemple, face à un feu routier, vous avez deux options : appuyer sur le frein pour stopper, ou sur l'accélérateur pour continuer. Le bon choix dépend des conditions que vous pouvez observer, donc en l'occurrence de la couleur du feu. Avec Python, vous n'allez pas freiner, mais traiter des alternatives comparables.

Comprendre l'instruction if...else

Avec Python, vous choisissez une des alternatives possibles en utilisant la clause `else` de l'instruction `if`. Une *clause* est un ajout à un bloc de code qui modifie la manière dont celui-ci fonctionne. La plupart des blocs de code supportent de multiples clauses. Dans ce cas, la clause `else` vous permet d'effectuer une tâche alternative lorsque l'instruction `if` n'est pas vérifiée, ce qui accroît l'utilité de celle-ci. En règle générale, on fait référence à la clause `else` de l'instruction `if` en l'appelant l'instruction `if...else`, les trois points indiquant qu'il se passe quelque chose entre ces deux mots.



N'oubliez jamais que la clause `else` n'est exécutée que si l'instruction `if` n'est pas vérifiée. Il est important de bien réfléchir à ce qu'implique le fait d'exécuter certaines tâches lorsqu'une ou plusieurs conditions sont fausses. Ceci peut parfois conduire à des conséquences inattendues.

Utiliser l'instruction if...else dans une application

L'exemple développé dans le fichier `SimpleIf3.py` est moins utile qu'il pourrait l'être dans le cas où l'utilisateur entre une valeur qui se trouve en dehors de l'intervalle demandé. Le fait de taper une donnée qui n'est pas du bon type provoque une erreur, mais saisir une valeur du bon type en dehors de l'intervalle spécifié ne fournit pas non plus une réponse très explicite. Dans l'exemple qui suit, vous allez découvrir un moyen de corriger ce problème en faisant appel à

une clause `else` afin de fournir une alternative à l'instruction `if` si la condition qu'elle teste est fausse. Vous le retrouverez dans le fichier exécutable `IfElse.py`.

1. **Ouvrez une fenêtre de fichier de Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Value = int(input("Tapez un nombre entre 1 et 10 : "))  
  
if (Value > 0) and (Value <= 10):  
    print("Vous avez tapé : ", Value)  
else:  
    print("Vous avez tapé une valeur incorrecte !")
```

Comme dans l'exemple précédent, cette application demande une valeur à l'utilisateur, et elle détermine ensuite si cette entrée est un entier qui appartient à l'intervalle spécifié. Cependant, dans ce cas, la clause `else` fournit en réponse un message alternatif lorsque l'utilisateur a débordé de l'intervalle voulu.



Remarquez que la clause `else` se termine par un deux-points, exactement comme l'instruction `if`. Ceci permet à Python de savoir où la clause se termine. Nombre de messages d'erreur apparaissent tout simplement parce qu'on a oublié d'ajouter ce caractère. Pensez-y !

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Un message vous demande de taper un nombre entre 1 et 10.

4. **Tapez 5 et appuyez sur Entrée.**

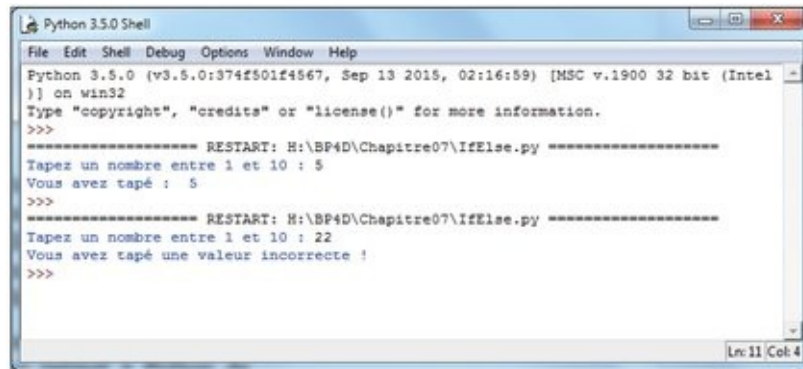
L'application détermine que cette valeur appartient à l'intervalle demandé, et elle affiche le message illustré sur la [Figure 7.3](#).

5. **Reprenez les Étapes 3 et 4, mais tapez cette fois 22**

au lieu de 5.

Cette fois, l'application affiche le message illustré sur la [Figure 7.5](#). L'utilisateur sait maintenant qu'il a entré une valeur ne respectant pas l'intervalle indiqué. Il pourra donc relancer l'application pour corriger son erreur.

Figure 7.5 : Il est toujours bon de fournir une explication lorsque l'utilisateur a effectué une saisie incorrecte.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\IfElse.py =====
Tapez un nombre entre 1 et 10 : 5
Vous avez tapé : 5
>>>
===== RESTART: H:\BP4D\Chapitre07\IfElse.py =====
Tapez un nombre entre 1 et 10 : 22
Vous avez tapé une valeur incorrecte !
>>>
```

Utiliser l'instruction if...elif dans une application

Vous prenez votre petit-déjeuner à l'hôtel. Vous constatez en lisant le menu que vous pouvez avoir des œufs, des crêpes, des gaufres ou encore des céréales. Une fois que vous avez choisi ce que vous voulez dans le menu, le serveur vous amène votre commande. Créer un menu dans lequel l'utilisateur peut effectuer une sélection nécessite quelque chose comme une instruction `if...else`, mais avec un petit quelque chose en plus. Ce petit quelque chose s'appelle ici la clause `elif`. Celle-ci sert à créer un autre jeu de conditions (en fait, son nom est juste une contraction de `else` et de `if`).

Dans l'exemple qui suit, nous allons créer un menu simple. Nous reviendrons un peu plus loin sur notre petit-déjeuner. Vous pouvez ouvrir directement le fichier téléchargeable `IfElif.py` ou suivre ces étapes pas

à pas :

1. **Ouvrez une fenêtre de fichier de Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
print("1. Rouge")
print("2. Orange")
print("3. Jaune")
print("4. Vert")
print("5. Bleu")
print("6. Pourpre")

Choice = int(input("Choisissez votre couleur favorite : "))

if (Choice == 1):
    print("Vous avez choisi Rouge !")
elif (Choice == 2):
    print("Vous avez choisi Orange !")
elif (Choice == 3):
    print("Vous avez choisi Jaune !")
elif (Choice == 4):
    print("Vous avez choisi Vert !")
elif (Choice == 5):
    print("Vous avez choisi Bleu !")
elif (Choice == 6):
    print("Vous avez choisi Pourpre !")
else:
    print("Vous avez fait un choix invalide !")
```

Cet exemple commence par afficher un menu. L'utilisateur voit une liste de choix, puis l'application lui demande de taper un nombre entier (avec l'utilisation de la fonction `int()` qui est placé dans la variable `Choice` afin de déterminer sa couleur préférée.

Une fois que l'utilisateur a effectué son choix, l'application recherche dans la liste les valeurs potentielles. Dans chaque cas, la valeur de `Choice` est comparée avec une certaine réponse afin de créer une condition pour chaque situation. Si, par exemple, l'utilisateur tape 1, l'application indique que la valeur choisie est le rouge. Si aucune des options choisies n'est correcte, la clause `else` est exécutée par défaut.

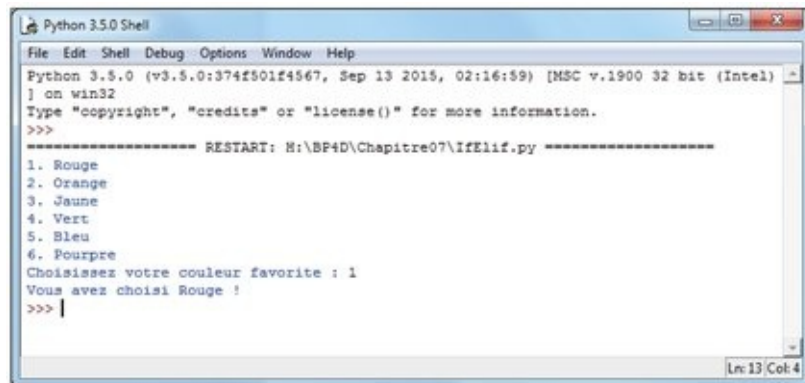
3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Un message vous demande de choisir votre couleur préférée.

4. **Tapez 1 et appuyez sur Entrée.**

L'application affiche le message illustré sur la [Figure 7.6](#).

Figure 7.6 : Les menus vous permettent de choisir une option dans une liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: M:\BP4D\Chapitre07\IfElif.py =====
1. Rouge
2. Orange
3. Jaune
4. Vert
5. Bleu
6. Pourpre
Choisissez votre couleur favorite : 1
Vous avez choisi Rouge !
>>>
```

5. **Reprenez les Étapes 3 et 4, mais tapez cette fois 5 au lieu de 1.**

L'application affiche un autre message.

6. **Reprenez les Étapes 3 et 4, mais tapez cette fois 8 au lieu de 1.**

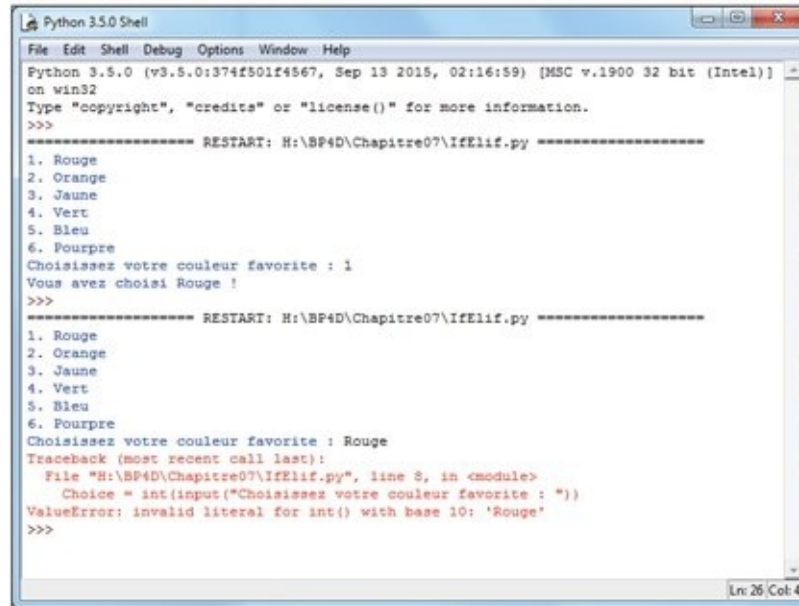
L'application vous dit que vous avez fait un choix invalide.

7. **Reprenez les Étapes 3 et 4, mais tapez cette fois Rouge au lieu de 1.**

Vous n'obtenez cette fois qu'un message d'erreur (voir la [Figure 7.7](#)). Toutes les applications que vous avez à créer devraient être capables de détecter les erreurs et les entrées incorrectes. Le Chapitre 9 vous montrera comment gérer ces erreurs.

Figure 7.7 :

Toutes vos applications devraient pouvoir détecter les erreurs de saisie.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\IfElif.py =====
1. Rouge
2. Orange
3. Jaune
4. Vert
5. Bleu
6. Pourpre
Choisissez votre couleur favorite : 1
Vous avez choisi Rouge !
>>>
===== RESTART: H:\BP4D\Chapitre07\IfElif.py =====
1. Rouge
2. Orange
3. Jaune
4. Vert
5. Bleu
6. Pourpre
Choisissez votre couleur favorite : Rouge
Traceback (most recent call last):
  File "H:\BP4D\Chapitre07\IfElif.py", line 8, in <module>
    Choice = int(input("Choisissez votre couleur favorite : "))
ValueError: invalid literal for int() with base 10: 'Rouge'
>>>
```



De nombreux langages de programmation proposent une instruction `switch` (commuter, si vous voulez) qui vous aide à créer des menus. Ce n'est pas le cas de Python. C'est pourquoi il existe des bibliothèques qui proposent ce genre de complément. Le Chapitre 13 explique comment travailler avec ces bibliothèques.

Utiliser des décisions imbriquées

Le processus de prise décision implique souvent plusieurs niveaux. Revenons à notre exemple d'hôtel-restaurant. Vous choisissez des œufs pour le petit déjeuner, ce qui constitue votre premier niveau de décision. Mais, maintenant, le serveur vous demande avec quoi vous voulez vos œufs. Il ne vous demanderait rien si vous aviez par exemple opté pour des céréales. Dans le cas des œufs, vous devez donc prendre une décision à un second niveau. Et selon ce que vous choisissez maintenant, il pourrait y avoir un

troisième niveau, ou encore une quatrième, et ainsi de suite. Voire même aucune...

Ce type de processus, où un choix dépend d'un autre, est appelé une *imbrication*. Les développeurs se servent souvent de cette technique pour créer des applications capables de prendre des décisions complexes à partir de saisies utilisateur variées. Les sections qui suivent décrivent plusieurs types d'imbrications permettant de demander à Python de prendre des décisions complexes.

Utiliser des instructions if ou if...else multiples

La technique la plus courante consiste à utiliser des combinaisons d'instructions `if` et `if...else`. Cette forme est souvent appelée un *arbre de sélection*, du fait de ses ressemblances avec les branches d'un arbre. Dans ce cas, vous suivez un certain chemin pour obtenir le résultat escompté. L'exemple proposé ici est disponible dans le fichier téléchargeable `MultipleIfElse.py`.

1. **Ouvrez une fenêtre de fichier de Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
One = int(input("Tapez un nombre entre 1 et 10: "))
Two = int(input("Tapez un nombre entre 1 et 10: "))

if (One >= 1) and (One <= 10):
    if (Two >= 1) and (Two <= 10):
        print("Votre nombre secret est : ", One * Two)
    else:
        print("La seconde valeur est incorrecte !")
else:
    print("La première valeur est incorrecte !")
```

Il s'agit simplement d'une extension de l'application `IfElse.py` déjà rencontrée plus haut dans ce chapitre. Remarquez cependant que l'indentation en est différente. La seconde instruction `if...else` est imbriquée à l'intérieur de la première. Cette indentation indique à Python qu'il y a un second niveau d'imbrication.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Un message vous demande de choisir votre couleur préférée.

4. **Tapez 5 et appuyez sur Entrée.**

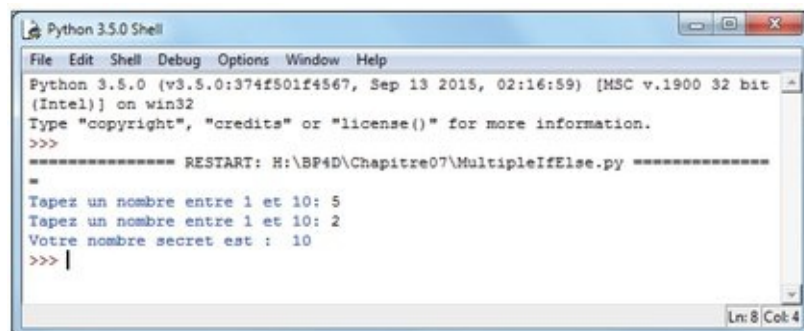
L'application vous demande une autre valeur comprise entre 1 et 10.

5. **Tapez 2 et appuyez sur Entrée.**

La combinaison des deux nombres va s'afficher (voir la [Figure 7.8](#)).

Figure 7.8 :

Plusieurs niveaux d'imbrication vous permettent d'effectuer des tâches plus complexes.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\MultipleIfElse.py =====
=
Tapez un nombre entre 1 et 10: 5
Tapez un nombre entre 1 et 10: 2
Votre nombre secret est : 10
>>> |
```

Dans cet exemple, les entrées sont les mêmes que dans `IfElse.py`. Ainsi, vous obtiendrez un message d'erreur si vous entrez une valeur qui ne se trouve pas dans l'intervalle indiqué. Ce message est adapté au rang de la saisie utilisateur (la première ou la seconde) de manière à ce que celui-ci sache quelle valeur est incorrecte.



Fournir des messages d'erreur spécifiques est toujours utile, car, dans le cas contraire, les utilisateurs risquent de se sentir frustrés et de ne pas comprendre

ce qui se passe. De plus, cela vous aide aussi à localiser plus facilement la source des problèmes.

Combiner d'autres types de décisions

Il est possible d'utiliser n'importe quelle combinaison d'instructions `if`, `if... else` et `if...elif` pour atteindre le but désiré. Vous pouvez imbriquer autant de niveaux que vous le souhaitez de blocs de code afin d'effectuer les contrôles voulus. Par exemple, le [Listing 7.1](#) vous propose de faire votre choix dans le menu du petit-déjeuner d'un hôtel-restaurant (vous le retrouverez dans le fichier téléchargeable `MultipleIfElif.py`).

Listing 7.1 : Créer un menu pour le petit-déjeuner.

```
print("1. Oeufs")
print("2. Crêpes")
print("3. Gaufres")
print("4. Céréales")
MainChoice = int(input("Que voulez-vous au petit déjeuner ? "))

if (MainChoice == 2):
    Meal = "Crêpes"
elif (MainChoice == 3):
    Meal = "Gaufres"

if (MainChoice == 1):
    print("1. Galette au froment")
    print("2. Beignet")
    print("3. Galette de seigle")
    print("4. Crêpe nature")
    Pancake = int(input("Choisissez un type de crêpe : "))

    if (Pancake == 1):
        print("Vous avez choisi des oeufs avec une galette au froment.")
    elif (Pancake == 2):
        print("Vous avez choisi des oeufs avec des beignets.")
    elif (Pancake == 3):
        print("Vous avez choisi des oeufs avec une galette de seigle.")
    elif (Pancake == 4):
        print("Vous avez choisi des oeufs avec une crêpe nature.")
    else:
        print("Nous avons des oeufs, mais pas ce type de crêpe.")

elif (MainChoice == 2) or (MainChoice == 3):
    print("1. Sirop")
    print("2. Confiture de fraises")
    print("3. Sucre en poudre")
    Topping = int(input("Choisissez une garniture : "))

    if (Topping == 1):
        print("Vous avez choisi " + Meal + " au sirop.")
    elif (Topping == 2):
        print("Vous avez choisi " + Meal + " à la confiture de fraises.")
    elif (Topping == 3):
        print("Vous avez choisi " + Meal + " avec du sucre en poudre.")
    else:
        print("Nous avons des " + Meal + ", mais pas cette garniture.")

elif (MainChoice == 4):
    print("Vous avez choisi des céréales.")
```

```
else:
    print("Désolé, nous n'avons pas cela dans notre menu !")
```

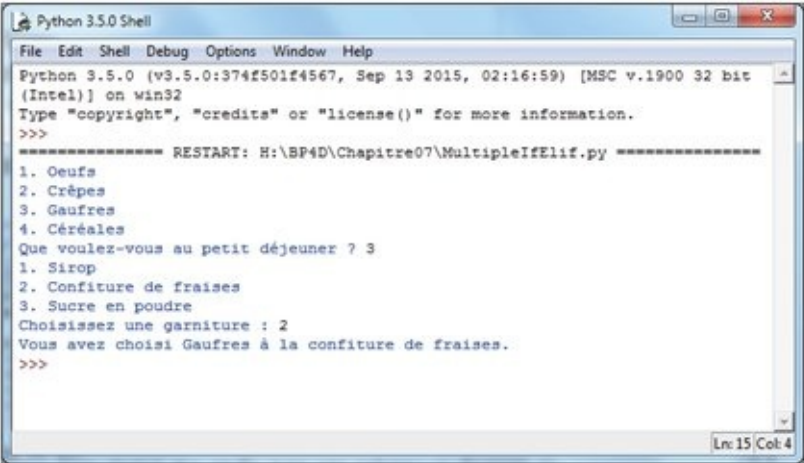
Cet exemple propose quelques fonctionnalités intéressantes. Vous pourriez par exemple croire qu'une instruction `if.. else` nécessite toujours une clause `else`. Or, cet exemple montre une situation dans laquelle cette clause n'a pas lieu d'être. Vous pouvez ici utiliser une instruction `if...elif` pour vous assurer que la variable `Meal` contient la valeur correcte, sans autre option à considérer.

La technique de sélection est en fait la même que celle que vous avez vue dans les exemples

précédents. Un utilisateur entre un nombre appartenant à l'intervalle voulu de manière à obtenir le résultat voulu. Certaines des sélections nécessitent un choix secondaire, et un menu adapté à une telle décision vous est proposé. Par exemple, les œufs sont accompagnés d'un certain autre plat, ce qui n'est pas le cas par exemple des céréales.

Remarquez aussi que cet exemple combine des variables et du texte d'une manière spécifique. Du fait qu'une garniture peut concerner aussi bien les crêpes que les gaufres, vous avez besoin d'une méthode permettant de spécifier de quoi il s'agit. La réponse est basée sur la valeur de la variable `Meal` qui a été définie à un niveau précédent de décision.

Figure 7.9 : De nombreuses applications reposent sur des niveaux de choix multiples dans des menus.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre07\MultipleIfElif.py =====
1. Oeufs
2. Crêpes
3. Gaufres
4. Céréales
Que voulez-vous au petit déjeuner ? 3
1. Sirop
2. Confiture de fraises
3. Sucre en poudre
Choisissez une garniture : 2
Vous avez choisi Gaufres à la confiture de fraises.
>>>
```

La meilleure façon de comprendre cet exemple est de faire différents tests. Essayez diverses combinaisons de menus pour voir de plus près comment tout cela se déroule. Par exemple, la [Figure 7.9](#) illustre ce que vous obtenez si vous demandez des gaufres avec de la confiture de fraises.

Chapitre 8

Effectuer des tâches répétitives

Dans ce chapitre :

- ▶ Effectuer une tâche un certain nombre de fois.
- ▶ Effectuer une tâche jusqu'à ce qu'elle soit terminée.
- ▶ Imbriquer les boucles.

Jusqu'ici, tous les exercices de ce livre effectuaient une série d'étapes juste une fois avant de se terminer. Mais le monde réel ne fonctionne pas de cette manière. La plupart des tâches effectuées par des êtres humains sont répétitives (et parfois aussi fastidieuses). Par exemple, votre docteur trouve que vous avez besoin de faire plus d'exercices physiques, et il vous dit de faire cinquante pompes tous les jours. Si vous vous arrêtez à une seule pompe, votre état de santé ne va pas s'améliorer, et vous n'aurez définitivement pas suivi les conseils du docteur. Bien entendu, comme vous savez exactement la quantité de pompes à faire, vous pouvez effectuer cette tâche médicale un nombre connu de fois. Python ne fait pas

de pompes, mais il est tout à fait capable de réaliser le même genre de répétition grâce à l'instruction `for`.

Malheureusement, vous ne savez pas toujours combien de fois une certaine tâche doit être réalisée. Supposons par exemple que, en tant que numismate averti, vous avez devant vous une pile de pièces réputées être rares. Vous prenez la pièce du dessus, vous l'examinez, et vous déterminez si cette pièce est ou non la plus belle du lot. Mais, quelle que soit à ce stade votre réponse, cela ne suffit pas à terminer la tâche. Vous devez examiner toutes les pièces une par une en les prenant chaque fois sur le dessus de la pile. Et ce n'est qu'une fois arrivé à la dernière que vous pourrez décider que votre tâche est terminée. Mais, comme vous ne savez pas au départ combien la pile contient de pièces, vous ne savez pas non plus combien de fois vous allez devoir pratiquer cette « dépilation ». Vous n'obtenez la réponse que quand la pile est vide. Python, lui aussi, sait effectuer ce type de travail répétitif en utilisant l'instruction `while`.



La plupart des langages de programmation appellent cette sorte de séquence répétitive une *boucle*. L'idée consiste à décrire la répétition comme une sorte de mouvement circulaire, dans lequel le code exécute une série de tâches encore et encore, et cela jusqu'à ce que la boucle se termine. Ces boucles sont un élément essentiel de multiples parties d'une application, notamment les menus. En fait, écrire aujourd'hui une application sans utiliser des boucles est virtuellement impossible.

Dans certains cas, vous devez même créer des boucles qui se trouvent dans d'autres boucles. C'est par exemple le cas pour produire une table de

multiplication. La boucle intérieure s'occupe des valeurs en colonne, et la boucle extérieure se charge des lignes (ou le contraire). Vous trouverez un exemple de ce type dans ce chapitre. En attendant, prenons les choses par le commencement.

Traiter des données en utilisant l'instruction for

Le premier bloc de code que rencontrent les programmeurs dès qu'il est question de boucle est l'instruction `for`. Il est d'ailleurs très difficile d'imaginer un langage de programmation qui serait dépourvu de cette instruction. En l'occurrence, la boucle est exécutée un nombre fixé de fois, et vous savez quel est ce nombre avant même que la boucle ne débute. Du fait que tout est connu à l'avance, `for` est certainement le type de boucle le plus simple à utiliser.

Comprendre l'instruction for

Une boucle `for` (*pour*) débute par l'instruction `for`. Cette instruction décrit le comportement de la boucle. Peu importe que cette description soit une série de lettres dans une chaîne, ou des éléments numériques dans une collection. Vous pouvez même spécifier une plage de valeurs en faisant appel à la fonction `range()`. En voici un exemple très simple :

```
for Letter in "Bonjour !":
```

La ligne d'instruction débute par le mot-clé `for`. L'élément suivant est une variable appelée `Letter` qui,

dans ce cas, contient un unique élément. Le mot-clé `in` indique à Python que la séquence continue. Ici, cette séquence prend la forme d'une chaîne de caractères, « Bonjour ! ». L'instruction `for` se termine par un deux-points, exactement comme les instructions de prise de décision décrites dans le Chapitre 7.

Sous cette instruction `for`, vous pouvez suivre les tâches répétitives à effectuer dans la boucle `for`. Python considère, comme d'habitude, que toutes les lignes indentées qui viennent après forment un bloc de code qui définit le contenu de la boucle. Là encore, le principe est le même que pour les instructions de prise de décision étudiées au Chapitre 7.

Créer une boucle for simple

La meilleure manière de voir comment fonctionne une boucle `for`, c'est encore d'en créer une. Dans cet exemple, la séquence va être basée sur une chaîne de caractères. La boucle va traiter chaque caractère un par un jusqu'à ce qu'il n'y en ait plus. Vous pouvez retrouver ce code dans le fichier téléchargeable

`SimpleFor.py`.

1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
LetterNum = 1

for Letter in "Bonjour !":
    print("La lettre ", LetterNum, " est ", Letter)
    LetterNum+=1
```

L'exemple commence par créer une variable, `LetterNum`,

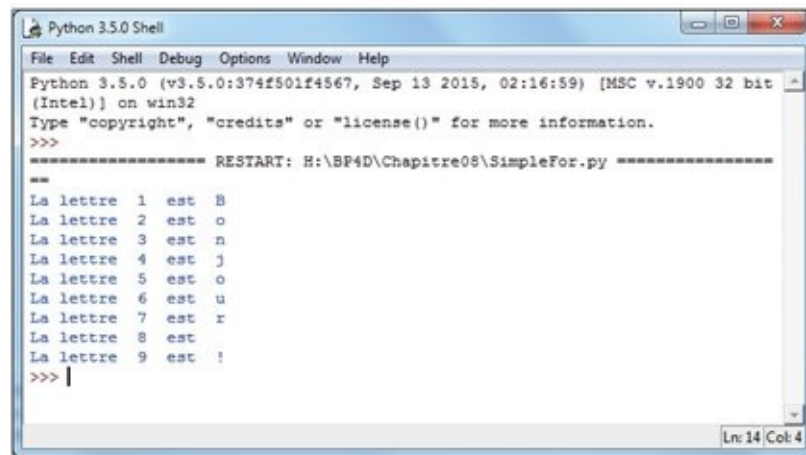
servant à assurer le suivi du nombre de lettres déjà traité. Chaque fois qu'un passage dans la boucle se termine, la variable `LetterNum` est incrémentée d'une unité.

L'instruction `for` parcourt la séquence de lettres de la chaîne « Bonjour ! ». Chaque lettre trouvée est placée dans la variable `Letter`. L'instruction qui suit affiche sous une forme lisible le contenu de nos deux variables, `LetterNum` et `Letter`.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. Elle affiche la liste des lettres de la chaîne, ainsi que leur rang (voir la [Figure 8.1](#)).

Figure 8.1 : Une boucle `for` permet par exemple de traiter l'une après l'autre tous les caractères d'une chaîne.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\SimpleFor.py =====
==
La lettre 1 est B
La lettre 2 est o
La lettre 3 est n
La lettre 4 est j
La lettre 5 est o
La lettre 6 est u
La lettre 7 est r
La lettre 8 est 
La lettre 9 est !
>>> |
```

Contrôler l'exécution avec l'instruction break

La vie est souvent faite d'exception à une règle. Par exemple, vous êtes chargé de mettre en place une ligne de production de montres connectées. Mais, à un certain moment, une certaine pièce vient à manquer dans l'approvisionnement de la ligne. Si cette pièce est indisponible, la production doit s'arrêter au milieu du cycle de fabrication. Le travail n'est pas terminé, mais il faut attendre que la pièce soit à nouveau

réapprovisionnée.

Les ordinateurs sont également concernés par ce genre de problème. Supposons que vous traitiez des données provenant d'une ressource en ligne. À un certain moment, le réseau rencontre un problème et la connexion est interrompue. Le flux de données cesse temporairement, et donc votre application n'a plus rien à faire, même si le nombre de tâches à réaliser n'a pas été accompli.



La clause `break` rend possible une rupture dans une boucle. Cependant, vous ne devez pas l'utiliser telle quelle. Il faut l'entourer d'une instruction `if` qui détermine la condition générant ce *break*. Cette instruction doit dire quelque chose comme ceci : si le flux n'arrive plus, alors quitter la boucle.

Dans l'exemple qui suit, vous allez voir ce qui se passe lorsque le compteur de la boucle atteint un certain niveau lors du traitement d'une chaîne de caractères. Certes, il n'est pas très complexe, et ce pour rester simple et compréhensible, mais il reflète ce qui se passe dans le monde réel lorsque le traitement d'une certaine donnée devient trop long (ce qui peut indiquer une condition d'erreur). Suivez ces étapes, ou ouvrez directement le fichier téléchargeable `ForBreak.py` :

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```

Value = input("Tapez un mot d'au plus 8 caractères: ")
LetterNum = 1

for Letter in Value:
    print("La lettre ", LetterNum, " est ", Letter)
    LetterNum+=1
    if LetterNum > 8:
        print("La chaîne est trop longue !")
        break

```

Cet exemple prolonge celui de la section précédente. Cependant, il permet cette fois à l'utilisateur de saisir une chaîne de caractères de longueur quelconque. Si cette chaîne contient plus de huit caractères, l'application cesse l'exécution de la boucle.

L'instruction `if` contient le code de la condition d'arrêt. Si `LetterNum` devient plus grand que 8, cela signifie que la chaîne est trop longue. Remarquez aussi le niveau d'indentation supplémentaire utilisé pour l'instruction `if`. Dans ce cas, l'utilisateur voit un message d'erreur lui indiquant que la chaîne est trop longue, et le code exécute une instruction `break` pour terminer la boucle.

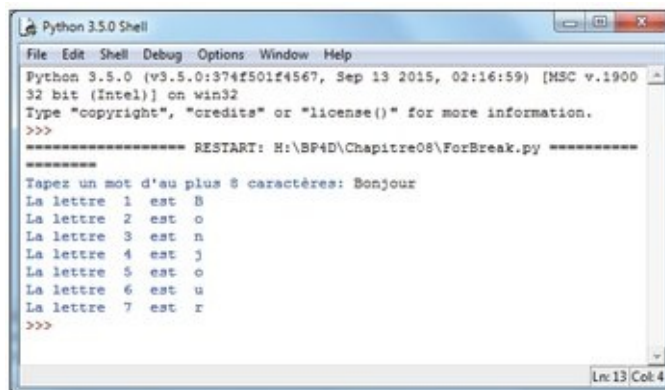
3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Elle demande à l'utilisateur de taper quelque chose.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application liste chaque caractère de la chaîne (voir la [Figure 8.2](#)).

Figure 8.2 : Une chaîne courte est traitée avec succès par l'application.

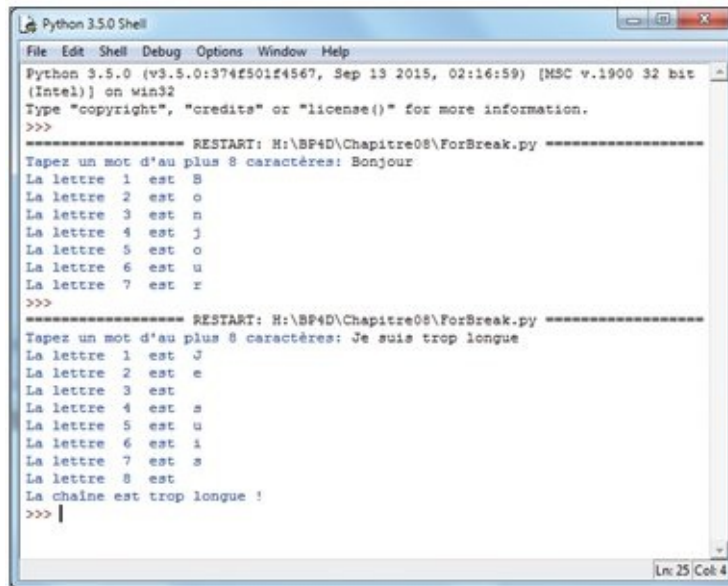


5. **Reprenez les Étapes 3 et 4, mais tapez cette fois la**

chaîne Je suis trop longue.

L'application affiche le message d'erreur attendu, puis elle stoppe la boucle une fois les 8 premiers caractères affichés (voir la [Figure 8.3](#)).

Figure 8.3 : Les chaînes longues sont tronquées pour s'assurer qu'elles ne dépassent pas la longueur autorisée.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\ForBreak.py =====
Tapez un mot d'au plus 8 caractères: Bonjour
La lettre 1 est B
La lettre 2 est o
La lettre 3 est n
La lettre 4 est j
La lettre 5 est o
La lettre 6 est u
La lettre 7 est r
>>>
===== RESTART: H:\BP4D\Chapitre08\ForBreak.py =====
Tapez un mot d'au plus 8 caractères: Je suis trop longue
La lettre 1 est J
La lettre 2 est e
La lettre 3 est s
La lettre 4 est s
La lettre 5 est u
La lettre 6 est i
La lettre 7 est s
La lettre 8 est 
La chaîne est trop longue !
>>>
```



Cet exemple ajoute un contrôle de la longueur de la chaîne à votre panoplie de traitement des erreurs potentielles. Le Chapitre 7 vous a montré comment s'assurer qu'une valeur appartient bien à un certain intervalle. Ici, ce contrôle est nécessaire pour s'assurer que des données, en l'occurrence une chaîne, ne dépassent pas la taille autorisée pour les champs d'un formulaire ou d'une base de données. De plus, limiter le format des saisies rend votre système moins sensible à certains modes de piratage, et donc plus sûr.

Contrôler l'exécution avec l'instruction continue

Parfois, vous pouvez avoir besoin de contrôler chaque élément d'une séquence, mais sans traiter certains d'entre eux. Supposons par exemple que vous deviez traiter toutes les informations sur des voitures dont les caractéristiques sont enregistrées dans une base de données, sauf les voitures bleues. Après tout, peut-être n'avez-vous pas besoin de vous attarder sur cette couleur particulière. La clause `break`, qui ne fait que terminer la boucle, ne convient pas ici. Sinon, vous ne verriez pas les autres véhicules de la base de données, dont vous ne connaissez bien sûr pas à l'avance la couleur.



L'alternative la plus courante à la clause `break` est la clause `continue`. Comme `break`, elle apparaît à l'intérieur d'une instruction `if`. Par contre, le déroulement de l'application se poursuit alors avec l'élément suivant de la séquence, au lieu de terminer totalement celle-ci.

Les étapes qui suivent vous montrent la différence entre `continue` et `break`. Ici, le code refuse de traiter la lettre `j`, mais il accepte toutes les autres lettres de l'alphabet. Vous pouvez retrouver cet exemple dans le fichier téléchargeable `ForContinue.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
LetterNum = 1

for Letter in "Bonjour !":
    if Letter == "j":
        continue
    print("La lettre j rencontrée, traitement interrompu.")
```

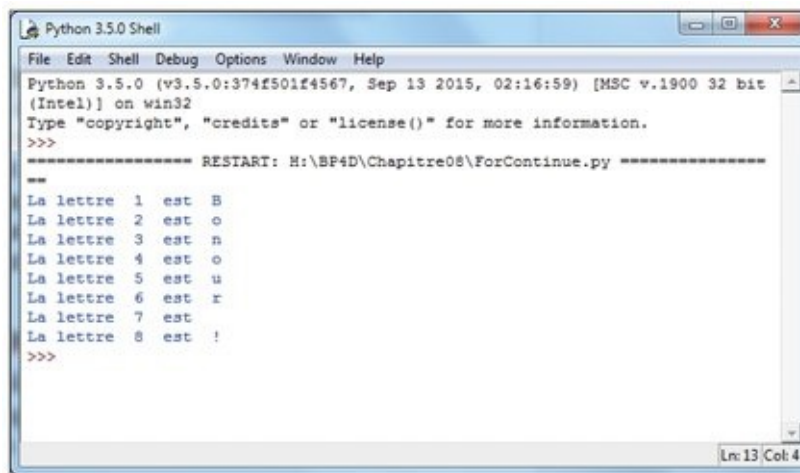
```
print("La lettre ", LetterNum, " est ", Letter)
LetterNum+=1
```


Ce code part d'un exemple étudié plus haut dans ce chapitre, mais il ajoute une instruction `if` avec la clause `continue` dans le bloc de code correspondant. Remarquez que l'indentation indique bien que la commande `print ()` qui suit appartient bien à ce bloc de code. En fait, vous ne verrez jamais ce message, car l'itération courante dans la boucle se termine immédiatement (autrement dit, à la différence de `break`), la boucle continue à s'exécuter en passant directement à l'itération suivante.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche la suite des lettres de la chaîne, ainsi que leur rang, mais en omettant la lettre `j` qui est traitée au niveau de la clause `continue` (voir la [Figure 8.4](#)).

Figure 8.4 : La clause `continue` permet d'éviter le traitement de certains éléments.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\ForContinue.py =====
==
La lettre 1 est B
La lettre 2 est o
La lettre 3 est n
La lettre 4 est o
La lettre 5 est u
La lettre 6 est r
La lettre 7 est
La lettre 8 est !
>>>
```

Contrôler l'exécution avec la clause `pass`

Le langage Python contient quelque chose qu'on ne trouve pas souvent dans son domaine : une seconde sorte de clause `continue`. La clause `pass` fonctionne presque de la même manière que `continue`, si ce n'est

qu'elle permet de terminer l'exécution du bloc de code dans lequel elle apparaît. L'exemple qui suit est strictement le même que le précédent, à une exception près : le remplacement de `continue` par `pass`. Vous le retrouvez également dans le fichier téléchargeable `ForPass.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
LetterNum = 1

for Letter in "Bonjour !":
    if Letter == "j":
        pass
        print("La lettre j rencontrée, non traitée.")
    print("La lettre ", LetterNum, " est ", Letter)
    LetterNum+=1
```

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche la suite des lettres de la chaîne, ainsi que leur rang. Du fait de la présence de la clause `pass`, le message concernant la lettre *j* est maintenant affiché.



NdT : En l'état, ce code ne donne pas la réponse telle qu'elle devrait apparaître. En effet, la clause `pass` n'empêche pas la lettre *j* de continuer à être traitée, et donc affichée à sa position, ce qui n'est pas logique. Il convient donc de l'éliminer de la série en incrémentant le compteur de lettres, et de poursuivre ensuite normalement le traitement via `continue`. D'où la variante suivante de ce code :

```

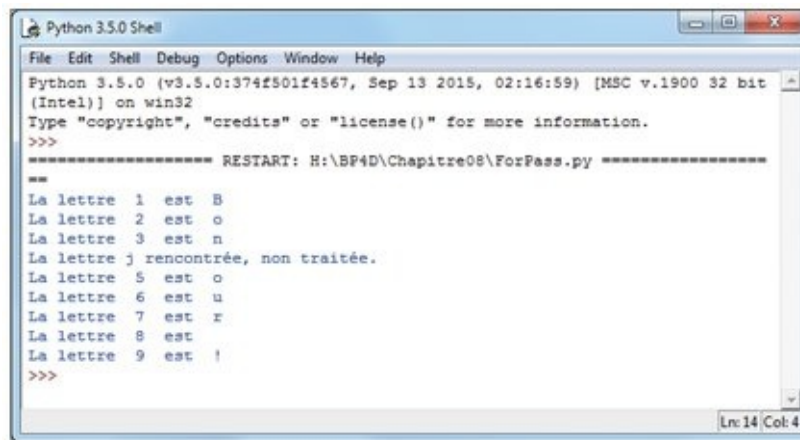
LetterNum = 1

for Letter in "Bonjour !":
    if Letter == "j":
        pass
        print("La lettre j rencontrée, non traitée.")
        LetterNum+=1
        continue
    print("La lettre ", LetterNum, " est ", Letter)
    LetterNum+=1

```

Comme l'illustre la [Figure 8.5](#), la réponse est maintenant cohérente.

Figure 8.5 : La clause `pass` permet d'effectuer un post-traitement sur une entrée non conforme ou indésirable.



```

Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\ForPass.py =====
==
La lettre 1 est B
La lettre 2 est o
La lettre 3 est n
La lettre j rencontrée, non traitée.
La lettre 5 est o
La lettre 6 est u
La lettre 7 est r
La lettre 8 est 
La lettre 9 est !
>>>

```



La clause `continue` permet d'éliminer en silence certains éléments spécifiques dans une séquence, et donc d'éviter d'exécuter du code supplémentaire pour ces éléments. Avec l'emploi de la clause `pass`, au contraire, le but est d'effectuer un post-traitement, par exemple enregistrer un état instantané dans un fichier journal, afficher un message destiné à l'utilisateur, ou encore gérer le problème d'une autre manière. Les clauses `continue` et `pass` sont donc très proches l'une de l'autre, mais elles s'emploient dans des situations distinctes.

Contrôler l'exécution avec la clause `else`

Python dispose encore d'une autre clause que l'on ne trouve pas dans tous les langages : `else` (*sinon*). Cette clause rend possible l'exécution de code même si vous n'avez aucun élément à traiter dans une séquence. Par exemple, vous pourriez tout simplement indiquer à l'utilisateur qu'il n'y a plus rien à faire. Et c'est ce que propose le code qui suit. Vous le trouverez également dans le fichier téléchargeable

`ForElse.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Value = input("Tapez moins de 8 caractères : ")
LetterNum = 1

for Letter in Value:
    print ("La lettre ", LetterNum, " est ", Letter)
    LetterNum+=1
else:
    print("La chaîne est vide.")
```

Là encore, il s'agit d'une variation sur le même thème, si ce n'est qu'un simple appui sur Entrée sans saisir quoi que ce soit déclenchera la colère de la clause `else`.

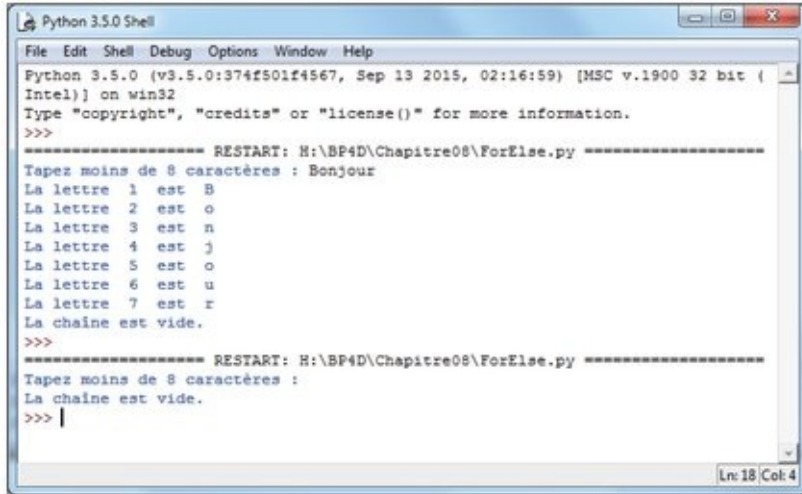
3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Elle demande à l'utilisateur de saisir quelque chose.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application liste tous les caractères de la chaîne, comme sur la [Figure 8.2](#).

Figure 8.6 : La clause `else` rend possible l'exécution de tâches basées sur une séquence vide.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\ForElse.py =====
Tapez moins de 8 caractères : Bonjour
La lettre 1 est B
La lettre 2 est o
La lettre 3 est n
La lettre 4 est j
La lettre 5 est o
La lettre 6 est u
La lettre 7 est r
La chaîne est vide.
>>>
===== RESTART: H:\BP4D\Chapitre08\ForElse.py =====
Tapez moins de 8 caractères :
La chaîne est vide.
>>> |
```

5. Répétez les Étapes 3 et 4, mais ne tapez aucun texte. Appuyez simplement sur Entrée.

Le message associé à la clause `else` vous indique maintenant que votre chaîne est vide (voir la [Figure 8.6](#)).



Il est facile de faire une mauvaise utilisation de la clause `else`, car une séquence vide ne signifie pas forcément l'absence d'une saisie. Cela peut également être le signal d'une erreur, ou du fait que d'autres conditions ont besoin d'un traitement particulier. Assurez-vous que vous comprenez comment l'application gère les données pour vous assurer que la clause `else` ne va pas en fait masquer une condition d'erreur, plutôt que de la rendre visible afin de pouvoir y remédier.

Traiter des données avec l'instruction `while`

L'instruction `while` (tant que) trouve son intérêt dans des situations où vous n'êtes pas sûr du nombre de données qu'une application doit traiter en boucle. Elle

demande à Python de continuer à répéter le bloc de code tant qu'il trouve des éléments. Ce type de boucle est très utile, par exemple si vous devez effectuer des tâches comme télécharger des fichiers dont la taille n'est pas connue à l'avance, ou encore diffuser des informations à partir d'une source comme une radio Internet. Toute application dans laquelle vous ne pouvez pas dire à l'avance quel est le volume de données à traiter est une bonne candidate à l'emploi de l'instruction `while`. Voyons cela de plus près.

Comprendre l'instruction while

L'instruction `while` travaille avec une condition au lieu d'une séquence. Cette condition indique que l'instruction `while` doit effectuer une certaine tâche, tant que cette condition reste vraie (ou jusqu'à ce qu'elle devienne fausse). Prenons l'exemple d'une file d'attente devant la caisse d'un grand magasin. La caissière s'occupe des clientes et des clients l'un après l'autre, et ce jusqu'à ce que la file d'attente soit vide. Bien entendu, de nouveaux clients peuvent se présenter à tout moment à la caisse, et il est donc impossible de savoir à l'avance combien de Caddies la caissière aura à traiter. Mais supposons maintenant que, au bout de 25 client(e)s, la caissière ait le droit de prendre une pause. Voici à peu près comment cela pourrait s'écrire :

```
while numClients <= 25 :  
    ## calculer la valeur du caddie du client courant
```

Cette séquence débute par le mot-clé `while`. Elle pose alors une condition, ici une variable `NumClients` dont la valeur doit être inférieure ou égale à 25 pour continuer. En l'état, rien ne précise la valeur initiale de la variable `NumClients`, pas plus que le code ne définit la

manière dont celle-ci évolue. La seule chose qui est connue au début de cette procédure, c'est que `NumClients` doit être inférieur ou égal à 25 pour que la boucle continue à être effectuée. Comme vous en avez maintenant l'habitude, l'instruction se termine par un deux-points, et les lignes de code qui vont suivre seront indentées.



Du fait que l'instruction `while` n'effectue pas une série de tâches un nombre déterminé de fois, il est possible de produire une *boucle sans fin*, autrement dit qui ne se termine jamais. Reprenons notre exemple. Si la valeur de `NumClients` est égale à zéro lorsque l'application rencontre l'instruction `while`, et que la valeur de la variable n'augmente jamais, la boucle sera répétée sans cesse jusqu'à la fin du monde (ou jusqu'à ce que l'ordinateur soit éteint, ce qui devrait prendre moins de temps). Les boucles sans fin peuvent provoquer toutes sortes de problèmes bizarres, comme de gros ralentissements, voire même un gel complet du système. Il faut donc à tout prix éviter de se retrouver dans une telle situation. En d'autres termes, vous devez *toujours* fournir une méthode pour que la boucle `while` se termine (à la différence d'une boucle `for`, dont on connaît à l'avance le nombre d'itérations). Il vous faut donc vous préoccuper de trois choses :

1. **Créer l'environnement de la condition (par exemple, affecter la valeur 0 à `NumClients`).**
2. **Tester la condition à l'intérieur de l'instruction `while` (par exemple, `NumClients <= 25`).**
3. **Mettre la condition à jour de manière à s'assurer que la condition ne sera plus vérifiée au bout d'un temps fini (par exemple en incrémentant `NumClients` dans le bloc de code `while`).**



Comme pour la boucle `for`, vous avez la possibilité de modifier le comportement par défaut de l'instruction `while`. En fait, vous avez accès aux mêmes quatre clauses que celles déjà étudiées avec `for` :

- ✓ `break` : Termine la boucle courante.
- ✓ `continue` : Termine immédiatement le processus pour l'élément courant.
- ✓ `pass` : Termine le traitement pour l'élément courant après avoir exécuté l'instruction dans le bloc `if`.
- ✓ `else` : Fournit une technique alternative lorsque les conditions de la boucle ne sont pas remplies.

Utiliser l'instruction while dans une application

Vous pouvez utiliser l'instruction `while` de multiples manières. Mais ici, nous allons nous en tenir à un exemple simple qui vous montrera le mode de fonctionnement de `while`. Nous allons afficher un compteur basé sur l'état d'une variable appelée `Sum`. Celle-ci est initialisée avec la valeur zéro, et le compteur s'arrête lorsque la variable atteint la valeur 5. Vous pouvez également retrouver cet exemple dans le fichier téléchargeable `SimpleWhile.py`.

1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**


```
Sum = 0  
  
while Sum < 5:  
    print(Sum)  
    Sum+=1
```

Ce code met en œuvre d'une manière très simple les trois tâches que vous devez effectuer lorsque vous travaillez avec une boucle `while`. Elle commence par initialiser la variable `Sum` avec la valeur 0, ce qui est la première étape à franchir. La condition elle-même apparaît sur la ligne de l'instruction `while`. Et la fin du code accomplit la troisième étape, c'est-à-dire actualiser la valeur de `Sum`. Pour que tout cela apparaisse clairement, le code affiche la valeur courante de la variable à chaque passage dans la boucle.

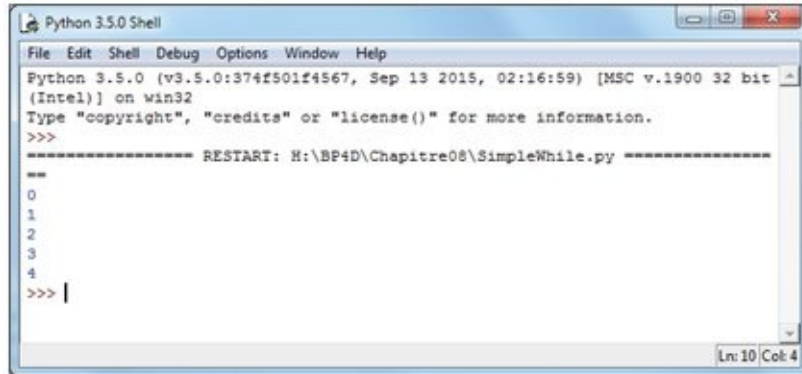


Une instruction `while` offre davantage de souplesse que la boucle `for`. La méthode employée pour actualiser la valeur de `sum` est relativement évidente. Cependant, vous pourriez utiliser n'importe quelle autre méthode nécessitée par l'application pour atteindre ses objectifs. Rien ne dit que vous devriez mettre à jour la variable `sum` d'une façon spécifique. Par exemple, vous pouvez tester l'état de trois ou quatre variables si vous le désirez. Bien entendu, plus la condition sera complexe, et plus vous risquerez d'entrer dans une boucle sans fin. Vous devez donc bien réfléchir à la manière de quitter votre boucle `while`.

3. **Choisissez la commande Run Module dans le menu Run.**

Python exécute la boucle `while` et affiche la séquence numérique illustrée sur la [Figure 8.7](#).

Figure 8.7 : Cette simple boucle `while` affiche une suite de nombres.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\SimpleWhile.py =====
==
0
1
2
3
4
>>> |
```

Imbriquer des boucles

Dans certains cas, il est possible d'utiliser au choix une boucle `for` ou une boucle `while` pour obtenir le même effet. Dans l'exemple qui suit, vous allez créer un générateur de table de multiplication en *imbriquant* une boucle `while` à l'intérieur d'une boucle `for`. Comme vous voulez que la sortie produite soit bien présentée, vous allez également le mettre en forme (nous reviendrons sur ce sujet dans le Chapitre 11). Cet exemple est également disponible dans le fichier téléchargeable `NestedLoop.py`.

1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
X = 1
Y = 1

print('{:>4}'.format(' '), end= ' ')

for X in range(1, 11):
    print('{:>4}'.format(X), end=' ')

print()

for X in range(1,11):
```

```
print('{:>4}'.format(X), end=' ')
while Y <= 10:
    print('{:>4}'.format(X * Y), end=' ')
    Y+=1
print()
Y=1
```

Cet exemple commence par créer deux variables, X et Y, chargées de contenir respectivement la valeur en ligne et en colonne de la table.

Pour que cette table soit lisible, le code doit créer un en-tête en haut de chaque colonne et à gauche de chaque ligne. Lorsque l'utilisateur voit la valeur 3 en haut, et la valeur 4 à gauche, il doit simplement les suivre pour savoir où les rangées correspondantes se coupent, et donc où se trouve leur produit.

La première instruction `print()` ajoute un espace (puisque rien n'apparaît dans le coin supérieur gauche de la table, comme l'illustre un peu plus loin la [Figure 8.8](#)). Toutes les autres instructions de formatage créent un espacement de quatre caractères de large. La partie `{ : >4}` du code détermine la taille de la colonne. La fonction `format (' ')` définit ce qui est affiché dans cet espace. L'attribut `end` de l'instruction `print ()` change le caractère de fin de ligne pour remplacer le retour chariot par un espace.

La première boucle `for` affiche les nombres de 1 à 10 en haut de la table. La fonction `range ()` permet de spécifier facilement cet intervalle. Vous devez lui passer la valeur de départ, donc 1 dans ce cas, et la valeur finale augmentée d'une unité, soit 11 ici.

Arrivé là, le curseur attend patiemment à la fin de la ligne d'en-tête. Pour le renvoyer à la ligne suivante, un simple appel à la fonction `print ()`, sans aucun argument, suffit à faire le travail.

Même si le morceau de code qui suit peut paraître un peu complexe, il suffit de le suivre ligne par ligne pour en comprendre le mécanisme. La table de multiplication calcule des valeurs allant de $1 * 1$ à $10 * 10$, et donc vous avez besoin de dix lignes et de dix colonnes pour afficher

tous les résultats de ces opérations. L'instruction `for` demande donc à Python de créer dix lignes.

Jetez maintenant un coup d'œil à la [Figure 8.8](#). À l'intérieur de la boucle, le premier appel à la commande `print ()` affiche l'en-tête de la ligne courante. Bien entendu, vous devez formater cette présentation, et c'est pourquoi le code utilise un espacement de quatre caractères se terminant par un espace vide (au lieu d'un retour chariot) pour pouvoir afficher les résultats des calculs sur la même ligne.

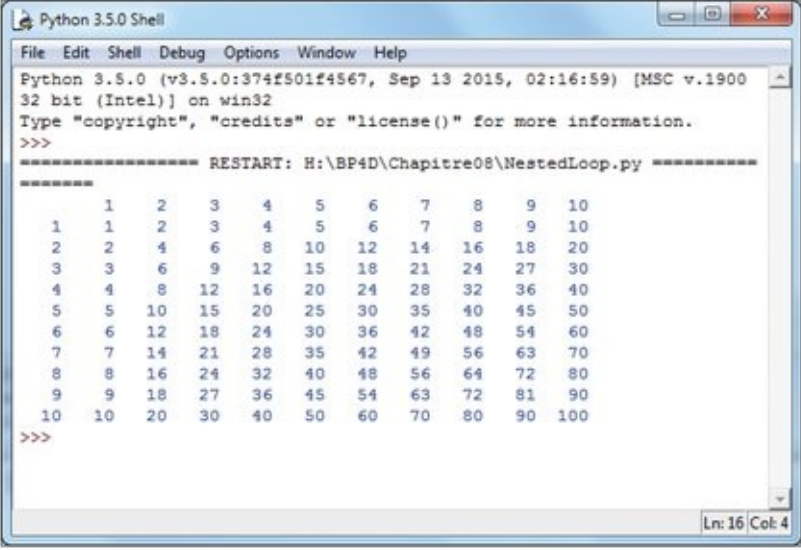
C'est maintenant qu'arrive la boucle `while`. Celle-ci gère l'affichage d'une colonne donnée, et ce pour la ligne courante. Les valeurs à placer dans les colonnes sont à chaque fois le produit de X par Y. Là encore, la sortie est formatée pour respecter la règle des quatre espaces (bien entendu, vous pourriez en choisir une autre, dès lors que le résultat est lisible et cohérent). La boucle `while` se termine une fois que Y a atteint sa valeur limite, 10, grâce à l'incrémentation `Y+=1`.

Vous revenez alors à la boucle `for`. L'instruction `print ()` termine la ligne courante en envoyant un retour chariot. De plus, Y doit être réinitialisé à la valeur 1 afin qu'il soit prêt pour la ligne suivante. Et ainsi de suite...

3. **Choisissez la commande Run Module dans le menu Run.**

Python affiche la table de multiplication illustrée sur la [Figure 8.8](#).

Figure 8.8 : Grâce au formatage, cette table de multiplication est plaisante à lire.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre08\NestedLoop.py =====
>>>
      1   2   3   4   5   6   7   8   9  10
1  1   2   3   4   5   6   7   8   9  10
2  2   4   6   8  10  12  14  16  18  20
3  3   6   9  12  15  18  21  24  27  30
4  4   8  12  16  20  24  28  32  36  40
5  5  10  15  20  25  30  35  40  45  50
6  6  12  18  24  30  36  42  48  54  60
7  7  14  21  28  35  42  49  56  63  70
8  8  16  24  32  40  48  56  64  72  80
9  9  18  27  36  45  54  63  72  81  90
10 10  20  30  40  50  60  70  80  90 100
>>>
```

Ln: 16 Col: 4

Chapitre 9

Les erreurs ? Quelles erreurs ?

Dans ce chapitre :

- ▶ Définir les problèmes de communications avec Python.
- ▶ Comprendre la cause des erreurs.
- ▶ Gérer les conditions d'erreur.
- ▶ Spécifier qu'une erreur s'est produite.
- ▶ Développer vos propres indicateurs d'erreur.
- ▶ Effectuer des tâches après une erreur.

La plupart du temps, un code d'une certaine complexité comporte des erreurs. Si votre application se bloque soudain sans raison apparente, il y a une erreur. Voir apparaître un message obscur est un autre type d'erreur. Cependant, nombre d'erreurs peuvent aussi se produire sans que vous ne receviez la moindre notification. Par exemple, une application pourrait effectuer les mauvais calculs sur une série de valeurs qui lui ont été fournies, ce qui donnerait un résultat erroné. Dans ce cas, vous pourriez ne jamais vous en apercevoir, jusqu'à ce que quelqu'un vous en avertisse ou que vous soyez pris d'un sérieux doute.

Les erreurs ont aussi besoin de cohérence. Vous pourriez par exemple en déclencher une dans certaines circonstances, disons par exemple quand le réseau est surchargé, et pas dans d'autres. En bref, des erreurs peuvent survenir dans toutes sortes de situations et pour toutes sortes de raisons. Ce chapitre se consacre donc à cette importante question et il vous explique comment réagir lorsque votre application rencontre des erreurs.

Il n'y a rien de surprenant à ce que des erreurs surgissent. Les applications sont écrites par des humains, et les humains font des erreurs. La plupart des développeurs appellent ces erreurs des *exceptions*, dans le sens où il s'agit d'exception à la règle. Puisque des exceptions se produisent dans les applications, vous devez les détecter et y réagir dès que possible. Détecter et traiter une exception relève de ce que l'on appelle un *gestionnaire d'erreur*, ou encore un *gestionnaire d'exception* (on parle aussi de système de gestion d'exception, ou d'erreur). Pour détecter correctement des erreurs, vous avez besoin de savoir quelle peut en être la cause, et pourquoi elles se produisent. Pour cela, vous devez *intercepter* l'exception, autrement dit l'examiner et si possible en faire quelque chose. Nous découvrirons donc aussi comment gérer des exceptions dans vos propres applications.

Parfois, votre code détecte une erreur dans l'application. Lorsque cela se produit, vous devez *lever* (raise) ou *propager* (throw) une exception. Vous pouvez rencontrer ces deux termes (avec des variantes) pour désigner à peu près la même chose : votre code a rencontré une erreur qu'il ne peut pas gérer, et il passe une information sur cette erreur à une autre pièce de code, le gestionnaire d'exception,

qui va interpréter, traiter et avec de la chance réparer cette erreur. Même si Python possède des tas de messages génériques qui couvrent la plupart des situations, certaines sont tout à fait spéciales, ce qui peut vous obliger à utiliser des messages d'erreur personnalisés (sans compter que ceux de Python sont tous en anglais). Par exemple, vous pourriez avoir besoin de fournir un support particulier pour une application de base de données, chose que Python ne couvre pas par défaut. Il est important de savoir quand il est nécessaire de traiter les exceptions localement, quand les envoyer au code appelant, et quand créer des exceptions spéciales, de manière à ce que chaque partie de l'application sache comment gérer l'exception. Nous reviendrons sur tous ces sujets dans ce chapitre.

Il y a aussi des circonstances où vous devez vous assurer que votre application gère une exception de manière élégante, même si cela implique de mettre fin à l'application. Python dispose pour cela de la clause `finally`, qui s'exécute toujours, même si une exception survient. Ceci permet par exemple de refermer proprement des fichiers ou d'effectuer d'autres tâches essentielles dans le bloc de code associé à cette clause. Nous verrons cela de plus près à la fin de chapitre.

Savoir pourquoi Python ne vous comprend pas

Les langages de programmation comme les ordinateurs sont des choses inanimées. Ils n'ont aucun état d'âme, aucun désir, aucune volonté personnelle. De plus, ils ne pensent pas. Ils acceptent

à peu près littéralement ce que les programmeurs leur disent de faire. Et c'est tout. Et c'est en même temps parfois très frustrant quand on a soudain l'impression (fausse) que le langage de programmation ou l'ordinateur semble vaquer à ses propres occupations sans se soucier du reste...



Ni Python ni l'ordinateur ne sont capables de comprendre ce que vous voulez faire lorsque vous tapez du code. Ils suivent vos instructions en les prenant au pied de la lettre. Par exemple, vous n'avez *pas* demandé à Python de supprimer un fichier de données, à moins qu'une certaine situation absurde se produise. Mais, si vous n'avez pas très clairement explicité les choses, il n'est pas impossible que Python supprime le fichier de données, situation absurde ou pas. Lorsque ce genre de chose se produit, on dit généralement que l'application *bogue*, ou qu'elle a un *bug*. Mais, en réalité, il s'agit simplement d'erreurs de codage qui pourraient être détectées et corrigées en faisant appel à un *débogueur* (un outil qui permet de stopper ou de mettre en pause l'exécution d'une application, d'examiner le contenu des variables, et plus généralement de disséquer l'application pour voir où se trouve le problème).

Des erreurs se produisent bien souvent dans des situations où le développeur fait certaines suppositions qui se révèlent fausses. Bien entendu, ceci comprend les réactions des utilisateurs, qui ne se soucient probablement pas du tout de l'extrême niveau de soin que vous avez pris pour écrire une application parfaite. L'utilisateur entre une mauvaise donnée. Python, qui n'est en rien concerné par ces problématiques, traite cette donnée, même si votre intention initiale était toute autre. Python ne

comprend rien à ces questions existentielles. Il effectue son travail en fonction des règles que vous avez définies. Ce qui implique qu'il vous appartient, et uniquement à vous, de créer des règles qui protègent les utilisateurs d'eux-mêmes.

Python n'est ni proactif ni créatif. Ce sont des qualités qui n'existent que chez le développeur. Si une erreur réseau se produit, ou si l'utilisateur fait quelque chose d'inattendu, Python est incapable de créer une solution afin de résoudre ce problème. Il ne fait qu'exécuter du code. Si vous n'avez rien prévu pour gérer une telle situation, il est probable que l'application plantera sans autre forme de procès (et éventuellement les données de l'utilisateur avec). Bien entendu, un développeur ne peut pas anticiper tous les cas possibles. C'est pourquoi la plupart des applications complexes comportent des erreurs (en l'occurrence, il s'agirait d'une erreur par omission).



Ne vous bercez d'aucune illusion : écrire un code parfaitement à l'épreuve des balles est juste une idée absurde. Vous devez être totalement convaincu qu'un certain nombre de *bogues* se produiront dans la durée de vie de vos applications, que la nature et les utilisateurs continueront à effectuer des actions auxquelles personne n'avait pensé, et que même le meilleur programmeur du monde ne peut pas anticiper toutes les conditions d'erreur possible. En d'autres termes, supposez *toujours* que votre application est sujette à des erreurs qui vont provoquer des exceptions. C'est en raisonnant ainsi que vous pourrez créer des applications non pas parfaites, mais plus efficaces et plus fiables.

Prendre en considération les sources d'erreurs

Vous pourriez peut-être deviner les sources d'erreurs potentielles de votre application en lisant dans le marc de café, mais vous vous doutez bien que cela risque de ne pas être très efficace. En fait, les erreurs se répartissent en catégories bien définies qui vous aident, du moins dans une certaine mesure, à mieux comprendre quand et où elles sont susceptibles de se produire. Les deux catégories principales sont les suivantes :

- Les erreurs qui surgissent à un moment spécifique
- Les erreurs qui sont d'un type spécifique

Les sections qui suivent explicitent ces notions. Le concept général, c'est que vous avez besoin de réfléchir à la classification des erreurs de manière à commencer à les trouver et les réparer dans vos applications avant même qu'elles ne deviennent un problème.

Erreurs surgissant à un moment spécifique

Ce type d'erreur se décompose en deux catégories principales :

- Au moment de la compilation
- Au moment de l'exécution

Dans tous les cas, votre application va mal. Voyons

donc cela de plus près.

Erreur de compilation

Une telle erreur se produit au moment où vous demandez à Python d'exécuter votre application. Avant même de commencer, il doit interpréter le code et le transcrire dans une forme compréhensible par l'ordinateur. L'ordinateur, de son côté, ne connaît qu'une langue extrêmement binaire, et de surcroît liée à son processeur et à son architecture. Si les instructions que vous avez saisies sont mal formées, ou si des informations indispensables manquent, Python ne peut pas effectuer la conversion dans le langage de la machine. Il renvoie donc une erreur que vous devez réparer pour que l'application puisse être lancée.

Fort heureusement, les erreurs de ce type sont les plus faciles à localiser et à fixer. Du fait que l'application ne peut même pas se lancer, l'utilisateur ne voit jamais ce genre d'erreur. C'est à vous de résoudre le problème en corrigeant le code erroné.



L'apparence d'une erreur de compilation devrait attirer votre attention sur le fait qu'il existe d'autres fautes de frappe ou d'autres omissions dans le code. Il est toujours bénéfique de contrôler les lignes de code environnantes afin de s'assurer qu'il n'y a pas d'autres problèmes potentiels qui pourraient ne se révéler qu'un moment de l'exécution.

Erreur d'exécution

C'est fait. Python a compilé votre code et lancé l'application. Et c'est à partir de là que peut se

produire une erreur d'exécution. De telles erreurs peuvent avoir des causes variées, et certaines sont plus difficiles à cerner que d'autres. Vous savez qu'une erreur d'exécution s'est produite si l'application cesse brusquement de fonctionner en affichant une exception, ou encore si l'utilisateur se plaint de recevoir une sortie erronée, ou bien encore de l'instabilité de l'application.



Toutes les erreurs d'exécution ne produisent pas une exception. Certaines peuvent se traduire par un gel du fonctionnement, par une sortie qui semble erratique, ou encore par des données endommagées. Certaines peuvent aussi affecter d'autres applications, voire même la plate-forme sur laquelle le code est exécuté. Votre responsabilité est donc engagée en tant que développeur !

De nombreuses erreurs d'exécution sont provoquées par des fautes de frappe, voire des oublis dans la saisie. Par exemple, mal orthographier le nom d'une variable empêchera Python de placer la bonne valeur au bon endroit (de ce point de vue, la confusion entre majuscule et minuscule est rédhibitoire). Oublier un argument facultatif, mais nécessaire dans l'appel à une méthode peut aussi provoquer des problèmes. En général, il est possible de localiser de telles erreurs en faisant appel aux services d'un débogueur, ou plus simplement en relisant votre code ligne par ligne.

Les erreurs d'exécution peuvent également être provoquées par des événements extérieurs non associés à votre code. Par exemple, l'utilisateur entre une information incorrecte et que l'application ne sait pas traiter, ce qui provoque une exception. Un problème sur le réseau peut tout aussi bien rendre

des données inaccessibles. Parfois aussi, c'est le matériel de l'ordinateur qui a une défaillance, ce qui provoque une erreur non repérable. Tous ces exemples forment des *erreurs par omission*, que votre application pourrait être en mesure de récupérer si votre application contient le code nécessaire à cela. Il est important de prendre en compte ces différents cas lorsque vous vous concevez votre application.

Distinguer les types d'erreurs

Connaître les types d'erreurs vous aide à comprendre où rechercher d'éventuels problèmes dans une application. Les exceptions fonctionnent comme de nombreuses autres choses de la vie. Par exemple, vous savez que les appareils électroniques ne marchent pas sans une source d'alimentation électrique. Si vous essayez d'allumer votre téléviseur, et qu'il ne se passe rien, la première chose à faire est de vérifier que son cordon d'alimentation est correctement branché dans la prise de courant.



Comprendre les types d'erreurs vous aide à localiser celles-ci plus vite, plus tôt et plus efficacement. Et donc aussi à commettre moins d'erreurs avec les erreurs. Les meilleurs développeurs savent bien qu'il est toujours plus facile de corriger les problèmes pendant le développement d'une application que lorsqu'elle est en service, car les utilisateurs sont par essence impatients, et ils veulent que les problèmes soient réparés immédiatement et correctement. De plus, en prenant cette question à bras le corps dès le début du cycle de développement limite la quantité de code à vérifier et à corriger.

Toute l'astuce consiste à savoir où regarder. Python (comme la plupart des autres langages de programmation) distingue ces types d'erreurs :

- Syntaxiques
- Sémantiques
- Logiques

Les sections qui suivent vont préciser cela en procédant par ordre de difficulté croissante.

Erreurs de syntaxe

Chaque fois que vous faites une faute de frappe quelconque, vous commettez une erreur de syntaxe. Bien souvent, celles-ci sont plutôt faciles à détecter, car le code n'exécute tout simplement pas. L'interpréteur peut même vous guider en affichant un message d'erreur qui vous signale l'emplacement de l'erreur. Cependant, certaines erreurs de syntaxe sont très délicates à découvrir.

Ainsi, Python est extrêmement sensible à la capitalisation des caractères. Il suffit donc de mettre une majuscule à la place d'une minuscule (ou l'inverse) dans une seule occurrence d'une variable pour que celle-ci ne fonctionne pas comme elle le devrait. Localiser ce genre de faute peut parfois être un véritable challenge.



La plupart des erreurs de syntaxe sont déclenchées lors de la compilation, et l'interpréteur les pointe pour vous. Dans ce cas, la correction est facilitée, puisque l'interpréteur vous dit généralement ce qu'il faut modifier, et ce avec une précision considérable. Même si ce n'est pas le cas, les erreurs de syntaxe empêchent l'application de fonctionner correctement,

et vous les retrouvez au cours des phases de test. Peu d'erreurs de ce type devraient persister au-delà, du moins si vous avez testé complètement votre application.

Erreurs sémantiques

Lorsque vous créez une boucle qui s'exécute plus de fois qu'il n'est nécessaire, vous ne recevez en général aucune information de l'application vous signalant qu'il y a une erreur. Du point de vue de l'application, tout semble normal, mais des boucles supplémentaires peuvent provoquer toutes sortes d'erreurs dans les données. Vous n'avez pas fait de faute de frappe, vous avez provoqué sans le vouloir une *erreur sémantique*.



Les erreurs sémantiques sont dues au fait que l'idée sous-jacente à une série d'étapes est erronée. Le résultat est incorrect, même si le code semble se dérouler normalement. De telles erreurs sont parfois ardues à localiser, et un bon débogueur est bien souvent votre meilleur allié (le Chapitre 19 présente des outils utiles pour travailler avec Python, y compris pour déboguer des applications).

Erreurs de logique

Certains ne font pas la distinction entre erreurs sémantiques et erreurs de logique, mais ils ont tort. Une erreur sémantique se produit lorsque le code est pour l'essentiel correct, mais que l'implémentation est mauvaise (comme une boucle qui s'exécute une fois de trop). Les erreurs de logique sont dues à un défaut de raisonnement du développeur. Cela se produit souvent lors de l'emploi incorrect d'un opérateur

relationnel ou logique. Mais elles peuvent aussi être dues à bien d'autres causes. Par exemple, un programmeur pense que les données qu'il traite sont toujours stockées sur le disque dur local. Et l'application risque donc fort de se comporter d'une manière inhabituelle si elle essaie de charger des données à partir d'un disque réseau.



Les erreurs logiques sont assez difficiles à corriger, puisque le problème ne se trouve pas dans le code, mais dans la conception qui a amené à l'écrire. C'est le processus mental de développement qui est fautif. Et donc la personne qui est responsable de cette erreur est la moins bien placée pour la localiser. Avoir une seconde paire d'yeux (disons, un partenaire avisé) est donc toujours une bonne idée. De même, formaliser avec précision les spécifications d'une application aidera beaucoup, car cela permettra de mieux suivre la logique cachée derrière les tâches que cette application doit réaliser.

Intercepter les exceptions

D'un point de vue général, un utilisateur ne devrait jamais voir un message ou une boîte de dialogue dont l'affichage est provoqué par une exception. L'application devrait toujours intercepter cette exception, et la traiter avant qu'elle n'arrive jusqu'à l'utilisateur. Bien entendu, le monde réel n'est pas aussi simple que cela, et certaines exceptions arrivent à passer à travers les mailles du filet. Pour autant, le but est d'essayer d'intercepter toute exception potentielle lors du développement d'une application. C'est le sujet des sections qui suivent.

Comprendre les exceptions intégrées à Python

Python est accompagné d'une grande quantité d'exceptions prédéfinies, bien plus que vous ne pourriez l'imaginer. Vous pouvez en voir la liste en consultant l'adresse Web <https://docs.python.org/3.5/library/exceptions.html>. La documentation partage la liste des exceptions en plusieurs catégories. En voici un rapide aperçu :

- ✓ **Classes de base** : Elles fournissent les blocs de construction essentiels pour les autres exceptions (par exemple, l'exception judicieusement appelée `Exception`). Vous pouvez cependant voir directement certaines d'entre elles lorsque vous travaillez avec une application (par exemple, `ArithmeticError`).

- ✓ **Exceptions dites concrètes** : Certaines applications peuvent rencontrer des erreurs difficiles à résoudre parce qu'il n'existe pas de bon procédé pour les traiter, ou parce qu'elles signalent un événement que l'application doit gérer elle-même. Par exemple, si un système vient à manquer de mémoire, Python génère une exception `MemoryError`. Résoudre ce problème est difficile, car il n'est pas toujours possible de libérer de la mémoire occupée par d'autres programmes. Lorsque l'utilisateur appuie sur une touche d'interruption (comme

Ctrl+C ou Suppr), Python génère une exception `KeyboardInterrupt`. C'est bien à l'application de réagir face à cette situation avant de passer à d'autres tâches.

✓ **Exceptions de l'OS** : Le système d'exploitation peut générer des erreurs que Python passe ensuite à votre application. Si, par exemple, celle-ci tente d'ouvrir un fichier qui n'existe pas, le système génère une erreur `FileNotFoundError`.

✓ **Avertissements** : Python essaie aussi parfois de vous avertir d'un événement inattendu, ou d'actions qui pourraient par la suite engendrer des erreurs. Par exemple, une tentative inappropriée d'utiliser une certaine ressource, comme une icône, va déclencher l'exception `ResourceWarning`. Il s'agit bien d'un simple avertissement, pas d'une erreur. Vous n'êtes donc pas verbalisé. Vous pouvez l'ignorer, mais vous devez savoir qu'une erreur peut survenir plus tard.

Gérer les exceptions de base

Pour gérer les exceptions, vous devez indiquer votre intention à Python puis fournir le code qui va effectuer ce genre de tâche. Vous avez pour cela de nombreuses méthodes à votre disposition. Nous allons dans ce qui suit commencer par le plus simple pour aller vers des méthodes plus complexes, mais aussi plus souples et plus puissantes.

Gérer une unique exception

Dans le Chapitre 7, le fichier `IfElse.py` (et les autres exemples) avait une manière bien à elle de traiter le cas où un utilisateur tapait une valeur non conforme. Une partie de la solution consistait à contrôler une plage de valeurs. Mais cela n'aidait pas à résoudre une situation dans laquelle l'utilisateur saisisait du texte au lieu d'une valeur numérique. Un gestionnaire d'exception fournit une solution plus élaborée à ce problème, comme le montrent les étapes qui suivent. Vous pouvez également retrouver directement ce code dans le fichier téléchargeable `BasicException1.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
try:
    Value = int(input("Tapez un nombre entre 1 et 10 : "))
except ValueError:
    print("Vous devez taper un nombre entre 1 et 10 !")
else:
    if (Value > 0) and (Value <= 10):
        print("Vous avez tapé : ", Value)
    else:
        print("La valeur que vous avez tapée est incorrecte !")
```

Le code qui est placé à l'intérieur du bloc `try` gère ses propres exceptions. Dans ce cas, cette gestion signifie obtenir de l'utilisateur une saisie via l'appel à `int (input ())`. Si une exception se produit à l'extérieur de ce bloc, elle ne sera pas gérée par celui-ci. Bien entendu, du coup, la tentation pourrait être grande de placer tout le code dans un seul bloc `try`. Mais ce serait en fait une très mauvaise idée. Un bon gestionnaire d'exception doit être petit et spécifique à un certain problème potentiel pour qu'il soit plus facile de localiser et donc de corriger celui-ci.



Le bloc `except` recherche une exception spécifique, dans ce cas `ValueError`. Si l'utilisateur provoque une exception `ValueError` en tapant par exemple **Bonjour** au lieu d'une valeur numérique, ce bloc d'exception particulier est exécuté. Si l'utilisateur génère une autre exception, le bloc `except` ne pourra pas la gérer.

Le bloc `except` contient tout le code qui est exécuté si le bloc de code `try` génère une exception. Tout le reste est lié à `try`, car vous ne voulez exécuter ce bloc que si l'utilisateur saisit une valeur correcte, autrement dit un nombre compris entre 1 et 10.

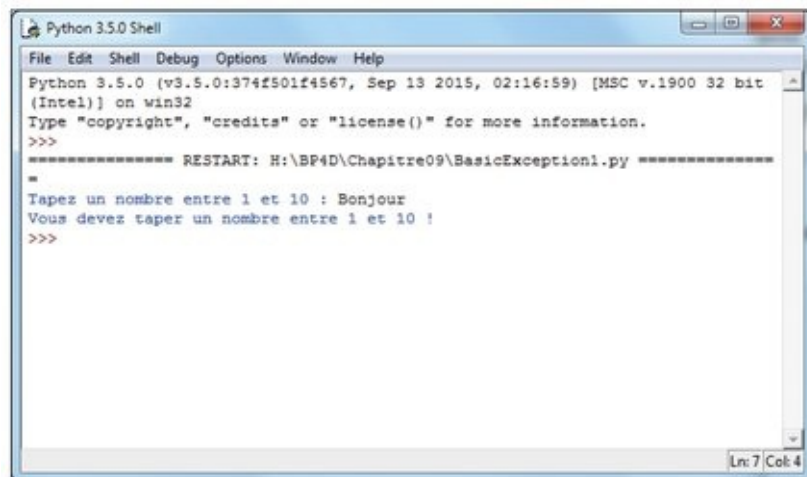
3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application vous demande de taper un nombre entre 1 et 10.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application affiche un message d'erreur (voir la [Figure 9.1](#)).

Figure 9.1 : Une saisie du mauvais type génère un message d'erreur au lieu d'une exception.



5. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 5.5.**

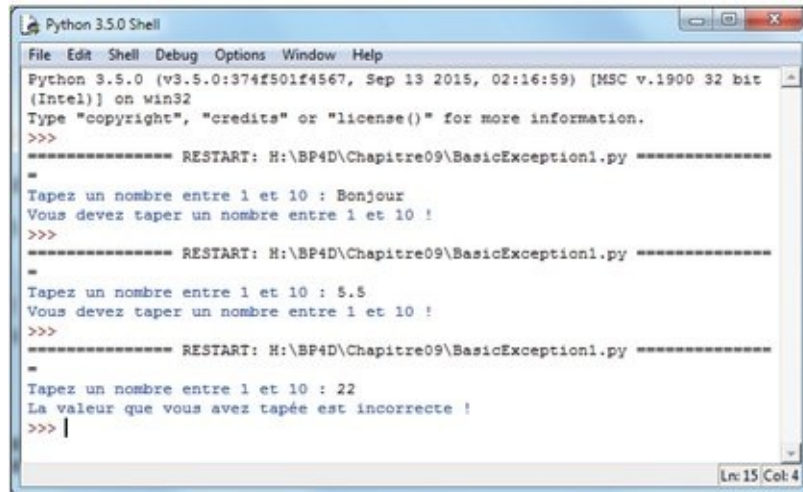
Vous obtenez le même message d'erreur.

6. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 22.**

L'application affiche le message qui demande de respecter

l'intervalle entier (voir la [Figure 9.2](#)). Le gestionnaire d'exception ne prend pas en charge ce genre d'erreur. Il faut donc le traiter ailleurs.

Figure 9.2 : Le gestionnaire d'exception ne traite pas le cas d'un entier qui n'appartient pas à l'intervalle demandé.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
=
Tapez un nombre entre 1 et 10 : Bonjour
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
=
Tapez un nombre entre 1 et 10 : 5.5
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
=
Tapez un nombre entre 1 et 10 : 22
La valeur que vous avez tapée est incorrecte !
>>> |
```

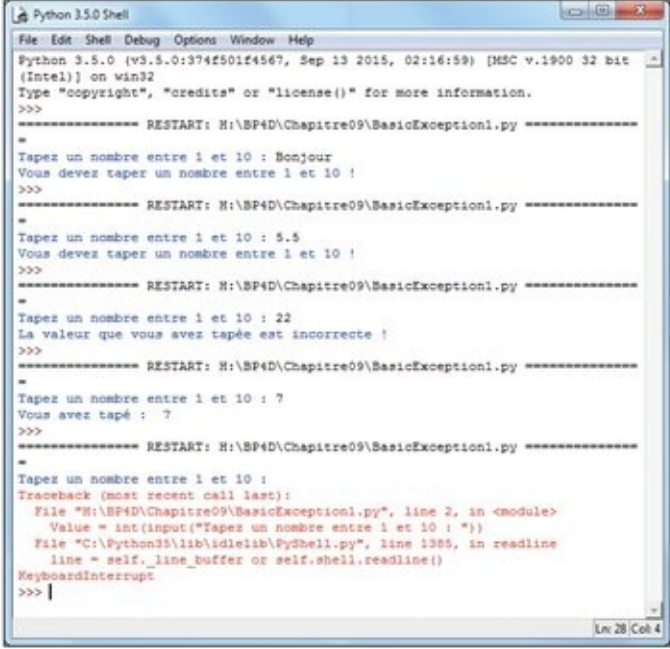
7. Reprenez les Étapes 3 et 4, mais en tapant cette fois 7.

Maintenant, l'application peut vous informer que votre nombre est correct et afficher sa valeur. Même si tout cela semble demander bien du travail pour obtenir ce niveau de contrôle, vous ne pourriez absolument pas être certain que votre application est correcte sans en passer par ce genre d'étapes.

8. Reprenez les Étapes 3 et 4, mais appuyez cette fois sur la combinaison Ctrl+C, ou Cmd+C, ou sur l'équivalent de votre propre système.

L'application va gérer une exception `KeyboardInterrupt`, comme l'illustre la [Figure 9.3](#). Du fait que cette exception n'est pas prise en charge par le code, elle reste problématique pour l'utilisateur. Nous verrons dans la suite de ce chapitre plusieurs techniques pour régler une telle situation.

Figure 9.3 : Dans cet exemple, le gestionnaire d'exception ne traite que le cas de `ValueError`.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
Tapez un nombre entre 1 et 10 : Bonjour
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
Tapez un nombre entre 1 et 10 : 5.5
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
Tapez un nombre entre 1 et 10 : 22
La valeur que vous avez tapée est incorrecte !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
Tapez un nombre entre 1 et 10 : 7
Vous avez tapé : 7
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException1.py =====
Tapez un nombre entre 1 et 10 :
Traceback (most recent call last):
  File "H:\BP4D\Chapitre09\BasicException1.py", line 2, in <module>
    Value = int(input("Tapez un nombre entre 1 et 10 : "))
  File "C:\Python35\lib\idlelib\PyShell.py", line 1385, in readline
    line = self._line_buffer or self.shell.readline()
KeyboardInterrupt
>>> |
```

Utiliser la clause `except` sans exception

Vous pouvez créer dans Python un gestionnaire d'exception générique, et donc sans spécifier une exception particulière. Dans la plupart des cas, il y a quelques bonnes raisons pour fournir une exception spécifique :

- ✓ Éviter de masquer une exception que vous n'avez pas prise en compte lors de la conception de l'application.
- ✓ S'assurer que d'autres savent précisément quelles exceptions votre application va gérer.
- ✓ Gérer les exceptions correctement en utilisant un code adapté à chacune d'elles.

Cependant, vous avez parfois besoin de quelque chose de plus générique, par exemple si vous utilisez une bibliothèque externe ou si vous interagissez avec un service externe. Les étapes qui suivent vous en proposent un exemple, que vous retrouverez également dans le fichier téléchargeable

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
try:
    Value = int(input("Tapez un nombre entre 1 et 10 : "))
except:
    print("Vous devez taper un nombre entre 1 et 10 !")
else:
    if (Value > 0) and (Value <= 10):
        print("Vous avez tapé : ", Value)
    else:
        print("La valeur que vous avez tapée est incorrecte !")
```

Vous constatez que la seule différence avec l'exemple précédent est la disparition de `ValueError` à la suite de la clause `except`. Le résultat, c'est qu'ici cette clause interceptera toute autre exception susceptible de se produire.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application vous demande de taper un nombre entre 1 et 10.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application affiche un message d'erreur (reportez-vous à la [Figure 9.1](#)).

5. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 5.5.**

Vous obtenez le même message d'erreur.

6. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 22.**

L'application affiche le message qui demande de respecter l'intervalle entier (reportez-vous à la [Figure 9.2](#)). Le gestionnaire d'exception ne prend pas en charge ce genre d'erreur. Il faut donc le traiter ailleurs.

7. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 7.**

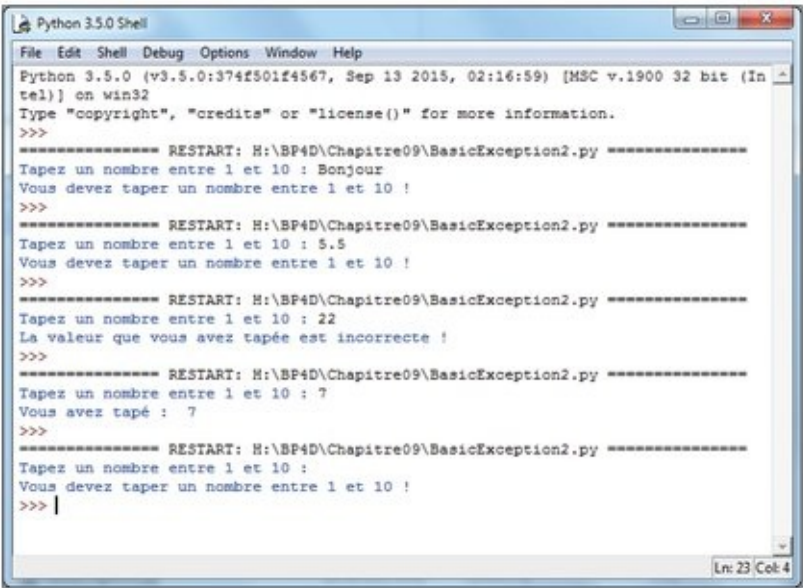
Maintenant, l'application peut vous informer que votre

nombre est correct et afficher sa valeur. Même si tout cela semble demander bien du travail pour obtenir ce niveau de contrôle, vous ne pourriez à nouveau absolument pas être certain que votre application est correcte sans en passer par ce genre d'étapes.

8. **Reprenez les Étapes 3 et 4, mais appuyez cette fois sur la combinaison Ctrl+C, ou Cmd+C, ou sur l'équivalent de votre propre système.**

Cette fois, vous obtenez le message d'erreur habituellement associé à une saisie incorrecte (voir la [Figure 9.4](#)). Ce message ne correspond pas à la situation, mais du moins l'aspect positif est qu'il n'y a pas eu de plantage de l'application. Aucune donnée n'a été perdue, et l'application peut repartir. Cette technique offre donc certains avantages, mais elle doit être maniée avec précaution.

Figure 9.4 : Un gestionnaire d'exception générique va intercepter également l'exception `KeyboardInterrupt`.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException2.py =====
Tapez un nombre entre 1 et 10 : Bonjour
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException2.py =====
Tapez un nombre entre 1 et 10 : 5.5
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException2.py =====
Tapez un nombre entre 1 et 10 : 22
La valeur que vous avez tapée est incorrecte !
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException2.py =====
Tapez un nombre entre 1 et 10 : 7
Vous avez tapé : 7
>>>
===== RESTART: H:\BP4D\Chapitre09\BasicException2.py =====
Tapez un nombre entre 1 et 10 :
Vous devez taper un nombre entre 1 et 10 !
>>> |
```

Exceptions et arguments

La plupart des exceptions ne fournissent pas d'arguments (c'est-à-dire une liste de valeurs que vous pouvez tester pour obtenir des informations supplémentaires). Une exception est ou n'est pas, là est la question. Cependant, quelques rares exceptions fournissent des arguments. Vous les verrez plus loin

dans ce livre. Ces arguments vous en disent plus sur l'exception et ils fournissent les détails dont vous avez besoin pour corriger le problème.



Pour couvrir tout de même l'éventail des possibilités, cette section vous propose un exemple simple qui génère une exception avec un argument. Vous pouvez si vous le souhaitez passer directement à la suite puisque ce sujet sera traité plus en détail par ailleurs. Sinon, vous avez aussi la possibilité d'ouvrir ce code depuis le fichier téléchargeable `ExceptionWithArguments.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
import sys

try:
    File = open('myfile.txt')
except IOError as e:
    print("Erreur lors de l'ouverture du fichier : \r\n" +
          "Numéro de l'erreur : {0}\r\n".format(e.errno) +
          "Texte de l'erreur : {0}".format(e.strerror))
else:
    print("Le fichier a bien été ouvert.")
    File.close();
```

Cet exemple utilise certaines fonctionnalités avancées. Par exemple, l'instruction `import` récupère le code d'un autre fichier. Vous en apprendrez davantage à ce sujet dans le Chapitre 10.

La fonction `open ()` ouvre un fichier auquel elle fournit un accès via la variable `File`. Nous reviendrons sur ce sujet dans le Chapitre 15. Étant donné que le fichier appelé `myfile.txt` n'existe pas dans le dossier de l'application, le système d'exploitation ne peut pas l'ouvrir. Il va donc informer Python de ce problème.

Essayer d'ouvrir un fichier inexistant provoque une exception `IOError`. Celle-ci donne accès à deux arguments :

- `errno` : Le numéro d'erreur entier renvoyé par le système d'exploitation.
- `strerror` : Renvoie une chaîne de caractères (en anglais) décrivant la nature de l'exception.

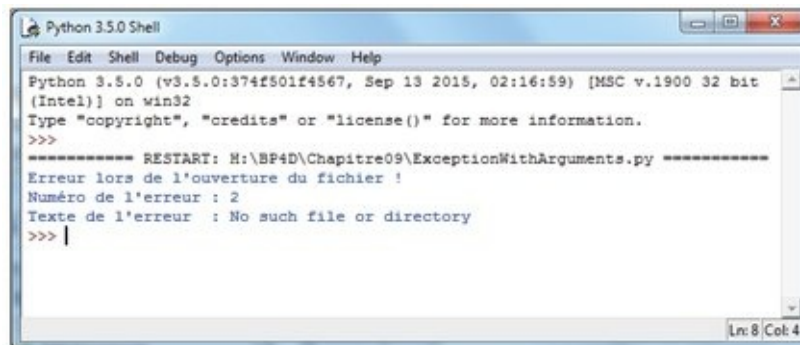
La clause `as` place l'information renvoyée par l'exception dans une variable appelée `e`, ce qui vous permet d'y accéder. Le bloc `except` contient un appel à `print ()` qui formate les informations fournies par l'exception afin de les rendre plus lisibles.

Si vous décidez de créer un fichier appelé `myfile.txt`, c'est la clause `else` qui serait exécutée. Dans ce cas, vous obtiendriez un message indiquant que le fichier a été ouvert normalement. Le code se termine alors en refermant le fichier, sans effectuer d'autre traitement.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application se contente d'afficher un rapport sur l'ouverture (impossible) du fichier, comme l'illustre la [Figure 9.5](#).

Figure 9.5 : Tenter d'ouvrir un fichier qui n'existe pas provoque forcément une exception.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: N:\BP4D\Chapitre09\ExceptionWithArguments.py =====
Erreur lors de l'ouverture du fichier !
Numéro de l'erreur : 2
Texte de l'erreur : No such file or directory
>>> |
```

Obtenir la liste des arguments d'une exception

La liste des éventuels arguments varie selon l'exception et de ce que fournit l'appelant. Il n'est pas toujours facile de se faire une idée de ce que vous pouvez obtenir lorsque vous recherchez des informations complémentaires. Une manière de gérer cette question consiste à afficher simplement toute cette liste en utilisant un code tel que celui-ci (voyez le fichier téléchargeable [GetExceptionArguments1.py](#)) :

```
import sys

try:
    File = open('myfile.txt')
except IOError as e:
    for Arg in e.args:
        print(Arg)
else:
    print("Le fichier a bien été ouvert.")
    File.close();
```

La propriété `args` contient toujours une liste des arguments de l'exception au format chaîne de caractères. Vous pouvez utiliser une simple boucle `for` pour imprimer chaque argument. Le seul problème avec cette approche est que vous n'obtenez pas le nom de ces arguments. Vous savez donc ce qu'ils contiennent, mais pas comment les appeler.

Une méthode plus complexe consiste à afficher à la fois le nom et la valeur des arguments. C'est ce que fait le code suivant (vous le retrouvez dans le fichier téléchargeable [GetExceptionArguments2.py](#)) :

```
import sys

try:
    File = open('myfile.txt')
except IOError as e:
    for Entry in dir(e):
        if (not Entry.startswith("_")):
            try:
                print(Entry, " = ", e.__getattr__(Entry))
            except AttributeError:
                print("Attribut ", Entry, " non accessible.")
else:
    print("Le fichier a bien été ouvert.")
    File.close();
```

Ici, vous commencez par obtenir une liste des attributs associés à l'objet argument de l'erreur en utilisant la fonction `dir()`. La sortie produite par cette fonction est une liste de chaînes de caractères contenant les noms des attributs affichables. Seuls les arguments dont le nom ne débute pas par un trait de soulignement (`_`) contiennent des informations utiles sur l'exception. Cependant, même certaines de ces entrées sont inaccessibles. C'est pourquoi vous devez placer la sortie de `print()` dans un second bloc `try...except` (voyez plus loin dans ce chapitre la section « Imbriquer des exceptions » pour plus de détails à ce sujet).

Le nom de l'attribut est simple à traiter, car il est contenu dans `Entry`. Pour obtenir la valeur correspondante, vous devez faire appel à la fonction `__getattr__` en lui fournissant le nom de l'attribut concerné. Lorsque vous exécutez ce code, vous voyez donc à la fois le nom et la valeur de chacun des attributs de l'argument pour cette exception particulière. Dans ce cas, la sortie produite par l'application se présenterait ainsi :

```
args = (2, 'No such file or directory')
Attribut characters_written non accessible.
errno = 2
filename = myfile.txt
filename2 = None
strerror = No such file or directory
winerror = None
with_traceback = <built-in method with_traceback of
FileNotFoundError object at 0x031810C0>
```

Gérer de multiples exceptions avec une seule clause except

La plupart des applications sont capables de générer

plusieurs exceptions pour une même ligne de code. Cette situation a déjà été rencontrée dans ce chapitre, notamment avec l'exemple `BasicException1.py`. La manière de gérer une telle situation dépend de ce que vous cherchez à obtenir, du type des exceptions et du degré de compétence éventuel de vos utilisateurs. Avec des utilisateurs plus ou moins novices, il est souvent préférable de leur dire que l'application a rencontré une erreur non récupérable, et d'enregistrer les détails dans un fichier journal placé dans le dossier de l'application ou bien dans une unité en réseau, ou encore envoyée vers un nuage Internet.



Utiliser une clause `except` unique pour gérer de multiples exceptions ne marche que s'il est possible de satisfaire aux besoins de tous les types d'exceptions à l'intérieur d'une source commune d'actions. Sinon, cela signifie que chaque exception doit être traitée individuellement. Les étapes qui suivent illustrent cette méthode de travail (cet exemple se trouve également dans le fichier téléchargeable `MultipleException1.py`).

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
try:
    Value = int(input("Tapez un nom entre 1 et 10: "))
except (ValueError, KeyboardInterrupt):
    print("Vous devez taper un nombre entre 1 et 10 !")
else:
    if (Value > 0) and (Value <= 10):
        print("Vous avez tapé : ", Value)
    else:
        print("La valeur que vous avez tapée est incorrecte !")
```



Ce code est très semblable à celui de `BasicException1.py`. Remarquez cependant que la clause `except` contient maintenant deux exceptions : `ValueError` et `KeyboardInterrupt`. De plus, ces exceptions sont placées entre parenthèses et séparées par une virgule.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application demande de taper un nombre entre 1 et 10.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application affiche un message d'erreur (reportez-vous à la [Figure 9.1](#)).

5. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 22.**

L'application affiche le message qui demande de respecter l'intervalle entier (reportez-vous à la [Figure 9.2](#)).

6. **Reprenez les Étapes 3 et 4, mais appuyez cette fois sur la combinaison Ctrl+C, ou Cmd+C, ou sur l'équivalent de votre propre système.**

Vous voyez le même message que pour une erreur de saisie (reportez-vous à la [Figure 9.1](#)).

7. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 7.**

L'application vous informe que vous avez tapé un chiffre qui se trouve bien dans l'intervalle demandé.

Gérer des exceptions multiples avec plusieurs clauses except

Lorsque vous travaillez avec des exceptions multiples, il est généralement préférable de placer chacune d'entre elles dans sa propre clause `except`. Cette approche vous permet de fournir un système de gestion personnalisé pour chaque exception, ce qui permet notamment à l'utilisateur de savoir plus précisément où se situe le problème. Bien entendu,

cette approche demande également plus de travail.

Les étapes qui suivent montrent comment procéder. Vous pouvez également vous reporter au fichier téléchargeable `MultipleException2.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
try:
    Value = int(input("Tapez un nombre entre 1 et 10 : "))
except ValueError:
    print("Vous devez taper un nombre entre 1 et 10 !")
except KeyboardInterrupt:
    print("Vous avez appuyé sur Ctrl+C!")
else:
    if (Value > 0) and (Value <= 10):

        print("Vous avez tapé : ", Value)
    else:
        print("La valeur que vous avez tapée est incorrecte !")
```



Remarquez ici l'emploi de plusieurs clauses `except`. Chacune sert à gérer une exception différente. Vous pouvez combiner les deux techniques, avec certaines clauses `except` traitant une exception unique, et d'autres prenant en charge plusieurs exceptions. Python vous permet d'utiliser la méthode qui convient le mieux à votre situation.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application demande de taper un nombre entre 1 et 10.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application affiche un message d'erreur (reportez-vous à la [Figure 9.1](#)).

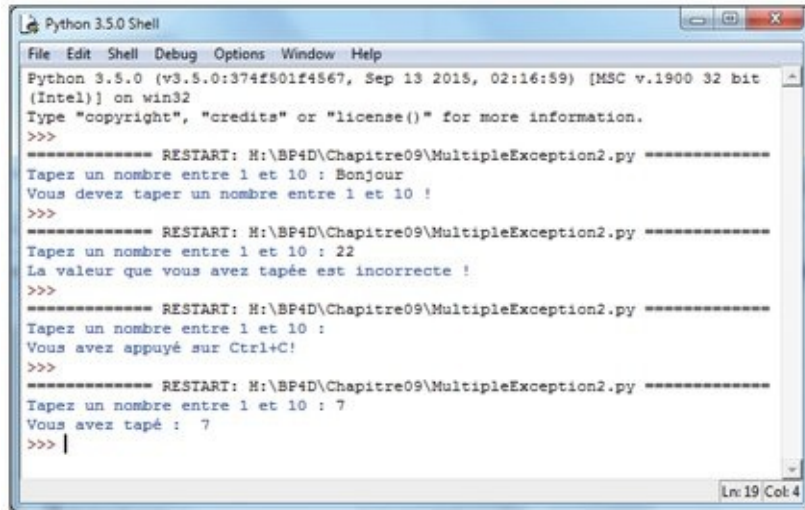
5. **Reprenez les Étapes 3 et 4, mais en tapant cette fois**

22.

L'application affiche le message qui demande de respecter l'intervalle entier (reportez-vous à la [Figure 9.2](#)).

Figure 9.6 :

Utiliser de multiples clauses `except` permet de personnaliser les messages d'erreur possibles.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException2.py =====
Tapez un nombre entre 1 et 10 : Bonjour
Vous devez taper un nombre entre 1 et 10 !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException2.py =====
Tapez un nombre entre 1 et 10 : 22
La valeur que vous avez tapée est incorrecte !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException2.py =====
Tapez un nombre entre 1 et 10 :
Vous avez appuyé sur Ctrl+C!
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException2.py =====
Tapez un nombre entre 1 et 10 : 7
Vous avez tapé : 7
>>> |
```

6. **Reprenez les Étapes 3 et 4, mais appuyez cette fois sur la combinaison Ctrl+C, ou Cmd+C, ou sur l'équivalent de votre propre système.**

Vous voyez maintenant un message spécifique qui indique à l'utilisateur où se situe le problème (voir la [Figure 9.6](#)).

7. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 7.**

L'application vous informe que vous avez tapé un chiffre qui se trouve bien dans l'intervalle demandé.

Gérer des exceptions en allant du plus spécifique au moins spécifique

Une stratégie de gestion des exceptions consiste à fournir des clauses `except` spécifiques pour toutes les exceptions connues, et des clauses `except` génériques pour toutes les exceptions non prévisibles ou inconnues. La documentation en ligne de Python

détaille la hiérarchie des exceptions (voyez l'adresse <https://docs.python.org/3.5/library/exceptions.html>).

Vous y remarquez par exemple que `BaseException` est l'exception de plus haut niveau. La plupart des exceptions sont dérivées de `Exception`. Si vous travaillez sur des opérations mathématiques, vous pouvez utiliser l'exception générique `ArithmeticError`, ou la plus spécifique `ZeroDivisionError`.

Python évalue les clauses `except` dans l'ordre où elles apparaissent dans le fichier du code source. La première rencontrée est examinée en premier, la seconde vient après, et ainsi de suite. Les étapes qui suivent vous aident à examiner un exemple destiné à démontrer l'importance qu'il y a à utiliser un ordre correct pour le traitement des exceptions. Vous le trouverez également dans le fichier téléchargeable

`MultipleException3.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
try:
    Value1 = int(input("Tapez le premier nombre : "))
    Value2 = int(input("Tapez le second nombre : "))
    Output = Value1 / Value2
except ValueError:
    print("Vous devez taper un nombre entier !")
except KeyboardInterrupt:
    print("Vous avez appuyé sur Ctrl+C !")
```

```
except ArithmeticError:
    print("Une erreur de math non définie s'est produite.")
except ZeroDivisionError:
    print("Tentative de division par zéro !")
else:
    print(Output)
```

Le code commence par vous demander d'entrer deux valeurs : `Value1` et `Value2`. Les deux premiers `except` gèrent une saisie inattendue. Les deux suivants prennent en charge

une exception mathématique, comme une division par zéro. Si tout s'est bien déroulé, le code se contente d'afficher le résultat de l'opération.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application demande de taper un premier nombre.

4. **Tapez Bonjour et appuyez sur Entrée.**

L'application affiche évidemment le message d'erreur `ValueError`.

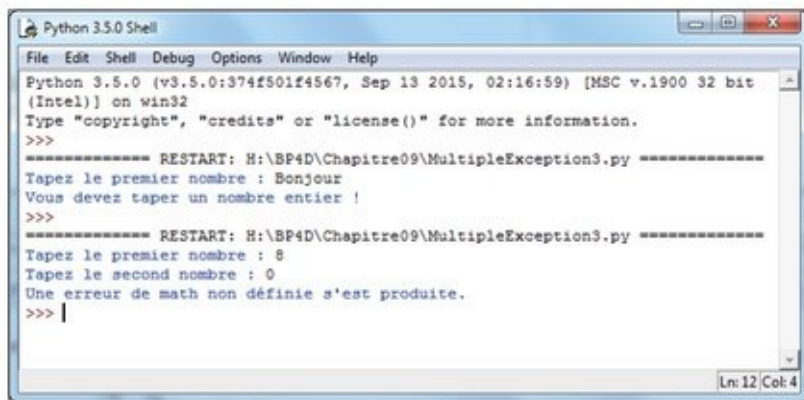
5. **Reprenez les Étapes 3 et 4, mais en tapant cette fois 8.**

Quand vous appuyez sur Entrée, l'application vous demande un second nombre.

6. **Tapez 0 et appuyez sur Entrée.**

Figure 9.7 :

L'ordre dans lequel Python traite les exceptions est important.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException3.py =====
Tapez le premier nombre : Bonjour
Vous devez taper un nombre entier !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException3.py =====
Tapez le premier nombre : 8
Tapez le second nombre : 0
Une erreur de math non définie s'est produite.
>>> |
```

Vous voyez cette fois le message d'erreur émis par l'exception `ArithmeticError` (voir la [Figure 9.7](#)). Cependant, ce n'est pas la bonne option. Vous devriez à la place voir le message d'erreur de l'exception `ZeroDivisionError`, qui est plus spécifique que `ArithmeticError`.

7. **Inversez l'ordre des deux dernières exceptions pour qu'elles se présentent ainsi :**

```
except ZeroDivisionError:
    print("Tentative de division par zéro !")
except ArithmeticError:
    print("Une erreur de math non définie s'est produite.")
```

8. **Reprenez les Étapes 5 et 6 ci-dessus.**

Cette fois, vous voyez s'afficher le message de l'exception `ZeroDivisionError`, qui vous indique que la division par zéro est interdite.

9. **Reprenez les Étapes 5 et 6 ci-dessus, mais en tapant comme second chiffre 2 au lieu de 0.**

Cette fois, l'application affiche normalement le résultat de la division (voir la [Figure 9.8](#)).



Notez que le résultat de la division de 8 par 2 donne un nombre en virgule flottante. C'est la réponse standard, sauf si vous demandez explicitement une valeur entière en utilisant l'opérateur `//`.

Figure 9.8 :

Quand on est sur la bonne route, c'est bien indiqué...

A screenshot of a Windows command prompt window titled "Python 3.5.0 Shell". The window shows the execution of a Python script named "MultipleException3.py". The script prompts the user to enter two numbers. The first run shows an error because the second number is 0. The second run shows an error because the second number is 0. The third run shows the correct result 4.0 because the second number is 2. The window has a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The status bar at the bottom right shows "Ln: 17 Col: 4".

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException3.py =====
Tapez le premier nombre : Bonjour
Vous devez taper un nombre entier !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException3.py =====
Tapez le premier nombre : 8
Tapez le second nombre : 0
Tentative de division par zéro !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException3.py =====
Tapez le premier nombre : 8
Tapez le second nombre : 2
4.0
>>>
```

Imbriquer des exceptions

Vous aurez parfois besoin de placer un gestionnaire d'exception à l'intérieur d'un autre. Ce processus est appelé *imbrication*. Lorsque vous procédez ainsi, Python essaie d'abord de trouver un gestionnaire d'exception dans les niveaux les plus bas, puis il remonte vers les couches extérieures. Vous pouvez

empiler autant de niveaux que vous le souhaitez pour rendre votre code aussi sûr qu'il est possible.

L'une des raisons les plus courantes d'utilisation de cette technique est la construction d'une partie de code dans laquelle vous demandez à l'utilisateur de saisir quelque chose. Vous placez ensuite la réponse obtenue dans une boucle pour vous assurer que vous avez bien obtenu l'information demandée. C'est ce qu'illustrent les étapes qui suivent. Vous pouvez également ouvrir cet exemple depuis le fichier téléchargeable `MultipleException4.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
TryAgain = True

while TryAgain:

    try:
        Value = int(input("Tapez un nombre entier : "))
    except ValueError:
        print("Vous devez taper un nombre entier !")

    try:
        DoOver = input("Essayer à nouveau (o/n) ? ")
    except:
        print("OK, à la prochaine fois !")
        TryAgain = False
    else:
        if (str.upper(DoOver) == "N"):
            TryAgain = False

except KeyboardInterrupt:
    print("Vous avez appuyé sur Ctrl+C !")
```

```
print("À la prochaine fois !")
TryAgain = False
else:
    print(Value)
    TryAgain = False
```

Ce code commence par créer une boucle de saisie. Cette méthode est en fait assez courante, car vous ne voulez pas que l'application se termine chaque fois qu'il y a une erreur de saisie. Cette boucle est simplifiée, et, normalement, vous

devriez créer une fonction séparée pour ce code.

Lorsque la boucle démarre, l'application demande à l'utilisateur de taper un nombre entier. Il peut s'agir de n'importe quelle valeur entière. Si l'utilisateur entre un nombre qui n'est pas un entier, ou encore s'il appuie sur Ctrl+C, Cmd+C ou une autre combinaison de touches d'interruption, le code de gestion des exceptions se déclenche. Sinon, l'application affiche la valeur fournie par l'utilisateur, et la variable `TryAgain` est mise à la valeur `False`, ce qui termine la boucle.

Une exception `ValueError` peut se produire lorsque l'utilisateur fait une erreur. Comme vous ne connaissez pas la raison de celle-ci, vous devez demander à l'utilisateur s'il veut faire une nouvelle tentative. Bien entendu, répéter cette procédure pourrait générer une autre exception. Le bloc de code intérieur `try...except` gère cette seconde saisie.



Notez l'emploi de la fonction `str.upper()` à la suite de la question posée à l'utilisateur. Elle permet de répondre par o ou par O sans avoir à distinguer minuscule et majuscule. Chaque fois que vous demandez à l'utilisateur de répondre de cette manière, convertir ce qu'il tape en majuscule est toujours une bonne idée, car cela vous permet de n'effectuer qu'une seule comparaison et réduit le nombre d'erreurs potentielles.



L'exception `KeyboardInterrupt` affiche deux messages, puis quitte automatiquement la boucle en affectant à la variable `TryAgain` la valeur `False`. Cette exception ne se produit que si l'utilisateur appuie sur une certaine combinaison de touches destinée à terminer l'application. Dans ce cas, cela signifie très certainement que l'utilisateur ne veut pas continuer.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application demande de taper un nombre entier.

4. **Tapez Bonjour et appuyez sur Entrée.**

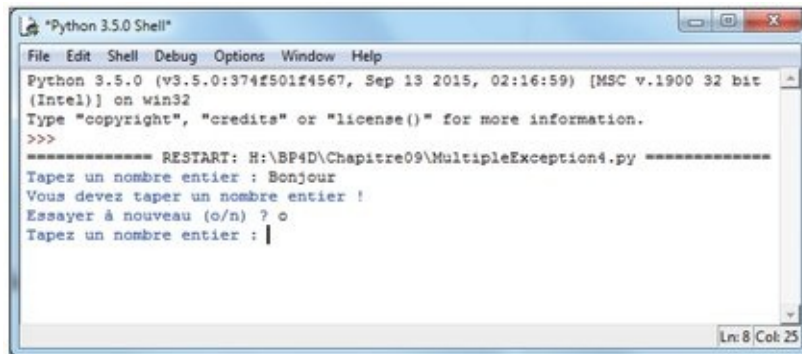
L'application affiche un message d'erreur et vous demande si vous voulez essayer à nouveau.

5. **Tapez O et appuyez sur Entrée.**

L'application vous demande à nouveau de taper un nombre entier (voir la [Figure 9.9](#)).

Figure 9.9 :

Utiliser une boucle permet de récupérer l'erreur de saisie.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException4.py =====
Tapez un nombre entier : Bonjour
Vous devez taper un nombre entier !
Essayer à nouveau (o/n) ? o
Tapez un nombre entier : |
```

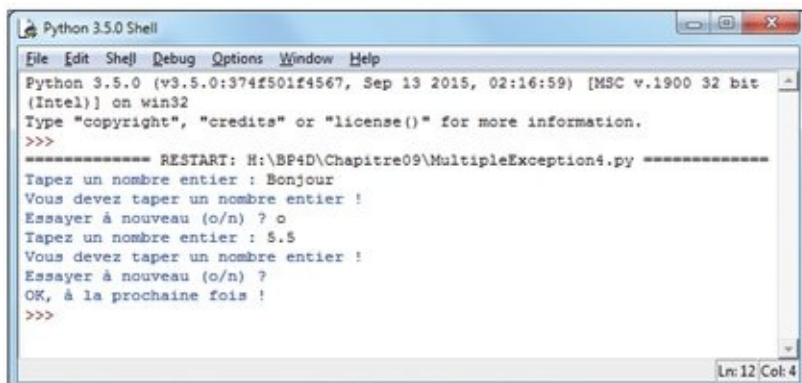
6. **Tapez 5.5 et appuyez sur Entrée.**

L'application affiche à nouveau son message d'erreur et vous demande si vous voulez essayer à nouveau.

7. **Appuyez sur la combinaison Ctrl+C, ou Cmd+C, ou sur l'équivalent de votre propre système.**

L'application se termine, comme l'illustre la [Figure 9.10](#). Notez que le message est émis par l'exception intérieure de la boucle. Dans ce cas, l'application n'atteint jamais le niveau supérieur, puisqu'elle a déjà répondu à l'exception générique.

Figure 9.10 : Le gestionnaire d'exception intérieur fournit le support pour la seconde saisie.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException4.py =====
Tapez un nombre entier : Bonjour
Vous devez taper un nombre entier !
Essayer à nouveau (o/n) ? o
Tapez un nombre entier : 5.5
Vous devez taper un nombre entier !
Essayer à nouveau (o/n) ? o
OK, à la prochaine fois !
>>>
```

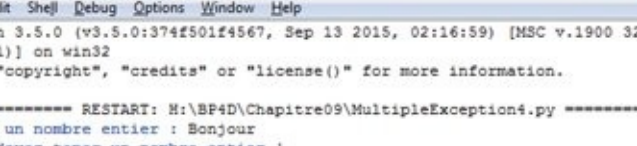

8. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application demande de taper un nombre entier.

9. Appuyez sur la combinaison **Ctrl+C**, ou **Cmd+C**, ou sur l'équivalent de votre propre système.

Là encore, l'application se termine. Mais remarquez que, cette fois, le message provient de l'exception extérieure (voir la [Figure 9.11](#)). Lors des Étapes 8 et 9, l'utilisateur quitte immédiatement l'application en appuyant sur la combinaison de touches engendrant une sortie immédiate. Cependant, l'application utilise deux gestionnaires d'exception différents pour traiter ce problème.

Figure 9.11 : Le
questionnaire
d'exception
extérieur fournit le
support pour la
première saisie.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException4.py =====
Tapez un nombre entier : Bonjour
Vous devez taper un nombre entier !
Essayer à nouveau (o/n) ? o
Tapez un nombre entier : 5.5
Vous devez taper un nombre entier !
Essayer à nouveau (o/n) ?
OK, à la prochaine fois !
>>>
===== RESTART: H:\BP4D\Chapitre09\MultipleException4.py =====
Tapez un nombre entier :
Vous avez appuyé sur Ctrl+C !
À la prochaine fois !
>>> |
```

Ln 17 Col 4

Lever des exceptions

Jusqu'ici, les exemples de ce chapitre ont réagi à des exceptions. Quelque chose se produit, et l'application fournit le support d'un gestionnaire d'exception pour traiter cet événement. Cependant, dans certaines circonstances, vous ne savez pas comment gérer tel ou tel événement problématique au cours du processus de développement. Par exemple, vous ne

pouvez pas traiter une erreur à un certain niveau, et vous devez donc la transmettre à un autre niveau. Dit autrement, il existe des situations dans lesquelles votre application doit générer une exception. On appelle souvent cela *lever une exception* (ou parfois *propager une exception*). Les sections qui suivent décrivent quelques scénarii courants dans lesquels vous avez à lever des exceptions de manière spécifique.

Lever des exceptions lors de conditions exceptionnelles

Cet exemple vous montre comment lever une exception simple, ne réclamant rien de spécial. Les étapes qui suivent créent l'exception, puis la gèrent immédiatement. Cet exemple se retrouve également dans le fichier téléchargeable `RaiseException1.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
try:
    raise ValueError
except ValueError:
    print("Exception ValueError !")
```

Évidemment, vous n'allez jamais écrire du code tel que celui-ci. Mais ce petit exemple vous montre comment les choses se passent au niveau le plus basique. Dans ce cas, l'appel à `raise` apparaît à l'intérieur d'un bloc `try...except`. Il fournit simplement le nom de l'exception qui a été levée (ou propagée). Vous pouvez également fournir des arguments dans la sortie pour fournir des informations

complémentaires.



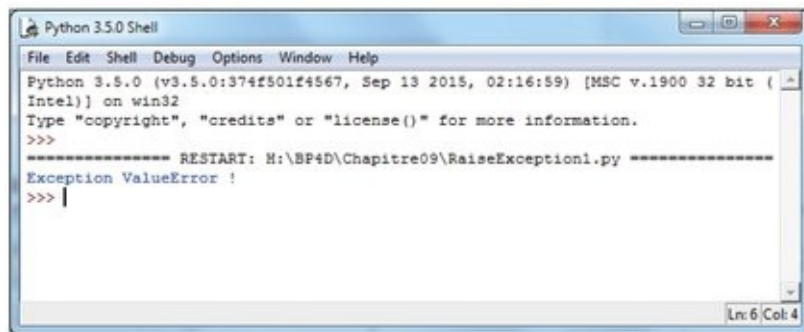
Remarquez que le bloc `try...except` ne contient pas de clause `else`, simplement parce qu'il n'y a rien de particulier à faire après l'appel. Même si cette forme est rare, elle est possible. La clause `else` est optionnelle, même si vous devriez pratiquement toujours en ajouter au moins une.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le message de l'exception (voir la [Figure 9.12](#)).

Figure 9.12 :

Lever une exception nécessite simplement un appel à `raise`.



Passer des informations sur une erreur à l'appelant

Python fournit un système de gestion des erreurs exceptionnellement souple en ce que vous pouvez passer des informations à l'*appelant* (le code qui appelle votre code) quelle que soit l'exception concernée. Bien entendu, cet appelant peut ne pas savoir que cette information est disponible, ce qui pose d'autres questions. Si vous travaillez avec le code de quelqu'un d'autre et que vous ne savez pas comment les choses se passent, vous pouvez toujours faire appel à la technique décrite plus haut dans

l'encadré « Obtenir la liste des arguments d'une exception ».

Vous pouvez vous demander s'il est possible de communiquer de meilleures informations lorsque vous travaillez avec l'exception `ValueError`, plutôt qu'avec une exception fournie nativement par Python. Les étapes qui suivent vous montrent comment modifier la sortie pour obtenir un tel résultat.

Vous pouvez également retrouver cet exemple dans le fichier téléchargeable `RaiseException2.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

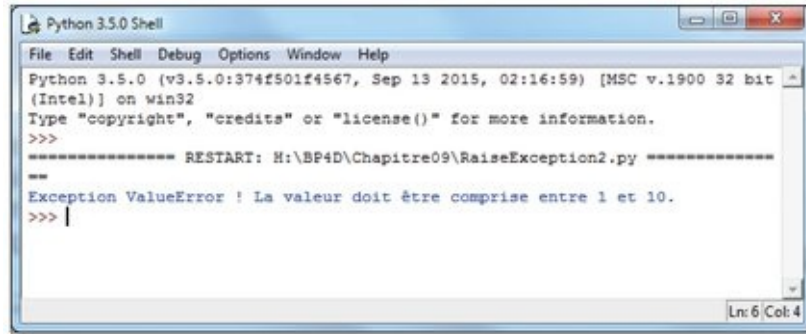
```
try:
    Ex = ValueError()
    Ex.strerror = "La valeur doit être comprise entre 1 et 10."
    raise Ex
except ValueError as e:
    print("Exception ValueError !", e.strerror)
```

L'exception `ValueError` ne fournit normalement pas un attribut appelé `strerror`, mais vous pouvez l'ajouter simplement en lui affectant une valeur comme dans cet exemple. Lorsque l'exception est levée, la clause `except` fonctionne comme d'habitude, si ce n'est qu'elle peut accéder à l'attribut via `e`, puis afficher l'information correspondante.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le nouveau message de l'exception (voir la [Figure 9.13](#)).

Figure 9.13 : Il est possible d'ajouter des informations sur une erreur à toute exception.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\RaiseException2.py =====
>>>
Exception ValueError ! La valeur doit être comprise entre 1 et 10.
>>> |
```

Créer et utiliser des exceptions personnalisées

Python fournit une grande quantité d'exceptions standard que vous devriez utiliser chaque fois qu'il est possible. Ces exceptions sont incroyablement souples, et vous pouvez même les modifier en fonction de vos besoins (dans des limites raisonnables et justifiées, bien entendu). Par exemple, la section précédente vous montre comment modifier une exception `ValueError` pour afficher des informations complémentaires. Cependant, il peut arriver qu'aucune exception standard ne réponde à votre problème. Par exemple, le nom de l'exception ne dit rien à l'utilisateur qui le renseigne sur ce qui se passe. Ou encore, vous travaillez sur une base de données spécifique, ou encore avec un certain service. Dans de tels cas, vous pouvez donc avoir besoin de créer une exception personnalisée.



L'exemple qui suit peut paraître un peu compliqué, car vous n'avez pas encore découvert la notion de classe. Nous y reviendrons en détail dans le Chapitre 14. Vous pouvez bien entendu passer cette section afin d'y revenir tranquillement plus tard.

L'exemple de cette section illustre une méthode rapide pour créer vos propres exceptions. Pour cela, vous devez créer une *classe* utilisant comme point de départ une exception existante. Pour rendre les choses un peu plus faciles, cet exemple crée une exception basée sur les fonctionnalités offertes par l'exception `ValueError`. L'avantage de cette approche, par rapport à celle proposée plus haut dans la section « Passer des informations sur une erreur à l'appelant », est qu'elle permet de dire à n'importe quelle personne qui vous suit ce qu'est l'ajout à l'exception `ValueError`. Elle rend également l'utilisation de l'exception modifiée plus simple. Cet exemple peut également être ouvert directement depuis le fichier téléchargeable `CustomException.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
class CustomValueError(ValueError):
    def __init__(self, arg):
        self.strerror = arg
        self.args = {arg}

try:
    raise CustomValueError("La valeur doit être comprise entre 1 et 10.")
except CustomValueError as e:
    print("Exception CustomValueError !", e.strerror)
```

Pour l'essentiel, vous retrouvez ici la fonctionnalité développée dans le code de la section « Passer des informations sur une erreur à l'appelant ». Cependant, elle place la même erreur à la fois dans `strerror` et dans `args`, de manière à ce que le développeur ait accès aux deux (comme cela devrait normalement se produire).

Le code commence par créer une classe appelée `CustomValueError`. Celle-ci prend comme point de départ la valeur de l'exception `ValueError`. La fonction `__init__()` fournit le moyen

de créer une nouvelle instance de cette classe. Vous pourriez à ce stade vous figurer une classe comme une sorte de plan d'architecte, et l'instance comme le bâtiment construit à partir de ce plan.



L'attribut `strerror` prend la valeur qui lui est directement affectée, tandis que `args` la reçoit sous forme d'une chaîne. Le membre `args` contient normalement un tableau de toutes les valeurs d'exception. Cette procédure est donc standard, même si dans le cas présent `args` ne contient qu'une seule valeur.

Le code servant à utiliser l'exception est ensuite considérablement plus simple que la modification directement de la valeur de `ValueError`. Tout ce que vous avez à faire, c'est d'appeler `raise` avec le nom de l'exception et les arguments que vous voulez lui passer. Tout cela tient en une ligne.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le message de l'exception personnalisée (voir la [Figure 9.14](#)).

Figure 9.14 : Les exceptions personnalisées facilitent la lecture de votre code.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre09\CustomException.py =====
Exception CustomValueError ! La valeur doit être comprise entre 1 et 10.
>>> |
```

Utiliser la clause finally

Normalement, vous voulez gérer une exception qui se produit sans provoquer un plantage de l'application. Cependant, il peut arriver que vous ne puissiez rien faire pour réparer les dommages. À ce stade, votre objectif devient donc de faire en quelque sorte atterrir l'application en douceur, en d'autres termes refermer les fichiers qui ont pu être ouverts et effectuer d'autres tâches de cette nature. Ceci permet d'éviter d'endommager des données ou de laisser le système dans un état instable. Ce qui est le minimum qu'on puisse demander à une application...

La clause `finally` fait partie de cette stratégie. Vous l'utilisez pour réaliser les tâches essentielles de dernière minute. Normalement, cette clause est assez courte et utilise uniquement des appels qui sont supposés pouvoir réussir sans poser davantage de problèmes. Il est essentiel de refermer les fichiers, de déconnecter l'utilisateur, d'effectuer quelques autres tâches ménagères, puis de laisser l'application se terminer avant qu'il n'arrive quelque chose d'absolument terrible (comme un crash violent du système). Cette nécessité étant posée, et bien mémorisée dans votre esprit, les étapes qui suivent vous proposent un exemple simple d'utilisation de la clause `finally`. Ce code est également disponible dans le fichier téléchargeable `ExceptionWithFinally.py`.

1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
import sys

try:
    raise ValueError
    print("Exception levée.")
except ValueError:
    print("Exception ValueError !")
    sys.exit()
finally:
    print("Prenez soin des détails de dernière minute.")

print("Ce code ne sera jamais exécuté.")
```

Dans cet exemple, le code lève une exception `ValueError`. La clause `except` s'exécute normalement dans ce genre de circonstances. L'appel à `sys.exit()` signifie que l'application se termine une fois l'exception gérée. Dans ce cas, la sortie de l'application s'effectue directement, ce qui fait que la fonction `print()` finale n'est jamais exécutée.



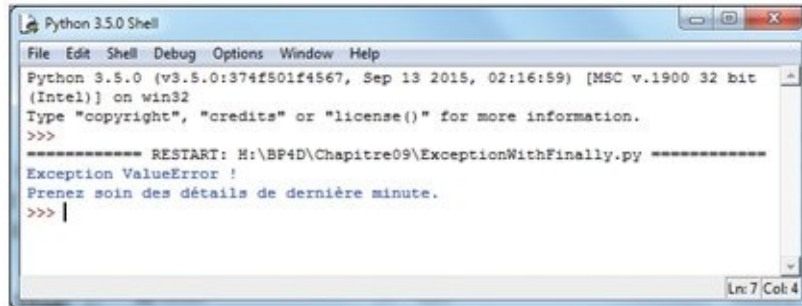
Le code associé à la clause `finally` est toujours exécuté. Le fait que l'exception se produise ou non n'a aucune importance dans ce cas. Le code que vous placez dans ce bloc doit donc avoir un caractère tout à fait général, puisque vous voulez qu'il soit exécuté systématiquement. Par exemple, si vous travaillez avec un fichier, vous placerez ici le code qui le referme pour vous assurer que les données qu'il contient seront en bon état sur le disque dur.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le message de la clause `except` ainsi que celui de la clause `finally` (voir la [Figure 9.15](#)). L'appel à `sys.exit()` empêche tout autre code de s'exécuter.

Figure 9.15 :

Utilisez la clause `finally` pour effectuer des actions spécifiques avant que l'application ne se termine.



4. **Transformez l'appel `raise ValueError` en commentaire en le faisant précéder de deux signes dièse, comme ceci :**

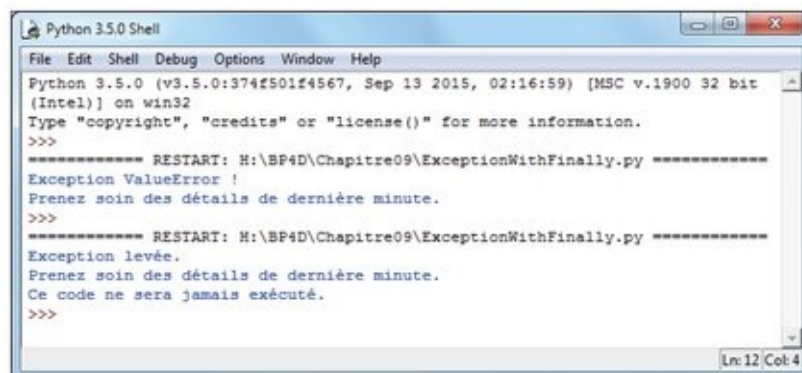
```
## raise ValueError
```

Supprimer l'exception permet de mieux montrer le fonctionnement de la clause `finally`.

5. **Sauvegardez le fichier sur le disque pour que Python voie le changement.**
6. **Choisissez la commande Run Module dans le menu Run.**

L'application affiche une série de messages, y compris celui de la clause `finally`, comme l'illustre la [Figure 9.16](#). Ceci prouve bien que la clause `finally` est toujours exécutée, ce qui explique pourquoi il faut l'utiliser avec un luxe de précautions.

Figure 9.16 : Il est essentiel de se rappeler que la clause `finally` est toujours exécutée.



Troisième partie

Effectuer des tâches courantes

Dans cette partie...

- ▶ Accéder aux modules de Python.
- ▶ Travailler avec les chaînes de caractères.
- ▶ Créer des listes d'objets.
- ▶ Utiliser des collections pour gérer les données efficacement.
- ▶ Développer des classes pour rendre le code réutilisable.

Chapitre 10

Interagir avec les modules

Dans ce chapitre :

- ▶ Organiser votre code.
- ▶ Ajouter à votre application du code provenant de sources extérieures.
- ▶ Localiser les bibliothèques de code sur votre disque dur.
- ▶ Regarder le code des bibliothèques.
- ▶ Accéder à la documentation en ligne de Python.

Les exemples de ce livre sont petits, mais les fonctionnalités qu'ils proposent sont également extrêmement limitées. Même des applications simples du monde réel contiennent facilement des milliers de lignes de code. En fait, les applications qui contiennent des *millions* de lignes de code sont relativement courantes. Imaginez maintenant que vous travaillez sur un fichier d'une taille suffisante pour comporter des millions de lignes de code. Vous n'arriverez très certainement jamais à rien, ou pas à grand-chose. Vous avez donc besoin d'organiser votre propre code en un ensemble de parties qui seront plus faciles à gérer, comme les exemples de ce livre. La

solution offerte par Python consiste à placer tout ce code dans des groupes appelés *modules*. Les modules disponibles via des sources externes, et qui contiennent du code répondant à des besoins génériques, sont appelés des *bibliothèques*.



Les modules sont stockés dans des fichiers séparés. Pour pouvoir utiliser un module, vous devez demander à Python de le charger dans l'application courante. Ce processus est appelé une *importation*. Vous importez donc un module ou une bibliothèque pour utiliser le code qu'il ou elle contient. Quelques exemples de ce livre ont déjà fait appel à cette technique en appelant l'instruction `import`. Dans ce chapitre, vous allez l'étudier plus en détail afin de savoir comment l'utiliser.

Lors de son démarrage, Python crée un pointeur vers les bibliothèques de portée générale dont il se sert. Dans ce cas, il vous suffit d'ajouter une instruction `import` suivie du nom d'une bibliothèque, et Python la retrouvera directement. Mais il est toujours utile de savoir comment localiser ces fichiers sur le disque, pour le cas où vous auriez besoin de les mettre à jour ou si vous voulez ajouter vos propres modules ou bibliothèques à la liste déjà copieuse des extensions de Python.

Le code des bibliothèques est généralement bien documenté. Certains développeurs pensent qu'ils n'ont pas besoin d'y jeter un coup d'œil, ce qui n'est pas totalement faux. Vous n'avez en effet pas besoin d'ouvrir ces fichiers pour vous en servir. Cependant, la lecture de ce code peut être intéressante pour comprendre son fonctionnement. De plus, cela peut vous aider à progresser et à découvrir des techniques

que vous n'auriez pas découvertes autrement. Ce genre de lecture est donc certes facultatif, mais cela peut grandement vous aider.

L'une des choses que vous avez besoin de savoir, c'est comment accéder à la documentation de Python concernant ses bibliothèques, et comment utiliser ces informations. Ce chapitre vous expliquera également ce qu'il faut connaître à ce sujet.

Créer des groupes de code

Il est important de regrouper des pièces de code de manière à rendre celui-ci plus facile à utiliser, à modifier et à comprendre. Plus une application prend de l'ampleur, et plus gérer tout ce code dans un seul fichier devient difficile. À un certain stade, cela devient même impossible tellement le fichier est devenu volumineux.



Le mot *code* est employé ici dans un sens très large. Le groupement de pièces de code peut concerner :

- ✓ Des classes
- ✓ Des fonctions
- ✓ Des variables
- ✓ Du code exécutable

La collection des classes, des fonctions, des variables et du code exécutable contenus dans un module est aussi connue sous le nom d'*attributs*. Un modèle possède des attributs auxquels vous pouvez accéder par leur nom. Les sections qui suivent précisent ce mode de fonctionnement.



Le code exécutable peut en fait être écrit dans un autre langage que Python. Par exemple, il est assez courant de trouver des modules qui sont écrits en C++. La raison principale est que cela peut rendre les applications Python plus rapides, moins gourmandes en ressources et mieux adaptées à des plates-formes variées. Cependant, cela rend en même temps les applications moins portables, à moins de disposer d'une version de ce code exécutable pour chaque plate-forme à laquelle votre application est destinée. De plus, ce « double langage » peut être plus difficile à gérer, puisque vous devez collaborer avec des développeurs capables de parler chacune des langues utilisées dans l'application.

La manière la plus courante de créer un module consiste à définir un fichier distinct qui contient le code que vous voulez séparer du reste de l'application. Par exemple, vous pourriez placer dans un module une routine d'impression dont une application se sert à de nombreuses reprises. Cette routine ne sert à rien seule, mais uniquement lorsque l'application en a besoin. Il est donc utile de la séparer du reste du code, et de plus intéressant de le faire puisque vous pourriez en avoir besoin dans d'autres applications. La possibilité de réutiliser du code est un des atouts majeurs des modules.

Pour rendre les choses plus faciles à comprendre, les exemples de ce chapitre font appel à un même module. Celui-ci ne fait rien de sensationnel. Il sert juste de base pour montrer les principes d'utilisation des modules. Tapez le code du [Listing 10.1](#), ou voyez le contenu du fichier téléchargeable `MyLibrary.py`.

Listing 10.1 : Un module de démonstration très simple.

```
def SayHello(Name):  
    print("Bonjour ", Name)  
    return  
  
def SayGoodbye(Name):  
    print("Au revoir ", Name)  
    return
```

Ce code contient deux fonctions très simples, l'une pour dire bonjour (`SayHello()`) et l'autre pour dire au revoir (`SayGoodbye()`). Dans les deux cas, vous fournissez juste un nom à afficher. Une fois son message affiché, chaque fonction renvoie le contrôle à l'appelant. Évidemment, les fonctions que vous allez créer seront bien plus complexes, mais celles-ci suffisent au propos de ce chapitre.

Importer des modules

Pour pouvoir utiliser un module, vous devez l'importer. Python place le code des modules avec le reste de l'application en mémoire, comme si vous aviez créé un très gros fichier. Rien n'est modifié dans le fichier tel qu'il est enregistré sur le disque. Toutes les sources restent bien distinctes. Seule change la manière dont Python « voit » le code.



Il y a deux façons d'importer des modules. Chaque technique est employée dans des circonstances spécifiques :

✍ `import` : Vous utilisez l'instruction `import` lorsque vous voulez importer un module entier. C'est la méthode la plus courante, car elle permet de gagner du temps et ne nécessite qu'une seule ligne de code. Cependant, cette approche consomme davantage de ressources mémoire que la technique suivante.

➤ `from...import` : Vous utilisez cette instruction lorsque vous voulez importer sélectivement des attributs spécifiques dans un module. Ceci permet de gagner des ressources, mais au prix d'une plus grande complexité. De plus, si vous essayez d'utiliser un attribut qui n'a pas été importé, Python provoquera une erreur. Certes, le module contient bien l'attribut voulu, mais Python ne peut pas le voir, car vous avez oublié de l'importer.

Changer le dossier courant de Python

Le dossier que Python utilise pour accéder au code influe sur les modules que vous pouvez importer. Les bibliothèques de Python lui-même sont toujours incluses dans la liste des emplacements auxquels il peut accéder, mais Python ne sait rien du dossier dont vous vous servez pour enregistrer votre code, à moins que vous ne lui indiquiez où il doit regarder. La méthode la plus simple consiste à modifier le dossier courant de Python pour qu'il pointe vers votre propre code. Pour cela, procédez ainsi :

1. **Ouvrez une fenêtre de Python en mode Shell.**
2. **Tapez `import os` et appuyez sur Entrée.**
Cette action importe la bibliothèque `os` de Python. Vous devez réaliser cette importation afin de modifier l'emplacement de travail utilisé pour ce chapitre.

3. Tapez maintenant `os.chdir('C :\\BP4D\\Chapitre10')` et appuyez sur Entrée.

Vous avez besoin d'utiliser le dossier qui contient les sources de votre propre projet sur votre disque local. L'appel ci-dessus utilise l'emplacement par défaut défini dans ce livre (voyez le Chapitre 4). Mais vous devrez peut-être modifier ce chemin en fonction de votre propre choix. Python peut maintenant accéder aux modules que vous allez créer dans ce chapitre.

Maintenant que vous avez une meilleure idée de l'importation des modules, il est temps d'y regarder de plus près. Les sections qui suivent vont vous aider à vous familiariser avec les deux techniques d'importation de Python.

Utiliser l'instruction `import`

L'instruction `import` est la méthode la plus courante pour importer un module dans Python. Cette approche est rapide, et elle s'assure que la totalité du contenu du module est disponible dans Python. Voyons de plus près comment les choses se passent :

- 1. Ouvrez une fenêtre Python en mode Shell.**
- 2. Faites pointer Python vers le dossier qui contient votre code source.**
Reportez-vous à l'encadré « Changer le dossier courant de Python ».
- 3. Tapez `import MyLibrary` et appuyez sur Entrée.**
Python importe le contenu du fichier `MyLibrary.py`, celui que vous avez créé plus haut dans ce chapitre. Ce module est

maintenant prêt à être utilisé.



Il est important de savoir que Python crée aussi un cache où il stocke ce module. Ce cache est enregistré dans un sous-dossier appelé `__pycache__`. Ouvrez le dossier de votre application, et vous pourrez remarquer sa présence. Par contre, vous devrez le supprimer si vous voulez éditer votre module. Sinon, Python continuera à utiliser la version qu'il a mise en cache, et celle que vous avez mise à jour.

4. **Tapez `dir(MyLibrary)` et appuyez sur Entrée.**

Vous allez voir un listing correspondant au contenu du module, dont notamment les fonctions `SayHello()` et `SayGoodbye()` (voir la [Figure 10.1](#)). Nous reviendrons sur les autres entrées affichées ici dans la section « Voir le contenu d'un module », plus loin dans ce chapitre.

5. **Tapez `MyLibrary.SayHello('Jean-Pierre')` et appuyez sur Entrée.**

La fonction `SayHello()` affiche le texte attendu (voir la [Figure 10.2](#)).

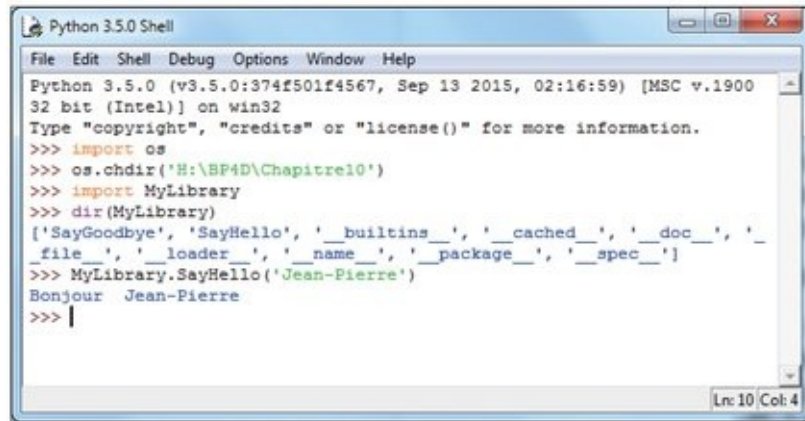


Vous devez faire précéder le nom de l'attribut, ici `SayHello()`, de celui du module, `MyLibrary`. Les deux éléments sont séparés par un point. Tout appel de ce type doit suivre ce schéma.

Figure 10.1 : La fonction `dir()` permet de vérifier que Python a bien importé le contenu du module.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('H:\BP4D\Chapitre10')
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__', '__
_file__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

Figure 10.2 : La fonction SayHello() vous prie le bonjour !



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('H:\BP4D\Chapitre10')
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__', '__
_file__', '__loader__', '__name__', '__package__', '__spec__']
>>> MyLibrary.SayHello('Jean-Pierre')
Bonjour Jean-Pierre
>>> |
```

6. Tapez maintenant `MyLibrary.SayGoodbye('Marie')` et appuyez sur Entrée.
Là encore, la fonction affiche le texte attendu.
7. Refermez la fenêtre de Python.

Utiliser l'instruction from... import

L'instruction `from...import` offre l'avantage de n'importer que les attributs d'un module dont vous avez besoin. Elle consomme donc moins de mémoire et de ressources système que l'instruction `import`. De plus, elle rend l'utilisation du module un peu plus facile, car certaines commandes, telles que `dir()`, montrent moins d'informations, ou seulement celles dont vous avez vraiment besoin. Vous obtenez donc uniquement ce que vous voulez et rien d'autre. Les étapes qui suivent explicitent l'emploi de l'instruction `from...import`.

1. Ouvrez une fenêtre Python en mode Shell.
2. Faites pointer Python vers le dossier qui contient votre code source.
Reportez à l'encadré « Changer le dossier courant de Python ».
3. Tapez `from MyLibrary import SayHello` et appuyez sur

Entrée.

Python importe la fonction `SayHello()` du module `MyLibrary`, et uniquement celle-ci.



Vous pouvez toujours importer la totalité du module si c'est ce que vous voulez. Les deux techniques pour obtenir ce résultat consistent soit à créer une liste d'attributs à importer en séparant leur nom par une virgule (comme dans `MyLibrary import SayHello, SayGoodbye`), soit à utiliser le caractère de substitution `*` pour remplacer un nom quelconque.

4. Tapez `dir(MyLibrary)` et appuyez sur Entrée.

Python va afficher un message d'erreur (voir la [Figure 10.3](#)). Il n'importe en effet que l'attribut que vous avez spécifié, et non le module entier. Par conséquent, le module **MyLibrary** n'est pas en mémoire. Seuls les attributs indiqués le sont.

5. Tapez `dir(SayHello)` et appuyez sur Entrée.

Cette fois, vous voyez une liste d'attributs qui sont associés à la fonction `SayHello()`, comme l'illustre la [Figure 10.4](#). À ce stade, il n'est pas important de comprendre comment ces attributs fonctionnent, mais vous en utiliserez certainement plus tard dans ce livre.

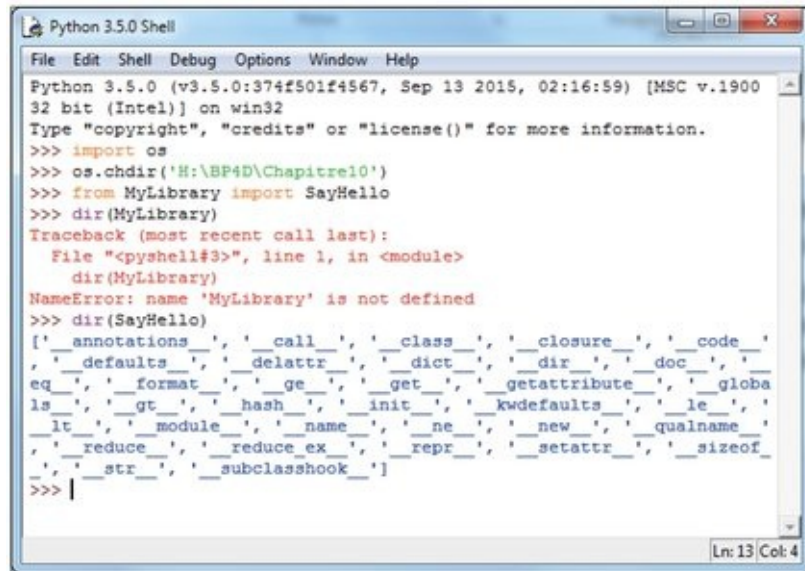
Figure 10.3 :

L'instruction `from... import` importe uniquement les éléments que vous avez spécifiquement demandés.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('H:\BP4D\Chapitre10')
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> |
```

Figure 10.4 :

Utilisez la fonction `dir()` pour obtenir des informations sur les attributs spécifiques que vous importez.

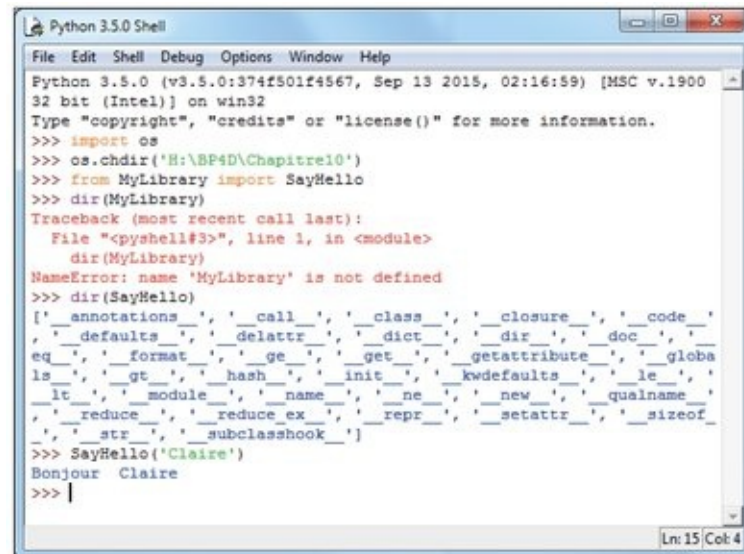


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('H:\BP4D\Chapitre10')
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> dir(SayHello)
['_annotations_', '_call_', '_class_', '_closure_', '_code_',
 '_defaults_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_',
 '_format_', '_ge_', '_get_', '_getattr_', '_global_',
 '_ls_', '_gt_', '_hash_', '_init_', '_kwdefaults_', '_le_',
 '_lt_', '_module_', '_name_', '_ne_', '_new_', '_qualname_',
 '_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_sizeof_',
 '_str_', '_subclasshook_']
>>> |
```

6. Tapez `SayHello('Claire')` et appuyez sur Entrée.

La fonction `SayHello()` produit le résultat escompté (voir la [Figure 10.5](#)).

Figure 10.5 : La fonction `SayHello()` ne nécessite plus le nom du module.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('H:\BP4D\Chapitre10')
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> dir(SayHello)
['_annotations_', '_call_', '_class_', '_closure_', '_code_',
 '_defaults_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_',
 '_format_', '_ge_', '_get_', '_getattr_', '_global_',
 '_ls_', '_gt_', '_hash_', '_init_', '_kwdefaults_', '_le_',
 '_lt_', '_module_', '_name_', '_ne_', '_new_', '_qualname_',
 '_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_sizeof_',
 '_str_', '_subclasshook_']
>>> SayHello('Claire')
Bonjour Claire
>>> |
```



Lorsque vous importez des attributs en utilisant l'instruction `from...import`, il n'est pas nécessaire de faire précéder leur nom de celui du module d'où ils proviennent. Ceci facilite donc

leur accès.



L'emploi de l'instruction `from...import` peut également provoquer des problèmes. Si deux attributs possèdent le même nom, vous ne pouvez importer que l'un d'entre eux. L'instruction `import` évite ce genre de collision, ce qui est important si vous devez importer un grand nombre d'attributs.

7. **Tapez `SayGoodbye('Antoine')` et appuyez sur Entrée.**

Vous n'avez importé que la fonction `SayHello()`. Python ne sait donc rien de la fonction `SayGoodbye()`, et il affiche par conséquent un message d'erreur. La nature même de l'instruction `from...import` peut être une source de problèmes si vous n'avez pas pris toutes vos précautions et que vous supposez qu'un certain attribut est présent, alors qu'il n'y est pas.

8. **Refermez la fenêtre de Python.**

Trouver des modules sur le disque

Pour pouvoir utiliser le code d'un module, Python doit être capable de le localiser et de le charger en mémoire. Ces chemins d'accès sont enregistrés dans une liste gérée par Python. Chaque fois que vous avez besoin d'importer un module, Python fait une recherche dans les chemins d'accès qu'il connaît afin de le retrouver. Ces informations proviennent de trois sources :

➤ **Variables d'environnement** : Le Chapitre 3 vous explique ce dont il s'agit. En particulier la variable d'environnement `PYTHONPATH` indique à Python où rechercher des modules sur le disque.

➤ **Dossier courant** : Plus haut dans ce

chapitre, vous avez appris que vous pouviez modifier le dossier courant de Python de manière à ce qu'il puisse localiser les modules utilisés par votre application.

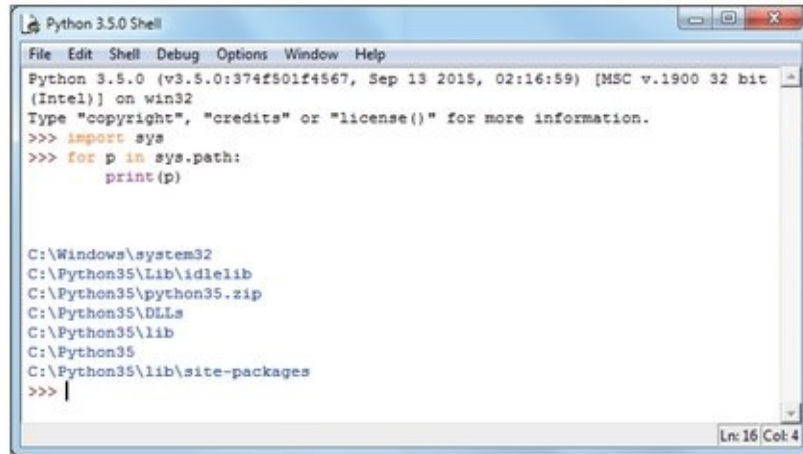
✓ **Dossiers par défaut** : Même si vous n'avez pas défini de variables d'environnement, et que le dossier courant ne contient pas de module utilisable, Python est tout de même capable de retrouver ses propres bibliothèques dans la liste des dossiers par défaut qui est incluse dans sa propre configuration.

Il est utile de connaître le ou les chemins d'accès courant, car une erreur dans une telle information, ou l'absence de celle-ci, peut faire échouer votre application. Voici comment procéder pour récupérer ces chemins d'accès courant :

1. **Ouvrez une fenêtre de Python en mode Shell.**
2. **Tapez `import sys` et appuyez sur Entrée.**
3. **Tapez `for p in sys.path` : et appuyez sur Entrée.**
Python indente automatiquement la nouvelle ligne en attendant la suite de la boucle `for`. L'attribut `sys.path` contient toujours une liste des chemins d'accès par défaut.
4. **Tapez `print(p)` et appuyez deux fois sur Entrée.**
Vous voyez s'afficher une liste de chemins d'accès (voir la [Figure 10.6](#)). Bien sûr, le résultat que vous obtenez diffère certainement de celui de la figure. Tout dépend de votre plate-forme, de la version de Python que vous utilisez, du dossier dans lequel Python est installé et des fonctionnalités que vous avez pu lui ajouter.

Figure 10.6 :

L'attribut `sys.path` contient une liste des chemins d'accès connus de Python.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> for p in sys.path:
>>>     print(p)

C:\Windows\system32
C:\Python35\Lib\idlelib
C:\Python35\python35.zip
C:\Python35\DLLs
C:\Python35\lib
C:\Python35
C:\Python35\lib\site-packages
>>> |
```

L'attribut `sys.path` est fiable, mais cela ne signifie pas qu'il sait tout sur les chemins d'accès connus de Python. Si vous constatez qu'une certaine information manque alors que vous pensez qu'elle devrait être présente, vous pouvez suivre ces étapes :

1. Tapez **`import os`** et appuyez sur Entrée.
2. Tapez **`os.environ['PYTHONPATH'].split(os.pathsep)`** maintenant et appuyez sur Entrée.



Une variable d'environnement appelée `PYTHONPATH` doit avoir été définie au préalable, à défaut de quoi vous obtiendrez un message d'erreur.

Si la variable `PYTHONPATH` existe, son contenu va s'afficher (voir la [Figure 10.7](#)).



L'attribut `sys.path` ne comprend pas de fonction `split()` (diviser). C'est pourquoi l'exemple utilise une boucle `for`. Par contre, l'attribut `os.environ['PYTHONPATH']` possède une telle fonction, ce qui permet de créer une liste de chemins individuels.



La fonction `split()` doit contenir une valeur permettant de découper une liste d'éléments en ses composants. La (autrement dit, une valeur prédéfinie, fixée une fois pour toutes) `os.pathsep` définit le séparateur entre ces différents éléments. Même si ce séparateur peut varier d'une plateforme à une autre, Python se charge de la question, ce qui vous permet d'utiliser partout ce même code.

3. Refermez la fenêtre de Python.



Vous pouvez également ajouter ou supprimer des éléments de `sys.path`. Si vous voulez ajouter les exemples du Chapitre 9 à votre liste de modules, vous tapez `sys.path.append(« C :\\BP4D\\Chapitre09 »)` et vous appuyez sur Entrée. Vous pourrez ensuite reprendre la boucle ci-dessus pour vérifier le résultat. De la même manière, il est aussi possible de supprimer un chemin d'accès en utilisant une syntaxe du genre `sys.path.remove(« C :\\BP4D\\Chapitre09 »)`.

Figure 10.7 : Vous devez demander des informations sur les variables d'environnement séparément.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> for p in sys.path:
>>>     print(p)
C:\Windows\system32
C:\Python35\Lib\idlelib
H:\BP4D\Chapitre10
C:\Python35\python35.zip
C:\Python35\DLLs
C:\Python35\lib
C:\Python35
C:\Python35\lib\site-packages
>>> import os
>>> os.environ['PYTHONPATH'].split(os.pathsep)
['H:\\BP4D\\Chapitre10']
>>> |
```

Voir le contenu d'un module

Python fournit plusieurs méthodes différentes pour voir le contenu d'un module. La plus courante est la fonction `dir()`, qui renvoie les attributs fournis par le module.

Si vous revenez à la [Figure 10.1](#), la liste contient bien d'autres entrées que nos fonctions `SayHello()` et `SayGoodbye()`. Ces attributs sont automatiquement générés par Python et ils servent à effectuer certaines tâches, ou à contenir certaines informations :

✓ `__builtins__` : Contient une liste de tous les attributs intégrés qui sont accessibles dans le module. Ce sont ces attributs qui sont automatiquement ajoutés par Python.

✓ `__cached__` : Vous indique le nom et l'emplacement du fichier mis en cache lorsque vous importez le module. Ce chemin d'accès est défini de façon relative par rapport au dossier courant de Python.

✓ `__doc__` : Affiche une information d'aide sur le module, du moins si celle-ci a été rédigée. Par exemple, si vous tapez `os.__doc__` et que vous appuyez sur Entrée, Python va afficher l'aide associée à la bibliothèque `os`.

✓ `__file__` : Vous indique le nom et l'emplacement du module. Le chemin d'accès est défini relativement au dossier courant de Python.

✓ `__initializing__` : Détermine si le module est en cours d'initialisation. Normalement, cet attribut renvoie la valeur `False`, puisque le module devrait déjà avoir terminé son initialisation à ce moment. Cet attribut peut cependant être utile si vous avez besoin d'attendre qu'un module ait terminé son chargement avant d'importer un

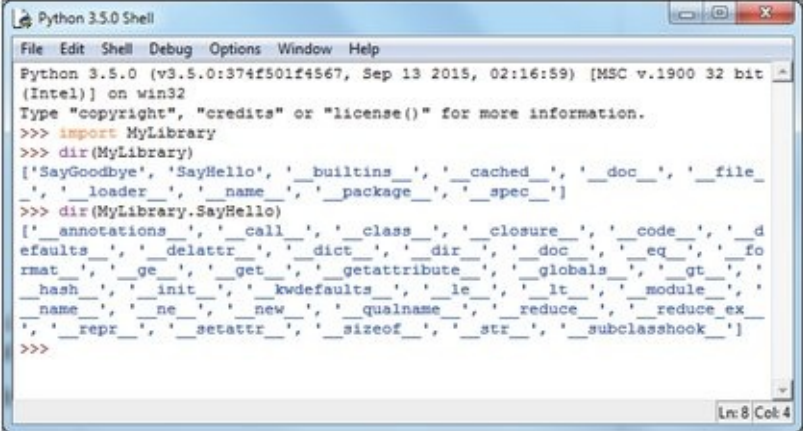
autre module dépendant du premier.

✓ `__loader__` : Affiche des informations sur le chargement du module. Vous ne devriez pas avoir à utiliser cet attribut.

✓ `__name__` : Rappelle simplement le nom du module.

✓ `__package__` : Sert uniquement au système d'importation pour faciliter le chargement et la gestion des modules. Vous n'avez pas à vous en soucier.

Figure 10.8 :
Plongez au cœur des modules pour mieux comprendre ce que vous utilisez dans Python.

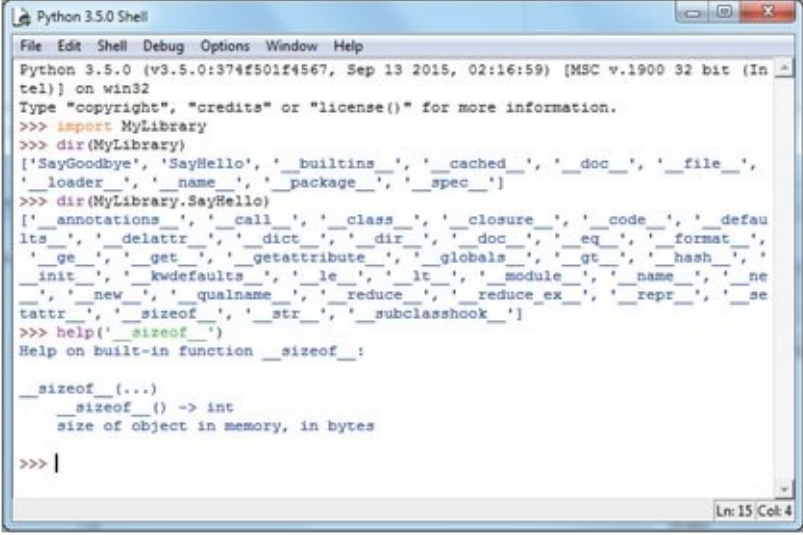


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '_builtins_', '_cached_', '_doc_', '_file_',
 '_loader_', '_name_', '_package_', '_spec_']
>>> dir(MyLibrary.SayHello)
['_annotations_', '_call_', '_class_', '_closure_', '_code_', '_d',
 'efaults_', '_delattr_', '_dict_', '_dir_', '_doc_', '_eq_', '_fo',
 'rmat_', '_ge_', '_get_', '_getattr_', '_globals_', '_gt_', '_',
 'hash_', '_init_', '_kwdefaults_', '_le_', '_lt_', '_module_', '_',
 'name_', '_ne_', '_new_', '_qualname_', '_reduce_', '_reduce_ex_',
 '_repr_', '_setattr_', '_sizeof_', '_str_', '_subclasshook_']
>>>
```

En fait, vous pouvez même plonger encore plus profondément dans la liste des attributs. Tapez **`dir(MyLibrary.SayHello)`** et appuyez sur Entrée. Vous voyez la sortie illustrée sur la [Figure 10.8](#).

Certaines de ces entrées, comme `__name__`, apparaissent aussi dans le listing du module. Cependant, certaines autres entrées peuvent attirer votre curiosité. Par exemple, vous voudriez savoir à quoi correspond `__sizeof__`. Pour cela, tapez `help(__sizeof__)` et appuyez sur Entrée. Vous obtenez une information d'aide très sommaire, mais utile (voir la [Figure 10.9](#)).

Figure 10.9 : Vous pouvez essayer d'obtenir de l'aide sur un certain attribut.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
>>> dir(MyLibrary.SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> help('__sizeof__')
Help on built-in function __sizeof__:

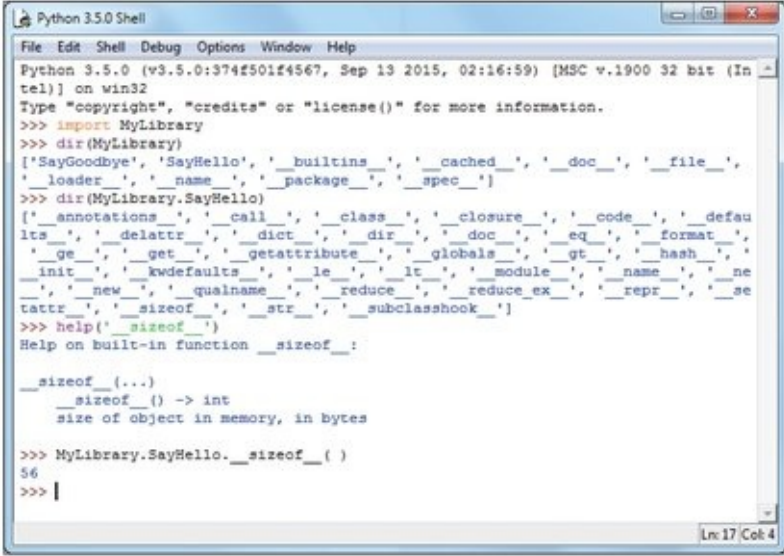
__sizeof__(...)
    __sizeof__() -> int
        size of object in memory, in bytes

>>> |
```

Python ne va pas vous manger si vous essayez l'attribut. Même si le Shell rencontre des problèmes, vous pouvez toujours ouvrir une autre fenêtre. Une autre manière de rechercher des informations sur un modèle consiste donc à tester simplement les attributs. Si vous tapez par exemple `MyLibrary.SayHello.__sizeof__()` et que vous appuyez sur Entrée, vous allez voir s'afficher la taille de la fonction `SayHello()` en octets, comme l'illustre la [Figure 10.10](#).

Contrairement à d'autres langages, Python rend public le code source de ses propres bibliothèques. Par exemple, si vous ouvrez le dossier `\Python35\Lib`, vous y trouverez toute une série de fichiers.py que vous pouvez ouvrir dans IDLE sans aucun problème. Essayez par exemple d'ouvrir la bibliothèque `os.py` déjà utilisée à plusieurs reprises dans ce chapitre. Vous allez voir s'afficher le contenu illustré sur la [Figure 10.11](#).

Figure 10.10 :
Utiliser les attributs
vous aide à mieux
comprendre leur
rôle et leur
fonctionnement.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', 'cached', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__']
>>> dir(MyLibrary.SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__default__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> help('__sizeof__')
Help on built-in function __sizeof__:

__sizeof__(...)
    __sizeof__() -> int
        size of object in memory, in bytes

>>> MyLibrary.SayHello.__sizeof__()
56
>>> |
```

Voir directement le contenu d'une bibliothèque peut vous aider à découvrir de nouvelles techniques de programmation et à mieux comprendre le fonctionnement de Python. Plus vous y passerez de temps, et mieux vous serez à même de construire des applications intéressantes.



Contentez-vous de consulter le contenu des bibliothèques, et surtout n'y modifiez rien ! Si vous changiez accidentellement des morceaux de code, vous pourriez introduire des bogues dans votre application. Ces problèmes apparaîtraient uniquement sur votre propre système, et pas sur d'autres plateformes. La prudence la plus extrême est donc de rigueur.

Figure 10.11 :
Voir directement un module peut vous aider à mieux le comprendre.



```
os.py - C:\Python35\Lib\os.py (3.5.0)
File Edit Format Run Options Window Help

"""OS routines for NT or Posix depending on what system we're on.

This exports:

- all functions from posix, nt or os, e.g. unlink, stat, etc.
- os.path is either posixpath or ntpath
- os.name is either 'posix', 'nt' or 'os'.
- os.curdir is a string representing the current directory ('.' or '..')
- os.pardir is a string representing the parent directory ('..' or '..')
- os.sep is the (or a most common) pathname separator ('/' or '\\' or '\\')
- os.extsep is the extension separator (always '.')
- os.altsep is the alternate pathname separator (None or '/')
- os.pathsep is the component separator used in $PATH etc
- os.linesep is the line separator in text files ('\n' or '\n' or '\r\n')
- os.defpath is the default search path for executables
- os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
"""

#
import sys, errno
import stat as st

__names = sys.builtin_module_names

# Note: more names are added to __all__ later.
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep", "linesep",
           "defpath", "name", "path", "devnull", "SEEK_SET", "SEEK_CUR",
           "SEEK_END", "Iscodes", "Iscodes", "get_exec_path", "fdopen",
           "popen", "extsep"]

def _exists(name):
    return name in globals()

def _get_exports_list(module):
    try:
```

Utiliser la documentation des modules de Python

Vous pouvez utiliser la fonction `doc()` chaque fois que vous en avez besoin pour obtenir de l'aide. Cependant, il existe un meilleur moyen d'étudier les modules et les bibliothèques fournis avec Python : c'est la documentation de celui-ci. Celle-ci apparaît parfois sous le nom Module Docs dans le dossier de Python. On y fait aussi référence sous le nom de `pydoc`. Mais quelle que soit l'appellation, cette documentation est une aide précieuse pour les développeurs. Voyons donc cela de plus près.

Ouvrir l'application *pydoc*

Pydoc est tout simplement une autre application

Python. Elle se trouve dans le dossier `\Python35\Lib` de votre système sous le nom de `pydoc.py`. Comme tout autre fichier `.py`, vous pouvez l'ouvrir dans IDLE et étudier son fonctionnement. Une autre manière d'y accéder consiste à choisir le raccourci `Python 3.5 Module Docs` de votre installation Python, ou encore à taper une commande en mode Invite de commandes ou Terminal.

L'application crée un serveur localisé qui fonctionne avec votre navigateur Web par défaut afin d'afficher des informations sur les modules et les bibliothèques de Python. Lorsque vous la lancez, vous voyez une Invite de commandes (ou un Terminal) semblable à l'illustration de la [Figure 10.12](#).

Figure 10.12 :

Lancer `pydoc` signifie ouvrir une fenêtre de commandes (un Terminal) pour démarrer le serveur.



Comme pour tout serveur, votre système peut vous demander une autorisation d'accès. Par exemple, votre pare-feu peut vous signaler qu'un programme inconnu tente d'accéder au système local. Vous devez alors valider la permission demandée pour accéder aux informations fournies par `pydoc`. Il en va de même le cas échéant pour votre antivirus. Sous Windows, et d'autres plates-formes, vous aurez donc peut-être besoin d'élever votre niveau de privilèges.

Normalement, le serveur ouvre automatiquement une

Figure 10.13 :
Votre navigateur Web affiche de nombreux liens dans la page d'index produite par pydoc.



L'Invite de commandes/Terminal offre deux commandes pour contrôler le serveur. Il vous suffit pour cela de taper la lettre correspondante :

- **b** : Ouvre une nouvelle copie du navigateur Web par défaut en y chargeant la page d'index.
- **q** : Stoppe le serveur.



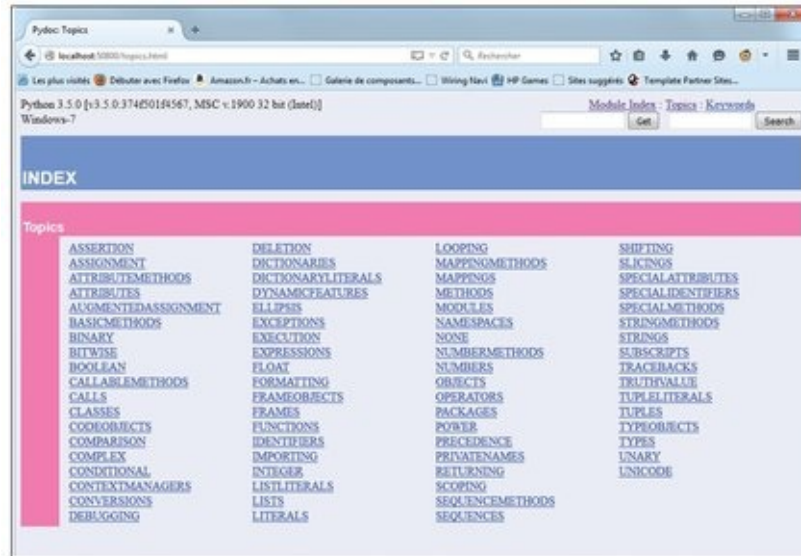
Lorsque vous avez terminé votre navigation, n'oubliez pas de taper **q** et d'appuyer sur Entrée pour refermer le serveur. Celui-ci va libérer les ressources qu'il consommait, et refermer tous les trous qu'il aurait pu provoquer dans votre pare-feu.

Utiliser les liens d'accès rapide

Si vous observez le haut de la [Figure 10.13](#), vous allez remarquer trois liens sur la droite. Ces liens fournissent un accès rapide aux fonctions du site. La première page affichée est toujours celle qui correspond au lien Module Index. Si vous avez besoin de revenir à cette page, il vous suffit donc à tout instant de cliquer sur ce lien.

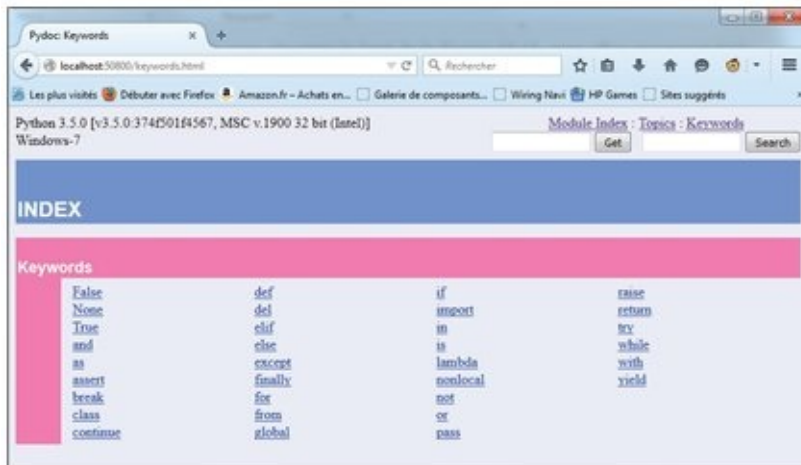
Le lien Topics ouvre la page illustrée sur la [Figure 10.14](#). Elle propose des liens vers des sujets essentiels de Python. Si vous voulez en savoir plus sur les valeurs booléennes, par exemple, cliquez sur le lien BOOLEAN. La page qui va s'afficher alors décrit la manière dont les valeurs booléennes fonctionnent dans Python. En bas de cette page, vous trouvez d'autres liens permettant d'accéder à des informations complémentaires utiles.

Figure 10.14 : La page Topics vous donne accès à des informations sur les sujets essentiels de Python.



Le lien Keyword donne accès à la page illustrée sur la [Figure 10.15](#). Vous y voyez une liste de mots-clés supportés par Python. Si vous voulez par exemple en savoir plus sur la création de boucles `for`, cliquez sur ce lien.

Figure 10.15 : La page Keywords contient une liste de mots-clés supportés par Python.



Taper un terme à rechercher

Les pages contiennent également deux champs de texte, en haut et vers la droite. Le premier est suivi

d'un bouton Get, et le second d'un bouton Search. Lorsque vous entrez un terme à rechercher dans le premier champ et que vous cliquez sur le bouton Get, vous obtenez la documentation qui concerne ce module ou cet attribut particulier. La [Figure 10.16](#) illustre ce comportement. Pour voir ce qui se passe, tapez **print** et cliquez sur Get.

Si vous tapez un critère dans le second champ et que vous cliquez sur le bouton Search, vous obtenez cette fois une liste de tous les sujets qui peuvent avoir un rapport avec ce terme. Essayez par exemple de taper **print** et de cliquer sur Search. La [Figure 10.17](#) illustre le résultat obtenu. Vous pouvez ensuite cliquer sur un autre lien, par exemple calendar, si vous voulez consulter des informations techniques sur la gestion de l'affichage des dates.

Figure 10.16 : Get permet d'obtenir des informations spécifiques sur le terme recherché.

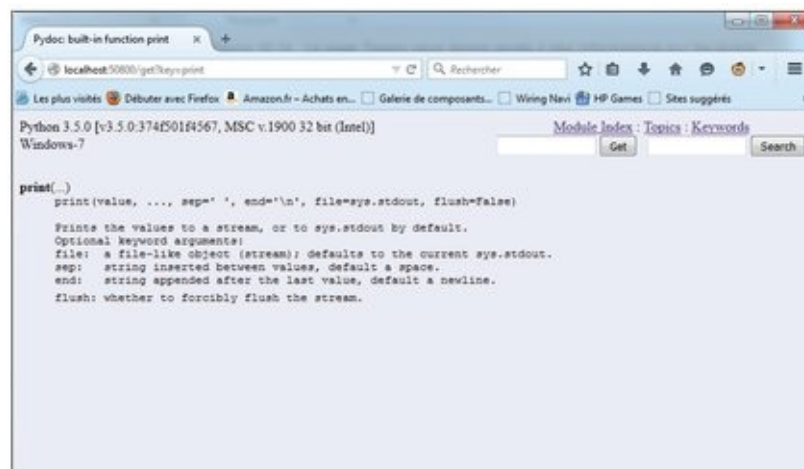
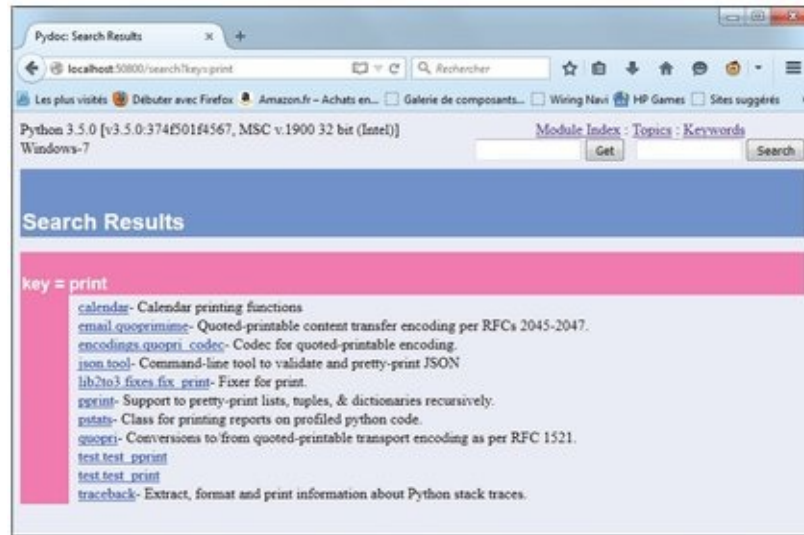


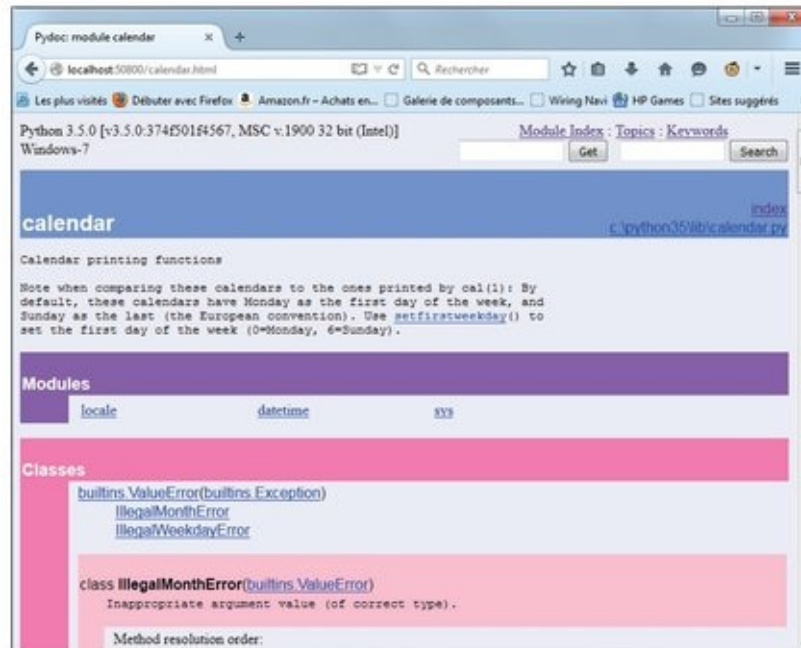
Figure 10.17 : Le bouton Search permet d'obtenir une liste de sujets liés au terme recherché.



Voir les résultats

Les réponses que vous obtenez peuvent varier considérablement. Certains sujets sont brefs, comme dans le cas de la [Figure 10.16](#), et donc peuvent facilement être lus, et même imprimés. Mais d'autres sont bien plus développés. Si vous cliquez par exemple sur le lien [calendar](#), visible sur la [Figure 10.17](#), vous allez voir s'afficher une page plutôt copieuse (voir la [Figure 10.18](#)).

Figure 10.18 :
Certaines pages contiennent des informations extrêmement copieuses.



Dans ce cas particulier, vous voyez des informations qui concernent le module, les erreurs gérées, ses classes, ses fonctions, ses données, et ainsi de suite. Bien entendu, la masse de renseignements à laquelle vous pouvez accéder dépend en partie de la complexité du sujet, et en partie du niveau de description fourni par le créateur du module. Si vous faites l'essai depuis la page Module Index avec le module MyLibrary utilisé dans ce chapitre, vous ne verrez qu'une liste de ses fonctions sans aucune description.



Bien documenter vos modules est important non seulement pour vous, mais aussi pour tous ceux qui pourraient avoir besoin de les utiliser plus tard.

Chapitre 11

Travailler avec les chaînes de caractères

Dans ce chapitre :

- ▶ Prendre en compte les différences entre les chaînes.
- ▶ Utiliser des caractères spéciaux dans les chaînes.
- ▶ Travailler avec des caractères uniques.
- ▶ Effectuer des tâches spécifiques sur les chaînes.
- ▶ Trouver ce dont vous avez besoin dans les chaînes.
- ▶ Modifier l'apparence de sortie des chaînes.

Votre ordinateur ne comprend rien aux chaînes de caractères. C'est juste un fait. Les ordinateurs connaissent les nombres, pas les chaînes. Si vous voyez une suite de caractères à l'écran, lui ne voit que des valeurs numériques. Cependant, les êtres humains, eux, voient des textes et vos applications doivent être capables de convertir les uns dans les autres. Heureusement, Python facilite beaucoup ce genre de travail. Il transcrit vos chaînes de caractères

en valeurs compréhensibles par l'ordinateur, et réciproquement.

Pour que vos chaînes de caractères soient utiles, vous devez pouvoir les manipuler. Bien entendu, cela signifie que vous pouvez les décomposer ou encore rechercher une certaine information. Ce chapitre décrit donc les méthodes dont vous disposez pour construire des chaînes de caractères, les décomposer, ou encore en utiliser les parties dont vous avez besoin. Tout cela constitue une partie importante de vos applications, car vos utilisateurs dépendent énormément de ce genre de travail (même si leur ordinateur n'a aucune idée particulière de ce qui se passe).

Une fois que votre chaîne est correctement mise en forme, vous devez la présenter d'une manière plaisante. Là encore, ce n'est pas le problème de l'ordinateur, mais le vôtre. Savoir comment formater la présentation des chaînes de caractères est important, car vos utilisateurs ont besoin de voir des informations dans un format qu'ils sont capables de lire. Lorsque vous aurez terminé la lecture de ce chapitre, vous devriez pouvoir créer, manipuler et formater des chaînes de caractères de manière à ce que vos utilisateurs puissent lire les bonnes informations au bon moment.

Comprendre que les chaînes sont différentes

La plupart des aspirants développeurs (et même des moins aspirants) ont parfois du mal à comprendre que les ordinateurs ne connaissent que deux choses : les 0

et les 1. Même les nombres les plus grands ne sont faits que de cela, de 0 et de 1. Les comparaisons n'utilisent rien d'autre. Pas plus que les modifications dans les données. En résumé, un ordinateur ne sait rien de ce qu'est une chaîne de caractères. C'est pourquoi ce sujet est un peu plus compliqué que la plupart des autres...



L'informatique ne sait rien des chaînes. Celles-ci sont faites de séquences de caractères individuels qui sont en réalité des valeurs numériques du point de vue de l'ordinateur. Le fait de travailler avec Python n'y change rien. C'est pourquoi les sections qui suivent sont aussi importantes. Elles vous aideront à mieux comprendre en quoi les chaînes sont si particulières, et donc à moins vous prendre la tête par la suite...

Définir une chaîne de caractères en utilisant des nombres

Pour créer un caractère, vous devez d'abord définir sa relation numérique. Plus important encore, tout le monde devrait être conscient que lorsqu'un certain nombre apparaît dans une application et qu'il est considéré comme étant un caractère, alors ce nombre subit une certaine conversion. L'une des méthodes les plus couramment admises est appelée ASCII (American Standard Code for Information Interchange). De ce fait, Python utilise ce code ASCII pour transformer la valeur 65 en la lettre A. Vous pouvez trouver une table d'équivalence à l'adresse <http://www.asciitable.com/>.



Chaque caractère doit être associé à un code

numérique différent. La lettre A a pour code 5. Par contre, la minuscule a a comme code 97. Pour l'ordinateur, ces deux lettres sont totalement différentes, même si la plupart des gens ne les différencient pas.

Les valeurs utilisées dans ce chapitre sont présentées sous une forme décimale. Pour autant, ce sont autant de 0 et de 1 du point de vue de l'ordinateur. Par exemple, la lettre majuscule A vaut en fait 01000001, et la minuscule a 01100001. Lorsque vous voyez sur l'écran un A ou un a, l'ordinateur, de son côté, ne voit que des valeurs binaires.



Bien entendu, l'idéal serait de n'avoir besoin que d'un seul jeu de caractères. De ce point de vue, il est évident que le système ASCII est très limité (même si Python n'a pas de problèmes avec les caractères accentués). C'est pourquoi il a été très largement étendu avec l'introduction du système Unicode. Mais c'est là un sujet qui dépasse largement nos besoins, et donc le cadre de ce livre.

Utiliser des caractères pour créer des chaînes

Python ne vous force pas à sauter au milieu de cerceaux pour créer des chaînes de caractères. En fait, le mot *chaîne* donne une bonne idée de ce qui se passe. Imaginez par exemple que vous avez un tas de perles et un fil. Vous enfiler une première perle, puis une autre, et ainsi de suite jusqu'à ce que le collier soit terminé. Vous finissez en mettant un fermoir, et c'est fait. Donc, un collier est un ensemble de perles

enfilées à la suite les unes des autres (vous pouvez remplacer les perles par autre chose, c'est juste pour l'image).

Le même concept s'applique aux chaînes de caractères. Si vous voyez affichée une phrase sur votre écran, vous savez qu'elle est constituée de caractères individuels qui sont alignés les uns à la suite des autres par le langage de programmation que vous utilisez. Ce langage crée une structure qui contient tous les caractères individuels, les uns mis à la suite des autres pour former une phrase. C'est le langage, et non l'ordinateur, qui sait qu'une certaine quantité de nombres (chaque nombre représentant un certain caractère) définit une chaîne formant une phrase.



Il est réellement important de savoir comment Python travaille avec les caractères. En effet, la plupart des fonctionnalités qu'il fournit dans ce domaine travaillent avec des caractères individuels. De surcroît, là où vous voyez une phrase, Python, de son côté, ne voit qu'une succession de nombres.

Contrairement à ce qui se passe avec la plupart des langages de programmation, Python vous permet de placer les chaînes de caractères entre des guillemets ou des apostrophes. Par exemple, « Bonjour la Terre ! » est une chaîne. Et 'Bonjour la Terre !' est également une chaîne. Python supporte également des guillemets et des apostrophes triplés qui vous permettent de créer des chaînes occupant plusieurs lignes. Les étapes qui suivent explorent certaines de ces fonctionnalités. Vous pouvez également ouvrir le fichier téléchargeable `BasicString.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

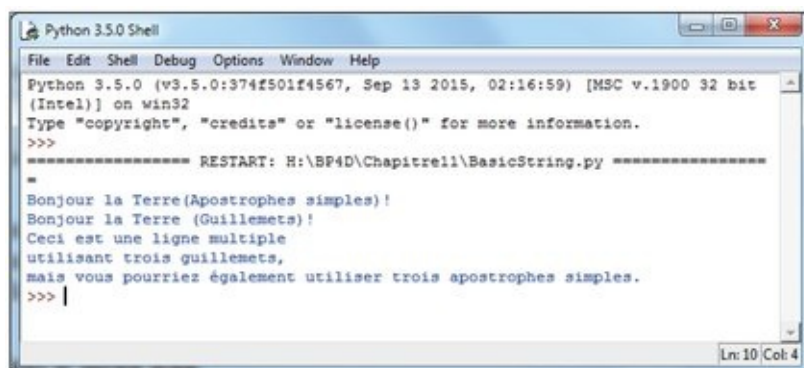
```
print('Bonjour la Terre(Apostrophes simples)!')
print("Bonjour la Terre (Guillemets)!")
print("""Ceci est une ligne multiple
utilisant trois guillemets,
mais vous pourriez également utiliser trois apostrophes simples.""")
```

Chacune des fonctions `print()` correspond à une méthode de travail différente : apostrophes, guillemets, ou encore triples guillemets. Toutes renvoient une chaîne de caractères parfaitement lisible.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le texte voulu. Remarquez que le dernier exemple répartit la phrase sur trois lignes, exactement comme dans le fichier source du code. Ce type de formatage permet de s'assurer que chaque ligne se termine exactement à l'emplacement souhaité (voir la [Figure 11.1](#)).

Figure 11.1 : Une chaîne est un ensemble de caractères attachés les uns aux autres.



Créer des chaînes comportant des caractères spéciaux

Certaines chaînes contiennent des caractères

spéciaux. Ils diffèrent des caractères alphanumériques et des signes de ponctuation que vous avez l'habitude d'utiliser. En fait, ils peuvent être répartis en plusieurs catégories :

➤ **Contrôle** : Une application a besoin de pouvoir déterminer si un caractère particulier n'est pas destiné à être affiché, mais plutôt à contrôler l'affichage. Tous les déplacements, par exemple, sont basés sur la position du *point d'insertion*, le trait clignotant que vous voyez sur l'écran quand vous tapez du texte. Par exemple, une tabulation ne se signale par rien de visible. Pourtant, elle fait bien partie du texte. De même, vous appuyez sur la touche Entrée lorsque vous voulez passer à la ligne suivante. Là encore, ceci se traduit par l'ajout d'un code numérique à votre texte.

➤ **Dessin** : Certains caractères ASCII permettent de réaliser des formes graphiques simples. Voyez par exemple le site <http://www.asciworld.com/>.

➤ **Typographie** : Nombre de caractères typographiques peuvent être affichés par un traitement de texte ou un éditeur évolué, comme le marqueur de fin de paragraphe, ou pied-de-mouche (¶).

➤ **Autre** : Selon le jeu de caractères que vous utilisez, les possibilités sont pratiquement sans fin. Vous pouvez trouver un caractère pour pratiquement tout ce dont vous pouvez avoir besoin. Tout le problème est d'indiquer à Python comment présenter ces caractères spéciaux.

Lorsque l'on travaille avec des chaînes, même simples, il est courant d'avoir à faire à des caractères de contrôle. Python fournit pour cela des séquences



dites d'échappement qui vous permettent de définir directement de tels caractères spéciaux.

Une *séquence d'échappement* transforme une lettre, comme un *a*, en lui donnant une nouvelle signification (comme un bip sonore). La combinaison de la barre oblique inverse (\) et d'une lettre (disons un *a*) sert à définir une telle séquence, ou code, d'échappement, dont le [Tableau 11.1](#) donne un aperçu.

Tableau 11.1 : Python et les séquences d'échappement.

<i>Séquence d'échappement</i>	<i>Signification</i>
\\	Barre oblique inverse (\)
\'	Apostrophe (')
\"	Guillemets (")
\a	Bip ASCII (BEL)
\b	Retour arrière ASCII (BS)
\f	Saut de page ASCII (FF)

<code>\n</code>	Saut de ligne Linefeed (LF)
<code>\r</code>	Retour chariot ASCII (CR)
<code>\t</code>	Tabulation horizontale ASCII (TAB)
<code>\uhhhh</code>	Caractère Unicode (un système de code étendu des caractères), <i>hhhh</i> désignant une valeur hexadécimale.
<code>\v</code>	Tabulation verticale ASCII (VT)
<code>\ooo</code>	Caractère ASCII, <i>ooo</i> désignant une valeur octale.
<code>\xhh</code>	Caractère ASCII,

hh désignant une valeur hexadécimale.

La meilleure manière de voir comment tout cela fonctionne, c'est bien sûr d'essayer. C'est ce que proposent les étapes qui suivent. Vous pouvez également les retrouver dans le fichier téléchargeable

`SpecialCharacters.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
print("Une partie de ce texte\r\nse trouve sur la ligne suivante.")
print("Ceci est un A avec un accent grave : \xC0.")
print("Ceci est un caractère dessiné : \u2562.")
print("Et voici un retour chariot: \266.")
print("Ceci est le signe de la division : \xF7.")
```

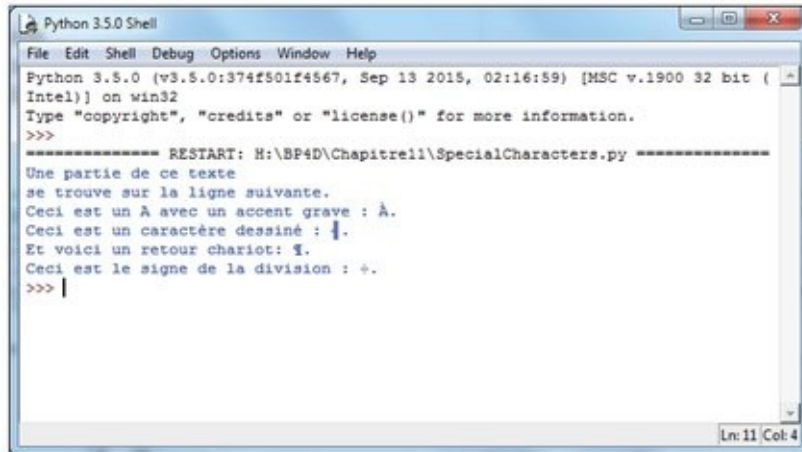
Ce code utilise plusieurs techniques dans le même but : produire un caractère spécial. Bien entendu, vous pouvez utiliser directement des caractères de contrôle (voyez la première ligne). Mais de nombreux autres sont accessibles en spécifiant une valeur hexadécimale sur deux chiffres (lignes 2 et 5), voire sur quatre chiffres dans le système étendu dit Unicode (ligne 3). Il est également possible de faire appel à des valeurs octales (ligne 4).

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche le texte ainsi que les caractères spéciaux voulus (voir la [Figure 11.2](#)).

Figure 11.2 :

Utiliser des caractères spéciaux permet d'afficher des informations particulières, ou encore de formater la sortie.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre11\SpecialCharacters.py =====
Une partie de ce texte
se trouve sur la ligne suivante.
Ceci est un À avec un accent grave : À.
Ceci est un caractère dessiné : |.
Et voici un retour chariot: ¶.
Ceci est le signe de la division : ÷.
>>> |
```



Python utilise un jeu de caractères standardisé, ce qui signifie que le résultat devrait être identique quelle que soit la plate-forme utilisée. Ce n'est cependant pas une certitude, et vous devriez donc tester votre application sur différents systèmes si l'emploi de caractères spéciaux y joue un rôle important. De plus, des configurations personnalisées d'ordinateur peuvent avoir une influence sur ce qui est affiché.

Sélectionner des caractères individuels

Vous savez maintenant que les chaînes sont formées de caractères individuels, comme des perles enfilées dans un collier. Python vous permet d'accéder directement à chaque perle, autrement dit à chaque caractère. C'est une fonctionnalité importante, puisqu'elle vous permet en particulier de créer de nouvelles chaînes à partir de l'original. De plus, vous pouvez combiner plusieurs chaînes pour en former une nouvelle. Le secret de cette technique, c'est l'emploi de crochets droits. Il vous suffit d'indiquer le

rang d'un caractère à l'intérieur de ces crochets pour le récupérer, comme ceci :

```
MyString = "Bonjour le monde"
print(MyString[0])
```



Dans cet exemple, la sortie serait le caractère *B*. Python débute en effet la numérotation des caractères d'une chaîne en partant de zéro pour le premier. Si vous tapez `print(MyString[1])`, la réponse sera donc *o*, et ainsi de suite.

Vous pouvez également récupérer une section de votre chaîne. Pour cela, vous spécifiez la position de début et la celle de fin en les séparant par un deux-points. Dans l'exemple ci-dessus, l'instruction `print(MyString[11 :16])` renverrait le mot *monde*. Le douzième caractère de la chaîne, qui a le rang 11 du point de vue de Python, est bien la lettre *m*.

Les étapes qui suivent mettent en œuvre quelques tâches simples basées sur la technique de sélection de caractères de Python. Vous les retrouvez également dans le fichier téléchargeable `characters.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
String1 = "Bonjour la Terre"
String2 = "Ici Python"

print(String1[0])
print(String1[0:5])
print(String1[5])
print(String1[6:])

String3 = String1[:6] + String2[:6]
print(String3)

print(String2[:7]*5)
```

Cet exemple commence par créer deux chaînes. Il pratique ensuite différentes manipulations sur la première. Remarquez que vous pouvez omettre le rang de début ou le rang de fin si vous voulez travailler avec le reste de la chaîne.

L'étape suivante combine deux sous-chaînes, en l'occurrence en plaçant leurs six premiers caractères dans une troisième chaîne.



Le signe d'addition sert à combiner deux chaînes de caractères en les plaçant bout à bout. C'est ce que l'on appelle la *concaténation*. Il s'agit d'une des opérations les plus utiles pour travailler avec les chaînes.

L'étape finale réalise une opération appelée *répétition*. Celle-ci permet d'obtenir un certain nombre de copies (cinq en l'occurrence) d'une chaîne ou d'une sous-chaîne.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche une série de combinaisons de chaînes (voir la [Figure 11.3](#)).

Figure 11.3 : Vous pouvez sélectionner diverses parties d'une chaîne de caractères.

A screenshot of a Python 3.5.0 Shell window. The window title is 'Python 3.5.0 Shell'. The menu bar includes 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The text area shows the following output:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit  
(Intel)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: H:\BP4D\Chapitre11\Characters.py =====  
B  
Bonjo  
Bonjo  
r la Terre  
BonjourIci Py  
Ici PytIci PytIci PytIci PytIci Pyt  
>>>
```

The status bar at the bottom right indicates 'Ln:11 Col:4'.

Trancher et couper les chaînes de

caractères

Accéder à des plages de caractères offre un certain degré de souplesse, mais cela ne vous permet pas en soi de manipuler le contenu des chaînes de caractères, ni de découvrir quoi que ce soit sur elles. Vous pourriez par exemple avoir besoin de convertir les lettres en majuscules, ou bien encore de savoir si une phrase contient toutes les lettres de l'alphabet.

Python offre de nombreuses fonctions pour manipuler les chaînes de caractères. Voici les plus courantes :

✓ `capitalize()` : Met la première lettre d'une chaîne en majuscule.

✓ `center(largeur, remplissage= » « )` : Centre une chaîne de manière à ce qu'elle remplisse l'espace spécifié par *largeur*. Si vous fournissez un caractère pour *remplissage*, la fonction l'utilisera. Sinon, elle se sert par défaut du caractère d'espacement pour obtenir la largeur voulue.

✓ `expandtabs (tailletab=8)` : Modifie les tabulations d'une chaîne en remplaçant l'espacement par défaut (8) par la valeur spécifiée dans *taille*tab.

✓ `isalnum()` : Retourne True si la chaîne a au moins un caractère et que tous les caractères sont alphanumériques (lettres ou chiffres).

✓ `isalpha()` : Retourne True si la chaîne a au moins un caractère et que tous les caractères sont alphabétiques (lettres uniquement).

✓ `isdecimal()` : Retourne True si une chaîne Unicode contient uniquement des caractères décimaux.

✓ `isdigit()` : Retourne True si une chaîne contient uniquement des chiffres (pas de lettres).

✓ `islower()` : Retourne True si une chaîne contient au moins un caractère alphabétique, et si

toutes les lettres sont en minuscules.

✓ `isnumeric()` : Retourne True si une chaîne Unicode contient uniquement des caractères numériques.

✓ `isspace()` : Retourne True si une chaîne contient uniquement des caractères blancs (espaces, tabulations, retours chariot, saut de ligne ou de page et tabulations verticales, sauf des retours arrière).

✓ `istitle()` : Retourne True si la première lettre de chaque mot est une majuscule. Attention : une lettre unique est comptée comme un mot.

✓ `isupper()` : Retourne True si une chaîne contient au moins un caractère alphabétique, et si toutes les lettres sont en majuscules.

✓ `join(seq)` : Crée une chaîne dans laquelle les caractères de la chaîne de base sont séparés par le contenu de `seq` d'une manière répétitive. Par exemple, si `MaChaine` contient « Hello », alors `print(MaChaine. join(« ! * ! »))` retournera `!Hello *Hello !`.

✓ `len(chaine)` : Renvoie la longueur de la chaîne.

✓ `ljust(largeur, remplissage= » « )` : Justifie à gauche une chaîne de manière à ce qu'elle tienne dans l'espace défini par `largeur`. Si vous ne fournissez pas un caractère de remplissage, la fonction utilisera des espaces pour créer une chaîne de la largeur désirée.

✓ `lower()` : Convertit toutes les lettres d'une chaîne en minuscules.

✓ `lstrip()` : Supprimer tous les espaces blancs au début d'une chaîne.

✓ `max(chaine)` : Retourne le caractère de la chaîne dont la valeur numérique est la plus élevée. Par exemple, `a` a un rang numérique supérieur à `A`.

✓ `min(chaine)` : Retourne le caractère de la chaîne dont la valeur numérique est la moins élevée.

Par exemple, A a un rang numérique inférieur à a.

✓ `rjust(largeur, remplissage= » « )` : Justifie à droite une chaîne de manière à ce qu'elle tienne dans l'espace défini par *largeur*. Si vous ne fournissez pas un caractère de remplissage, la fonction utilisera des espaces pour créer une chaîne de la largeur désirée.

✓ `rstrip()` : Supprimer tous les espaces blancs à la fin d'une chaîne.

✓ `split(chaîne= » « , num=string.count(chaîne))` : Partage une chaîne en sous-chaîne en utilisant le délimiteur spécifié par *chaîne*, ou par un espace si cet argument n'est pas spécifié. Par exemple, si votre chaîne d'origine contient *Un Jour Sans Fin*, le résultat par défaut sera Un, Jour, Sans et Fin. La valeur *num* permet de spécifier le nombre de sous-chaînes à renvoyer (toutes par défaut).

✓ `splitlines(num=string.count('\n'))` : Partage une chaîne contenant des changements de ligne (\n) en autant de chaînes individuelles. Vous pouvez spécifier avec *num* le nombre de chaînes à retourner.

✓ `strip()` : Supprime tous les espaces en début et en fin de chaîne.

✓ `swapcase()` : Inverse la casse de chaque caractère alphabétique de la chaîne.

✓ `title()` : Retourne une chaîne dans laquelle la première lettre de chaque mot est en majuscule, le reste étant en minuscules.

✓ `upper()` : Convertit toutes les lettres d'une chaîne en minuscules.

✓ `zfill(largeur)` : Renvoie une chaîne complétée à gauche par des zéros de manière à ce que la chaîne résultat ait la largeur indiquée. Cette fonction est conçue pour formater des chaînes

contenant des valeurs numériques. Le signe original est conservé s'il est présent.

Jouer un peu avec ces fonctions peut vous aider à mieux les comprendre. C'est ce que proposent les étapes qui suivent. Vous pouvez également les retrouver dans le fichier téléchargeable `Functions.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
MyString = "  Bonjour la Terre  "

print(MyString.upper())

print(MyString.strip())

print(MyString.center(21, "**"))
print(MyString.strip().center(21, "**"))

print(MyString.isdigit())
print(MyString.istitle())

print(max(MyString))

print(MyString.split())
print(MyString.split()[0])
```

Ce code commence par créer une chaîne appelée `MyString`. Remarquez qu'elle contient des espaces au début et à la fin. La tâche initiale consiste à convertir tous les caractères en majuscules.

Supprimer les espaces en trop est une tâche courante dans les applications. C'est le rôle de la fonction `strip()`. La fonction `center()` vous permet à l'inverse compléter une chaîne de manière à ce qu'elle remplisse une certaine largeur. Si vous combinez les deux, le résultat est différent de ce que fournit la fonction `center()` seule.



Vous pouvez combiner plusieurs fonctions pour obtenir le

résultat voulu. Python exécute chaque fonction l'une après l'autre, en allant de la gauche vers la droite. L'ordre dans lequel ces fonctions apparaissent affecte donc la sortie. Il faut par conséquent bien veiller à ne pas se tromper de sens. Si le résultat que vous obtenez est différent de ce que vous vouliez obtenir, essayez en premier lieu de changer l'ordre des fonctions.

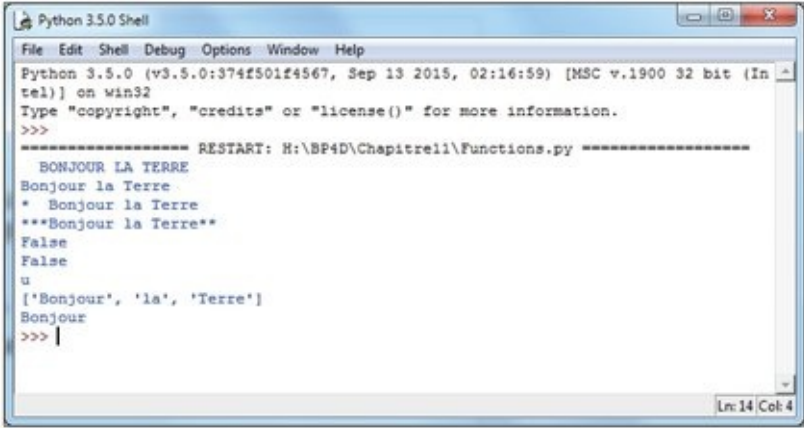
Certaines fonctions travaillent sur une chaîne passée en argument, plutôt que sur une instance de la chaîne. C'est par exemple le cas de la fonction `max()`. Si vous tapez `MyString.max()`, Python renverra une erreur. La liste développée au début de cette section spécifie les cas concernés.

Lorsque vous travaillez avec des fonctions qui renvoient une liste, vous pouvez accéder à chaque membre individuel de celle-ci en spécifiant un indice. Par exemple, la fonction `split()` partage une chaîne en sous-chaînes. La dernière instruction de l'exemple permet de récupérer la première d'entre elles (celle qui a le rang 0). Nous reviendrons sur les listes dans le Chapitre 12.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche une série de chaînes modifiées (voir la [Figure 11.4](#)).

Figure 11.4 :
Python offre de nombreuses fonctions pour manipuler les chaînes de caractères.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre11\Functions.py =====
BONJOUR LA TERRE
Bonjour la Terre
* Bonjour la Terre
**Bonjour la Terre**
False
False
u
['Bonjour', 'la', 'Terre']
Bonjour
>>> |
```

Localiser une valeur dans une chaîne

Vous avez parfois besoin de localiser une valeur dans une chaîne. Par exemple, vous voudriez savoir si une certaine chaîne contient le mot *Bonjour*. L'un des objets essentiels de la création et de la manipulation des données est justement la possibilité d'y effectuer ensuite des recherches pour retrouver des informations spécifiques. Et les chaînes de caractères ne sont pas différentes : elles sont essentiellement utiles lorsque vous pouvez y retrouver ce dont vous avez besoin rapidement et sans problèmes. Python fournit de nombreuses fonctions pour effectuer ce genre de travail. Voici celles qui sont le plus couramment utilisées :

✓ `count(chaîne1, debut= 0, fin=len(chaîne2))` : Compte le nombre d'occurrences de *chaîne1* dans *chaîne2*. Vous pouvez spécifier en plus un indice de début et de fin.

✓ `endswith(suffixe, debut=0, fin=len(chaîne))` : Renvoie True si une chaîne se termine par les caractères spécifiés dans *suffixe*. Vous pouvez spécifier en plus un indice de début et de fin.

✓ `find(chaîne1, debut=0, fin=len(chaîne2))` : Détermine si *chaîne1* se trouve dans *chaîne2* et renvoie sa position. Vous pouvez spécifier en plus un indice de début et de fin.

✓ `index(chaîne1, debut=0, fin=len(chaîne2))` : Comme `find()`, mais lève une exception si *chaîne1* n'est pas trouvée.

✓ `replace(ancien, nouveau [, max])` : Remplace toutes les occurrences de la séquence spécifiée dans *ancien* par celle spécifiée dans *nouveau*. Vous pouvez spécifier en plus un indice de début et de fin.

✓ `rfind(chaîne1, debut=0, fin=len(chaîne2))` : Comme `find()`,

mais en partant de la fin de la chaîne et non du début de celle-ci.

✓ `rindex(chaîne, debut=0, fin=len(chaîne))` : Comme `index()`, mais en partant de la fin de la chaîne et non du début de celle-ci.

✓ `startswith(prefixe, debut=0, fin=len(chaîne))` : Renvoie `True` si une chaîne débute par les caractères spécifiés dans *prefixe*. Vous pouvez spécifier en plus un indice de début et de fin.

Retrouver les données dont vous avez besoin est une tâche essentielle en programmation, et ce quelle que soit l'application que vous créez. Les étapes qui suivent vous proposent un exemple de ce genre de recherche. Vous pouvez également le retrouver dans le fichier téléchargeable `SearchString.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
SearchMe = "La pomme est rouge et la luzerne est verte !"

print(SearchMe.find("est"))
print(SearchMe.rfind("est"))

print(SearchMe.count("est"))

print(SearchMe.startswith("La"))
print(SearchMe.endswith("La"))

print(SearchMe.replace("pomme", "voiture")
      .replace("luzerne", "camionnette"))
```

L'exemple commence par créer la chaîne `SearchMe`, dans laquelle se trouvent deux instances du mot `est`. Ceci permet de montrer en quoi les recherches diffèrent selon leur point de départ. Avec `find()`, le code recherche le mot `est` en partant du début de la chaîne. Avec `rfind()`, elle part de la fin de la chaîne.

Bien entendu, vous ne savez généralement pas combien de

fois une certaine séquence de caractères apparaît dans une chaîne. La fonction `count()` vous permet de déterminer cette valeur.

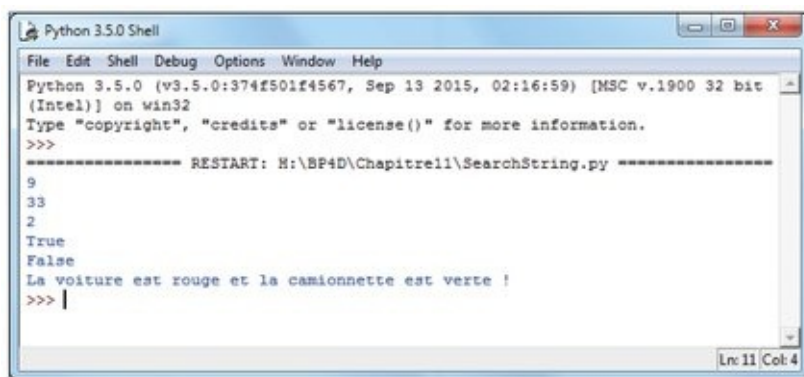
Selon la nature des données avec lesquelles vous travaillez, il est possible que celles-ci possèdent un formatage particulier. Vous pouvez alors tirer profit de cette disposition, afin par exemple de déterminer si une chaîne ou une sous-chaîne débute ou se termine par une séquence particulière. Vous pourriez ainsi vous servir de cette technique pour localiser un numéro de composant.

L'instruction finale remplace les mots *pomme* et *luzerne* respectivement par *voiture* et *camionnette*. Remarquez que ce code est placé sur deux lignes. Cette façon de procéder permet de le lire plus lisible.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche une série de réponses (voir la [Figure 11.5](#)). Vous pouvez constater que les recherches retournent des résultats différents selon qu'elles sont effectuées en partant du début de la chaîne ou de la fin de celle-ci. Choisir la fonction qui convient est essentiel pour vous assurer que vous obtiendrez les bons résultats.

Figure 11.5 :
Effectuer des recherches dans des chaînes de caractères.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: M:\BP4D\Chapitre11\SearchString.py =====
9
33
2
True
False
La voiture est rouge et la camionnette est verte !
>>> |
```

Formater les chaînes de caractères

Python vous permet de formater vos chaînes de caractères de multiples manières. Le rôle premier du formatage consiste à présenter les chaînes pour qu'elles soient à la fois plaisantes et faciles à comprendre. Cela ne signifie pas dans ce cas ajouter des effets particuliers, ou afficher de jolis caractères, mais concerne uniquement le mode de présentation des données, par exemple des valeurs numériques ou monétaires.

La plupart des options de formatage font appel à la fonction `format()`. Vous créez des spécifications de formatage à l'intérieur de la chaîne, puis vous utilisez `format()` pour ajouter des données à celle-ci. Une spécification peut être très simple. Exemple : deux crochets `{}` forment un conteneur pour y saisir des données. Ce conteneur peut servir à créer des effets spéciaux. Ainsi, `{0}` contiendrait le premier élément de donnée d'une chaîne. Lorsque les éléments de donnée sont numérotés, vous pouvez même les répéter pour qu'ils apparaissent plusieurs fois dans la chaîne.

La spécification de formatage est précédée d'un deux-points. Si vous voulez définir un formatage simple, les crochets ne contiennent que ce deux-points et l'option voulue. Par exemple, `{ :f}` affichera un nombre en numérotation fixe. Si vous voulez numéroté les entrées, la spécification `{0 :f}` fera la même chose, mais pour le premier élément. Les spécifications de formatage suivent ce format, les mots en italiques étant bien entendu à remplacer par les valeurs voulues :

```
[[remplissage] alignement][signe][#][0][largeur][,][.précision][type]
```

Les spécifications que vous trouverez à l'adresse

<https://docs.python.org/3/library/string.html>

fournissent des détails plus approfondis, mais voici un aperçu de ce que signifient ces différentes entrées :

✓ **Remplissage** : Définit le caractère utilisé lorsque le remplissage ne suffit pas à remplir l'espace assigné.

✓ **Alignement** : Spécifie la manière dont les données sont alignées à l'intérieur de l'espace d'affichage. Vous pouvez utiliser ces différents modes :

- < : Alignement à gauche
- > : Alignement à droite
- ^ : Centré
- = : Justifié

✓ **Signe** : Détermine l'utilisation du signe lors de la sortie :

- + : Les nombres positifs ont un signe plus, et les nombres négatifs un signe moins.
- - : Les nombres négatifs ont un signe moins.
- <espace> : Les nombres positifs sont précédés d'un espace et les nombres négatifs ont un signe moins.

✓ **#** : Spécifie que la sortie devrait utiliser le format d'affichage alternatif pour les nombres. Par exemple, les valeurs hexadécimales prendront le préfixe 0x.

✓ **0** : Spécifie que la sortie doit tenir compte du signe, et que la valeur doit être complétée par des zéros pour fournir une sortie cohérente.

✓ **Largeur** : Détermine la largeur totale du champ de donnée (même si celle-ci n'occupe pas tout cet espace).

✓ **,** : Spécifie que la valeur numérique doit avoir

une virgule comme séparateur des milliers.

➤ **.précision** : Détermine le nombre de chiffres après le point décimal.

➤ **Type** : Spécifie le type de la sortie, même si l'entrée ne correspond pas. Les types sont répartis en trois groupes :

- *Chaîne* : Utilisez un `s` ou rien pour spécifier une chaîne.
- *Entier* : Les types entiers sont les suivants : `b` (binaire), `c` (caractère), `d` (décimal), `o` (octal), `x` (hexadécimal avec des lettres en minuscules), `X` (hexadécimal avec des lettres en majuscules) et `n` (valeur décimale sensible à la localisation géographique utilisant le caractère approprié comme séparateur des milliers).
- *Virgule flottante* : Les types en virgule flottante sont les suivants : `e` (exposant utilisant une lettre `e` minuscule), `E` (exposant utilisant une lettre `E` majuscule), `f` (valeur décimale fixée en minuscule), `F` (valeur décimale fixée en majuscule), `g` (format général en minuscule), `G` (format général en majuscule), `n` (valeur sensible à la localisation géographique utilisant le caractère approprié comme séparateur décimal et des milliers), et `%` (pourcentage).

Les éléments qui spécifient le formatage doivent apparaître dans l'ordre indiqué, sinon Python ne pourra pas les traiter correctement. Dans ce cas, il affichera un message d'erreur au lieu du formatage spécifié. Les étapes qui suivent illustrent cette technique. Vous pouvez également les retrouver dans le fichier téléchargeable `Formatted.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Formatted = "{:d}"
print(Formatted.format(7000))

Formatted = "{:,d}"
print(Formatted.format(7000))

Formatted = "{:^15,d}"
print(Formatted.format(7000))

Formatted = "{:*^15,d}"
print(Formatted.format(7000))

Formatted = "{:*^15.2f}"
print(Formatted.format(7000))

Formatted = "{:>15X}"
print(Formatted.format(7000))

Formatted = "{:<#15x}"
print(Formatted.format(7000))

Formatted = "Une {1} {0} et un {2} {0}."
print(Formatted.format("bleu(e)", "voiture", "camion"))
```

Cet exemple débute par un champ formaté en tant que valeur décimale. Il ajoute ensuite une virgule décimale. L'étape qui suit consiste à rendre le champ plus large qu'il n'est besoin pour contenir la donnée, et à centrer celle-ci à l'intérieur de ce champ. Finalement, un astérisque à la sortie de manière à remplir ce champ.

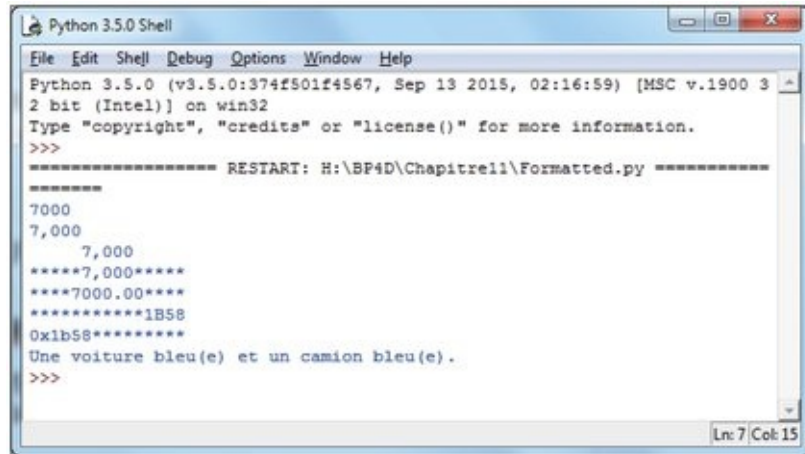
Bien entendu, l'exemple propose d'autres types de données. Les étapes suivantes affichent la même donnée dans un format préfixé, ainsi qu'en mode hexadécimal avec représentation en majuscules ou en minuscules. Dans le premier cas, la sortie est alignée à droite, et à gauche dans le second.

Enfin, l'exemple montre comment tirer profit de champs numérotés à votre avantage. Dans ce cas, des valeurs préfixées sont ensuite remplacées par des chaînes personnalisées.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche les données selon différentes formes (voir la [Figure 11.6](#)).

Figure 11.6 : Le formatage vous permet de présenter vos données comme vous le voulez.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 3
2 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre11\Formatted.py =====
>>>
7000
7,000
    7,000
*****7,000*****
****7000.00****
*****1B58
0x1b58*****
Une voiture bleu(e) et un camion bleu(e).
>>>
```

Chapitre 12

Gérer des listes

Dans ce chapitre :

- ▶ Comprendre l'importance des listes.
- ▶ Générer des listes.
- ▶ Parcourir des listes.
- ▶ Travailler avec les éléments des listes.
- ▶ Modifier le contenu d'une liste.
- ▶ Localiser des éléments spécifiques dans des listes.
- ▶ Mettre les listes dans l'ordre.
- ▶ Savoir utiliser l'objet Counter.

Des tas de gens perdent facilement de vue que la plupart des techniques de programmation sont basées sur le monde réel. Une des raisons tient au fait que les programmeurs utilisent souvent des termes qui ne parlent pas au commun des mortels. Par exemple, vous dites que vous rangez quelque chose dans une boîte. Les développeurs, de leur côté, diront qu'ils affectent une valeur à une *variable*. Les listes sont différentes. Après tout, tout le monde fait des listes pour tout un tas de choses : des listes de courses, des listes de bagages pour les vacances, des listes de tâches à réaliser, et ainsi de suite. Ce chapitre traite

donc de quelque chose que vous utilisez déjà dans la vie de tous les jours. La seule différence, c'est que vous devez envisager ce sujet du point de vue de Python.

Si certaines personnes trouvent qu'il est difficile de travailler avec les listes, c'est vraisemblablement parce qu'elles n'ont pas l'habitude de réfléchir aux listes qu'elles créent. Lorsque vous rédigez une liste, vous écrivez simplement une suite d'éléments dans un ordre qui a un sens pour vous. Parfois, vous reprenez votre liste pour en changer l'ordre (mais pourquoi ont-ils encore déplacé le rayon dans le magasin ?). Ou bien vous parcourez la liste du bout du doigt pour vérifier si vous n'avez rien oublié. En fait, tout ce que vous pouvez faire avec vos propres listes est également réalisable avec Python. Le point essentiel, c'est que vous devez bien réfléchir à ce que vous faites si vous voulez que Python comprenne votre objectif.

Les listes sont incroyablement importantes dans Python. Ce chapitre vous présente les concepts utilisés pour créer, gérer, rechercher et afficher des listes (parmi d'autres tâches). Lorsque vous l'aurez terminé, vous serez à même d'utiliser des listes pour rendre vos applications Python plus robustes, plus rapides et plus souples. En fait, vous pourriez bien vous demander comment vous avez fait sans elles par le passé. La chose importante à se rappeler, c'est que les listes font depuis pratiquement toujours partie de votre quotidien. Il n'y a aucune différence ici, si ce n'est que vous devez maintenant réfléchir plus à fond aux actions à mettre en œuvre pour que vos listes soient totalement efficaces.

Organiser les informations dans une application

Les gens créent des listes pour organiser des informations, et en faciliter l'accès ainsi que les modifications. Vous utilisez des listes dans Python pour les mêmes raisons. Dans de nombreuses situations, vous avez besoin d'une forme d'organisation de vos données pour pouvoir gérer celles-ci. Par exemple, vous pourriez vouloir créer un « quelque chose » qui contiendrait les jours de la semaine ou les mois de l'année. Les noms de ces éléments apparaîtraient dans une liste, comme si vous les aviez écrits sur une feuille de papier dans le monde réel. Les sections qui suivent approfondissent ces notions.

Définir une organisation à l'aide de listes

Dans Python, une liste se définit comme étant un type de séquence. Les *séquences* fournissent un certain procédé permettant à de multiples données de coexister en tant qu'entités distinctes dans un même espace de stockage. Prenons l'exemple d'un grand immeuble : en bas, vous trouvez plusieurs blocs de boîtes aux lettres. Chacun contient une rangée de boîtes, et chaque boîte contient du courrier. Python supporte d'autres types de séquences au nom parfois mystérieux (nous y reviendrons aussi au Chapitre 13) :

- Tuples
- Dictionnaires

- ✓ Piles
- ✓ Files
- ✓ Deques



De toutes les séquences, les listes sont les plus faciles à comprendre, et les plus proches du monde réel. Se familiariser avec des listes vous aide à devenir plus à même de travailler avec d'autres types de séquences fournissant davantage de puissance et de souplesse. L'essentiel à retenir, c'est que les données sont enregistrées dans une liste comme si vous les écriviez sur une feuille de papier, un élément après l'autre, comme l'illustre la [Figure 12.1](#). La liste a un début, un milieu et une fin. De plus, même si ce n'est pas forcément ce que vous feriez dans la vie courante, Python numérote automatiquement chacun des éléments de la liste.

Figure 12.1 : Une liste est simplement une suite d'éléments.



Comprendre comment les ordinateurs voient les listes

Les ordinateurs ne voient pas les listes de la même façon que vous. Il n'a ni bloc-notes ni crayon. Il a juste de la mémoire. L'ordinateur enregistre donc chaque

élément d'une liste dans un emplacement mémoire distinct, comme l'illustre la [Figure 12.2](#). Cette mémoire est continue. Lorsque vous ajoutez de nouveaux éléments, ils viennent donc prendre place dans le prochain emplacement en mémoire.

Figure 12.2 :

Chaque élément ajouté à une liste est inséré dans le prochain emplacement en mémoire.



D'une certaine manière, une liste se comporte dans l'ordinateur un peu comme une série de boîtes alignées : le premier élément est rangé dans la première boîte, le second dans la deuxième, et ainsi de suite. L'avantage, c'est bien sûr que la mémoire de l'ordinateur peut contenir un très grand nombre de ces boîtes, beaucoup plus que votre armoire à chaussures...



Python étant vraisemblablement mieux organisé que vous, les emplacements utilisés pour stocker les éléments d'une liste sont numérotés. Cette numérotation commence à 0 (sans doute pour des raisons de facilité). Prenons l'exemple d'une liste de noms de mois. Elle comportera douze éléments numérotés de 0 à 11. Il est évidemment essentiel de bien intégrer cette manière de procéder.

Selon le type d'information que vous enregistrez dans les différentes « boîtes », celles-ci n'ont pas besoin d'avoir toutes la même taille. Python vous permet par exemple de placer une chaîne de caractères dans une boîte, un entier dans une autre, et une valeur en

virgule flottante dans une troisième. L'ordinateur ne sait pas, et n'a pas besoin de savoir, ce qui se trouve dans chacune des boîtes. Tout ce qu'il voit, c'est une longue suite de chiffres qui pourraient représenter n'importe quoi. C'est Python qui effectue tout le travail nécessaire au traitement des données en fonction de leur type, et qui, si vous demandez l'élément numéro 5, vous renvoie l'élément numéro 5 et rien d'autre.



En règle générale, il est préférable de créer des listes formant des ensembles de données plus faciles à gérer. En vous limitant par exemple à une liste composée uniquement de nombres entiers, vous pouvez facilement faire des suppositions sur ce que vous pouvez y trouver, sans perdre votre temps à tenter de retrouver la bonne valeur. Pour autant, il y a bien entendu des cas où des données de nature différente doivent être associées. Cependant, souvenez-vous que cela impose de déterminer le type des données lorsque vous recherchez des informations afin de pouvoir traiter correctement celles-ci. Prendre une chaîne pour un entier pourrait provoquer des problèmes dans votre application (ce qui n'est pas le cas dans d'autres langages de programmation, qui imposent généralement de ne créer que des listes formées d'éléments de même type).

Créer des listes

Avant de pouvoir faire quoi que ce soit avec une liste, vous devez évidemment commencer par créer celle-ci. Comme cela a été mentionné plus haut, les listes

de Python peuvent mélanger différents types de données. Mais c'est toujours une bonne pratique de n'utiliser qu'un seul type chaque fois que c'est possible. Les étapes qui suivent vous montrent comment créer une liste dans Python.

1. **Ouvrez une fenêtre Python en mode Shell.**

Vous retrouvez l'indicatif habituel.

2. **Tapez `List1 = [« Un », 1, « Deux », True]` et appuyez sur Entrée.**

Python va créer une liste appelée `List1`. Elle contient deux chaînes (Un et Deux), une valeur entière (1) et une valeur booléenne (True). Bien entendu, vous ne pouvez rien voir de plus, puisque Python traite la commande sans rien dire (ce qui est bon signe).



Souvenez-vous que Python affiche chaque type de données dans une couleur différente, ce qui fournit un bon indice et vous aide à réduire les risques d'erreurs lorsque vous créez une liste.

3. **Tapez `print(List1)` et appuyez sur Entrée.**

Vous voyez s'afficher la liste en entier (voir la [Figure 12.3](#)). Notez que les chaînes apparaissent entre apostrophes, quelle que soit la manière dont vous les avez saisies.

Figure 12.3 : Le contenu de la liste `List1`.

A screenshot of a Python 3.5.0 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The text inside the shell shows the following commands and output:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = ["Un", 1, "Deux", True]
>>> print(List1)
['Un', 1, 'Deux', True]
>>>
```

The output shows the list `['Un', 1, 'Deux', True]` with each element color-coded: strings in quotes, integers in blue, and booleans in red.

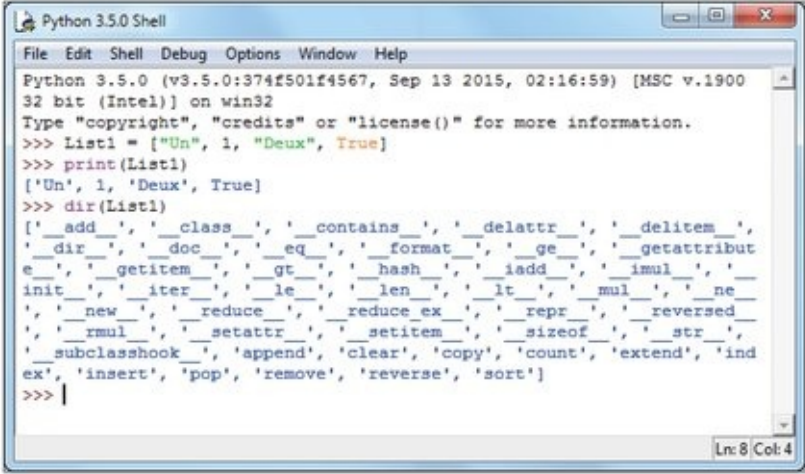
4. **Tapez `dir(List1)` et appuyez sur Entrée.**

Python affiche une série d'actions réalisables avec les listes, comme l'illustre la [Figure 12.4](#). Vous pouvez constater au

passage que la sortie est aussi une liste...

Figure 12.4 :

Python affiche une liste des actions que vous pouvez effectuer sur une liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = ["Un", 1, "Deux", True]
>>> print(List1)
['Un', 1, 'Deux', True]
>>> dir(List1)
['_add_', '_class_', '_contains_', '_delattr_', '_delitem_',
'_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_
e_', '_getitem_', '_gt_', '_hash_', '_iadd_', '_imul_', '_
init_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_
_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_reversed_
_', '_rmul_', '_setattr_', '_setitem_', '_sizeof_', '_str_',
'_subclasshook_', '_append_', '_clear_', '_copy_', '_count_', '_extend_', '_ind
ex_', '_insert_', '_pop_', '_remove_', '_reverse_', '_sort_']
>>> |
```

5. Vous pouvez refermer la fenêtre de Python.



Lorsque vous travaillez avec un objet complexe, souvenez-vous que la commande `dir()` vous montre toujours les tâches que vous pouvez réaliser avec cet objet. Les principales actions dont vous disposez sont celles qui ne sont pas précédées (et suivies) de traits de soulignement. Ici, ce sont les suivantes :

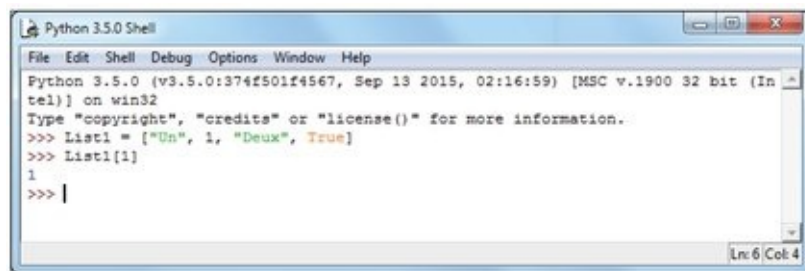
- `append` (ajoute)
- `clear` (efface)
- `copy` (copie)
- `count` (compte)
- `extend` (étend)
- `index` (index)
- `insert` (insère)
- `pop` (élimine)
- `remove` (supprime)
- `reverse` (inverse)
- `sort` (trie)

Accéder aux listes

Une fois que vous avez créé une liste, vous voulez accéder aux éléments qu'elle contient. La section précédente vous a montré comment `print ()` et `dir ()` vous permettaient d'interagir avec une liste, mais il y a bien entendu des tas d'autres possibilités.

1. **Ouvrez une fenêtre Python en mode Shell.**
Vous retrouvez l'indicatif habituel.
2. **Tapez `List1 = [« Un », 1, « Deux », True]` et appuyez sur Entrée.**
Python va créer une liste appelée `List1`.
3. **Tapez `List1[1]` et appuyez sur Entrée.**
La valeur 1 s'affiche (voir la [Figure 12.5](#)).

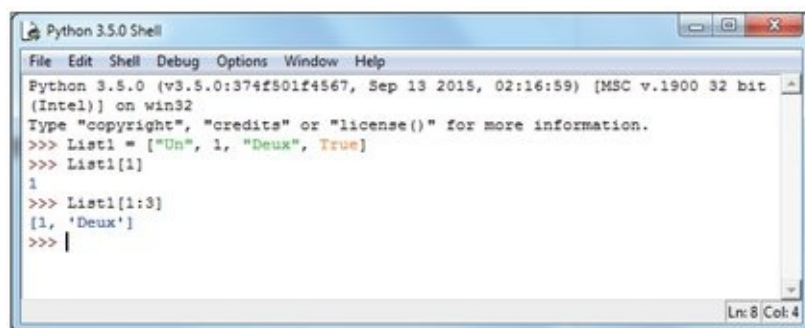
Figure 12.5 :
Python affiche
l'élément demandé
dans la liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = ["Un", 1, "Deux", True]
>>> List1[1]
1
>>> |
```

4. **Tapez `List1[1 :3]` et appuyez sur Entrée.**
Cette fois, Python affiche deux éléments (voir la [Figure 12.6](#)). Lorsque vous définissez un intervalle, la dernière valeur est toujours supérieure d'une unité au nombre d'éléments renvoyés. Ici, cela signifie que vous obtenez les éléments 1 et 2, et non 1 à 3.

Figure 12.6 : Un
intervalle permet
de renvoyer
plusieurs valeurs.



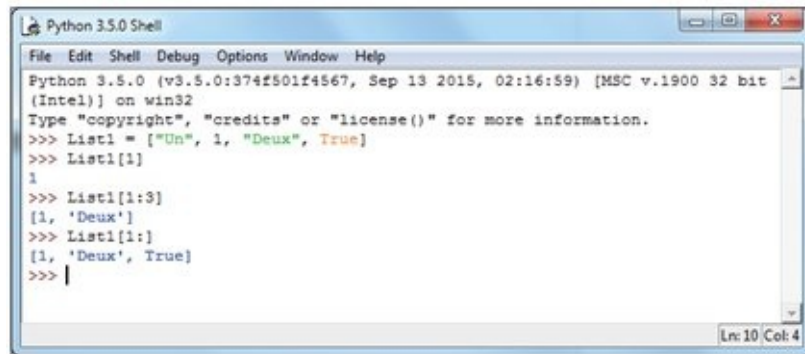
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = ["Un", 1, "Deux", True]
>>> List1[1]
1
>>> List1[1:3]
[1, 'Deux']
>>> |
```


5. **Tapez `List1[1:]` et appuyez sur Entrée.**

Cette fois, vous obtenez tous les éléments du rang 1 jusqu'à la fin de la liste (voir la [Figure 12.7](#)). L'absence du second indice est simplement interprétée comme signifiant « jusqu'à la fin de la liste ».

Figure 12.7 :

Afficher les éléments jusqu'à la fin de la liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = ["Un", 1, "Deux", True]
>>> List1[1]
1
>>> List1[1:3]
[1, 'Deux']
>>> List1[1:]
[1, 'Deux', True]
>>> |
```

6. **Tapez `List1[:3]` et appuyez sur Entrée.**

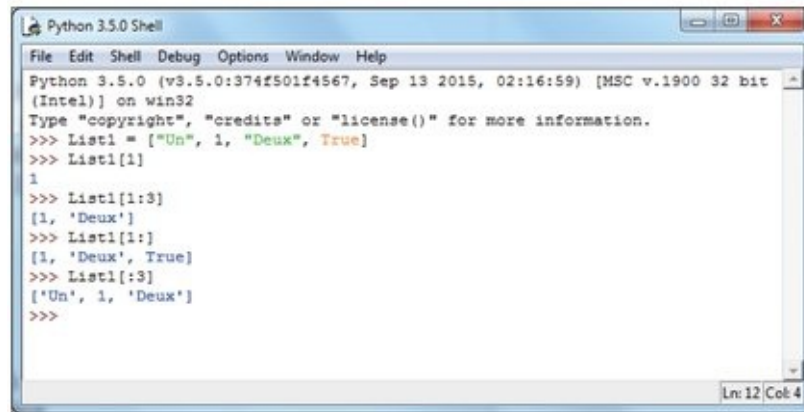
Python affiche maintenant les trois premiers éléments de la liste, autrement dit ceux dont le rang va de 0 à 2 (voir la [Figure 12.8](#)).

7. **Vous pouvez refermer la fenêtre de Python.**



Même si cela paraît un peu étrange, Python vous permet d'utiliser des indices négatifs. Dans ce cas, la liste est parcourue de la fin vers le début, et non l'inverse. Si vous tapez par exemple ici `List1[-2]`, vous obtiendrez le second élément en partant de la fin, soit Deux. De même, `List1[-3]` renverrait 1. Dans ce cas, l'élément le plus à droite aura pour indice -1.

Figure 12.8 :
Afficher des
éléments de la liste
en partant du
premier (au rang
0).



Parcourir les listes

Pour automatiser le traitement des listes, vous avez besoin de les parcourir. La méthode la plus simple consiste à utiliser une boucle `for`, comme l'illustrent les étapes qui suivent. Cet exemple peut également être retrouvé dans le fichier téléchargeable `ListLoop.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

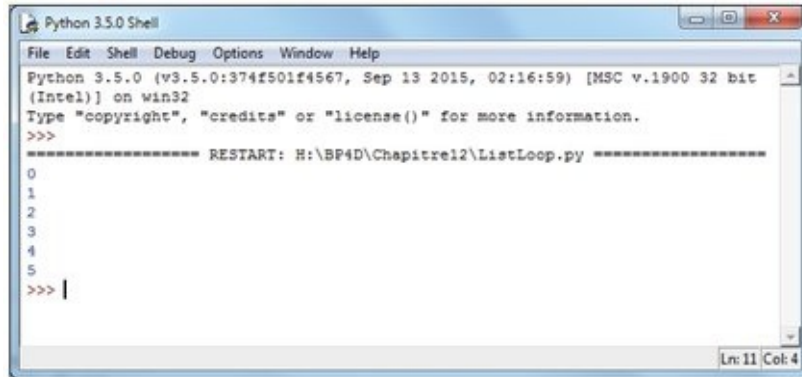
```
List1 = [0, 1, 2, 3, 4, 5]  
  
for Item in List1:  
    print(Item)
```

Cet exemple commence par créer une liste de valeurs numériques. Il utilise ensuite une boucle `for` pour obtenir chaque élément et l'afficher.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche les valeurs individuelles de la liste, ligne par ligne (voir la [Figure 12.9](#)).

Figure 12.9 : Une boucle permet d'obtenir facilement chaque élément d'une liste avant d'appliquer les traitements appropriés.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre12\ListLoop.py =====
0
1
2
3
4
5
>>> |
```

Modifier des listes

Vous pouvez modifier à votre guise le contenu d'une liste. Cela signifie changer une entrée particulière, en ajouter une nouvelle, ou encore supprimer une entrée existante. Python vous offre pour cela toute une série de fonctions :

- ✓ `append()` : Ajoute une nouvelle entrée à la fin de la liste.
- ✓ `clear()` : Supprime toutes les entrées de la liste.
- ✓ `copy()` : Crée une copie de la liste et la place dans une nouvelle liste.
- ✓ `extend()` : Ajoute des éléments d'une liste existante à la liste courante.
- ✓ `insert()` : Ajoute une nouvelle entrée à la position spécifiée dans la liste.
- ✓ `pop()` : Supprime une entrée à la fin de la liste.
- ✓ `remove()` : Supprime une entrée à la position spécifiée dans la liste.

Les étapes qui suivent vous montrent comment modifier une liste. Vous retrouverez plus tard les mêmes fonctions appliquées dans des situations plus concrètes. Pour l'instant, il s'agit simplement de se

familiariser avec la gestion des listes.

1. **Ouvrez une fenêtre de Python en mode Shell.**
2. **Tapez `List1 = []` et appuyez sur Entrée.**

Python crée une liste appelée `List1`.



Ici, les crochets sont vides. Autrement dit, `List1` ne contient aucune entrée. Il est possible de spécifier des listes vides qui seront remplies par la suite. En fait, c'est précisément ainsi que de nombreuses listes débutent, car vous ne savez pas toujours quelles informations elles vont contenir jusqu'à ce que l'utilisateur commence à interagir avec elles.

3. **Tapez `len(List1)` et appuyez sur Entrée.**

La fonction `len()` renvoie la valeur 0, comme l'illustre la [Figure 12.10](#). Ce qui est normal, puisque la liste est pour l'instant vide. Dans ce cas, vous savez que vous ne pouvez pas effectuer certaines actions, comme supprimer un élément, puisque justement il n'y en a pas encore.

Figure 12.10 :

N'hésitez pas dans vos applications à vérifier si une liste est vide.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = []
>>> len(List1)
0
>>>
```

4. **Tapez `List1.append(1)` et appuyez sur Entrée.**
5. **Tapez `len(List1)` et appuyez sur Entrée.**

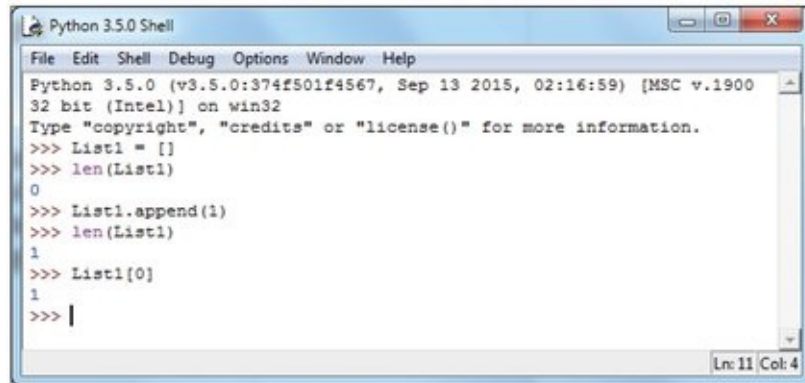
La fonction `len()` renvoie maintenant une longueur égale à 1.

6. **Tapez `List1[0]` et appuyez sur Entrée.**

Vous constatez que le premier élément de la liste `List1` vaut bien 1 (voir la [Figure 12.11](#)).

Figure 12.11 :

Ajouter un élément change la longueur de la liste et place cet élément à la fin de celle-ci.



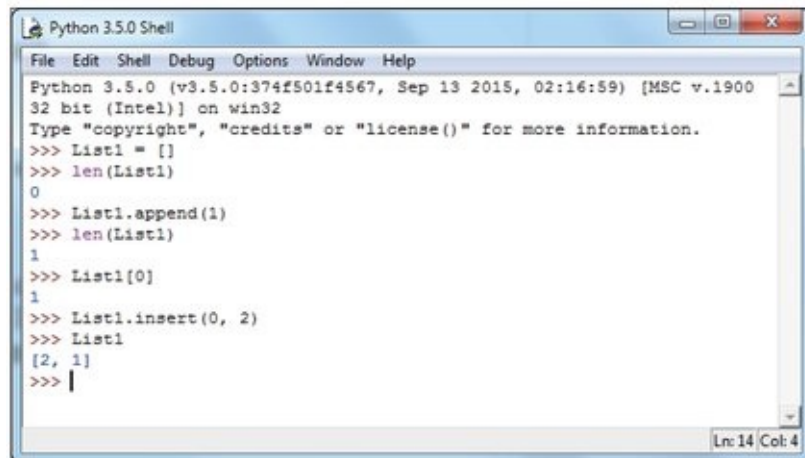
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = []
>>> len(List1)
0
>>> List1.append(1)
>>> len(List1)
1
>>> List1[0]
1
>>> |
```

7. **Tapez `List1.insert(0, 2)` et appuyez sur Entrée.**

La fonction `insert()` nécessite deux arguments. Le premier est l'indice de l'insertion, en l'occurrence l'élément de rang 0. Le second est l'objet que vous voulez insérer à cet emplacement, donc 2 dans ce cas.

Figure 12.12 :

Vous pouvez ajouter de nouveaux éléments à la position voulue.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = []
>>> len(List1)
0
>>> List1.append(1)
>>> len(List1)
1
>>> List1[0]
1
>>> List1.insert(0, 2)
>>> List1
[2, 1]
>>> |
```

8. **Tapez `List1` et appuyez sur Entrée.**

Python a ajouté le nouvel élément à la liste. Cet élément est venu se placer tout au début de celle-ci, comme l'illustre la [Figure 12.12](#).

9. **Tapez `List2 = List1.copy()` et appuyez sur Entrée.**

La nouvelle liste, `List2`, est une copie exacte de `List1`. Cette technique est souvent employée pour créer une version temporaire d'une liste de manière à ce que l'utilisateur puisse y faire des modifications sans toucher à la liste originale, ce qui peut éviter de nombreux problèmes.

Lorsque l'utilisateur a terminé, l'application peut alors décider si la liste temporaire doit simplement être détruite, ou si elle doit être recopiée dans la liste d'origine, ce qui permet de valider les modifications apportées.

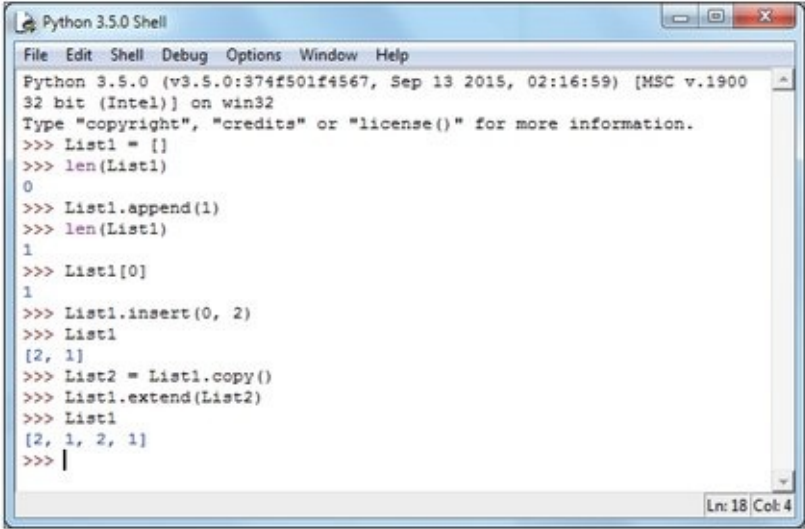
10. **Tapez `List1.extend(List2)` et appuyez sur Entrée.**

Python recopie le contenu de `List2` à la fin de `List1`. L'extension est couramment utilisée pour consolider deux listes.

11. **Tapez `List1` et appuyez sur Entrée.**

Vous constatez que la copie et l'extension ont bien fonctionné. `List1` contient maintenant les valeurs 2, 1, 2 et 1 (voir la [Figure 12.13](#)).

Figure 12.13 : La copie et l'extension fournissent des méthodes pour déplacer rapidement des données.

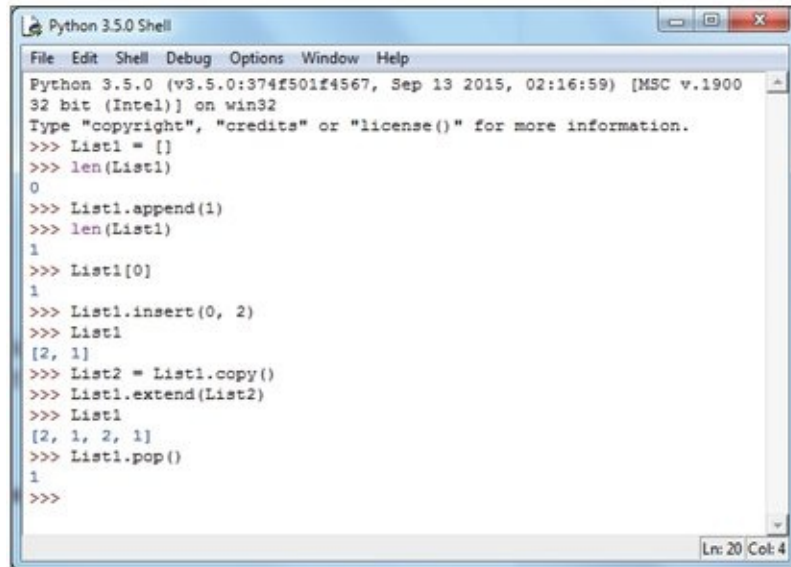


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = []
>>> len(List1)
0
>>> List1.append(1)
>>> len(List1)
1
>>> List1[0]
1
>>> List1.insert(0, 2)
>>> List1
[2, 1]
>>> List2 = List1.copy()
>>> List1.extend(List2)
>>> List1
[2, 1, 2, 1]
>>> |
```

12. **Tapez `List1.pop()` et appuyez sur Entrée.**

Python affiche comme résultat 1 (voir la [Figure 12.14](#)). Ce 1 était le dernier élément de la liste. La fonction `pop()` l'a supprimé tout en indiquant ce qu'il détruisait.

Figure 12.14 : La fonction `pop()` supprime les éléments placés en fin de liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> List1 = []
>>> len(List1)
0
>>> List1.append(1)
>>> len(List1)
1
>>> List1[0]
1
>>> List1.insert(0, 2)
>>> List1
[2, 1]
>>> List2 = List1.copy()
>>> List1.extend(List2)
>>> List1
[2, 1, 2, 1]
>>> List1.pop()
1
>>>
```

13. **Tapez `List1.remove(1)` et appuyez sur Entrée.**
Cette fois, Python supprime l'élément de rang 1. Contrairement à la fonction `pop()`, `remove()` n'affiche pas la valeur de l'élément supprimé.
14. **Tapez `List1.clear()` et appuyez sur Entrée.**
La liste est maintenant nettoyée de tous ses éléments.
15. **Tapez `len(List1)` et appuyez sur Entrée.**
Vous constatez que la sortie est bien égale à 0. La liste `List1` est définitivement vide. À ce stade, vous venez d'essayer toutes les méthodes de modification des listes fournies par Python. Vous pouvez bien entendu prolonger cet exemple avec vos propres manipulations jusqu'à ce que vous vous sentiez à l'aise avec cette gestion des listes.
16. **Refermez la fenêtre de Python.**



**Utiliser des opérateurs
avec les listes**

Les listes peuvent aussi se servir d'opérateurs pour effectuer certaines tâches. Par exemple, si vous voulez créer une liste contenant quatre occurrences du mot *Bonjour*, vous pouvez écrire `MaListe = ['Bonjour'] * 4` afin de la remplir. L'opérateur de multiplication (`*`) indique combien de fois l'élément indiqué doit être répété. Il est essentiel de bien comprendre que chaque nouvel élément est totalement distinct des autres. Par conséquent, le contenu de `MaListe` serait ici `[' Bonjour', 'Bonjour', 'Bonjour', 'Bonjour']`.

Vous pouvez aussi remplir une liste en faisant appel à la concaténation. Par exemple, l'instruction `MaListe = [« Bonjour »] + [« le monde »] + [« ! »] * 4` créerait six éléments dans `MaListe`. Le premier serait `'Bonjour'`, suivi de `'le monde'`, puis de quatre points d'exclamation.

L'opérateur `in` fonctionne également avec les listes. Dans ce chapitre, nous allons utiliser une méthode directe et simple (qui est l'approche recommandée). Cependant, vous pouvez parfaitement aussi vouloir aller droit au but en écrivant par exemple `'Bonjour' in MaListe`. En supposant que `MaListe` est celle créée au paragraphe précédent, la sortie de cette instruction serait `True` (vrai).

Faire des recherches dans les listes

Il n'est pas très facile de modifier une liste si vous ne

savez pas ce qu'elle contient. La possibilité d'effectuer des recherches dans les listes est donc une tâche essentielle. Les étapes qui suivent démontrent comment vous pouvez rechercher une valeur spécifique dans une liste. Cet exemple peut également être trouvé dans le fichier téléchargeable

`SearchList.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Colors = ["Rouge", "Orange", "Jaune", "Vert", "Bleu"]

ColorSelect = ""

while str.upper(ColorSelect) != "QUITTER":
```

```
    ColorSelect = input("Sélectionnez le nom d'une couleur: ")
    if (Colors.count(ColorSelect) >= 1):
        print("La couleur existe dans la liste !")
    elif (str.upper(ColorSelect) != "QUITTER"):
        print("La liste ne contient pas cette couleur.")
```

L'exemple commence par créer une liste nommée `Colors`. Celle-ci contient des noms de couleurs. Il crée également une variable appelée `ColorSelect` destinée à contenir le nom de la couleur que l'utilisateur veut retrouver. L'application entre ensuite dans une boucle où elle demande de saisir un nom de couleur, qui est placé dans `ColorSelect`. Tant que cette variable ne contient pas le mot QUITTER (ou toute variante de ce mot, l'application le convertissant en majuscules), la boucle répète sa question.

Lorsque l'utilisateur entre un nom de couleur, l'application demande à la liste de compter le nombre d'occurrences de ce mot. Si cette valeur est au moins égale à 1, cela signifie que la liste contient bien ce mot, et un message est affiché en conséquence. Dans le cas contraire, l'application affiche un message d'erreur.



Cet exemple utilise une clause `elif` pour tester le cas où la version en majuscules de `colorSelect` contient le mot `QUITTER`. Cette technique permet de s'assurer qu'un message particulier ne sera pas émis si l'utilisateur veut terminer l'application (mais on pourrait bien entendu lui faire dire aussi : Au revoir !). Cette pratique est particulièrement utile pour éviter toute confusion, voire même une perte de données dans le cas où l'application effectuerait une tâche non désirée.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application vous demande d'entrer le nom d'une couleur.

Figure 12.15 : Un message de succès est émis si la couleur appartient bien à la liste.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre12\SearchList.py =====
Sélectionnez le nom d'une couleur: Bleu
La couleur existe dans la liste !
Sélectionnez le nom d'une couleur: |
```

4. **Tapez Bleu et appuyez sur Entrée.**

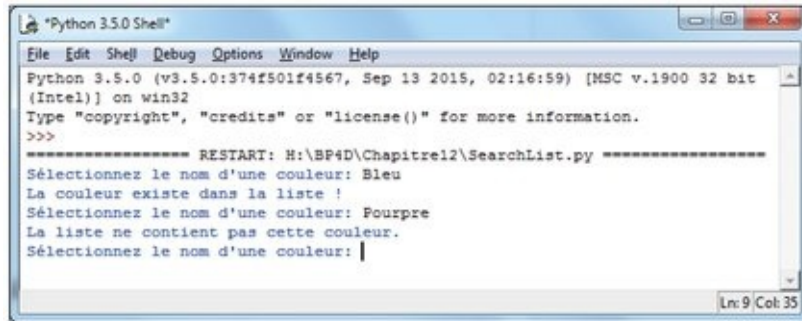
Un message vous indique cette couleur appartient bien à la liste (voir la [Figure 12.15](#)).

5. **Tapez Pourpre et appuyez sur Entrée.**

Cette fois, un message vous informe que cette couleur n'appartient pas à la liste (voir la [Figure 12.16](#)).

Figure 12.16 :

Entrer une couleur qui n'appartient pas à la liste génère un message d'échec.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre12\SearchList.py =====
Sélectionnez le nom d'une couleur: Bleu
La couleur existe dans la liste !
Sélectionnez le nom d'une couleur: Pourpre
La liste ne contient pas cette couleur.
Sélectionnez le nom d'une couleur: |
```

6. Tapez Quitter et appuyez sur Entrée.

L'application se termine en douceur. Aucun message n'est émis à la sortie.

Trier des listes

L'ordinateur peut localiser des informations dans une liste indépendamment de leur position dans celle-ci. Mais, en fait, ces recherches sont bien plus faciles dans le cas de listes importantes si celles-ci sont classées d'une certaine manière. Pour autant, la raison principale justifiant le fait de trier une liste est tout simplement de simplifier la lecture des informations par l'utilisateur. Les êtres humains s'y retrouvent plus facilement lorsque les choses sont bien classées.

L'exemple qui suit part d'une liste dont les éléments sont placés dans un ordre quelconque. La liste est ensuite triée, et son contenu est affiché. Vous le retrouverez également dans le fichier téléchargeable

SortList.py.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```

Colors = ["Rouge", "Orange", "Jaune", "Vert", "Bleu"]

for Item in Colors:
    print(Item, end=" ")

print()

Colors.sort()

for Item in Colors:
    print(Item, end=" ")

print()

```

Cet exemple commence par créer une liste de couleurs dans un ordre quelconque. Cette liste est affichée pour montrer cet apparent désordre. Notez le end final sur la ligne de la fonction `print()`. Il permet de s'assurer que tous les éléments sont affichés sur une seule et même ligne.



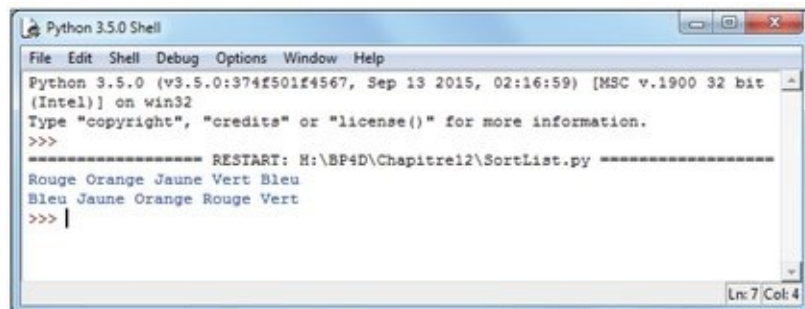
Trier une liste est très simple. Une fois la fonction `sort()` appelée, la liste peut être affichée dans l'ordre alphabétique.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche les deux états de la liste : non triée, puis triée (voir la [Figure 12.17](#)).

Figure 12.17 :

Trier une liste demande simplement à appeler la fonction `sort()`.



Le tri peut aussi s'effectuer dans l'ordre inverse. Pour cela, vous utilisez la fonction `reverse()`. Celle-ci doit

figurer sur une ligne différente. Dans ce cas, l'exemple précédent ressemblerait à ceci :

```
Colors = ["Rouge", "Orange", "Jaune", "Vert", "Bleu"]

for Item in Colors:
    print(Item, end=" ")

print()

Colors.sort()
Colors.reverse()

for Item in Colors:
    print(Item, end=" ")

print()
```

Travailler avec l'objet Counter

Il arrive que vous ayez une source de données, et que vous vouliez simplement savoir combien de fois une certaine chose se produit (comme l'apparence d'un certain élément dans la liste). Si celle-ci est courte, vous pouvez simplement compter les éléments. Par contre, si la liste est réellement longue, il est pratiquement impossible d'obtenir un résultat parfaitement fiable. Supposons ainsi que vous ayez décidé d'écrire un roman appelé *Guerre et paix*. Vous voulez alors connaître la fréquence de certains mots dans ce roman-fleuve. Cette tâche serait évidemment impossible sans l'aide d'un ordinateur.



L'objet Counter vous permet de compter rapidement des éléments. Il est de plus incroyablement facile à utiliser. Vous ne retrouverez dans d'autres circonstances dans ce livre, mais ici il va uniquement nous servir avec des listes. L'exemple développé dans ce qui suit crée une liste contenant des éléments répétitifs, puis il compte le nombre d'occurrences de ces éléments. Vous pouvez également le retrouver dans le fichier téléchargeable `UseCounterWithList.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from collections import Counter

MyList = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1, 5]
ListCount = Counter(MyList)

print(ListCount)

for ThisItem in ListCount.items():
    print("L'élément : ", ThisItem[0],
          " apparaît : ", ThisItem[1], " fois")

print("La valeur 1 apparaît {0} fois."
      .format(ListCount.get(1)))
```

Pour utiliser l'objet `Counter`, vous devez l'importer depuis la bibliothèque `collections`. Bien entendu, si vous devez travailler dans votre application avec d'autres types de collections, vous pouvez importer la totalité de ce module en tapant l'instruction `import collections`.

Cet exemple commence par créer une liste appelée `MyList`. Celle-ci contient des valeurs numériques répétées. Vous pouvez facilement voir que certains de ces éléments apparaissent plusieurs fois. Cette liste est placée dans un objet de type `Counter` appelé `ListCount`. Vous pouvez créer des objets `Counter` de différentes manières, mais celle-ci est la mieux adaptée dans le cas des listes.



L'objet `Counter` et la liste ne sont en aucun cas connectés. Si le contenu de la liste change, vous devrez recréer l'objet `Counter`, car il n'est pas actualisé dans ce cas. Une alternative consiste à appeler d'abord la méthode `clear()`, suivie d'un appel à `update()` pour remplir l'objet `Counter` avec les nouvelles données.

L'application affiche la valeur de `ListCount` de diverses manières. La première sortie correspond à ce que vaut le

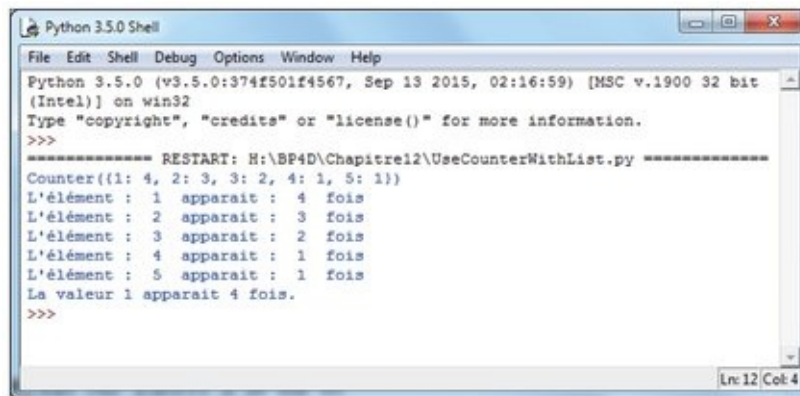
compteur `Counter` sans aucune manipulation. La seconde sortie affiche les différents éléments de `MyList` ainsi que le nombre de fois où ils apparaissent dans la liste. Pour obtenir ce résultat, vous utilisez la fonction `items()`. Enfin, l'exemple montre comment obtenir le décompte pour un élément individuel en faisant appel à la fonction `get()`.

3. Choisissez la commande Run Module dans le menu Run.

Une fenêtre Python en mode Shell va s'ouvrir. Vous pouvez alors voir les résultats renvoyés en utilisant l'objet `Counter` (voir la [Figure 12.18](#)).

Figure 12.18 :

L'objet `Counter` est utile pour obtenir des statistiques intéressantes avec de longues listes.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre12\UseCounterWithList.py =====
Counter({1: 4, 2: 3, 3: 2, 4: 1, 5: 1})
L'élément : 1 apparaît : 4 fois
L'élément : 2 apparaît : 3 fois
L'élément : 3 apparaît : 2 fois
L'élément : 4 apparaît : 1 fois
L'élément : 5 apparaît : 1 fois
La valeur 1 apparaît 4 fois.
>>>
```

Remarquez que les informations sont enregistrées dans l'objet `Counter` sous la forme d'une paire formée d'une clé et d'une valeur. Le Chapitre 13 reviendra sur ce sujet plus en détail. Tout ce que vous avez besoin de savoir pour l'instant, c'est que l'élément trouvé dans `MyList` devient dans `ListCount` une clé qui identifie le nom unique de l'élément. La valeur contient le nombre de fois où cet élément apparaît dans `MyList`.

Chapitre 13

Collecter toutes sortes de données

Dans ce chapitre :

- ▶ Définir une collection.
- ▶ Utiliser des tuples.
- ▶ Utiliser des dictionnaires.
- ▶ Développer des piles en utilisant des listes.
- ▶ Utiliser le module queue.
- ▶ Utiliser des dequeues.

Les gens collectent toutes sortes de choses. Il peut s'agir de piles de disques, de cartes postales, de bouchons de champagne, de timbres, bref tout ce que vous voulez. Les collections que vous rencontrez lorsque vous écrivez des applications ne sont pas différentes de ce qui se passe dans le monde réel. Une *collection* est tout simplement un groupe d'éléments semblables, rangés dans un même lieu, et généralement organisés sous une forme facilement compréhensible.



Ce chapitre traite des collections au sens large. L'idée

centrale qui préside à toute collection consiste à créer un environnement dans lequel la collection est correctement gérée, et vous permettant de localiser avec précision ce que vous voulez à n'importe quel moment. Par exemple, des rayonnages sont parfaits pour y classer des livres, des DVD, et toutes sortes d'objets plats. Par contre, des timbres seront bien mieux dans un classeur adapté. Et on pourrait multiplier les exemples. Même si le mode de rangement change, cela ne retire rien au fait qu'il s'agit à chaque fois de collections. Ceci vaut également pour les collections traitées par des ordinateurs. Certes, il y a une différence entre une *pile* et une *file*, mais cela ne change rien à l'essentiel : vous avez besoin de méthodes qui vous permettent de bien gérer vos données et d'y accéder facilement lorsque vous en avez besoin.

Comprendre les collections

Le Chapitre 12 vous a initié aux séquences. Une *séquence* est une succession de valeurs qui sont rassemblées dans un conteneur. La séquence la plus simple est la chaîne, qui est une succession de caractères. Viennent ensuite les listes décrites dans le Chapitre 12, qui sont des successions d'objets. Même si une chaîne et une liste sont toutes deux des séquences, elles possèdent des différences significatives. Dans le cas d'une chaîne, par exemple, vous pouvez mettre tous les caractères en minuscules, ce qui n'a pas de sens avec une liste. D'un autre côté, les listes vous permettent d'ajouter de nouveaux éléments, ce qu'une chaîne ne supporte pas. Les connexions sont tout simplement un autre type de séquence, mais un peu plus complexe que les



chaînes et les listes.

Quelle que soit la séquence que vous utilisez, elle supporte deux fonctions : `index ()` et `count()`. La fonction `index ()` renvoie la position d'un élément spécifié dans une séquence. Vous pouvez par exemple obtenir ainsi la position d'un caractère dans une chaîne ou d'un objet dans une liste. La fonction `count ()` retourne le nombre d'occurrences d'un élément spécifique dans une liste. La nature de cet élément spécifique dépend évidemment de celle de la liste.

Vous pouvez utiliser des collections pour créer avec Python des structures de type base de données. Chaque type de collection a un objet différent, et vous utilisez ces divers types de manière spécifique. L'idée importante à retenir ici est que les collections sont une autre forme de séquence. Et, à ce titre, elles supportent également les fonctions `index ()` et `count ()`.

Python est conçu pour être extensible. Cependant, il possède un jeu de base de collections qui peuvent servir à créer la plupart des applications. Ce Chapitre décrit les collections les plus courantes :

➤ **Tuple** : Un tuple est une collection servant à créer des séquences semblables à des listes, mais en plus complexes. L'un des avantages est que vous pouvez imbriquer le contenu d'un tuple. Cette fonctionnalité vous permet par exemple de créer des structures servant à gérer des informations sur le personnel d'une entreprise, ou encore des couples de coordonnées x-y.

➤ **Dictionnaire** : Comme avec un vrai dictionnaire, vous pouvez créer des paires

clé/valeur (sur le modèle mot/définition). Un dictionnaire permet des recherches incroyablement rapides, et rend le classement des données beaucoup plus faciles.

✓ *queue* : Ce terme désigne une file, c'est-à-dire une collection de type FIFO (First In, First Out, ou premier entré, premier sorti). C'est le principe de la file d'attente devant un guichet.



Python propose une variante appelée LifoQueue, soit une collection de type LIFO (Last In, First Out, ou dernier entré premier sorti). C'est le principe de la pile, où l'on prend d'abord l'élément du dessus, celui qui a été posé en dernier.

✓ *deque* : Sous ce terme barbare se cache une queue à double fin, c'est-à-dire une file dans laquelle vous pouvez ajouter ou supprimer des éléments au début ou à la fin, mais pas au milieu. Vous pouvez utiliser un deque en tant que collection de type *queue*, ou comme pile, ou tout autre type de collection dans laquelle vous ajouter ou supprimez des éléments de manière ordonnée (à la différence des listes, des tuples et des dictionnaires qui autorisent un accès aléatoire).

Travailler avec les tuples

Un *tuple* est une collection servant à créer des listes complexes, dans lesquelles un tuple peut être imbriqué dans un autre. Ceci vous permet donc de créer des hiérarchies. Une hiérarchie peut être aussi simple que l'arborescence de votre disque dur, mais il

peut aussi s'agir par exemple de l'organigramme complet de votre entreprise. L'idée est que vous pouvez créer des structures de données complexes avec un tuple.



Les tuples sont « immutables », ce qui signifie que vous ne pouvez pas les changer. Vous pouvez créer un nouveau tuple portant le même nom, et le modifier d'une certaine manière, mais vous ne pouvez pas le faire avec un tuple existant. Par contre, les listes, elles, sont mutables, et donc modifiables. Cette nature des tuples peut paraître un désavantage, mais cette immutabilité a toutes sortes de vertus, notamment en termes de sécurité et de rapidité. De plus, ce type d'objet est plus facile à gérer avec des systèmes multiprocesseurs.

Les étapes qui suivent vous montrent comment vous pouvez interagir avec un tuple dans Python :

1. **Ouvrez une fenêtre de Python en mode Shell.**
Vous voyez l'indicatif habituel.
2. **Tapez `MyTuple = ('Rouge', 'Vert', 'Bleu')` et appuyez sur Entrée.**

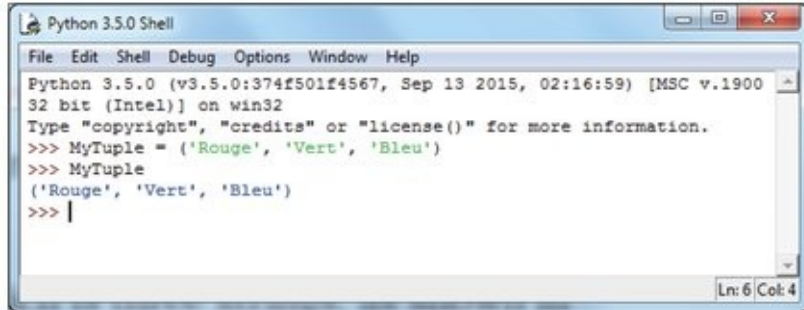


Comme nous l'avons vu dans le Chapitre 12, les éléments d'une liste sont placés entre des crochets droits. Ceux d'un tuple sont délimités par des parenthèses.

Python crée un tuple contenant trois chaînes.

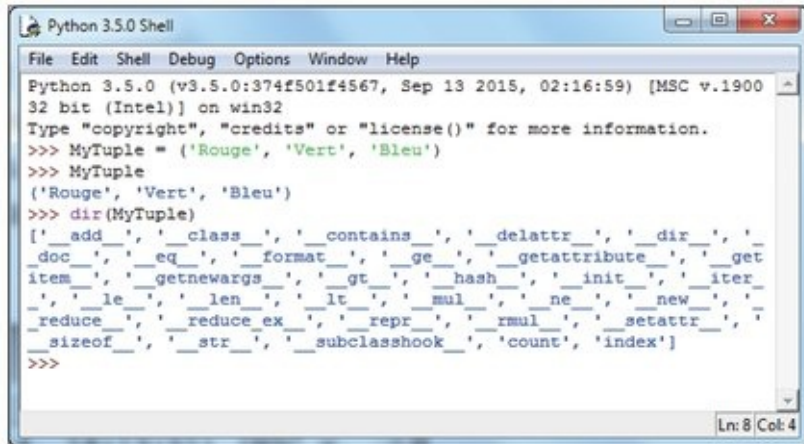
3. **Tapez `MyTuple` et appuyez sur Entrée.**
Python affiche le contenu de **MyTuple** (voir la [Figure 13.1](#)). Remarquez que les chaînes sont délimitées par des apostrophes, même si vous avez utilisé des guillemets pour les définir.

Figure 13.1 : Les tuples utilisent des parenthèses, pas des crochets droits.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> MyTuple = ('Rouge', 'Vert', 'Bleu')
>>> MyTuple
('Rouge', 'Vert', 'Bleu')
>>>
```

Figure 13.2 : Peu de fonctions semblent utilisables avec les tuples.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> MyTuple = ('Rouge', 'Vert', 'Bleu')
>>> MyTuple
('Rouge', 'Vert', 'Bleu')
>>> dir(MyTuple)
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattr_', '_get_
item_', '_getnewargs_', '_gt_', '_hash_', '_init_', '_iter_
_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_', '_
reduce_', '_reduce_ex_', '_repr_', '_rmul_', '_setattr_', '_
sizeof_', '_str_', '_subclasshook_', 'count', 'index']
>>>
```

4. Tapez `dir(MyTuple)` et appuyez sur Entrée.

Python affiche la liste des fonctions que vous pouvez utiliser avec les tuples (voir la [Figure 13.2](#)). Remarquez le peu de fonctions utilisables, ainsi que la présence des fonctions `index()` et `count()`.



Les apparences sont parfois trompeuses. Par exemple, vous pouvez ajouter de nouveaux éléments avec la fonction `__add__()`. Lorsque vous travaillez avec des objets Python, regardez toutes les entrées avant de prendre la bonne décision.

5. Tapez `MyTuple = MyTuple.__add__('Pourpre',)` et appuyez sur Entrée.

Ce code ajoute un tuple à `MyTuple`, et place le résultat dans une nouvelle copie de `MyTuple`. L'ancienne version de `MyTuple` est détruite après cet appel.



La fonction `__add__()` accepte uniquement des tuples en entrée. Ceci signifie que l'ajout doit être placé entre parenthèses. De plus, lorsque vous créez un tuple ne comprenant qu'une seule entrée, celle-ci doit être suivie d'une virgule. C'est une règle propre à Python. À défaut, vous obtiendrez un message d'erreur comme celui-ci :

```
TypeError: can only concatenate tuple (not "str") to tuple
```

Figure 13.3 :

Cette nouvelle copie de `MyTuple` contient une entrée supplémentaire.

```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> MyTuple = ('Rouge', 'Vert', 'Bleu')
>>> MyTuple
('Rouge', 'Vert', 'Bleu')
>>> dir(MyTuple)
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattribute_', '_getit
em_', '_getnewargs_', '_gt_', '_hash_', '_init_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_', '_redu
ce_', '_reduce_ex_', '_repr_', '_rmul_', '_setattr_', '_size
of_', '_str_', '_subclasshook_', '_count_', '_index_']
>>> MyTuple = MyTuple.__add__(('Pourpre',))
>>> MyTuple
('Rouge', 'Vert', 'Bleu', 'Pourpre')
>>> |
```

6. **Tapez `MyTuple` et appuyez sur Entrée.**

L'ajout à **MyTuple** apparaît en fin de liste, comme l'illustre la [Figure 13.3](#). Notez qu'il se situe au même niveau que les autres éléments.

7. **Tapez `MyTuple = MyTuple.__add__(('Jaune', ('Orange', 'Noir')))` et appuyez sur Entrée.**

Cette étape ajoute trois nouvelles couleurs. Cependant, Orange et Noir sont insérés en tant que tuple à l'intérieur du tuple principal, ce qui crée une hiérarchie. Ces deux entrées sont traitées comme une seule entité à l'intérieur du tuple principal.



Vous pouvez remplacer la fonction `__add__()` par l'opérateur de concaténation. Si vous vouliez par exemple ajouter la couleur Magenta en début de liste, vous pourriez taper **`MyTuple = ('Magenta',) + MyTuple`**.

8. **Tapez `MyTuple[4]` et appuyez sur Entrée.**

Python affiche le membre de **`MyTuple`** ayant pour rang 4, c'est-à-dire Jaune. Les tuples utilisent des indices pour accéder à leurs membres individuels, exactement comme les listes. Vous pouvez également définir une plage. Tout ce qu'il est possible de faire avec des indices dans une liste l'est également avec un tuple.

9. **Tapez `MyTuple[5]` et appuyez sur Entrée.**

Vous voyez maintenant un tuple qui contient les chaînes Orange et Noir. Bien entendu, votre but n'est pas d'utiliser ces éléments sous la forme d'un tuple.



Vous pouvez déterminer si un certain indice a renvoyé un autre tuple, plutôt qu'une valeur, en testant son type. Dans ce cas, par exemple, l'instruction **`type(MyTuple[5]) == tuple`** renverrait la valeur `True`.

10. **Tapez `MyTuple[5][0]` et appuyez sur Entrée.**

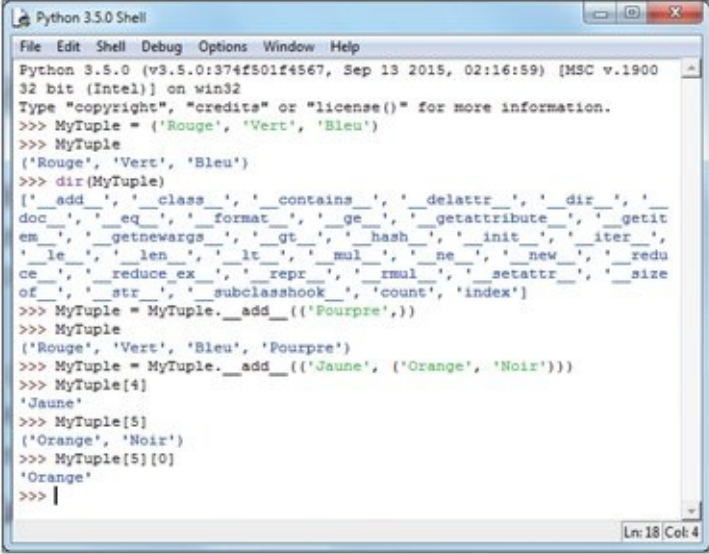
La sortie va maintenant afficher la couleur Orange, autrement dit le premier élément du tuple ajouté lors de l'Étape 7. La [Figure 13.4](#) illustre la progression de cet exemple. Les indices apparaissent toujours dans l'ordre de leur niveau dans la hiérarchie.



En utilisant une combinaison d'indices et la fonction `__add__()` (ou l'opérateur de concaténation), vous pouvez créer des applications évolutives basées sur des tuples. Par exemple, vous pouvez supprimer un

élément d'un tuple en le rendant égal à une plage de valeurs. Vous pourriez supprimer de cette manière le tuple qui contient les chaînes Orange et Noir en tapant **MyTuple = MyTuple[0 :5]**.

Figure 13.4 : Les indices permettent d'accéder aux membres individuels d'un tuple.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> MyTuple = ('Rouge', 'Vert', 'Bleu')
>>> MyTuple
('Rouge', 'Vert', 'Bleu')
>>> dir(MyTuple)
['_add_', '_class_', '_contains_', '_delattr_', '_dir_', '_doc_', '_eq_', '_format_', '_ge_', '_getattr_', '_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_', '_iter_', '_le_', '_len_', '_lt_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_rmul_', '_setattr_', '_size_', '_str_', '_subclasshook_', '_count_', '_index_']
>>> MyTuple = MyTuple._add_ (('Pourpre',))
>>> MyTuple
('Rouge', 'Vert', 'Bleu', 'Pourpre')
>>> MyTuple = MyTuple._add_ (('Jaune', ('Orange', 'Noir')))
>>> MyTuple[4]
'Jaune'
>>> MyTuple[5]
('Orange', 'Noir')
>>> MyTuple[5][0]
'Orange'
>>> |
```

Travailler avec les dictionnaires

Un dictionnaire Python fonctionne selon le même principe que votre *Larousse* ou votre *Robert* préféré. Vous créez une clé qui est associée à une valeur (pour un dictionnaire du monde réel, vous remplacez juste *clé* par *mot*, et *valeur* par *définition*). Comme les listes, les dictionnaires sont mutables, ce qui signifie que vous pouvez les modifier selon vos besoins. Le principal intérêt des dictionnaires est d'accélérer les recherches. Une clé est toujours unique et courte, ce qui fait que l'ordinateur passe beaucoup moins de temps à rechercher les données dont vous avez besoin.

Les sections qui suivent montrent comment créer et utiliser des dictionnaires. Grâce à eux, il est possible

de pallier les déficiences du langage Python. Ainsi, la plupart des langages de programmation possèdent une instruction `switch` qui sert essentiellement à proposer un menu dans lequel l'utilisateur fait son choix. Or, cette instruction manque à Python, ce qui fait que vous devez faire appel à des instructions `if...elif` pour effectuer ce genre de tâche (certes, cela fonctionne, mais cette méthode manque de clarté).

Créer et utiliser un dictionnaire

Le principe ressemble beaucoup à celui des listes, si ce n'est que vous devez maintenant définir des paires clé/valeur. Il faut pour cela respecter certaines règles :

- ✓ **La clé doit être unique.** Si vous entrez une clé qui existe déjà, elle vient simplement remplacer la définition antérieure.
- ✓ **La clé doit être immuable.** Cela signifie que la clé peut être une chaîne, un nombre ou encore un tuple. Mais elle ne peut pas être une liste.

Il n'y a aucune restriction sur les valeurs associées aux clés. Il peut s'agir d'un objet Python quelconque, ce qui vous permet par exemple de vous servir d'un dictionnaire pour accéder aux informations sur les membres du personnel de votre entreprise, ou toute autre donnée complexe. Les étapes qui suivent vous aideront à mieux comprendre le fonctionnement de ces dictionnaires.

1. **Ouvrez une fenêtre de Python en mode Shell.**
Vous voyez l'indicatif habituel.
2. **Tapez `Colors = {'Paul' : 'Bleu', 'Annie' : 'Rouge', 'Sarah' : 'Jaune'}` et appuyez sur Entrée.**

Python vient de créer un dictionnaire comprenant trois entrées : des prénoms de personnes, et la couleur favorite de chacune. Remarquez la manière dont les entrées du dictionnaire sont fabriquées. La clé vient en premier, suivie d'un deux-points, puis de la valeur. Les entrées sont séparées l'une de l'autre par une virgule.

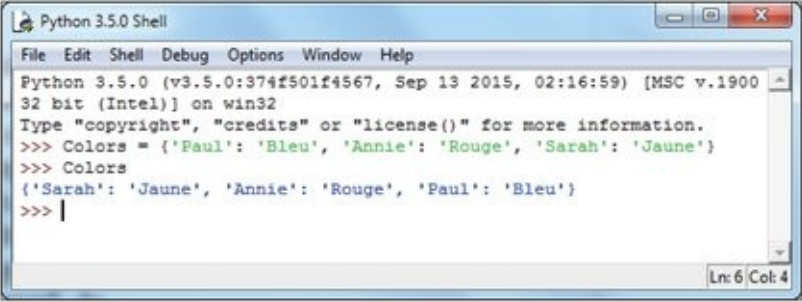
3. **Tapez `Colors` et appuyez sur Entrée.**

Python affiche les paires clé/valeur, comme l'illustre la [Figure 13.5](#). Remarquez que les clés sont triées selon un certain ordre dont Python a le secret. Un dictionnaire organise automatiquement les clés pour rendre leur accès plus rapide, ce qui signifie que les temps de réponse seront très courts, même dans le cas d'un volume de données important. La contrepartie, c'est que la création d'un dictionnaire est plus lente que celle d'une liste, puisque l'ordinateur doit effectuer ce travail de classement.

4. **Tapez `Colors['Annie']` et appuyez sur Entrée.**

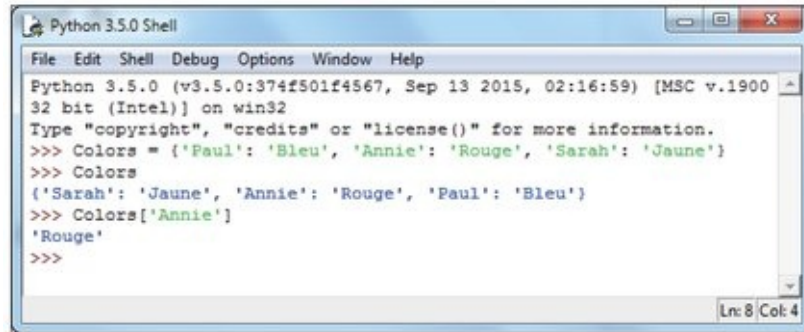
Vous voyez la couleur associée à Annie, le Rouge (voir la [Figure 13.6](#)). Utiliser une chaîne comme clé, plutôt qu'un index numérique, facilite la lecture du code ainsi que sa maintenance. En procédant ainsi, les dictionnaires vous permettront de gagner énormément de temps sur le long terme (c'est d'ailleurs pourquoi ils sont si populaires). Cependant, vous devez aussi prendre en compte le coût du temps nécessité par leur création, ainsi que la mobilisation supplémentaire de ressources que cela implique.

Figure 13.5 : Un dictionnaire place ses entrées dans un certain ordre.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Colors = {'Paul': 'Bleu', 'Annie': 'Rouge', 'Sarah': 'Jaune'}
>>> Colors
{'Sarah': 'Jaune', 'Annie': 'Rouge', 'Paul': 'Bleu'}
>>> |
```

Figure 13.6 : Les dictionnaires rendent les valeurs facilement accessibles et bien documentées.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Colors = {'Paul': 'Bleu', 'Annie': 'Rouge', 'Sarah': 'Jaune'}
>>> Colors
{'Sarah': 'Jaune', 'Annie': 'Rouge', 'Paul': 'Bleu'}
>>> Colors['Annie']
'Rouge'
>>>
```

5. **Tapez Colors.keys() et appuyez sur Entrée.**

Python affiche la liste des clés présentes dans le dictionnaire Colors (voir la [Figure 13.7](#)). Vous pouvez vous servir de ces clés pour automatiser l'accès au dictionnaire.

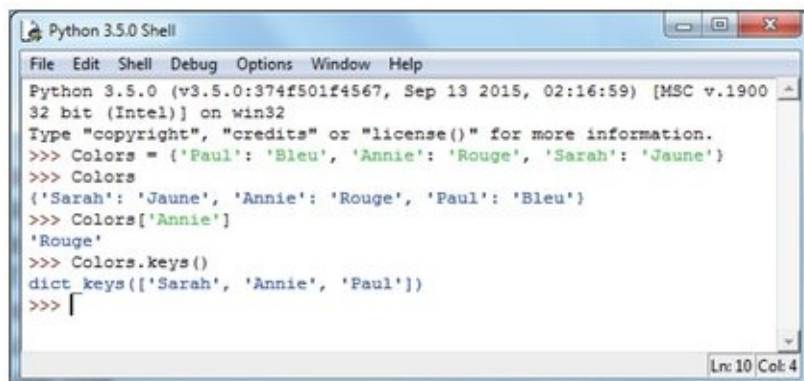
6. **Tapez le code suivant en appuyant sur Entrée après chaque ligne, et une seconde fois sur Entrée après la dernière ligne :**

```
for Item in Colors.keys():
    print("{0} aime la couleur {1}."
        .format(Item, Colors[Item]))
```



Cet exemple affiche tous les noms des utilisateurs ainsi que leur couleur préférée (voir la [Figure 13.8](#)). Utiliser des dictionnaires peut nettement faciliter l'affichage d'informations. De plus, un emploi judicieux des clés permet d'utiliser celles-ci dans les sorties.

Figure 13.7 : La liste des clés du dictionnaire Colors.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Colors = {'Paul': 'Bleu', 'Annie': 'Rouge', 'Sarah': 'Jaune'}
>>> Colors
{'Sarah': 'Jaune', 'Annie': 'Rouge', 'Paul': 'Bleu'}
>>> Colors['Annie']
'Rouge'
>>> Colors.keys()
dict_keys(['Sarah', 'Annie', 'Paul'])
>>> [
```

7. **Tapez Colors[« Sarah »] = « Pourpre » et appuyez sur Entrée.**

Le contenu du dictionnaire est mis à jour pour refléter la nouvelle couleur préférée de Sarah.

8. **Tapez Colors.update({'« Pierre »' : « Orange »}) et appuyez sur Entrée.**

Une nouvelle entrée est ajoutée au dictionnaire.

9. **Placez votre curseur à la fin de la troisième ligne de code saisie lors de l'Étape 6 et appuyez sur Entrée.**



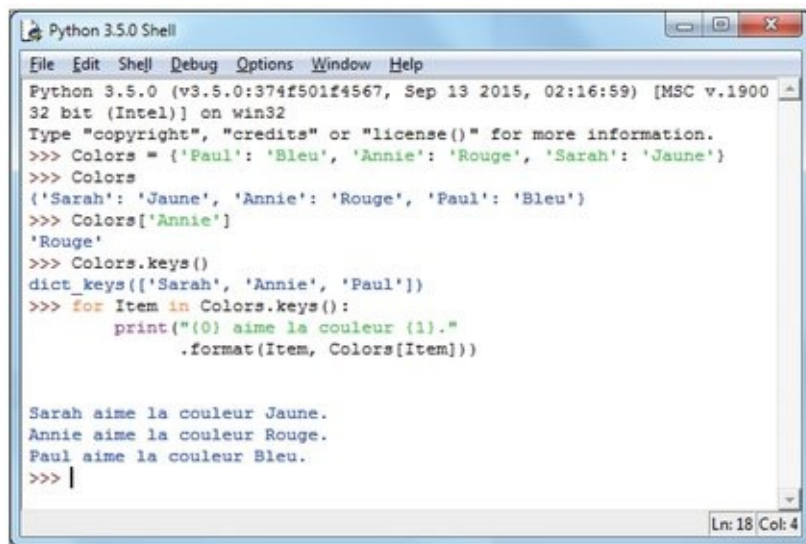
L'éditeur crée automatiquement une copie de ce code. Cette technique permet de gagner pas mal de temps lorsque vous faites des expériences dans le mode Shell de Python, puisque vous pouvez ainsi réutiliser du code sans avoir à le ressaisir.

10. **Appuyez deux fois sur Entrée.**

Vous voyez la sortie actualisée illustrée sur la [Figure 13.9](#). Remarquez que Pierre est ajouté à l'index du dictionnaire (qui n'est manifestement pas un simple ordre alphabétique). De plus, la nouvelle couleur préférée de Sarah est bien présente.

Figure 13.8 :

Créez des clés parlantes de manière à pouvoir les utiliser facilement dans les sorties.

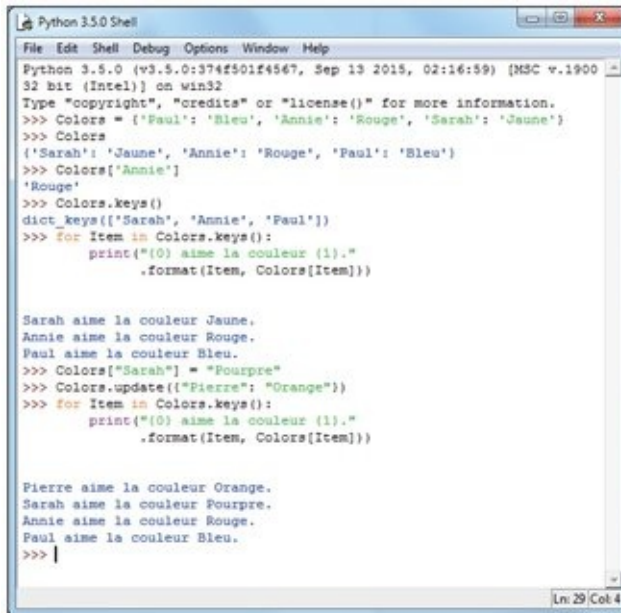


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Colors = {'Paul': 'Bleu', 'Annie': 'Rouge', 'Sarah': 'Jaune'}
>>> Colors
{'Sarah': 'Jaune', 'Annie': 'Rouge', 'Paul': 'Bleu'}
>>> Colors['Annie']
'Rouge'
>>> Colors.keys()
dict_keys(['Sarah', 'Annie', 'Paul'])
>>> for item in Colors.keys():
    print("{} aime la couleur {}".format(item, Colors[item]))

Sarah aime la couleur Jaune.
Annie aime la couleur Rouge.
Paul aime la couleur Bleu.
>>> |
```

11. **Tapez del Colors[« 'Paul'】 et appuyez sur Entrée.**
Python supprime l'entrée Paul du dictionnaire.
12. **Répétez les Étapes 9 et 10.**
Vous vérifiez que Paul a bien disparu.
13. **Tapez len(Colors) et appuyez sur Entrée.**
La valeur affichée, 3, vérifie que le dictionnaire contient trois entrées, et non plus quatre.
14. **Tapez Colors.clear() et appuyez sur Entrée.**
15. **Tapez len(Colors) et appuyez sur Entrée.**
La réponse est maintenant 0, puisque vous venez de vider le dictionnaire.
16. **Vous pouvez refermer la fenêtre de Python.**

Figure 13.9 : Les dictionnaires sont faciles à modifier.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Colors = {'Paul': 'Bleu', 'Annie': 'Rouge', 'Sarah': 'Jaune'}
>>> Colors
{'Sarah': 'Jaune', 'Annie': 'Rouge', 'Paul': 'Bleu'}
>>> Colors['Annie']
'Rouge'
>>> Colors.keys()
dict_keys(['Sarah', 'Annie', 'Paul'])
>>> for item in Colors.keys():
    print("{} aime la couleur {}".format(item, Colors[item]))

Sarah aime la couleur Jaune.
Annie aime la couleur Rouge.
Paul aime la couleur Bleu.
>>> Colors['Sarah'] = 'Pourpre'
>>> Colors.update({'Pierre': 'Orange'})
>>> for item in Colors.keys():
    print("{} aime la couleur {}".format(item, Colors[item]))

Pierre aime la couleur Orange.
Sarah aime la couleur Pourpre.
Annie aime la couleur Rouge.
Paul aime la couleur Bleu.
>>> |
```

Remplacer l'instruction switch par un dictionnaire

La plupart des langages de programmation disposent d'une instruction `switch` qui permet d'effectuer des sélections dans des menus. L'utilisateur dispose de plusieurs options et il doit en choisir une. Le programme effectue ensuite l'action correspondante.

Voici à quoi pourrait ressembler une telle instruction dans un autre langage (bien entendu, cela ne signifie rien pour Python) :

```
switch(n)
{
    case 0:
        print("Vous avez choisi bleu.");
        break;
    case 1:
```

```
        print("Vous avez choisi jaune.");
        break;
    case 2:
        print("Vous avez choisi vert.");
        break;
}
```

L'application présente normalement une interface de type menu, récupère une valeur correspondant à la sélection effectuée par l'utilisateur, puis effectue la ou les tâches définies à partir de l'instruction `switch`. Cette méthode est plus claire et plus simple que l'emploi d'une série d'instructions `if...elif` pour obtenir le même résultat.

Mais si Python ne propose pas de type d'instruction, vous pouvez tout de même la simuler en utilisant un dictionnaire. C'est ce que vous montrent les étapes suivantes. Vous pouvez également les retrouver dans le fichier téléchargeable `PythonSwitch.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
def PrintBlue():
    print("Vous avez choisi bleu !\r\n")

def PrintRed():
    print("Vous avez choisi rouge !\r\n")

def PrintOrange():
    print("Vous avez choisi orange !\r\n")

def PrintYellow():
    print("Vous avez choisi jaune !\r\n")
```

Avant que le code puisse faire quoi que ce soit, vous devez définir les tâches à accomplir. Chacune de ces fonctions définit donc une tâche associée au choix d'une couleur. Un seul de ces éléments peut être appelé à un moment donné.

3. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
ColorSelect = {  
    0: PrintBlue,  
    1: PrintRed,  
    2: PrintOrange,  
    3: PrintYellow  
}
```

Ce code définit le dictionnaire. Chaque clé correspond à une partie d'une instruction de type `switch`. Les valeurs spécifient ce qu'il faut faire en fonction de la valeur de la clé. Les fonctions que vous avez créées plus haut sont les actions associées à la clause `switch` virtuelle, celle qui se trouvent normalement entre l'instruction `case` et la clause `break`.

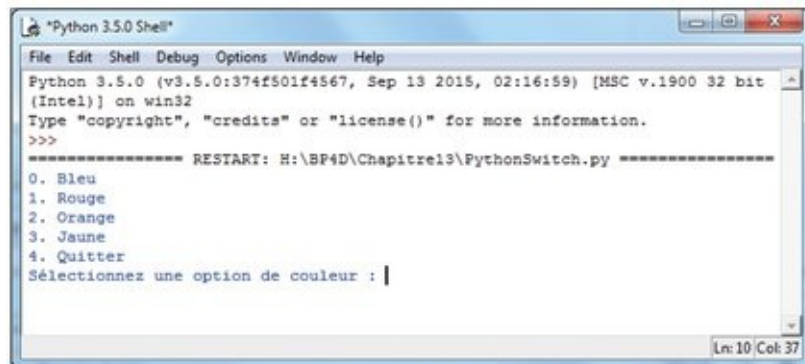
4. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
Selection = 0  
  
while (Selection != 4):  
    print("0. Bleu")  
    print("1. Rouge")  
    print("2. Orange")  
    print("3. Jaune")  
    print("4. Quitter")  
  
    Selection = int(input("Sélectionnez une option de couleur : "))  
  
    if (Selection >= 0) and (Selection < 4):  
        ColorSelect[Selection]()
```

Vous voyez ici la partie de l'exemple qui affiche l'interface utilisateur. Le code commence par créer une variable de saisie, `Selection`. Il rentre ensuite dans une boucle jusqu'à ce que l'utilisateur saisisse la valeur 4, qui termine l'opération. Lors de chaque boucle, l'application affiche une liste d'options, puis attend que l'utilisateur tape un chiffre. Quand c'est fait, elle vérifie la saisie. Une valeur entre 0 et 3

affiche une des fonctions définies auparavant dans le dictionnaire.

Figure 13.10 :
L'application commence par afficher un menu.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\PythonSwitch.py =====
0. Bleu
1. Rouge
2. Orange
3. Jaune
4. Quitter
Sélectionnez une option de couleur : |
```

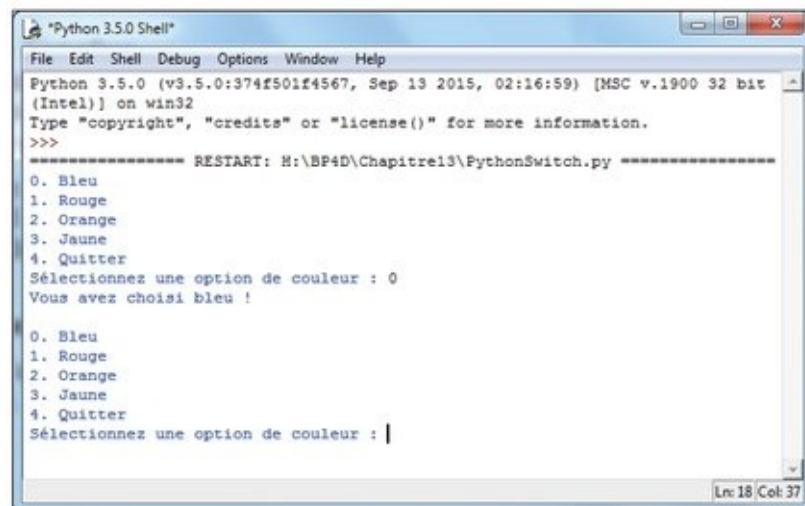
5. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche un menu semblable à celui de la [Figure 13.10](#).

6. **Tapez 0 et appuyez sur Entrée.**

L'application vous dit que vous avez sélectionné la couleur bleue, puis elle affiche de nouveau le menu (voir la [Figure 13.11](#)).

Figure 13.11 :
L'application affiche la réponse, puis présente à nouveau le menu.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\PythonSwitch.py =====
0. Bleu
1. Rouge
2. Orange
3. Jaune
4. Quitter
Sélectionnez une option de couleur : 0
Vous avez choisi bleu !
0. Bleu
1. Rouge
2. Orange
3. Jaune
4. Quitter
Sélectionnez une option de couleur : |
```

7. **Tapez 4 et appuyez sur Entrée.**

L'application se termine.

Créer des piles en utilisant des listes

Une pile est une structure de programmation utile, car vous pouvez l'utiliser pour sauvegarder l'environnement d'exécution d'une application (comme l'état des variables et d'autres attributs), ou encore pour déterminer l'ordre d'une exécution. Malheureusement, Python n'offre pas ce type de structure dans ses collections. Cependant, il dispose des listes, et vous pouvez utiliser celles-ci pour créer des piles parfaitement acceptables. Les étapes qui suivent vous montrent comment réaliser ce travail. Cet exemple est également disponible dans le fichier téléchargeable `ListStack.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```

MyStack = []
StackSize = 3

def DisplayStack():
    print("État courant de la pile :")
    for Item in MyStack:
        print(Item)

def Push(Value):
    if len(MyStack) < StackSize:
        MyStack.append(Value)
    else:
        print("La pile est pleine !")

def Pop():
    if len(MyStack) > 0:
        MyStack.pop()
    else:
        print("La pile est vide.")

Push(1)
Push(2)
Push(3)
DisplayStack()
input("Appuyez sur une touche quand vous êtes prêt...")

Push(4)
DisplayStack()
input("Appuyez sur une touche quand vous êtes prêt...")

Pop()
DisplayStack()
input("Appuyez sur une touche quand vous êtes prêt...")

Pop()
Pop()
Pop()
DisplayStack()

```

L'application crée une liste, ainsi qu'une variable servant à déterminer la taille maximale de la pile (une pile possède normalement une taille pourvue d'une certaine limite). Celle-ci est réellement petite, mais suffisante pour cet exemple.



Les piles fonctionnent en poussant une valeur sur leur sommet, puis en dépilant les valeurs ainsi stockées. Les fonctions `Push ()` et `Pop ()` effectuent ces deux tâches. Le code ajoute une fonction `DisplayStack ()` pour visualiser le contenu de la pile.

Le code restant permet de vérifier les fonctionnalités de la pile en y poussant des valeurs, puis en les dépilant. Quatre sections du code servent à effectuer ce test.

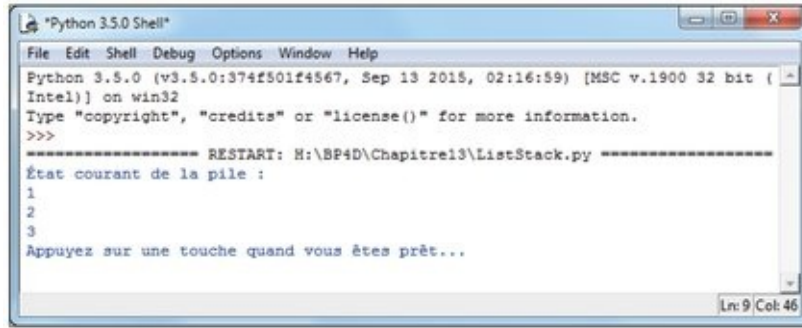
3. Choisissez la commande **Run Module** dans le menu

Run.

Une fenêtre Python en mode Shell va s'ouvrir. L'application remplit la pile, puis elle affiche son état (voir la [Figure 13.12](#)). Dans ce cas, 3 se trouve au sommet de la pile, puisque c'est la dernière valeur entrée.

Figure 13.12 :

Dans une pile, les valeurs sont placées les unes au-dessus des autres.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\ListStack.py =====
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
```

4. Appuyez sur Entrée.

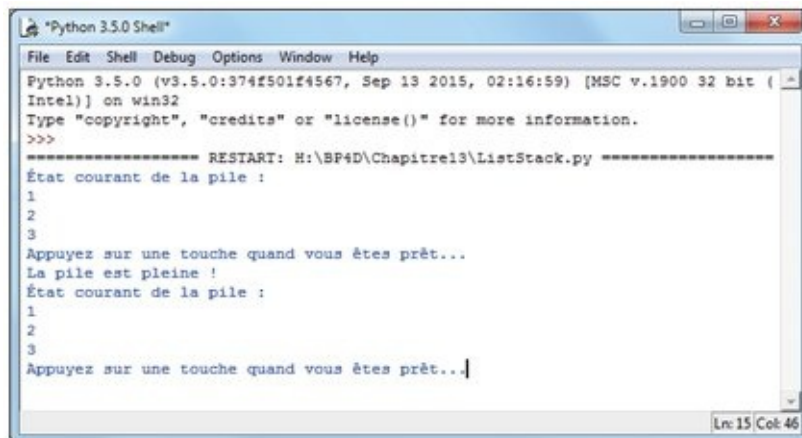
L'application tente de placer une nouvelle valeur sur le sommet de la pile. Mais comme celle-ci est déjà pleine, l'opération échoue (voir la [Figure 13.13](#)).

5. Appuyez sur Entrée.

Cette fois, l'application retire une valeur au sommet de la pile. Le nombre 3 disparaît donc, comme l'illustre la [Figure 13.14](#).

Figure 13.13 :

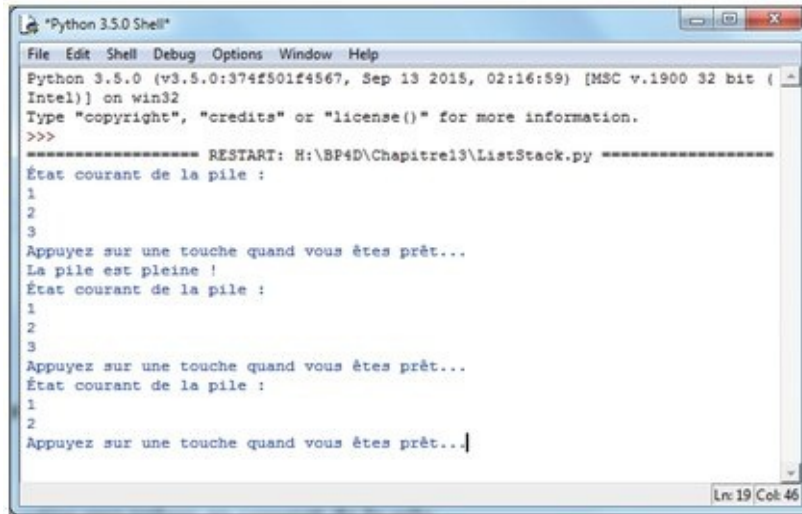
Lorsque la pile est pleine, elle ne peut plus accepter une nouvelle valeur.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\ListStack.py =====
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
La pile est pleine !
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
```

Figure 13.14 :

Supprimer une valeur au sommet de la pile.



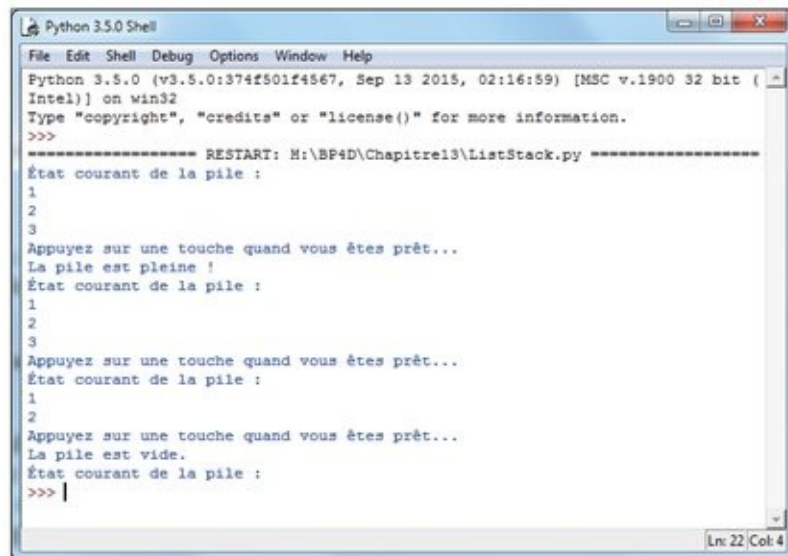
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\ListStack.py =====
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
La pile est pleine !
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
État courant de la pile :
1
2
Appuyez sur une touche quand vous êtes prêt...
Ln: 19 Col: 46
```

6. Appuyez sur Entrée.

L'application essaie de dépiler plus de valeurs que n'en contient la pile, ce qui provoque une erreur (voir la [Figure 13.15](#)). Toute implémentation de pile doit être capable de détecter aussi bien les débordements (trop d'entrées) que l'inverse (trop peu d'entrées).

Figure 13.15 :

Assurez-vous que votre pile détecte les débordements comme l'insuffisance de valeurs.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\ListStack.py =====
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
La pile est pleine !
État courant de la pile :
1
2
3
Appuyez sur une touche quand vous êtes prêt...
État courant de la pile :
1
2
Appuyez sur une touche quand vous êtes prêt...
La pile est vide.
État courant de la pile :
>>> |
Ln: 22 Col: 4
```

7. Vous pouvez quitter la fenêtre de Python.

Travailler avec les files

Une file fonctionne différemment d'une pile. C'est exactement le principe de la file d'attente à un guichet. Vous arrivez et vous vous placez à la fin de celle-ci. Au fur et à mesure que les autres personnes sont servies, vous avancez dans la file, jusqu'au moment où c'est à votre tour. Cette structure est souvent utilisée pour gérer les listes de tâches, ou encore pour gérer le flux d'un programme (comme dans le monde réel, en fait). Les étapes qui suivent vous montrent comment créer une file. Vous retrouverez également cet exemple dans le fichier téléchargeable `QueueData.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
import queue

MyQueue = queue.Queue(3)

print(MyQueue.empty())
input("Appuyez sur une touche quand vous êtes prêt...")

MyQueue.put(1)
MyQueue.put(2)
print(MyQueue.full())
input("Appuyez sur une touche quand vous êtes prêt...")

MyQueue.put(3)
print(MyQueue.full())
input("Appuyez sur une touche quand vous êtes prêt...")

print(MyQueue.get())
print(MyQueue.empty())
print(MyQueue.full())
input("Appuyez sur une touche quand vous êtes prêt...")

print(MyQueue.get())
print(MyQueue.get())
```

Pour créer une file, vous devez importer le module `queue`. Celui-ci contient en fait divers types de files, mais cet exemple utilise uniquement le classique FIFO (First In, First Out, premier entré, premier sorti).



Lorsqu'une file est vide, la fonction `empty ()` renvoie `True`. De même, lorsque la file est pleine, la fonction `full ()` renvoie également `True`. En testant l'état vide ou plein, vous pouvez déterminer si vous avez besoin d'effectuer d'autres travaux avec la file, ou si vous pouvez lui ajouter d'autres informations. Ces deux fonctions vous aident donc à gérer vos files. Il n'est pas possible de parcourir une file à l'aide d'une boucle `for` comme dans d'autres types de collections. C'est pourquoi il vous faut faire appel à `empty ()` et à `full ()` pour connaître son état.

Les deux fonctions utilisées pour travailler avec les données d'une file sont `put ()`, qui ajoute une nouvelle information, et `get()`, qui supprime une donnée. L'une des difficultés, avec les files, est que si vous tentez d'y placer plus de données qu'elles ne peuvent en contenir, elles attendent simplement que de la place se libère pour les y stocker. À moins d'utiliser une *application multithread* (une qui se sert de plusieurs « fils » pour exécuter plus d'une action à la fois), cette situation pourrait se terminer par un gel de votre application.

3. **Choisissez la commande Run Module dans le menu Run.**

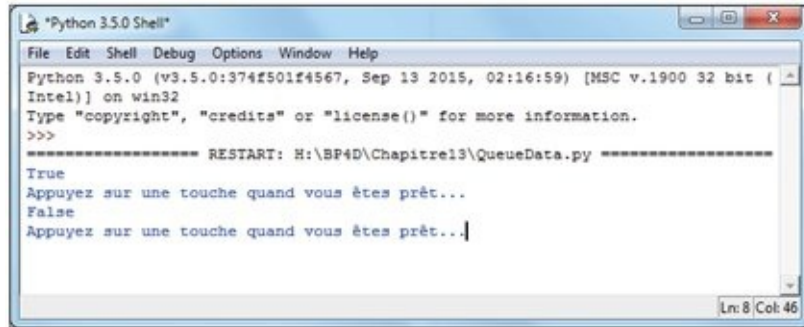
Une fenêtre Python en mode Shell va s'ouvrir. L'application teste l'état de la file. Dans ce cas, la sortie produite devrait être `True`, ce qui indique que la file est vide.

4. **Appuyez sur Entrée.**

L'application ajoute deux nouvelles valeurs à la file. Celle-ci n'est donc plus vide, comme l'illustre la [Figure 13.16](#).

Figure 13.16 :

Lorsque l'application ajoute de nouvelles valeurs à la file, celle-ci indique qu'elle n'est plus vide.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\QueueData.py =====
True
Appuyez sur une touche quand vous êtes prêt...
False
Appuyez sur une touche quand vous êtes prêt...
Ln: 8 Col: 46
```

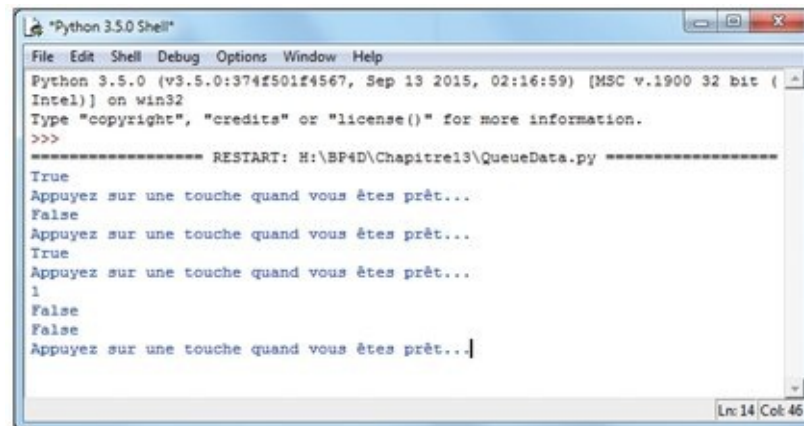
5. Appuyez sur Entrée.

L'application ajoute encore une entrée à la file. Celle-ci est donc maintenant pleine, puisqu'elle est conçue pour contenir trois éléments. La fonction `full ()` retourne donc la valeur `True`.

6. Appuyez sur Entrée.

Pour libérer de la place dans la file, la fonction `get ()` retire l'une des entrées, puis renvoie celle-ci. Du fait que la première valeur affectée à la file était 1, la fonction `print ()` affiche cette valeur. De plus, les deux fonctions `empty()` et `full ()` donnent maintenant comme réponse `False` (voir la [Figure 13.17](#)).

Figure 13.17 : Le suivi de leur état est une tâche essentielle avec les files.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\QueueData.py =====
True
Appuyez sur une touche quand vous êtes prêt...
False
Appuyez sur une touche quand vous êtes prêt...
True
Appuyez sur une touche quand vous êtes prêt...
1
False
False
Appuyez sur une touche quand vous êtes prêt...
Ln: 14 Col: 46
```

7. Appuyez sur Entrée.

L'application supprime les deux entrées restantes. Le code affiche donc les valeurs 2 et 3.

8. Vous pouvez quitter la fenêtre de Python.

Travailler avec des deque

Un *deque* est simplement une file dans laquelle vous pouvez ajouter ou supprimer des éléments à chaque extrémité. Pour fixer les idées, vous pouvez imaginer un deque comme une sorte de ligne horizontale. Certaines fonctions individuelles agissent sur le début et sur la fin de cette ligne pour voir l'agrandir ou au contraire la raccourcir.

Les étapes qui suivent vous aident à créer un exemple qui montre l'usage d'un deque. Vous pouvez également le retrouver dans le fichier téléchargeable

`DequeData.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**


```

import collections

MyDeque = collections.deque("abcdef", 10)

print("État initial :")
for Item in MyDeque:
    print(Item, end=" ")

print("\r\n\r\nAjout et extension à droite")
MyDeque.append("h")
MyDeque.extend("ij")
for Item in MyDeque:
    print(Item, end=" ")
print("\r\nMyDeque contient {0} éléments."
      .format(len(MyDeque)))

print("\r\n\r\nPop à droite")
print("Pop de {0}".format(MyDeque.pop()))
for Item in MyDeque:
    print(Item, end=" ")

print("\r\n\r\nAjout et extension à gauche")
MyDeque.appendleft("a")
MyDeque.extendleft("bc")
for Item in MyDeque:
    print(Item, end=" ")
print("\r\nMyDeque contient {0} éléments."
      .format(len(MyDeque)))

print("\r\n\r\nPop à gauche")
print("Pop de {0}".format(MyDeque.popleft()))
for Item in MyDeque:
    print(Item, end=" ")

print("\r\n\r\nSuppression")
MyDeque.remove("a")
for Item in MyDeque:
    print(Item, end=" ")

```

L'implémentation des deque se trouve dans le module collections, qui est donc importé au début du code. Lorsque vous créez un deque, vous pouvez spécifier comme ici (mais facultativement) une liste d'éléments itérables, donc qui peuvent être parcourus, ainsi qu'une taille maximale.



Un deque fait la différence entre ajouter un élément, et ajouter un groupe d'éléments. Dans le premier cas, vous utilisez les fonctions `append ()` et `appendleft ()`. Dans le second cas, vous faites appel aux fonctions `extend()` et `extendleft ()`. Pour supprimer un élément à la fois, vous disposez des fonctions `pop ()` et `popleft ()`. Ces fonctions renvoient l'élément supprimé, ce qui permet de l'afficher. La fonction `remove ()` est particulière, car elle part toujours de l'extrémité gauche et supprime la première instance de la donnée spécifiée.

Contrairement à certaines autres collections, un objet deque peut être parcouru à l'aide d'une boucle for chaque fois que c'est nécessaire.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche les résultats illustrés sur la [Figure 13.18](#).



Il est important de suivre de près la sortie. Notez comment la taille du deque change à chaque fois. Une fois que l'application a supprimé le *j*, le deque contient encore huit éléments. Lorsque l'application étend le deque en lui ajoutant trois éléments, la longueur affichée n'est pourtant que de dix éléments. Dans ce cas, la longueur maximum a été dépassée, et le supplément disparaît simplement.

Figure 13.18 : Un deque fournit notamment des fonctions permettant de gérer les extrémités d'une file.

A screenshot of a Python 3.5.0 Shell window. The window title is "Python 3.5.0 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The text area shows the following output:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre13\DequeData.py =====
État initial :
a b c d e f

Ajout et extension à droite
a b c d e f h i j
MyDeque contient 9 éléments.

Pop à droite
Pop de j
a b c d e f h i

Ajout et extension à gauche
c b a a b c d e f h
MyDeque contient 10 éléments.

Pop à gauche
Pop de c
b a a b c d e f h

Suppression
b a b c d e f h
>>> |
```

The status bar at the bottom right indicates "Lnc 26 Col 4".

Chapitre 14

Créer et utiliser des classes

Dans ce chapitre :

- ▶ Définir les caractéristiques d'une classe.
- ▶ Spécifier les composants d'une classe.
- ▶ Créer votre propre classe.
- ▶ Travailler avec les classes dans une application.
- ▶ Travailler avec les sous-classes.

Vous avez déjà travaillé avec de nombreuses classes dans les chapitres précédents. Par exemple, nombre d'entre elles sont faciles à construire et à utiliser parce qu'elles reposent sur des classes Python. Même si ces classes ont peu été invoquées dans les chapitres précédents, elles ont bel et bien été utilisées, sans pour autant qu'il soit nécessaire de s'y attarder immédiatement.

Les classes rendent le code Python plus lisible, plus compréhensible et plus simple à utiliser. Vous vous servez de classes pour créer des conteneurs afin d'y placer votre code, de manière à ce que celui-ci soit réparti de façon claire et nette. Vos utilisateurs voient vos classes comme des boîtes noires : il y a des

données qui y entrent, et d'autres données qui en sortent.



Arrivé à un certain point de votre progression, vous aurez besoin de construire des classes pour éviter que vos applications ne ressemblent à ce genre de code spaghetti que l'on trouvait dans d'anciens programmes. Il était alors difficile de savoir où les choses commençaient et où elles se terminaient. Maintenir un tel code était pratiquement impossible. Et certaines organisations finissaient même par s'en débarrasser.

Il est donc utile de comprendre que les classes sont un moyen d'empaqueter votre code. Dans ce chapitre, vous découvrirez comment procéder de manière à rendre nos applications plus utiles, plus simples, et plus faciles à gérer.

Comprendre les classes

Une classe est essentiellement une méthode permettant d'empaqueter du code. L'idée est de pouvoir réutiliser ce code, de rendre les applications plus fiables et de réduire les problèmes de sécurité. Les classes bien conçues sont des boîtes noires qui acceptent certaines valeurs en entrée et en renvoient d'autres en sortie. En résumé, une classe ne devrait créer aucune surprise pour personne, tout en ayant un comportement prévu et prévisible. Comment les classes arrivent à faire ce travail n'a pas d'importance, et en masquer les détails fait d'ailleurs partie des bonnes pratiques de programmation.

Avant de progresser dans la théorie des classes, vous

avez besoin de connaître quelques termes dont vous aurez besoin par la suite. Certains d'entre eux sont spécifiques à Python, et d'autres non.



- **Classe** : Définit une sorte de moule pour créer un objet. Par exemple, un architecte réalise des plans sur papier, et il s'assure via des calculs techniques que le bâtiment une fois construit va correspondre au projet initial. Python fonctionne sur le même principe.



- **Variable de classe** : Fournit un espace de stockage utilisé par toutes les instances d'une classe. Une telle variable est définie à l'intérieur de la classe, mais en dehors des méthodes de celle-ci. Elles ne sont pas utilisées très souvent, car elles présentent un risque de sécurité potentiel, puisque chaque méthode de la classe a accès aux mêmes informations. De surcroît, les variables de classe sont visibles dans la classe elle-même, plutôt que dans chaque instance de celle-ci. Cela entraîne donc des risques de contamination.
- **Membre de donnée** : Définit une variable de classe ou d'instance servant à contenir des données associées à une classe ou à ses objets.
- **Surcharge de fonction** : Crée plus d'une version d'une fonction, ce qui se traduit par des comportements différents. La tâche principale d'une fonction peut rester la même, mais les entrées sont différentes et les sorties éventuellement aussi. Cette surcharge permet

d'obtenir une souplesse accrue de manière à ce qu'une fonction puisse agir de différentes manières dans une même application.

✓ **Héritage** : Utilise une classe parent pour créer des classes enfants qui possèdent les mêmes caractéristiques que leur ancêtre. Les classes enfants possèdent généralement des fonctionnalités supplémentaires, ou fournissent des comportements différents de ceux de leur parent.

✓ **Instance** : Définit un objet créé à partir des spécifications fournies par une classe. Python peut produire de multiples instances d'une classe pour effectuer les tâches nécessitées par une application. Chaque instance est unique.

✓ **Variable d'instance** : Fournit un emplacement de stockage utilisé par une méthode unique d'une instance d'une classe. La variable est définie à l'intérieur d'une méthode. Ces variables sont considérées comme étant plus sûres que les variables de classe, car une seule méthode de la classe peut y accéder. Les données sont passées entre les méthodes en utilisant des arguments, ce qui en autorise un meilleur contrôle et une meilleure gestion.



✓ **Instanciation** : Action consistant à créer une instance d'une classe. L'objet résultant est une instance unique de la classe.

✓ **Méthode** : Définit le terme utilisé pour les fonctions qui font partie de la classe. Même si les méthodes et les fonctions définissent pour l'essentiel le même élément, les méthodes sont considérées comme étant plus spécifiques, car

seules les classes peuvent en posséder.

➤ **Objet** : Définit une instance unique d'une classe. L'objet contient toutes les méthodes et les propriétés de la classe originale. Cependant, les données diffèrent pour chaque objet. Les emplacements utilisés sont spécifiques à chaque objet, même si les données sont identiques.

➤ **Surcharge des opérateurs** : Crée plusieurs versions d'une fonction associée à un opérateur tel que +, - / ou *, ce qui produit des comportements différents. La tâche essentielle de l'opérateur peut être la même, mais la manière dont il interagit avec les données diffère. La surcharge des opérateurs est utilisée pour offrir une souplesse supplémentaire afin qu'un opérateur soit à même de travailler de diverses manières avec les applications.

Les classes et leurs composants

Une classe a une construction spécifique. Chaque partie d'une classe effectue une tâche particulière qui lui donne ses caractéristiques. Bien entendu, la classe commence avec un conteneur qui va l'englober en entier. C'est ce que nous allons préciser dans la prochaine section. Les sections qui suivent décrivent les autres parties d'une classe afin de mieux comprendre comment elles contribuent à celle-ci dans sa globalité.

Créer la définition d'une classe

Une classe n'a pas besoin d'être particulièrement

complexe. En fait, vous pouvez créer simplement un conteneur, y placer un élément, et appeler classe le résultat. Bien sûr, celle-ci ne fera pas grand-chose, mais vous pourrez tout de même l'*instancier* (c'est-à-dire demander à Python de fabriquer un objet à partir de ce moule). Vous pourrez alors travailler avec cette instance comme si l'objet avait toujours été là. Les étapes qui suivent vous aident à comprendre les notions de base sous-jacentes en créant une classe aussi simple que possible.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**

```
class MyClass:
    MyVar = 0
```

La première ligne définit le conteneur de la classe. Celui-ci est constitué du mot `class` suivi du nom de cette classe. Toutes les classes doivent débiter précisément de cette manière.

La seconde ligne est la suite de la classe. Ici, vous voyez une variable de la classe appelée `MyVar` dont la valeur est égale à 0. Chaque instance de cette classe possédera le même nom de variable, avec la même valeur initiale.

3. **Tapez `MyInstance = MyClass()` et appuyez sur Entrée.**

Vous venez de créer une instance de la classe `MyClass` appelée `MyInstance`. Bien entendu, vous voulez vérifier tout cela. C'est ce que fait l'Étape 4.

4. **Tapez `MyInstance.MyVar` et appuyez sur Entrée.**

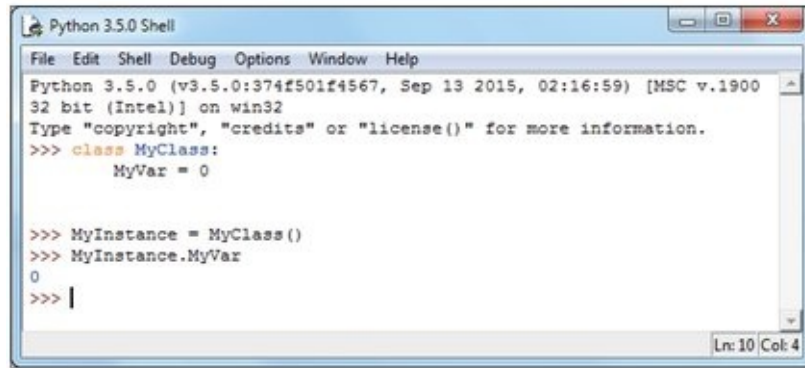
La sortie est, comme on pouvait s'y attendre, égale à 0 (voir la [Figure 14.1](#)). Ce qui prouve que `MyInstance` possède bien une variable de classe appelée `MyVar`.

5. **Tapez `MyInstance.__class__` et appuyez sur Entrée.**

Python affiche la classe utilisée pour créer l'instance, comme l'illustre la [Figure 14.2](#). Cette sortie vous informe que cette classe fait partie du module `__main__`, ce qui signifie que vous l'avez saisie directement dans le Shell.

Figure 14.1 :

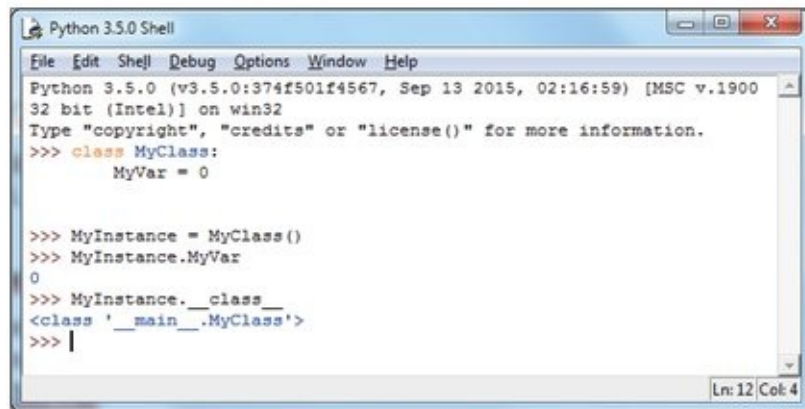
L'instance contient bien la variable de la classe.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    MyVar = 0

>>> MyInstance = MyClass()
>>> MyInstance.MyVar
0
>>> |
```

Figure 14.2 : Vous pouvez vérifier que l'instance a bien été créée à partir de la classe `MyClass`.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    MyVar = 0

>>> MyInstance = MyClass()
>>> MyInstance.MyVar
0
>>> MyInstance.__class__
<class '__main__.MyClass'>
>>> |
```

6. **Gardez cette fenêtre ouverte pour la prochaine section.**

Les classes et leurs attributs intégrés

Lorsque vous créez une classe, vous pourriez penser que tout ce que vous obtenez, c'est cette classe et rien d'autre. Cependant, Python y ajoute des fonctionnalités intégrées. Par exemple, dans la section précédente, vous avez tapé `__class__` et appuyé sur

Entrée. L'attribut `__class__` est directement intégré. Vous n'avez pas à le créer. Du fait que Python l'ajoute automatiquement, vous n'avez pas à vous en soucier. Une telle fonctionnalité est suffisamment utile pour que toute classe la possède, d'où le travail de Python en arrière-plan. Les étapes qui suivent vont vous montrer comment travailler avec les attributs intégrés des classes.

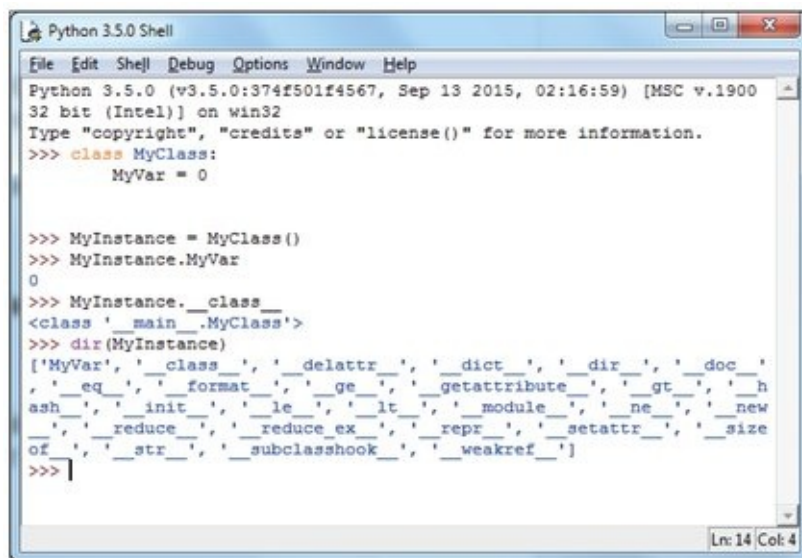
1. **Revenez à la fenêtre de Python ouverte dans la section précédente.**

Si vous avez passé ces étapes, il est encore temps de les reprendre.

2. **Tapez `dir(MyInstance)` et appuyez sur Entrée.**

Une liste d'attributs va s'afficher (voir la [Figure 14.3](#)). Ces attributs fournissent des fonctionnalités spécifiques pour votre classe. Ces mêmes attributs vont se retrouver dans toutes les classes que vous aurez à créer, et donc vous pourrez toujours compter sur ces fonctionnalités dans vos applications.

Figure 14.3 : La fonction `dir()` vous indique quels sont les attributs intégrés présents.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    MyVar = 0

>>> MyInstance = MyClass()
>>> MyInstance.MyVar
0
>>> MyInstance.__class__
<class '__main__.MyClass'>
>>> dir(MyInstance)
['MyVar', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__h
ash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new
__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__size
of__', '__str__', '__subclasshook__', '__weakref__']
>>>
```

3. **Tapez `help('__class__')` et appuyez sur Entrée.**

Python affiche des informations sur l'attribut `__class__` (voir la [Figure 14.4](#)). Vous pouvez utiliser la même technique pour

en apprendre plus sur tous les attributs que Python ajoute à votre classe.

Figure 14.4 :

Python fournit une aide pour chacun des attributs qu'il ajoute à votre classe.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
>>> help('class')
Help on class module in module builtins:

class_ = class module(object)
| module(name[, doc])
|
| Create a module object.
| The name must be a string; the optional doc argument can have any
| type.
|
| Methods defined here:
|
| _delattr__(self, name, /)
|     Implement delattr(self, name).
|
| _dir__(...)
|     __dir__() -> list
|     specialized dir() implementation
|
| _getattr__(self, name, /)
|     Return getattr(self, name).
|
| _init__(self, /, *args, **kwargs)
|     Initialize self. See help(type(self)) for accurate signature.
|
| _new__(*args, **kwargs) from builtin.type
|     Create and return a new object. See help(type) for accurate s
| ignature.
|
| _repr__(self, /)
|     Return repr(self).
|
| _setattr__(self, name, value, /)
|     Implement setattr(self, name, value).
|
|-----
| Data descriptors defined here:
|
| _dict_
|
|>>>
```

4. **Vous pouvez maintenant refermer la fenêtre de Python.**

Travailler avec les méthodes

Les méthodes sont simplement un autre terme pour les fonctions qui résident dans les classes. Vous créez des méthodes, et vous travaillez avec elles, exactement comme vous le faites avec des fonctions, si ce n'est que le mot méthode est employé exclusivement dans le cas des classes. Vous pouvez créer deux types de méthodes : celles qui sont associées à la classe elle-même, et celles qui sont associées avec une instance de cette classe. Il est important de faire la différence entre les deux. Les sections qui suivent précisent ces notions.

Créer des méthodes de classe

Une *méthode de classe* peut être exécutée directement dans la classe sans qu'une instance de celle-ci ne soit créée. C'est souvent nécessaire pour disposer de fonctionnalités applicables à toutes les instances, comme celles que vous avez déjà utilisées avec la classe `str`. Par exemple, dans le Chapitre 9, le code de `MultipleException4.py` fait appel à la fonction `str.upper()`. Les étapes qui suivent vous montrent comment créer et utiliser une méthode de classe.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**

```
class MyClass:
    def SayHello():
        print("Bonjour le monde !")
```

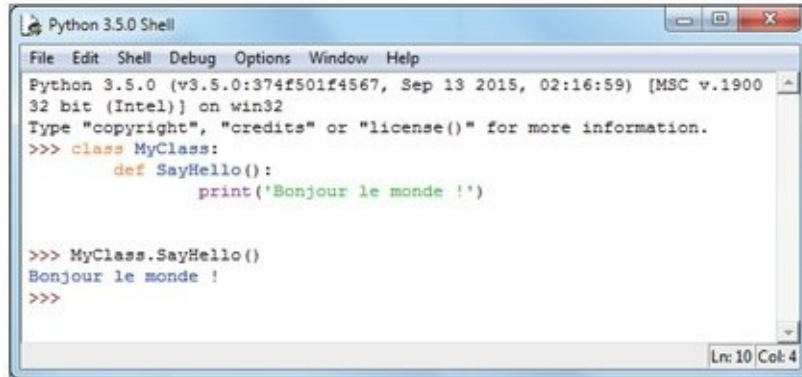
Cet exemple contient un unique attribut, `SayHello()`. Cette méthode ne reçoit aucun argument et ne renvoie aucune valeur. Elle se contente d'afficher un message. Pour autant, elle suffit à notre propos.

3. **Tapez `MyClass.SayHello()` et appuyez sur Entrée.**

Vous obtenez bien l'affichage prévu (voir la [Figure 14.5](#)). Remarquez que vous n'avez pas eu besoin de créer une instance de la classe. La méthode est immédiatement disponible.

Figure 14.5 :

Cette méthode de classe affiche un simple message.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    def SayHello():
        print('Bonjour le monde !')

>>> MyClass.SayHello()
Bonjour le monde !
>>>
```

4. Refermez la fenêtre de Python.



Une méthode de classe fonctionne uniquement avec les données de celle-ci. Elle ne sait rien des informations qui peuvent être associées avec une instance de la classe. Vous pouvez lui passer des données en argument, et la méthode retournera les valeurs qu'elle aura traitées, mais elle ne peut pas accéder à ce qui relève des instances. Vous devez donc être particulièrement vigilant lorsque vous créez de telles méthodes, puisqu'elles doivent être autosuffisantes.

Créer des méthodes d'instance

Une méthode d'instance est attachée aux instances individuelles. Vous les utilisez pour manipuler les données que la classe doit gérer. Une méthode d'instance ne peut évidemment être utilisée qu'une fois une instance créée.



Toutes les méthodes d'instance acceptent au moins un argument appelé `self`. Cet argument pointe sur l'instance particulière dont l'application se sert pour manipuler des données. Sans cet argument `self`, la

méthode ne saurait pas quelles données traiter. Cependant, `self` n'est pas considéré comme un argument accessible. Sa valeur est fournie par Python lui-même et vous ne pouvez pas la changer.

Les étapes qui suivent vous montrent comment créer et utiliser des méthodes d'instance dans Python.

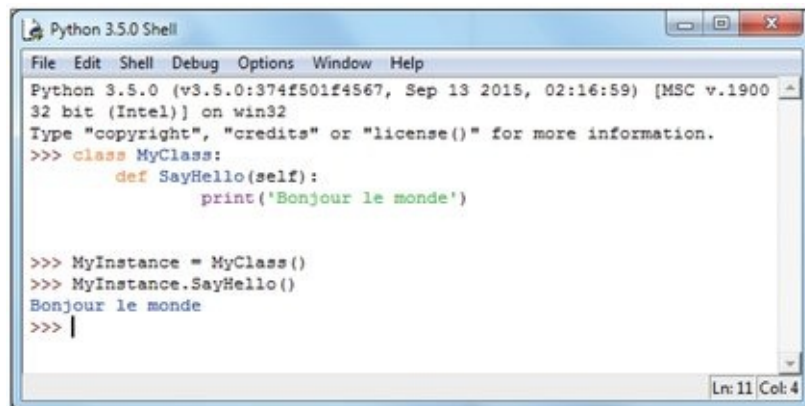
1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**

```
class MyClass:  
    def SayHello(self):  
        print("Bonjour le monde !")
```

Cet exemple contient un unique attribut, `SayHello()`. Cette méthode ne reçoit aucun argument et ne renvoie aucune valeur. Elle se contente d'afficher un message. Pour autant, elle suffit à notre propos.

3. **Tapez `MyInstance = MyClass()` et appuyez sur Entrée.**
Python crée une instance de `MyClass` appelée `MyInstance`.
4. **Tapez `MyInstance.SayHello()` et appuyez sur Entrée.**
Vous voyez le message illustré sur la [Figure 14.6](#).

Figure 14.6 : Le message est appelé depuis l'objet `MyInstance`.



5. Vous pouvez refermer la fenêtre de Python.

Travailler avec les constructeurs

Un *constructeur* est un type spécial de méthode que Python appelle lorsqu'il instancie un objet en utilisant les définitions trouvées dans votre classe. Python s'appuie sur ce constructeur pour effectuer différentes tâches, comme l'*initialisation* (l'affectation de valeurs) des variables d'instance dont l'objet a besoin lors de sa création. Les constructeurs peuvent aussi vérifier qu'il y a suffisamment de ressources pour l'objet, et effectuer toute autre procédure d'initialisation à laquelle vous pouvez penser.



Le nom du constructeur est toujours le même, `__init__`. Il peut si nécessaire accepter des arguments. Si vous fabriquez une classe sans constructeur, Python le fait automatiquement à votre place. Dans ce cas, ce constructeur par défaut ne fait rien. Toute classe doit posséder un constructeur, même s'il s'agit simplement du constructeur par défaut de Python.

Les étapes qui suivent illustrent cette notion de constructeur.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**


```
class MyClass:
    Greeting = ""
    def __init__(self, Name="le monde"):
        self.Greeting = Name + " !"
    def SayHello(self):
        print("Bonjour {}".format(self.Greeting))
```

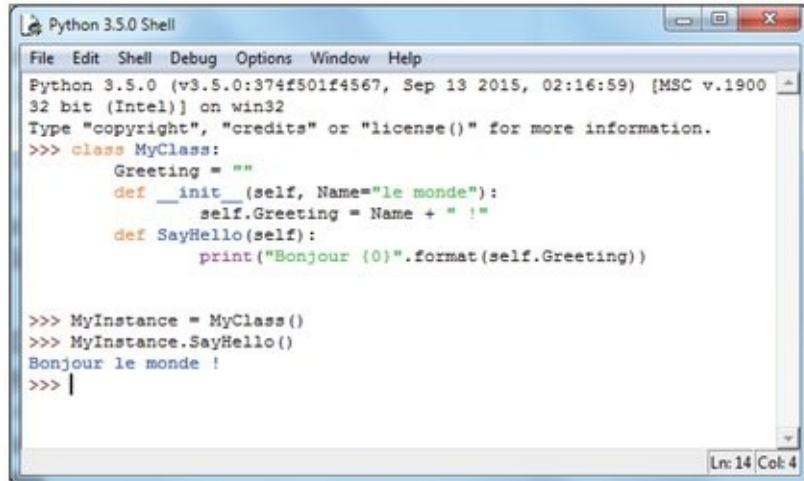
Vous trouvez ici un premier exemple de surcharge de fonction. Dans ce cas, il y a deux versions de `__init__`. La première ne demande aucune entrée particulière, puisqu'elle se contente de la valeur par défaut de la variable `Name`, « le monde ». La seconde nécessite un nom en entrée. Elle définit la valeur de `Greeting` avec cette entrée, et y ajoute un point d'exclamation. La méthode `SayHello()` est, pour le reste, la même que dans les exemples précédents.



Python ne supporte pas les vraies surcharges de fonctions. De nombreux aficionados de la Programmation Orientée Objet (POO) considèrent que les valeurs par défaut sont différentes de la surcharge. Cependant, le résultat est au final le même, et c'est de toute manière la seule technique offerte par Python. Dans une « vraie » surcharge, vous verriez de multiples copies de la même fonction, chacune pouvant traiter différemment ses entrées.

3. **Tapez `MyInstance = MyClass()` et appuyez sur Entrée.**
Python crée une instance de `MyClass` appelée `MyInstance`.
4. **Tapez `MyInstance.SayHello()` et appuyez sur Entrée.**
Vous voyez le message illustré sur la [Figure 14.7](#). Remarquez que ce message fournit la valeur générique du salut.
5. **Tapez `MyInstance = MyClass('Pierre')` et appuyez sur Entrée.**
Python crée une instance de `MyClass` appelée `MyInstance`.
6. **Tapez `MyInstance.SayHello()` et appuyez sur Entrée.**
Vous voyez cette fois le message illustré sur la [Figure 14.8](#). Le salut est devenu plus personnalisé.

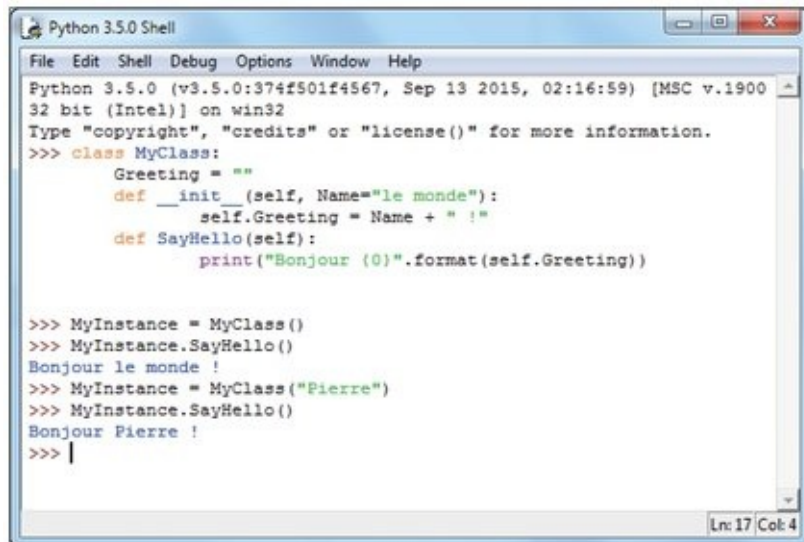
Figure 14.7 : La première version du constructeur fournit une valeur par défaut pour le nom.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    Greeting = ""
    def __init__(self, Name="le monde"):
        self.Greeting = Name + " !"
    def SayHello(self):
        print("Bonjour {}".format(self.Greeting))

>>> MyInstance = MyClass()
>>> MyInstance.SayHello()
Bonjour le monde !
>>> |
```

Figure 14.8 : Fournir un nom au constructeur donne une sortie personnalisée.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    Greeting = ""
    def __init__(self, Name="le monde"):
        self.Greeting = Name + " !"
    def SayHello(self):
        print("Bonjour {}".format(self.Greeting))

>>> MyInstance = MyClass()
>>> MyInstance.SayHello()
Bonjour le monde !
>>> MyInstance = MyClass("Pierre")
>>> MyInstance.SayHello()
Bonjour Pierre !
>>> |
```

7. Vous pouvez refermer la fenêtre de Python.

Travailler avec les variables

Vous savez que les *variables* sont des conteneurs chargés de stocker des données. Lorsque vous travaillez avec des classes, vous devez prendre en compte la manière dont vos informations sont enregistrées et gérées. Une classe peut contenir à la fois des variables de classe et des variables d'instance. Les premières sont définies comme

appartenant à la classe elle-même, tandis que les secondes font partie des méthodes. Les sections qui suivent vous montrent comment utiliser ces deux types de variables.

Créer des variables de classe

Les variables de classe fournissent un accès global aux données manipulées par la classe. Dans la plupart des cas, vous les initialisez en utilisant le constructeur de manière à vous assurer qu'elles contiennent une valeur correcte connue. Suivez ces étapes :

1. **Ouvrez une fenêtre de fichier Python.**
Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**

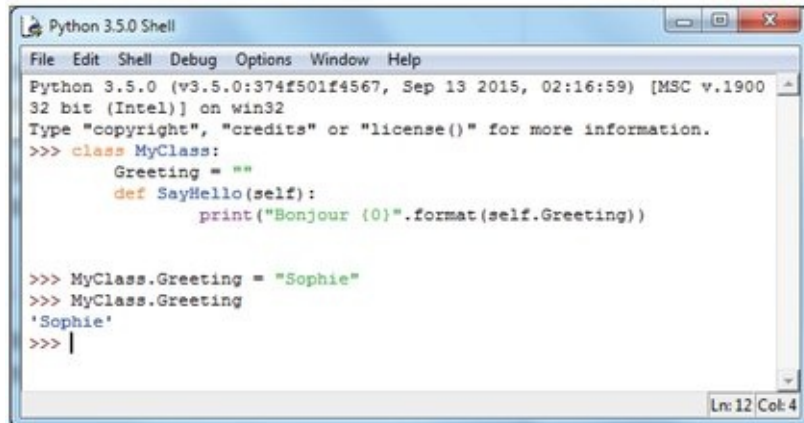
```
class MyClass:  
    Greeting = ""  
    def SayHello(self):  
        print("Bonjour {}".format(self.Greeting))
```

C'est une variante de la section précédente, mais cette version ne contient pas de constructeur. Normalement, vous devriez définir un constructeur pour vous assurer que la variable de classe est correctement initialisée. Cependant, cette série d'étapes vous montre comment les variables de classe peuvent mal tourner.

3. **Tapez `MyClass.Greeting = 'Sophie'` et appuyez sur Entrée.**
Cette instruction affecte à la variable `Greeting` une autre valeur que celle utilisée pour créer la classe. La grande question est maintenant de savoir comment les choses vont se passer.
4. **Tapez `MyClass.Greeting` et appuyez sur Entrée.**
Vous constatez que la valeur de `Greeting` a bien été

modifiée (voir la [Figure 14.9](#)).

Figure 14.9 : Vous pouvez modifier la valeur de la variable de classe.

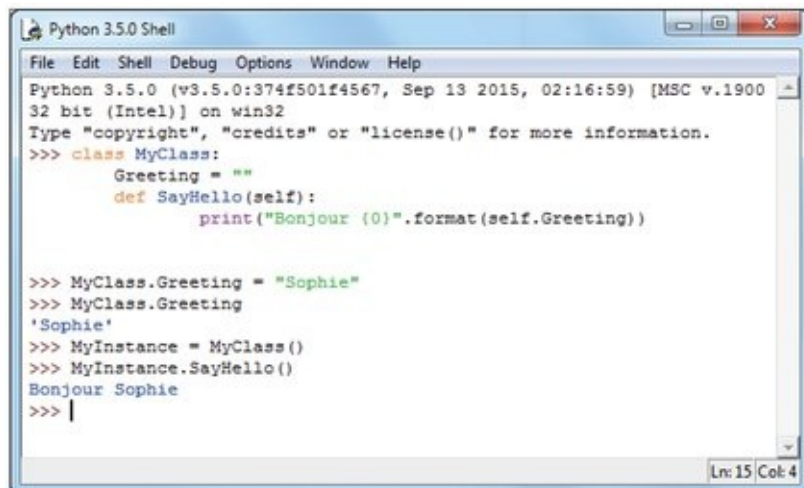


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    Greeting = ""
    def SayHello(self):
        print("Bonjour {}".format(self.Greeting))

>>> MyClass.Greeting = "Sophie"
>>> MyClass.Greeting
'Sophie'
>>> |
```

5. Tapez **MyInstance = MyClass()** et appuyez sur Entrée. Python crée une instance de `MyClass` appelée `MyInstance`.

Figure 14.10 : Les modifications apportées à `Greeting` se répercutent sur toutes les instances de la classe.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    Greeting = ""
    def SayHello(self):
        print("Bonjour {}".format(self.Greeting))

>>> MyClass.Greeting = "Sophie"
>>> MyClass.Greeting
'Sophie'
>>> MyInstance = MyClass()
>>> MyInstance.SayHello()
Bonjour Sophie
>>> |
```

6. Tapez **MyInstance.SayHello()** et appuyez sur Entrée. Vous voyez s'afficher le message illustré sur la [Figure 14.10](#). Le changement que vous avez apporté à la variable `Greeting` s'est répercuté sur l'instance de la classe. Bien sûr, ce n'est pas réellement un problème dans cet exemple, mais imaginez ce qui pourrait se produire dans une application réelle si quelqu'un voulait provoquer des problèmes.



Ce petit exemple illustre les dangers des variables de classe. Les deux leçons à en retenir sont les suivantes :

- Évitez chaque fois que possible les variables de classe pour empêcher une utilisation abusive.
- Initialisez toujours les variables de classe avec une valeur connue et sans danger dans le code du constructeur.

7. Vous pouvez refermer la fenêtre de Python.

Créer des variables d'instance

Les variables d'instance sont définies uniquement à l'intérieur d'une méthode. Les arguments d'entrée de ces méthodes sont considérés comme faisant partie des variables d'instance, car elles n'existent qu'en même temps que la méthode correspondante. Utiliser des variables d'instance est généralement plus sûr que des variables de classe, puisqu'il est plus facile de les contrôler et de s'assurer que l'appelant a fourni les bonnes entrées. Suivez donc ces étapes :

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (puis une seconde fois sur Entrée après la dernière ligne) :**

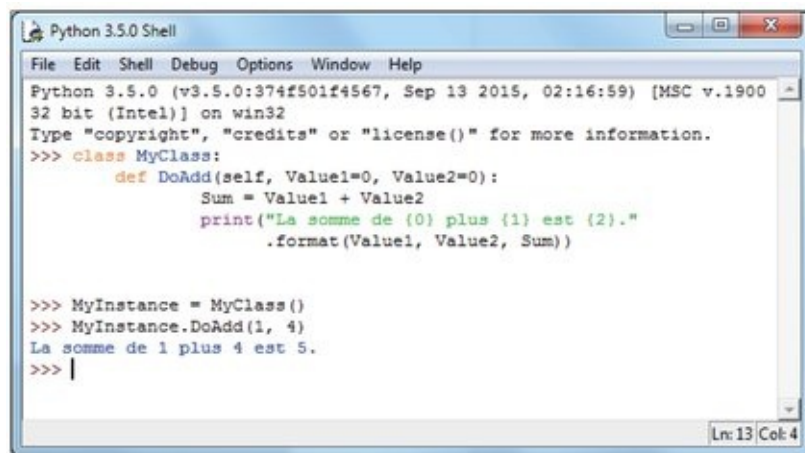
```
class MyClass:
    def DoAdd(self, Value1=0, Value2=0):
        Sum = Value1 + Value2
        print("La somme de {0} plus {1} est {2}."
              .format(Value1, Value2, Sum))
```

Vous avez ici trois variables d'instance. Les arguments en entrée, `Value1` et `Value2`, ont pour valeur par défaut 0. De cette manière, la méthode `DoAdd` ne peut pas échouer sous le prétexte que l'utilisateur aurait oublié de fournir une valeur. Bien entendu, celui-ci pourrait par mégarde saisir autre

chose qu'un nombre, et vous devriez donc effectuer les contrôles nécessaires dans votre code. La troisième variable d'instance, Sum, est la somme de `value1` et de `value2`. Le code se contente d'effectuer l'opération et d'afficher le résultat.

3. **Tapez `MyInstance = MyClass()` et appuyez sur Entrée.** Python crée une instance de `MyClass` appelée `MyInstance`.
4. **Tapez `MyInstance.DoAdd(1, 4)` et appuyez sur Entrée.** Vous voyez le message illustré sur la [Figure 14.11](#), soit la somme de 1 et de 4.

Figure 14.11 :
L'instance affiche simplement la somme de deux nombres.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> class MyClass:
    def DoAdd(self, Value1=0, Value2=0):
        Sum = Value1 + Value2
        print("La somme de {0} plus {1} est {2}."
              .format(Value1, Value2, Sum))

>>> MyInstance = MyClass()
>>> MyInstance.DoAdd(1, 4)
La somme de 1 plus 4 est 5.
>>> |
```

5. **Vous pouvez refermer la fenêtre de Python.**

Utiliser des méthodes avec des listes d'arguments variables

Il peut arriver que vous deviez créer des méthodes avec un nombre variable d'arguments. Gérer ce type de situation est quelque chose que Python sait bien faire. Vous pouvez créer deux sortes d'arguments en nombre variable :

- ✓ `*args` : Fournit une liste d'arguments non nommés.
- ✓ `*kwargs` : Fournit une liste d'arguments

nommés.



Les noms réels des arguments n'a pas d'importance, mais les développeurs Python utilisent la syntaxe `*args` et `**kwargs` pour d'autres programmeurs sachent qu'il s'agit d'une liste variable d'arguments. Remarquez que la première variante est précédée d'un unique astérisque, tandis qu'il y en a deux dans la seconde. Les étapes qui suivent montrent comment utiliser ces deux approches dans une application. Vous retrouverez également cet exemple dans le fichier téléchargeable `VariableArgs.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
class MyClass:
    def PrintList1(*args):
        for Count, Item in enumerate(args):
            print("{0}. {1}".format(Count, Item))

    def PrintList2(**kwargs):
        for Name, Value in kwargs.items():
            print("{0} aime le {1}".format(Name, Value))

MyClass.PrintList1("Rouge", "Bleu", "Vert")
MyClass.PrintList2(Georges="Rouge", Annie="Bleu",
                    Sarah="Vert")
```

Pour cette démonstration, vous voyez les arguments implémentés dans une méthode de classe. Il ne serait pas plus compliqué de réaliser la même chose dans une méthode d'instance.



Observez soigneusement `PrintList1 ()`. Vous y retrouvez la technique de parcours dans une liste à l'aide d'une boucle

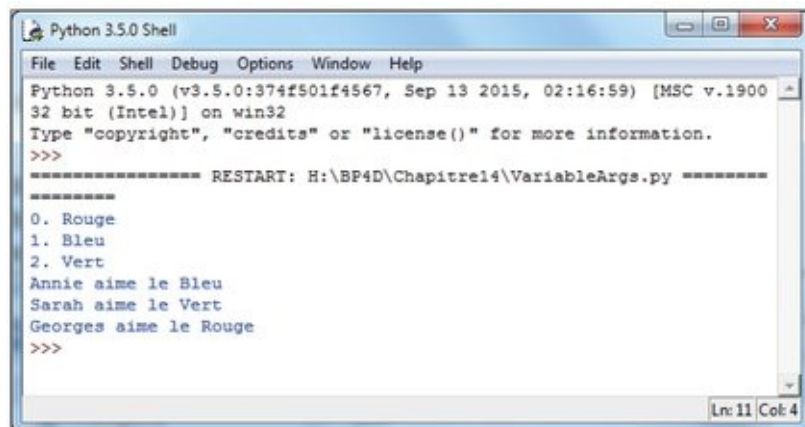
for. Dans ce cas, la fonction `enumerate ()` renvoie à la fois un compteur (la position dans la boucle) et la chaîne qui a été passée en argument à la fonction.

La fonction `PrintList2 ()` accepte en entrée un dictionnaire. Comme dans le cas de `PrintList1 ()`, cette liste peut être d'une longueur quelconque. Cependant, vous devez traiter les éléments `items ()` trouvés dans le dictionnaire pour récupérer les valeurs individuelles.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche la sortie illustrée sur la [Figure 14.12](#). Les listes individuelles peuvent être d'une longueur quelconque. Vous pourriez à ce stade jouer un peu avec cette technique. Essayez par exemple de mixer des noms et des chaînes dans la première liste pour voir ce qui se passe. Essayez aussi d'ajouter des valeurs booléennes. L'essentiel, c'est que cela vous permet de créer des méthodes extrêmement flexibles si tout ce que vous voulez est une liste de valeurs en entrée.

Figure 14.12 : Le code peut traiter n'importe quel nombre d'entrées dans la liste.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre14\VariableArgs.py =====
>>>
0. Rouge
1. Bleu
2. Vert
Annie aime le Bleu
Sarah aime le Vert
Georges aime le Rouge
>>>
```

Surcharger les opérateurs

Dans certaines situations, vous voudriez être capable de faire quelque chose de particulier en tant que

résultat de l'utilisation d'un opérateur standard tel que l'addition (+). En fait, Python ne fournit parfois pas de comportement par défaut, car il n'a justement pas de comportement par défaut à implémenter. Quelle qu'en soit la raison, la surcharge des opérateurs rend possible l'affectation de nouvelles fonctionnalités à des opérateurs existants, de manière à ce qu'ils fassent ce que vous voulez, et pas ce que Python a prévu. Les étapes qui suivent vous montrent comment cela est possible. Vous retrouverez également cet exemple dans le fichier téléchargeable `OverloadOperator.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
class MyClass:
    def __init__(self, *args):
        self.Input = args

    def __add__(self, Other):
        Output = MyClass()
        Output.Input = self.Input + Other.Input
        return Output

    def __str__(self):
        Output = ""
        for Item in self.Input:
            Output += Item
            Output += " "
        return Output

Value1 = MyClass("Rouge", "Vert", "Bleu")
Value2 = MyClass("Jaune", "Pourpre", "Cyan")
Value3 = Value1 + Value2

print("{0} + {1} = {2}"
      .format(Value1, Value2, Value3))
```

Cet exemple illustre plusieurs techniques différentes. Le constructeur `__init__()` montre une méthode pour créer une variable d'instance attachée à l'objet `self`. Vous pouvez utiliser cette approche pour créer autant de variables qu'il est nécessaire.



Lorsque vous créez vos propres classes, aucun opérateur `+` n'est défini dans la plupart des cas, du moins tant que vous ne l'avez pas fait. La seule exception se produit lorsque vous héritez d'une classe existante dans laquelle cet opérateur a été défini (pour plus de détails, voyez plus loin dans ce chapitre la section « Étendre des classes pour en créer de nouvelles »). Pour pouvoir ajouter des entrées `MyClass` ensemble, vous devez donc définir la méthode `__add__()`, qui équivaut à l'opérateur `+`.

Le code utilisé pour la méthode `__add__()` peut paraître un peu étrange, mais il faut le regarder ligne par ligne. Il commence par fabriquer un nouvel objet, `output`, à partir de `MyClass`. Rien n'est ajouté à `output` à ce stade.

Il s'agit donc d'un objet vierge. Les deux objets que vous voulez ajouter, `self.Input` et `other.Input`, sont en fait des tuples (revoyez à ce sujet la section « Travailler avec les tuples » dans le Chapitre 13). Le code place la somme de ces deux objets dans `output.Input`. La méthode `__add__()` retourne ensuite le résultat à l'appelant.

Bien entendu, vous voudriez savoir pourquoi vous ne pouvez tout simplement pas additionner les deux entrées comme vous le feriez pour des nombres. La réponse est que vous finissez en sortie par un tuple, et non par un objet `MyClass`. Le type de la sortie pourrait être changé, ce qui changerait aussi l'utilisation ultérieure de l'objet résultant.

Pour afficher correctement `MyClass`, vous avez aussi besoin de définir une méthode `__str__()`. Dans ce cas, la sortie est une *chaîne délimitée par un espace* (autrement dit, chacun des éléments de la chaîne est séparé d'un autre par un espace). Cette chaîne contient chacune des valeurs trouvées dans `self.Input`. Bien entendu, la classe que vous créez peut renvoyer n'importe quelle chaîne permettant de représenter pleinement l'objet.

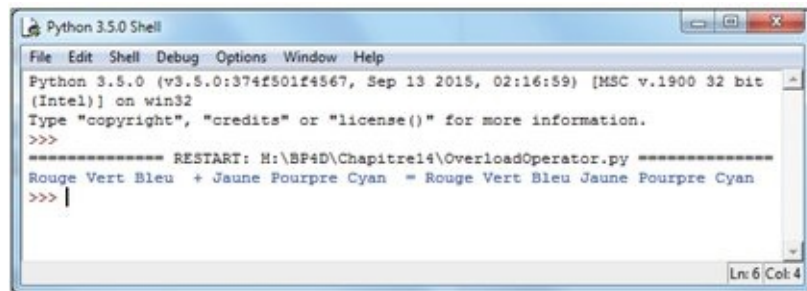
La procédure principale crée deux objets de test, `value1` et

Value2, puis elle les additionne et place le résultat dans Value3 afin d'afficher la réponse.

3. Choisissez la commande Run Module dans le menu Run.

Une fenêtre Python en mode Shell va s'ouvrir. La [Figure 14.13](#) illustre l'affichage obtenu. Certes, cela fait beaucoup de code pour une simple sortie, mais ce résultat démontre que vous pouvez créer des classes qui contiennent tout ce dont elles ont besoin, et qui sont totalement fonctionnelles.

Figure 14.13 : Le résultat de la somme de deux objets MyClass est un troisième objet de même type.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre14\OverloadOperator.py =====
Rouge Vert Bleu + Jaune Pourpre Cyan = Rouge Vert Bleu Jaune Pourpre Cyan
>>> |
```

Créer une classe

Tout le matériau présenté depuis le début de ce chapitre a pour but de vous aider à créer vos propres classes. C'est ce que vous allez faire ici afin d'obtenir une classe que vous pourrez placer dans un module externe, et réutiliser éventuellement dans une application.

Le [Listing 14.1](#) montre le code dont vous avez besoin pour créer la classe. Vous le retrouverez également dans le fichier téléchargeable `MyClass.py`.

Listing 14.1 :
Créer une classe
externe.

```
class MyClass:
    def __init__(self, Name="Samuel", Age=32):
        self.Name = Name
        self.Age = Age

    def GetName(self):
        return self.Name

    def SetName(self, Name):
        self.Name = Name

    def GetAge(self):
        return self.Age

    def SetAge(self, Age):
        self.Age = Age

    def __str__(self):
        return "{0} est âgé(e) de {1} ans.".format(self.Name,
                                                    self.Age)
```

Dans ce cas, la classe commence par créer un objet avec deux variables d'instance : `Name` et `Age`. Pour le cas où l'utilisateur ne fournirait pas ces informations, ces variables sont initialisées avec les valeurs `Samuel` et `32`.



Cet exemple fournit une nouvelle fonctionnalité des classes, ce que la plupart des développeurs appellent un *accesseur*. Pour l'essentiel, celui-ci fournit un accès à la valeur sous-jacente. On peut également trouver des variations autour de ce nom. Il y a d'abord les accesseurs proprement dits, ou encore *getters*. Ainsi, `GetName` et `GetAge` sont des accesseurs/getters. Ils offrent un accès en lecture seule à la valeur sous-jacente. Les méthodes `SetName` et `SetAge` sont des *mutateurs*, ou encore *setters*. Elles fournissent un accès en écriture seule à la valeur sous-jacente. Utiliser une combinaison de méthodes comme celle-ci vous permet de vérifier que les entrées possèdent le type voulu, ou encore que l'appelant a l'autorisation de voir l'information.

Comme pour toute autre classe que vous créez, vous avez besoin de définir la méthode `__str__()` si vous voulez que l'utilisateur puisse afficher l'objet. Ici, la classe fournit une sortie formatée qui liste les deux

variables d'instance.

Utiliser la classe dans une application

La plupart du temps, vous utilisez des classes externes lorsque vous travaillez avec Python. Il est assez rare de laisser les classes dans le fichier de l'application elle-même, car celle-ci pourrait rapidement devenir très volumineuse et pratiquement ingérable. De plus, réutiliser le même code dans une autre application serait difficile. Les étapes qui suivent vous aident à utiliser la classe `MyClass` créée dans la section précédente. Cet exemple se trouve également dans le fichier téléchargeable `MyClassTest.py`.

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
import MyClass

SamuelRecord = MyClass.MyClass()
AnnieRecord = MyClass.MyClass("Annie", 44)

print(SamuelRecord.GetAge())
SamsRecord.SetAge(33)

print(AnnieRecord.GetName())
AnnieRecord.SetName("Sophie")

print(SamuelRecord)
print(AnnieRecord)
```



Cet exemple de code commence par importer le module `MyClass`. Le nom de ce module est celui utilisé pour l'enregistrer sur le disque, pas le nom de la classe. Un même module peut parfaitement contenir de multiples

classes, et il faut donc éviter toute confusion à ce sujet.

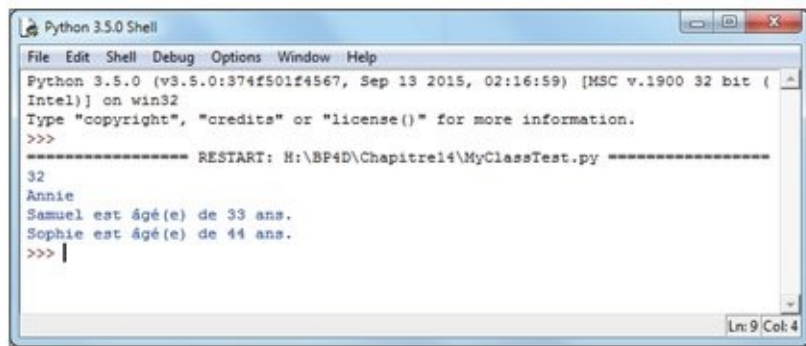
Une fois le module importé, l'application crée deux objets `MyClass`. Notez que vous spécifiez d'abord le nom du module, suivi du nom de la classe. Le premier objet, `SamuelRecord`, utilise les valeurs par défaut. Le second, `AnnieRecord`, définit des valeurs personnalisées.

L'anniversaire de Samuel étant vraisemblablement passé, l'application lui ajoute une année de plus. D'autre part, il y a eu une confusion de prénom, et la titulaire du poste est Sophie et non Annie. Le code corrige donc cette erreur. Enfin, la dernière étape affiche les informations actualisées.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. L'application affiche les messages illustrés sur la [Figure 14.14](#).

Figure 14.14 : La sortie montre que la classe est pleinement fonctionnelle.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre14\MyClassTest.py =====
32
Annie
Samuel est âgé(e) de 33 ans.
Sophie est âgé(e) de 44 ans.
>>> |
```

Étendre des classes pour en créer de nouvelles

Comme vous pouvez l'imaginer, créer une classe pleinement fonctionnelle, susceptible d'être utilisée dans des applications du monde réel, demande beaucoup de temps et de travail, car les classes sont conçues pour effectuer de nombreuses tâches. Fort heureusement, Python supporte une fonctionnalité appelée *héritage*. Cet héritage vous permet de

recupérer les fonctionnalités voulues d'une classe *parent* lorsque vous créez une classe *enfant*. Éliminer les fonctions dont vous n'avez pas besoin, et en ajouter d'autres vous permet de créer des classes relativement rapidement et avec moins d'efforts de votre part. De plus, du fait que le code parent a déjà été testé, vous avez moins de soucis à vous faire quant au bon fonctionnement de votre nouvelle classe. Les sections qui suivent vous montrent comment profiter de ce processus d'héritage.

Construire la classe enfant

Les classes parents sont normalement des super-ensembles de quelque chose. Vous pourriez par exemple créer une classe parent appelée `Voiture`, puis définir des classes enfants représentant divers modèles de véhicules. Dans notre exemple, nous allons plus modestement fabriquer une classe parent appelée `Animals`, puis nous en servir pour nous intéresser à une classe enfant dédiée à certains gallinacés, `Chicken`. Bien entendu, vous pourriez facilement ajouter d'autres classes enfants pour d'autres types d'animaux, comme une classe `Gorille`. Cependant, nous nous contenterons ici d'une classe parent, et d'une classe enfant. Vous les retrouvez dans le [Listing 14.2](#), ainsi que dans le fichier téléchargeable `Animals.py`.

Listing 14.2 :
Construire une
classe parent et
une classe enfant.

```
class Animal:
    def __init__(self, Name="", Age=0, Type=""):
        self.Name = Name
        self.Age = Age
        self.Type = Type

    def GetName(self):
        return self.Name

    def SetName(self, Name):
        self.Name = Name

    def GetAge(self):
        return self.Age

    def SetAge(self, Age):
        self.Age = Age

    def GetType(self):
        return self.Type

    def SetType(self, Type):
        self.Type = Type

    def __str__(self):
        return "{0} est un {1} âgé de {2} ans.".format(self.Name,
                                                    self.Type,
                                                    self.Age)

class Chicken(Animal):
    def __init__(self, Name="", Age=0):
        self.Name = Name
        self.Age = Age
        self.Type = "poulet"

    def SetType(self, Type):
        print("Désolé, {0} sera toujours un {1}."
              .format(self.Name, self.Type))

    def MakeSound(self):
        print("{0} dit Cot, Cot, Cot, Codette !".format(self.Name))
```

La classe `Animal` gère trois caractéristiques : le nom (`Name`), l'âge (`Age`) et le type (`Type`). Une véritable application aurait vraisemblablement besoin de davantage de paramètres, mais ceux-ci nous suffiront pour cet exemple. Le code contient également les accesseurs nécessaires pour ces trois caractéristiques. La méthode `__str__()` complète le tableau en affichant un message simple qui décrit ce qu'est l'animal.

La classe `Chicken` hérite de la définition de la classe `Animal`. Remarquez l'emploi de `Animal` entre parenthèses après le nom de la classe `Chicken`. Ceci signale à Python que `Chicken` est un enfant de `Animal`, et donc qu'il hérite des caractéristiques de celui-ci.

Notez aussi que le constructeur `Chicken` n'accepte comme entrées que `Name` et `Age`. L'utilisateur n'a donc pas besoin de fournir une valeur `Type`, puisque vous savez déjà que l'animal en question est un poulet. Le nouveau constructeur remplace celui de `Animal`. Les trois attributs sont toujours en place, mais `Type` est compris directement dans le constructeur `Chicken`.

Quelqu'un pourrait être tenté de faire preuve d'originalité en définissant le poulet comme étant un gorille. De ce point de vue, la classe `Chicken` prend le pas sur le mutateur `SetType()`. En tentant de changer le type `Chicken`, l'utilisateur ne fait que recevoir un message. Normalement, ce genre de problème devrait être traité par une exception, mais la solution proposée ici fonctionne mieux, car elle rend le code plus clair.

Finalement, le code de `Chicken` ajoute une fonctionnalité supplémentaire, `MakeSound()`. Si quelqu'un veut entendre le cri du poulet, il n'aura pas besoin d'écouteurs, mais il verra du moins le message correspondant s'afficher à l'écran.

Tester la classe dans une application

Tester la classe `Chicken` signifie tester également la classe `Animal`, puisque la première est une extension de la seconde. Certaines fonctionnalités sont différentes, mais tout n'a pas besoin d'être utilisé en même temps. La classe `Animal` est simplement un parent d'un type spécifique d'animaux, en l'occurrence des poulets. Les étapes qui suivent illustrent le fonctionnement de la classe `Chicken` de manière à montrer comment fonctionne l'héritage. Vous pouvez

également retrouver ce code dans le fichier téléchargeable `AnimalsTest.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
import Animals

MyChicken = Animals.Chicken("Arthur", 2)
print(MyChicken)
MyChicken.SetAge(MyChicken.GetAge() + 1)
print(MyChicken)
MyChicken.SetType("Gorille")
print(MyChicken)
MyChicken.MakeSound()
```

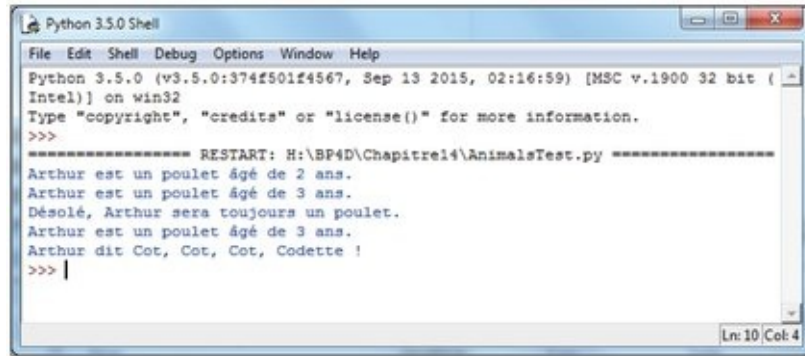
La première étape consiste à importer le module `Animals`. Souvenez-vous que vous importez toujours le nom du fichier, pas la ou les classes qu'il contient. Ici, le fichier `Animals.py` contient deux classes : `Animal` et `Chicken`.

L'exemple crée un objet « poulet », `MyChicken`, appelé Arthur, de deux ans d'âge. Il traite ensuite cet objet de différentes manières. Par exemple, Arthur prend un peu d'âge, et donc le code actualise celui-ci. Remarquez comment le code combine l'usage d'un mutateur/setter, `SetAge()`, avec un accesseur/getter, `GetAge()`, pour effectuer ce calcul. Après chaque modification, le code affiche les objets résultants. L'étape finale demande à Arthur de prononcer quelques mots.

3. **Choisissez la commande Run Module dans le menu Run.**

Une fenêtre Python en mode Shell va s'ouvrir. Vous y voyez les étapes utilisées pour travailler avec `MyChicken` comme l'illustre la [Figure 14.15](#). Comme vous pouvez le constater, l'héritage simplifie grandement la création de nouvelles classes lors que celles-ci possèdent suffisamment de points communs pour qu'il soit intéressant de construire une classe parent qui contiendra l'essentiel du code.

Figure 14.15 :
Arthur prononce
quelques mots
après son
anniversaire.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre14\AnimalsTest.py =====
Arthur est un poulet âgé de 2 ans.
Arthur est un poulet âgé de 3 ans.
Désolé, Arthur sera toujours un poulet.
Arthur est un poulet âgé de 3 ans.
Arthur dit Cot, Cot, Cot, Codette !
>>> |
```

Ln: 10 / Col: 4

Quatrième partie

Effectuer des tâches avancées

Dans cette partie...

- ▶ Créer un fichier.
- ▶ Lire un fichier.
- ▶ Mettre à jour un fichier.
- ▶ Supprimer un fichier.
- ▶ Envoyer un e-mail.

Chapitre 15

Enregistrer des données dans des fichiers

Dans ce chapitre :

- ▶ Comprendre comment les stockages permanents fonctionnent avec les applications.
- ▶ Décider comment travailler avec des contenus enregistrés de manière permanente.
- ▶ Enregistrer un nouveau fichier.
- ▶ Lire le contenu d'un fichier.
- ▶ Modifier le contenu d'un fichier.
- ▶ Supprimer un fichier du disque.

Jusqu'ici, nous nous sommes beaucoup préoccupés de l'affichage d'informations sur l'écran. Mais, en réalité, les applications sont centrées sur le besoin de travailler avec des données d'une certaine manière. Les données sont au cœur de toutes les applications, car c'est ce qui intéresse l'utilisateur. Vous risquez d'être extrêmement désappointé le jour où vous aurez à présenter une superbe application, et où les utilisateurs présents ne seront intéressés que par une chose : est-ce que votre programme va les aider à terminer plutôt leur travail ? En fait, les meilleures

applications sont invisibles, mais elles présentent les données de la façon la plus appropriée possible aux besoins des utilisateurs.

Si les données sont au centre des applications, alors leur enregistrement sous une forme permanente est également important. Pour la plupart des développeurs, cette question tourne autour de supports de stockage tels qu'un disque dur, un disque SSD, une clé USB, ou toute autre technologie comparable (l'enregistrement des données sur un nuage Internet fonctionne également très bien, mais il s'agit d'un sujet qui nécessite des techniques de programmation spécifiques, et qui dépasse donc les frontières de ce livre). Les données présentes en mémoire sont temporaires, puisqu'elles disparaissent quand la machine est arrêtée. Un support de stockage permanent est donc une nécessité pour pouvoir les retrouver de session en session.



Ce chapitre vous aide également à comprendre les quatre opérations de base que vous pouvez effectuer sur les fichiers : créer, lire, mettre à jour et supprimer. Quelle que soit la manière dont sont gérés les supports de stockage permanents, vos applications doivent être capables d'effectuer ces quatre opérations afin de fournir une solution complète à l'utilisateur. Bien entendu, tout cela réclame de la sécurité, de la fiabilité et des contrôles. Ce chapitre vous aide aussi à définir quelques règles concernant *l'intégrité des données*, disons pour simplifier ce qui concerne les erreurs potentielles qui peuvent surgir lors des quatre opérations de base.

Comprendre le fonctionnement des supports de stockage permanents

Vous n'avez pas besoin de comprendre absolument chaque détail sur le fonctionnement des supports de stockage permanents pour les utiliser. Par exemple, savoir comment le plateau d'un disque tourne (à condition qu'il tourne) n'a aucune importance. Pour autant, la plupart des plates-formes adoptent à ce sujet un ensemble de principes de base, qui ont été développés au fil du temps, et ce depuis la préhistoire de l'informatique.

Les données sont enregistrées dans des *fichiers*. Vous avez certainement déjà un certain nombre de connaissances là-dessus, puisque toute application un tant soit peu utile repose sur des fichiers. Lorsque vous ouvrez par exemple un document dans votre traitement de texte, vous ouvrez en réalité un fichier de données contenant (mais pas seulement) les mots que vous ou quelqu'un d'autre a tapé.

Typiquement, les noms des fichiers possèdent une *extension* qui en définit le type. Ces extensions sont standardisées pour chaque application, et sont séparées de la partie principale du nom par un point, comme dans `MesDonnées.txt`. Dans ce cas, `.txt` est l'extension du fichier, et vous avez probablement sur votre ordinateur une application qui est capable d'ouvrir ce type de document (en fait, vous devriez même en avoir plusieurs, car l'extension `.txt`, qui désigne un fichier de texte brut, est très courante).

En interne, les fichiers structurent les données d'une certaine manière afin qu'elles puissent être facilement lues et écrites par la ou les applications qui s'en

servent. Toute application doit connaître la structure des fichiers qu'elle a à traiter afin de pouvoir interagir avec les données contenues dans ces fichiers.

Les exemples de ce chapitre vont utiliser une structure de fichier simple afin de proposer un code facilement compréhensible, mais sachez que, dans le monde réel, les structures des fichiers peuvent être d'une grande complexité.

Les fichiers seraient pratiquement impossibles à localiser si vous les entassiez tous au même endroit sur votre disque dur. C'est pourquoi ils ont besoin d'être organisés selon un schéma précis, en l'occurrence un système de *dossiers* (certains disent aussi *répertoires*). Tout système de stockage permanent est basé sur un tel système de dossiers (ou de répertoires) pour aider à organiser les données, ainsi qu'à rendre les fichiers individuels nettement plus faciles à retrouver. Pour localiser tel ou tel fichier afin d'interagir avec les données qu'il contient, vous devez évidemment savoir dans quel dossier il se trouve.

Les dossiers sont organisés à leur tour d'une manière hiérarchisée qui commence par le niveau le plus élevé du disque dur. Par exemple, si vous avez téléchargé les fichiers contenant les exemples de ce livre, vous devriez retrouver ceux-ci dans un dossier appelé `BP4D`. Cependant, ce dossier ne contient en réalité aucun code source, mais uniquement des *sous-dossiers* correspondant à chaque chapitre. Et c'est dans ces sous-dossiers que se trouve le code source des exemples. Ainsi, pour le présent chapitre, vous devez regarder ce que contient le dossier `BP4D\Chapitre15`.



Remarquez l'utilisation de la barre oblique inversée (\) pour séparer le nom du dossier de celui d'un sous-dossier. Certaines plates-formes emploient la barre oblique standard (/). C'est évidemment un élément que vous devez vérifier sur votre propre système.

La hiérarchie des dossiers est appelée un *chemin d'accès*. Du fait que Python doit savoir où retrouver les ressources dont vous avez besoin à partir des chemins d'accès que vous lui fournissez, nous n'y reviendrons pas spécialement dans ce livre. Par exemple, `H:\BP4D\Chapitre15` est le nom du chemin d'accès complet au code source de ce chapitre sur mon ordinateur Windows. Une telle définition trace la route du début jusqu'à la fin. On appelle cette syntaxe un *chemin d'accès absolu*. Une autre méthode consiste à partir du dossier courant, puis à poursuivre le parcours dans la hiérarchie jusqu'au but à atteindre. On parle alors de *chemin d'accès relatif*.

Créer du contenu

Un fichier peut contenir des données structurées ou non structurées. Une base de données, qui est formée d'enregistrements contenant des informations spécifiques, est un bon exemple de *données structurées*. Une base de données du personnel comprendrait des colonnes pour les noms, l'adresse, la fonction, ainsi qu'un identifiant personnel, et ainsi de suite. Un exemple de *données non structurées* est fourni par un fichier de traitement de texte dans lequel est enregistré du texte ayant un contenu quelconque dans un ordre indéterminé. Pour autant, dans les deux cas, l'application doit savoir comment

effectuer les opérations de base sur ce fichier. En particulier, son contenu doit être préparé de sorte que l'application soit capable d'en lire les données, ou encore d'en écrire de nouvelles.

Mais dans le cas d'un traitement de texte, le document doit suivre une série de règles. Supposons qu'il s'agisse d'un simple texte brut de forme. Mais, même ainsi, chaque paragraphe doit posséder une sorte de délimiteur qui indique à l'application où il se termine et où commence le suivant. L'application peut alors lire le paragraphe jusqu'à ce qu'elle trouve ce délimiteur, puis commencer un nouveau paragraphe. Plus le traitement de texte devient évolué, et plus il doit structurer ses données. Par exemple, la mise en forme de certaines parties du texte, ou plus simplement la police de caractère utilisée, sont autant d'informations qui doivent être enregistrées dans le fichier du document.



Les éléments qui rendent utilisable le contenu d'un stockage permanent sont le plus souvent masqués. Tout ce que vous voyez lorsque vous travaillez avec un fichier, ce sont les données elles-mêmes. Et tout ce qui relève du formatage au sens large reste invisible pour tout un tas de raisons, notamment :

- Un caractère de contrôle, comme un retour chariot ou un saut de ligne, est normalement invisible par défaut au niveau de la plate-forme.
- L'application se sert de combinaisons de caractères spéciaux, comme des virgules, des guillemets ou encore des tabulations, pour délimiter les données. Ces combinaisons sont traitées par l'application lorsqu'elle lit les données.

- ✓ Au cours du processus de lecture, les caractères sont convertis dans une autre forme, comme dans le cas d'un traitement de texte qui doit interpréter la mise en forme des caractères. Cette mise en forme apparaît sur l'écran, mais, en arrière-plan, le fichier contient des codes spéciaux servant à définir ce formatage.
- ✓ Le contenu du fichier peut être enregistré dans un format spécifique, comme du HTML pour une page Web, ou encore du XML pour un document Word. Ce format alternatif est interprété et présenté à l'écran d'une manière compréhensible par l'utilisateur.



Il existe bien entendu d'autres règles. Ainsi, le format `.docx` de Microsoft Word enregistre les données non seulement au format XML, mais aussi en les compressant au format Zip. Ceci permet de réduire le volume occupé par les documents sur le disque.

Maintenant que vous avez une meilleure idée de ce que peut réclamer la préparation d'un contenu destiné à être enregistré de manière permanente, il est temps de passer à un exercice pratique. Ici, la stratégie de formatage va être assez simple. Cet exemple accepte une saisie, puis il la formate pour la préparer à l'enregistrement, et il présente cette version formatée à l'écran (mais sans la sauvegarder sur le disque pour l'instant). Vous le retrouverez également dans les fichiers téléchargeables `FormattedData.py` (qui contient la classe servant à formater l'information) et `FormattedDataTest.py` (qui affiche les données à l'écran).



Du fait que l'exemple va se dérouler en plusieurs étapes, faites d'abord une copie du dossier \BP4D\Chapitre15 sous un nouveau nom (par exemple BP4D\Chapitre15a).

1. Ouvrez une fenêtre de fichier Python.

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
class FormatData:
    def __init__(self, Name="", Age=0, Married=False):
        self.Name = Name
        self.Age = Age
        self.Married = Married

    def __str__(self):
        OutString = "{0}', {1}, {2}".format(
            self.Name,
            self.Age,
            self.Married)
        return OutString
```

Cette classe est simplifiée. Normalement, vous devriez y trouver des accesseurs (reportez-vous à ce sujet au Chapitre 14) et du code d'interception des erreurs. Mais cela suffit pour notre démonstration.

L'élément principal à observer est la fonction `__str__()`. Remarquez qu'elle formate la sortie d'une manière spécifique. La valeur chaîne Name est délimitée par des apostrophes. De plus, chaque valeur est séparée de l'autre par une virgule. Cette mise en forme est en fait courante. Il s'agit du format CSV (Comma Separated Value), qui est employé sur de nombreuses plates-formes, car il est facile à interpréter et ne contient que des informations purement textuelles.

3. Sauvegardez ce code sous le nom de FormattedData.py.



Si vous vous servez du fichier téléchargeable, ne sauvegardez pas cette version allégée de `FormattedData.py` dans le dossier `BP4D\Chapitre15` ! Vous écraseriez purement et simplement la version originale. N'oubliez pas de faire au préalable une copie de travail de ce dossier afin de ne rien perdre (ou plutôt de vous éviter d'avoir à faire les saisies qui suivront à la main...).

4. Ouvrez une autre fenêtre de fichier de Python.

5. Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :

```
from FormattedData import FormatData

NewData = [FormatData("Georges", 65, True),
            FormatData("Sophie", 47, False),
            FormatData("Pierre", 52, True)]

for Entry in NewData:
    print(Entry)
```

Ce code commence par importer la classe `FormatData` du fichier `FormattedData.py`. Ici, ce fichier ne contient qu'une seule classe. Mais vous devez garder cette technique présente à l'esprit pour le cas où vous n'auriez besoin que d'une seule classe dans un module qui en contient plusieurs.

La plupart du temps, vous travaillez sur plusieurs enregistrements à la fois lorsque vous sauvegardez des données sur le disque. Il peut par exemple s'agir d'un texte entier, ou encore des lignes d'une base de données. Ici, l'exemple crée une liste d'enregistrements et les place dans `NewData`. En l'occurrence, `NewData` représente la totalité du document. Bien entendu, dans une application dite de production la représentation prendrait vraisemblablement d'autres formes, mais l'idée est la même.

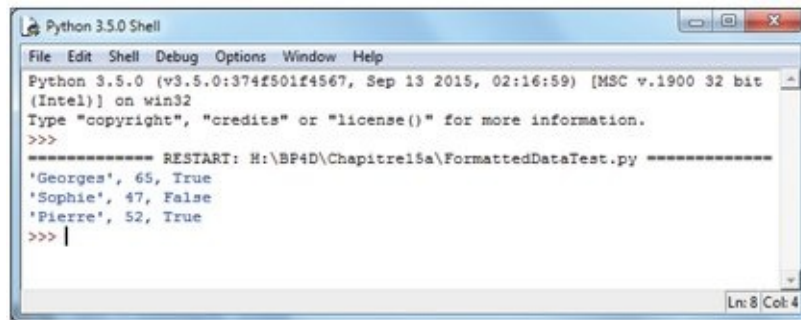
Toute application qui a besoin de sauvegarder des données doit passer pour cela par un certain type de boucle de sortie. Pour l'instant, cette boucle consiste simplement à afficher les données sur l'écran. Nous passerons plus loin à

leur enregistrement physique.

3. Choisissez la commande **Run Module** dans le menu **Run**.

Une fenêtre Python en mode Shell va s'ouvrir. La [Figure 15.1](#) illustre l'affichage obtenu. Il montre comment les données apparaîtraient dans le fichier. Dans ce cas, chaque enregistrement est séparé du suivant par une combinaison retour chariot et saut de ligne. En d'autres termes, Georges, Sophie et Pierre sont tous des enregistrements distincts dans le fichier. Les *champs* (ou éléments de donnée) sont séparés par une virgule. Le champ de texte est placé entre apostrophes pour éviter toute confusion avec d'autres types de données.

Figure 15.1 : Les données de cet exemple sont converties au format CSV.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre15a\FormattedDataTest.py =====
'Georges', 65, True
'Sophie', 47, False
'Pierre', 52, True
>>> |
```

Créer un fichier

Toute donnée créée par l'utilisateur (et/ou l'application) et qui est destinée à être réutilisée pendant plus d'une session doit être sauvegardée sur un support permanent quelconque. Créer un fichier, puis y placer les données voulues est donc une part essentielle du travail avec Python. Vous pouvez utiliser les étapes qui suivent pour créer le code qui va écrire les données sur le disque dur. Cet exemple est également disponible dans les fichiers téléchargeables `FormattedData.py` et `CreateCSV.py`.



À nouveau, je vous conseille de travailler sur une nouvelle copie du dossier BP4D\Chapitre15.

1. **Ouvrez une fenêtre de fichier Python et chargez-y le fichier FormattedData.py.**

Vous devriez retrouver le code créé dans la section précédente. Dans ce qui suit, nous allons le modifier de manière à ce que la classe puisse maintenant sauvegarder un fichier sur le disque.

2. **Ajoutez l'instruction import suivante tout au début du fichier :**

```
import csv
```



Le module `csv` de Python contient tout ce qu'il faut pour travailler avec les fichiers CSV.

Python supporte nativement un grand nombre de types de fichiers, et il existe également des bibliothèques provenant de sources externes. En cas de besoin, vous devriez donc trouver votre bonheur soit directement dans Python, soit en faisant une recherche sur le Web. Malheureusement, il n'existe pas de liste claire sur les types de fichiers supportés. Vous pouvez le cas échéant consulter les adresses suivantes :

<https://docs.python.org/3/library/archiving.html> (formats d'archives) et

<https://docs.python.org/3/library/fileformats.html> (formats de fichiers).

3. **Tapez le code suivant à la fin du fichier en appuyant sur Entrée à la fin de chaque ligne :**

```
def SaveData(Filename = "", DataList = []):
    with open(Filename,
              "w", newline='\n') as csvfile:
        DataWriter = csv.writer(
            csvfile,
            delimiter='\n',
            quotechar=" ",
            quoting=csv.QUOTE_NONNUMERIC)
        DataWriter.writerow(DataList)
        csvfile.close()
        print("Données enregistrées !")
```



Assurez-vous absolument que `SaveData()` est correctement indentée. Si vous ajoutez `SaveData()`, mais que cette méthode n'est pas indentée comme il faut sous la classe `FormatData`, Python considérera qu'il s'agit d'une fonction indépendante de cette classe. La manière la plus simple de s'y prendre consiste à suivre la même indentation que pour les fonctions `__init__()` et `__str__()`.

Remarquez que la méthode accepte deux arguments en entrée : le nom du fichier utilisé pour sauvegarder les données, et une liste d'éléments à enregistrer. Il s'agit ici d'une méthode de classe, et non d'une méthode d'instance. Vous verrez plus loin en quoi cette façon de procéder est un avantage. Par défaut, l'argument `DataList` est vide. De cette manière, la méthode ne provoquera pas d'exception si l'appelant ne lui passe rien. Bien entendu, il serait possible d'ajouter du code pour traiter le cas d'une liste vide considérée comme une erreur.



L'instruction `with` demande à Python d'effectuer une série de tâches à partir d'une ressource spécifique, en l'occurrence un fichier ouvert de type `csvfile`, et appelé `TestFile.csv`. La fonction `open()` accepte différentes entrées selon la manière dont vous l'utilisez. Ici, vous l'ouvrez en mode écriture (signifié par la lettre `w`, pour *write*). L'attribut `newline` indique à Python qu'il doit traiter le caractère de contrôle `\n`

comme un saut de ligne.

Pour écrire la sortie, vous avez besoin d'un objet particulier. Ici, l'objet `DataWriter` est configuré pour utiliser le type de fichier `csvfile`, pour définir `\n` comme délimiteur, marquer les éléments par un espace, et utiliser des apostrophes uniquement avec les valeurs non numériques. À ce stade, nous supposons que c'est exactement ce dont vous avez besoin pour rendre votre sortie utilisable.

En fait, écrire des données demande moins d'efforts que ce que vous pourriez croire. Un simple appel à `DataRow.writerow()` avec `DataList` en entrée est tout ce dont vous avez besoin. Refermez toujours vos fichiers lorsque vous n'en avez plus besoin. Cette action vide les données sur le disque dur en s'assurant qu'elles sont bien écrites. Le code se termine en vous informant que l'opération s'est déroulée avec succès.

4. **Sauvegardez ce code sous le nom de `FormattedData.py`.**



Si vous vous servez du fichier téléchargeable, ne sauvegardez pas cette version allégée de `FormattedData.py` dans le dossier `BP4D\Chapitre15` ! Vous écraseriez purement et simplement la version originale. N'oubliez pas de faire au préalable une copie de travail de ce dossier afin de ne rien perdre (ou plutôt de vous éviter d'avoir à faire les saisies qui suivront à la main...).

5. **Ouvrez une autre fenêtre de fichier de Python.**
6. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from FormattedData import FormatData

NewData = [FormatData("Georges", 65, True),
            FormatData("Sophie", 47, False),
            FormatData("Pierre", 52, True)]

FormatData.SaveData("TestFile.csv", NewData)
```

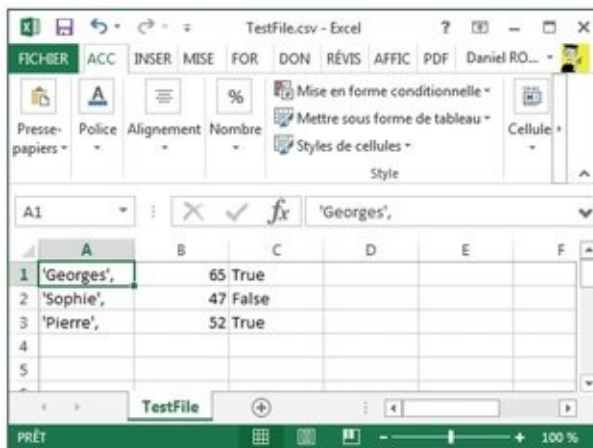
Cet exemple ressemble à celui de la section précédente.

Cependant, au lieu de vous contenter d'afficher le contenu des données, vous l'enregistrez dans un fichier en appelant `FormatData. SaveData ()`. C'est l'une des situations où l'emploi d'une méthode d'instance n'aurait pas d'intérêt particulier, puisque cela vous obligerait en plus à créer une instance de `FormatData` qui ne pourrait rien faire de plus pour vous.

7. Choisissez la commande **Run Module** dans le menu **Run**.

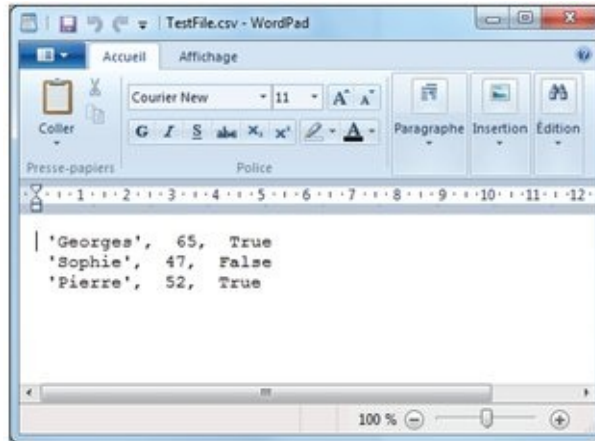
Une fenêtre Python en mode Shell va s'ouvrir. Un simple message vous indique le succès de l'opération. Bien entendu, ce message ne vous dit rien sur vos données. Vous savez simplement que votre nouveau fichier se trouve dans le même dossier que votre code, et qu'il s'appelle `TestFile.csv`. La plupart des plates-formes disposent d'une application par défaut pour ouvrir ce type de fichier. Dans le cas de Windows, il s'agit généralement de Microsoft Excel, ou encore de WordPad (parmi d'autres). Les [Figures 15.2](#) et [15.3](#) illustrent ces deux cas. À chaque fois, la sortie ressemble étonnamment à celle de la [Figure 15.1](#).

Figure 15.2 : La sortie de l'application telle qu'elle apparaît dans Excel (avec le cas échéant un peu de remise en forme).



	A	B	C	D	E	F
1	'Georges',	65	True			
2	'Sophie',	47	False			
3	'Pierre',	52	True			
4						
5						

Figure 15.3 : La sortie de l'application telle qu'elle apparaît dans WordPad.



Lire le contenu d'un fichier

Vos données sont maintenant enregistrées sur le disque dur. C'est très bien, car elles sont ainsi protégées. Mais ce n'est pas très utile tant que vous ne pouvez pas les relire. Pour cela, vous devez les charger en mémoire et effectuer sur elles les traitements voulus. Les étapes qui suivent vous montrent comment lire des données sur le disque pour pouvoir les afficher à l'écran. Cet exemple peut également être retrouvé dans les fichiers téléchargeables `FormattedData.py` and `ReadCSV.py`.



Là encore, je vous conseille de travailler sur une nouvelle copie du dossier `BP4D\Chapitre15`.

- Ouvrez une fenêtre de fichier Python et chargez-y le fichier `FormattedData.py`.**

Vous devriez retrouver le code créé dans la section précédente. Dans ce qui suit, nous allons le modifier de manière à ce que la classe puisse maintenant relire un fichier sauvegardé sur le disque.

- Tapez le code suivant à la fin du fichier en appuyant sur Entrée à la fin de chaque ligne :**

```
def ReadData(Filename = ""):
    with open(Filename,
              "r", newline='\n') as csvfile:
        DataReader = csv.reader(
            csvfile,
            delimiter="\n",
            quotechar=" ",
            quoting=csv.QUOTE_NONNUMERIC)

        Output = []
        for Item in DataReader:
            Output.append(Item[0])

    csvfile.close()
    print("Données lues !")
    return Output
```



À nouveau, assurez-vous absolument que `ReadData()` est correctement indentée. Si vous ajoutez `ReadData()`, mais que cette méthode n'est pas indentée comme il faut sous la classe `FormatData`, Python considérera qu'il s'agit d'une fonction indépendante de cette classe. La manière la plus simple de s'y prendre consiste à suivre la même indentation que pour les fonctions `__init__()` et `__str__()`.

Ouvrir un fichier en lecture n'est pas très différent de l'ouvrir en écriture. La principale différence est que vous avez besoin de spécifier l'argument `r` (pour *read*) au lieu de `w` (pour *write*) dans la fonction `open()`. Sinon, les arguments sont exactement les mêmes, et ils fonctionnent de la même manière.



N'oubliez pas que vous partez d'un fichier de texte lorsque vous travaillez avec un fichier `.csv`. Certes, il y a des délimiteurs, mais ce n'est tout de même que du texte. Lorsque vous lisez celui-ci en mémoire, vous devez donc rebâtir la structure Python. Dans ce cas, `output` est initialement une liste vide.

Le fichier contient actuellement trois enregistrements séparés par le caractère de contrôle `\n`. Python lit chaque enregistrement via une boucle `for`. Pour lui, tant qu'il n'est

pas arrivé au dernier enregistrement, ce qu'il obtient, ce sont deux entrées de liste. La première contient les données, et la seconde est vide. Seule la première vous intéresse. Ces entrées sont ajoutées à output jusqu'à obtention d'une liste de tous les enregistrements qui apparaissent dans le fichier.

Là encore, n'oubliez pas de refermer votre fichier lorsque vous en avez fini avec lui. Finalement, le code affiche un message de succès avant de retourner `output` (donc la liste des enregistrements) à l'appelant.

3. **Sauvegardez ce code dans FormattedData.py.**
4. **Ouvrez une autre fenêtre de fichier de Python.**
5. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from FormattedData import FormatData  
  
NewData = FormatData.ReadData("TestFile.csv")  
  
for Entry in NewData:  
    print(Entry)
```

Le code de `ReadCSV.py` commence par importer la classe `FormatData`. Il crée ensuite un objet `NewData`, qui est une liste, en appelant `FormatData.ReadData`. Remarquez que l'utilisation d'une méthode de classe est un bon choix dans ce cas, car cela permet de rendre le code plus simple et plus court. L'application fait ensuite appel à une boucle `for` pour afficher le contenu de `NewData`.

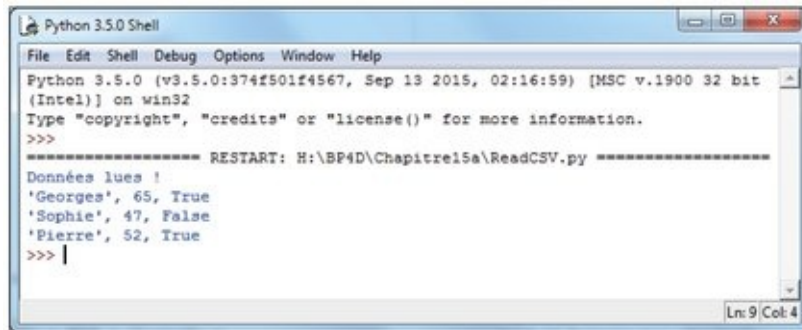
6. **Choisissez la commande Run Module dans le menu Run.**

Vous voyez la sortie illustrée sur la [Figure 15.4](#). Vous pouvez constater que le résultat est le même que sur la [Figure 15.1](#), même si les données ont été écrites sur le disque, puis lues depuis celui-ci. C'est ainsi que les applications qui écrivent et lisent des données sont supposées travailler. Les données doivent en effet avoir la même apparence avant que vous ne les enregistriez et après les avoir relues. Sinon, cela signifierait que l'application ne joue pas correctement son

rôle.

Figure 15.4 :

L'application affiche les données lues dans le fichier.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre15a\ReadCSV.py =====
Données lues !
'Georges', 65, True
'Sophie', 47, False
'Pierre', 52, True
>>> |
```

Mettre à jour le contenu d'un fichier

Certains trouvent que mettre à jour un fichier est une tâche complexe. En fait, ce n'est vrai que si vous voyez tout cela comme une tâche unique. En fait, ce processus se décompose en trois temps :

1. **Lire le contenu du fichier en mémoire.**
2. **Modifier la présentation en mémoire des données.**
3. **Écrire le résultat du traitement dans le fichier.**

Dans la plupart des applications, la seconde étape peut même être à son tour découpée en plusieurs actions :

- ✓ Fournir une présentation à l'écran des données.
- ✓ Autoriser des ajouts à la liste des données.
- ✓ Autoriser des suppressions de la liste des données.
- ✓ Autoriser la modification de données existantes, ce qui peut d'ailleurs se réaliser en ajoutant un nouvel enregistrement contenant les données modifiées, puis en supprimant l'ancienne version.

Dans ce chapitre, vous avez appris à écrire un fichier sur le disque, puis à relire son contenu. Vous avez aussi ajouté des enregistrements à une liste et présenté ceux-ci à l'écran. Il ne reste donc qu'une seule tâche intéressante à étudier : la suppression de données dans une liste. Comme indiqué ci-dessus, le processus de modification revient bien souvent à ajouter un nouvel enregistrement dans lequel sont copiées les données à modifier, puis à supprimer l'ancienne version une fois les changements terminés.



Ne tombez pas en syncope en pensant que vous devriez effectuer chacune des activités mentionnées dans cette section pour chacune de vos applications. Ainsi, un programme de monitoring n'aura pas besoin d'afficher des données à l'écran (sauf si c'est indispensable avant de devenir totalement ennuyeux). Une application de journalisation ne fait que créer de nouvelles entrées (elle ne supprime rien et ne modifie rien). Une messagerie permet d'ajouter des enregistrements et d'en supprimer, mais pas de les modifier (vous ne pouvez rien changer dans les e-mails que vous recevez, uniquement les supprimer ou y répondre, créant ainsi un nouveau message). D'un autre côté, un traitement de texte a besoin de savoir tout faire. C'est donc à vous de réfléchir à cette question selon l'application que vous développez.

Séparer l'interface utilisateur des activités qui se déroulent en arrière-plan est important. Pour rester simple, l'exemple qui suit se concentre sur le travail à effectuer en arrière-plan pour apporter des modifications au fichier créé plus haut. Ces étapes vous montrent comment lire, modifier et enregistrer des données dans un fichier afin de le tenir à jour. Les modifications se décomposent en trois étapes : une

addition, un changement, et une suppression. Pour que vous puissiez effectuer plusieurs tests, les données modifiées sont sauvegardées dans un nouveau fichier. Cet exemple peut également être retrouvé dans les fichiers téléchargeables `FormattedData.py` and `UpdateCSV.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from FormattedData import FormatData
import os.path

if not os.path.isfile("Testfile.csv"):
    print("SVP exécutez le fichier d'exemple CreateFile.py !")
    quit()

NewData = FormatData.ReadData("TestFile.csv")
for Entry in NewData:
    print(Entry)
```

```
print("\r\nAjout d'un enregistrement pour Antoine.")
NewRecord = "Antoine", 23, False
NewData.append(NewRecord)
for Entry in NewData:
    print(Entry)

print("\r\nSuppression de l'enregistrement de Pierre.")
Location = NewData.index("Pierre", 52, True)
Record = NewData[Location]
NewData.remove(Record)
for Entry in NewData:
    print(Entry)

print("\r\nModification de l'enregistrement de Sophie.")
Location = NewData.index("Sophie", 47, False)
Record = NewData[Location]
Split = Record.split(",")
NewRecord = FormatData(Split[0].replace("'", ""),
                        int(Split[1]),
                        bool(Split[2]))
NewRecord.Married = True
NewRecord.Age = 48
NewData.append(NewRecord.__str__())
NewData.remove(Record)
for Entry in NewData:
    print(Entry)

FormatData.SaveData("ChangedFile.csv", NewData)
```

Cet exemple est plus long que les précédents. Il commence par s'assurer que le fichier `Testfile.csv` est bien présent dans le dossier de l'application. C'est un contrôle que vous devriez

effectuer systématiquement. Dans ce cas, vous ne créez pas un nouveau fichier. Vous ne faites que modifier un fichier, qui doit donc effectivement exister. Sinon, l'application se termine tout de suite.

L'étape suivante consiste à lire les données dans `NewData`. Cette partie du code ressemble à ce qui a été développé dans la section précédente.



Vous avez déjà étudié du code utilisant des fonctions de liste dans le Chapitre 12. Cet exemple les met en œuvre dans un travail pratique. La fonction `append()` ajoute un nouvel enregistrement à `NewData`. Remarquez cependant que les données sont ajoutées sous la forme d'une chaîne, et non d'un objet `FormatData`. Du fait que, dans cet exemple, les données sont enregistrées sous la forme de chaîne dans le fichier disque, c'est aussi ce que vous obtenez quand vous les lisez. Vous pouvez au choix procéder comme dans ce code, ou bien créer un objet `FormatData` puis utiliser la méthode `__str__()` afin de sortir les données sous une forme de chaîne.

La prochaine étape consiste à supprimer un enregistrement de `NewData`. Pour cela, vous devez tout d'abord retrouver cet enregistrement. Bien entendu, cela n'est pas bien difficile avec seulement quatre enregistrements (n'oubliez pas qu'Antoine vient d'être ajouté). Dans une situation plus complexe, vous devriez faire appel à la fonction `index()`. Celle-ci retourne la position de l'enregistrement, valeur que vous pouvez ensuite utiliser pour le retrouver. Lorsque c'est fait, il ne vous reste plus qu'à supprimer l'enregistrement périmé en appelant la fonction `remove()`.

Modifier l'enregistrement de Sophie semble plus compliqué, mais, là encore, l'essentiel de la question se résume à gérer l'enregistrement des chaînes sur le disque. Lorsque vous obtenez l'enregistrement à partir de `NewData`, vous récupérez une chaîne formée de trois éléments. La fonction `split()` produit une liste qui contient ces trois entrées sous la

forme de chaînes, ce qui ne convient pas pour cette application. De plus, le nom de Sophie est délimité à la fois par des guillemets et des apostrophes.

La manière la plus simple de traiter ce problème consiste à créer un objet `FormatData` et à convertir chacune de ces chaînes dans le format approprié. Cela signifie supprimer les guillemets en trop pour le nom, convertir la seconde valeur en entier (`int`), et la troisième en un booléen (`bool`). La classe `FormatData` ne fournit pas d'accesseurs, et donc l'application doit modifier directement les champs `Married` et `Age`. L'emploi d'accesseurs serait évidemment une meilleure technique.

L'application ajoute ensuite le nouvel enregistrement et supprime l'ancien de `NewData`. Remarquez l'utilisation de `NewRecord.__str().__` pour convertir le nouvel enregistrement de l'objet `FormatData` en chaîne afin de respecter les conventions CSV.

L'acte final consiste à sauvegarder l'enregistrement modifié. Normalement, vous devriez conserver le même fichier, puisqu'il s'agit d'une mise à jour de celui-ci. Cependant, le code crée ici un nouveau fichier pour que vous puissiez comparer les deux états, avant et après, ou encore effectuer d'autres tests.

3. Choisissez la commande Run Module dans le menu Run.



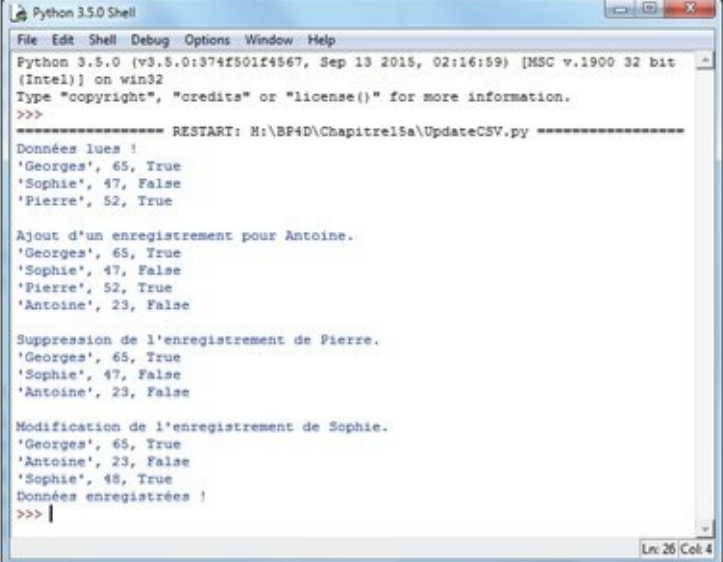
Vous voyez la sortie illustrée sur la [Figure 15.5](#). Remarquez que l'application liste les changements au fur et à mesure des opérations, de manière à ce que vous puissiez vérifier l'état de `NewData`. Ce genre de technique est utile lors du développement d'une application afin de vérifier pas à pas les étapes d'une procédure. Bien entendu, le code d'affichage doit être retiré avant de finaliser l'application.

4. Ouvrez le fichier ChangedFile.csv en utilisant l'application appropriée.

Sur la [Figure 15.6](#), la sortie est affichée avec WordPad, mais le résultat serait identique avec d'autres programmes. Dans tous les cas, vous devriez retrouver les mêmes données.

Figure 15.5 :

L'application montre chaque modification au fur et à mesure de son exécution.



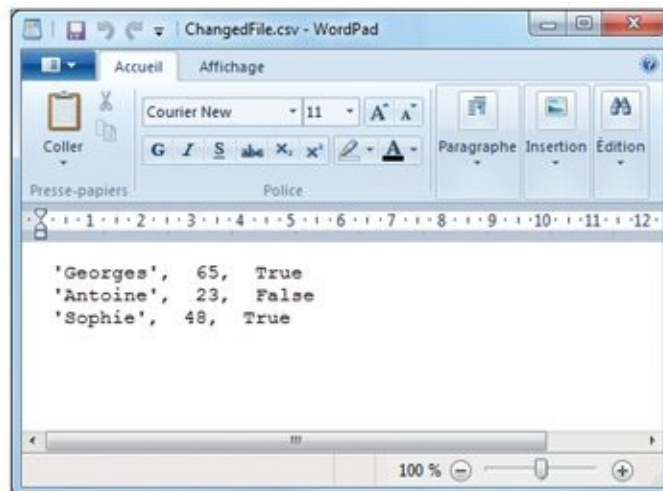
```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: H:\BP4D\Chapitre15a\UpdateCSV.py =====
Données lues !
'Georges', 65, True
'Sophie', 47, False
'Pierre', 52, True

Ajout d'un enregistrement pour Antoine.
'Georges', 65, True
'Sophie', 47, False
'Pierre', 52, True
'Antoine', 23, False

Suppression de l'enregistrement de Pierre.
'Georges', 65, True
'Sophie', 47, False
'Antoine', 23, False

Modification de l'enregistrement de Sophie.
'Georges', 65, True
'Antoine', 23, False
'Sophie', 48, True
Données enregistrées !
>>> |
```

Figure 15.6 : Les informations mises à jour sont enregistrées dans le fichier ChangedFile.csv.



Supprimer un fichier

La section précédente vous a expliqué comment ajouter, supprimer et modifier des enregistrements dans un fichier. Cependant, vous pouvez aussi avoir besoin à un moment donné de supprimer celui-ci. Les

étapes qui suivent vous montrent comment procéder. Vous pouvez également le retrouver dans le fichier téléchargeable `DeleteCSV.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
import os
os.remove("ChangedFile.csv")
print("Fichier supprimé !")
```



Cette tâche semble simple, et c'est effectivement le cas. Tout ce dont vous avez besoin, c'est d'un appel à la fonction `os.remove()` en lui fournissant le nom du fichier ainsi que son chemin d'accès (par défaut, celui-ci est le dossier courant, et vous devrez donc spécifier le chemin complet si celui-ci est différent). La facilité avec laquelle il est possible de supprimer un fichier est telle que cela en devient presque angoissant. Il est donc plus que recommandé de poser des garde-fous.

Bien entendu, il est aussi possible de supprimer d'autres éléments, et il y a donc certaines fonctions qu'il est utile de connaître :

- `os.rmdir()` : Supprime le dossier spécifié. Celui-ci doit être vide. Dans le cas contraire, Python affichera un message d'exception.
- `shutil.rmtree()` : Supprime le dossier spécifié, ainsi que tous ses sous-dossiers et tous les fichiers qui s'y trouvent. Cette fonction est particulièrement dangereuse, car elle détruit tout sans aucun contrôle (Python suppose que vous savez *vraiment* ce que vous faites). Vous risquez donc au passage de perdre de

précieuses données...

3. **Choisissez la commande Run Module dans le menu Run.**

L'application affiche le message Fichier supprimé ! Vous pouvez le vérifier en regardant le contenu du dossier qui contenait `changedFile.csv`.

Chapitre 16

Envoyer un e-mail

Dans ce chapitre :

- ▶ Comprendre le cheminement des e-mails.
- ▶ Développer une application de messagerie.
- ▶ Tester l'application de messagerie.

Dans ce chapitre, vous allez apprendre comment envoyer un e-mail en utilisant Python. Mais, et c'est le plus important, il concerne ce qui peut se passer lorsque vous essayez de communiquer vers le monde extérieur (autrement dit, à l'extérieur de votre système). Mais si ce chapitre se concentre sur un sujet, les e-mails, il contient aussi des principes que vous pouvez utiliser pour d'autres tâches. Par exemple, si vous travaillez pour un service externalisé, vous avez souvent besoin de créer quelque chose qui ressemble à ce que vous devez faire pour traiter des messages.

Pour que tout cela reste compréhensible, nous allons nous baser sur la notion de courrier postal en tant qu'équivalent des e-mails dans le monde réel. La comparaison est d'ailleurs justifiée, car les e-mails sont effectivement conçus sur le même modèle.

Même si, en d'autres temps, ce type d'échange nécessitait que l'expéditeur et le destinataire soient connectés en même temps (ce que l'on appellerait aujourd'hui un *chat*), nous allons considérer ce sujet tel qu'il se présente aujourd'hui, c'est-à-dire un mécanisme d'échange de documents de divers types.

Les exemples de ce chapitre reposent sur la disponibilité d'un serveur SMTP (Simple Mail Transfert Protocol, ou protocole de transfert de courrier simple, si vous voulez). Voyez à ce sujet l'encadré « Courrier et serveur SMTP ».

Courrier et serveur SMTP

Lorsque vous travaillez avec une messagerie, vous pouvez voir régulièrement quelque chose qui fait référence à l'acronyme SMTP (Simple Mail Transfert Protocol). Bien entendu, la chose peut vous paraître très ésotérique, et même technique, ce qui est plutôt vrai. En fait, l'essentiel est de savoir que cela marche, et que vous pouvez donc envoyer des messages sans souci particulier. Pour autant, comprendre qu'il s'agit d'un peu plus qu'une boîte noire qui prend le message que veut envoyer l'expéditeur pour l'expédier quelque part ailleurs peut ne pas être inutile.

Si vous prenez les lettres de l'acronyme SMTP dans l'ordre inverse, vous voyez plusieurs éléments :

✓ **Protocole** : C'est un ensemble standardisé de règles. Les e-mails ont

besoin d'un protocole accepté et reconnu par tout le monde. Sinon, vous seriez incapable de recevoir ou d'envoyer des messages (et pire encore, il n'y aurait même pas d'Internet sans protocoles).

✓ **Transfert de messages** : Les documents sont envoyés d'un endroit à un autre, selon le même modèle que du courrier postal. Dans le cas des e-mails, toute l'affaire repose sur de courtes commandes que votre application envoie vers un serveur SMTP. Par exemple, la commande MAIL FROM indique au serveur SMTP qui envoie le message, tandis que la commande RCPT TO lui explique où l'envoyer.

✓ **Simple** : Là, c'est juste pour dire que ce protocole ne réclame que peu d'efforts et de ressources. En règle générale, plus c'est simple, et plus c'est fiable.

Si vous voulez vraiment en savoir plus, essayez de voir ce que dit l'adresse <http://computer.howstuffworks.com/e-mail-messaging/email.htm>, et en particulier la page de ce site (<http://computer.howstuffworks.com/e-mail-messaging/email3.htm>).

Comprendre ce qui se passe lorsque vous envoyez un e-mail

Les e-mails sont devenus si courants et si fiables que la plupart des gens ne se posent même pas la question de savoir comment un tel miracle est possible. En fait, la même question vaut pour le

courrier postal. Si vous y réfléchissez un peu, le simple fait qu'un courrier, qu'il soit posté dans une boîte aux lettres ou expédié via un logiciel de messagerie, parte d'un point A pour arriver sans erreur et sans encombre à un point B semble être chose pratiquement impossible. Et pourtant, elle tourne, comme disait Copernic ! Les sections qui suivent examinent ce qui se passe lorsque vous écrivez un message électronique, que vous cliquez sur Envoyer, et que le destinataire le reçoit pratiquement dans l'instant qui suit.

Un e-mail, c'est comme du courrier

La meilleure manière d'envisager les choses, c'est de considérer que lire un e-mail est comme lire une lettre sur papier. Lorsque vous écrivez une lettre, il vous faut deux morceaux de papier au minimum. Le premier sert à rédiger votre lettre, et le second est une enveloppe. En supposant que le service postal est honnête (et il l'est !), le papier contenu dans l'enveloppe n'est jamais examiné par qui que ce soit d'autre que le destinataire. On peut en dire autant d'un message électronique. Enfin, en théorie...

Un e-mail se décompose ainsi :

➤ **Message** : C'est le contenu de l'e-mail, qui est formé de deux éléments :

- *En-tête* : C'est la partie de l'e-mail qui contient le sujet, la liste des destinataires, ainsi que d'autres compléments, comme le degré d'urgence du message.

- *Corps* : C'est le message proprement dit. Celui-ci peut être rédigé en texte brut, formaté en HTML, contenir un ou plusieurs documents, ou toute combinaison possible de tous ces éléments.

✓ **Enveloppe** : C'est ici un conteneur pour le message. L'enveloppe fournit des informations sur l'expéditeur et le destinataire, exactement comme pour un courrier papier. Par contre, vous n'avez pas besoin de coller un timbre.

Pour écrire et lire vos e-mails, vous avez besoin d'un logiciel de messagerie. Lors de la configuration de celui-ci, vous définissez les renseignements qui concernent votre compte d'utilisateur, ce qui permet au « service postal » de savoir que vous avez bien le droit d'envoyer et de recevoir des messages en passant par leurs « boîtes aux lettres » et leurs « facteurs ».

Lorsque vous rédigez un message et que vous cliquez sur Envoyer :

1. **L'application place votre missive, l'en-tête en premier, dans une enveloppe virtuelle qui comprend les informations sur l'expéditeur (vous) et le destinataire.**
2. **L'application utilise les informations de votre compte pour contacter le serveur SMTP et lui transmettre votre message.**
3. **Le serveur SMTP lit uniquement les informations qu'il trouve sur l'enveloppe. Il vérifie que vous êtes bien un expéditeur autorisé, puis il envoie le message au destinataire.**
4. **L'application de messagerie du destinataire se connecte à un serveur local, elle y récupère le message, puis affiche le contenu de celui-ci sur son**

écran.

Évidemment, le processus réel est un peu plus compliqué que cette description. Mais, pour l'essentiel, c'est ainsi que les choses se passent. En fait, les étapes essentielles sont identiques au fonctionnement de n'importe quel service postal. Vous prenez la place de votre ordinateur, et le destinataire en fait de même. Le serveur SMTP est remplacé par un bureau de poste, le serveur local par un centre de tri, et les connexions électroniques par un facteur ou une factrice. Dans tous les cas, quelqu'un génère un message, celui-ci est transporté d'une manière ou d'une autre jusqu'à son destinataire, et celui-ci ouvre le message pour le lire (il peut aussi le jeter à la corbeille – encore une image qui relie le monde virtuel au monde réel).

Définir les parties de l'enveloppe

Il y a une différence entre la manière dont l'enveloppe d'un e-mail est configurée et celle dont elle est effectivement gérée. Lorsque vous voyez l'enveloppe d'un e-mail, elle semble ressembler parfaitement à celle d'un courrier contenant l'adresse de l'expéditeur et celle du destinataire. En fait, sur une enveloppe papier, vous trouvez non seulement le nom du destinataire, mais aussi son adresse postale en entier. Et la même chose est vraie pour l'expéditeur, du moins si vous avez correctement rempli toutes les zones. Tous ces éléments déterminent où le service postal devrait déposer le courrier, et où il devrait le renvoyer dans le cas où le destinataire habite une adresse inconnue ou bien s'il a déménagé.

Cependant, lorsqu'un serveur SMTP traite l'enveloppe

d'un e-mail, il doit regarder des adresses spécifiques, et c'est là où son chemin avec une enveloppe papier se met à diverger. En effet, une adresse e-mail contient des éléments spécifiques, notamment ceux-ci :

➤ **Hôte** : L'hôte est à peu près comme les informations inscrites sur une enveloppe. L'adresse de l'hôte fournit l'adresse utilisée par la carte électronique qui est physiquement connectée à l'Internet, et il faut alors gérer tout ce que l'Internet consume ou fournit comme trafic pour cette machine particulière. Un PC, ou une autre plate-forme, peut utiliser des ressources Internet de multiples manières, mais l'adresse de l'hôte reste toujours la même.

➤ **Port** : Un port est un peu comme l'adresse de la rue sur une enveloppe papier. Elle spécifie quelle partie du système devrait recevoir le message. Par exemple, un serveur SMTP utilise généralement pour l'envoi du courrier le port 25. Par contre, les serveurs dits POP3 (Point Of Presence), qui traitent l'arrivée des messages, se servent typiquement du port 110. Votre navigateur mobile utilise normalement le port 80 pour communiquer avec les sites Web. Par contre, les sites dits sécurisés (ceux qui utilisent le protocole https au lieu de http) font appel au port 443. Voyez par exemple à ce sujet l'adresse

https://en.wikipedia.org/wiki/List_of_TCP_and_UDP

➤ **Nom d'hôte local** : Il s'agit de la version « humainement » lisible de la combinaison entre l'hôte et le port. Par exemple, le site Web www.ChezMoi.fr pourrait être résolu en une adresse du genre 55.225.163.40 :80 (où la première partie est l'adresse de l'hôte, et les

chiffres qui suivent les deux-points le port). Python se charge de ces détails en arrière-plan, et vous n'avez donc normalement pas à vous en soucier. Mais il est utile de savoir que cela existe.

Maintenant que vous connaissez ces généralités, il est temps d'y regarder de plus près. Les sections qui suivent décrivent l'enveloppe d'un e-mail de façon plus précise.

Hôte

L'adresse *de l'hôte* est l'identificateur utilisé pour une connexion à un serveur. De même qu'une adresse sur une enveloppe n'est pas la position réelle du destinataire, l'adresse de l'hôte ne définit pas le serveur réel, mais plutôt l'emplacement de ce serveur.



La connexion utilisée pour accéder à la combinaison d'une adresse d'hôte et d'un port est appelée *socket* (c'est comme une prise de courant...). La provenance de ce nom étrange n'a pas d'importance. L'essentiel, c'est que vous pouvez l'utiliser pour trouver toutes sortes d'informations utiles pour comprendre comment fonctionne la messagerie électronique. Les étapes qui suivent vont vous aider à voir tout cela en action. Ce qui compte le plus, c'est de bien comprendre l'idée générale sur ce qu'est l'enveloppe d'un e-mail et les adresses qu'il contient.

1. **Ouvrez une fenêtre de Python en mode Shell.**

Vous voyez l'indicatif habituel.

2. **Tapez `import socket` et appuyez sur Entrée.**

Pour pouvoir travailler avec des *sockets*, vous devrez tout d'abord importer la bibliothèque correspondante. Celle-ci contient toutes sortes d'attributs compliqués, et il faut donc

s'en servir en prenant des précautions. Mais on y trouve aussi des fonctions intéressantes qui peuvent vous aider à mieux voir comment sont gérées les adresses Internet.

3. **Tapez `socket.gethostbyname("localhost")` et appuyez sur Entrée.**

Vous voyez s'afficher une adresse d'hôte, en l'occurrence `127.0.0.1` qui est l'adresse standard associée au nom d'hôte `localhost` (en fait, celle de votre propre système).

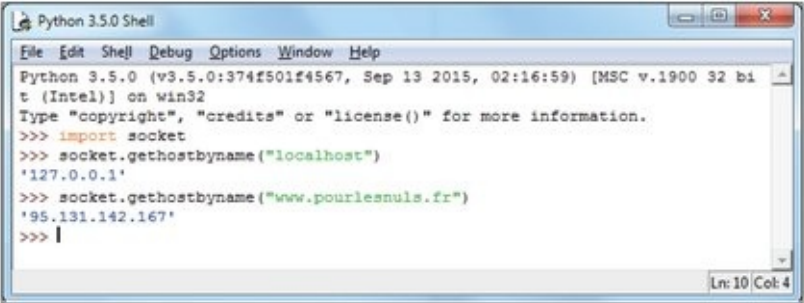
4. **Tapez `socket.gethostbyaddr("127.0.0.1")` et appuyez sur Entrée.**

Attendez-vous à une surprise. Vous obtenez en fait un tuple dont le contenu peut éventuellement vous laisser quelques doutes sur ce qui se passe dans votre machine. Vous constatez cependant que le nom de votre machine est bien présent dans la liste.

5. **Tapez `socket.gethostbyname("www.pourlesnuls.fr")` et appuyez sur Entrée.**

Vous voyez s'afficher la sortie illustrée sur la [Figure 16.1](#). Il s'agit du site Web d'un éditeur bien connu... Le point principal, c'est que cette adresse fonctionne où que vous vous trouviez, et quoi que vous fassiez, exactement comme si elle était inscrite sur une enveloppée papier. Un service postal utilise des adresses qui sont uniques, et l'Internet agit de la même manière.

Figure 16.1 : Les adresses que vous utilisez pour envoyer des e-mails sont uniques.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bi
t (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.gethostbyname("localhost")
'127.0.0.1'
>>> socket.gethostbyname("www.pourlesnuls.fr")
'95.131.142.167'
>>> |
```

6. **Vous pouvez refermer la fenêtre de Python.**

Port

Un *port* est un point d'entrée spécifique dans un serveur. L'adresse de l'hôte spécifie l'emplacement,

mais le port définit où précisément entrer. Même si vous ne précisez pas un numéro de port chaque fois que vous utilisez une adresse d'hôte, cette information est implicite. L'accès est toujours autorisé en combinant l'adresse de l'hôte et le port. Les étapes qui suivent illustrent ce mode de fonctionnement.

1. **Ouvrez une fenêtre de Python en mode Shell.**

Vous voyez l'indicatif habituel.

2. **Tapez `import socket` et appuyez sur Entrée.**

Rappelez-vous qu'un *socket* fournit à la fois l'adresse de l'hôte et le port. Vous l'utilisez pour créer une connexion qui comprend ces deux éléments.

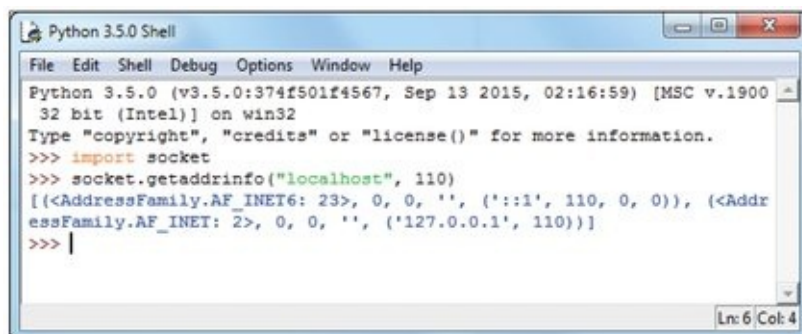
3. **Tapez `socket.getaddrinfo("localhost", 110)` et appuyez sur Entrée.**

La première valeur est le nom de l'hôte sur lequel vous voulez obtenir des informations. La seconde valeur est un numéro de port sur cet hôte, en l'occurrence le port 110.

Vous obtenez une sortie semblable à celle qui est illustrée sur la [Figure 16.2](#). Elle est formée de deux tuples : l'un pour le protocole Internet version 6 (IPv6), l'autre pour le protocole Internet version 4 (IPv4). Chaque tuple contient plusieurs entrées dont vous n'avez en fait pas à vous soucier, puisque vous n'en aurez vraisemblablement jamais besoin. Cependant, la dernière entrée, `('127.0.0.1', 110)`, vous montre l'adresse et le port pour l'hôte local.

Figure 16.2 :

L'hôte local fournit des adresses IPv6 et IPv4.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.getaddrinfo("localhost", 110)
[(<AddressFamily.AF_INET6: 23>, 0, 0, '', ('::1', 110, 0, 0)), (<Addr
essFamily.AF_INET: 2>, 0, 0, '', ('127.0.0.1', 110))]
>>> |
```

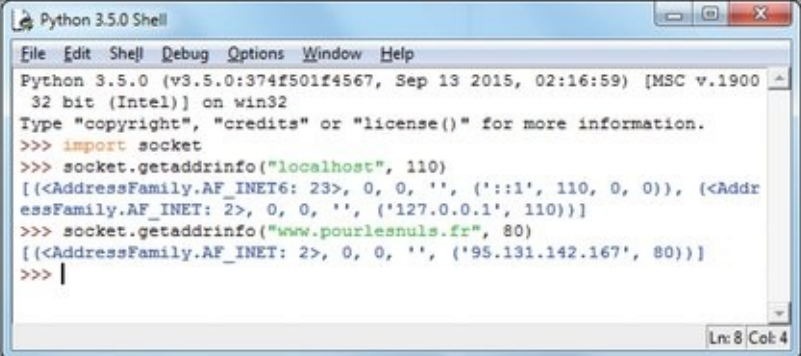
4. **Tapez `socket.getaddrinfo("www.pourlesnuls.fr", 80)` et appuyez sur Entrée.**

La [Figure 16.3](#) illustre la sortie produite par cette commande. Remarquez qu'elle donne uniquement une adresse IPv4 pour le port 80 du site. La méthode `socket.getaddrinfo()` fournit une indication utile pour déterminer comment vous pouvez accéder à un emplacement particulier. Notez qu'à ce jour, la quantité d'adresses IPv4 est proche de la saturation, le nouveau protocole IPv6, beaucoup plus efficace, prenant du temps à se déployer.

5. **Tapez `socket.getservbyport(25)` et appuyez sur Entrée.**

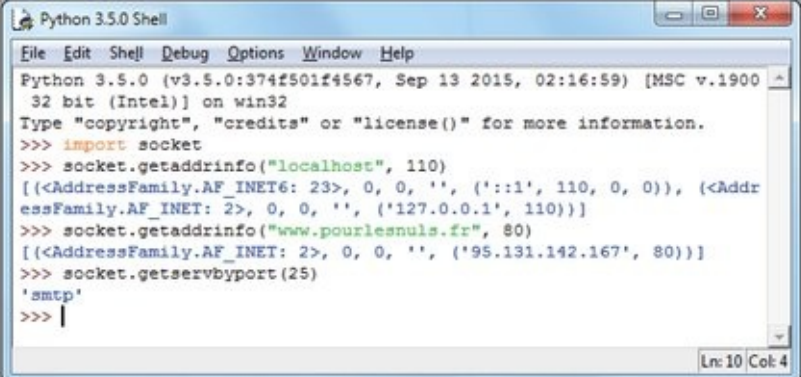
Vous pouvez voir la sortie illustrée sur la [Figure 16.4](#). La méthode `socket.getservbyport()` fournit le moyen de déterminer comment est utilisé un certain port. Le port 25 est dédié au support SMTP sur tous les serveurs. Si vous accédez par exemple à l'adresse `127.0.0.1 : 80`, vous demandez en fait le serveur SMTP sur l'hôte local. En résumé, un port fournit un type spécifique d'accès dans de nombreuses situations.

Figure 16.3 : La plupart des emplacements Internet ne fournissent qu'une adresse IPv4.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.getaddrinfo("localhost", 110)
[(<AddressFamily.AF_INET6: 23>, 0, 0, '', ('::1', 110, 0, 0)), (<AddressFamily.AF_INET: 2>, 0, 0, '', ('127.0.0.1', 110))]
>>> socket.getaddrinfo("www.pourlesnuls.fr", 80)
[(<AddressFamily.AF_INET: 2>, 0, 0, '', ('95.131.142.167', 80))]
>>> |
```

Figure 16.4 : Les ports standardisés procurent des services spécifiques sur chaque serveur.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.getaddrinfo("localhost", 110)
[(<AddressFamily.AF_INET6: 23>, 0, 0, '', ('::1', 110, 0, 0)), (<AddressFamily.AF_INET: 2>, 0, 0, '', ('127.0.0.1', 110))]
>>> socket.getaddrinfo("www.pourlesnuls.fr", 80)
[(<AddressFamily.AF_INET: 2>, 0, 0, '', ('95.131.142.167', 80))]
>>> socket.getservbyport(25)
'smtp'
>>> |
```

6. **Vous pouvez refermer la fenêtre de Python.**



L'information sur le port n'est pas toujours disponible. Lorsque vous ne précisez pas de port, Python utilise une valeur par défaut. Cependant, faire confiance à ce port par défaut n'est pas une bonne idée, car vous ne savez pas exactement à quel service vous accédez. De plus, certains systèmes se servent d'affectation non standard des ports pour des raisons de sécurité. Vous devriez donc prendre l'habitude d'utiliser le numéro du port et de vous assurer qu'il correspond bien à la tâche que vous voulez réaliser.

Nom d'hôte local

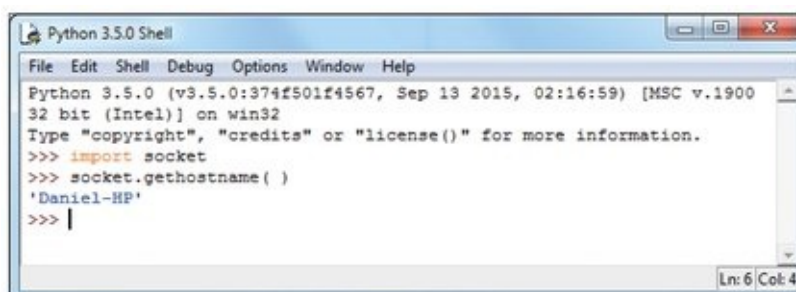
Il s'agit simplement de la forme lisible par un être humain de l'adresse de l'hôte. En fait, les humains ont du mal à comprendre quelque chose comme `127.0.0.1` (et c'est encore bien pire pour la version IPv6). Par contre, un nom d'hôte local est bien plus facile à saisir. Il y a un serveur spécial et une configuration tout aussi spéciale qui permet de traduire les noms locaux en adresses d'hôtes, mais vous n'avez pas à vous en soucier dans ce livre (ni plus généralement ailleurs). Mais il est utile d'avoir quelques notions à ce sujet si votre application se plante soudainement sans raison apparente.

La section « Hôte », plus haut dans ce chapitre, vous a présenté la méthode `socket.gethostbyaddr()`, qui transforme une adresse en nom d'hôte. Vous avez aussi vu le processus inverse avec la méthode `socket.gethostbyname()`. Les étapes qui suivent vous aideront à saisir certaines nuances lors du travail avec le nom d'hôte :

1. **Ouvrez une fenêtre de Python en mode Shell.**
Vous voyez l'indicatif habituel.
2. **Tapez `import socket` et appuyez sur Entrée.**
3. **Tapez `socket.gethostname()` et appuyez sur Entrée.**

Vous voyez s'afficher le nom de votre système local, comme l'illustre la [Figure 16.5](#). Bien entendu, le nom de votre propre système est différent du mien. Mais le principe est le même dans tous les cas.

Figure 16.5 : Vous avez parfois besoin de connaître le nom du système local.

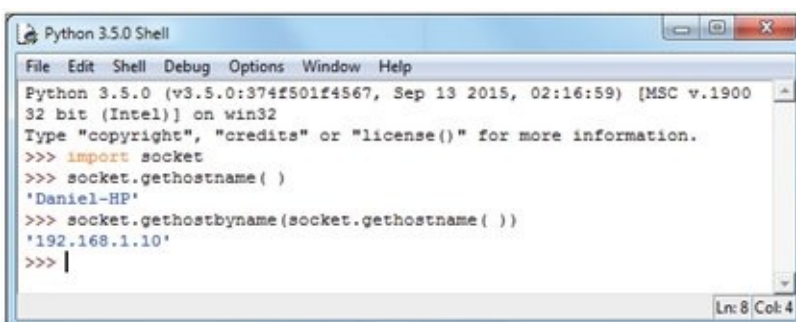


```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.gethostname()
'Daniel-HP'
>>> |
```

4. Tapez `socket.gethostbyname(socket.gethostname())` et appuyez sur Entrée.

Vous voyez maintenant l'adresse IP du système local, comme l'illustre la [Figure 16.6](#). Là encore, ce que vous obtenez chez vous est différent de ce qui apparaît sur la figure. Mais il s'agit d'une méthode que vous pouvez utiliser dans vos applications afin de déterminer l'adresse de l'expéditeur. Cette méthode fonctionne sur n'importe quel système.

Figure 16.6 : Évitez chaque fois que possible d'utiliser des valeurs figées pour le système local.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900
32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import socket
>>> socket.gethostname()
'Daniel-HP'
>>> socket.gethostbyname(socket.gethostname())
'192.168.1.10'
>>> |
```

Définir les parties du courrier

L'enveloppe d'une adresse de messagerie est ce que le serveur SMTP utilise pour router l'e-mail. Cependant, cette enveloppe ne comprend aucun

contenu. Cela, c'est le rôle de la lettre proprement dite. Il faut éviter toute confusion à ce sujet, sachant que la lettre contient aussi le nom de l'expéditeur et celui du destinataire. En fait, ces informations apparaissent dans le courrier exactement comme c'est le cas pour des lettres à caractère professionnel. Elles sont là pour attirer votre attention, et c'est tout. Si vous envoyez un courrier avec ce genre de présentation, la personne qui travaille au centre de tri de même que le facteur ou la factrice n'ont pas besoin de savoir quelles adresses sont écrites sur la lettre. Seule l'enveloppe les intéresse.



Puisque seules comptent pour la bonne circulation du courrier les adresses qui se trouvent sur l'enveloppe, rien ne garantit que ces informations soient correctes dans la lettre elle-même...

La partie « lettre » d'un e-mail est formée de plusieurs éléments, exactement comme pour l'enveloppe. Ils sont pour l'essentiel au nom de trois :

- ✓ **Expéditeur** : Cette information vous dit qui a envoyé le message. Elle contient uniquement l'adresse de messagerie de l'expéditeur.
- ✓ **Destinataire(s)** : Cette information vous dit qui doit recevoir le message. Il s'agit en fait d'une liste d'adresses de messagerie. Cette liste peut évidemment être réduite à un seul élément.
- ✓ **Message** : Contient l'information que vous voulez que le destinataire voie. Celle-ci peut contenir les éléments suivants :
 - *De* : L'adresse humainement lisible de l'expéditeur.

- *À* : L'adresse humainement lisible du ou des destinataires.
- *CC* : Une ou plusieurs autres adresses mises en copie.
- *Objet* : Le sujet du message.
- *Documents* : un ou plusieurs documents, y compris le texte qui apparaît dans le message.

Les e-mails peuvent, le cas échéant, être assez complexes et longs. Ils sont susceptibles de contenir de multiples informations supplémentaires. Pour autant, la plupart des messages ne contiennent que ces simples composants, et c'est la base dont vous avez besoin pour envoyer des e-mails à partir de vos applications. Les sections qui suivent décrivent plus en détail le processus utilisé pour générer une lettre et ses composants.

Définir le message

Envoyer à quelqu'un une enveloppe vide, c'est possible, mais ce n'est pas très excitant. Pour que votre message serve à quelque chose, vous devez lui donner un certain contenu. Python supporte diverses méthodes pour créer des e-mails. La manière la plus simple et la plus fiable consiste à utiliser la fonctionnalité dite MIME (pour Multipurpose Internet Mail Extensions).

Comme la plupart des fonctions de messagerie, les types MIME sont standardisés, et sont donc indépendants de toute plate-forme. Il en existe de nombreux formats. Vous pouvez en apprendre à ce sujet en consultant la page <https://docs.python.org/3/library/email.mime.html>. Les formats les plus courants dans le cas de la messagerie

sont les suivants :

- ✓ **MIMEApplication** : Fournit une méthode pour envoyer et recevoir l'entrée ou la sortie d'une application.
- ✓ **MIMEAudio** : Contient un fichier audio.
- ✓ **MIMEImage** : Contient un fichier image.
- ✓ **MIMEMultipart** : Permet à un message de combiner différentes sous-parties, comme du texte et des images graphiques.
- ✓ **MIMEText** : Contient du texte qui peut être au format ASCII, HTML, ou un autre format standard.

Bien que vous puissiez créer n'importe quelle sorte de message avec Python, le cas le plus simple est celui d'un e-mail qui ne contient que du texte brut. L'absence de tout formatage dans le contenu vous permet de vous concentrer sur les techniques permettant de produire le message, plutôt que sur l'aspect de celui-ci. Les étapes qui suivent vous montrent comment se déroule ce processus, même si votre message n'est pas réellement envoyé ici.

1. **Ouvrez une fenêtre de Python en mode Shell.**
Vous voyez l'indicatif habituel.
2. **Tapez le code suivant en appuyant sur Entrée après chaque ligne :**

```
from email.mime.text import MIMEText
msg = MIMEText("Bonjour le monde !")
msg['Subject'] = "Message de test"
msg['From'] = "John Mueller <John@JohnMuellerBooks.com>"
msg['To'] = "John Mueller <John@JohnMuellerBooks.com>"
```



Ici, le message est un texte brut. Avant de pouvoir faire quoi

que ce soit, vous devez importer la classe correspondante, c'est-à-dire `MIMEText`. Pour créer des formats de message différents, vous devriez importer d'autres classes, ou encore le module `email.mime` en entier.

Le constructeur `MIMEText()` nécessite un texte en entrée. Il s'agit du corps de votre message, ce qui fait qu'il peut être assez long. Pour cet exemple, nous nous contentons d'un simple salut.

Arrivé là, il faut affecter des valeurs à des attributs standards. Cet exemple en montre trois qui doivent être systématiquement définis : `Subject` (l'objet du message), `From` (le nom sous forme humaine et l'adresse de messagerie de l'expéditeur), et `To` (le nom sous forme humaine et l'adresse de messagerie du destinataire).

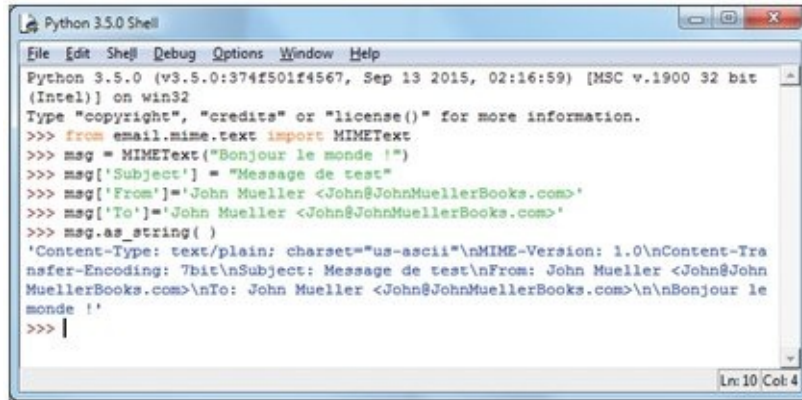
3. Tapez `msg.as_string()` et appuyez sur Entrée.

Vous obtenez la sortie illustrée sur la [Figure 16.7](#). Elle vous montre à quoi ressemble réellement le message. Si vous avez déjà regardé comment se présentait la source d'un e-mail au format texte dans votre application de messagerie, vous ne devriez pas être trop dépaysé.

La section `Content-Type` reflète le type de message que vous avez créé, en l'occurrence `text/plain`, donc du texte brut. La valeur de `charset` spécifie la nature du jeu de caractères utilisé dans le message, de manière à ce que le programme de messagerie du destinataire sache comment le gérer. La partie `MIME-Version` spécifie la version du type MIME, ce qui est aussi utile pour le programme de messagerie qui va décoder tout cela. Finalement, `Content-Transfer-Encoding` détermine comment le message est converti en un flux de bits avant d'être expédié au destinataire.

Figure 16.7 :

Python ajoute certaines informations indispensables pour que votre message puisse être traité correctement.



```
Python 3.5.0 Shell
File Edit Shell Debug Options Window Help
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> from email.mime.text import MIMEText
>>> msg = MIMEText("Bonjour le monde !")
>>> msg['Subject'] = "Message de test"
>>> msg['From'] = "John Mueller <John@JohnMuellerBooks.com>"
>>> msg['To'] = "John Mueller <John@JohnMuellerBooks.com>"
>>> msg.as_string()
'Content-Type: text/plain; charset="us-ascii"\nMIME-Version: 1.0\nContent-Transfer-Encoding: 7bit\nSubject: Message de test\nFrom: John Mueller <John@JohnMuellerBooks.com>\nTo: John Mueller <John@JohnMuellerBooks.com>\n\nBonjour le monde !'
>>> |
```

Spécifier la méthode de transmission

Nous avons vu plus haut comment l'enveloppe est utilisée pour transférer un message d'un point à un autre. Le processus d'envoi d'un e-mail implique la définition d'une méthode de transmission. Certes, Python crée l'enveloppe pour vous et effectue la transmission, mais vous devez tout de même définir les particularités de celle-ci. Les étapes qui suivent vous aideront à comprendre l'approche la plus simple dans le cas de Python. Elles ne conduisent à aucune transmission effective et réussie, à moins de les adapter à une configuration SMTP spécifique.

1. **Utilisez la fenêtre de Python qui est restée ouverte à la fin de la section « Définir le message ».**
Reportez-vous si nécessaire aux étapes de cette section.
2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne (et une seconde fois à la suite de la dernière) :**

```
import smtplib
s = smtplib.SMTP('localhost')
```

Le module `smtplib` contient tout ce dont vous avez besoin pour créer l'enveloppe du message et pour envoyer celui-ci. La première étape du processus consiste à créer une connexion au serveur SMTP, dont vous fournissez le nom dans le constructeur. Si le serveur SMTP indiqué n'existe

pas, l'application va générer immédiatement une erreur vous disant que l'hôte a expressément refusé la connexion.

3. **Tapez `s.sendmail('AdresseExpéditeur', ['AdresseDestinataire'], msg.as_string())` et appuyez sur Entrée.**

Pour que cette étape fonctionne, vous devez remplacer `AdresseExpéditeur` et `AdresseDestinataire` par des adresses de messagerie réelles. Vous ne devez pas inclure ici le format « humainement lisible » de ces adresses.

Il s'agit de l'étape qui crée effectivement l'enveloppe, insère le message proprement dit dans celle-ci, et envoie le tout au destinataire. Notez bien que ces mêmes informations sont fournies de manière totalement indépendante dans l'enveloppe et dans le message lui-même, ce dernier n'étant pas lu par le serveur SMTP.

4. **Refermez la fenêtre de Python.**

Les messages et leurs sous-types

Plus haut, dans la section « Définir le message », nous avons vu les principaux types MIME utilisés, comme texte ou application. Cependant, si les messages ne devaient reposer que sur ces types, la transmission de messages cohérents à quelqu'un d'autre pourrait parfois être difficile. Le problème, c'est que le type des informations n'est pas suffisamment explicite. Par exemple, si vous envoyez un message qui ne contient que du texte, encore faut-il savoir de quel genre de texte il s'agit avant de pouvoir le traiter, et présupposer tel ou tel comportement par défaut n'est pas une bonne idée. Un message texte pourrait être formé sous la forme de texte brut, mais aussi être en réalité une page HTML. Le type principal n'est donc pas suffisant. Les messages réclament donc un sous-type. Par exemple, si vous voulez envoyer une page HTML, le type est bien textuel, mais le sous-type est `html` (ce qui permet au programme de messagerie du destinataire de savoir ce qu'il doit interpréter). Le type et le sous-type sont séparés par une barre oblique. Si



vous regardez la source du message, vous constaterez alors que la valeur de `Content-Type` indique `test/ html`.

En théorie, le nombre des sous-types n'est pas limité, du moins dès lors que la plate-forme dispose d'une méthode adaptée pour traiter chaque cas. Cependant, la réalité est évidemment plus complexe, puisque l'expéditeur n'est pas censé connaître la configuration de la machine du destinataire (à moins que les deux ne se soient mis d'accord par avance pour utiliser un sous-type spécifique). Pour en apprendre plus à ce sujet, vous pouvez par exemple consulter l'adresse <http://www.freeformatter.com/mime-types-list.html>.

Créer un e-mail

Jusqu'ici, vous avez vu comment fonctionnent l'enveloppe et le message. Il est temps maintenant de joindre les deux bouts pour voir comment tout cela se déroule. Les sections qui suivent vous proposent de créer deux messages, le premier sous la forme de texte brut, et le second formaté en HTML. Aucune fioriture particulière n'étant proposée ici, ces deux styles de messages devraient être universellement acceptés.

Travailler avec un message texte

Les messages composés uniquement de texte représentent la méthode la plus efficace et moins gourmande en ressources pour envoyer des informations. Mais, évidemment, ce sont aussi eux qui contiennent le moins d'informations. Vous pouvez

utiliser des émoticônes pour souligner certaines choses, mais l'absence de formatage pose parfois problème. Les étapes qui suivent vous montrent comment créer un message textuel simple en utilisant Python. Vous retrouverez également cet exemple dans le fichier téléchargeable `TextMessage.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from email.mime.text import MIMEText
import smtplib

msg = MIMEText("Bonjour le monde !")

msg['Subject'] = 'Message de test'
msg['From'] = 'SenderAddress'
msg['To'] = 'RecipientAddress'

s = smtplib.SMTP('localhost')
s.sendmail('SenderAddress',
           ['RecipientAddress'],
           msg.as_string())

print("Message envoyé !")
```

Cet exemple combine tout ce que vous avez vu jusqu'ici dans ce chapitre. Mais c'est la première fois que tout est mis ensemble. Remarquez que, comme dans tout courrier qui se respecte, vous créez d'abord le message et vous remplissez ensuite l'enveloppe.

3. **Choisissez la commande Run Module dans le menu Run.**

L'application vous dit que le message a été envoyé.

**Vous et votre serveur
SMTP**

Si vous avez essayé l'exemple de ce chapitre sans le modifier, vous vous demandez vraisemblablement ce qui ne va pas. Il est peu probable que votre système ait un serveur SMTP connecté à l'emplacement localhost. La raison pour laquelle celui-ci est utilisé dans les exemples est de simplement fournir une sorte de conteneur destiné à être remplacé par des informations liées à votre propre système.

Pour que ces exemples fonctionnent réellement, vous avez besoin d'un serveur SMTP ainsi que d'un compte de messagerie issu du monde réel. Bien entendu, vous pourriez installer tout le logiciel nécessaire pour créer un tel environnement sur votre système (jetez par exemple un coup d'œil du côté de Sendmail, un produit libre accessible depuis [l'adresse http://www.sendmail.com/sm/open_source/download](http://www.sendmail.com/sm/open_source/download)

Cela étant dit, votre système d'exploitation est livré avec sa propre messagerie, et la manière la plus simple de voir comment ces exemples fonctionnent consiste à utiliser le même serveur SMTP que celui dont vous vous servez pour envoyer vos e-mails. Pour connaître l'adresse du serveur SMTP, voyez dans les paramètres de votre compte de messagerie. Fournissez alors cette adresse dans les exemples, ainsi que celle de votre messagerie, et bien entendu une adresse pour le destinataire (qui peut être celle d'un autre compte que vous possédez). Sinon, les exemples ne marcheront pas.

Travailler avec un message HTML

Il s'agit typiquement d'un message texte comportant un formatage particulier. Les étapes qui suivent vous montrent comment créer un message HTML en utilisant Python. Vous retrouverez également cet exemple dans le fichier téléchargeable `HTMLMessage.py`.

1. **Ouvrez une fenêtre de fichier Python.**

Vous pouvez par exemple lancer Python en mode Shell, puis choisir dans le menu File la commande New File.

2. **Tapez le code suivant en appuyant sur Entrée à la fin de chaque ligne :**

```
from email.mime.text import MIMEText
import smtplib

msg = MIMEText(

    "<h1>Titre</h1><p>Bonjour le monde !</p>", "html")

msg['Subject'] = 'Test de message HTML'
msg['From'] = 'SenderAddress'
msg['To'] = 'RecipientAddress'

s = smtplib.SMTP('localhost')
s.sendmail('SenderAddress',
           ['RecipientAddress'],
           msg.as_string())

print("Message envoyé !")
```

Le corps du message est le même que dans l'exemple précédent, si ce n'est qu'il est maintenant formaté à l'aide de balises HTML. Il ne s'agit pas ici d'une page complète, mais uniquement d'un titre de niveau H1 et d'un paragraphe.



Le plus important ici est l'argument `html` qui change le sous-type de `text/plain` en `text/html`. De cette manière, le système de destination saura comment traiter ce message. Si vous oubliez cet argument, votre correspondant verra tout le code HTML, ce qui ne l'aidera pas forcément à comprendre

ce que vous voulez lui dire.

3. **Choisissez la commande Run Module dans le menu Run.**

L'application vous dit que le message a été envoyé.

Consulter ses messages

À ce stade, vous devriez avoir deux ou trois messages en attente, selon la manière dont vous avez suivi les exemples de ce chapitre. Pour les voir, votre application de messagerie doit recevoir ces e-mails du serveur. Supposons la question réglée.

Si votre application de messagerie offre la possibilité de consulter la source des messages, vous constaterez que vous y retrouvez bien les informations qui ont été vues plus haut dans ce chapitre. Rien n'est différent, car le message n'est pas modifié au cours de son voyage.



L'intérêt de créer votre propre application pour envoyer et recevoir des e-mails n'est pas le fait que cela soit intéressant ou pratique, mais d'apporter une grande souplesse. Comme vous avez pu le constater dans cette courte introduction à ce sujet, vous pouvez contrôler chaque aspect des messages lorsque vous créez votre propre application. Python cache tous les détails techniques, ce qui fait que vous pouvez vous concentrer sur l'essentiel, c'est-à-dire produire et transmettre des messages en utilisant les arguments corrects.

Cinquième partie

Les Dix Commandements

Dans cette partie...

- ▶ Découvrir des ressources qui vous permettront d'améliorer votre expérience de programmation en Python.
- ▶ Devenir un professionnel grâce à Python.
- ▶ Trouver les outils dont vous avez besoin pour travailler plus efficacement avec Python.
- ▶ Étendre les pouvoirs de Python en lui ajoutant des bibliothèques.

Chapitre 17

Dix ressources de programmation à découvrir

Dans ce chapitre :

- ▶ Utiliser la documentation de Python.
- ▶ Accéder à un tutoriel Python interactif.
- ▶ Créer des applications en ligne avec Python.
- ▶ Étendre Python avec des bibliothèques tierces.
- ▶ Obtenir un meilleur éditeur pour Python.
- ▶ Contrôler la syntaxe de votre application Python.
- ▶ Travailler avec XML.
- ▶ Devenir un codeur professionnel avec un minimum d'efforts.
- ▶ Vaincre l'obstacle Unicode.
- ▶ Créer des applications rapides.

Ce livre est un bon point de départ pour votre expérience de programmeur en Python, mais vous vous voudrez évidemment disposer de ressources additionnelles une fois arrivé à un certain stade. Ce chapitre vous propose donc dix excellentes ressources dont vous pourrez vous servir pour améliorer votre

expérience de développeur. Elles vous permettront de gagner du temps et de l'énergie lorsque vous créerez votre prochaine et éblouissante application.



Bien entendu, ce chapitre ne constitue que le début de votre expérience en ce qui concerne les ressources Python. Vous pouvez trouver en ligne des librairies entières de documentation Python, ainsi que des montagnes de code tout fait. Il serait même possible d'écrire plusieurs livres rien que sur les bibliothèques Python. Le but est ici de vous donner des idées pour alimenter vos propres recherches. Il n'est pas la fin de l'histoire, juste son début.

Travailler avec la documentation de Python en ligne

Une part essentielle du travail avec Python consiste à savoir ce qui est disponible dans le langage de base, et comment l'étendre pour effectuer d'autres tâches. La documentation de Python (<https://docs.python.org/3/>) contient bien plus qu'une simple référence au langage, telle que vous en disposez lorsque vous installez Python sur votre système. En fait, la documentation en ligne traite de nombreux sujets :

- Les nouvelles fonctionnalités de la dernière version en date.
- Accéder à un tutoriel avancé.
- Référence complète du langage et des bibliothèques.
- Comment installer et configurer Python.
- Comment effectuer des tâches spécifiques

avec Python.

- ✓ Comment installer des modules Python provenant d'autres sources (en tant que moyen d'étendre Python).
- ✓ Comment distribuer les modules Python que vous créez vous-même.
- ✓ Comment étendre Python en utilisant `C/C++` et comment embarquer les nouvelles fonctionnalités que vous créez.
- ✓ Instructions de référence pour les développeurs en `C/C++` qui veulent étendre leurs applications en utilisant Python.
- ✓ Pages de questions fréquemment posées (FAQ).



Toutes ces informations sont fournies sous une forme facilement accessible et compréhensible (à condition bien entendu d'avoir une bonne maîtrise de la langue anglaise). En plus des tables des matières habituelles, vous disposez de multiples index pour rechercher les informations dont vous avez besoin. Par exemple, si vous voulez localiser un module particulier, vous pouvez le faire en cliquant sur le lien `modules`, en haut et droite de la page Web.

C'est aussi depuis la page principale de la documentation en ligne que vous pouvez signaler des problèmes avec Python. Malgré toutes ses qualités, Python ne peut pas, comme tout autre langage d'ailleurs, être totalement exempt de bogues. Localiser et corriger ceux-ci permet d'améliorer Python grâce à la participation de la communauté de ses utilisateurs.

Utiliser le tutoriel LearnPython.org

Il existe de nombreux tutoriels dédiés à Python, et de nombreux aussi qui font du bon travail. Mais ils manquent tous d'une fonctionnalité spéciale que vous trouvez si vous utilisez le tutoriel LearnPython.org (<http://www.learnpython.org/>) : l'interactivité. Au lieu de vous contenter de lire ce qui concerne tel ou tel aspect de Python, cette interactivité vous permet en plus de tester les choses par vous-même.

Si l'on considère que ce livre représente un tutoriel d'initiation, LearnPython.org offre en plus un tutoriel de niveau avancé. Il traite de nombreux sujets, tels que :

- **Générateurs** : Des fonctions spécialisées qui retournent des itérateurs.
- **Compréhensions de listes** : Une méthode qui génère de nouvelles listes à partir de listes existantes.
- **Arguments de fonction multiples** : Une extension des méthodes décrites dans la section « Utiliser des méthodes avec des listes d'arguments variables », dans le Chapitre 14.
- **Expressions régulières** : Des techniques permettant de détecter des motifs particuliers de caractères, comme des numéros de téléphone.
- **Gestion des exceptions** : Une extension des méthodes décrites dans le Chapitre 9.
- **Jeux** : Concerne un type spécial de listes (ou *sets*) qui ne contiennent jamais d'entrées dupliquées.
- **Sérialisation** : Montre comment utiliser une méthodologie de stockage de données appelée JavaScript Object Notation (JSON).

- ✓ **Fonctions partielles** : Une technique servant à créer des versions spécialisées de fonctions simples dérivant de fonctions plus complexes. Par exemple, si vous avez une fonction `multiply()` qui demande deux arguments, une fonction partielle `double()` pourrait ne nécessiter qu'un seul argument qui serait toujours multiplié par deux.
- ✓ **Introspection du code** : Fournit la possibilité d'examiner des classes, des fonctions et des mots-clés afin de déterminer leur objet et leurs fonctionnalités.
- ✓ **Décorateurs** : Une méthode permettant d'appliquer des modifications simples à des objets appelables.

Programmer pour le Web avec Python

Ce livre propose des exemples dits de bureau pour des raisons de simplicité. Cependant, de nombreux développeurs sont spécialisés dans la création d'applications en ligne de diverses sortes avec Python. Le site Web Programming in Python (<https://wiki.python.org/moin/WebProgramming>) vous aide à franchir le pas. Il ne traite pas simplement d'un type d'application en ligne, mais de pratiquement tous les sujets qui concernent ce domaine d'application. Vous y trouvez trois rubriques principales associées à diverses sous-rubriques :

- ✓ Serveur :
 - Développement d'application côté serveur
 - Création de scripts CGI (Common Gateway Interface)

- Solutions serveur
- Développement de systèmes de gestion de contenus (ou CMS)
- Développement de services Web

➤ Client :

- Interfaçage avec des navigateurs existants et des technologies correspondantes
- Création de clients Web spécialisés
- Accession aux données selon diverses technologies, y compris des services Web

➤ Autres :

- Création de solutions communes pour la programmation Web en Python
- Interaction avec des systèmes de gestion de bases de données (SGBD)
- Conception de maquettes d'applications Web
- Conception de solutions Intranet

Obtenir des bibliothèques supplémentaires

Le site Pythonware (<http://www.pythonware.com/>) semble à première vue extrêmement limité jusqu'à ce que vous commenciez à cliquer sur les liens. Il vous donne alors accès à de nombreuses bibliothèques tierces qui peuvent vous aider à effectuer nombre de tâches supplémentaires avec Python. Même si tous ces liens vous proposent des ressources utiles, vous devriez commencer par le premier, Downloads (<http://effbot.org/downloads>). Ce site de téléchargement présente des tas d'outils

intéressants :

- **aggdraw** : Bibliothèque qui vous aide à créer des dessins sans effet d'escalier (anti-aliasing).
- **celementtree** : Module complémentaire à la bibliothèque `elementtree` permettant de travailler plus rapidement et plus efficacement avec des données XML.
- **console** : Interface pour Windows permettant de créer de meilleures applications en mode console.
- **effbot** : Collection de compléments et d'utilitaires, dont le lecteur de flux RSS EffNews.
- **elementsoap** : Bibliothèque qui vous aide à créer des connexions SOAP (Simple Object Access Protocol) vers des fournisseurs de services Web.
- **elementtidy** : Complément à la bibliothèque `elementtree` qui vous aide à créer des arborescences XML mieux présentées et plus fonctionnelles que celles produites avec les outils standard de Python.
- **elementtree** : Bibliothèque qui vous permet d'interagir avec des données XML plus efficacement qu'avec les outils standard de Python.
- **exemaker** : Utilitaire permettant de créer un exécutable à partir d'un script Python de manière à pouvoir l'exécuter comme toute autre application sur votre machine.
- **ftpparse** : Bibliothèque pour travailler avec les sites FTP.
- **grabscreen** : Bibliothèque pour effectuer des captures d'écran.
- **imaging** : Fournit la distribution source pour la bibliothèque PIL (Python Imaging Library) qui permet d'ajouter des fonctions de traitement

d'image à l'interpréteur Python. Disposer de cette source peut servir à personnaliser PIL en fonction de besoins spécifiques.

✓ **pil** : Installateurs binaires pour PIL, ce qui en facilite l'installation. (Notez qu'il existe d'autres compléments à PIL, comme pilfonts ou encore pilwmf).

✓ **pythondoc** : Utilitaire servant à créer une documentation à partir des commentaires de votre code Python, à la manière de JavaDoc.

✓ **squeeze** : Utilitaire servant à convertir une application Python comportant de multiples fichiers en une distribution ne contenant qu'un ou deux fichiers et qui s'exécutera normalement avec l'interpréteur Python.

✓ **tkinter3000** : Une bibliothèque de widgets Python contenant nombre de sous-produits. Les *widgets* sont essentiellement des morceaux de code qui créent des contrôles, comme des boutons, destinés à des applications disposant d'une interface utilisateur graphique. Il existe également de nombreuses extensions à tkinter3000, comme wckgraph qui sert à ajouter le support de graphes à une application.

Créer des applications plus rapidement avec un environnement de développement interactif

Un environnement de développement interactif (ou IDE) vous aide à créer des applications dans un langage spécifique. L'éditeur livré avec Python est tout à fait suffisant pour les exemples de ce livre, mais il est plutôt limité lorsqu'il est temps d'aller plus loin. Par exemple, IDLE ne fournit pas de fonctions de

débogage avancées. De plus, il n'est pas adapté si vous voulez créer des applications graphiques.

Vous pouvez discuter avec cinquante développeurs, et vous trouverez presque autant d'avis différents dès qu'il s'agit d'environnement de développement interactif. Chacun a son produit préféré, et n'a pas vraiment envie d'en essayer un autre. Les développeurs passent de nombreuses heures à apprendre et à maîtriser tel ou tel IDE, ainsi qu'à l'étendre pour l'adapter à leurs besoins spécifiques (du moins, si cela est possible).



La difficulté de changer plus tard d'IDE explique pourquoi il est important d'en essayer plusieurs avant de faire un choix définitif. D'autre part, l'une des raisons majeures qui expliquent cela est aussi le fait que les types de projets risquent d'être incompatibles, ce qui signifie que vous devriez recréer les vôtres chaque fois que vous changez d'éditeur. Mais ce n'est bien sûr pas la seule. La page PythonEditors (<https://wiki.python.org/moin/PythonEditors>) vous propose une impressionnante liste d'environnements de développement Python dans laquelle vous pouvez choisir ceux que vous souhaiteriez tester. Ce tableau décrit les particularités de chaque éditeur de manière à ce que vous puissiez éliminer immédiatement certains choix.

Vérifier votre syntaxe avec plus de facilité

L'éditeur IDLE fournit un certain niveau de mise en évidence de la syntaxe, ce qui est utile pour localiser

certaines erreurs. Par exemple, si vous faites une faute de frappe dans un mot-clé, il n'apparaîtra pas dans la couleur correspondante. Ceci vous permet de repérer immédiatement ce genre de faute, et donc d'effectuer tout de suite la correction sans avoir à lancer l'application pour retrouver l'élément fautif (parfois après plusieurs heures de débogage).

L'utilitaire `python.vim` (http://www.vim.org/scripts/script.php?script_id=790) fournit une mise en évidence améliorée de la syntaxe de Python qui vous permet de trouver les erreurs dans vos scripts encore plus facilement.

Cet utilitaire est en lui-même un script, ce qui lui permet d'agir rapidement et efficacement sur n'importe quelle plate-forme. De plus, vous pouvez adapter son comportement à vos besoins particuliers.

Utiliser XML à votre avantage

Le langage XML (eXtensible Markup Language) est utilisé pour l'enregistrement de données de tous types dans la plupart des applications modernes ayant une certaine substance. Vous avez probablement quantité de fichiers XML stockés sur votre système sans même le savoir, car les données XML peuvent apparaître sous des extensions variées. Par exemple de nombreux fichiers `.config`, qui servent à définir des paramètres par défaut d'applications, s'appuient sur le langage XML. En résumé, la question n'est pas de savoir si vous allez rencontrer des données XML en écrivant des applications Python, mais quand.

XML offre de nombreux avantages par rapport à

d'autres modes d'enregistrement des données. Par exemple, il est indépendant de la plate-forme. Vous pouvez utiliser du XML sur n'importe quel système, et le même fichier sera lisible partout, du moins dès lors que ce système est capable de reconnaître ce format. Cette indépendance de XML explique pourquoi il est associé à tant d'autres technologies, comme des services Web. De plus, XML est relativement facile à apprendre. Et, comme il s'agit de texte, vous pouvez généralement réparer les problèmes sans trop de difficultés.



Il est important d'apprendre XML lui-même. Vous pouvez à cet effet trouver un bon tutoriel comme celui proposé par le site W3Schools (<http://www.w3schools.com/xml/default.ASP>). Certains développeurs s'aperçoivent qu'ils n'arrivent pas à comprendre certains matériaux Python spécifiques qui supposent de savoir écrire des fichiers XML basiques. Ce qui est bien avec le site de W3Schools, c'est qu'il partage l'apprentissage en chapitres qui permettent de progresser pas à pas, notamment pour :

- Découvrir les bases de XML
- Valider vos fichiers XML
- Utiliser XML avec JavaScript (celui-ci étant prééminent dans de nombreux scénarii d'applications en ligne)
- Découvrir les technologies associées à XML
- Utiliser des techniques XML avancées
- Travailler avec des exemples qui permettent de voir XML en action

W3Schools au cœur des technologies de l'informatique

L'une des ressources les plus utilisées pour l'apprentissage en ligne des technologies de l'informatique. La page principale du site se trouve à l'adresse <http://www.w3schools.com/>. Partant de là, vous pouvez explorer chacune des technologies Web permettant de construire toutes les applications modernes que vous pouvez imaginer. Le site couvre en effet de multiples langages :

- ✓ HTML
- ✓ CSS
- ✓ JavaScript
- ✓ SQL
- ✓ JQuery
- ✓ PHP
- ✓ XML
- ✓ ASP.NET

Cependant, vous devriez réaliser qu'il s'agit juste d'un point de départ pour les développeurs Python. Servez-vous du site W3Schools pour vous former aux technologies sous-jacentes, puis appuyez-vous sur les ressources spécifiques à Python pour parfaire vos connaissances. La plupart des développeurs Python ont besoin d'une combinaison de différents matériaux pour atteindre leurs objectifs et faire une réelle différence dans le codage de leurs

applications.

Une fois que vous avez acquis les connaissances de base, vous avez besoin d'une ressource qui vous montre comment utiliser XML avec Python. Voyez notamment à ce sujet le site <http://pyxml.sourceforge.net/topics/>. Grâce à ces deux ressources, votre expertise en XML et ses rapports avec Python devrait rapidement se développer.

Éviter les erreurs courantes des débutants

Absolument tout le monde fait des erreurs de codage, même le *geek* du bas qui programme depuis trente ans (il a commencé à la maternelle). Bien sûr, personne n'aime faire des erreurs, et certains refusent même de l'avouer, mais tout le monde en commet. Il ne faut donc pas en avoir honte. Une erreur se répare, et la vie continue.



Bien entendu, il y a une différence entre commettre une erreur, et commettre une erreur courante, évitable. Oui, même des professionnels font parfois des erreurs courantes, mais c'est moins probable parce qu'ils ont déjà rencontré par le passé ce genre de situation et qu'ils sont entraînés à les éviter. Vous pouvez prendre avantage sur la concurrence en évitant ces erreurs auxquelles n'échappent pas les débutants. Voyez notamment pour cela les deux ressources suivantes :

➤ Python : Common Newbie Mistakes, Part 1

(<http://blog.amir.rachum.com/blog/2013/07/06/pyt-common-newbie-mistakes-part-1/>)

✓ Python : Common Newbie Mistakes, Part 2
(<http://blog.amir.rachum.com/blog/2013/07/09/pyt-common-newbie-mistakes-part-2/>)

Il existe de nombreuses autres ressources à disposition des débutants, mais celles-ci sont succinctes et faciles à comprendre. Vous pouvez les lire en assez peu de temps, prendre quelques notes pour pouvoir y revenir plus tard, et éviter ces erreurs embarrassantes qui jalonnent l'existence des apprentis développeurs.

Comprendre Unicode

Bien que ce livre essaie de vous éviter cette ornière, vous ne pourrez peut-être pas l'éviter si vous écrivez des applications sérieuses (et internationalisées). Malheureusement, Unicode fait partie de ces sujets dont un haut comité a *autrefois* décidé de tout, ce qui fait que l'on finit par avoir des définitions approximatives et une multitude de standards qui prétendent le définir. En bref, il n'existe aucune définition de ce qu'est Unicode.

Vous risquez donc de rencontrer des standards Unicode variés dès lors que vous commencerez à travailler avec des applications Python plus avancées, et notamment avec des langues diverses et variées (dont il semble que chacune possède sa propre version d'Unicode). Voici quelques ressources que vous pourriez consulter avant de vous lancer dans ce type de projet :

- ✓ The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses !)
([http://www.joelonsoftware.com/articles/Unicode.h](http://www.joelonsoftware.com/articles/Unicode.html))
- ✓ The Updated Guide to Unicode on Python
(<http://lucumr.pocoo.org/2013/7/2/the-updated-guide-to-unicode/>)
- ✓ Python Encodings and Unicode
(<http://eric.themoritzfamily.com/python-encodings-and-unicode.html>)
- ✓ Unicode Tutorials and Overviews
(<http://www.unicode.org/standard/tutorial-info.html>)
- ✓ Explain it like I'm five : Python and Unicode ?
(<https://www.reddit.com/r/Python/comments/1g62/>)
- ✓ Unicode Pain
(<http://nedbatchelder.com/text/unipain.html>)

Rendre vos applications Python plus rapides

Rien n'ennuie plus un utilisateur qu'une application lente. Dans ce cas, vous allez pouvoir compter vos utilisateurs sur les doigts d'une main. La lenteur est une cause majeure du rejet d'applications dans les milieux professionnels. Une organisation peut dépenser beaucoup d'argent pour bâtir une application qui impressionne et qui fait tout, mais personne ne l'utilisera si elle rame trop, ou si elle rencontre des problèmes de performance.



La notion de performance est un compromis entre fiabilité, sécurité et vitesse. Trop de développeurs se

focalisent sur la rapidité sans pour autant atteindre leur objectif. Il est important de vérifier chaque partie de votre application quant à son utilisation des ressources pour vous assurer que vous utilisez les meilleures techniques de codage.

Il existe de nombreuses ressources qui peuvent vous aider à mieux comprendre cette question de performances lorsqu'il s'agit d'applications Python. L'une des meilleures est sans doute le site A guide to analyzing Python performance (<http://www.huynh.com/posts/python-performance-analysis/>). L'auteur prend le temps d'expliquer pourquoi quelque chose forme un goulot d'étranglement, plutôt que de dire simplement qu'il en est un. Une fois que vous aurez consulté cet article, passez au niveau suivant : PythonSpeed Performance Tips (<https://wiki.python.org/moin/PythonSpeed/PerformanceTips>)

Chapitre 18

Dix domaines où faire fortune avec Python

Dans ce chapitre :

- ▶ Utiliser Python dans l'assurance qualité.
- ▶ Tracer votre chemin dans une petite organisation.
- ▶ Employer Python pour des besoins de scripts spécifiques.
- ▶ Travailler en tant qu'administrateur.
- ▶ Démontrer des techniques de programmation.
- ▶ Fouiller dans les données de localisation.
- ▶ Explorer des données de toutes sortes.
- ▶ Travailler avec des systèmes embarqués.
- ▶ Traiter des données scientifiques.
- ▶ Analyser des données en temps réel.

Bien entendu, le titre de ce chapitre est un peu provocateur, et ne vous en prenez pas à moi si vous n'arrivez pas à faire réellement fortune grâce à Python. Cela étant, vous pouvez littéralement écrire n'importe quelle application dans n'importe quel langage de programmation si vous y consacrez suffisamment de temps, de patience et d'efforts. Dans

certains cas, ce temps, cette patience et ces efforts peuvent être considérables. En résumé, tout ou presque est possible. Mais tout n'en vaut pas la peine. Utiliser le bon outil pour un travail est toujours un plus dans un monde où la rapidité fait loi, et où prendre son temps devient une forme de gaspillage.

Python excelle dans certains types de tâches, ce qui implique qu'il convient mieux à certains types de programmes. La nature des programmes que vous pouvez réaliser avec Python détermine donc votre possible orientation professionnelle et la manière de réussir dans vos choix. Par exemple, Python n'est pas un bon choix pour écrire des pilotes de périphériques, comme C/C++, et il est donc peu probable que votre carrière avec Python puisse s'orienter vers ce type d'industrie. De même, Python est capable de travailler avec des bases de données, mais pas au même niveau de profondeur que d'autres langages spécifiquement conçus pour cela, comme SQL. Par contre, la formation est une voie très intéressante, car Python est un grand langage pour se former, et donc former les autres, à la programmation.

Les sections qui suivent décrivent certaines des activités qui utilisent Python de manière régulière de façon à ce que vous puissiez vous faire une idée du genre de projets que vous aimeriez conduire grâce à vos nouvelles connaissances. Bien entendu, ce petit tour d'horizon ne couvre pas tous les champs possibles.

Utiliser Python dans l'assurance qualité

De nombreuses organisations possèdent un

département chargé de l'assurance qualité, c'est-à-dire de la vérification des applications pour s'assurer que celles-ci fonctionnent parfaitement (du moins autant qu'il est possible). Il existe sur le marché de nombreux langages de script dédiés à cette activité, mais Python est de ce point de vue un excellent outil du fait de sa souplesse incroyable. De plus, vous pouvez l'utiliser dans des environnements multiples, et donc autant côté client que côté serveur. Cela signifie que vous pouvez apprendre un seul langage, et l'utiliser pour effectuer des tests partout où cela est nécessaire, et quel que soit l'environnement concerné.



Dans ce scénario, le développeur connaît habituellement un autre langage, tel C++, et il se sert de Python pour tester une application écrite en C++. Cependant, la personne en charge de l'assurance qualité n'a pas besoin d'un tel niveau de connaissances dans tous les cas. Dans certaines situations, un test écrit en Python peut suffire à confirmer qu'une application se comporte comme prévu, ou pour contrôler les fonctionnalités d'un fournisseur de services extérieur. Ce sont là des points à préciser avec l'organisation dans laquelle vous souhaiteriez travailler, de même que le niveau de qualification requis en fonction des besoins de cette organisation.

**De l'utilité de connaître
plusieurs langages de
programmation**

Pour la plupart des organisations, connaître de multiples langages de programmation est un grand plus (et même parfois une obligation). Bien entendu, si vous êtes employeur, il est préférable de rechercher un maximum de compétences si vous devez embaucher quelqu'un. Connaître plusieurs langages de programmation signifie que vous pouvez travailler sur plusieurs postes et offrir une plus grande valeur à l'organisation. Réécrire une application dans un autre langage demande du temps, c'est aussi une source d'erreurs, et de surcroît cela demande un investissement qui peut être important. C'est pourquoi la plupart des entreprises recherchent des gens capables d'assurer la maintenance d'une application dans le langage avec lequel elle a été écrite, plutôt qu'avoir à la réécrire.

De votre point de vue, connaître plusieurs langages signifie que vous pouvez prétendre à un emploi plus intéressant, et que vous n'aurez pas à refaire toujours la même chose jour après jour. De plus, cela tend à réduire les risques de frustration. De nos jours, la plupart des grandes organisations se servent de composants informatiques écrits dans divers langages. Pour comprendre une application, et comment elle fonctionne, vous devez donc connaître chacun des langages utilisés pour la construire.

Une autre conséquence, c'est que cela permet d'apprendre plus vite de nouveaux langages. Au bout de quelque temps, vous commencez à mieux voir comment les langages de programmation peuvent en quelque sorte

cohabiter, vous passez moins de temps à en apprendre les bases, et vous êtes en capacité de vous attaquer plus rapidement à des sujets avancés. Plus vite vous pouvez apprendre de nouvelles technologies, et meilleures seront vos opportunités de trouver un emploi intéressant et bien rémunéré. En résumé, connaître plusieurs langages de programmation ouvre des tas de portes.

Devenir responsable informatique dans une petite organisation

Dans une petite organisation, il est courant que le service informatique n'ait de service que le nom, et qu'il se réduise en fait à une ou deux personnes. Cela signifie que vous devez être capable d'effectuer une large variété de tâches rapidement et efficacement. Avec Python, vous pouvez écrire des utilitaires et des applications maison assez vite. Même si Python peut ne pas répondre aux besoins d'une organisation importante du fait qu'il s'agit d'un langage interprété (et donc potentiellement plus sensible aux intrusions ou à une mauvaise utilisation), son emploi dans une petite organisation fait sens, car vous disposez d'un contrôle accru et vous pouvez apporter plus rapidement les changements et les évolutions nécessaires. De plus, la possibilité d'utiliser Python dans des environnements variés réduit le besoin de se servir d'un autre outil pour répondre à vos besoins.



Même si cela n'est pas forcément évident au premier abord, Python peut également être un outil tout à fait

utilisable dans des situations bien spécifiques. Par exemple, vous ne pouvez pas directement vous servir de scripts Python avec IIS (Internet Information Server). Par contre, il est possible d'y ajouter un support de script Python à ce produit en utilisant les étapes trouvées dans la base de connaissances de Microsoft à l'adresse <https://support.microsoft.com/fr-fr/kb/276494>. Si vous n'êtes pas sûr qu'une certaine application soit capable d'utiliser des scripts écrits avec Python, effectuez une recherche en ligne pour trouver les informations dont vous avez besoin.

Applications et scripts Python

Nombre de produits peuvent utiliser Python pour créer des scripts. C'est ce que fait par exemple Maya (<http://www.autodesk.com/products/maya/overview>). En sachant quels produits évolués supportent Python, vous serez mieux à même de vous rapprocher de sociétés qui les utilisent. Voici quelques exemples de produits qui s'appuient sur Python pour la création de scripts :

- ✓ 3ds Max
- ✓ Abaqus
- ✓ Blender
- ✓ Cinema 4D
- ✓ GIMP
- ✓ Google App Engine
- ✓ Houdini
- ✓ Inkscape
- ✓ Lightwave
- ✓ Modo
- ✓ MotionBuilder
- ✓ Nuke

- ✓ Paint Shop Pro
- ✓ Scribus
- ✓ Softimage

Il s'agit juste de la partie émergée de l'iceberg. Par exemple, certains jeux se servent aussi de Python comme outil de script. Comme vous pouvez le constater, le champ est large !

Administrer un réseau

De nombreux administrateurs se servent de Python pour effectuer diverses tâches, comme surveiller l'état de santé du réseau, ou encore automatiser des procédures. Comme ce sont souvent des gens pressés, tout ce qu'ils savent faire pour accélérer les processus est du temps de gagné. En fait, certains outils de gestion de réseau, tels que Trigger (<http://trigger.readthedocs.org/en/latest/>) sont écrits en Python. Nombre de ces outils sont libres de droits (*open source*) et en chargement libre, ce qui vous permet de les essayer facilement sur votre réseau.

L'essentiel est de comprendre que connaître Python peut réduire considérablement votre charge de travail et vous aider à effectuer vos tâches plus facilement. Si vous voulez voir certains scripts écrits plus spécifiques pour la gestion des réseaux, une liste de vingt-cinq projets est accessible à l'adresse <http://freecode.com/tags/network-management>.

Apprendre aux autres

De nombreux professeurs recherchent une méthode

rapide et efficace pour enseigner les technologies de l'informatique. Ainsi, Raspberry Py (<https://www.raspberrypi.org/>) est un petit système informatique bon marché et incroyablement simple à configurer, et qui peut être connecté à un moniteur ou même un téléviseur pour proposer de nombreuses applications dans le domaine de la formation (et dans d'autres domaines aussi). De plus, Python joue un grand rôle dans l'usage du Raspberry Py en tant que support de formations à la programmation (http://ww38.piprogramming.org/main/?page_id=372).



En réalité, les enseignants utilisent souvent Python pour étendre les fonctionnalités natives du Raspberry Py afin de réaliser toutes sortes d'applications intéressantes (<https://www.raspberrypi.org/blog/tag/python/>). À titre d'indicatif, je vous conseille d'aller voir Boris, le Dino-Bot Twitter(<https://www.raspberrypi.org/blog/boris-the-twitter-dino-bot/>). Très intéressant. En résumé, si vous avez un projet éducatif en tête, combiner le Raspberry Py et Python est une idée fantastique.

Aider les gens à choisir un emplacement

Un Système d'Information Géographique (SIG) fournit un moyen de visualiser des données de localisation dans un but d'aménagement, d'implantation ou encore commercial. Par exemple, un SIG constitue une aide qui peut être précieuse afin de déterminer le meilleur emplacement possible pour une nouvelle activité. Mais un SIG peut avoir bien d'autres applications, qu'il s'agisse de communiquer des informations sur un certain lieu ou pour présenter à

d'autres personnes des localisations physiques. De surcroît, de nombreux produits de SIG utilisent Python comme langage de choix. En fait, il existe de multiples informations spécifiques à Python dans ce domaine. Voyez par exemple :

- ✓ The GIS and Python Software Laboratory (<http://gispython.org/>)
- ✓ Python and GIS Resources (<http://www.gislounge.com/python-and-gis-resources/>)
- ✓ GIS Programming and Automation (<https://www.e-education.psu.edu/geog485/node/17>)

De nombreux produits spécialisés, comme ArcGIS (<http://www.esri.com/software/arcgis>), s'appuient sur Python pour l'automatisation des tâches. Des communautés entières se regroupent autour de ces applications, comme par exemple Python for ArcGIS (<http://resources.arcgis.com/en/communities/python/>). Vous voyez que vos nouvelles connaissances peuvent être employées dans bien d'autres domaines que l'informatique proprement dite.

Explorer des données

Tout le monde collecte des données sur tout et sur tous. Essayer de tamiser ces montagnes de données est une tâche impossible sans outils d'automatisation évolués. La nature flexible de Python, combinée avec son langage concis qui rend les changements extrêmement rapides, en fait un favori pour les personnes qui ont besoin d'explorer des données au quotidien. Vous pouvez voir à ce sujet un joli livre en

ligne intitulé *A Programmer's Guide to Data Mining* (<http://guidetodatamining.com/>).



L'expression « explorer des données » est une des manières de parler de fouille de données, de forage de données, ou encore de prospection de données, bref de *data mining*.

De ce point de vue, Python facilite cette exploration de données. L'un des principaux objectifs de l'exploration de données est la détection de tendances, ce qui signifie rechercher des cohérences ou des trames de multiples sortes. L'utilisation d'intelligence artificielle avec Python rend possible ce type de reconnaissance. Un article sur ce sujet, *Practical Data Mining with Python* (<https://dzone.com/refcardz/data-mining-discovering-and>), vous aidera à comprendre cela de plus près, et peut-être même à créer le bon outil pour développer les ventes de votre employeur (ou vous installer à votre compte grâce à vos découvertes...).



L'exploration de données a bien d'autres champs d'application, y compris localiser de nouvelles planètes autour des étoiles !

Interagir avec des systèmes embarqués

Des systèmes embarqués, il y en a partout et pour tout. Par exemple, si vous utilisez un thermostat programmable, vous interagissez avec un système embarqué. Le Raspberry Py, déjà mentionné dans ce chapitre, est un exemple de système embarqué, bien entendu nettement plus complexe qu'un thermostat.

De nombreux systèmes embarqués reposent sur Python pour leur programmation. En fait, il existe même une forme spéciale de Python, Embedded Python

(<https://wiki.python.org/moin/EmbeddedPython>), dédiée à ce type d'application. Vous pouvez également retrouver sur YouTube une vidéo qui montre l'utilisation de Python pour réaliser un système embarqué (<https://www.youtube.com/watch?v=WZoeqnsY9AY>).



Python est aussi le langage de choix pour divers systèmes de sécurité ou de télécommande dans le champ de l'automobile (<http://www.pythencarsecurity.com/>), ou encore de la domotique (<http://www.linuxjournal.com/article/8513>).

Python est très populaire dans les systèmes embarqués, car il ne nécessite pas de compilation. Il suffit par exemple pour un professionnel de créer une mise à jour de la partie logicielle de son produit, et de télécharger tout simplement le fichier Python. L'interpréteur utilise immédiatement le nouveau fichier sans avoir à en passer par diverses phases de reconfiguration, comme c'est le cas avec de nombreux autres langages.

Travailler avec des données scientifiques

Python est mieux adapté aux traitements de données numériques et scientifiques que la plupart des autres langages de programmation. Quantité de modules dédiés à ces tâches sont d'ailleurs accessibles via

l'adresse

<https://wiki.python.org/moin/NumericAndScientific>.

Les scientifiques aiment Python, car il est léger, facile à apprendre et suffisamment précis dans le traitement de leurs données. Il est possible de produire des résultats en utilisant juste quelques lignes de code. Bien entendu, ces mêmes résultats pourraient aussi être obtenus avec un autre langage de programmation. Mais celui-ci ne disposerait pas forcément de modules préexistants pour effectuer la tâche voulue, et il faudrait écrire bien plus de lignes de code pour atteindre le même but.



Les deux domaines qui disposent de modules Python dédiés sont les sciences de l'espace et celles de la vie. Par exemple, il existe un module servant à effectuer des tâches relatives à la physique solaire. Vous pouvez également trouver un module servant au travail sur le génome. Si vous vous orientez dans un champ scientifique, il y a de bonnes chances pour que vos connaissances en Python aient un impact significatif sur votre capacité à produire des résultats rapidement pendant que vos collègues en seront encore à essayer de comprendre comment analyser les données.

Effectuer des analyses de données en temps réel

De nombreuses décisions réclament des données fiables, précises et en temps réel. Souvent, ces données proviennent de sources multiples, et nécessitent donc un certain degré d'analyse avant de devenir utiles. De nombreuses personnes utilisent

Python pour explorer ces sources d'informations disparates, effectuer les analyses requises, puis présenter le tableau final au décideur qui a commandité ce travail. Traiter manuellement ce genre de tâche prendrait un temps fou. En fait, cela deviendrait très vite du temps perdu, puis la locomotive serait déjà loin avant que vous ne trouviez comment y attacher le wagon... Python permet d'effectuer ce type de tâche suffisamment rapidement pour qu'une décision puisse avoir un impact maximal.

De plus, grâce à Python, les modifications sont faciles à effectuer. Changer quelques lignes de code dans un module interprété suffit généralement pour arriver au but recherché, sans qu'il soit besoin de se préoccuper de choses comme recompiler l'application, et ainsi de suite.



Si ce chapitre vous a présenté quelques champs d'application de Python, savoir aller au-delà et ne pas se limiter soi-même est essentiel. Des tas de gens ont besoin d'analyses en temps réel dans des tas d'activités très différentes. Lancer une fusée dans l'espace, contrôler des flux de production, s'assurer que des colis sont livrés à temps, et des myriades d'autres activités sont liées à une exploitation en temps réel donnée dans ces conditions de précision et de fiabilité qui en assurent l'efficacité. Vous pourriez peut-être créer votre propre emploi simplement en utilisant Python pour effectuer des analyses de données diverses et variées en temps réel.

Chapitre 19

Dix outils intéressants

Dans ce chapitre :

- ▶ Pister les bogues des applications.
- ▶ Créer un endroit sûr pour tester les applications.
- ▶ Installer votre application sur le système d'un utilisateur.
- ▶ Documenter votre application.
- ▶ Écrire le code de votre application.
- ▶ Rechercher des erreurs dans les applications.
- ▶ Travailler dans un environnement interactif.
- ▶ Tester une application.
- ▶ Gérer les instructions `import` dans votre application.
- ▶ Suivre les versions d'une application.

Python, comme la plupart des autres langages de programmation, dispose d'outils divers et variés provenant de programmeurs ou de sociétés tierces. Un *outil* est un utilitaire qui améliore les capacités naturelles de Python. Par exemple, un débogueur est un outil, car il s'agit bien d'un utilitaire qui vient en complément de Python, ce qui n'est pas le but d'une bibliothèque. Les bibliothèques, de leur côté, sont là

pour créer de meilleures applications (nous reviendrons sur certaines d'entre elles dans le Chapitre 20).

Mais cette distinction ne réduit pas pour autant la liste des outils qui viennent renforcer le pouvoir de Python. Celui-ci vous permet en effet d'accéder à une grande quantité d'outils de toutes sortes, qu'ils soient généralistes ou extrêmement spécialisés. Le site dédié

<https://wiki.python.org/moin/DevelopmentToolslesparta>
(cette liste n'étant pas forcément complète) :

- AutomatedRefactoringTools (automatisation)
- BugTracking (détection des bogues)
- ConfigurationAndBuildTools (configuration et construction)
- DistributionUtilities (distribution)
- DocumentationTools (documentation)
- IntegratedDevelopmentEnvironments (environnements de développement intégrés)
- PythonDebuggers (débugueurs)
- PythonEditors (éditeurs)
- PythonShells (shells)
- SkeletonBuilderTools (prototypage de dossiers et de fichiers)
- TestSoftware (tests)
- UsefulModules (utilitaires divers)
- VersionControl (contrôle de version)

Évidemment, un unique chapitre ne peut pas couvrir tous ces sujets. Nous nous limiterons donc à quelques-uns des outils parmi les plus intéressants, c'est-à-dire ceux qui méritent de retenir votre attention. Une fois mis en appétit par ce chapitre, vous devriez rechercher d'autres sortes d'outils en ligne. Vous pourriez bien constater alors que l'outil que vous

pensiez avoir à fabriquer vous-même existe déjà, et même parfois sous différentes formes.

Pister les bogues avec Roundup Issue Tracker

Vous pouvez utiliser de multiples sites de pistage des bogues (donc des erreurs dans les applications), comme Github (<https://github.com/>), Google Code (<https://code.google.com/>), BitBucket (<https://bitbucket.org/>), ou encore Launchpad (<https://launchpad.net/>). Mais ces sites publics ne sont généralement pas aussi pratiques qu'un logiciel spécialisé. Parmi cette gamme d'outils, Roundup Issue Tracker (<http://roundup.sourceforge.net/>) est l'un des tout meilleurs. Il devrait être opérationnel sur n'importe quelle plate-forme supportée par Python, et il offre des fonctionnalités de base ne nécessitant pas de travail supplémentaire de votre part :

- Dépistage des bogues
- Gestion de liste de tâches

Vous pouvez également ajouter d'autres fonctionnalités qui rendent ce produit assez spécial. Cependant, cela nécessite l'installation de compléments supplémentaires, comme un gestionnaire de bases de données (SGBD). Les instructions fournies vous indiquent ce que vous devez installer et ce qui est compatible. Après quoi, vous devez encore effectuer diverses mises à jour. Voyez à ce sujet ce qu'indique la page principale de Round Issue Tracker.

Créer un environnement virtuel avec VirtualEnv

Il y a de multiples raisons qui justifient la création d'environnements virtuels, mais la principale est certainement de fournir un espace sûr et bien connu pour tester vos applications Python. En utilisant à chaque fois le même environnement de test, vous pouvez vous assurer que l'application fonctionne de manière stable, et ce jusqu'à ce qu'elle soit prête à être essayée dans un environnement de production. VirtualEnv (<https://pypi.python.org/pypi/virtualenv>) vous permet de créer un environnement de travail qui vous permet de tester votre application en amont, ou encore diagnostiquer des problèmes qui pourraient se produire dans un certain environnement. Il est important de se rappeler qu'il y a trois niveaux de test que vous devriez effectuer :

- **Bogues** : Ce sont les erreurs que contient votre application.
- **Performance** : Pour valider le comportement de votre application en termes de rapidité, de fiabilité et de sécurité.
- **Commodité** : Il s'agit de vérifier que votre application correspond aux besoins de l'utilisateur et qu'elle réagira de la manière voulue aux saisies de celui-ci.



Du fait même de la manière dont les applications Python sont utilisées (voyez aussi le Chapitre 18 à ce sujet), vous n'avez généralement pas besoin de les exécuter dans un environnement virtuel une fois qu'elles sont en phase de production. La plupart des

applications Python ont besoin d'accéder au monde extérieur, et s'isoler dans un environnement virtuel pourrait les couper de celui-ci. Mais cela vaut uniquement pour les phases de production...

Ne faites jamais de tests sur un serveur de production

Une erreur assez courante consiste à tester une application sur un système destiné en fait à la production. Parmi toutes les raisons qui justifient cette précaution, le risque de perdre des données est sans doute le plus important. Si vous permettez à vos utilisateurs d'accéder à une version non finalisée de votre application, qui est susceptible de contenir encore des bogues capables d'endommager une base de données ou d'autres sources, le résultat risque d'être catastrophique, autant pour vos données que pour le jugement de l'utilisateur sur votre travail.

Vous devez donc également réaliser que vous n'avez qu'une seule chance de faire bonne impression. De nombreux projets logiciels échouent parce que les utilisateurs se détournent avant le résultat final. L'application est terminée, mais personne ne l'utilise du fait d'une mauvaise impression de départ. Les utilisateurs n'ont qu'une idée en tête : faire leur travail et rentrer chez eux. S'ils voient une application dont ils pensent qu'elle leur fait

perdre leur temps, ils vont voir ailleurs.

Des applications non finalisées peuvent aussi comporter des failles de sécurité permettant à des personnes malhonnêtes d'accéder à votre réseau. Si vous laissez la porte ouverte et que n'importe qui peut rentrer chez vous, tout le reste est sans importance. Une fois que les méchants ont franchi le pas de la porte, s'en débarrasser est pratiquement impossible. Et même si vous y arrivez, les dégâts déjà commis peuvent être considérables. Boucher de telles failles est notoirement difficile, parfois impossible. En résumé, ne testez jamais votre application sur un serveur de production, car les coûts induits par cette erreur pourraient être considérables.

Installer votre application avec PyInstaller

Les utilisateurs n'ont aucune envie de passer trop de temps à installer votre application, même si cela pourrait au final leur rendre service. De plus, des utilisateurs manquant d'expérience pourraient même ne pas arriver jusqu'à la fin du processus. Vous devez donc prévoir une méthode d'installation sur le système de l'utilisateur qui soit à l'abri des catastrophes (presque) naturelles. C'est ce que permettent des outils d'installation tels que PyInstaller (<http://www.pyinstaller.org/>). Ils font un joli paquet avec votre application, de manière à ce que vos utilisateurs puissent l'installer facilement.

Fort heureusement, PyInstaller fonctionne sur toutes les plates-formes supportées par Python, ce qui fait que vous n'avez besoin que d'un seul outil pour répondre à tous vos besoins. De plus, vous pouvez aussi obtenir si vous le souhaitez un support spécifique à une certaine plate-forme. Mais, dans la plupart des cas, il vaut mieux s'en tenir à une configuration généraliste, à moins d'en avoir vraiment besoin. Si vous utilisez une méthode d'installation spécifique à une plate-forme, elle ne pourra évidemment s'effectuer que sur celle-ci.



Nombre d'outils d'installation sont dédiés à une certaine plate-forme. C'est par exemple souvent le cas de ceux qui créent un fichier exécutable, ce fichier risquant de n'être accepté que sur un système d'exploitation donné. Il est important d'utiliser un produit capable de fonctionner partout où votre application doit être employée. De cette manière, vous ne risquerez pas de vous retrouver face à des utilisateurs qui seraient incapables de l'installer. À quoi bon disposer d'un langage pratiquement universel si l'outil d'installation est trop spécialisé ?

Éviter les produits orphelins

Certains outils Python que l'on trouve sur l'Internet sont *orphelins*, ce qui signifie que le développeur ne les supporte plus activement. Des programmeurs continuent à utiliser ces outils, tout simplement parce qu'ils aiment leurs fonctionnalités ou la manière dont ils

fonctionnent. Cependant, cela représente toujours un risque, car vous ne pouvez pas savoir si tel outil sera compatible avec la dernière version de Python. La meilleure approche consiste donc à ne faire usage que d'outils qui restent pleinement supportés par leurs développeurs.

Si vous devez absolument utiliser un outil orphelin, peut-être parce que c'est le seul capable d'effectuer une certaine tâche, assurez-vous qu'il dispose d'une communauté active. Même si le développeur est passé à autre chose, cette communauté devrait être capable de fournir des informations dont vous pouvez avoir besoin. Sinon, vous risquez de perdre beaucoup de temps pour n'arriver au final à rien de bien.

Construire une documentation de développement avec pdoc

Il y a deux types de documentations associés aux applications : celle qui est destinée à l'utilisateur, et celle qui concerne le ou les développeurs. La première explique à l'utilisateur comment se servir de l'application. La seconde montre comment l'application marche. Une bibliothèque ne nécessite qu'une seule sorte de documentation, celle qui est sur le versant développeur, tandis qu'une application de bureau n'a besoin que d'une documentation dédiée à l'utilisateur. Un service peut avoir besoin des deux.

Pour tout ce qui concerne la documentation côté

développeur, vous avez à votre disposition une solution simple, pdoc (<https://github.com/BurntSushi/pdoc>). Cet utilitaire s'appuie sur les commentaires que vous placez dans votre code. La sortie produite est au format texte ou HTML. Il est également possible d'exécuter pdoc d'une manière permettant de diriger la sortie via un serveur Web afin que les autres personnes puissent consulter votre documentation dans leur navigateur habituel. Cette solution remplace epydoc, qui n'est plus supporté par son concepteur.

Développer le code de l'application avec Komodo Edit

Plusieurs chapitres ont abordé la question des environnements de développements interactifs (ou IDE), mais sans faire de recommandations particulières. L'environnement que vous pouvez être amené à choisir dépend de vos besoins en tant que développeur, de votre niveau de compétence, et du type d'applications que vous voulez créer. Certains IDE sont en effet meilleurs que d'autres pour tel ou tel type de tâche. L'un des meilleurs outils généralistes pour les développeurs débutants est Komodo Edit (<http://komodoide.com/komodo-edit/>). Ses fonctionnalités de base sont gratuites, et elles sont largement suffisantes pour couvrir vos besoins, en tout cas bien meilleures que celles d'IDLE. En voici quelques exemples :

- Support de multiples langages de programmation.
- Complétion automatique des mots-clés.
- Contrôle de l'indentation.

- Support de projets, ce qui permet de disposer d'un code partiel avant même de débiter.
- Excellent support.

De plus, ce qui fait de Komodo Edit un produit à part est qu'il est évolutif. Si vous trouvez que la version de base de Komodo Edit ne suffit plus à vos besoins, vous pouvez passer à la version Komodo IDE (<http://komodoide.com/>), qui inclut quantité de fonctionnalités supplémentaires de niveau professionnel, comme le profilage du code (une fonction qui teste la rapidité d'exécution de l'application) ou encore un accès simplifié aux bases de données.

Déboguer votre application avec pydbgr

Un éditeur haut de gamme, tel que Komodo IDE, est fourni avec son propre débogueur. Même Komodo Edit, d'ailleurs, possède un tel outil, bien entendu plus simple. Cependant, si vous utilisez quelque chose de plus petit, de moins cher et de moins sophistiqué qu'un environnement de haut niveau, il est fort possible que vous ne disposiez d'aucun débogueur. Un *débogueur* vous aide à localiser les erreurs dans vos applications et à les réparer. Meilleur est cet outil, et moins vous avez à dépenser d'efforts pour localiser et réparer vos erreurs. Si votre éditeur ne contient pas ce dont vous avez besoin, vous pouvez faire appel à un outil externe tel que pydbgr (<https://code.google.com/p/pydbgr/>).



Un débogueur raisonnablement bon contient un bon nombre de fonctionnalités standard, comme la

colorisation du code (l'utilisation d'un système de couleurs pour indiquer des choses comme les mots-clés). Cependant, pydbgr offre aussi certaines fonctions moins standard qui en font un bon choix si votre éditeur est dépourvu de débogueur.

➤ **Smart Eval** : La commande `eval` vous aide à voir ce qui se passe lorsque vous exécutez une ligne de code, avant même qu'elle ne le soit dans l'application. Elle vous aide à effectuer une analyse du genre « que se passe-t-il si... » afin de voir ce qui va mal avec l'application.

➤ **Débogage hors processus** : Normalement, vous devez déboguer des applications qui se trouvent sur la même machine. En fait, le débogueur fait partie du processus de l'application elle-même, ce qui signifie qu'il peut interférer avec l'opération de débogage. Cette fonctionnalité permet de scinder les deux processus, et y compris d'exécuter l'application et le débogueur sur deux machines différentes.

➤ **Inspection du byte code** : Voir comment le code que vous écrivez est transformé en *byte code* (celui que comprend l'interpréteur Python) peut parfois aider à résoudre des problèmes délicats.

➤ **Filtrage et suivi des événements** : Lorsque votre application passe dans la moulinette du débogueur, elle génère des événements qui aident ce dernier à comprendre ce qu'il se passe. Par exemple, passer à la ligne de code suivante provoque un événement, le retour d'un appel de fonction génère un autre événement, et ainsi de suite. Cette fonction permet donc d'assurer le suivi de ces événements et la manière dont le débogueur y réagit.

Entrer dans un environnement interactif avec IPython

Le mode Shell de Python est suffisant pour la plupart des tâches interactives. Vous l'avez utilisé de façon intensive dans ce livre. Cependant, vous avez évidemment constaté qu'il a certaines insuffisances. Bien entendu, la première d'entre elles est que cet environnement est purement textuel, et que vous devez taper les commandes l'une après l'autre pour exécuter une certaine tâche. Un outil plus avancé, comme IPython (<http://ipython.org/>) peut rendre l'environnement interactif plus « amical » en fournissant une interface améliorée qui vous évite d'avoir à vous rappeler de tous les noms de commandes.



IPython est en fait plus qu'un simple « shell ». Il fournit un environnement dans lequel vous pouvez interagir avec Python de multiples manières, par exemple en affichant des graphiques qui montrent le résultat des formules que vous créez avec Python. De plus, IPython est conçu pour servir d'interface capable de s'accommoder avec d'autres langages.

L'une des fonctionnalités les plus excitantes de IPython est sa capacité à travailler dans un environnement en mode parallèle (autrement dit avec plusieurs « fils », ou threads, de traitement simultanés).

Tester les applications Python avec PyUnit

Arrivé à un certain stade, vous devez tester votre application pour vous assurer qu'elle fonctionne comme prévu. Vous pouvez effectuer un test en tapant une commande à la fois pour en vérifier le résultat, ou bien automatiser ce processus. Évidemment, la seconde approche est meilleure, car vous avez vraiment envie de rentrer manger un jour, et les tests manuels sont extrêmement lents (surtout si vous faites des erreurs, ce qui est pratiquement inévitable). Des produits tels que PyUnit (<https://wiki.python.org/moin/PyUnit>) rendent ce processus nettement plus facile.

Le bon point de ce produit est que vous pouvez en fait écrire du code Python pour effectuer les tests. Votre script est simplement une autre application spécialisée qui recherche les problèmes dans l'application principale.



Ne croyez pas qu'un script soit forcément exempt de bogues. Ces scripts sont conçus pour être extrêmement simples, ce qui limite bien entendu les risques. Mais ceux-ci restent possibles. Si vous n'arrivez à comprendre quel est le problème dans votre application, vous devrez donc vérifier le script de test.

Améliorer votre code avec Isort

Cela peut sembler presque futile, mais le code est parfois un méli-mélo de lignes, surtout si vous n'avez pas placé toutes vos instructions `import` au début du fichier, et ce dans l'ordre alphabétique. Dans certains cas, il devient difficile, si ce n'est impossible, de comprendre ce qui se passe. L'utilitaire Isort

(<http://timothycrosley.github.io/isort/>) effectue un si petit, mais utile travail, consistant à classer vos instructions `import` en bon ordre. Cela peut avoir un effet important sur la lisibilité et la compréhension du code, ainsi que sur la facilité de modification de celui-ci.

Savoir de quoi a besoin tel ou tel module peut aider à localiser des problèmes potentiels. Si vous obtenez quelque part une version plus ancienne d'un module nécessaire sur votre système, connaître les besoins exacts de votre application facilitera la recherche du module voulu plus aisée.

De plus, tout cela est également important lorsque vient le moment de distribuer votre application aux utilisateurs. Si celui-ci dispose des bons modules, vous aurez la garantie (ou presque) que votre application fonctionnera comme prévu.

Contrôler les versions en utilisant Mercurial

Les applications que vous avez créées avec les exemples de ce livre ne sont pas très complexes. En fait, lorsque vous voudrez franchir un palier, vous n'aurez pas tout de suite à vous préoccuper du contrôle des versions. Par contre, si vous travaillez dans un cadre professionnalisé dans lequel vous créez des applications de productivité que vos utilisateurs ont besoin de faire fonctionner en permanence, ce contrôle devient essentiel. Le *contrôle de version* est simplement l'acte consistant à suivre les modifications apportées à une application au fil du temps. Si vous dites par exemple que vous utilisez MonApp 1.2, vous

faites référence à la version 1.2 de l'application MonApp. Ceci permet à chacun de savoir quelle version il utilise lors de corrections de bogues et autres types de support.

Il existe de nombreux outils de contrôle de version dédiés à Python. L'un des plus intéressants est Mercurial (<https://www.mercurial-scm.org/>). Il est disponible sur pratiquement n'importe quelle plateforme reconnue par Python.

Un autre avantage de Mercurial, c'est qu'il est gratuit. Même si vous recherchez par la suite un outil de production plus avancé, vous pourrez déjà acquérir une certaine expérience de ce domaine en travaillant avec Mercurial sur un ou deux projets.



Le fait d'enregistrer chaque version d'une application dans un emplacement distinct, de manière à ce que les modifications puissent être défaites ou refaites à volonté, est appelé une *gestion du code source*. Cela peut sembler pour beaucoup être une tâche compliquée. Comme l'environnement de Mercurial est assez indulgent, cela vous permet de vous initier sans drame à la gestion de code source. Pouvoir revenir au code d'une version antérieure du code source est essentiel lorsqu'il faut réparer des problèmes créés par une nouvelle version.

De surcroît, Mercurial dispose d'une très bonne documentation en ligne (<https://www.mercurial-scm.org/wiki/Tutorial>). Bien entendu, cela débute par le fait d'installer correctement Mercurial. Vous apprendrez aussi à créer un dépôt (un emplacement dans lequel les versions successives sont enregistrées) et à l'utiliser dans le cadre du

développement de votre code. Une fois ces tutoriels parcourus, vous devriez avoir une bonne idée sur le fonctionnement du contrôle du code source, et avoir compris pourquoi le contrôle de version est si important dans le développement d'une application.

Chapitre 20

Dix bibliothèques à connaître

Dans ce chapitre :

- ▶ Sécuriser vos données.
- ▶ Travailler avec des bases de données.
- ▶ Découvrir le monde avec le géocodage.
- ▶ Présenter une interface graphique aux utilisateurs.
- ▶ Créer des tableaux.
- ▶ Travailler avec des graphiques.
- ▶ Trouver les informations dont vous avez besoin.
- ▶ Accéder à du code Java depuis une application Python.
- ▶ Accéder aux ressources du réseau local.
- ▶ Utiliser les ressources trouvées en ligne.

Python vous procure une très grande puissance pour créer des applications d'un niveau moyen. Cependant, la plupart des applications ne sont pas « moyennes » dans le sens où elles nécessitent des traitements spéciaux pour atteindre leurs objectifs. C'est là où les bibliothèques entrent en jeu. Une bonne bibliothèque

étend les capacités de Python de manière à ce que celui-ci supporte vos besoins spécifiques de programmation. Par exemple, vous pouvez avoir à traiter des statistiques, ou à interagir avec un appareillage scientifique. Tout cela nécessite le passage par une bibliothèque.



L'un des meilleurs emplacements pour trouver une liste de bibliothèques en ligne est le site UsefulModules, à l'adresse <https://wiki.python.org/moin/UsefulModules>. Bien entendu, ce n'est pas le seul endroit. Voyez par exemple la page <http://doda.co/7-python-libraries-you-should-know-about/> qui fournit une description relativement complète de sept bibliothèques utiles. Si vous travaillez sur une plate-forme spécifique, il existe également des sites plus spécifiquement dédiés à tel ou tel environnement. Dans le cas de Windows, voyez par exemple la page Unofficial Windows Binaries for Python Extension Packages, à l'adresse <http://www.lfd.uci.edu/~gohlke/pythonlibs/>.

Le propos de ce chapitre n'est pas d'ajouter de nouvelles références à votre liste, probablement déjà bien remplie, de bibliothèques potentiellement candidates. Il se contente de vous proposer dix bibliothèques qui fonctionnent sur n'importe quelle plate-forme, et qui fournissent des services de base susceptibles de répondre aux besoins de tout un chacun. C'est donc plutôt un point de départ pour préparer vos prochaines aventures de programmation.

Développer un environnement sécurisé avec PyCrypto

La sécurité des données est un élément essentiel dans tout effort de programmation. Les applications informatiques permettent de manipuler et d'utiliser des données de toutes sortes. Cependant, ces applications doivent protéger leurs données, ou sinon tous les efforts déployés risquent d'être vains. Ce sont les données qui constituent la véritable valeur, l'application n'étant qu'un outil à leur service. La protection des données comporte de multiples aspects, concernant notamment leur vol ou encore leur manipulation à des fins toutes autres que celles prévues initialement. C'est là où des bibliothèques de cryptographie, comme PyCrypto (<https://www.dlitz.net/software/pycrypto/>), entrent en jeu.



Le rôle principal de cette bibliothèque est de transformer vos données en quelque chose que les autres ne peuvent pas lire. Ce type de technique est appelé *encryptage*. Lorsque vous relisez des données ainsi cryptées, une procédure de *décryptage* les décode pour leur redonner leur format d'origine afin que l'application puisse les traiter. Au cœur de ce dispositif se trouve la notion de *clé*, qui est un code servant à encrypter et à décrypter les données. S'assurer de la sécurité de cette clé est donc également une des tâches essentielles de votre application. Vous pouvez lire les données parce que vous possédez la clé. Les autres ne peuvent pas les lire, car ils ne possèdent pas cette clé.

Interagir avec des bases de données grâce à SQLAlchemy

Une *base de données* est une méthode organisée d'enregistrement de données répétitives ou structurées sur un disque. Par exemple, des *enregistrements* (les entrées individuelles dans la base de données) de clients sont répétitifs, puisque chaque client est associé aux mêmes types d'informations : nom, adresse, numéro de téléphone, adresse de messagerie, et ainsi de suite. La manière d'organiser tout cela détermine le type de la base de données que vous utilisez. Certaines bases sont spécialisées dans l'organisation de données textuelles, d'autres dans des structures tabulées, d'autres encore dans la collecte d'informations provenant d'un instrument scientifique, et ainsi de suite. Les bases de données peuvent avoir une structure arborescente ou séquentielle, et tout un vocabulaire plutôt ésotérique circule à leur propos. Mais ce jargon, pour la plus grande part, concerne surtout les types qui sont chargés d'administrer toutes ces bases de données, pas vous.



Le type le plus couramment utilisé est appelé Système de Gestion de Base de Données Relationnelle (ou SGBDR). Il est basé sur des tables qui sont organisées en enregistrements et en champs, ou dit autrement, en lignes et en colonnes (comme un tableau que vous tracez sur une feuille de papier). Chaque *champ* correspond à une colonne qui contient toujours le même genre d'information, par exemple le nom des clients. Les tables sont liées les unes aux autres de diverses manières permettant de créer des relations complexes. Par exemple, chaque client peut avoir une ou plusieurs entrées dans une table des achats. La table des clients et celle des achats sont donc liées par une certaine relation.

Un SGBDR s'appuie sur un langage spécial pour accéder aux enregistrements individuels. Ce langage est appelé SQL, pour Structured Query Language (langage de requête structuré). Bien entendu, vous avez besoin dans ce cas d'un outil vous permettant d'interagir aussi bien avec le SGBDR qu'avec SQL. C'est là où intervient SQLAlchemy (<http://www.sqlalchemy.org/>). Ce produit réduit la quantité de travail à fournir pour interroger une base de données, afin par exemple de récupérer l'enregistrement d'un client, d'en créer un nouveau, de faire des mises à jour, ou encore de supprimer l'enregistrement d'un client parti voir ailleurs si le ciel y est plus bleu.

Voir le monde avec Google Maps

Le *géocodage* (qui consiste à retrouver des coordonnées géographiques, comme la longitude et la latitude à partir d'une certaine donnée, par exemple une adresse) a d'importantes applications dans notre monde moderne. Des hordes de gens se servent sans le savoir de ce géocodage pour trouver un bon restaurant, une location pour les prochaines vacances, ou encore localiser un excursionniste perdu quelque part dans les montagnes. Bien entendu, la détermination d'un trajet entre deux points relève des mêmes techniques. La bibliothèque *googlemaps* (<https://pypi.python.org/pypi/googlemaps/>) vous permet d'établir la connexion avec le célèbre Google Maps.

En plus de vous expliquer comment vous rendre à un certain endroit, ou de vous aider à trouver une âme perdue dans le désert, Google Maps peut aussi aider

dans des applications de type SIG (voyez également à ce sujet le Chapitre 18). Un SIG permet notamment de choisir un certain emplacement dans un certain but, ou encore de déterminer pourquoi un certain emplacement est meilleur qu'un autre pour répondre à un certain objectif. En bref, Google Maps offre à votre application une fenêtre vers le monde extérieur pouvant aider vos utilisateurs à prendre des décisions.

Ajouter une interface utilisateur graphique avec TkInter

Les utilisateurs aiment les interfaces graphiques, car elles sont plus amicales et demandent moins d'efforts de réflexion qu'une pure ligne de commande. Il existe de nombreux produits capables de procurer à vos applications un visage graphique. Cependant, celui qui est le plus utilisé est TkInter (<https://wiki.python.org/moin/TkInter>). Il est apprécié des développeurs de par sa facilité d'emploi. Il s'agit en fait d'une interface pour Tcl/Tk (Tool Command Language/ Toolkit). De nombreux langages utilisent Tcl/Tk (<http://www.tcl.tk/>) comme base pour créer une interface utilisateur graphique.



Vous pouvez ne pas être enthousiaste à l'idée d'ajouter une interface graphique à votre application. Bien entendu, cela prend du temps et ne rend pas votre application plus fonctionnelle (en fait, elle la ralentit le plus souvent). Mais n'oubliez pas que les utilisateurs aiment les interfaces graphiques. Si vous voulez que votre application connaisse une large diffusion, vous devrez certainement passer sous leurs fourches caudines.

Préserver des données tabulées avec PrettyTable

Afficher des données tabulées d'une manière facilement compréhensible par l'utilisateur est important. Au travers des exemples de ce livre, vous avez appris que Python enregistre ce type de données sous une forme qui convient surtout aux développeurs. Mais les besoins des utilisateurs sont différents. Ils veulent que les choses soient organisées de façon compréhensible pour eux, et autant que possible plaisantes à consulter. La bibliothèque PrettyTable (<https://pypi.python.org/pypi/PrettyTable>) peut grandement vous aider à obtenir ces résultats.

Sonorisier votre application avec PyAudio

Le son est un moyen pratique de transmettre certaines informations à vos utilisateurs. Vous devez avancer bien prudemment sur ce terrain, puisque certaines personnes peuvent ne pas entendre les sons, tandis qu'un abus de sons pourrait également venir perturber le cours normal du travail. Pour autant, des informations audio sont parfois d'une aide précieuse pour communiquer des informations supplémentaires aux utilisateurs (ou simplement pour rendre vos applications un peu plus attractives).

L'une des meilleures bibliothèques permettant d'associer du son à vos applications Python, et ce indépendamment de la plate-forme, est PyAudio (<http://people.csail.mit.edu/hubert/pyaudio/>). Cette bibliothèque permet d'enregistrer et de rejouer des sons (par exemple, vous voulez que vos utilisateurs

puissent enregistrer un mémo vocal afin de retrouver une liste de tâches à accomplir le moment venu).



Travailler avec le son sur un ordinateur implique toujours des compromis. Par exemple, une bibliothèque qui n'est pas liée à une certaine plate-forme sera incapable de profiter de certaines fonctionnalités spécifiques à tel ou tel système. De plus, elle peut ne pas supporter tous les formats audio utilisables sur une plate-forme spécifique. Utiliser une bibliothèque indépendante de la plate-forme permet de fournir un support sonore de base en s'en tenant à ce qui est connu et reconnu par tous.

Python et le son

Il est important de réaliser que les données sonores se présentent sous de multiples formes dans les ordinateurs. Les services multimédias de base fournis par Python apportent de ce point de vue des fonctionnalités de base (voyez la documentation en ligne à l'adresse <https://docs.python.org/3/library/mm.html>). En dehors de la lecture, ces services permettent également d'enregistrer certains types de fichiers audio, mais le choix des formats est très limité. De plus, certains modules, tel que winsound (<https://docs.python.org/3/library/winsound.html>) sont dépendants de la plate-forme, et vous ne pouvez donc pas les utiliser pour une application se voulant généraliste.

L'étage du dessus, autrement dit celui qui permet de disposer de fonctionnalités audio plus poussées, est habité par des bibliothèques comme PyAudio. Vous pouvez trouver une liste de ces bibliothèques à l'adresse <https://wiki.python.org/moin/Audio>. Celles-ci s'attachent surtout à répondre à des besoins professionnels. Autrement dit, la Hi-Fi n'est pas dans leur viseur.

Les joueurs, de leur côté, ont besoin d'un support audio bien plus évolué, par exemple pour entendre les pas d'un monstre avant qu'il ne les assassine par l'arrière. Il existe pour cela des bibliothèques spécialisées, comme PyGame (<http://www.pygame.org/news.html>). Bien sûr, ce type de bibliothèque nécessite un équipement audio plutôt haut de gamme, et énormément de travail pour arriver à sonoriser votre application. Pour voir une liste de ces bibliothèques, consultez la page <https://wiki.python.org/moin/PythonGameLibraries>.

Manipuler des images avec PyQtGraph

Les humains sont attirés par tout ce qui est visuel. Si vous présentez à quelqu'un un tableau d'informations, puis que vous montrez ces mêmes informations sous une forme graphique, c'est cette dernière version qui va forcément l'emporter. Les graphiques aident les gens à mieux voir les tendances et à comprendre pourquoi les données suivent une certaine courbe. Cependant, transformer un tableau d'informations en

un dessin formé de pixels affichés sur l'écran n'est pas chose facile. C'est pourquoi vous avez besoin d'une bibliothèque spécialisée, telle que PyQtGraph (<http://www.pyqtgraph.org/>) pour rendre ce travail plus simple.

Même si la bibliothèque est conçue autour des besoins de domaines scientifiques, mathématiques ou techniques, il n'y a aucune raison de s'en passer pour d'autres objectifs. PyQtGraph supporte les vues 2D et 3D, et vous pouvez l'utiliser pour générer des graphiques à partir de données numériques variées. La sortie est complètement interactive, ce qui signifie que l'utilisateur peut sélectionner des zones sur une image pour les améliorer, ou effectuer d'autres types de manipulation. De plus, cette bibliothèque est livrée avec de nombreuses options utiles, comme des contrôles ou des boutons, qui facilitent le processus de codage.



Contrairement à la plupart des autres bibliothèques présentées dans ce chapitre, PyQtGraph n'est pas autonome. En d'autres termes, vous ne pouvez l'utiliser qu'en installant d'autres produits. Cela n'a rien de surprenant au regard des multiples tâches que doit accomplir PyQtGraph. Vous avez besoin des éléments suivants :

- Python version 2.7 ou plus
- PyQt version 4.8 ou plus (<https://wiki.python.org/moin/PyQt>) ou PySide (<https://wiki.python.org/moin/PySide>)
- numpy (<http://www.numpy.org/>)
- scipy (<http://www.scipy.org/>)
- PyOpenGL (<http://pyopengl.sourceforge.net/>)

Localiser vos informations avec IRLib

Localiser vos informations peut devenir difficile lorsque celles-ci prennent de l'ampleur. Imaginez que votre disque dur est une grosse base de données arborescente et non structurée, de surcroît dépourvue d'un index utile. Chaque fois qu'une structure atteint une taille critique, des données sont tout simplement perdues (essayez juste de retrouver vos photos des avant-dernières vacances, et vous allez vite comprendre l'idée). Intégrer un certain type de fonction de recherche dans votre application est donc important pour que vos utilisateurs puissent retrouver un fichier perdu, ou d'autres informations.



Il existe de nombreuses bibliothèques Python dévolues à la recherche d'informations. Le problème avec la plupart d'entre elles, c'est qu'elles sont difficiles à installer ou qu'elles ne fournissent pas un support cohérent pour toutes les plates-formes. En fait, certaines ne fonctionnent que sur une ou deux plates-formes. Cependant, IRLib (<https://github.com/gr33ndata/irlib>) est écrit en pur Python, ce qui garantit son indépendance vis-à-vis de la plate-forme. Si IRLib ne répond pas à vos besoins, assurez-vous que le produit que vous allez choisir fournit les fonctionnalités de recherche voulues sur toutes les plates-formes que vous sélectionnez et que son installation n'est pas inabordable.

IRLib crée un index à partir des informations avec lesquelles vous voulez travailler. Vous pouvez ensuite sauvegarder cet index sur le disque dur. Le mécanisme de recherche localise une ou plusieurs entrées répondant au mieux au critère qui a été défini.

Créer des liens avec Java en utilisant JPyte

Python donne accès à une immense galerie de bibliothèques dont vous ne pourrez jamais faire le tour. Cependant, vous pouvez rencontrer une situation dans laquelle vous trouvez une bibliothèque Java qui serait parfaite pour ce que vous voulez réaliser, mais que vous ne pouvez pas utiliser dans votre application Python, à moins d'apprendre à jongler. La bibliothèque JPyte (<http://jpype.sourceforge.net/>) rend possible l'accès à la plupart des bibliothèques Java (mais pas à toutes) directement dans Python. Elle permet de créer un pont entre les deux langages au niveau du code interprété. Vous n'avez donc pas à vous lancer dans des contorsions extrêmes pour que votre application Python communique avec Java.

Convertir votre application Python en Java

Il existe différentes manières d'obtenir une certaine interopérabilité entre deux langages. Créer un pont entre eux, comme le fait JPyte, est une de ces méthodes. Une autre alternative consiste à convertir le code créé dans un langage dans le format d'un langage différent. C'est l'approche employée par JPython (<https://wiki.python.org/jython/>).

Cet utilitaire convertit le code Python en code

Java, de manière à profiter pleinement des fonctionnalités de Java tout en conservant celles que vous appréciez dans Python.

Bien entendu, cette interopérabilité n'est pas un chemin pavé de roses. Dans le cas de JPy, vous n'avez pas accès à certaines bibliothèques Java. En plus, cette approche pénalise la rapidité d'exécution, car JPy doit en permanence convertir les appels et les données. Avec JPython, le problème est que vous perdez la possibilité de modifier votre code une fois celui-ci converti. Tout changement ultérieur créerait une incompatibilité entre le code Python original et son équivalent Java. Il n'existe donc aucune solution parfaite pour profiter du meilleur des deux mondes.

Accéder à des ressources réseau locales avec Twisted Matrix

Selon la configuration de votre réseau, vous pouvez avoir besoin d'accéder à des fichiers ou à d'autres ressources, ce que vous ne pouvez pas faire en utilisant les fonctions natives de la plate-forme. Dans ce cas, vous avez besoin d'une bibliothèque rendant possible cet accès, comme c'est le cas avec Twisted Matrix (<https://twistedmatrix.com/trac/>). L'idée de base de cette bibliothèque, c'est de vous fournir les appels dont vous avez besoin pour établir une connexion, et ce quel que soit le type de protocole utilisé.

Ce qui rend cette bibliothèque si pratique, c'est notamment sa nature pilotée par les événements. Cela signifie que votre application n'a pas à suspendre son cours en attendant que le réseau réponde. De plus, le pilotage par événements facilite grandement l'implémentation de communications asynchrones (c'est-à-dire dans lesquelles une requête est envoyée par une routine, puis est gérée par une autre routine complètement distincte de la première).

Accéder à des ressources Internet en utilisant des bibliothèques

Même si des produits tels que Twisted Matrix peuvent gérer les communications en ligne, une bibliothèque dédiée au protocole HTTP est souvent une meilleure option lorsque vous travaillez avec Internet, car elle est plus rapide et offre davantage de fonctionnalités. Si vous avez spécifiquement besoin d'un support HTTP ou HTTPS, utiliser une bibliothèque comme `httplib2` (<https://github.com/jcgregorio/httplib2>) est une bonne idée. Cette bibliothèque est écrite en pur Python. Elle rend aussi la gestion de fonctionnalités spécifiques à `http`, comme définir une valeur `Keep-Alive`, relativement facile (une valeur `Keep-Alive` définit la durée pendant laquelle un port reste ouvert dans l'attente d'une réponse, si bien que l'application n'a pas à recréer en permanence la connexion, ce qui se traduirait par une perte de temps et un gaspillage des ressources).

Vous pouvez utiliser `httplib2` pour n'importe quelle méthodologie spécifique à Internet. Elle fournit par exemple un support complet pour les méthodes de requête `GET` et `POST`. Elle contient également des routines

pour les méthodes de compression standards utilisées sur Internet, comme *deflate* ou *gzip*. Elle supporte aussi un certain niveau d'automatisation des tâches, comme la récupération de ressources déjà mises en cache dans les requêtes `PUT`.

Index

« Pour retrouver la section qui vous intéresse à partir de cet index, utilisez le moteur de recherche »

A

Assesseurs

Affectation

Apostrophes

Appelant

- passer des informations à IP

Applications

- ajouter des commentaires

- améliorer la rapidité

- améliorer le code

- contrôler les versions

- convertir en java

- créer

- déboguer

- définition des

- documenter

- exécuter

- fichiers et

- informations et

- installer

- interagir avec l'utilisateur

- ordinateur et

- procédures et

- pydoc

- Python

sauvegarder
scientifiques
scripts et
se poser des questions sur les autres
sonoriser
tester
utiliser des classes dans les

Arbre de sélection

Args

Arguments

exceptions et
fonctions et
méthodes d'instance et
nombre
variable
variable du

nommés

non nommés

passer des
propriété args

self

transmettre

par nom

par position

valeur par défaut

ASCII

Astérisque

Attributs

classes et
générés par python
listes d'
newline
sys.path

B

Bases de données

Bibliothèques

- code source

- ressources Internet et

Bloc de code

Bogues

Booléen

Boucles

- Condition d'arrêt

- Contrôler

 - avec l'instruction continue

 - avec l'instruction else

 - avec l'instruction pass

for

imbriquer

quitter avec l'instruction break

sans fin

simples

while

break

C

C

Caractères

- accéder aux

- autres

- de contrôle

- de dessin

- de typographie

- sélectionner

spéciaux

Chaînes de caractères

- alphabétiques

- alphanumériques

- capitaliser

- caractères

 - autres

 - de contrôles

 - de dessin

 - de typographie

 - spéciaux

centrer

décimales

définir

- en utilisant des caractères

- en utilisant des nombres

délimitées par un espace

entre apostrophes

entre guillemets

exceptions et

formater

justifier

localiser

- des occurrences

- des valeurs

- la fin

longueur

manipuler

numériques

partager

remplacer des valeurs

remplir

répétitives

retrouver dans des

sélectionner des caractères
séquences d'échappement
supprimer les espaces
tabulations et
trancher et couper
vierges

Champ
Chemin d'accès
absolu
relatif

CLAS
Classes
attributs et
composants
comprendre les
créer
enfant
étendre
instancier
méthodes
de
et

objets et
parent
tester
utiliser dans des applications
variables de

Clauses
Clés
Code exécutable
Collections
LIFO

Commandes

def
exit()
help()
ord()
print()
quit()
str()
type()

Commentaires

comme aide-mémoire
pour empêcher du code de s'exécuter

Communication

Comparaisons

multiples
ordinateurs et

Conditions

Connaître plusieurs langages de
programmation

Constructeurs

__init__ 300

Continue

Contrôle de version

Counterl

D

Data mining

Dates

Débogueur

Décision

tâches multiples et

Décisions

imbriquées

Dequeues

Dictionnaires

clés

instruction switch et

Données

analyse en temps réel

bases de

collecter

enregistrer

exploration des

listes et

membres de

non structurées

ordinateurs et

Python et

rechercher

retourner depuis une fonction

scientifiques

sécurité des

structurées

types de

tabulées

Dossiers

Chemin d'accès

Organisation des

Supprimer

E

elif

else

Else

e-mails

consulter

contenu

- courrier et
créer
 - au format HTML
 - au format texte

- destinataire
- en-tête
- enveloppe
- envoyer
- expéditeur
- message
- méthode de transmission
- types MIME et

Enregistrements

Entiers

- base
- binaires
- hexadécimaux

Environnements virtuels

Erreurs

- de compilation
- de logique
- de syntaxe
- d'exécution
- sémantiques
- sources potentielles
- types de

except

- exceptions multiples et
- utiliser sans exception

Exceptions

- arguments et
- chaînes de caractères et
- clause finally et

- gérer
- imbriquer
- intercepter
- lever
- liste des arguments
- multiples
- passer des informations à l'appelant
- personnalisées
- prédéfinies de Python
- propager
- stratégie de gestion des
- uniques

- Exposant
- Expressions
- Extension

F

- Fichiers
 - applications et
 - créer
 - de modules
 - données et
 - extension
 - formats de
 - HTML
 - lire le contenu
 - mettre à jour
 - NEWS
 - noms de
 - ouvrir
 - py
 - README
 - refermer
 - sauvegarder
 - supprimer

XML

FIFO

Files

finally

Fonctions

appelant

appeler

append()

appendleft()

arguments et

capitalize()

center()

chaînes de caractères et

clear()

clear

Code et

copy()

count()

date()

Définir

dir()

doc()

empty()

endswith()

expandtabs()

extend()

find()

float()

format()

full()

get()

index()

__init__()

input()

insert()

int()

isalnum()
isalpha()
isdecimal()
isdigit()
islower()
isnumeric()
isspace()
istitle()
isupper()
items()
join()
len()
lower()
lstrip()
max()
min()
Nombre variable
arguments
open()
passer des arguments
pop()
Pop()
popleft()
Push()
put()
range()
remove()
replace()
retourner des données
réutilisables
reverse()
rfind()
rindex()
rjust()
rstrip()
sort()
split()

splitlines()
startswith()
__str__()
strip()
surcharger
swapcase()
time()
titile()
upper()
zfill()

for

Formatage

affichage alternatif
alignement et
largeur et
précision et
remplissage et
séparateur de milliers
signes et
type de la sortie et

G

Géocodage

Gestionnaire d'erreur

Getters

Google Maps

Graphiques

Groupe de code

Guillemets

H

Héritage

Heures

Hôte
adresse de l'
local

HTML
HTTP
httplib2
HTTPS

/

IDLE
codage des couleurs
codes de couleur
commandes standard et
configurer
exécuter une application
indentation du code
lancer
obtenir de
aide
ouvrir une fenêtre
quitter
Raccourcis clavier

if
Indentation
Informations
données et
enregistrer
listes et
ordinateurs et
rechercher

Instance
méthode d'
variable d'

Instances

Instanciación

Instancier

Instructions

break

continue

Else

except

exceptions multiples et

Sans exception

finally

for

from...import

if

combiner

multiples

if...elif

combiner

if...else

combiner

multiples

import

collections

utiliser

pass

switch

try

while

condition et

utiliser

with

IPython
IRLib
Isort

J

Java
JPyte

K

Komodo Edit

L

Langage de programmation
Python

LearPython.org

LIFO

Ligne de commandes
maîtriser
refermer
taper une commande

Listes

accéder aux
ajouter un élément
copier
créer
d'attributs
deques et
effacer le contenu
étendre
files et
insérer des éléments

- modifier
 - objet counter et
 - opérateurs et
 - ordinateurs et
 - parcourir
 - pires et
 - rechercher dans les
 - supprimer
 - des éléments
 - un élément de fin de

- trier

- Localhost
- Localisation

M

- Mantisse
- Mercurial
- Messagerie
- Méthodes
 - arguments variables
 - classes et
 - close()
 - constructeurs et
 - de classe
 - d'instance
 - arguments et
- variables d'instance et

- MIME
 - sous-types

- Modules
 - Datetime

- documentation des
- documenter
- importer
- trouver
- voir le contenu

Mutateurs

N

Nombres complexes

O

Objets

Opérandes

Opérateurs

- arithmétiques

- au niveau du bit

- d'affectation

- définir les

- d'identité

- listes et

- logiques

- membres

- ordre de priorité

- relationnels

- surcharger

- ternaire

- travailler avec les

- unaires

Ordinateur

- applications et

- code binaire et

- communication et

comparaisons et
informations et
langage de programmation et
listes et
mode de fonctionnement
parler à

Outils
Orphelins

Outils de développement

P

pass
pdoc
PERL
Pile
Piles
Plates-formes
Port
PrettyTable
print()
Procédures
 applications et
 écrire des
 réfléchir aux

Protocoles
Prototype
PyAudio
PyCrypto
pydbg
pydoc
 liens d'accès rapide
 rechercher un terme
 visualiser les résultats

PyQtGraph

Pytho documentation des modules

Python

- accéder à

 - sous Linux

 - sous Windows

 - sur un Mac

- accéder à des ressources internet

- administration de réseau et

- aide

 - de

 - directe

 - générale

- améliorer

 - la rapidité des applications

 - le code

- analyse de données en temps réel

- applications utiles

- attributs générés par

- bases de données et

- bibliothèques tierces

- C# et

- chaînes de caractères et

- choisir

- classes

- commandes

- comparer avec d'autres langages

- convertir les applications en java

- créer une application

- Data mining et

- déboguer les applications

- dictionnaires

- documentation en ligne

- documenter les applications

- domaines d'utilisation

données
et
scientifiques et

dossier
courant
par défaut

éditeurs
environnement
de développement interactif
interactif
virtuels et

erreurs
et

exceptions prédéfinies
fichiers.py
fonctions
de chaînes de caractères

formation et
formats de fichiers et
graphiques et
IDLE
importer des modules
installer
sous Linux
sous Windows
sur un Mac

interagir avec l'utilisateur
intérêt de
interfaces graphiques en
Java et
lancer
ligne de commandes

listes et
mode Shell
options
outils
PERL et
plates-formes et
prendre des décisions
programmer pour le web
prototype
quitter
l'aide

rechercher de l'aide
ressources
scripts
sécurité des données et
SIG et
son et
système(s)
de production et
embarqués et

taper une commande
télécharger
tester les applications
tester l'installation
trouver des modules
tutoriels
types de données
unicode et
variables
d'environnement
d'environnement et

vérifier la syntaxe
version
voir le contenu des modules
XML et

Python arguments
PythonEditors
Python.vim
Pythonware
PyUnit

Q

Queues

R

Raspberry Py
Réseau
Roundup Issue Tracker

S

Scripts
Séquence
d'échappement

Serveur SMTP
Setters
SGBDR
SIG
Signe #
SMTP
Socket
Son
Sous-dossiers
supprimer

SQLAlchemy
Supports de stockage

Système(s)
de Gestion de Base de Données
Relationnelle
embarqués

T

Tabulations
TkInter
try
Tuples
immutabilité
travailler avec les

Twisted Matrix
Types de données multiplicité des

U

Unicode

V

Variables
affecter une valeur
booléennes
chaînes de caractères
complexes
dates et heures
de classe
d'environnement
déterminer le type
d'instance
initialisation

entières

en virgule flottante
numériques
ERRORLEVEL
PYTHONPATH

Virgule flottante
VirtualEnv

W

W3Schools
While

X

XML
Apprendre