INFORMATIQUE

PYTHON

Comprendre les bases et maîtriser la programmation

BILL LUBANOVIC

deboeck SUPÉRIEUR

INFORMATIQUE

Python

Comprendre les bases et maîtriser la programmation

BILL LUBANOVIC

Traduction de Bernard Desgraupes

deboeck SUPÉRIEUR

Ouvrage original :

Rue du bosquet 7, B - 1348 Louvain-la-Neuve

Dépôt légal :
Bibliothèque Nationale, Paris : avril 2022
Bibliothèque royale de Belgique, Bruxelles : 2022/13647/035

ISBN 978-2-8073-3473-1

Avec mon affection à Mary, Tom & Roxie, et Karin & Erik

Table des matières

Avant-propos

Comme promis dans le titre, ce livre vous présente l'un des langages de programmation le plus populaire au monde : Python. Il s'adresse aussi bien aux programmeurs débutants qu'aux programmeurs plus expérimentés qui souhaitent ajouter Python aux langages qu'ils connaissent déjà.

Dans la plupart des cas, il est plus facile d'apprendre un langage informatique qu'un langage humain. Il y a moins d'ambiguïtés ou d'exceptions qu'il faut garder à l'esprit. Python est l'un des plus cohérents et les plus limpides des langages informatiques. Il concilie facilité d'apprentissage, facilité d'utilisation et puissance expressive.

Les langages informatiques sont constitués de *données* (semblables aux noms dans les langues parlées) et d'*instructions* ou de *code* (semblables aux verbes). Vous avez besoin des deux. Dans des chapitres alternés, vous ferez l'apprentissage du code et des structures de base de Python, puis vous apprendrez à les combiner et à développer des structures plus avancées. Les programmes que vous lirez et écrirez deviendront plus longs et plus complexes. En utilisant une analogie avec le travail du bois, nous commencerons avec un marteau, des clous et des bouts de bois. Au cours de la première moitié de ce livre, nous présenterons des composants plus spécialisés, jusqu'à parvenir à l'équivalent des tours à bois et autres outillages évolués.

Vous apprendrez non seulement le langage mais aussi ce que vous pouvez en faire. Nous commencerons par le langage Python et sa bibliothèque standard fournie par défaut mais je vais également vous montrer comment trouver, télécharger, installer et utiliser de bons packages développés par des tiers. Je mettrai l'accent sur tout ce que j'ai trouvé d'utile en plus de 10 années de développement en production avec Python, plutôt que sur des sujets marginaux ou des bricolages complexes.

Bien qu'il s'agisse d'une introduction, j'ai inclus certains sujets avancés que je souhaite exposer. Des domaines tels que les bases de données et le Web sont bien sûr couverts mais la technologie évolue rapidement. On peut désormais attendre d'un programmeur Python qu'il ait aussi des notions sur le cloud computing, l'apprentissage automatique ou le streaming d'événements. Vous trouverez ici des informations sur toutes ces questions.

Python a des fonctionnalités spéciales qui sont plus performantes qu'une simple adaptation de paradigmes provenant d'autres langages que vous connaissez peut-être. Par exemple, utiliser for et des itérateurs est un moyen plus direct de créer une boucle que d'incrémenter manuellement une variable de compteur.

Lorsque vous apprenez quelque chose de nouveau, il est difficile de déterminer quels termes sont spécifiques, au-delà de leur acception familière, et quels concepts sont réellement importants. En d'autres termes, « Que faut-il apprendre » ? Je vais souligner les termes et les idées qui ont une signification ou une importance particulière avec Python, mais pas trop à la fois. Du véritable code Python sera introduit tôt et souvent.

J'inclurai une note comme celle-ci lorsque quelque chose pourrait prêter à confusion, ou s'il existe une manière pythonique plus appropriée de le faire.

Python n'est pas parfait. Je vais vous montrer des choses qui paraissent étranges ou qui devraient être évitées – et vous proposer des alternatives que vous pouvez utiliser à la place.

De temps en temps, mes opinions sur certains sujets (tels que l'héritage d'objets ou les modèles MVC et REST pour le Web) peuvent différer un peu de la sagesse commune. À vous de voir ce que vous en pensez.

Public concerné

Ce livre s'adresse à tous ceux qui souhaitent apprendre l'un des langages informatiques le plus populaire au monde, que vous ayez ou non déjà appris la programmation.

Changements dans la deuxième édition

Qu'est-ce qui a changé depuis la première édition ?

- Une centaine de pages supplémentaires, y compris des photos de chats.
- Deux fois plus de chapitres, chacun étant plus court maintenant.
- Un premier chapitre consacré aux types de données, aux variables et aux noms.
- De nouvelles fonctionnalités Python standard telles que les "*f-chaînes*" (f-strings).
- Des packages tiers nouveaux ou améliorés.
- De nouveaux exemples de code tout du long.
- Une annexe sur le matériel et les logiciels de base, pour les nouveaux programmeurs.
- Une annexe sur *asyncio*, pour les programmeurs plus expérimentés.
- Les technologies « New stack » : conteneurs, nuages, analyse des données et apprentissage automatique.
- Des conseils pour décrocher un travail de programmation en Python.

Qu'est-ce qui n'a pas changé ? Les exemples utilisant de la mauvaise poésie et des canards. Ils sont à feuillage persistant.

Plan

La première partie (chapitres 1 à 11) explique les bases de Python. Vous devriez lire ces chapitres dans l'ordre. Je travaille à partir des structures de données et de code les plus simples, en les combinant en cours de route dans des programmes plus détaillés et plus réalistes. La deuxième partie (chapitres 12 à 22) montre comment Python est utilisé dans des domaines d'application spécifiques tels que le Web, les bases de données, les réseaux, etc. Lisez ces chapitres dans l'ordre qui vous convient.

Voici un bref aperçu des chapitres et des annexes, en incluant certains des termes que nous rencontrerons par la suite :

Chapitre 1, Un avant-goût de Python
: Les programmes informatiques ne sont pas si différents des instructions que vous rencontrez tous les jours. Quelques petits programmes Python vous donneront un aperçu de l'apparence, des capacités et des utilisations du langage dans le monde réel. Vous verrez comment exécuter un programme Python dans son i*nterpréteur interactif* (ou *shell*) ou à partir d'un *fichier texte* enregistré sur votre ordinateur.

Chapitre 2, Données : Types, Valeurs, Variables et Noms
: Les langages informatiques mélangent données et instructions. Différents types de données sont stockés et traités différemment par l'ordinateur. Ils peuvent permettre à leurs valeurs d'être modifiées ou non (elles sont dites respectivement *modifiables* ou *immuables*). Dans un programme Python, les données peuvent être *littérales* (des nombres comme 78, des chaînes de *texte* comme "gaufre") ou représentées par des *variables* nommées. Python traite les variables comme des *noms*, ce qui est différent de nombreux autres langages et a des conséquences importantes.

Chapitre 3, Nombres
: Ce chapitre présente les types de données les plus simples de Python : *booléens*, *entiers* et *nombres à virgule flottante*. Vous apprendrez également les opérations mathématiques de base. Les exemples utilisent l'interpréteur interactif de Python comme une calculatrice.

Chapitre 4, Choisir avec if
: Nous allons jongler avec les substantifs (types de données) et les verbes (structures de programme) du langage Python pendant plusieurs chapitres. Le code Python exécute normalement une ligne à la fois, du début à la fin d'un programme. La structure de code `if` vous permet d'exécuter différentes lignes de code, en fonction de certaines conditions sur les données.

Chapitre 5, Chaînes de texte
: Retour aux substantifs et au monde des *chaînes* de texte. Apprenez à créer, combiner, modifier, récupérer et afficher des chaînes de caractères.

Chapitre 6, Boucles avec for et while
: Retour aux verbes encore une fois et deux façons de faire une boucle : `for` et `while`. Vous découvrirez un concept Python de base : les itérateurs.

Chapitre 7, Tuples et Listes
: C'est le moment d'introduire les premières structures de données internes de haut niveau de Python : les listes et les tuples. Ce sont des séquences de valeurs, comme des LEGO pour construire des structures de données plus complexes. Parcourez-les avec des *itérateurs* et créez rapidement des listes avec des *compréhensions*.

Chapitre 8, Dictionnaires et ensembles
: Les *dictionnaires* (ou *dicts*) et les *ensembles* (ou *sets*) vous permettent d'enregistrer les données par leurs valeurs plutôt que par leur position. Cela s'avère très pratique et fera rapidement partie de vos fonctionnalités préférées.

Chapitre 9, Fonctions
: Assemblez entre elles les données et les structures de code des chapitres précédents pour comparer, choisir ou répéter. Enveloppez le code dans des *fonctions* et gérez les erreurs au moyen des *exceptions*.

Chapitre 10, Oh Oh : Objets et classes
: Le mot objet est un peu flou, mais important dans de nombreux langages informatiques, y compris Python. Si vous avez fait de la *programmation orientée objet* dans d'autres langages, Python est un peu plus laxiste. Ce chapitre explique comment utiliser des objets et des classes, et quand il est préférable d'utiliser des alternatives.

Chapitre 11, Modules, packages et bonus
: Ce chapitre montre comment évoluer vers des structures de code plus larges : *modules*, *packages* et *programmes*. Vous verrez où placer le code et les données, comment manipuler des données en entrée et en sortie, gérer les options, visiter la bibliothèque standard Python et jeter un coup d'œil à ce qui existe au-delà.

Chapitre 12, Manipuler et modifier les données
: Apprenez à gérer (ou modifier) les données comme un pro. Ce chapitre porte sur le texte et les données binaires, les délices des caractères Unicode et la recherche de texte avec des expressions régulières. Il présente également les types de données *bytes* et *bytearray*, équivalents des chaînes qui contiennent des valeurs binaires brutes au lieu de caractères de texte.

Chapitre 13, Calendriers et horloges
: Les dates et les heures peuvent être difficiles à gérer. Ce chapitre présente les problèmes courants et des solutions utiles.

Chapitre 14, Fichiers et répertoires
: Le stockage de données de base utilise des *fichiers* et des *répertoires*. Ce chapitre vous montre comment les créer et les utiliser.

Chapitre 15, Données temporelles : processus et concurrence
: Ceci est le premier chapitre sur les tâches de bas niveau. Son thème est la gestion temporelle des données – comment utiliser les *programmes*, les *processus* et les *threads* pour faire plus de choses simultanément (*concurrence*). Les ajouts asynchrones récents de Python sont mentionnés, avec des détails dans l'annexe C.

Chapitre 16, Données en boîte : le stockage permanent
: Les données peuvent être stockées et récupérées avec des fichiers de base et des répertoires au sein des systèmes de fichiers. Elles acquièrent une certaine structure avec des formats de texte courants tels que CSV, JSON et XML. À mesure que les données deviennent plus volumineuses et plus complexes, elles ont besoin des services de *bases de données* – des bases *relationnelles* traditionnelles ou certains réservoirs de données *NoSQL* plus récents.

Chapitre 17, Données spatiales : les réseaux
: Envoyez votre code et vos données à travers des réseaux au moyen des *services*, des *protocoles* et des *API*. Les exemples vont des *sockets* TCP de bas niveau, aux bibliothèques de *messagerie* et aux systèmes de mise en file d'attente, en passant par le déploiement dans le *cloud*.

Chapitre 18, Le Web démêlé
: Le Web a son propre chapitre : clients, serveurs, API et frameworks. Vous allez *explorer* et *fouiller* des sites Web, puis créer de vrais sites Web avec des paramètres de *requête* et des *gabarits*.

Chapitre 19, Devenir Pythoniste
: Ce chapitre contient des astuces pour les développeurs Python – installation avec pip et virtualenv, utilisation des IDE (plateformes de développement), test, débogage, journalisation, contrôle de version et documentation. Il vous aide également à trouver et installer des packages tiers utiles, à créer votre propre code pour le réutiliser et à savoir où obtenir plus d'informations.

Chapitre 20, Illustrations avec Python
: Les gens font des choses étonnantes avec Python dans les arts : graphisme, musique, animation et jeux.

Chapitre 21, Py au travail
: Python a des applications spécifiques pour les entreprises : visualisation de données (tracés, graphiques et cartes), sécurité et réglementation.

Chapitre 22, Python pour les sciences
: Au cours des dernières années, Python est devenu l'un des principaux langages scientifiques : mathématiques et statistiques, sciences physiques, biosciences et médecine. L'*analyse des données* et l'*apprentissage automatique* sont des atouts notables. Ce chapitre traite de NumPy, SciPy et Pandas.

Annexe A, Matériel et logiciel pour les débutants
Si vous êtes assez novice en programmation, cette annexe décrit comment matériel et logiciels fonctionnent réellement. Il présente certains termes que vous rencontrerez sans cesse.

Annexe B, Installer Python 3
: Si vous n'avez pas encore Python 3 sur votre ordinateur, cette annexe vous montre comment l'installer, que vous exécutiez Windows, macOS, Linux ou une autre variante d'Unix.

Annexe C, Quelque chose de complètement différent : Async
: Python a ajouté des fonctionnalités asynchrones dans différentes versions, et elles ne sont pas faciles à comprendre. Je les mentionne au fur et à mesure qu'elles apparaissent dans divers chapitres mais j'ai réservé leur discussion détaillée pour cette annexe.

Annexe D, Solutions des exercices
: Cette annexe contient les réponses aux exercices de fin de chapitre. Ne jetez pas un œil ici tant que vous n'avez pas essayé les exercices vous-même, sinon vous risquez d'être transformé en triton.

Annexe E, Antisèches
: Cette annexe contient des antisèches à utiliser comme référence rapide.

Versions de Python

Les langages informatiques évoluent avec le temps à mesure que les développeurs ajoutent des fonctionnalités et corrigent des erreurs. Les exemples de ce livre ont été écrits et testés en utilisant Python version 3.7. La version 3.7 était la plus récente au moment de l'édition de ce livre, et je parlerai de ses ajouts les plus notables. La version 3.8 doit être publiée fin 2019, et j'inclurai quelques éléments qu'elle introduit. Si vous voulez savoir ce qui a été ajouté à Python et quand, essayez la page What's New in Python (Quoi de neuf en Python). C'est une référence technique un peu lourde lorsque vous débutez avec Python, mais qui peut se révéler utile par la suite si vous devez faire fonctionner des programmes sur des ordinateurs avec différentes versions de Python.

Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique
: Indique les nouveaux termes, URL, adresses e-mail, noms de fichier et extensions de fichier.

`Largeur constante`
: Utilisé pour les listings de programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que des variables, des fonctions et des types de données.

`Largeur constante en gras`
Affiche les commandes ou tout autre texte qui doit être tapé littéralement par l'utilisateur.

`Italique à largeur constante`
Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

Cette icône indique un conseil, une suggestion ou une note générale.

Cette icône indique un avertissement ou une mise en garde.

Utilisation d'exemples de code

Les exemples de code et les exercices de ce livre sont disponibles pour un téléchargement en ligne. https://github.com/madscheme/introducing-python Ce livre est là pour vous aider à faire votre travail. En général, vous pouvez utiliser le code de ce livre dans vos programmes et votre documentation. Vous n'avez pas besoin de nous contacter pour obtenir notre autorisation, sauf si vous reproduisez une partie importante du code. Par exemple, l'écriture d'un programme qui utilise plusieurs morceaux de code de ce livre ne nécessite pas d'autorisation. La vente ou la distribution d'exemples de livres O'Reilly nécessite une autorisation. Répondre à une question en citant ce livre et en citant un exemple de code ne nécessite pas d'autorisation. L'incorporation d'une quantité importante d'exemples de code de ce livre dans la documentation de votre produit nécessite une autorisation.

Nous apprécions mais nous ne demandons pas d'attribution. Une attribution comprend généralement le titre, l'auteur, l'éditeur et l'ISBN. Par exemple : « *Introduction à Python* par Bill Lubanovic (O'Reilly). Copyright 2020 Bill Lubanovic, 978-1-492-05136-7. »

Si vous pensez que votre utilisation d'exemples de code ne relève pas de l'utilisation équitable ou de l'autorisation donnée ici, n'hésitez pas à nous contacter à *permissions@oreilly.com*.

Remerciements

Mes sincères remerciements aux responsables d'édition et aux lecteurs qui ont contribué à améliorer ce produit :

Corbin Collins, Charles Givre, Nathan Stocks, Dave George et Mike James

PREMIÈRE PARTIE

Les bases de Python

CHAPITRE 1

Un avant-goût de Python

> Seuls les langages laids deviennent populaires. Python est la seule exception.
>
> —Don Knuth

Mystères

Commençons par deux mini-mystères et leurs solutions. Que pensez-vous des deux lignes suivantes ?

```
(Rangée 1): (RS) K18, ssk, k1, retournez l'ouvrage.
(Rangée 2): (WS) Sl 1 pwise, p5, p2tog, p1, retourner.
```

Cela semble technique, comme une sorte de programme informatique. En fait, c'est un *modèle de tricot* ; plus précisément, un fragment décrivant comment tourner le talon d'une chaussette, comme celui de la figure 1-1.

Figure 1-1. Chaussettes tricotées

Cela a autant de sens pour moi qu'une grille de Sudoku pour l'un de mes chats, mais ma femme le comprend parfaitement et, si vous tricotez, vous aussi.

Essayons un autre texte mystérieux, trouvé sur une fiche. Vous comprendrez immédiatement son objectif, même si vous ne connaissez peut-être pas le produit final :

```
    1/2 C. de beurre ou margarine
    1/2 C. de crème
    2 1/2 C. de farine
    1 pincée de sel
    1 cuillère à soupe de sucre
    4 c. de pommes de terre émincées (froides)

  Assurez-vous que tous les ingrédients sont froids avant d'ajouter la farine.
  Mélangez tous les ingrédients.
  Pétrissez bien.
  Formez 20 boules. Conserver au froid jusqu'à l'étape suivante.
  Pour chaque boule :

    Étalez de la farine sur un chiffon.
    Faites rouler la boule en cercle avec un rouleau à pâtisserie rainuré.
    Faites frire sur une plaque chauffante jusqu'à ce que des taches brunes
    apparaissent.
    Retournez et faites frire l'autre côté.
```

Même si vous ne cuisinez pas, vous avez probablement reconnu qu'il s'agit d'une *recette*[1] : une liste d'ingrédients alimentaires suivie des instructions de préparation. Mais qu'est-ce que cela produit ? Ce sont des *lefse*, un mets norvégien qui ressemble à une tortilla (Figure 1-2). Badigeonnez de beurre et de confiture ou de ce que vous voulez, roulez-le et dégustez.

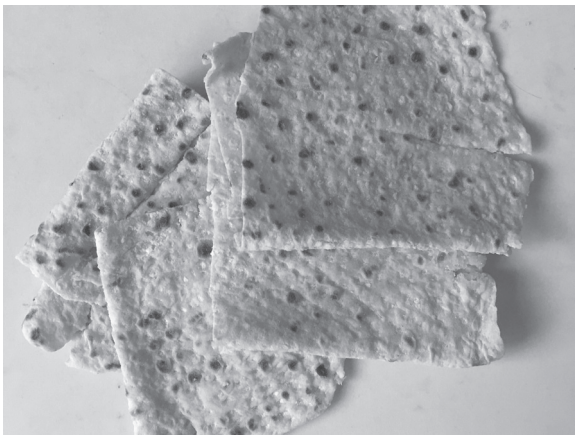

Figure 1-2. Des lefse norvégiens

1 Habituellement trouvé uniquement dans les livres de cuisine et les cozies.

Le patron de tricot et la recette culinaire partagent certaines caractéristiques :

- Un *vocabulaire* régulier de mots, d'abréviations et de symboles. Certains peuvent être familiers, d'autres mystérieux.
- Des règles sur ce qui peut être dit et comment, autrement dit une *syntaxe*.
- Une *séquence d'opérations* à effectuer dans l'ordre.
- Parfois, une répétition de certaines opérations (une *boucle*), comme la méthode pour faire frire chaque morceau de lefse.
- Parfois, une référence à une autre séquence d'opérations (en termes informatiques, une *fonction*). Dans la recette, vous devrez peut-être vous référer à une autre recette pour émincer les pommes de terre.
- Une connaissance présumée du *contexte*. La recette suppose que vous savez ce qu'est l'eau et comment la faire bouillir. Le modèle de tricot suppose que vous savez tricoter à l'endroit et à l'envers sans vous piquer trop souvent.
- Certaines *données* à utiliser, à créer ou à modifier : pommes de terre et fil.
- Les *outils* utilisés pour travailler avec les données : casseroles, mixeurs, fours, aiguilles à tricoter.
- Un *résultat* attendu. Dans nos exemples, quelque chose pour vos pieds et quelque chose pour votre estomac. Ne les mélangez pas.

Peu importe comment vous les appelez – idiomes, jargon, langages rudimentaires – vous en voyez des exemples partout. Le jargon fait gagner du temps aux personnes qui le connaissent, tout en mystifiant le reste d'entre nous. Essayez de déchiffrer une chronique de journal sur le bridge si vous ne jouez pas à ce jeu, ou un article scientifique si vous n'êtes pas un savant (ou même si vous l'êtes, mais dans un domaine différent).

Petits programmes

Vous retrouverez toutes ces idées dans des programmes informatiques, qui sont eux-mêmes composés avec des petits langages spécialisés pour que les humains puissent dire aux ordinateurs ce qu'ils doivent faire. J'ai utilisé le modèle de tricot et la recette pour montrer que la programmation n'est pas une chose mystérieuse. Il s'agit en grande partie d'apprendre les bons mots et les bonnes règles.

Maintenant, cela aide beaucoup s'il n'y a pas trop de mots et de règles, et si vous n'avez pas besoin d'en apprendre trop à la fois. Notre cerveau ne peut retenir trop de choses à la fois.

Voyons enfin un vrai programme informatique (Exemple 1-1). Que pensez-vous que cela exécute ?

Exemple 1-1. countdown.py

```
for countdown in 5, 4, 3, 2, 1, "hey!":
    print(countdown)
```

Si vous avez deviné que c'est un programme Python qui affiche les lignes

```
5
4
3
2
1
hey!
```

alors vous savez que Python est plus facile à apprendre qu'une recette ou un modèle de tricot. Et vous pouvez vous entraîner à écrire des programmes Python en tout confort et en toute sécurité dans votre bureau, à l'abri des dangers de l'eau bouillante et des aiguilles acérées.

Le programme Python contient des mots et des symboles spéciaux (`for`, `in`, `print`, des virgules, des deux-points, des parenthèses, etc.) qui sont des éléments importants de la *syntaxe* (ou les règles) du langage. La bonne nouvelle est que Python a une syntaxe plus agréable et moins prolixe que la plupart des langages informatiques. Il semble plus naturel, presque comme une recette.

L'exemple 1-2 est un autre petit programme Python ; il sélectionne un sortilège de Harry Potter dans une *liste* Python et l'affiche.

Exemple 1-2. spells.py

```
spells = [
    "Riddikulus!",
    "Wingardium Leviosa!",
    "Avada Kedavra!",
    "Expecto Patronum!",
    "Nox!",
    "Lumos!",
    ]
print(spells[3])
```

Les sortilèges individuels sont des *chaînes* Python (séquences de caractères de texte entre guillemets). Ils sont séparés par des virgules et inclus dans une *liste* Python définie par des crochets ([et]). Le mot `spells` est une *variable* qui donne un nom à la liste afin que nous puissions en faire quelque chose. Dans cet exemple, le programme affiche le quatrième sortilège :

```
Expecto patronum!
```

Pourquoi avons-nous écrit `3` si nous voulions le quatrième ? Une liste Python telle que la liste `spells` est une séquence de valeurs, accessibles par leur *position* par rapport au début de la liste. La première valeur se trouve à la position `0` et la quatrième valeur à la position `3`.

Les gens comptent habituellement à partir de 1, il peut donc sembler étrange de compter à partir de 0. Mais cela aide à penser en termes de décalage plutôt que de rang. Ceci est effectivement un exemple de la façon dont les programmes informatiques diffèrent parfois de l'usage d'une langue courante.

Les listes sont des *structures de données* très courantes en Python, et le chapitre 7 montrera comment les utiliser.

Le programme de l'Exemple 1-3 affiche une citation d'un des Trois Stooges (personnages comiques du cinéma américain), mais référencée par le nom de celui qui l'a prononcée plutôt que par sa position dans une liste.

Exemple 1-3. quotes.py

```
quotes = {
    "Moe": "Un type malin, hein ?",
    "Larry": "Aïe!",
    "Curly": "Nyuk nyuk!",
    }
stooge = "Curly"
print (stooge, "a dit :", quotes[stooge])
```

Si vous exécutiez ce petit programme, il afficherait ce qui suit :

```
Curly a dit : Nyuk nyuk!
```

quotes est une variable qui nomme un *dictionnaire* Python – une collection de *clés* uniques (dans cet exemple, le nom du Stooge) et des *valeurs* associées (ici, une citation notable de ce Stooge). À l'aide d'un dictionnaire, vous pouvez stocker et rechercher des éléments par nom, ce qui est souvent une alternative utile à une liste.

L'exemple de la variable `spells` utilise des crochets ([et]) pour créer une liste Python, et l'exemple de la variable quotes utilise des accolades ({ et }) pour créer un dictionnaire Python. De plus, un symbole deux-points (:) est utilisé pour associer chaque clé du dictionnaire à sa valeur. Vous en apprendrez plus sur les dictionnaires au chapitre 8.

J'espère que ce n'était pas trop de syntaxe à la fois. Dans les prochains chapitres, vous découvrirez progressivement davantage de ces petites règles.

Un programme plus important

Et maintenant passons à quelque chose de complètement différent : l'exemple 1-4 présente un programme Python effectuant une série de tâches plus complexes. Ne vous attendez pas à comprendre exactement comment le programme fonctionne ; c'est à cela que sert ce livre ! Le but est de vous présenter l'allure générale typique d'un programme Python non trivial. Si vous connaissez d'autres langages informatiques, vous pourrez apprécier en quoi Python s'en distingue. Même sans connaître encore Python, pouvez-vous comprendre grosso modo ce que fait chaque ligne avant de lire l'explication qui suit ce programme ?

Vous avez déjà vu des exemples de listes et de dictionnaires Python, et cet exemple ajoute quelques fonctionnalités supplémentaires.

Dans des versions antérieures de ce livre, l'exemple en question se connectait à un site Web YouTube et récupérait des informations sur ses vidéos les plus populaires, comme « Charlie Bit My Finger ». Cela a bien fonctionné jusqu'à la fin de la rédaction de la deuxième impression. C'est à ce moment-là que Google a laissé tomber ce service et que l'exemple de programme de sélection a cessé de fonctionner. Notre nouvel exemple 1-4 va sur un autre site qui devrait durer plus longtemps – la *Wayback Machine* sur l'Internet Archive (https://archive.org/), un service gratuit qui a sauvegardé des milliards de pages Web (et films, émissions de télévision, musiques, jeux et autres produits numériques) sur une période de 20 ans. Vous verrez d'autres exemples de telles *API Web* au chapitre 18.

Le programme vous demande de saisir une URL et une date. Ensuite, il demande à Wayback Machine s'il possède une copie de ce site Web à cette date. S'il en trouve une, il renvoie les informations au programme Python, qui écrit l'URL et l'affiche dans votre navigateur Web. Le but est de montrer comment Python gère une variété de tâches : récupérer ce que vous avez saisi, communiquer sur Internet avec un site Web, récupérer du contenu, en extraire une URL et faire afficher cette URL par votre navigateur Web.

Si nous récupérions une page Web normale remplie de texte au format HTML, nous aurions besoin de savoir comment l'afficher, ce qui représente beaucoup de travail que nous déléguons volontiers aux navigateurs Web. Nous pourrions également essayer d'extraire les parties que nous voulons (voir plus de détails sur l'*exploration web* au chapitre 18). L'un ou l'autre choix nécessiterait plus de travail et un programme plus long. Au lieu de cela, la Wayback Machine renvoie les données au format *JSON*. JSON (JavaScript Object Notation) est un format de texte lisible par les humains et qui décrit les types, les valeurs et l'ordre des données qu'il contient. C'est un autre type de langage rudimentaire qui est devenu un moyen populaire pour échanger des données entre différents langages et systèmes informatiques. Vous en apprendrez plus sur JSON au chapitre 12.

Les programmes Python peuvent traduire du texte JSON en *structures de données* Python – comme vous le verrez dans les prochains chapitres – comme si vous aviez écrit vous-même un programme pour les créer. Notre petit programme sélectionne juste un morceau (l'URL de l'ancienne page provenant du site Internet Archive). Encore une fois, c'est un programme Python complet que vous pouvez exécuter vous-même. Nous n'avons inclus qu'une vérification minimale des erreurs pour que l'exemple reste succinct. Les numéros de ligne ne font pas partie du programme ; ils sont inclus pour vous aider à suivre la description que nous fournissons à la suite de ce programme.

Exemple 1-4. archive.py

```
1 import webbrowser
2 import json
3 from urllib.request import urlopen
4
5 print("Recherchons un ancien site Web.")
6 site = input("Taper l'URL du site : ")
```

```
7 era = input("Taper l'année, le mois, le jour, comme 20150613 : ")
8 url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
9 response = urlopen(url)
10 contents = response.read()
11 text = contents.decode("utf-8")
12 data = json.loads(text)
13 try:
14     old_site = data["archived_snapshots"]["closest"]["url"]
15     print("Trouvé cette copie : ", old_site)
16     print("Elle devrait apparaître dans votre navigateur maintenant.")
17     webbrowser.open(old_site)
18 except:
19     print("Désolé, rien trouvé", site)
```

Ce petit programme Python fait beaucoup de choses en quelques lignes assez lisibles. Vous ne connaissez pas encore tous les termes mais vous les apprendrez dans les prochains chapitres. Voici ce qui se passe, ligne par ligne :

1. *Importer* (mettre à disposition de ce programme) tout le code du module de la *bibliothèque standard* Python appelé `webbrowser`.
2. Importer tout le code du module de la bibliothèque standard Python appelé `json`.
3. Importer uniquement la *fonction* `urlopen` à partir du module de la bibliothèque standard appelé `urllib.request.`
4. Une ligne vide pour une meilleure lisibilité.
5. Afficher un texte initial sur votre écran.
6. Poser une question à propos d'une URL, lire ce que vous saisissez et l'enregistrer dans une *variable* du programme appelée `site`.
7. Poser une autre question, cette fois en lisant une année, un mois et un jour, puis enregistrer la réponse dans une variable appelée `era`.
8. Construire une variable chaîne appelée `url` pour que la Wayback Machine recherche sa copie du site à la date que vous avez indiquée.
9. Se connecter au serveur Web à cette URL et demander un *service Web* particulier.
10. Obtenir les données de réponse et les affecter à la variable `contents`.
11. *Décoder* la variable `contents` en une chaîne de caractères au format JSON et l'affecter à la variable `text`.
12. Convertir `text` en `data` – des structures de données Python.
13. Vérification des erreurs : un bloc `try` pour essayer d'exécuter les quatre lignes suivantes et, en cas d'échec, exécuter la dernière ligne du programme (après le mot `except`).
14. Si nous avons obtenu une correspondance pour ce site et la date, extraire sa valeur d'un *dictionnaire* Python à trois niveaux. Noter que cette ligne et les deux suivantes sont en retrait. C'est ainsi que Python sait qu'elles vont avec la ligne `try` précédente.

15. Afficher l'URL que nous avons trouvée.
16. Afficher ce qui se passera après l'exécution de la ligne suivante.
17. Faire afficher l'URL que nous avons trouvée dans votre navigateur Web.
18. Si quelque chose a échoué dans les quatre lignes précédentes, Python passe à cet endroit.
19. En cas d'échec, afficher un message et le site que nous recherchions. Ceci est indenté car cela ne doit être exécuté que si la ligne d'exception précédente s'exécute.

Quand j'ai exécuté ceci dans une fenêtre de terminal, j'ai tapé l'URL d'un site et une date, et j'ai obtenu cette sortie de texte :

```
$ python archive.py
Recherchons un ancien site Web.
Taper l'URL du site : lolcats.com
Taper l'année, le mois, le jour, comme 20150613 : 20151022
Trouvé cette copie : http://web.archive.org/web/20151102055938/http://www.
lolcats.com/
```

Elle devrait apparaître dans votre navigateur maintenant.

Et la Figure 1-3 montre ce qui est apparu dans mon navigateur.

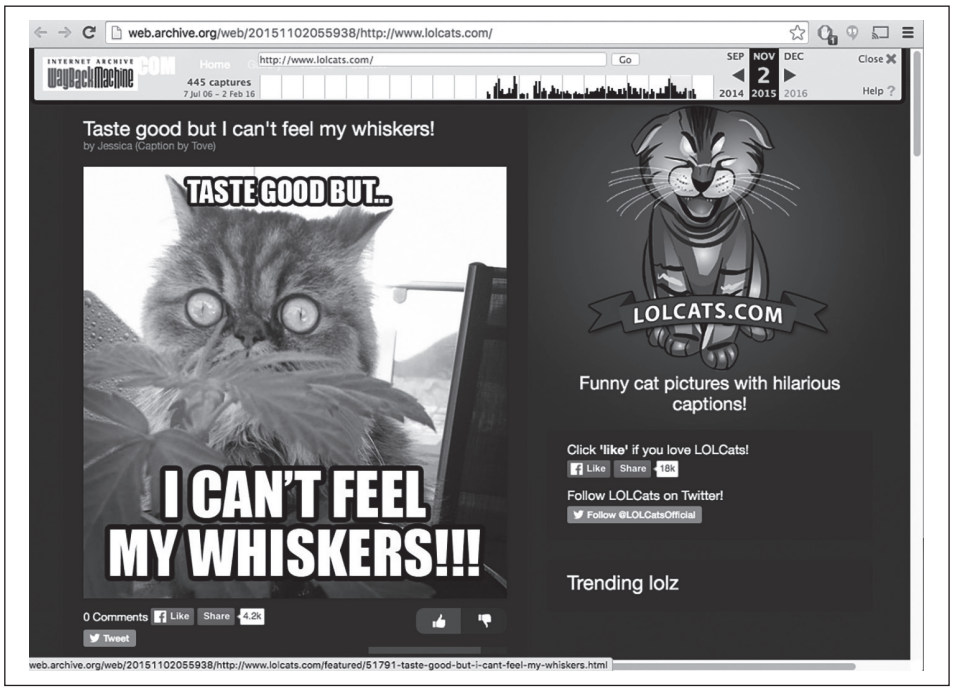

Figure 1-3. Récupéré sur la Wayback Machine

Dans l'exemple précédent, nous avons utilisé certains des modules de *bibliothèque standard* de Python (programmes qui sont inclus avec Python lors de son installation), mais ils n'ont rien de sacré.

Python possède une mine d'excellents logiciels tiers. L'exemple 1-5 est une réécriture du programme précédent qui accède au site Web Internet Archive en utilisant un package Python externe appelé `requests`.

Exemple 1-5. archive2.py

```
import webbrowser
import requests

print("Recherchons un ancien site Web.")
site = input("Taper l'URL du site : ")
era = input("Taper l'année, le mois, le jour, comme 20150613 : ")
url = "http://archive.org/wayback/available?url=%s&timestamp=%s" % (site, era)
response = requests.get(url)
data = response.json()
try:
    old_site = data["archived_snapshots"]["closest"]["url"]
    print("Trouvé cette copie : ", old_site)
    print("Elle devrait apparaître dans votre navigateur maintenant.")
    webbrowser.open(old_site)
except:
    print("Désolé, rien trouvé", site)
```

Cette nouvelle version est plus courte et me semble plus lisible pour la plupart des gens. Vous en apprendrez plus sur les requêtes au chapitre 18 et sur les packages Python créés en externe de manière générale au chapitre 11.

Python dans le monde réel

Alors, est-ce que l'apprentissage de Python vaut d'y consacrer du temps et des efforts ? Python existe depuis 1991 (plus ancien que Java, plus jeune que C) et figure systématiquement parmi les cinq langages informatiques les plus populaires. Des gens sont payés pour écrire des programmes Python – des choses sérieuses que vous utilisez tous les jours, comme Google, YouTube, Instagram, Netflix et Hulu. Je l'ai utilisé pour des applications de production dans de nombreux domaines. Python a une réputation de productivité qui séduit les organisations en évolution rapide.

Vous trouverez Python dans de nombreux environnements informatiques, notamment :

- La ligne de commande dans une fenêtre de moniteur ou de terminal
- Les interfaces utilisateur graphiques (GUI), y compris le Web
- Le Web, côté client et côté serveur
- Des serveurs backend prenant en charge de gros sites populaires
- Le *cloud* (serveurs gérés par des tiers)
- Des appareils mobiles
- Des appareils intégrés

Les programmes Python vont de *scripts* individuels, tels que ceux que vous avez vus jusqu'à présent dans ce chapitre, à des systèmes comportant des millions de lignes.

L'enquête 2018 des développeurs Python (Python Developers' Survey) fournit des chiffres et des graphiques sur la place actuelle de Python dans le monde informatique.

Nous examinerons ses utilisations dans le cadre des sites Web, de l'administration système ou de la manipulation de données. Dans les derniers chapitres, nous verrons des utilisations spécifiques de Python dans le domaine des arts, de la science et des affaires.

Python contre le langage de la Planète X

Comment Python se compare-t-il aux autres langages ? Où et quand choisiriez-vous l'un plutôt que l'autre ? Dans cette section, je montre des exemples de code écrits dans d'autres langages, juste pour que vous puissiez voir à quoi ressemble la concurrence. Vous *n'êtes pas* censé les comprendre si vous n'avez pas travaillé avec eux. (Au moment où vous arriverez à l'exemple final de Python, vous serez même peut-être soulagé de ne pas avoir à travailler avec certains d'entre eux.)

Chaque programme est censé afficher un numéro et en dire un peu plus sur le langage.

Si vous utilisez un terminal ou une fenêtre de terminal, le programme qui lit ce que vous tapez, l'exécute et affiche les résultats est appelé le *shell*. Le shell de Windows est appelé `cmd` ; il exécute des fichiers de commandes avec le suffixe `.bat`. Linux et d'autres systèmes de type Unix (y compris macOS) ont de nombreux programmes shell. Le plus populaire est appelé `bash` ou `sh`. Le shell a une logique simple et des capacités rudimentaires, comme l'expansion de symboles génériques tels que * dans les noms de fichiers. Vous pouvez enregistrer des commandes dans des fichiers appelés *scripts shell* et les exécuter ultérieurement. Ce sont peut-être les premiers programmes que vous avez rencontrés en tant que programmeur. Le problème est que les scripts shell ne s'étendent pas bien au-delà de quelques centaines de lignes, et ils sont beaucoup plus lents que les autres langages. L'extrait suivant montre un petit programme shell :

```
#! /bin/sh
language=0
echo "Language $language: I am the shell. So there."
```

Si vous l'avez enregistré dans un fichier sous le nom `test.sh` et que vous l'avez exécuté avec `sh test.sh`, vous verrez ce qui suit sur votre écran de terminal :

```
Language 0: I am the shell. So there.
```

Les anciens piliers C et C++ sont des langages de bas niveau, utilisés lorsque la vitesse est primordiale. Votre système d'exploitation et nombre de ses programmes (y compris le programme python sur votre ordinateur) sont probablement écrits en C ou C++.

Tous les deux sont plus difficiles à apprendre et à entretenir. Vous devez veiller à de nombreux détails tels que la *gestion de la mémoire*, ce qui peut entraîner des plantages et des problèmes difficiles à diagnostiquer. Voici un petit programme écrit en C :

```
   #include <stdio.h>
    int main(int argc, char *argv[]) {
       int language = 1;
       printf("Language %d: I am C! See? Si!\n", language);
       return 0;
}
```

C++ a des similitudes avec le C mais a aussi développé des fonctionnalités propres :

```
    #include <iostream>
    using namespace std;
    int main() {
       int language = 2;
       cout << "Language " << language << \
           ": I am C++ ! Pay no attention to my little brother !" << \
           endl;
       return(0);
}
```

Java et C # sont des successeurs de C et C ++ qui évitent certains des problèmes de leurs ancêtres – en particulier la gestion de la mémoire – mais peuvent être quelque peu verbeux. L'exemple qui suit montre un peu de Java :

```
    public class Anecdote {
       public static void main (String[] args) {
           int language = 3;
           System.out.format("Language %d: I am Java ! So there !\n", language);
       }
}
```

Si vous n'avez pas écrit de programmes dans un de ces langages, vous vous demanderez peut-être : pourquoi tout ce *bazar* ? Nous voulions seulement afficher une simple ligne de texte. Certains langages véhiculent un lourd bagage syntaxique. Vous en apprendrez plus à ce sujet au chapitre 2.

C, C ++ et Java sont des exemples de *langages statiques*. Ils vous obligent à spécifier certains détails de bas niveau tels que le type des données pour l'ordinateur. L'annexe A montre comment un type de données, comme un entier, utilise un nombre spécifique de bits sur votre ordinateur et ne peut faire que des opérations entières. En revanche, les *langages dynamiques* (également appelés *langages de script*) ne vous obligent pas à déclarer de type pour les variables avant de les utiliser.

Un langage dynamique polyvalent, utilisé pendant de nombreuses années, a été Perl. Perl est très puissant et possède de nombreuses bibliothèques. Pourtant, sa syntaxe peut être gênante et le langage semble avoir perdu de son élan au cours des dernières années au profit de Python et Ruby. L'exemple qui suit vous régalera d'un bon mot Perl :

```
    my $language = 4;
    print "Language $language: I am Perl, the camel of languages.\n";
```

Ruby est un langage plus récent. Il emprunte un peu à Perl et est populaire principalement à cause de *Ruby on Rails*, un environnement pour le développement Web. Il est utilisé dans de nombreux domaines identiques à Python, et le choix de l'un ou de l'autre peut se résumer à une question de goût ou de bibliothèques disponibles pour votre application particulière. Voici un extrait de code Ruby :

```
language = 5
puts "Language #{language}: I am Ruby, ready and aglow."
```

PHP, que vous pouvez voir dans l'exemple suivant, est très populaire pour le développement Web car il permet de combiner facilement HTML et code. Cependant, le langage PHP lui-même a un certain nombre de pièges, et PHP n'est pas devenu un langage généraliste en dehors du Web. Voici à quoi il ressemble :

```
<?PHP
$language = 6;
echo "Language $language: I am PHP, a language and palindrome.\n";
?>
```

Go (ou Golang, si vous essayez de faire une recherche Google) est un langage récent qui tente d'être à la fois efficace et convivial :

```
package main
import "fmt"
func main() {
  language := 7
  fmt.Printf("Language %d: Hey, ho, let's Go!\n", language)
}
```

Une autre alternative moderne à C et C ++ est Rust :

```
fn main() {
    println!("Language {}: Rust here!", 8)
}
```

Qu'est-ce qui nous reste ? Ah oui, Python :

```
language = 9
print(f"Language {language}: I am Python. What's for supper?")
```

Pourquoi Python ?

Une des raisons, pas nécessairement la plus importante, est sa popularité. Dans des mesures variées, Python est :

- Le langage de programmation majeur dont la croissance est la plus rapide, comme vous pouvez le voir sur la figure 1-4.
- Les éditeurs de l'indice TIOBE de juin 2019 affirment : « Ce mois-ci, Python a de nouveau atteint un niveau record de l'indice TIOBE de 8,5 %.

Si Python peut maintenir ce rythme, il remplacera probablement C et Java dans 3 à 4 ans, devenant ainsi le langage de programmation le plus populaire au monde. »

- Langage de programmation de l'année 2018 (TIOBE), et premier au classement de IEEE Spectrum et PyPL.
- Le langage le plus populaire pour les cours d'introduction à l'informatique dans les grandes universités américaines.
- Le langage officiel d'enseignement dans les lycées en France.

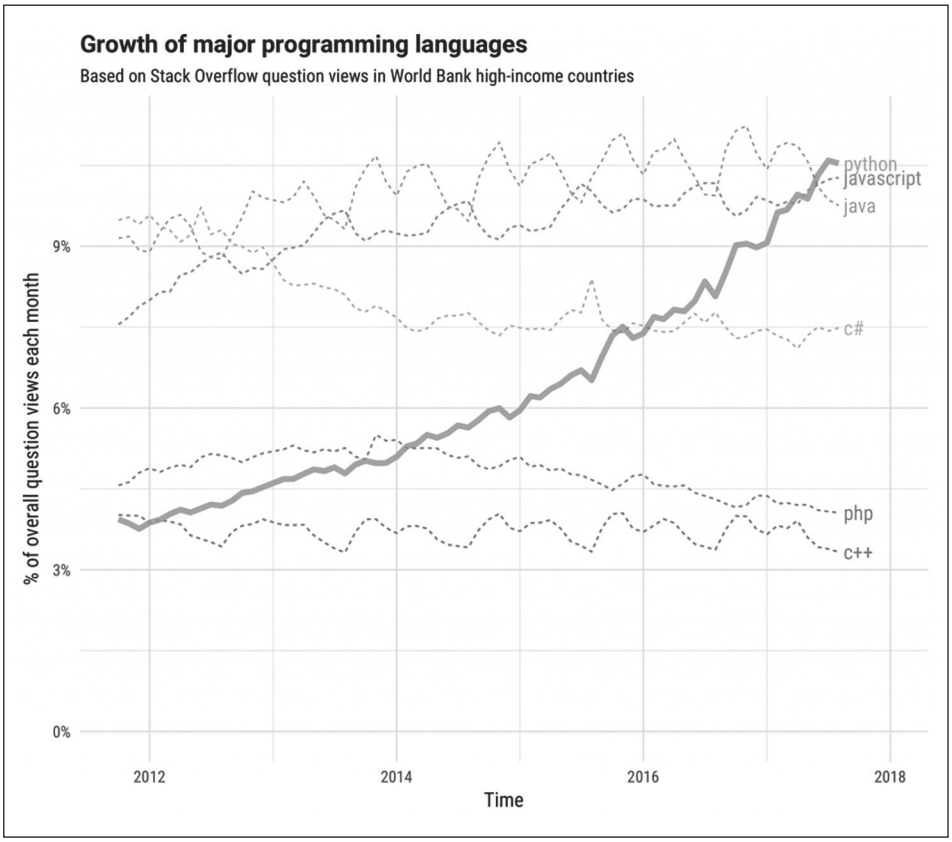

Figure 1-4. La croissance de Python l'emporte sur les principaux langages de programmation.

Plus récemment, il est devenu extrêmement populaire dans les mondes de la science des données et de l'apprentissage automatique. Si vous souhaitez décrocher un emploi de programmation bien rémunéré dans un domaine intéressant, Python est maintenant un bon choix. Et si vous recrutez, il existe un nombre croissant de développeurs Python expérimentés.

Mais *pourquoi* est-il populaire ? Les langages de programmation ne font pas exactement preuve de charisme. Quelles sont les raisons sous-jacentes ?

Python est un bon langage généraliste de haut niveau. Son design le rend très *lisible*, ce qui est plus important qu'il n'y paraît. Chaque programme informatique est écrit une seule fois, mais est lu et révisé plusieurs fois, souvent par de nombreuses personnes. Être lisible facilite aussi l'apprentissage et la mémorisation ce qui, en conséquence, le rend plus *facile à écrire*. Comparé à d'autres langages populaires, Python a une courbe d'apprentissage douce qui vous rend productif très tôt, ce qui n'exclut pas des aspects plus profonds que vous pouvez explorer à mesure que vous gagnez en expertise.

La relative concision de Python vous permet d'écrire des programmes plus courts que leurs équivalents dans un langage statique. Des études ont montré que les programmeurs ont tendance à produire à peu près le même nombre de lignes de code par jour – quelle que soit le langage – donc, réduire de moitié les lignes de code double votre productivité automatiquement. Python est l'arme, pas si secrète, de nombreuses entreprises qui pensent que c'est important.

Et bien sûr, Python est libre d'accès, comme quand on parle de bière (prix) ou de parole (liberté). Écrivez tout ce que vous voulez avec Python et utilisez-le n'importe où, librement. Personne ne peut lire votre programme Python et dire : « C'est un joli petit programme que vous avez là. Ce serait dommage si quelque chose lui arrivait. »

Python fonctionne presque partout et a des « piles incluses » – une vaste collection de logiciels utiles fournis dans sa bibliothèque standard. Ce livre présente de nombreux exemples utilisant la bibliothèque standard mais aussi du code Python externe utile.

Mais, la meilleure raison pour utiliser Python est peut-être surprenante : les gens *aiment* généralement programmer dans ce langage et ne voient pas ça comme un mal nécessaire pour accomplir des choses. Il ne les gêne pas dans leur travail. Une citation fréquemment entendue est qu'il « s'accorde bien avec votre cerveau ». Souvent, les développeurs diront qu'il leur manque un concept Python lorsqu'ils doivent travailler dans un autre langage. Et c'est ce qui distingue Python de la plupart de ses pairs.

Pourquoi pas Python ?

Python n'est pas le meilleur langage pour toutes les situations.

Il n'est pas installé partout par défaut. L'annexe B vous montre comment installer Python si vous ne l'avez pas déjà sur votre ordinateur.

Il est assez rapide pour la plupart des applications, mais il n'est peut-être pas assez rapide pour certaines des applications les plus exigeantes. Si votre programme passe la plupart de son temps à calculer des choses (techniquement très *dépendant du processeur*), un programme écrit en C, C++, C#, Java, Rust ou Go fonctionnera généralement plus rapidement que son équivalent Python. Mais pas toujours !

Voici quelques solutions :

- Parfois, un meilleur *algorithme* (une solution par étapes) en Python surpasse un algorithme inefficace en C. La plus grande vitesse de développement en Python vous donne plus de temps pour expérimenter des alternatives.
- Dans de nombreuses applications (notamment, le Web), un programme se tourne les pouces en attendant une réponse d'un serveur sur un réseau. La CPU (unité centrale de traitement, la *puce* de l'ordinateur qui fait tous les calculs) est à peine impliquée ; par conséquent, les temps complets d'exécution avec les programmes statiques ou dynamiques seront proches.
- L'interpréteur Python standard est écrit en C et peut être étendu avec du code C. J'en parle un peu au chapitre 19.
- Les interpréteurs Python sont de plus en plus rapides. Java était terriblement lent à ses débuts, et beaucoup de recherches et d'argent ont été nécessaires pour l'accélérer. Python n'appartenant pas à une société, ses améliorations ont donc été plus graduelles. Dans la section « PyPy » à la page 369, je parle du projet *PyPy* et de ses implications.
- Vous avez peut-être une application extrêmement exigeante, et quoi que vous fassiez, Python ne répond pas à vos besoins. Les alternatives habituelles sont C, C ++ et Java. Go (qui ressemble à Python mais fonctionne comme C) ou Rust pourraient également mériter le détour.

Python 2 contre Python 3

Une petite complication est qu'il existe deux versions de Python. Python 2 existe depuis toujours et est préinstallé sur les ordinateurs Linux et Apple. C'est un excellent langage, mais rien n'est parfait. Dans les langages informatiques, comme dans de nombreux autres domaines, certaines erreurs sont esthétiques et faciles à corriger, tandis que d'autres sont difficiles. Certains correctifs fondamentaux sont *incompatibles* : les nouveaux programmes écrits avec eux ne fonctionneront pas avec l'ancien système Python, et les anciens programmes écrits avant ce correctif ne fonctionneront plus avec le nouveau système.

Le créateur de Python (Guido van Rossum) et d'autres ont décidé en 2008 de regrouper les correctifs fondamentaux et de les présenter sous le nom de Python 3. Python 2 est le passé et Python 3 est l'avenir. La version finale de Python 2 est la 2.7, et elle continuera d'exister encore quelque temps, mais c'est la fin de cette série ; il n'y aura pas de Python 2.8. La fin du support du langage Python 2 date de janvier 2020. Les correctifs de sécurité et autres ne sont plus apportés, et de nombreux packages Python importants devraient abandonner le support de Python 2 après cette date. Les systèmes d'exploitation abandonneront bientôt Python 2 ou feront de 3 leur nouveau réglage par défaut. La conversion du logiciel Python populaire vers Python 3 a été graduelle, mais nous avons maintenant bien dépassé le point de bascule. Tous les nouveaux développements se font désormais en Python 3.

Ce livre traite de Python 3. Il ressemble beaucoup à Python 2. Le changement le plus évident est que la commande print est devenue une fonction en Python 3, vous devez donc l'appeler avec des parenthèses entourant ses arguments. Le changement le plus important est la gestion des caractères *Unicode*, qui est traitée au chapitre 12. Je signalerai d'autres différences significatives au fur et à mesure qu'elles apparaîtront.

Installer Python

Plutôt que d'encombrer ce chapitre, tous les détails sur l'installation de Python 3 se trouvent dans l'annexe B. Si vous n'avez pas Python 3 ou si vous n'en êtes pas sûr, allez-y et voyez ce que vous devez faire pour votre ordinateur. Cela peut parfois faire grincer des dents mais vous n'aurez besoin de le faire qu'une seule fois.

Exécuter Python

Après avoir installé une copie de travail de Python 3, vous pouvez l'utiliser pour exécuter les programmes Python de ce livre ainsi que votre propre code Python. Comment exécuter concrètement un programme Python ? Il existe deux manières principales :

- L'*interpréteur interactif* intégré de Python (également appelé son *shell*) est le moyen le plus simple d'expérimenter de petits programmes. Vous tapez les commandes ligne par ligne et voyez immédiatement les résultats. Grâce au couplage étroit entre la saisie et le résultat, vous pouvez expérimenter plus rapidement. J'utiliserai l'interpréteur interactif pour montrer les fonctionnalités du langage, et vous pouvez taper les mêmes commandes dans votre propre environnement Python.
- Pour tout le reste, stockez vos programmes Python dans des fichiers texte, normalement avec l'extension *.py*, et exécutez-les en tapant `python` suivi du nom de ces fichiers.

Essayons les deux méthodes maintenant.

Utilisation de l'interpréteur interactif

La plupart des exemples de code de ce livre utilisent l'interpréteur interactif intégré. Si vous tapez les commandes que vous voyez dans les exemples et que vous obtenez les mêmes résultats, vous saurez que vous êtes sur la bonne voie.

Vous démarrez l'interpréteur en tapant simplement le nom du programme Python principal sur votre ordinateur : il devrait s'appeler `python` ou `python3` ou quelque chose de similaire. Pour le reste de ce livre, nous supposons qu'il s'appelle `python` ; si le vôtre a un nom différent, saisissez ce nom partout où vous voyez `python` dans un exemple de code.

L'interpréteur interactif fonctionne presque exactement de la même manière que Python fonctionne sur les fichiers, à une exception près : lorsque vous tapez une commande qui a une valeur, l'interpréteur interactif affiche sa valeur automatiquement pour vous. Cela ne fait pas partie du langage Python, c'est simplement une fonctionnalité de l'interpréteur interactif pour vous éviter de taper `print()` tout le temps.

Par exemple, si vous démarrez Python et tapez le nombre `27` dans l'interpréteur, il sera affiché en écho sur votre terminal (si vous écrivez une ligne avec le nombre `27` dans un fichier, Python ne sera pas contrarié, mais vous ne verrez rien s'afficher lorsque vous exécuterez le programme) :

```
$ python
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 27
27
```

Dans l'exemple précédent, `$` représente *l'invite* de la fenêtre du terminal vous demandant de taper une commande telle que python. Nous utilisons ce symbole pour les exemples de code de ce livre, bien que votre invite puisse être différente.

À propos, `print()` fonctionne également dans l'interpréteur chaque fois que vous souhaitez faire afficher quelque chose :

```
>>> print(27)
27
```

Si vous avez essayé ces exemples avec l'interpréteur interactif et que vous avez obtenu les mêmes résultats, vous venez d'exécuter du code Python réel (bien que succinct). Dans les prochains chapitres, vous passerez des commandes sur une ligne à des programmes Python plus longs.

Utilisation de fichiers Python

Si vous écrivez `27` dans un fichier individuel et que vous l'exécutez via Python, cela fonctionnera, mais cela n'affichera rien. Dans les programmes Python non interactifs normaux, vous devez appeler la fonction `print` pour faire afficher des éléments :

```
print(27)
```

Créons un fichier script Python et exécutons-le :

1. Ouvrez votre éditeur de texte.

2. Tapez une ligne avec l'instruction `print(27)` telle quelle.

3. Enregistrez dans un fichier appelé *test.py*. Assurez-vous d'enregistrer sous forme de texte brut et non pas dans un format « riche » tel que RTF ou Word. Vous n'avez pas l'obligation d'utiliser le suffixe *.py* pour vos fichiers de programme Python mais cela aide à se souvenir de quoi il s'agit.

4. Si vous utilisez une interface graphique, comme presque tout le monde, ouvrez une fenêtre de terminal.[2]

2 Si vous n'êtes pas sûr de ce que cela signifie, reportez-vous à l'annexe B pour plus de détails sur les différents systèmes d'exploitation.

5. Exécutez votre programme en tapant ce qui suit :

```
$ python test.py
```

Vous devriez voir une seule ligne de sortie :

```
27
```

Cela a-t-il fonctionné ? Si c'est le cas, félicitations pour l'exécution de votre premier programme Python autonome.

Comment aller plus loin ?

Vous allez taper des commandes sur une installation Python réelle et elles doivent suivre la syntaxe légale de Python. Plutôt que de déverser toutes les règles de syntaxe en même temps, nous les parcourrons progressivement dans les prochains chapitres.

La méthode de base pour développer des programmes Python consiste à utiliser un éditeur de texte et une fenêtre de terminal. J'utilise des affichages en texte brut dans ce livre, montrant parfois des sessions de terminal interactives et parfois des morceaux de fichiers Python. Vous devez savoir qu'il existe également beaucoup de bons *environnements de développement intégrés* (IDE) pour Python. Ceux-ci peuvent comporter des interfaces graphiques avec des capacités avancées pour l'édition de texte et l'affichage d'aide. Vous pouvez en apprendre davantage sur certains d'entre eux au chapitre 19.

Votre moment de zénitude

Chaque langage informatique a son propre style. Dans la préface, j'ai mentionné qu'il existe souvent une manière *pythonique* de s'exprimer. Un petit verset libre exprimant succinctement la philosophie Python se trouve incorporé au programme (pour autant que je sache, Python est le seul langage à inclure un tel œuf de Pâques). Tapez simplement `import this` dans votre interpréteur interactif, puis appuyez sur la touche Entrée chaque fois que vous ressentirez le besoin d'un instant de zénitude (le texte s'affiche en anglais mais nous en donnons ici la traduction) :

```
>>> import this
Le Zen de Python, par Tim Peters

Le beau vaut mieux que le laid.
L'explicite vaut mieux que l'implicite.
Le simple vaut mieux que le complexe.
Complexe vaut mieux que compliqué.
À plat vaut mieux qu'imbriqué.
Clairsemé vaut mieux que dense.
La lisibilité compte.
Les cas particuliers ne le sont pas assez pour enfreindre les règles.
Pourtant l'aspect pratique l'emporte sur la pureté.
Les erreurs ne doivent jamais passer en silence.
À moins d'être explicitement réduites au silence.
Face à l'ambiguïté, refusez la tentation de deviner.
Il devrait y avoir une – et de préférence une seule – façon évidente de le faire.
Bien que cette manière ne soit pas évidente au début, sauf si vous êtes
hollandais.
```

```
Maintenant vaut mieux que jamais.
Bien que jamais vaille souvent mieux que *tout de suite*.
Si l'implémentation est difficile à expliquer, c'est que l'idée est mauvaise.
Si la mise en œuvre est facile à expliquer, cela peut être une bonne idée.
Les espaces de noms sont une excellente idée – utilisons-les plus!
```

J'évoquerai des exemples de ces pensées tout au long du livre.

À venir

Le chapitre suivant traite des types de données et des variables Python. Cela vous préparera aux chapitres suivants qui détaillent les types de données et les structures de code.

À faire

Ce chapitre était une introduction au langage Python : ce qu'il fait, à quoi il ressemble et où il se situe dans le monde de la programmation. À la fin de chaque chapitre, je suggère quelques mini-projets pour vous aider à vous souvenir de ce que vous venez de lire et à vous préparer à ce qui va arriver.

1.1 Si vous n'avez pas encore installé Python 3 sur votre ordinateur, faites-le maintenant. Lisez l'annexe B pour plus de détails sur votre système informatique.

1.2 Démarrez l'interpréteur interactif Python 3. Encore une fois, les détails sont dans l'annexe B. Il devrait afficher quelques lignes à propos de lui-même, puis une seule ligne commençant par >>>. C'est votre invite à taper des commandes Python.

1.3 Exercez-vous un peu avec l'interpréteur. Utilisez-le comme une calculatrice et tapez ceci : `8 * 9`. Appuyez sur la touche Entrée pour voir le résultat. Python devrait afficher `72`.

1.4 Tapez le nombre `47` et appuyez sur la touche Entrée. Vous a-t-il affiché `47` sur la ligne suivante ?

1.5 Maintenant, tapez `print(47)` et appuyez sur `Entrée`. Est-ce que cela a également affiché `47` sur la ligne suivante ?

CHAPITRE 2

Données : Types, Valeurs, Variables et Noms

> Une bonne renommée est préférable à de grandes richesses.
>
> —Proverbes 22:1

Sous le capot, tout dans votre ordinateur n'est qu'une séquence de *bits* (voir Annexe A). L'une des idées de l'informatique est que nous pouvons interpréter ces bits comme nous le voulons – comme des données de différentes tailles et différents types (nombres, caractères de texte) ou même comme du code informatique lui-même. Nous utilisons Python pour définir des segments de ces bits à des fins diverses, et pour les faire circuler vers ou depuis l'unité centrale (CPU).

Nous commencerons par les *types* de données de Python et les *valeurs* qu'ils peuvent contenir. Ensuite, nous verrons comment représenter les données sous forme de valeurs *littérales* et de *variables*.

Les données Python sont des objets

Vous pouvez visualiser la mémoire de votre ordinateur sous la forme d'une longue série d'étagères. Chaque emplacement sur l'une de ces étagères de mémoire a une largeur d'un octet (huit bits) et les emplacements sont numérotés de 0 (le premier) à la fin. Les ordinateurs modernes ont des milliards d'octets de mémoire (gigaoctets), de sorte que les étagères rempliraient un immense entrepôt imaginaire.

Un programme Python peut accéder à une partie de la mémoire de votre ordinateur grâce au système d'exploitation. Cette mémoire est utilisée pour le code du programme lui-même et pour les données qu'il manipule. Le système d'exploitation garantit que le programme ne peut pas aller lire ou écrire dans d'autres emplacements de mémoire sans en obtenir l'autorisation.

Les programmes gardent une trace de *l'emplacement* (adresse mémoire) où leurs bits résident et de ce qu'ils *représentent* (type de données). Pour votre ordinateur, ce ne sont que des séquences de bits. Les mêmes bits signifient des choses différentes, selon le type que nous leur attribuons. Le même segment de bits peut représenter l'entier `65` ou le caractère de texte `A`.

Différents types peuvent occuper des nombres différents de bits. Lorsque vous entendez parler d'une « machine 64 bits », cela signifie qu'un entier utilise 64 bits (8 octets) sur cette machine.

Certains langages inscrivent et récupèrent ces valeurs brutes en mémoire, en tenant compte de leurs tailles et de leurs types. Au lieu de gérer directement ces valeurs de données brutes, Python encapsule dans la mémoire chaque valeur de données (booléens, entiers, flottants, chaînes, et même de grandes structures de données, fonctions et programmes) sous forme *d'objet*. Il y a tout un chapitre (chapitre 10) consacré à la manière de définir vos propres objets en Python. Pour l'instant, nous ne parlons que des objets prédéfinis qui gèrent les types de données de base.

En utilisant l'analogie des étagères de mémoire, vous pouvez considérer les objets comme des boîtes de taille variable occupant des espaces sur ces étagères, comme illustré sur la Figure 2-1. Python fabrique ces boîtes d'objets, les place dans des espaces vides sur les étagères et les supprime lorsqu'elles ne sont plus utilisées.

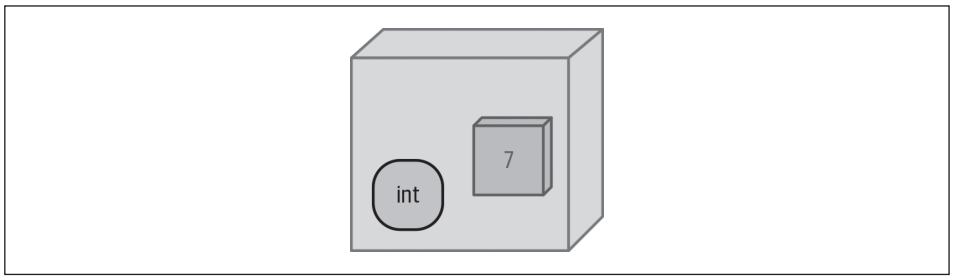

Figure 2-1. Un objet est comme une boîte ; celui-ci correspond à un entier de valeur 7.

En Python, un objet est un bloc de données contenant au moins les éléments suivants :

- Un *type* qui définit ce qu'il peut faire (voir la section suivante).
- Un *identifiant* unique pour le distinguer des autres objets.
- Une *valeur* cohérente avec son type.
- Un *compteur de références* (en anglais *reference count*) qui traque le nombre de fois que cet objet est utilisé.

Son identifiant est unique et désigne son emplacement sur l'étagère. Son type est comme un tampon d'usine sur la boîte, indiquant ce qu'il peut faire. Si un objet Python est un entier, il a le type int et pourrait (entre autres choses, comme on le verra au chapitre 3) être ajouté à un autre int. Si nous imaginons la boîte comme étant en plastique transparent, nous pouvons voir la valeur à l'intérieur. Vous apprendrez à utiliser le compteur de références d'ici quelques sections lorsque nous parlerons de variables et de noms.

Les types

Le tableau 2-1 présente les types de données de base de Python. La deuxième colonne (Type) contient le nom Python de ce type. La troisième colonne (Modifiable ?) indique si la valeur peut être modifiée après sa création, ce que j'expliquerai plus en détail dans la section suivante. Les exemples montrent un ou plusieurs éléments littéraux de ce type. Et la dernière colonne (Chapitre) vous renvoie au chapitre de ce livre où ce type est examiné en détail.

Table 2-1. Types de données de Python

Nom	Type	Modifiable ?	Exemples	Chapitre
Booléen	`bool`	non	`True, False`	Chapitre 3
Entier	`int`	non	`47, 25 000, 25_000`	Chapitre 3
Virgule flottante	`float`	non	`3.14, 2.7e5`	Chapitre 3
Complexe	`complex`	non	`3j,5 + 9j`	Chapitre 22
Chaîne de texte	`str`	non	`'alas', "alack", '''a verse attack'''`	Chapitre 5
Liste	`list`	oui	`['Winken', 'Blinken', 'Nod']`	Chapitre 7
Tuple	`tuple`	non	`(2, 4, 8)`	Chapitre 7
Octets	`bytes`	non	`b'ab\xff'`	Chapitre 12
Table d'octets	`bytearray`	oui	`bytearray(…)`	Chapitre 12
Ensemble	`set`	oui	`set([3, 5, 7])`	Chapitre 8
Ensemble figé	`frozenset`	non	`frozenset(['Elsa', 'Otto'])`	Chapitre 8
Dictionnaire	`dict`	oui	`{'game': 'bingo', 'dog': 'dingo', 'drummer': 'Ringo'}`	Chapitre 8

Après les chapitres sur ces types de données de base, vous verrez au chapitre 10 comment créer de nouveaux types.

Mutabilité

> Rien ne dure hormis la mutabilité.
>
> —Percy Shelley

Le type détermine également si la *valeur* des données contenues dans la boîte peut être modifiée (*mutable*) ou demeurer constante (*immutable*). Considérez un objet immuable comme une boîte scellée, mais avec des côtés clairs, comme la figure 2-1 ; vous pouvez voir la valeur, mais vous ne pouvez pas la modifier. Par la même analogie, un objet modifiable (mutable) est comme une boîte avec un couvercle : non seulement vous pouvez voir la valeur à l'intérieur, vous pouvez également la changer ; cependant, vous ne pouvez pas changer son type.

Python est un langage *fortement typé*, ce qui signifie que le type d'un objet ne change pas, même si sa valeur est modifiable (Figure 2-2).

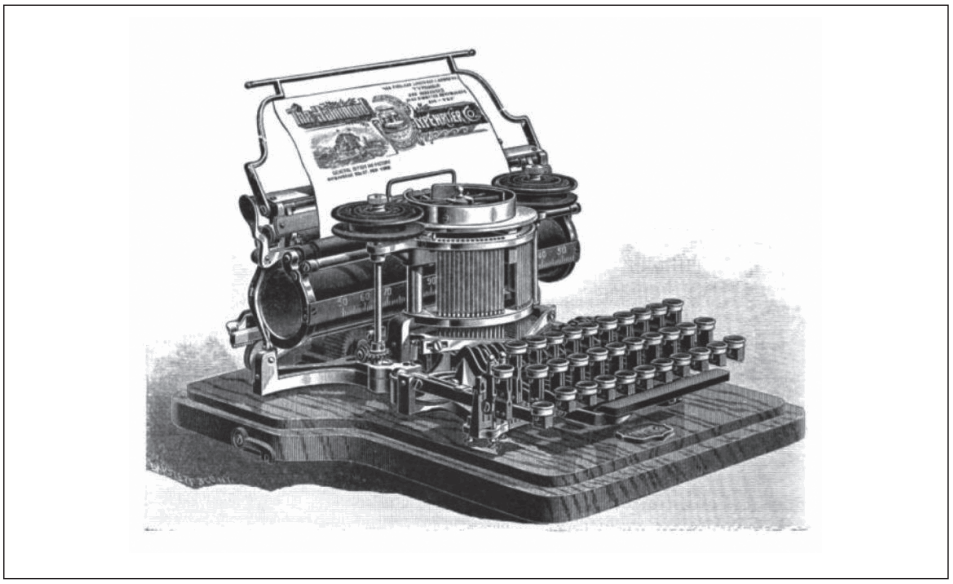

Figure 2-2. Fortement typé ne signifie pas une frappe forte sur les touches.

Valeurs littérales

Il existe deux façons de spécifier des valeurs de données en Python :

- *Littérale*
- *Variable*

Dans les chapitres suivants, vous verrez en détail la façon de spécifier des valeurs littérales pour différents types de données : les entiers sont une séquence de chiffres, les flottants contiennent un point décimal, les chaînes de texte sont entourées de guillemets, etc. Mais, dans le reste de ce chapitre, pour épargner nos doigts, nos exemples n'utiliseront que des entiers décimaux courts et une ou deux listes Python. Les entiers décimaux sont comme des entiers en mathématiques : une séquence de chiffres de `0` à `9`. Il y a quelques aspects supplémentaires concernant les entiers (comme les signes et les bases non décimales) que nous examinerons au chapitre 3.

Variables

Maintenant, nous sommes arrivés à un concept clé dans les langages informatiques.

Python, comme la plupart des langages informatiques, vous permet de définir des *variables*, des noms de valeurs dans la mémoire de votre ordinateur que vous souhaitez utiliser dans un programme.

Les noms de variables de Python doivent respecter quelques règles :

- Ils ne peuvent contenir que les caractères suivants :
  — Lettres minuscules (de `a` à `z`)
  — Lettres majuscules (`A` à `Z`)
  — Chiffres (`0` à `9`)
  — Soulignement (_)
- Ils sont *sensibles à la casse* : `thing`, `Thing` et `THING` sont des noms différents.
- Ils doivent commencer par une lettre ou un trait de soulignement, pas un chiffre.
- Les noms qui commencent par un trait de soulignement sont traités spécialement (ce que nous verrons au chapitre 9).
- Ils ne peuvent pas être l'un des *mots réservés* de Python (également appelés *mots-clés*). Les mots réservés[1] 1 sont :

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Dans un programme Python, vous pouvez trouver les mots réservés avec les instructions

```
>>> help("keywords")
```

ou :

```
>>> import keyword
>>> keyword.kwlist
```

Voici quelques noms valides :

- `a`
- `a1`
- `a_b_c___95`
- `_abc`
- `_1a`

1 `async` et `await` sont apparus dans Python 3.7.

Les noms suivants au contraire ne sont pas valides :

- `1`
- `1a`
- `1_`
- `name!`
- `another-name`

Affectation

En Python, vous utilisez le signe `=` pour affecter une valeur à une variable.

Nous avons tous appris en arithmétique à l'école primaire que `=` signifie *égal à*. Alors pourquoi de nombreux langages informatiques, y compris Python, utilisent-ils le signe `=` pour l'affectation ? L'une des raisons est que les claviers standards manquent d'alternatives logiques telles qu'une touche pour une flèche pointant vers la gauche, et le signe `=` ne semble pas trop déroutant. En outre, dans les programmes informatiques, vous utilisez bien plus souvent l'affectation que vous ne testez l'égalité.

Les programmes ne sont *pas* comme l'algèbre. Lorsque vous avez appris les mathématiques à l'école, vous avez vu des équations comme celle-ci :

```
y = x + 12
```

Vous calculeriez l'expression en injectant une valeur pour `x`. Si vous avez donné à `x` la valeur `5`, alors `5 + 12` vaut `17` et donc la valeur de `y` serait `17`. Injectez la valeur `6` pour `x` et alors vous obtiendrez `18` pour `y`, et ainsi de suite.

Les lignes de programme informatique peuvent ressembler à des équations, mais leur signification est différente. En Python et dans d'autres langages informatiques, `x` et `y` sont des *variables*. Python sait qu'une séquence nue de chiffres comme `12` ou `5` est un entier littéral. Voici donc un petit programme Python qui imite cette équation, affichant la valeur résultante de `y` :

```
>>> x = 5
>>> y = x + 12
>>> y
17
```

C'est la grande différence entre les mathématiques et les programmes : en mathématiques, = signifie *l'égalité* des deux membres d'une expression tandis que, dans les programmes, cela désigne l'affectation : *vous affectez la valeur de droite à la variable de gauche.*

Dans les programmes également, tout ce qui se trouve à droite du signe égal doit avoir une valeur (cela s'appelle être *initialisé*). Le terme de droite peut être une valeur littérale, ou une variable à laquelle une valeur a déjà été attribuée, ou une combinaison. Python sait que `5` et `12` sont des entiers littéraux.

La première ligne attribue la valeur entière `5` à la variable `x`. Nous pouvons maintenant utiliser la variable `x` dans la ligne suivante. Lorsque Python lit `y = x + 12`, il effectue les opérations suivantes :

- Il voit le signe = au milieu
- Il sait qu'il s'agit d'une affectation
- Il calcule le côté droit (en obtenant la valeur de l'objet référencé par `x` et en lui ajoutant `12`)
- Il assigne le résultat à la variable de gauche, `y`

Ensuite taper le nom de la variable `y` (dans l'interpréteur interactif) affichera sa nouvelle valeur.

Si vous aviez démarré votre programme directement avec la ligne `y = x + 12`, Python aurait généré une *exception* (une erreur), car la variable `x` n'aurait pas encore de valeur :

```
>>> y = x + 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Vous aurez un aperçu complet des exceptions au chapitre 9. En informatique, on dit que ce x n'a pas été *initialisé*.

En algèbre, vous pouvez travailler en sens inverse et attribuer une valeur à `y` afin de calculer `x`. Pour faire cela en Python, il faudrait obtenir les valeurs littérales et les variables initialisées situées à droite avant d'attribuer sa valeur à `x` situé à gauche :

```
>>> y = 5
>>> x = 12-y
>>> x
7
```

Les variables sont des noms, pas des emplacements

Il est maintenant temps de faire un point crucial sur les variables en Python : *les variables ne sont rien d'autre que des noms*. Ceci est différent de nombreux autres langages informatiques et c'est une notion clé à savoir sur Python, en particulier lorsque nous aborderons des objets *mutables* comme les listes. L'affectation *ne copie pas* une valeur ; elle *attache simplement un nom* à l'objet qui contient les données. Le nom fait *référence* à une chose, il n'est pas la chose elle-même. Voyez le nom comme une étiquette avec du texte qui serait rattachée par une ficelle à la boîte objet et située ailleurs dans la mémoire de l'ordinateur (Figure 2-3).

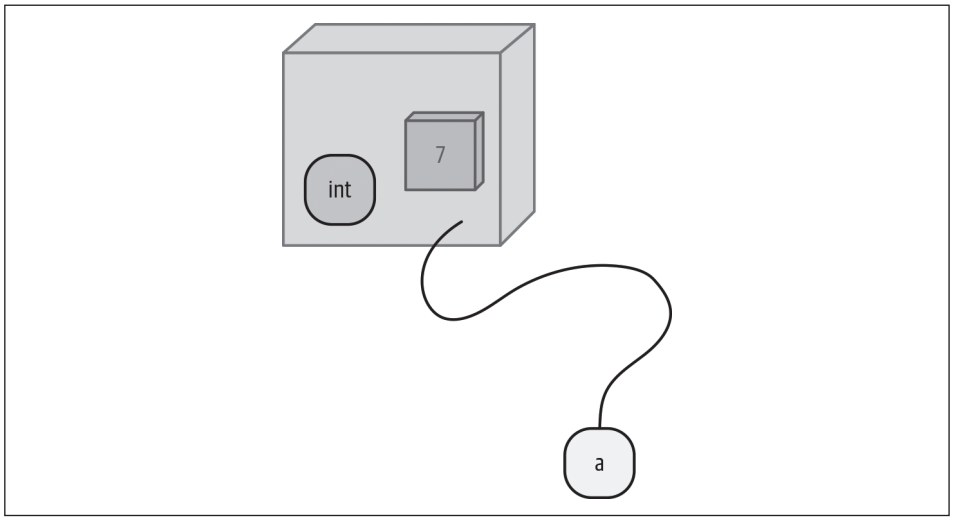

Figure 2-3. Les noms pointent vers des objets (la variable a pointe vers un objet entier ayant la valeur 7)

Dans d'autres langages, la variable elle-même a un type et se rattache à un emplacement de la mémoire. Vous pouvez modifier la valeur à cet endroit, mais cela doit rester du même type. C'est pourquoi les langages *statiques* vous font déclarer le type des variables. Python ne le fait pas, car un nom peut faire référence à n'importe quoi, et nous obtenons la valeur et le type en « tirant la ficelle » jusqu'à l'objet des données lui-même. Cela fait gagner du temps, mais il y a quelques inconvénients :

- Vous pouvez mal orthographier une variable et obtenir une exception car elle ne fait référence à rien, et Python ne le vérifie pas automatiquement comme le font les langages statiques. Le chapitre 19 explique les méthodes permettant de vérifier vos variables à l'avance pour éviter cela.
- La vitesse brute de Python est plus lente qu'un langage comme le C. Il fait faire plus de travail à l'ordinateur pour que vous n'ayez pas à le faire vous-même.

Essayez ceci avec l'interpréteur interactif (illustré à la Figure 2-4) :

1. Comme précédemment, attribuez la valeur `7` au nom de variable `a`. Cela crée une boîte d'objet contenant la valeur entière `7`.
2. Affichez la valeur de `a`.
3. Attribuez `a` à `b`, en faisant pointer `b` également sur la boîte d'objet contenant `7`.
4. Affichez la valeur de `b` :

```
>>> a = 7
>>> print(a)
7
>>> b = a
>>> print(b)
7
```

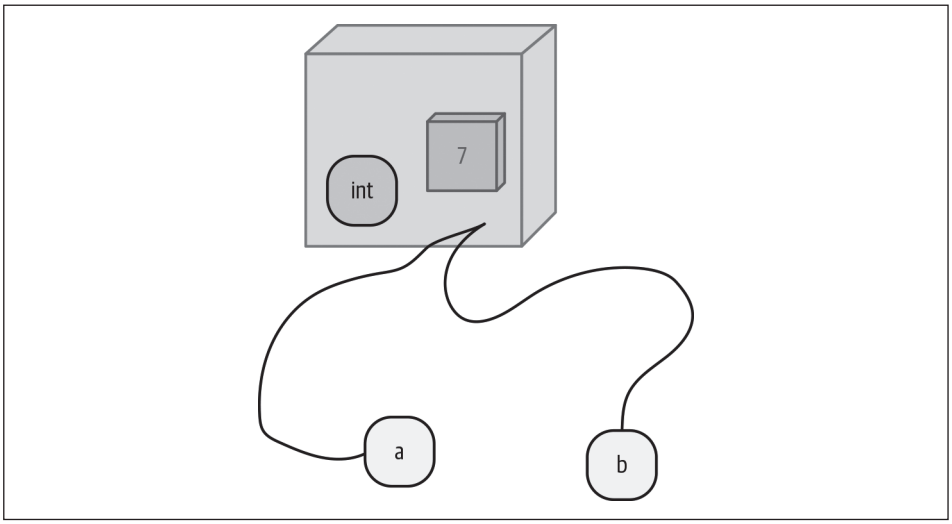

Figure 2-4. Copie d'un nom (maintenant la variable b pointe également vers le même objet entier)

En Python, si vous voulez connaître le type de quoi que ce soit (une variable ou une valeur littérale), vous pouvez utiliser `type(`*`chose`*`)`. `type()` est l'une des fonctions intégrées de Python. Si vous souhaitez vérifier si une variable pointe vers un objet d'un type spécifique, utilisez *`isinstance(type)`* :

```
>>> type(7)
<class 'int'>
>>> type(7) == int
True
>>> isinstance(7, int)
```

Quand je mentionne une fonction, je mets des parenthèses `()` après son nom pour bien indiquer qu'il s'agit d'une fonction plutôt que du nom d'une variable ou d'autre chose.

Essayons cela avec plus de valeurs littérales (*`58, 99.9, 'abc'`*) et de variables (*`a, b`*) :

```
>>> a = 7
>>> b = a
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>> type(58)
```

```
<class 'int'>
>>> type(99.9)
<class 'float'>
>>> type('abc')
<class 'str'>
```

Une classe est la définition d'un objet ; le chapitre 10 aborde les classes plus en détail. En Python, « classe » et « type » signifient à peu près la même chose.

Comme vous l'avez vu, lorsque vous utilisez une variable en Python, il recherche l'objet auquel elle fait référence. En coulisses, Python est actif, créant souvent des objets temporaires qui seront supprimés une ou deux lignes plus tard.

Reprenons un exemple précédent :

```
>>> y = 5
>>> x = 12-y
>>> x
7
```

Dans cet extrait de code, Python a effectué les opérations suivantes :

- Création d'un objet entier avec la valeur `5`
- Faire pointer la variable `y` vers cet objet `5`
- Incrémenter le compteur de référence de l'objet ayant la valeur `5`
- Création d'un autre objet entier avec la valeur `12`
- Soustraire la valeur de l'objet vers lequel `y` pointe (`5`) de la valeur `12` dans l'objet (anonyme) contenant cette valeur
- Affecter la valeur obtenue (`7`) à un nouvel objet entier (pour le moment anonyme)
- Faire pointer la variable `x` vers ce nouvel objet
- Augmenter le compteur de références de ce nouvel objet vers lequel `x` pointe
- Rechercher la valeur de l'objet vers lequel `x` pointe (`7`) et l'afficher

Lorsque le compteur de référence d'un objet atteint zéro, aucun nom ne pointe plus vers lui et il n'a donc pas besoin de rester. Python a un outil joliment nommé *ramasse-miettes* (en anglais, *garbage collector*) qui récupère la mémoire des objets qui ne sont plus utilisés. Imaginez quelqu'un derrière ces étagères de mémoire, retirant les boîtes obsolètes afin de les recycler.

Dans cet exemple, nous n'avons plus besoin des objets contenant les valeurs `5`, `12` ou `7`, ni des variables `x` et `y`. Le ramasse-miettes de Python peut choisir de les envoyer au paradis des objets[2], ou d'en garder quelques-uns pour des raisons de performances étant donné que les petits entiers ont tendance à être beaucoup utilisés.

2 Ou à l'île des objets abandonnés.

Attribution à plusieurs noms

Vous pouvez attribuer une valeur à plusieurs noms de variable en même temps :

```
>>> two = deux = zwei = 2
>>> two
2
>>> deux
2
>>> zwei
2
```

Réattribuer un nom

Étant donné que les noms pointent vers des objets, la modification de la valeur attribuée à un nom fait simplement pointer le nom vers un nouvel objet. Le compteur de références de l'ancien objet est décrémenté et celui du nouveau est incrémenté.

Copier

Comme vous l'avez vu dans la figure 2-4, l'affectation d'une variable existante `a` à une nouvelle variable nommée `b` fait simplement pointer `b` vers le même objet que `a`. Si vous sélectionnez l'étiquette `a` ou `b` et tirez la ficelle, vous aboutirez au même objet.

Si l'objet est immuable (comme un entier), sa valeur ne peut pas être modifiée, les deux noms sont donc essentiellement en lecture seule. Essayez ceci :

```
>>> x = 5
>>> x
5
>>> y = x
>>> y
5
>>> x = 29
>>> x
29
>>> y
5
```

Lorsque nous avons assigné *x* à *y*, cela a fait pointer le nom *y* vers l'objet entier avec la valeur `5` vers lequel *x* pointait également. La modification de *x* l'a fait pointer vers un nouvel objet entier avec la valeur `29`. Cela n'a pas changé celui contenant `5` vers lequel *y* pointe toujours.

Mais si les deux noms pointent vers un objet *modifiable* (*mutable*), vous pouvez modifier la valeur de l'objet via l'un ou l'autre nom, et vous verrez la valeur modifiée lorsque vous utiliserez l'un ou l'autre des noms. Quand on ne le sait pas, cela peut surprendre.

Une *liste* est un tableau de valeurs modifiables, et le chapitre 7 les aborde dans tous leurs détails. Pour cet exemple, a et b pointent chacun vers une liste comportant trois éléments entiers :

```
>>> a = [2,4,6]
>>> b = a
>>> a
[2, 4, 6]
>>> b
[2, 4, 6]
```

Ces éléments de la liste (`a[0]`, `a[1]` et `a[2]`) sont eux-mêmes comme des noms, pointant vers des objets entiers contenant les valeurs `2`, `4` et `6`. L'objet liste maintient ses éléments dans l'ordre.

Maintenant, changez le premier élément de la liste, via le nom `a`, et voyez que `b` a également changé :

```
>>> a[0] = 99
>>> a
[99, 4, 6]
>>> b
[99, 4, 6]
```

Lorsque le premier élément de la liste est modifié, il ne pointe plus vers l'objet de valeur `2`, mais un nouvel objet de valeur `99`. La liste est toujours de type liste, mais sa valeur (les éléments et leur ordre) est modifiable.

Choisir de bons noms de variables

> Il disait des choses vraies, mais leur donnait des noms erronés.
>
> — Elizabeth Barrett Browning

Il est surprenant de voir à quel point il est important de choisir de bons noms pour vos variables. Jusqu'à présent, dans de nombreux exemples de code, j'ai utilisé des noms jetables comme *`a`* et *`x`*. Dans les programmes réels, vous aurez beaucoup plus de variables à suivre à la fois, et vous devrez trouver un équilibre entre concision et clarté. Par exemple, il est plus rapide de taper `num_loons` plutôt que `number_of_loons` ou `gaviidae_inventory`, mais c'est plus parlant que simplement `n`.

PYTHON

Comprendre les bases et maîtriser la programmation

Destiné aux étudiants, aux programmeurs débutants et à tous ceux qui abordent Python, ce livre part des bases de ce langage pour aller vers des sujets plus complexes. Il propose de nombreux tutoriels et recettes de codes ainsi que les meilleures pratiques pour les tests, le débogage et d'autres astuces de développement.

Les exercices de fin de chapitre vous aident à mettre en pratique ce que vous avez appris.

- Découvrez les types de données simples, les opérations mathématiques de base et les manipulations textuelles
- Utilisez des techniques de gestion des données avec des structures intégrées
- Explorez la structure du code Python, y compris les fonctions
- Écrivez de gros programmes Python à l'aide de modules et de packages
- Plongez dans les objets, les classes et autres fonctionnalités orientées objet
- Examinez le stockage, y compris les bases de données relationnelles ou NoSQL
- Créez des clients Web, des serveurs, des API et des services
- Gérez les tâches système telles que les programmes, les processus et les threads
- Initiez-vous aux tâches concurrentielles et à la programmation réseau

Bill Lubanovic, programmeur sous Unix, sous interfaces utilisateur, dans le domaine des bases de données et sur le web, développe des services pour l'imagerie médicale.

Le traducteur, Bernard Desgraupes, *ancien élève de l'École Normale Supérieure de la rue d'Ulm et agrégé de mathématiques, a mené une carrière de maître de conférences à l'Université Paris X. Il travaille actuellement sur des projets de modélisation financière et de simulation.*

DANS LA MÊME COLLECTION

ISBN : 978-2-8073-3473-1

deboeck SUPÉRIEUR

www.deboecksuperieur.com