

Jason R. Briggs

La programmation
accessible
aux enfants !

PYTHON pour les KIDS

Dès 10 ans



EYROLLES

PYTHON pour les KIDS

La programmation accessible à tous !

Python est un langage de programmation puissant, expressif, facile à apprendre et amusant. Il est compatible avec Mac, Windows et Linux.

Python pour les kids donne vie à Python et t'emmène, ainsi que tes parents, dans l'univers de la programmation. Avec des trésors de patience, Jason R. Briggs te guidera parmi les bases, à mesure que tu t'essaieras à des exemples de programmes uniques et parfois hilarants, qui mettent en lumière des monstres voraces, des sorciers, des agents secrets, des corbeaux voleurs et d'autres curiosités du genre. Les définitions des termes utilisés, le code colorisé et expliqué en détail, ainsi que des illustrations en couleurs agrémentent l'apprentissage et le rendent plus aisés.

Les fins de chapitres proposent des puzzles de programmation pour t'entraîner. À la fin du livre, tu auras programmé deux jeux complets : un clone du fameux jeu de pong (balle bondissante et raquette) et « M. Filiforme court vers la sortie », un jeu de plates-formes avec des sauts, des animations et bien plus.

Tout au long de cette aventure, tu apprendras à :

- te servir des structures de données fondamentales comme les listes, les tuples et les dictionnaires ;
- organiser et réutiliser ton code à l'aide de fonctions, de classes et de modules ;
- utiliser les structures de contrôle comme les boucles et les instructions conditionnelles ;
- dessiner des formes et des motifs à l'aide du module de la tortue de Python ;
- créer des jeux, des animations et d'autres merveilles avec tkinter.

Pourquoi les adultes seraient-ils seuls à s'amuser ? Python pour les kids est ton ticket d'entrée dans le monde merveilleux de la programmation.

À propos de l'auteur

Jason R. Briggs programme depuis l'âge de huit ans. Il a rédigé des programmes en tant que développeur et architecte système.



www.editions-eyrolles.com





Jason R. Briggs

PYTHON pour les KIDS



EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Traduction autorisée de l'ouvrage en langue anglaise intitulé
Python for Kids
de Jason Briggs (ISBN : 9781593274078),
publié par No Starch Press.

All Rights Reserved.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.

© 2013 by Jason Briggs / No Starch Press pour l'édition en langue anglaise
© Groupe Eyrolles, 2015, pour la présente édition, ISBN : 978-2-212-14088-0
© Traduction française : William Piette

À propos de l'auteur

Jason R. Briggs programme depuis l'âge de huit ans, lorsqu'il a appris le BASIC sur un TRS-80 de Radio Shack. Sur le plan professionnel, il a ensuite rédigé des programmes en tant que développeur et architecte système. Il a en outre collaboré au magazine *Java developer's journal*. Ses articles ont été publiés dans *JavaWorld*, *ONJava* et *ONLamp*. *Python pour les kids* est son premier livre.

Pour contacter Jason, adressez-vous à l'éditeur par courriel à ahabian@eyrolles.com, qui transmettra.

À propos de l'illustrateur

Miran Lipovača est l'auteur de *Learn you a haskell for great good!*. Il aime la boxe, jouer de la guitare basse et, bien entendu, dessiner. Il éprouve une fascination pour la *Danse macabre* et le nombre 71. Quand il passe des portes à ouverture automatique, il aime prétendre qu'il les ouvre en réalité à la force de son esprit.

À propos des relecteurs techniques

Récemment diplômé de l'école Nueva à l'âge de 15 ans, Josh Pollock est étudiant à l'école secondaire supérieure de Lick-Wilmerding de San Francisco. Il a commencé à programmer en Scratch à l'âge de 9 ans et en TI-BASIC à 11 ans, puis a débuté en Python et Java à 12 ans, et en UnityScript à 13 ans. Outre la programmation, Josh aime jouer de la trompette, le développement de jeux informatiques et discuter avec les gens de sujets intéressants comme les disciplines académiques scientifiques, technologiques, mathématiques et d'ingénierie.

Maria Fernandez possède une maîtrise en linguistique appliquée, s'intéresse à l'informatique et aux technologies depuis plus de vingt ans. Elle a enseigné l'anglais à de jeunes femmes réfugiées dans le cadre du *Global village project* de Géorgie. Elle réside actuellement dans le nord de la Californie et travaille dans un service d'expérimentation pédagogique.

Remerciements

Ma situation est comparable à celle où je devrais monter sur scène pour recevoir un prix ou une récompense et où je me rendrais compte que j'ai laissé dans un autre pantalon la liste des personnes que je voudrais remercier. Je suis assuré d'oublier quelqu'un, or la musique d'introduction roule déjà, le rideau s'ouvre et me voici bientôt sur scène.

Ceci étant dit, voici une liste certainement incomplète des gens à qui je voue une immense gratitude pour m'avoir aidé à porter ce livre à un stade que je pense bon.

Merci à l'équipe de No Starch, en particulier à Bill Pollock, pour avoir appliquée une dose libérée mais réfléchie de « que penserait un enfant », lors des corrections et de la publication de ce texte. Lorsque vous programmez depuis des années, il est trop facile de perdre de vue la difficulté de certaines notions et matières pour les débutants ; Bill a été d'une aide inestimable pour souligner ces parties plus compliquées, souvent négligées. Mes remerciements aussi à Serena Yang, extraordinaire directrice de rédaction, qui, je l'espère, ne s'est pas trop arraché les cheveux à apporter correctement la coloration syntaxique aux plus de 300 pages de code.

Un énorme remerciement à l'endroit de Miran Lipovača pour ses illustrations particulièrement brillantes, et même au-delà. Vraiment, si j'avais dû dessiner moi-même ces illustrations, je crois que le lecteur se poserait souvent la question : « Mais qu'est-ce que c'est ? Est-ce un ours, un chien ? Et ça, c'est supposé être un arbre ? Non !? »

Merci aussi aux relecteurs. Je leur présente mes excuses si certaines de leurs suggestions n'ont pas été mises en œuvre à la fin. Vous aviez sans doute raison et je ne peux reprocher qu'à moi-même les probables gaffes. Des remerciements particuliers à Josh pour certaines excellentes suggestions et de très bonnes idées. Et des excuses aussi à Maria pour l'avoir obligée à remanier occasionnellement du code mis en forme de manière un peu bizarre.

Merci à ma femme et à ma fille, qui ont dû vivre avec un époux et un père vivant le nez sur l'écran encore plus souvent que d'habitude.

À ma maman, pour les encouragements sans fin qu'elle m'a adressés au cours de toutes ces années. Enfin, merci à mon papa d'avoir acheté un ordinateur dès les années 1970 et d'avoir su composer d'emblée avec moi, qui voulais l'utiliser autant que lui. Rien de tout cela n'eût été possible sans lui.

TABLE DES MATIÈRES

Avant-propos	1
Pourquoi Python ?	2
Comment apprendre à programmer ?	2
À qui est destiné ce livre ?	3
Que contient ce livre ?	4
Site d'accompagnement	5
Amuse-toi bien !	5
PARTIE 1	
APPRENDRE À PROGRAMMER	
1	
Les serpents rampent, mais pas tous	9
Quelques mots à propos du langage.....	10
Installer Python	11
Installer Python sous Windows 7 (ou 8).....	11
Installer Python sur Mac OS X	13
Installer Python sous Linux (Ubuntu).....	16
Dès que Python est installé	17
Enregistrer des programmes Python.....	18
Ce que tu as appris	20
2	
Calculs et variables	21
Calculer avec Python	22
Les opérateurs de Python	23
L'ordre des opérateurs	24
Les variables sont comme des étiquettes.....	25
Utiliser les variables	26
Ce que tu as appris	29

3	Chaînes, listes, tuples et dictionnaires	31
Les chaînes	32	
Créer des chaînes	32	
Gérer les problèmes de chaînes	33	
Insérer des valeurs dans des chaînes.....	36	
Multiplier des chaînes.....	37	
Plus puissantes que les chaînes : les listes	38	
Ajouter des éléments à une liste	41	
Supprimer des éléments d'une liste	42	
Arithmétique de liste	42	
Tuples	45	
Dictionnaires	45	
Ce que tu as appris	48	
Puzzles de programmation	48	
1. Favoris	48	
2. Compter les combattants	48	
3. Salutations.....	48	
4	Dessiner avec une tortue	49
Utiliser le module turtle de Python	50	
Créer un canevas	50	
Déplacer la tortue	52	
Ce que tu as appris	57	
Puzzles de programmation	57	
1. Un rectangle	57	
2. Un triangle	57	
3. Un carré sans coins.....	57	
5	Poser des questions avec if et else	59
Instructions if	60	
Un bloc est un groupe d'instructions	60	
Des conditions pour comparer des choses	62	
Instructions si-alors-sinon	64	
Instructions if et elif	65	
Combiner des conditions.....	66	
Variables sans valeur : None	66	
Différence entre chaînes et nombres	67	
Ce que tu as appris	70	

Puzzles de programmation.....	70
1. Es-tu riche ?.....	70
2. Barres chocolatées.....	70
3. Juste le bon nombre	71
4. Affronter des ninjas.....	71

6 Tourner en boucle 73

Utiliser les boucles for	74
Tant que nous parlons de boucles : while	81
Ce que tu as appris	83
Puzzles de programmation.....	83
1. La boucle Bonjour	84
2. Nombres pairs.....	84
3. Mes cinq ingrédients préférés.....	84
4. Ton poids sur la lune.....	85

7 Recycler du code avec des fonctions et des modules 87

Utiliser des fonctions	88
Qu'est-ce qu'une fonction ?	89
Variables et portée	90
Utiliser des modules	92
Ce que tu as appris	95
Puzzles de programmation.....	95
1. Fonction de base du poids sur la lune	95
2. Fonction poids sur la lune avec les années	96
3. Programme de poids sur la lune.....	96

8 Classes et objets 97

Organiser les choses en classes	98
Enfants et parents.....	99
Ajouter des objets aux classes	100
Définir des fonctions de classes	101
Ajouter des caractéristiques à une classe avec des fonctions	101
Pourquoi utiliser des classes et des objets ?	103
Objets et classes en images.....	105
Autres fonctionnalités des objets et des classes	107
Fonctions héritées.....	107

Fonctions appelant d'autres fonctions	108
Initialiser un objet	110
Ce que tu as appris	111
Puzzles de programmation	111
1. Moulinet de girafe	111
2. Fourche de tortues	112

9 Fonctions intégrées de Python 113

Utiliser des fonctions intégrées	114
La fonction abs	114
La fonction bool	115
La fonction dir	116
La fonction eval	118
La fonction exec	119
La fonction float	120
La fonction int	121
La fonction len	121
Les fonctions max et min	122
La fonction range	123
La fonction sum	125
Manipuler des fichiers	125
Créer un fichier de test	125
Ouvrir un fichier en Python	128
Écrire dans des fichiers	129
Ce que tu as appris	130
Puzzles de programmation	130
1. Code mystère	130
2. Message caché	131
3. Copier un fichier	131

10 Modules utiles de Python 133

Créer des copies avec le module copy	134
Suivre les mots-clés avec le module keyword	137
Nombres aléatoires avec le module random	137
Obtenir un nombre au hasard avec randint	137
Sélectionner un élément au hasard dans une liste avec choice	139
Mélanger les éléments d'une liste avec shuffle	140
Contrôler le shell avec le module sys	140
Quitter le shell Python avec la fonction exit	140
Lire avec l'objet stdin	141

Écrire dans l'objet <code>stdout</code>	141
Connaitre la version de Python utilisée.....	142
Se jouer du temps avec le module <code>time</code>	142
Convertir une date avec <code>asctime</code>	144
Obtenir la date et l'heure selon le fuseau horaire.....	144
Hiberner quelque temps avec <code>sleep</code>	145
Enregistrer des informations avec le module <code>pickle</code>	146
Ce que tu as appris	147
Puzzles de programmation.....	148
1. Copier des voitures	148
2. Objets favoris avec <code>pickle</code>	148

11

Autres graphismes avec la tortue 149

Dessiner un carré, pour commencer	150
Dessiner une voiture.....	154
Voir la vie en couleurs	156
Une fonction pour dessiner un cercle plein	157
Dessiner en noir et blanc	158
Fonction de dessin de carré	159
Dessiner des carrés pleins	160
Dessiner des étoiles pleines	162
Ce que tu as appris	163
Puzzles de programmation.....	164
1. Dessiner un octogone	164
2. Dessiner un octogone plein	164
3. Autre fonction de dessin d'étoile.....	165

12

De meilleurs graphismes avec tkinter 167

Créer un bouton à cliquer.....	169
Utiliser des paramètres nommés	171
Créer le canevas de dessin	172
Dessiner des lignes	172
Dessiner des rectangles et des carrés	174
Dessiner de nombreux rectangles	176
Définir la couleur	178
Dessiner des arcs	181
Dessiner des polygones.....	183
Afficher du texte	185
Afficher des images.....	186
Créer une animation de base	188

Réagir à un événement	191
Autres façons d'utiliser l'identifiant	193
Ce que tu as appris	195
Puzzles de programmation	195
1. Remplir l'écran de triangles	195
2. Le triangle mobile	195
3. La photo mobile	196

PARTIE 2 REBONDIR !

13

Débuter ton premier jeu : Rebondir !	199
Frapper la balle magique	200
Créer le canevas du jeu	200
Créer la classe de la balle	201
Ajouter de l'action	204
Déplacer la balle	204
Faire rebondir la balle	206
Changer la direction de départ de la balle	207
Ce que tu as appris	210

14

Achever ton premier jeu : Rebondir !	211
Ajouter la raquette	212
Déplacer la raquette	213
DéTECTER quand la balle touche la raquette	215
Ajouter un facteur chance	218
Ce que tu as appris	222
Puzzles de programmation	222
1. Retarder le début du jeu	222
2. Un véritable « Partie terminée »	223
3. Accélérer la balle	223
4. Enregistrer le score du joueur	223

PARTIE 3

M. FILIFORME COURT VERS LA DROITE

15

Créer les graphismes du jeu M. Filiforme

227

Plan du jeu de M. Filiforme	228
Obtenir Gimp	228
Créer les éléments de jeu	230
Préparer une image à fond transparent	231
Dessiner M. Filiforme	231
Dessiner les plates-formes	234
Dessiner la porte	234
Dessiner l'arrière-plan	235
Gérer la transparence	236
Ce que tu as appris	237

16

Développer le jeu de M. Filiforme

239

Créer la classe Jeu	240
Définir le titre de la fenêtre et créer le canevas	240
Terminer la fonction <code>_init_</code>	241
Créer la fonction boucle_principale	242
Créer la classe Coords	244
Vérifier les collisions	245
Les lutins entrent en collision horizontalement	246
Les lutins entrent en collision verticalement	248
Assembler le tout – Code final de détection de collision	248
Créer la classe Lutin	251
Ajouter les plates-formes	252
Ajouter un objet plate-forme	253
Ajouter d'autres plates-formes	254
Ce que tu as appris	256
Puzzles de programmation	256
1. Damier	256
2. Damier à deux images alternées	257
3. Étagère et lampe	257

17		
Créer M. Filiforme		259
Initialiser le personnage en fil de fer	260	
Charger les images du personnage	260	
Définir les variables	261	
Lier les touches aux actions	262	
Tourner le personnage vers la gauche ou la droite	263	
Faire sauter le personnage	264	
Ce que nous avons jusqu'ici	264	
Ce que tu as appris	266	
18		
Achever le jeu de M. Filiforme		267
Animer le personnage	268	
Créer la fonction animer	268	
Connaitre l'emplacement du personnage	271	
Déplacer le personnage	272	
Tester le lutin du personnage	280	
La porte !	281	
Créer la classe LutinPorte	281	
Déetecter la porte	282	
Ajouter l'objet porte	282	
Le jeu final	283	
Ce que tu as appris	289	
Puzzles de programmation	290	
1. Tu as gagné !	290	
2. Animer la porte	290	
3. Plates-formes mobiles	290	
Conclusion		
Et à partir de là ?		291
Programmation graphique et de jeux	292	
PyGame	292	
Langages de programmation	294	
Java	294	
C/C++	294	
C#	295	
PHP	296	
Objective-C	296	
Perl	297	
Ruby	297	

JavaScript	297
Et pour finir	298

Annexe
Mots-clés de Python **299**

AND	300
AS	300
ASSERT	300
BREAK	301
CLASS	301
CONTINUE	302
DEF	303
DEL	303
ELIF	303
ELSE	304
EXCEPT	304
FINALLY	304
FOR	304
FROM	304
GLOBAL	306
IF	306
IMPORT	307
IN	307
IS	308
LAMBDA	308
NOT	308
OR	308
PASS	309
RAISE	310
RETURN	311
TRY	311
WHILE	311
WITH	312
YIELD	312

Glossaire 313

Index 319



AVANT-PROPOS

Pourquoi apprendre la programmation informatique ?

La programmation encourage la créativité, le raisonnement et la résolution de problèmes. Le programmeur a l'opportunité de créer quelque chose à partir de rien, d'utiliser la logique pour transformer les constructions de programmation en une forme que l'ordinateur peut exécuter et, quand les choses ne se passent pas tout à fait comme prévu, d'exploiter ses capacités de résolution pour trouver ce qui ne va pas. La programmation est une activité amusante, parfois pleine de défis, voire quelquefois frustrante – eh oui ! –, et les compétences acquises grâce à elle sont utiles à la fois à l'école et au travail... Même si ta future carrière n'a rien à voir avec l'informatique !

En plus, si tu n'as rien d'autre à faire, la programmation, c'est franchement chouette pour passer un agréable après-midi, alors qu'il ne fait pas beau dehors.

Pourquoi Python ?

Python est un langage de programmation facile à apprendre ; il possède également des caractéristiques réellement utiles pour le programmeur débutant. Le code est assez facile à lire, comparé à d'autres langages de programmation, et il dispose d'une console de commandes interactive pour y entrer tes programmes afin de les voir fonctionner. En plus de sa structure de langage simple et de la console interactive qui facilitent les expériences, Python propose certaines fonctionnalités qui améliorent fortement l'apprentissage et permettent de rassembler des animations simples pour créer tes propres jeux. Parmi celles-ci, on trouve le module turtle, inspiré des célèbres graphismes avec la tortue – il est utilisé par le langage Logo depuis les années 1960 – et conçu pour une utilisation éducative. Il y a aussi le module tkinter qui permet de créer des interfaces graphiques évolutives grâce à la bibliothèque Tk.

Comment apprendre à programmer ?

Comme pour tout ce que tu essaies pour la première fois, il vaut mieux toujours débuter par les bases. Donc, commence par les premiers chapitres et résiste à l'envie de sauter directement aux derniers. Personne ne peut jouer une symphonie du premier coup, simplement en prenant un instrument. Les élèves pilotes d'avion ne commencent pas à piloter un avion tant qu'ils n'ont pas compris et assimilé les commandes essentielles. Les gymnastes ne sont (généralement) pas capables de réaliser des saltos arrière dès leur premier essai. Si tu passes trop vite à la suite, non seulement tu n'auras pas en tête les idées de base mais, en plus, le contenu des chapitres suivants te paraîtra plus compliqué qu'il ne l'est réellement.

À mesure que tu avances dans ta lecture, essaie chacun des exemples proposés pour voir comment ils fonctionnent. À la fin de la plupart des chapitres, tu trouveras des puzzles de programmation. Essaie de les réaliser et de les résoudre, car ils t'aideront à améliorer tes connaissances en programmation. Retiens que plus tu assimiles les bases, plus il te sera facile de comprendre les idées plus complexes qui suivront.

Si tu rencontres des choses frustrantes ou trop complexes à résoudre, voici quelques pistes qui te seront très utiles.

1. Découpe un problème en parties plus petites. Essaie de comprendre ce que chaque petit extrait de code fait, ou réfléchis à une seule partie d'une idée difficile (concentre-toi sur une portion de code au lieu d'essayer de comprendre tout en même temps).
2. Si cela ne suffit pas à t'aider, le mieux est souvent de laisser un moment le problème de côté. Dors dessus et reviens-y un autre jour. Souvent, cette méthode aide à résoudre de nombreux problèmes, et elle est particulièrement utile pour les programmeurs.

À qui est destiné ce livre ?

Ce livre a été rédigé pour toute personne qui s'intéresse à la programmation, qu'il s'agisse d'un enfant ou d'un adulte approchant la programmation pour la première fois. Si tu veux apprendre à rédiger tes propres logiciels, au lieu de simplement utiliser les programmes réalisés par d'autres, *Python pour les kids* constitue un excellent point de départ.

Dans les chapitres suivants, tu trouveras des informations pour installer Python, démarrer la console et réaliser des calculs simples, afficher du texte à l'écran, créer des listes, réaliser des opérations élémentaires de contrôle de flux à l'aide des instructions `if`, ainsi que des boucles `for`. Au passage, tu comprendras ce que signifient les instructions `if` et les boucles `for` ! Tu apprendras à réutiliser du code dans des fonctions, les éléments fondamentaux des classes et des objets, ainsi que les descriptions de quelques-uns des nombreux modules et fonctions intégrés dans Python.

Des chapitres inspectent les graphismes à l'aide de la tortue, sous forme simple ou plus évoluée, ainsi que l'utilisation du module `tkinter` pour dessiner à l'écran de l'ordinateur. À la fin de nombreux chapitres, les puzzles de programmation de complexité variée sont là pour graver dans le marbre les nouvelles connaissances acquises, en te poussant à rédiger par toi-même de petits programmes.

Ensuite, lorsque tu auras construit tes connaissances fondamentales de la programmation, tu apprendras à écrire tes propres jeux. Tu développeras deux jeux graphiques pour maîtriser la détection de collision, les événements et les différentes techniques d'animation.

Les exemples de ce livre utilisent pour la plupart la console d'environnement d'exécution d'IDLE (qui signifie « environnement de développement intégré »), intégrée à Python. IDLE fournit ce qu'on appelle la « coloration syntaxique », le copier-coller (comme quand tu copies et colles dans les autres applications) et une fenêtre d'éditeur où tu peux enregistrer du code pour plus tard. C'est donc à la fois un environnement interactif pour l'expérimentation et une sorte d'éditeur de texte. Les exemples proposés fonctionnent aussi parfaitement dans une console standard et un éditeur de texte traditionnel, mais la coloration syntaxique d'IDLE et son environnement légèrement plus convivial facilitent la compréhension ; c'est pourquoi le tout premier chapitre te montre comment le mettre en place.

Que contient ce livre ?

Voici un bref aperçu de ce que tu trouveras dans cet ouvrage.

- Le chapitre 1 forme une introduction à la programmation, avec des instructions pour installer Python pour la première fois.
- Le chapitre 2 présente les calculs de base et les variables, tandis que le chapitre 3 décrit certains types fondamentaux de Python, comme les chaînes, les listes et les *tuples*.
- Le chapitre 4 propose un avant-goût du module `turtle`. On saute de la programmation élémentaire au déplacement d'une tortue, sous la forme d'une flèche à l'écran.
- Le chapitre 5 examine les variantes des conditions et de l'instruction `if`, puis le chapitre 6 évalue les boucles `for` et `while`.
- Au chapitre 7, tu commences à utiliser et à créer des fonctions puis, au chapitre 8, tu examines les classes et les objets. Tu abordes suffisamment d'idées de base pour assimiler certaines techniques de programmation dont tu auras besoin dans les chapitres de développement de jeux, qui viennent plus tard dans le livre. À ce stade, la matière commence à devenir plus compliquée.
- Le chapitre 9 passe en revue les principales fonctions intégrées de Python et le chapitre 10 poursuit avec quelques modules

(essentiellement des conteneurs de fonctionnalités utiles), installés par défaut avec le langage.

- Le chapitre 11 revient sur le module `turtle` et propose d'essayer quelques formes plus complexes. Le chapitre 12 aborde l'utilisation du module `tkinter` pour créer des graphismes encore plus évolués.
- Aux chapitres 13 et 14, tu réalises ton premier jeu, « Rebon-dit ! », qui se fonde sur les connaissances acquises dans les chapitres précédents. Aux chapitres 15 à 18, tu crées un autre jeu, « M. Filiforme court vers la sortie ». Ces chapitres devraient te poser quelques petits soucis ; alors, si rien ne va, télécharge le code à partir du site d'accompagnement et compare ton code à ces exemples qui fonctionnent.
- Dans la postface, nous jetons un coup d'œil à la boîte à outils `PyGame` et à d'autres langages de programmation fort appréciés.
- Enfin, dans l'annexe, nous reprenons en détail les mots-clés de Python et le glossaire propose des définitions des termes informatiques (**en gras dans le texte**) utilisés tout au long de ce livre.

Site d'accompagnement

Si tu as besoin d'aide au cours de ta lecture, essaie d'aller voir du côté du site web d'accompagnement, à l'adresse <http://www.editions-eyrolles.com/go/pykids>. Tu y trouveras des téléchargements pour tous les exemples du livre et d'autres puzzles de programmation, ainsi que les solutions des puzzles de programmation, au cas où tu te sentirais perdu ou si tu veux comparer tes solutions avec celles proposées.

Amuse-toi bien !

À mesure que tu avances dans ce livre, rappelle-toi que la programmation peut être divertissante et qu'il ne faut pas la considérer comme du travail. Pense à la programmation comme à une manière de créer certains jeux et applications amusants, que tu pourras partager avec tes amis par exemple.

Apprendre à programmer constitue un formidable exercice mental et les résultats peuvent être très gratifiants, c'est-à-dire qu'ils t'apporteront beaucoup de satisfaction. Par-dessus tout, quoi que tu fasses, amuse-toi bien !

PARTIE 1

APPRENDRE À PROGRAMMER





1

LES SERPENTS RAMPENT, MAIS PAS TOUS

Un **programme** informatique est un ensemble d'instructions qui font **exécuter** certaines actions par un ordinateur. Ce n'est pas la partie matérielle, physique de l'ordinateur (comme les fils, les cartes, le disque dur, et ainsi de suite), mais le « truc » caché qui fonctionne grâce à ce matériel. Un programme informatique, ou simplement programme comme nous le nommerons généralement, constitue l'ensemble des commandes qui indiquent à ce matériel muet ce qu'il doit faire. Un **logiciel** est une série, une « collection » de programmes.

Sans les programmes, presque tous les appareils que tu utilises tous les jours s'arrêteraient de fonctionner ou seraient tout simplement beaucoup moins utiles. D'une manière ou d'une autre, les programmes contrôlent non seulement ton ordinateur, mais également les consoles de jeux, les jeux vidéo, les téléphones portables et les systèmes GPS des voitures. Les logiciels contrôlent aussi des appareils moins évidents, comme les écrans plats des télévisions et leurs télécommandes, ainsi que des radios, des lecteurs DVD, des fours et même certains réfrigérateurs récents. Même les moteurs des voitures, les feux tricolores sur les routes, les éclairages publics, les feux de signalisation des trains, les panneaux d'affichage électriques et les ascenseurs sont contrôlés par des programmes.

Les **programmes** sont un peu comme les pensées. Si tu ne pensais pas, tu resterais probablement assis sur le sol, le regard vide, à te baver sur le T-shirt (beurk !). Ta pensée « lève-toi » est une instruction, ou une commande, qui ordonne à ton corps de te lever sur tes deux jambes. De la même façon, les programmes indiquent aux ordinateurs ce qu'ils doivent faire.

Si tu sais écrire des programmes informatiques, tu peux réaliser toutes sortes de choses. Évidemment, tu n'es peut-être pas capable d'écrire des programmes pour contrôler des voitures, des feux de signalisation ou ton réfrigérateur (du moins pas tout au début), mais tu peux déjà réaliser des pages web, tes propres jeux ou même un programme pour t'aider à faire tes devoirs !

Quelques mots à propos du langage

Comme les humains, les ordinateurs emploient plusieurs langues pour communiquer ; dans leur cas, nous parlons de « langages de programmation ». Un langage de programmation est simplement une façon particulière de parler à un ordinateur, une manière d'utiliser des instructions comprises à la fois par lui et les humains.

Certains langages de programmation portent le nom de personnes (comme Ada et Pascal), d'autres sont construits à partir d'acronymes (comme Basic et Fortran). Le nom Python trouve son origine dans une série télévisée humoristique britannique des années 1970, franchement burlesque (*non-sense*, comme disent les Anglais), qui s'intitulait *Monty Python's Flying Circus*, ce qui n'a donc rien à voir avec le serpent !

Le langage de programmation Python possède quelques caractéristiques qui le rendent extrêmement utile pour les débutants. La plus importante vient du fait qu'il te permet d'écrire vraiment rapidement

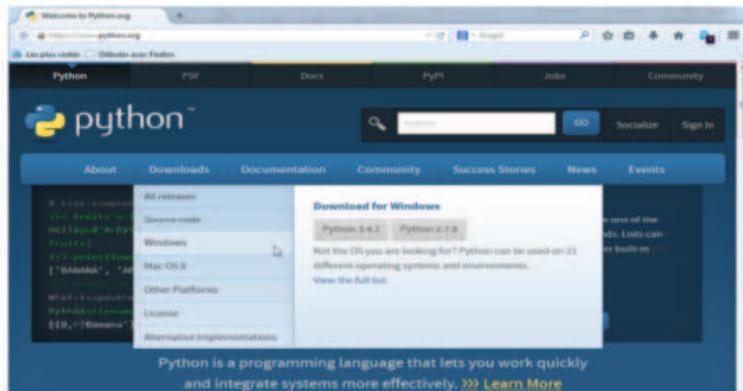
des **programmes** simples mais efficaces. Python ne possède que peu de symboles compliqués, comme les accolades (`{ }`), le dièse (`#`) et le symbole dollar (`$`), qui rendent les autres langages de programmation si difficiles à lire et, donc, moins conviviaux pour les débutants.

Installer Python

L'**installation** de Python est assez simple. Nous verrons comment procéder pour Windows 7 (ou 8), Mac OS X et Linux (Ubuntu). Lors de son installation, tu devras aussi créer un raccourci pour le programme IDLE, un environnement de développement intégré qui aide à rédiger des programmes en Python. Si ce langage est déjà installé sur ton ordinateur, va directement à la section « Dès que Python est installé » (page 17).

Installer Python sous Windows 7 (ou 8)

Pour installer Python sous Microsoft Windows 7 (ou 8), utilise un navigateur web. Rends-toi à l'adresse <http://www.python.org> et télécharge le dernier installeur de Python 3 pour Windows. Cherche le menu **Download** (qui signifie « télécharger »), puis place ta souris dessus pour voir se dérouler le menu correspondant. Le bouton **Python 3.4.1** te permet de télécharger l'installateur pour Windows. Une version plus récente de Python existe peut-être : préfère-la.



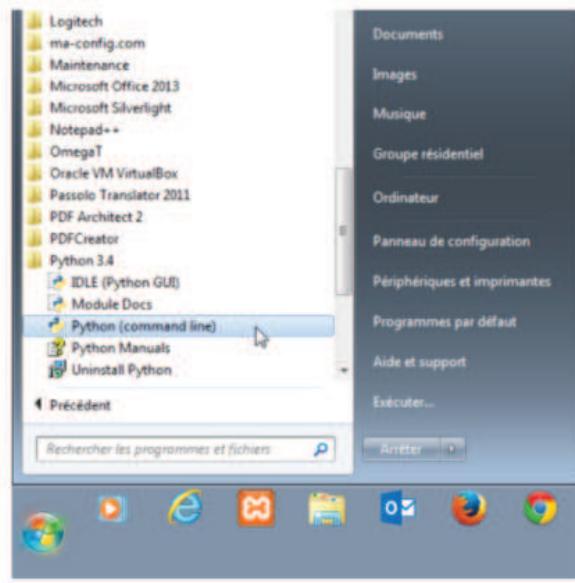
NOTE

La version exacte de Python que tu télécharges importe peu, pourvu qu'elle commence par le chiffre 3, soit ici 3.4.1.

Lorsque le téléchargement de l'installateur pour Windows est terminé, double-clique sur son icône et suis les instructions pour installer Python à son emplacement par défaut. Voici comment procéder.

1. Sélectionne **Install for All Users**, puis clique sur **Next** (suivant).
2. Accepte le dossier d'installation proposé par défaut ; n'oublie pas de noter son nom (probablement **C:\Python34** ou **C:\Python35**). Clique sur **Next**.
3. Ignore la partie **Customize Python** de l'installation et clique sur **Next**.

À la fin du processus d'installation, tu devrais disposer d'une entrée **Python 3** dans ton menu **Démarrer>Tous les programmes** :

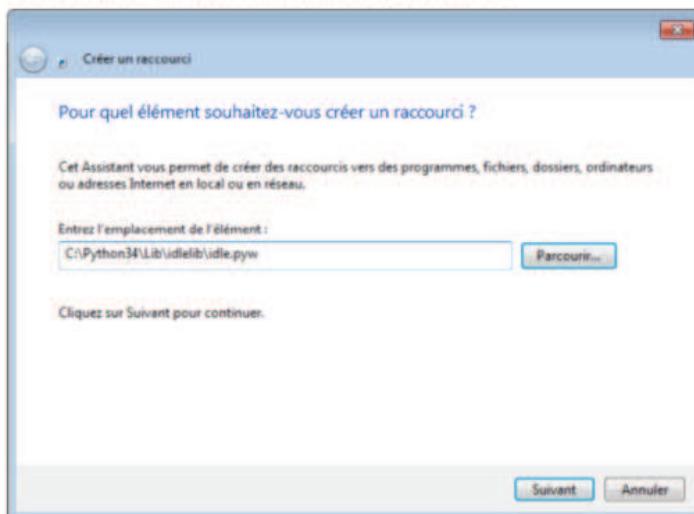


Suis ensuite les étapes ci-après pour ajouter le raccourci **Python 3** sur ton Bureau.

1. Clique droit sur le Bureau et sélectionne **Nouveau>Raccourci** dans le menu contextuel.
2. Saisis ce qui suit dans la case **Entrez l'emplacement de l'élément** (vérifie que le dossier que tu entres est bien celui que tu as noté précédemment).

c:\Python34\Lib\idlelib\idle.pyw

Voici la boîte de dialogue qui doit apparaître :



3. Clique sur **Suivant** pour aller dans la suite de la boîte de dialogue.
4. Saisis le nom **IDLE**, puis clique sur **Terminer** pour créer le raccourci.

Maintenant, passe à la section « Dès que Python est installé » (page 17) pour tes premiers essais.

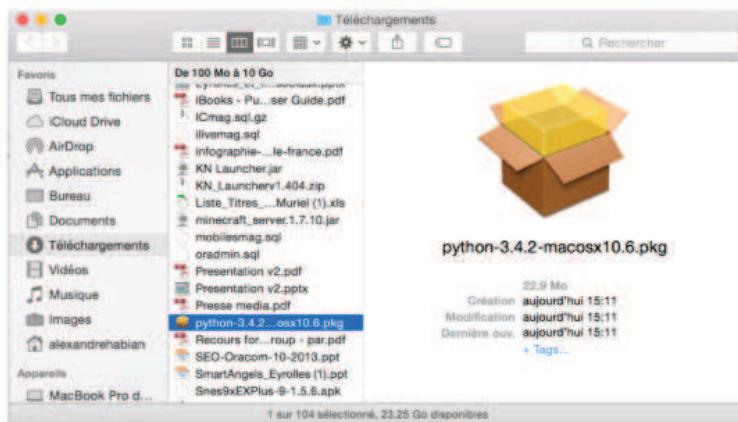
Installer Python sur Mac OS X

Si tu utilises un Mac, Python y est très probablement déjà installé, mais il s'agit sans doute d'une ancienne version du langage. Pour vérifier que tu dispose bien de la dernière, ouvre le navigateur web et va à l'adresse www.python.org pour télécharger le dernier installateur pour Mac. Clique sur le menu **Downloads**, puis sur **Mac OS X**. La page suivante te donne accès à la dernière version, **Latest Python 3 Release**. C'est celle qu'il te faut.

Deux installateurs différents sont disponibles. Celui que tu choisisra dépend de la version de Mac OS X dont tu disposes. Pour la connaître, clique sur l'icône **Apple** dans la barre de menus du haut, puis clique sur **À propos de ce Mac**. Choisis un installateur comme suit.

- Si tu utilises une version de Mac OS X comprise entre 10.3 et 10.5, choisis la version 32 bits de Python 3 pour i386/PPC.
- Si tu utilises une version 10.6 ou supérieure, sélectionne la version 64 bits/32 bits de Python pour x86-64.

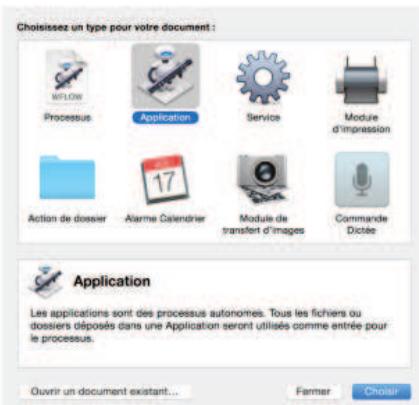
Dès que le téléchargement du fichier est terminé (il affiche l'extension de fichier **.pkg**), double-clique dessus. Tu peux alors voir une fenêtre qui montre le contenu du fichier :



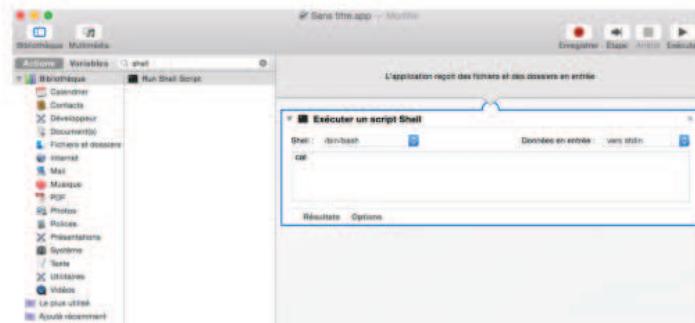
Dans cette fenêtre, double-clique sur **Python-TonNumérodeVersion.pkg**, puis suis les instructions pour installer le logiciel. À un moment, tu seras invité à entrer le mot de passe d'administrateur du Mac pour que Python puisse s'installer ; si tu ne le connais pas, demande à tes parents de le saisir.

Il te faut ensuite ajouter un script au Bureau pour lancer l'application IDLE de Python.

1. Clique sur l'icône **Spotlight**, la petite loupe dans le coin supérieur droit de l'écran.
2. Dans la boîte de dialogue qui s'affiche, saisis **Automator**.
3. Clique sur l'application en forme de robot, lorsqu'elle apparaît dans le menu. Elle se trouve dans une section nommée **Applications**.
4. Lorsque l'Automator démarre, sélectionne le modèle **Application** :



5. Clique sur **Choisir** pour poursuivre.
6. Parmi les actions listées, cherche **Run Shell Script** et glisse-le dans le volet vide de droite. Tu obtiens quelque chose comme ce qui suit :



7. Dans la zone de texte, tu peux voir le mot **cat**. Sélectionne-le et remplace-le par le texte suivant (tout, depuis **open** jusqu'à **args**) :

```
open -a "/Applications/Python 3.4/IDLE.app" --args
```

Ensuite, tu devras peut-être changer le nom de **dossier**, selon la version de Python installée.

1. Clique sur **Fichier>Enregistrer** et entre le nom **IDLE** au moment de l'enregistrement du raccourci.

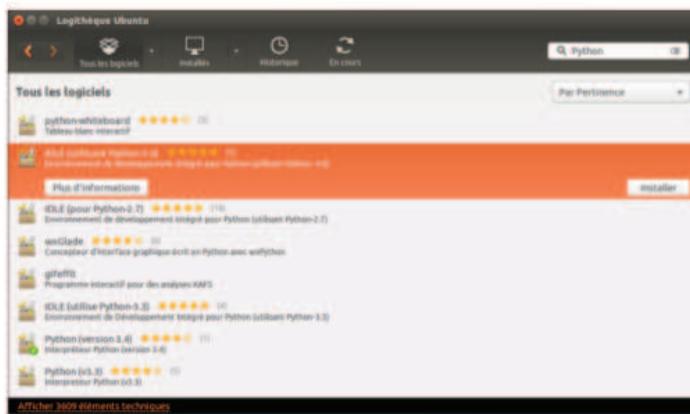
2. Sélectionne **Bureau** dans la boîte de dialogue **Emplacement**, puis clique sur **Enregistrer**.

À présent, passe à la section « Dès que Python est installé » (page 17) pour tes premiers essais.

Installer Python sous Linux (Ubuntu)

Python est préinstallé avec la distribution de Linux (Ubuntu), mais très probablement avec une ancienne version. Les étapes suivantes expliquent comment installer Python 3 sous Ubuntu 14.x.

1. Clique sur le bouton de la Logithèque Ubuntu dans la barre de lanceurs latérale (l'icône ressemble à un sac orange – si tu ne la trouves pas, tu peux toujours cliquer sur l'icône du tableau de bord et saisir **Logiciel** dans la case de recherche).
2. Entre **Python** dans la case de recherche du coin supérieur droit de la Logithèque Ubuntu.
3. Dans la liste de logiciels qui s'affiche, sélectionne la dernière version d'IDLE, qui apparaît sous le nom **IDLE (utilisant Python-3.4)** dans l'exemple.



4. Clique sur **Installer**.
5. Saisis le mot de passe d'administration pour permettre l'installation du logiciel (si tu ne le connais pas, demande à tes parents de le saisir), puis clique sur **S'authentifier**.

NOTE

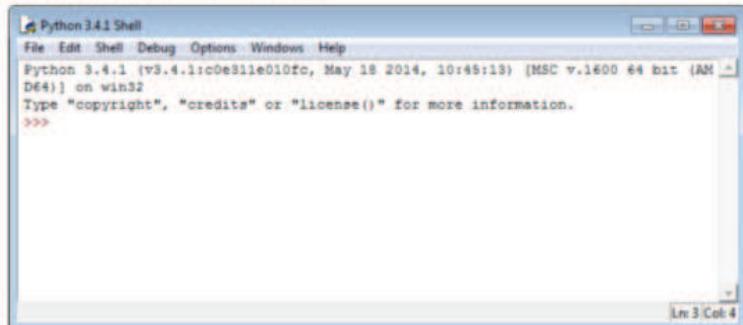
Dans certaines versions de Linux, il se peut que tu ne puisses voir que la version Python (v3.3) dans le menu principal (au lieu d'IDLE). Tu peux installer cette version à la place.

Maintenant que tu dispose de la dernière version de Python installée sur l'ordinateur, nous pouvons l'essayer.

Dès que Python est installé

À ce stade, tu devrais voir une icône **IDLE** sur le Bureau de Windows ou de Mac OS X. Si tu utilises Ubuntu, dans la barre de lanceurs apparaît l'icône **IDLE (using Python-3.4)**. Sous d'autres distributions de Linux, dans le menu **Applications**, il y a normalement un nouveau groupe intitulé **Programmation**, qui contient l'application **IDLE (using Python-3.4)** (ou une version plus récente).

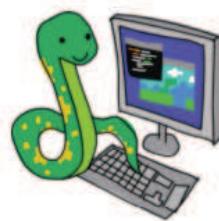
Double-clique sur l'icône ou clique sur l'option de menu pour voir s'afficher la fenêtre suivante :



Cette fenêtre porte le doux nom de **shell Python** ou d'environnement d'exécution de Python. Elle fait partie de l'environnement de développement intégré du langage. Les trois symboles « plus grand que » (**>>>**) sont ce que nous appelons l'« invite de commande ».

Essayons d'entrer quelques commandes à l'invite. Commençons par celle-ci :

```
>>> print("Bonjour tout le monde")
```



Vérifie bien que tu écris les guillemets (" "). Appuie sur la touche **Entrée** du clavier après avoir entré toute la ligne. Si tu as saisi la commande correctement, tu devrais obtenir ceci (la partie en bleu) :

```
>>> print("Bonjour tout le monde")
Bonjour tout le monde
>>>
```



L'invite de commande réapparaît pour te faire savoir que le **shell Python** est prêt à recevoir d'autres commandes.

Félicitations ! Tu as créé ton premier programme. Le mot **print** est une commande de Python que nous désignons sous le nom de **fonction**. Celle-ci imprime, ou plutôt affiche à l'écran, tout ce qui se trouve entre les parenthèses. En fait, tu as donné à l'ordinateur l'instruction d'afficher les mots **Bonjour tout le monde** ; l'ordinateur et toi pouvez tous deux comprendre cette instruction.

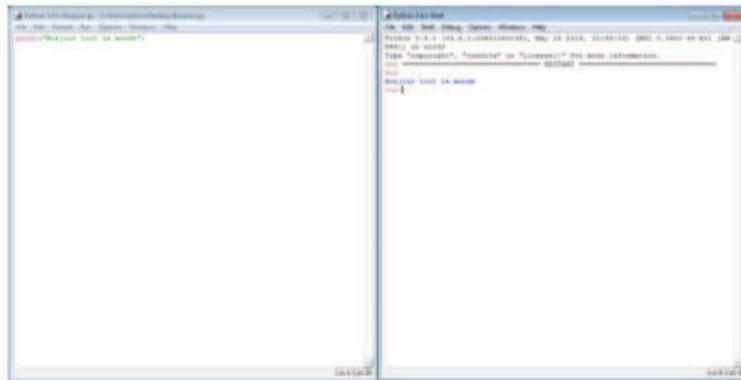
Enregistrer des programmes Python

Les programmes ne seraient pas vraiment utiles s'il fallait que tu les réécrives chaque fois que tu veux les utiliser, même si tu les imprimes sur papier pour pouvoir te les rappeler par la suite. Bon, si la réécriture est possible pour de très courtes suites d'instructions, des programmes plus longs, comme un traitement de texte, peuvent contenir des millions de lignes de code. Si tu imprimes ces lignes de programme sur papier, tu te retrouveras vite avec plus de 100 000 pages ! Imagine que tu doives les entreposer à la maison, en espérant qu'un courant d'air ne vienne pas les faire s'envoler...

Heureusement, tu peux enregistrer tes programmes pour les réutiliser plus tard. Il suffit d'ouvrir une fenêtre IDLE. Pour cela, dans le menu du shell, clique sur **File>New File** (cela signifie **Fichier>Nouveau fichier**). Une nouvelle fenêtre apparaît, avec ***Untitled*** dans la barre de titre. Entre la ligne suivante dans cette fenêtre :

```
print("Bonjour tout le monde")
```

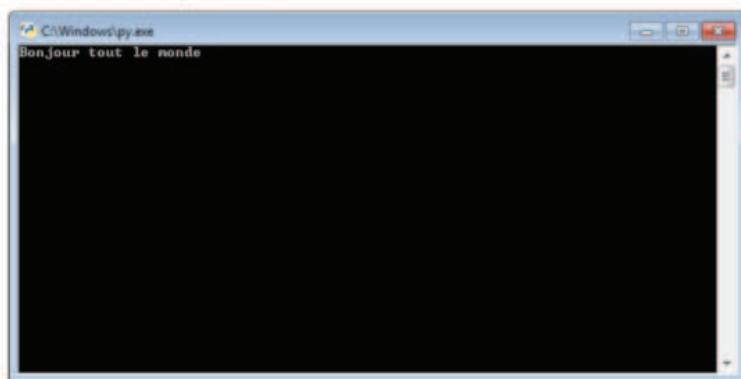
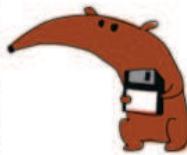
À présent, clique sur **File>Save**(qui signifie **Fichier>Enregistrer**). Lorsque la **boîte de dialogue** te demande un nom de fichier, saisis **Bonjour.py** et navigue jusqu'à ton Bureau. Clique ensuite sur **Enregistrer**. Il reste à **exécuter (run)** le programme : clique sur le menu **Run>Run Module**. Avec un peu de chance, le programme enregistré s'exécute et tu obtiens ceci (fenêtre de droite) :



À ce stade, si tu décides de fermer la fenêtre du shell mais de laisser la fenêtre **Bonjour.py** ouverte, il te suffit de cliquer de nouveau sur **Run>Run Module** pour voir réapparaître le shell Python et le programme s'y exécuter de nouveau. Pour rouvrir le shell Python sans exécuter le programme, clique sur le menu **Run>Python shell**.

Après l'exécution du code, une nouvelle icône apparaît sur ton Bureau, nommée **Bonjour.py**. Si tu double-cliques dessus, une fenêtre à fond noir s'affiche brièvement, puis disparaît. Que s'est-il passé ?

Tu viens d'entrevoir la console en ligne de commande – ou tout simplement console (semblable au **shell**) –, qui démarre, affiche **Bonjour tout le monde**, puis quitte et disparaît. Voici ce que tu aurais pu voir si tu avais été un super-héros avec une vision plus rapide que l'éclair, avant que la fenêtre ne disparaisse :



Outre les menus des fenêtres, tu peux aussi utiliser des raccourcis clavier, c'est-à-dire des combinaisons de touches du clavier, pour créer une nouvelle fenêtre de shell, enregistrer un fichier et exécuter un programme.

- Sous Windows et Ubuntu, **Ctrl+N** (appuie simultanément sur ces deux touches) ouvre une nouvelle fenêtre de shell ; **Ctrl+S** enregistre le fichier en cours lorsque tu as terminé de le modifier ; **F5** lance l'exécution du programme en cours.
- Sous Mac OS X, utilise **⌘-N** pour ouvrir une nouvelle fenêtre de shell ; **⌘+S** enregistre le fichier en cours ; pour exécuter le programme, appuie sur la touche **Fn** (fonction) et maintiens-la enfoncée, puis presse **F5**.

Ce que tu as appris

Dans ce chapitre, nous avons commencé par des choses simples, avec l'application **Bonjour tout le monde** – c'est en fait le programme essayé par presque tout le monde lors de la découverte d'un nouveau langage de programmation informatique. Au chapitre suivant, nous réaliserons des choses un peu plus utiles à partir du shell Python.



CALCULS ET VARIABLES

À présent, Python est installé et tu sais comment démarrer son shell. C'est donc le moment d'en faire quelque chose. Nous allons débuter avec des calculs simples, puis évoluer vers les variables. Les variables fournissent une manière de stocker des choses dans un programme et aident à écrire des programmes vraiment utiles.

Calculer avec Python

Pour trouver le produit de deux nombres, par exemple $8 \times 3,57$, il te faut probablement prendre une calculatrice ou une feuille et un stylo. Et si tu utilisais le **shell Python** pour effectuer ce calcul ? Essayons.

Démarre le shell : double-clique sur l'icône **IDLE** sur le Bureau ou, si tu utilises Ubuntu, sur l'icône **IDLE** dans le lanceur. À l'invite de commande, entre cette opération :

```
>>> 8 * 3.57  
28.56
```

Pour multiplier deux nombres en Python, il faut utiliser l'astérisque (*) au lieu du signe de multiplication habituel (×). Note aussi que le nombre 3,57 s'écrit 3.57, avec un point décimal au lieu de la virgule.

Voyons ce que donne un calcul un peu plus utile.

Imaginons que tu creuses dans le fond du jardin et que tu y trouves un sac contenant 20 pièces d'or. Le lendemain, tu te faufiles à la cave pour les placer dans la machine à dupliquer à vapeur de ton génial inventeur de grand-père (par chance, les 20 pièces y rentrent parfaitement). Tu entends un sifflement et quelques bruits bizarres et, quelques heures plus tard, en sortent 10 nouvelles pièces étincelantes de plus.

Combien de pièces aurais-tu dans ton coffre à trésor si tu faisais cela pendant un an ? Sur le papier, les formules pour le calculer ressembleraient à ceci :

$$10 \times 365 = 3\,650 \text{ et } 20 + 3\,650 = 3\,670$$

Bien évidemment, ces calculs sont faciles à effectuer avec une calculatrice ou sur papier, mais tu peux tout aussi bien les réaliser avec le shell Python. D'abord, il faut multiplier les 10 pièces par les 365 jours de l'année pour obtenir 3650, puis additionner les 20 pièces initiales à ce résultat pour obtenir 3670.

```
>>> 10 * 365  
3650  
>>> 20 + 3650  
3670
```

Et maintenant, que se passe-t-il si un corbeau découvre ton trésor et entre chaque semaine dans ta chambre pour voler 3 pièces ?

Au bout d'une année de ce jeu-là, combien te resterait-il de pièces ? Voici à quoi ressemblent les calculs dans le shell :

```
>>> 3 * 52  
156  
>>> 3670 - 156  
3514
```

D'abord, il faut multiplier **3** par les **52** semaines d'une année, ce qui donne **156** pièces volées. Ensuite, ce nombre doit être soustrait du total de pièces que tu avais (**3670**) ; il te resterait **3514** pièces à la fin de l'année.

Ceci est un programme très simple. Dans ce livre, tu apprendras à étendre de telles idées pour écrire des programmes certainement plus utiles.

Les opérateurs de Python

Dans le shell Python, il est possible de faire des multiplications, des additions, des soustractions et des divisions, parmi bien d'autres opérations mathématiques que nous n'allons pas aborder tout de suite. Les symboles de base utilisés par Python pour les effectuer s'appellent des **opérateurs**. Le tableau 2-1 les énumère.

Tableau 2-1. Opérateurs de base de Python

Symbol	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division

La barre oblique (**/**) sert pour la division, parce qu'elle ressemble à la barre que tu utilises quand tu veux représenter une fraction. Si, par exemple, tu as 100 pirates et 20 gros barils, et que tu veux compter le nombre de pirates qui peuvent se cacher dans chacun, tu divises 100 par 20 ($100 \div 20$) en écrivant dans le shell de Python **100 / 20**. Retiens que la barre oblique est celle qui s'incline vers la droite.



L'ordre des opérateurs

Dans un langage de programmation, nous nous servons souvent des parenthèses pour contrôler l'ordre des opérations. Tout ce qui utilise un opérateur est appelé une « opération ». La multiplication et la division ont un ordre plus élevé que l'addition et la soustraction, ce qui signifie qu'elles sont effectuées en premier. Autrement dit, quand tu saisies une formule de calcul en Python, les multiplications et les divisions ont lieu avant les additions et les soustractions.

Ainsi, pour traduire le calcul de l'exemple suivant, il faut dire « multiplier **30** par **20**, puis ajouter **5** au résultat (**605**) » :

```
>>> 5 + 30 * 20  
605
```

Pour changer l'ordre des opérations, il faut ajouter des parenthèses, par exemple autour des deux premiers nombres :

```
>>> (5 + 30) * 20  
700
```

Le résultat du calcul est **700** et non plus **605** : les parenthèses indiquent à Python qu'il faut d'abord effectuer l'opération à l'intérieur et, ensuite seulement, l'opération en dehors de celles-ci. Cet exemple signifie « ajouter **5** à **30**, puis multiplier le tout par **20** ».

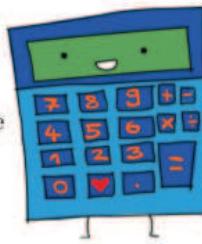
Il est possible d'imbriquer les parenthèses, c'est-à-dire de les placer les unes à l'intérieur des autres, comme ici :

```
>>> ((5 + 30) * 20) / 10  
70.0
```

Dans ce cas, Python évalue (ou calcule) d'abord le contenu des parenthèses les plus à l'intérieur, puis celles les plus à l'extérieur, pour terminer avec l'opérateur de division.

Autrement dit, cette formule demande « d'ajouter **5** à **30**, puis de multiplier le résultat par **20**, puis de diviser le tout par **10** ». Si on la décompose en étapes, cela donne ceci :

- l'ajout de **5** à **30** donne **35** ;
- la multiplication de **35** par **20** donne **700** ;
- la division de **700** par **10** donne la réponse finale, **70**.



Si nous n'avions pas utilisé de parenthèses, le résultat aurait été assez différent :

```
>>> 5 + 30 * 20 / 10  
65.0
```

Ici, **30** est d'abord multiplié par **20** (= 600), puis **600** est divisé par **10** (= 60) et, enfin, **5** est ajouté au tout pour obtenir **65**.

ATTENTION *Retiens que la multiplication et la division ont toujours lieu avant l'addition et la soustraction, à moins que des parenthèses ne soient là pour contrôler l'ordre des opérations.*

Les variables sont comme des étiquettes

En programmation, une **variable** décrit un endroit où ranger des informations telles que des nombres, du texte, des listes de nombres et de texte, et ainsi de suite. Une variable peut être aussi vue comme une étiquette qui désigne quelque chose.

Par exemple, pour créer une variable nommée **fred**, nous utilisons le signe égal (**=**), suivi de l'information que la variable désigne (ou « étiquette »). Dans ce qui suit, nous créons la variable **fred** et nous indiquons à Python qu'elle étiquette le nombre **100**, ce qui ne veut pas dire qu'une autre variable ne peut pas aussi avoir la même valeur :

```
>>> fred = 100
```

Pour savoir quelle valeur une variable étiquette, tape **print** dans le shell, suivi du nom de la variable entre parenthèses, comme ceci :

```
>>> print(fred)  
100
```

Tu peux aussi dire à Python de modifier la variable **fred** pour qu'elle désigne une autre valeur, ici la valeur **200** :

```
>>> fred = 200  
>>> print(fred)  
200
```

À la première ligne, nous disons que **fred** désigne le nombre **200**. À la deuxième, nous demandons ce que désigne **fred**, juste pour vérifier que la modification a bien eu lieu. Python affiche le résultat à la dernière ligne.

Il est aussi possible d'utiliser plusieurs étiquettes (ou variables) pour désigner la même valeur :

```
>>> fred = 200  
>>> jean = fred  
>>> print(jean)  
200
```

Dans cet exemple, nous indiquons à Python d'étiqueter avec le nom (de variable) `jean` la même chose que `fred` à l'aide du signe `=` entre `jean` (à gauche) et `fred` (à droite).

Ceci dit, `fred` n'est pas un nom très utile pour une variable, parce qu'il ne nous dit pas grand-chose à propos de l'usage que nous compsons en faire. Ce serait beaucoup mieux d'appeler notre variable `nombre_de_pieces`, par exemple, au lieu de `fred` :

```
>>> nombre_de_pieces = 200  
>>> print(nombre_de_pieces)  
200
```

C'est bien plus clair puisque, à l'évidence, nous parlons de 200 pièces.

Les noms de **variables** peuvent comporter des lettres, des chiffres et le caractère souligné (`_`), mais ils ne commencent pas par un nombre. Tu peux tout utiliser pour former des noms de variables : d'une simple lettre (comme `a`) jusqu'à de longues phrases, mais évite les caractères accentués du français (`é, à, è`, etc.). Un nom de variable ne contient jamais d'espace ; utilise donc le caractère de soulignement pour séparer les mots. Parfois, pour faire quelque chose de rapide, un nom de variable court est préférable. Le nom que tu choisiras dépend en fait de ce que tu veux qu'il désigne et de la façon dont tu veux l'utiliser dans le code.

Maintenant que tu sais comment créer des variables, voyons comment les employer.

Utiliser les variables

Te rappelles-tu les calculs que nous avons effectués pour connaître le nombre de pièces que tu aurais à la fin de l'année si tu pouvais en créer de nouvelles avec la géniale invention de ton grand-père ? Nous avions cette suite d'opérations :

```
>>> 20 + 10 * 365  
3670  
>>> 3 * 52  
156  
>>> 3670 - 156  
3514
```

qui peuvent se transformer en une seule ligne de code :

```
>>> 20 + 10 * 365 - 3 * 52  
3514
```

À présent, si nous changeons les nombres en variables ? Essaie d'entrer ce qui suit :

```
>>> pieces_trouvees = 20  
>>> pieces_magiques = 10  
>>> pieces_volees = 3
```

Ces trois lignes créent les variables `pieces_trouvees`, `pieces_magiques` et `pieces_volees`.

Dès lors, nous pouvons récrire la formule comme ceci :

```
>>> pieces_trouvees + pieces_magiques * 365 - pieces_volees * 52  
3514
```

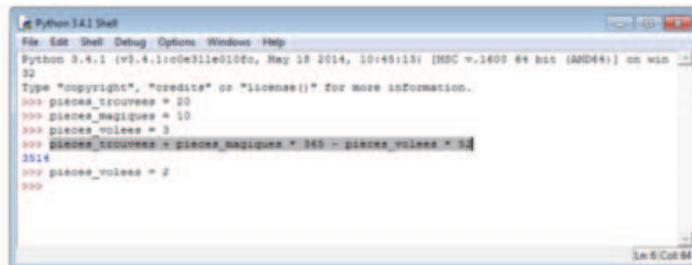
Tu constates que tu obtiens la même réponse ; quel est donc l'intérêt ? En fait, tout réside dans la magie des variables. Que se passerait-il si tu placais un épouvantail devant ta fenêtre et si le corbeau ne pouvait plus voler que deux pièces au lieu de trois ? Comme tu utilises une variable, il suffit d'en changer le contenu pour qu'elle contienne le nouveau nombre et, ensuite, sa valeur sera changée partout où elle est utilisée dans la formule. Pour modifier en 2 le contenu de la variable `pieces_volees`, entre :



```
>>> pieces_volees = 2
```

Ensuite, nous pouvons copier-coller la formule de calcul pour obtenir le nouveau résultat. Voici comment procéder.

1. Sélectionne d'abord le texte à copier : clique avec la souris au début de la ligne (après les `>>>`) et maintiens le bouton pressé, tout en déplaçant la souris jusqu'à la fin de la ligne, comme montré ici.

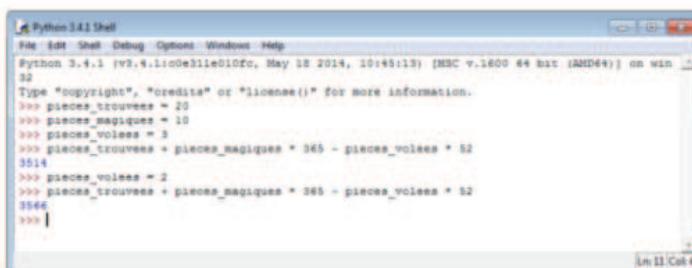


A screenshot of the Python 3.4.1 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Windows, Help. The title bar says "Python 3.4.1 (v3.4.1:10e088e3d0fc, May 18 2014, 10:45:13) [MSC v.1600 64 bit (AMD64)] on win". The code area shows:

```
>>> pieces_trouvees = 20
>>> pieces_magiques = 10
>>> pieces_volees = 3
>>> pieces_trouvees + pieces_magiques * 365 - pieces_volees * 52
3514
>>> pieces_volees = 2
>>> pieces_trouvees + pieces_magiques * 365 - pieces_volees * 52
3566
>>>
```

The status bar at the bottom right indicates "Ln 6 Col 64".

2. Presse et maintiens la touche **Ctrl** enfonceée (ou la touche **⌘** si tu utilises un Mac), presse en même temps la touche **C** pour copier le texte sélectionné (nous écrirons toujours **Ctrl+C** pour désigner cette opération).
3. Clique dans la dernière ligne d'invite de commande, celle après `pieces_volees = 2,`
4. Maintiens de nouveau la touche **Ctrl** enfonceée et, en même temps, appuie sur **V** pour coller le texte sélectionné (nous écrirons **Ctrl+V**).
5. Appuie sur **Entrée** pour voir le résultat du nouveau calcul.



A screenshot of the Python 3.4.1 Shell window, identical to the previous one except for the status bar which now says "Ln 11 Col 4". The code area shows the same calculations as before, but the final result is different due to the modification of the variable `pieces_volees`.

N'est-ce pas bien plus facile que de retaper à la main toute la formule ?

De la même manière, tu peux essayer de modifier les autres variables, puis copier (**Ctrl+C**) et coller (**Ctrl+V**) la formule de calcul pour voir les effets de tes changements. Par exemple, si tu donnes

chaque fois un coup de pied dans la machine à dupliquer de ton grand-père et si elle produit chaque jour 3 pièces de plus, tu obtiendras **4661** pièces à la fin de l'année :

```
>>> pieces_magiques = 13  
>>> pieces_trouvees + pieces_magiques * 365 - pieces_volees * 52  
4661
```

À l'évidence, se servir des variables pour des calculs aussi simples que celui-ci n'est qu'à peine utile. En réalité, nous n'avons pas encore fait grand-chose d'utile jusqu'à présent... Retiens toutefois que les variables offrent une manière d'étiqueter des choses, pour les réemployer après.

Ce que tu as appris

Dans ce chapitre, tu as appris à écrire des formules simples à l'aide d'**opérateurs** de Python et à te servir des parenthèses pour contrôler l'ordre des opérations (l'ordre selon lequel Python évalue les portions de calcul). Tu as ensuite créé des **variables**, que tu as utilisées dans des calculs.



CHAÎNES, LISTES, TUPLES ET DICTIONNAIRES

Jusque-là, nous avons effectué des calculs de base en Python et tu as découvert les **variables**. Dans ce chapitre, nous examinons quelques-uns des autres éléments de programme en Python : tu utiliseras les **chaînes de caractères** pour afficher des messages dans tes **programmes** (comme Prêt ou Partie terminée dans un jeu) ; tu découvriras aussi la manière d'utiliser les listes, les tuples et les dictionnaires pour stocker des collections de choses.

Les chaînes

En programmation, le texte s'appelle généralement **chaîne de caractères**, ou simplement chaîne (tu retrouveras aussi souvent le terme anglais *string*). Pour bien comprendre cette notion, imagine une chaîne comme une suite ou une collection de lettres. Par exemple, toutes les lettres, tous les nombres et tous les symboles de ce livre forment une chaîne, de même que tes nom et adresse. En fait, le premier **programme** en Python que tu as créé au chapitre 1, [Bonjour tout le monde](#), utilisait déjà une chaîne.

Créer des chaînes

Pour créer une chaîne en Python, il faut placer des guillemets verticaux ("") autour du texte, parce que les langages de programmation ont besoin de distinguer différents types de valeurs. Tu dois indiquer à l'ordinateur si une valeur est un nombre, une chaîne ou autre chose. Par exemple, reprenons notre **variable** `fred` du chapitre 2 et utilisons-la pour étiqueter une chaîne :

```
fred = "Les gorilles ont de grosses narines, pourquoi ? Parce qu'ils ont de gros doigts !"
```

Ensuite, pour voir ce que contient `fred`, entrons `print(fred)` :

```
>>> print(fred)
Les gorilles ont de grosses narines, pourquoi ? Parce qu'ils ont de gros doigts !
```

Tu peux aussi te servir des apostrophes verticales ('') pour créer une chaîne, comme suit :

```
>>> fred = 'Elle est rose et à peluche ? Une peluche rose !'
>>> print(fred)
Elle est rose et à peluche ? Une peluche rose !
```

En revanche, si tu essaies de saisir plusieurs lignes de texte dans ta chaîne avec une seule apostrophe ou un seul guillemet, ou si tu commences avec l'une et termines avec l'autre, alors le **shell Python** t'adresse un message d'erreur. Entre, par exemple, la ligne suivante :

```
>>> fred = "Je suis rouge, je monte et je descends. Qui suis-je ?"
```

Tu obtiens le résultat suivant :

SyntaxError: EOL while scanning string literal

Ce message signale une **erreur de syntaxe**, parce que tu n'as pas respecté les règles de fermeture d'une chaîne par une apostrophe ou un guillemet.

La **syntaxe** est l'arrangement et l'ordre des mots dans une phrase ou, dans ce cas précis, la disposition et l'ordre des mots et des symboles dans le programme. Ainsi, `SyntaxError` indique que tu as fait quelque chose et dans un ordre auquel Python ne s'attendait pas, ou que le langage pensait voir quelque chose que tu as oublié de mettre. Dans ce message d'erreur, `EOL` représente la fin de ligne (*end-of-line*), tandis que le reste explique que Python a atteint la fin de la ligne et n'a pas trouvé le guillemet attendu de fermeture de la chaîne.

Pour avoir plus d'une ligne de texte dans une chaîne (ce qui s'appelle une « chaîne multiligne »), utilisez plutôt trois apostrophes ('''), puis appuyez sur `Entrée` et entrez les lignes comme suit :

```
>>> fred = '''Je suis rouge, je monte et je descends. Qui suis-je?  
Une tomate dans un ascenseur !'''
```

Affichons le contenu de `fred` pour vérifier que cela fonctionne :

```
>>> print(fred)  
Je suis rouge, je monte et je descends. Qui suis-je ?  
Une tomate dans un ascenseur !
```

Gérer les problèmes de chaînes

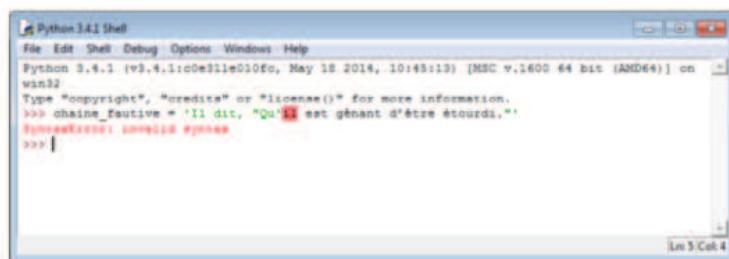
Prenons à présent un exemple problématique d'une chaîne qui provoque l'affichage d'un message d'erreur :

```
>>> chaine_fautive = 'Il dit, "Qu'il est gênant d'être étourdi."  
SyntaxError: invalid syntax
```

À la première ligne, nous essayons de créer une chaîne, définie comme la variable `chaine_fautive`, entourée d'apostrophes ('), mais qui contient aussi un mélange d'apostrophes dans les mots `Qu'il` et `d'être`, en plus de guillemets (""). Quel fouillis !

Rappelle-toi que Python n'est pas en soi aussi intelligent qu'un être humain : tout ce qu'il voit, c'est une chaîne qui contient `Il dit`, `"Qu,` suivi d'un tas d'autres caractères auxquels il ne s'attend pas. Lorsqu'il voit une apostrophe ou un guillemet, il comprend qu'une

chaîne commence et s'attend donc à ce qu'elle se termine après l'apostrophe ou le guillemet correspondant qui suit sur la même ligne. Dans ce cas-ci, le début de la chaîne est indiqué par une apostrophe ('), juste avant **I**, tandis que la fin de la chaîne, du moins en ce qui concerne Python, est indiquée par l'apostrophe juste après le **u** de **Qu**. Heureusement, IDLE met en évidence l'endroit où les choses ne vont plus :



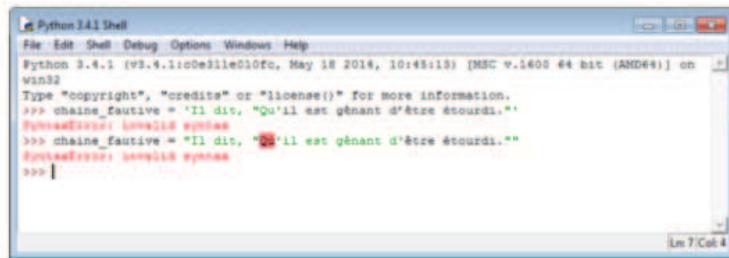
```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:735f1137e07d, May 16 2014, 10:45:13) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> chaine_fautive = 'Il dit, "Qu'ul' est gênant d'être étourdi."
SyntaxError: invalid syntax
>>>
```

La dernière ligne d'IDLE t'indique l'erreur qui s'est produite, soit ici une erreur de syntaxe.

L'utilisation de guillemets verticaux ("") au lieu d'apostrophes ('') produit encore une erreur :

```
>>> chaine_fautive = "Il dit, "Qu'il est gênant d'être étourdi."
SyntaxError: invalid syntax
```

Ici, Python voit une chaîne entourée de guillemets verticaux, qui contient les lettres **I**l **d**it, (suivies d'un espace). Tout ce qui suit cette chaîne provoque l'**erreur** :



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:735f1137e07d, May 16 2014, 10:45:13) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> chaine_fautive = 'Il dit, "Qu'il est gênant d'être étourdi.'
SyntaxError: invalid syntax
>>> chaine_fautive = "Il dit, "Qu'il est gênant d'être étourdi."
SyntaxError: invalid syntax
>>>
```

Le problème est que, selon le point de vue de Python, la chaîne s'arrête aux premiers guillemets (ou à la première apostrophe) et ne comprend pas ce tu as voulu faire avec ce qui suit sur la même ligne.

La solution se trouve dans la chaîne multiligne, vue plus haut, avec les trois apostrophes verticales (''''), qui permettent de combiner les guillemets et les apostrophes dans les chaînes de caractères sans provoquer d'erreur. En fait, entre les paires de triples apostrophes, tu peux placer n'importe quelle combinaison de guillemets et d'apostrophes, à condition, bien entendu, de ne pas essayer de glisser trois apostrophes de suite dans cette chaîne de caractères ! Donc, la chaîne correcte, ou en tout cas sans erreur, serait :



```
>>> chaine_correcte = '''Il dit, "Qu'il est gênant d'être étourdi."'''
```

Toutefois, il y a mieux. Si tu veux vraiment utiliser des guillemets ou des apostrophes au lieu des triples apostrophes, alors tu peux faire appel à la barre oblique inversée (\) avant chaque apostrophe ou guillemet que comporte la chaîne. Tu utilises dans ce cas un caractère d'échappement. Il dit à Python que tu sais qu'il y a des apostrophes ou des guillemets dans la chaîne mais que tu veux qu'il les ignore, jusqu'à atteindre la fin de la chaîne.

Les caractères d'échappement compliquent la lecture du contenu des chaînes. C'est pourquoi il est sans doute préférable d'utiliser les triples apostrophes. Mais attention, dans certains cas, comme dans des petits extraits de code, il est parfois nécessaire d'y recourir. Par conséquent, il est important que tu saches à quoi servent ces barres obliques inversées.

Voici quelques exemples qui montrent comment cela fonctionne :

```
❶ >>> apos_str = 'Il dit, "Qu\'il est gênant d\'être étourdi."'
❷ >>> guil_str = "Il dit, \"Qu'il est gênant d'être étourdi.\""
>>> print(apos_str)
Il dit, "Qu'il est gênant d'être étourdi."
>>> print(guil_str)
Il dit, "Qu'il est gênant d'être étourdi."
```

D'abord, en ❶, nous créons une chaîne entourée d'apostrophes et plaçons une barre oblique inversée devant chaque apostrophe contenue dans la chaîne, tandis qu'en ❷, nous réalisons une chaîne entourée de guillemets et plaçons la barre oblique inversée devant chaque guillemet. Dans les lignes suivantes, nous affichons les variables. Remarque bien que les barres obliques inversées n'appa-

raissent pas dans les chaînes affichées. Note aussi les `_str` à la fin des noms de variables. Il s'agit d'une astuce toute simple qui indique que ces variables contiennent a priori des chaînes.

Insérer des valeurs dans des chaînes

Pour afficher un message avec le contenu d'une variable, il est possible d'insérer, ou d'intégrer, une valeur de chaîne dans la chaîne du message, grâce à `%s` qui sert en quelque sorte de marqueur pour une valeur à ajouter plus tard. **Intégrer des valeurs** est une manière qu'ont les programmeurs de dire « insérer une valeur ici ». Si, par exemple, tu veux qu'un programme Python calcule et stocke le nombre de points obtenus dans un jeu, pour l'ajouter plus tard dans une phrase du genre « tu as obtenu ____ points », insère `%s` à la place de la valeur, puis indique à Python où aller chercher cette valeur, comme ceci :

```
>>> monscore = 1000
>>> message = 'Tu as obtenu %s points'
>>> print(message % monscore)
Tu as obtenu 1000 points
```

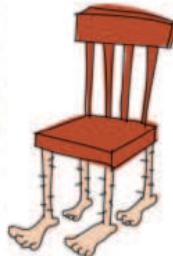
Ici, nous avons créé une variable `monscore` avec la valeur `1000` et la variable `message` qui contient les mots `Tu as obtenu %s points`, où `%s` est ce que nous appelons un espace réservé pour le nombre de points, c'est-à-dire un indicateur de l'endroit dans la chaîne où devra apparaître ce nombre de points. À la ligne suivante, le `print(message)` comporte le symbole `%` pour dire à Python de remplacer le `%s` du contenu du message par la valeur de la variable `monscore`. Le résultat à l'affichage donne `Tu as obtenu 1000 points`. Note qu'il n'est pas nécessaire de passer par une variable pour donner la valeur : nous aurions pu écrire simplement `print(message % 1000)`.

Il est aussi possible de passer des valeurs différentes pour le caractère générique `%s`, à l'aide de variables différentes, comme dans l'exemple suivant :

```
>>> texte_blaque = '%s : outil pour trouver les meubles dans le noir'
>>> partiecorps1 = 'Genou'
>>> partiecorps2 = 'Tibia'
>>> print(texte_blaque % partiecorps1)
Genou : outil pour trouver les meubles dans le noir
>>> print(texte_blaque % partiecorps2)
Tibia : outil pour trouver les meubles dans le noir
```

Ici, nous créons trois variables. La première, `texte_blaque`, reçoit la chaîne avec l'indicateur `%s`. Les deux autres sont `partiecorps1` et `partiecorps2`. Nous affichons `texte_blaque`, de nouveau suivi de l'**opérateur %** pour y remplacer le `%s` par le contenu des variables `partiecorps1` et `partiecorps2` et générer des messages différents.

Nous pouvons aussi utiliser plusieurs caractères génériques dans une chaîne, comme ceci :



```
>>> nombres = 'Que dit un %s à un %s ? Jolie ceinture !'
>>> print(nombres % (0, 8))
Que dit un 0 à un 8 ? Jolie ceinture !
```

Multiplier des chaînes

Que vaut 10 multiplié par 5 ? La réponse est bien entendu 50. Mais alors, que vaut 10 multiplié par a ? Python suggère une réponse :

```
>>> print(10 * 'a')
aaaaaaaaaa
```

Un programmeur Python peut exploiter cette approche pour aligner des chaînes avec un nombre déterminé d'espaces lors de l'affichage de messages dans le shell. Essayons, par exemple, d'afficher une lettre dans le shell. Sélectionne le menu **File>New File**, puis saisis le code suivant :

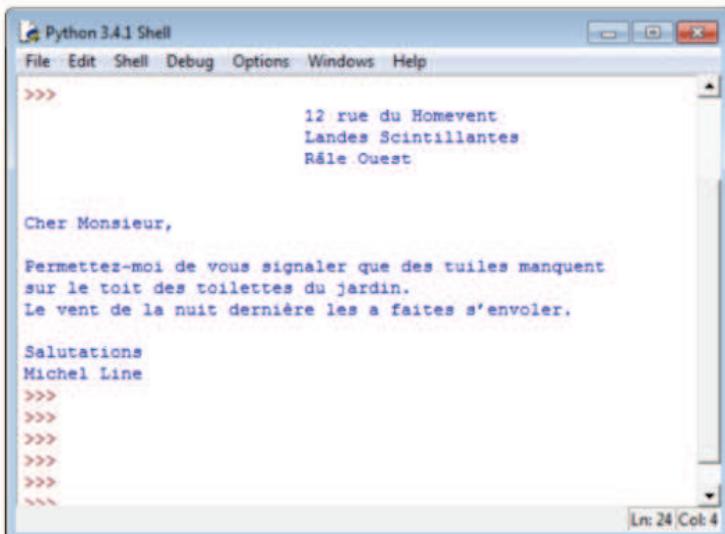
```
espaces = ' ' * 25
print('%s 12 rue du Homevent' % espaces)
print('%s Landes Scintillantes' % espaces)
print('%s Vent d'Ouest' % espaces)
print()
print()
print('Cher Monsieur,')
print()
print('Permettez-moi de vous signaler que des tuiles manquent')
print('sur le toit des toilettes du jardin.')
print("Le vent de la nuit dernière les a faites s'envoler.")
print()
print('Salutations')
print('Michel Line')
```

Dès que tu as tapé ces lignes de code dans la fenêtre d'IDLE, sélectionne le menu **File>Save As** et nomme le fichier `malettre.py`.

NOTE

À partir d'ici, lorsque tu verras **Enregistrer sous : unnomdefichier.py**, au-dessus d'un extrait de code, tu sauras que tu devras, comme indiqué dans l'exemple précédent, cliquer sur le menu **File>New File**, entrer le code dans la fenêtre qui apparaît, puis enregistrer ce fichier avec le nom donné.

À la première ligne de cet exemple, nous créons la variable **espaces** à partir d'un caractère espace multiplié par **25**. Nous utilisons ensuite cette variable dans les trois lignes d'adresses suivantes pour aligner le texte sur la droite du shell. Le résultat obtenu à l'affichage des **print** est le suivant :



The screenshot shows the Python 3.4.1 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The command line starts with '>>>'. The output consists of several lines of text, all right-aligned. It includes an address ('12 rue du Homevent Landes Scintillantes Râle Ouest'), a salutation ('Cher Monsieur,'), a message ('Permettez-moi de vous signaler que des tuiles manquent sur le toit des toilettes du jardin. Le vent de la nuit dernière les a faites s'envoler.'), and a signature ('Salutations Michel Line'). There are also several blank lines and the text 'Ln: 24 Col: 4' at the bottom right.

Non seulement, la multiplication d'un caractère permet d'aligner du texte, mais nous pouvons aussi remplir la fenêtre du shell de messages totalement inutiles. Essaie, par exemple, ceci :

```
>>> print(1000 * 'snif')
```

Plus puissantes que les chaînes : les listes

La liste de courses « pattes d'araignée, orteil de crapaud, œil de triton, aile de chauve-souris, beurre de limace et écailles de serpent »

n'est pas d'un genre tout à fait habituel, sauf si tu es sorcier à tes heures ! Nous allons l'utiliser comme premier exemple pour illustrer les différences entre les chaînes et les listes.

Nous pourrions enregistrer cette liste d'éléments dans la variable `liste_sorcier` à l'aide d'une chaîne comme ceci :

```
>>> liste_sorcier = "pattes d'araignée, orteil de crapaud, ↪  
    oeil de triton, aile de chauve-souris, ↪  
    beurre de limace, écailles de serpent"  
>>> print(liste_sorcier)  
pattes d'araignée, orteil de crapaud, oeil de triton,  
aile de chauve-souris, beurre de limace, écailles de  
serpent
```

Cependant, il est aussi possible de créer une liste, un type d'objet assez magique de Python que nous pouvons ensuite manipuler. Voici à quoi ressembleraient les éléments, écrits sous forme de liste :



```
>>> liste_sorcier = ["pattes d'araignée", 'orteil de crapaud', ↪  
    'oeil de triton', 'aile de chauve-souris', ↪  
    'beurre de limace', 'écailles de serpent']  
>>> print(liste_sorcier)  
['pattes d'araignée', 'orteil de crapaud', 'oeil de triton', 'aile de  
chauve-souris', 'beurre de limace', 'écailles de serpent']
```

ATTENTION *Dans les deux cas, nous avons pattes d'araignée avec une apostrophe. Nous avons vu dans la section précédente que nous devons soit entourer la chaîne avec des guillemets, soit placer un caractère d'échappement devant l'apostrophe dans la chaîne elle-même ('pattes d\'araignée'). En fait, si tu adoptes la solution du caractère d'échappement, au moment de l'affichage de la liste du deuxième exemple, Python convertit automatiquement 'pattes d\'araignée' en "pattes d'araignée", tandis que les autres éléments sont affichés entourés d'apostrophes, comme définis initialement.*

La création d'une liste demande un peu plus d'efforts, mais elle offre plus de possibilités qu'une chaîne parce qu'il est possible d'en manipuler les éléments. Par exemple, affichons le troisième élément de `liste_sorcier`, soit '`oeil de triton`', en donnant sa position, c'est-à-dire l'indice dans la liste, entre crochets (`[]`), comme ceci :

```
>>> print(liste_sorcier[2])  
œil de triton
```

Pardon ? Mais c'est le troisième élément de la liste ?! C'est vrai, mais les indices des listes commencent toujours par zéro : le premier élément d'une liste a l'indice **0**, le deuxième **1** et le troisième **2**. Même si ce n'est pas toujours clair pour les êtres humains, ça l'est pour les ordinateurs !

Il est également possible de modifier un élément d'une liste, bien plus facilement que dans une chaîne. Ainsi, si au lieu de l'œil de triton, il nous faut une langue d'escargot dans notre liste, voici comment faire (remarque l'usage des guillemets et des apostrophes) :

```
>>> liste_sorcier[2] = "langue d'escargot"  
>>> print(liste_sorcier)  
["pattes d'araignée", 'orteil de crapaud', "langue d'escargot", 'aile de chauve-souris', 'beurre de limace', 'écaillles de serpent']
```

Ceci remplace l'élément situé à l'indice **2** dans la liste (donc à la troisième position), anciennement l'œil de triton, par une langue d'escargot.

Peut-être veux-tu maintenant afficher un sous-ensemble d'une liste ? Pour ce faire, utilise le deux-points (**:**) dans les crochets droits. Par exemple entre ce qui suit pour trouver les ingrédients, du troisième au cinquième de la liste (pour constituer un savoureux sandwich) :



```
>>> print(liste_sorcier[2:5])  
['langue d'escargot', 'aile de chauve-souris', 'beurre de limace']
```

Écrire **[2:5]** revient au même que de dire « montre la plage des éléments depuis l'indice **2** jusque, mais non compris, l'indice **5** » – autrement dit, les éléments **2**, **3** et **4**.

Les listes stockent toutes sortes d'autres éléments, comme des nombres :

```
>>> quelques_nombres = [1, 2, 5, 10, 20]
```

mais également des chaînes :

```
>>> quelques_chaines = ['Ce', "n'est", 'pas', 'sorcier']
```

ou encore un mélange de nombres et de chaînes :

```
>>> nombres_et_chaines = [2, 2, 'chooses', "l'", 1, 6, 'C', 'rond',
                           'C', 'pas', 'carré']
>>> print(nombres_et_chaines)
[2, 2, 'chooses', "l'", 1, 6, 'C', 'rond', 'C', 'pas', 'carré']
```

Et les listes peuvent même contenir d'autres listes :

```
>>> nombres = [1, 2, 3, 4]
>>> chaines = ["J'ai", 'cogné', 'mon', 'orteil', 'et', 'ça', 'fait', 'mal']
>>> maliste = [nombres, chaines]
>>> print(maliste)
[[1, 2, 3, 4], ["J'ai", 'cogné', 'mon', 'orteil', 'et', 'ça', 'fait', 'mal']]
```

Ce dernier exemple de listes dans une liste crée trois variables : `nombres` avec quatre nombres, `chaines` avec huit chaînes et `maliste`, qui contient `nombres` et `chaines`. La troisième liste, `maliste`, ne comporte en réalité que deux éléments, parce que c'est une liste de noms de variables et non une liste du contenu de ces variables.

Ajouter des éléments à une liste

Pour ajouter des éléments à une liste, tu dispose de la fonction `append`. Une **fonction** est un extrait de code qui indique à Python de faire quelque chose de précis. Dans ce cas-ci, `append` ajoute un élément tout à la fin de la liste.

Par exemple, pour ajouter un rot d'ours (pourvu que tu puisses en trouver un !) à la liste de courses du sorcier, il suffit de faire ceci :

```
>>> liste_sorcier.append("rot d'ours")
>>> print(liste_sorcier)
["pattes d'araignée", 'orteil de crapaud', "langue d'escargot", 'aile de
chauve-souris', 'beurre de limace', 'écailles de serpent', "rot d'ours"]
```

Tu peux encore ajouter d'autres éléments de magie à la liste du sorcier de la même manière :

```
>>> liste_sorcier.append('mandragore')
>>> liste_sorcier.append('cigué')
>>> liste_sorcier.append('gaz des marais')
```

La liste du sorcier devient la suivante :

```
>>> print(liste_sorcier)
["pattes d'araignée", "orteil de crapaud", "langue d'escargot", 'aile de
chauve-souris', 'beurre de limace', 'écaillles de serpent', "rot d'ours",
'mandragore', 'ciguë', 'gaz des marais']
```

Il est clair que notre sorcier est prêt à concocter une potion magique plus qu'efficace !

Supprimer des éléments d'une liste

Pour supprimer des éléments d'une liste, utilise le **mot-clé del** (abréviation de *delete* qui signifie « supprimer »). Ainsi, pour supprimer le sixième élément de la liste du sorcier, **écaillles de serpent**, saisis ceci :

```
>>> del liste_sorcier[5]
>>> print(liste_sorcier)
["pattes d'araignée", "orteil de crapaud", "langue d'escargot", 'aile de
chauve-souris', 'beurre de limace', "rot d'ours", 'mandragore', 'ciguë',
'gaz des marais']
```

NOTE

Pour rappel, l'indice commence à zéro pour le premier élément donc `liste_sorcier[5]` indique le sixième élément de la liste.

Et voici comment supprimer les éléments que nous avons ajoutés tout à la fin (mandragore, ciguë et gaz des marais) :

```
>>> del liste_sorcier[8]
>>> del liste_sorcier[7]
>>> del liste_sorcier[6]
>>> print(liste_sorcier)
["pattes d'araignée", "orteil de crapaud", "langue d'escargot", 'aile de
chauve-souris', 'beurre de limace', "rot d'ours"]
```

Arithmétique de liste

Il est possible de regrouper des listes en les additionnant, comme pour l'addition de nombres, à l'aide du signe plus (+). Ainsi, si nous avons deux listes, `liste1` avec les nombres 1 à 4 et `liste2` avec quelques mots, alors nous pouvons les joindre et les afficher à l'aide de `print` et du signe +, comme ceci :

```
>>> liste1 = [1, 2, 3, 4]
>>> liste2 = ["J'ai", 'trébuché', 'et', 'je', 'suis', 'tombé']
>>> print(liste1 + liste2)
[1, 2, 3, 4, "J'ai", 'trébuché', 'et', 'je', 'suis', 'tombé']
```

Additionnons maintenant les deux listes et plaçons le résultat dans une autre variable :

```
>>> liste1 = [1, 2, 3, 4]
>>> liste2 = ['Quand', 'ça', 'va', 'mal', 'je', 'mange', 'du', 'chocolat']
>>> liste3 = liste1 + liste2
>>> print(liste3)
[1, 2, 3, 4, 'Quand', 'ça', 'va', 'mal', 'je', 'mange', 'du', 'chocolat']
```

ou multiplions une liste par un nombre. Ainsi, si tu multiplies `liste1` par `5`, tu écris `liste1 * 5` et tu obtiens :

```
>>> liste1 = [1, 2]
>>> print(liste1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2]
```

Cela indique en fait à Python de répéter cinq fois la liste `liste1`.

En revanche, la division (`/`) et la soustraction (`-`) ne donnent que des erreurs, comme le montrent ces exemples :

```
>>> liste1 / 20
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    liste1 / 20
TypeError: unsupported operand type(s) for /: 'list' and 'int'

>>> liste1 - 20
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    liste1 - 20
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

Pourquoi cela ? En fait, la jointure de deux listes par `+` et la répétition d'une liste avec `*` sont des opérations évidentes, qui vont suffisamment de soi pour que Python les effectue. Et ces opérations prennent aussi un sens dans le monde réel. Par exemple, si tu as deux listes de courses et que tu décides de les regrouper, donc de les additionner, tu pourrais recopier à la main tous les éléments des deux listes sur une nouvelle, ou les joindre bout à bout. De même, si

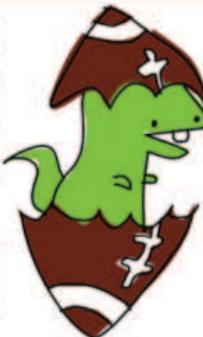
tu décides de multiplier une liste de courses par 3, par exemple, cela reviendrait à la recopier trois fois sur du papier (ou à la photocopier).

Maintenant, que signifierait diviser une liste ? Imagine, par exemple, quel sens cela aurait de diviser une liste de six nombres (de 1 à 6) par deux ? En fait, ce serait possible au moins de trois manières différentes :

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

Tu pourrais séparer la liste en deux au milieu, après le premier élément ou prendre un point de séparation au hasard. La réponse n'est donc pas simple et, quand tu demandes à Python de diviser une liste, il ne sait pas du tout quoi faire. C'est pour cela qu'il répond par une erreur.

La même chose vaut aussi pour l'addition d'une liste à n'importe quoi d'autre qu'une liste. Ce n'est pas possible non plus. Voici, par exemple, ce qui se produit si tu ajoutes le nombre 50 à `listel` :



```
>>> listel + 50
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    listel + 50
TypeError: can only concatenate list (not "int") to list
```

Pourquoi une erreur se produit-elle ? La question est plutôt : que signifie « ajouter 50 à une liste » ? Faut-il ajouter 50 à chacun de ses éléments ? Mais que faire si parmi ces éléments, certains ne sont pas des nombres ? Faut-il simplement ajouter 50 à la fin (comme le ferait `append`) ou au début de la liste ?

En programmation, les commandes doivent toujours fonctionner exactement de la même manière, chaque fois que tu les saisies. L'ordinateur est un peu bête : il ne voit les choses qu'en noir ou blanc. Alors, si tu lui demandes de prendre des décisions trop compliquées, il se défend en levant les bras au ciel, autrement dit en affichant des messages d'erreur.

Tuples

Un tuple, ou n-uplet, ressemble à une liste qui utiliserait des parenthèses. En voici un exemple :

```
>>> sfib = (0, 1, 1, 2, 3)
>>> print(sfib[3])
2
```

Nous définissons ici une variable `sfib` qui contient les nombres **0, 1, 1, 2 et 3**. Ensuite, comme pour une liste, nous affichons l'élément d'indice **3** du tuple à l'aide de `print(sfib[3])`.

La principale différence entre un tuple et une liste est que celui-ci ne peut plus changer à partir du moment où il a été créé. Par exemple, si nous essayons de remplacer la première valeur du tuple `sfib` par le nombre **4**, comme nous avons remplacé des valeurs dans notre liste `liste_sorcier`, nous obtenons un message d'erreur :

```
>>> sfib[0] = 4
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    sfib[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Dès lors, quel est l'intérêt d'utiliser un tuple plutôt qu'une liste ? Fondamentalement parce que, dans certains cas, il est utile d'employer quelque chose dont on sait qu'il ne change jamais. Si tu crées un tuple contenant deux éléments, il aura toujours ces deux éléments au sein de lui.

Dictionnaires

En Python, un *dict* (également désigné par *map*), abréviation de dictionnaire, est une collection de choses, comme les listes et les tuples. Sa principale particularité est que chacun de ses éléments possède une clé et une valeur correspondante. Un dictionnaire peut être comparé à une association entre une clé et une valeur, donc à une « application » au sens mathématique du terme.

Prenons l'exemple de personnes et de leurs sports favoris. Nous pourrions taper ces informations dans une liste, avec le nom de chaque personne suivi de son sport favori, comme ceci :

```
>>> sports_favoris = ['Ralph Williams', 'Football', 'Michael Tippett', 'Basket-ball', 'Edward Elgar', 'Base-ball', 'Rebecca Clarke', 'Volley-ball', 'Ethel Smyth', 'Badminton', 'Frank Bridge', 'Rugby']
```

Si je te demande quel est le sport préféré de Rebecca Clarke, il te reste à fouiller dans cette liste pour trouver que la réponse est le volley-ball. Mais imagine le travail de recherche, si la liste comportait une centaine de personnes, voire plus !

À présent, stockons ces informations dans un dictionnaire, avec le nom de personne comme clé et son sport préféré comme valeur. Le code Python prend l'allure suivante :

```
>>> sports_favoris = {'Ralph Williams' : 'Football', 'Michael Tippett' : 'Basket-ball', 'Edward Elgar' : 'Base-ball', 'Rebecca Clarke' : 'Volley-ball', 'Ethel Smyth' : 'Badminton', 'Frank Bridge' : 'Rugby'}
```

Les deux-points (:) servent à séparer chaque clé de sa valeur, tandis que chaque clé ou valeur est entourée d'apostrophes. Remarque aussi que l'ensemble des éléments d'un dictionnaire est entouré d'accolades ({}) lors de sa création, et non plus de parenthèses ni de crochets.

Le résultat est un dictionnaire (chaque clé s'associe à une valeur déterminée), comme dans le tableau 3-1.

Tableau 3-1. Les clés pointent vers des valeurs du dictionnaire des sports préférés.

Clé	Valeur
Ralph Williams	Football
Michael Tippett	Basket-ball
Edward Elgar	Base-ball
Rebecca Clarke	Volley-ball
Ethel Smyth	Badminton
Franck Bridge	Rugby

À présent, pour trouver le sport préféré de Rebecca Clarke, tu peux accéder au dictionnaire `sports_favoris` en précisant son nom en guise de clé, comme ceci :

```
>>> print(sports_favoris['Rebecca Clarke'])
Volley-ball
```

Pour supprimer une valeur d'un dictionnaire, utilise sa clé. Par exemple, voici comment supprimer Ethel Smyth :

```
>>> del sports_favoris['Ethel Smyth']
>>> print(sports_favoris)
{'Rebecca Clarke': 'Volley-ball', 'Michael Tippett': 'Basket-ball',
'Ralph Williams': 'Football', 'Edward Elgar': 'Base-ball', 'Frank
Bridge': 'Rugby'}
```

Pour remplacer une valeur, utilise aussi sa clé :

```
>>> sports_favoris['Ralph Williams'] = 'Hockey sur glace'
>>> print(sports_favoris)
{'Rebecca Clarke': 'Volley-ball', 'Michael Tippett': 'Basket-ball',
'Ralph Williams': 'Hockey sur glace', 'Edward Elgar': 'Base-ball', 'Frank
Bridge': 'Rugby'}
```

Nous remplaçons le sport favori `Football` par le `Hockey sur glace`, à partir de la clé `Ralph Williams`.

Comme tu peux le constater, travailler avec des dictionnaires équivaut à peu près à manipuler des listes ou des tuples, sauf que tu ne peux pas joindre des dictionnaires à l'aide de l'opérateur plus (+). Si tu essaies, tu obtiens un message d'erreur :

```
>>> sports_favoris = {'Rebecca Clarke': 'Volley-ball', ✎
    'Michael Tippett' : 'Basket-ball', ✎
    'Ralph Williams' : 'Hockey sur glace', ✎
    'Edward Elgar' : 'Base-ball', ✎
    'Frank Bridge' : 'Rugby'}
>>> couleurs_favorites = {'Malcolm Warner' : 'Pois roses', ✎
    'James Baxter' : 'Rayures orange', ✎
    'Sue Lee' : 'Cachemire pourpre'}
>>> sports_favoris + couleurs_favorites
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Pour Python, la jonction de dictionnaires n'a aucun sens ; il lève donc les bras bien haut !

Ce que tu as appris

Dans ce chapitre, tu as vu la manière dont Python utilise des **chaînes** pour stocker du texte, des listes et des tuples pour manipuler des éléments multiples. Tu as appris que les éléments d'une liste peuvent être modifiés et que tu peux joindre une liste à une autre, mais que les valeurs d'un tuple ne peuvent plus changer. Tu as vu aussi comment utiliser des dictionnaires pour stocker des valeurs avec les clés qui les identifient.

Puzzles de programmation

Voici quelques expériences à tenter par toi-même. Les réponses sont disponibles sur le site d'accompagnement du livre.

1. Favoris

Crée une liste de tes loisirs favoris et donne-lui le nom de variable **jeux**. Crée ensuite une liste de tes nourritures préférées et nomme la variable **nourritures**. Joins les deux listes et appelle le résultat **favoris**. Enfin, affiche le contenu de la variable **favoris**.

2. Compter les combattants

Supposons que tu aies 3 bâtiments avec 25 ninjas cachés sur le toit de chaque bâtiment, ainsi que 2 tunnels avec 40 samouraïs cachés dans chaque tunnel. Combien de ninjas et de samouraïs sont prêts à se battre ? Une seule formule suffit dans le shell Python pour calculer cela.

3. Salutations

Crée deux variables : l'une pointe vers ton prénom et l'autre vers ton nom de famille. Réalise ensuite une chaîne qui utilise des espaces réservés pour afficher ton nom dans un message avec ces deux variables, du genre **Bonjour, Kévin Costume !**.



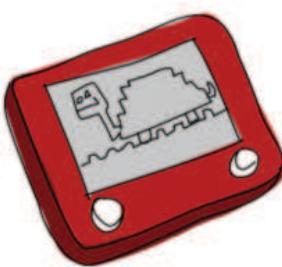
DESSINER AVEC UNE TORTUE

En Python, une tortue est un outil bien pratique, qui imite l'animal du monde réel. Nous connaissons la tortue comme étant un reptile qui se déplace très lentement et transporte sa maison sur son dos. Dans le monde de Python, il s'agit d'une petite flèche noire qui se déplace lentement à l'écran. En vérité, du fait que la tortue de Python laisse une trace tout au long de ses déplacements à l'écran, il s'agirait plutôt d'un escargot ou d'une limace.

La tortue est une sympathique façon d'apprendre les bases du graphisme en informatique. C'est pour cela que nous allons l'utiliser pour dessiner quelques traits et formes simples.

Utiliser le module `turtle` de Python

En Python, un **module** est une manière de fournir du code de programmation intéressant pour l'utiliser dans un autre programme car, entre autres choses, il contient des fonctions que l'on peut réutiliser. Nous en apprendrons plus au chapitre 7, mais Python possède un module spécial, appelé `turtle`, que nous pouvons de suite exploiter pour apprendre comment les ordinateurs dessinent des images sur un écran. Il sert à programmer des graphismes vectoriels, c'est-à-dire simplement basés sur des lignes, des points et des courbes.



Voyons comment fonctionne la tortue. Démarrer d'abord le shell de Python : clique sur l'icône de ton bureau (ou, si tu utilises Ubuntu, clique sur le lanceur de la marge ; sous d'autres versions de Linux, clique sur le menu **Applications>Programmation>IDLE**). Ensuite, pour indiquer à Python d'utiliser la tortue, tu dois importer le module `turtle`, comme suit :

```
>>> import turtle
```

L'importation d'un module indique à Python que tu veux l'utiliser.

NOTE

Si tu utilises Linux (Ubuntu) et si tu reçois une erreur à ce stade, il te faut probablement installer `tkinter`. Pour ce faire, ouvre la Logithèque Ubuntu et entre `python-tk` dans la zone de recherche. L'élément `Tkinter - Écrire des applications Tk avec Python` devrait apparaître dans la fenêtre. Clique sur `Installer`. Tu devras entrer le mot de passe d'administration donc, si tu l'ignores, demande à tes parents de l'entrer.

Créer un canevas

Maintenant que tu as importé le module `turtle`, il est nécessaire de créer un **canevas** (ou *canvas* en anglais), c'est-à-dire un espace vide pour y dessiner, exactement comme la toile blanche de l'artiste. Pour cela, nous appelons la fonction `Pen` (stylo) du module `turtle`, qui crée automatiquement un nouveau canevas. Entre ce qui suit dans le shell de Python :

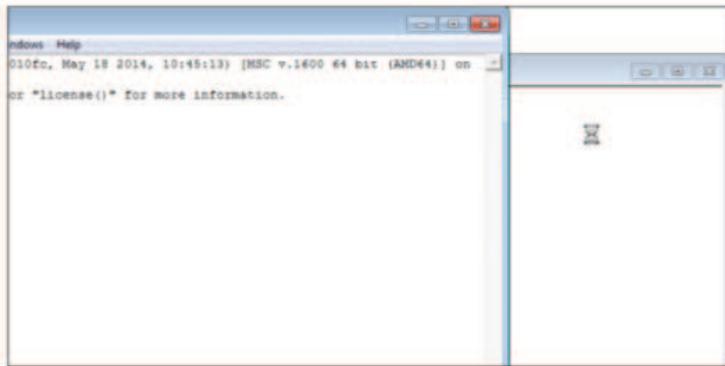
```
>>> t = turtle.Pen()
```

Apparaît alors une zone blanche, le canevas, avec une flèche au centre, dans le style de ceci :



Et la flèche au centre de cette fenêtre est notre tortue, pas vraiment ressemblante, il faut le reconnaître.

Si la fenêtre de la tortue s'affiche derrière celle du shell de Python, tu vas penser que cela ne fonctionne pas correctement. Et si tu déplaces le curseur de ta souris sur cette fenêtre, tu peux le voir se changer en un sablier, comme ceci :



Ceci se produit pour diverses raisons : tu n'as pas démarré le shell à partir de l'icône de ton bureau (si tu utilises Windows ou un Mac), tu as cliqué sur **IDLE (Python GUI)** dans le menu de démarrage de Windows ou IDLE n'est pas installé correctement. Essaie de quitter et de redémarrer le shell à partir de l'icône d'IDLE du bureau

que tu as créée au chapitre 1. Si cela ne fonctionne pas, utilise alors la console Python au lieu du shell, comme ceci.

- Sous Windows, clique sur le menu **Démarrer>Tous les programmes**, puis sur le groupe **Python 3.4**, enfin sur **Python (command line)**.
- Sous Mac OS X, clique sur l'icône **Spotlight**, la petite loupe dans le coin supérieur droit de l'écran. Dans la boîte de dialogue qui s'affiche, entre **Terminal**. Puis tape **python** lorsque le terminal est ouvert.
- Sous Ubuntu, clique sur **Terminal** s'il est présent parmi les lanceurs, ou clique sur le bouton du tableau de bord, entre **Terminal**, puis clique sur l'icône du **Terminal**. Sous d'autres versions de Linux, ouvre le menu **Applications**, ouvre le **Terminal** (dans les **Accessoires** ou les **Utilitaires**, selon ton Linux) et entre **python**.

Déplacer la tortue

Pour envoyer des instructions à la tortue, tu fais appel à des fonctions disponibles dans la variable **t** que tu viens juste de créer, comparables à la fonction **Pen** du module **turtle**. Ainsi, l'instruction **forward** (en avant) indique à la tortue d'avancer vers l'avant, en fait dans le sens de la flèche. Pour dire à la tortue d'avancer de 50 pixels, tape la commande suivante :

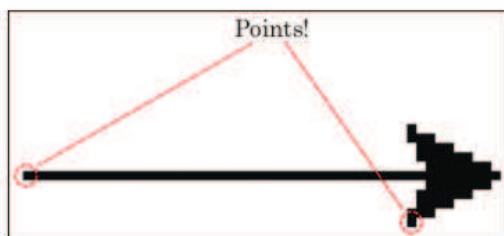


```
>>> t.forward(50)
```

Tu devrais alors voir le résultat suivant :



La tortue s'est déplacée de 50 pixels vers l'avant. Un **pixel** est un point de l'écran. C'est le plus petit élément que tu puisses représenter sur un écran d'ordinateur. Tout ce que tu vois sur ton écran est fait de pixels, de minuscules points carrés. Si tu pouvais zoomer sur le canevas, c'est-à-dire l'agrandir, tu verrais que la flèche représentant le chemin de la tortue est simplement réalisée avec tout un tas de points. Les graphismes d'ordinateur ne sont faits que de cela.



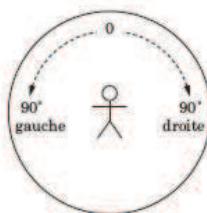
À présent, nous allons indiquer à la tortue de tourner de 90 degrés à gauche à l'aide de la commande suivante :

```
>>> t.left(90)
```

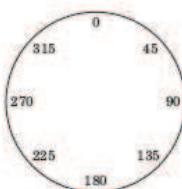
Si tu n'as pas encore appris les degrés jusqu'ici, voici comment te les représenter. Imagine que tu es debout au centre d'un cercle et que tu regardes devant toi.

- La direction dans laquelle tu regardes représente 0 degré.
- Si tu tends ton bras gauche sur ta gauche, il indique la direction 90 degrés à gauche.
- Si tu tends ton bras droit vers la droite, il indique la direction 90 degrés à droite.

Tu peux voir les trois directions dans la figure suivante :

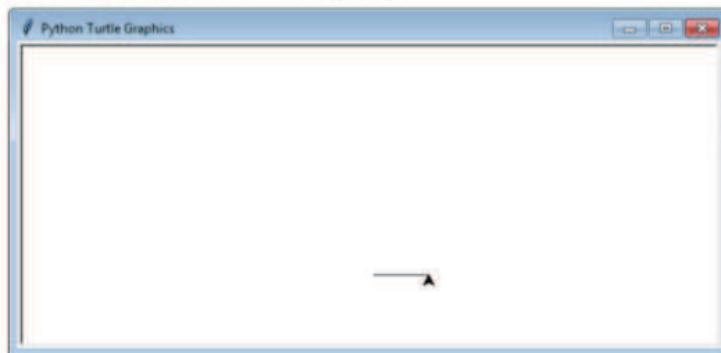


Si tu restes sur place et poursuis autour du cercle vers la droite à partir de ton bras droit, les 180 degrés sont directement derrière toi, tandis que ton bras gauche pointe vers 270 degrés. Enfin, 360 degrés correspondent à un tour complet, pour revenir à 0. Les degrés se mesurent de 0 à 360. Si tu les mesures vers la droite à partir de 0 degré en face, tu peux les reporter sur le cercle par paliers (ou « incrément ») de 45 degrés, comme suit :



Quand la tortue de Python tourne vers la gauche (`left`), elle pivote sur elle-même pour faire face à la nouvelle direction, comme tu le ferais avec ton corps pour faire face à la direction que pointe ton bras gauche, qui correspond à 90 degrés vers la gauche.

Comme la tortue pointait vers la droite au départ, la commande `t.left(90)` la fait pointer vers le haut :



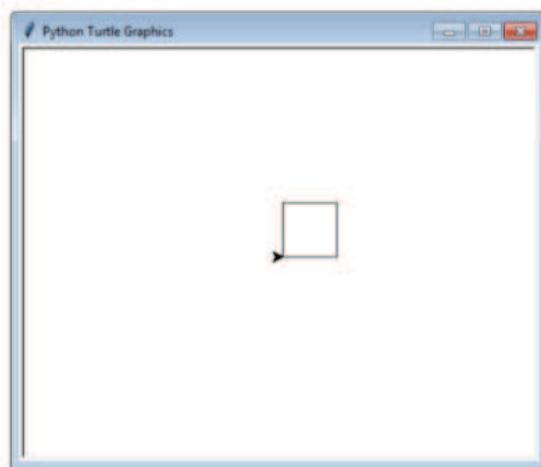
NOTE

Appeler `t.left(90)` revient au même qu'appeler `t.right(270)`. Ceci est aussi valable pour l'appel de `t.right(90)`, qui équivaut à `t.left(270)`. Représente-toi bien le cercle et familiarise-toi avec les degrés.

À présent, dessinons un carré. Ajoute les lignes de code suivantes à celles déjà entrées :

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

La tortue dessine un carré et se retrouve dans la position initiale :



Pour réinitialiser le canevas, utilise la fonction `reset`. Elle efface le canevas et place la tortue à son emplacement et dans la direction de départ.

```
>>> t.reset()
```

La fonction `clear` efface également le canevas, mais laisse la tortue dans son état actuel.

```
>>> t.clear()
```

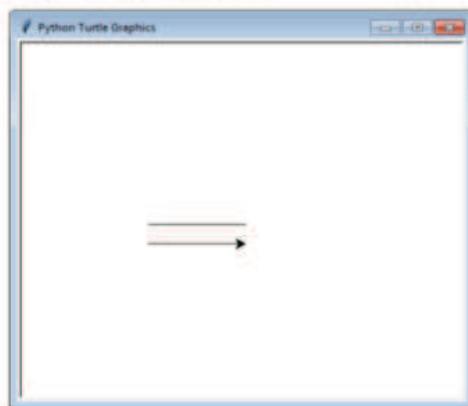
Il est aussi possible de faire pivoter la tortue vers la droite, avec `right`, et même de la faire reculer sur ses pas, avec `backward`. La fonction `up` relève le stylo de la page (autrement dit, indique à la tortue de cesser de dessiner), tandis que `down` lui indique de recommencer à dessiner. Toutes ces fonctions s'écrivent de la même manière que celles que nous avons utilisées.

Essayons de dessiner autre chose à l'aide de quelques-unes de ces commandes. Cette fois, nous lui demandons de dessiner deux traits. Saisis le code suivant :

```
>>> t.reset()
>>> t.backward(100)
>>> t.up()
>>> t.right(90)
>>> t.forward(20)
>>> t.left(90)
>>> t.down()
>>> t.forward(100)
```

Dans ces lignes, nous réinitialisons le canevas et la tortue à son emplacement de départ, avec `t.reset()`. Puis, nous déplaçons la tortue de 100 pixels en arrière, la fonction `t.up()` relève le stylo et arrête le dessin.

La commande `t.right(90)` fait tourner la tortue à 90 degrés vers la droite, pour pointer vers le bas de l'écran. Avec `t.forward(20)`, nous déplaçons la tortue de 20 pixels vers le bas, toujours sans dessiner, puisque la commande `up` a relevé le stylo à la troisième ligne. Puis, nous faisons pivoter la tortue de 90 degrés vers la gauche avec `t.left(90)` pour nous tourner vers la droite de l'écran. Ensuite, nous déposons le stylo sur la page avec la fonction `down`, pour recommencer à dessiner. Enfin, nous dessinons un trait en avant, parallèle au premier trait, avec `t.forward(100)`. Les deux traits que nous avons dessinés ont l'allure suivante :



Ce que tu as appris

Dans ce chapitre, tu as appris à utiliser le module `turtle` de Python. Nous avons tracé quelques traits simples à l'aide des pivotements `left` (à gauche) et `right` (à droite), ainsi que des fonctions `forward` (en avant) et `backward` (en arrière). Tu as vu comment faire cesser la tortue de dessiner avec `up` et recommencer à dessiner avec `down`. Tu as également découvert que la tortue pivote en degrés.

Puzzles de programmation

Essaie de dessiner par toi-même ces quelques formes à l'aide de la tortue. Si cela ne va pas ou pour comparer tes solutions avec celles proposées, les réponses sont disponibles sur le site web d'accompagnement du livre.

1. Un rectangle

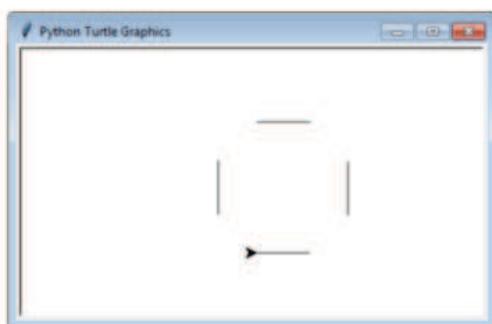
Crée un nouveau canevas à l'aide de la fonction `Pen` du module `turtle`, puis dessine un rectangle.

2. Un triangle

Crée un nouveau canevas mais, cette fois, dessine un triangle. Retourne au diagramme en cercle avec les degrés pour te rappeler dans quels sens faire pivoter la tortue.

3. Un carré sans coins

Écris un programme pour tracer les quatre lignes montrées ci-après (les longueurs des lignes n'ont pas d'importance, juste la forme) :





POSER DES QUESTIONS AVEC IF ET ELSE

En programmation, il arrive souvent de devoir poser des questions auxquelles on répond par oui ou non, pour décider ensuite de faire quelque chose de particulier en fonction de la réponse. Par exemple, nous pourrions demander « as-tu plus de 20 ans ? » et, si la réponse est oui, répondre par « tu es trop vieux ! ».

Ce genre de question s'appelle une condition. Nous combinons ces conditions et leurs réponses dans des instructions `if` (si, en français). Les conditions peuvent être bien plus compliquées que cette simple question et les instructions `if` sont combinables pour traiter des choix dépendant de plusieurs questions.

Ce chapitre montre comment utiliser des instructions `if` pour construire des programmes.

Instructions if

En Python, une instruction `if` peut s'écrire comme suit :

```
>>> age = 13  
>>> if age > 20:  
    print('Tu es trop vieux !')
```

Comme tu peux le voir, l'instruction est constituée du mot-clé `if`, puis d'une condition suivie d'un deux-points (`:`), comme dans `age > 20:`. Les lignes qui suivent le deux-points doivent former un bloc et, si la réponse à la question est oui (c'est-à-dire que la condition est vraie, ou `True` comme l'on dit en programmation Python), alors les commandes dans ce bloc sont exécutées. Maintenant, voyons comment écrire des blocs et des conditions.



Un bloc est un groupe d'instructions

Un bloc de code est un ensemble regroupé d'instructions de programmation. Lorsque la condition `if age > 20:` est vraie, tu pourrais en faire un peu plus que simplement afficher « Tu es trop vieux ! » et, par exemple, afficher d'autres phrases, comme ceci :

```
>>> age = 25  
>>> if age > 20:  
    print('Tu es trop vieux !')  
    print('Que fais-tu là ?')  
    print('Pourquoi ne vas-tu pas tondre la pelouse ?')
```

Ce bloc de code est formé de trois instructions `print` qui ne sont exécutées que si la condition `age > 20` est vraie. Chacune des lignes du bloc est décalée de quatre espaces vers la droite par rapport à l'instruction `if` juste au-dessus. Si nous pouvions afficher les espaces, tu verrais ceci :

```
>>> age = 25  
>>> if age > 20:  
    print('Tu es trop vieux !')  
    print('Que fais-tu là ?')  
    print('Pourquoi ne vas-tu pas tondre la pelouse ?')
```

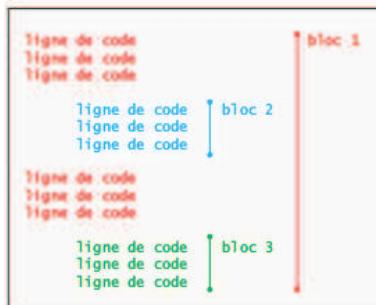
En Python, les caractères d'espacement, comme les tabulations (avec la touche `Tab` du clavier) et les espaces (avec la barre d'espace),

ont un sens bien déterminé. Les lignes dont le code débute au même emplacement (en retrait du même nombre d'espaces depuis la marge gauche) sont regroupées en un bloc ; chaque fois que tu débutes une nouvelle ligne par plus d'espaces que la précédente, tu démarres un nouveau sous-bloc qui fait partie du précédent. Voici un schéma pour mieux comprendre :



Nous regroupons ensemble les instructions dans des blocs, parce qu'ils s'associent et qu'il faut les exécuter ensemble.

Si tu changes le retrait, tu crées généralement de nouveaux blocs. Dans l'exemple suivant, trois blocs séparés sont créés, simplement en changeant le retrait.



Ici, même si les blocs 2 et 3 ont le même retrait, ils sont considérés comme différents, parce qu'un bloc avec un retrait moindre (moins d'espaces) existe entre les deux.

Dans le même genre, si tu crées un bloc dont une ligne a quatre espaces de retrait et la ligne suivante six espaces, tu vas provoquer

une erreur lors de l'exécution, parce que Python s'attend à ce que tu utilises le même nombre d'espaces dans toutes les lignes d'un même bloc. Voici un exemple de ce qu'il ne faut surtout pas faire :

```
>>> if age > 20:  
    print('Tu es trop vieux !')  
        print('Que fais-tu là ?')
```

Dans cet exemple, les espaces sont visibles pour que tu voies la différence. Dans la troisième ligne, il y a six espaces, au lieu de quatre à la ligne juste avant.

Lorsque tu essaies d'exécuter ce code, IDLE met en évidence la ligne quand il détecte un problème : il surligne en rouge les espaces fautifs et affiche un message du type `SyntaxError` pour expliquer l'erreur (*indent* signifie retrait) :

```
>>> age = 25  
>>> if age > 20:  
    print('Tu es trop vieux !')  
        print('Que fais-tu là ?')  
SyntaxError: unexpected indent
```

Python ne s'attendait pas à voir deux espaces supplémentaires au début de la deuxième ligne `print` et il ne sait pas si cette ligne fait partie d'un nouveau bloc ou du même bloc, donc il lève les bras et affiche une erreur.

NOTE

L'utilisation cohérente des retraits, c'est-à-dire chaque fois de la même manière, rend la lecture du code plus aisée. Si tu commences un programme en écrivant quatre espaces au début d'un bloc, continue d'utiliser quatre espaces au début des autres blocs du programme, puis huit pour un bloc de niveau inférieur et ainsi de suite. Vérifie aussi que chaque ligne d'un même bloc débute avec le même nombre d'espaces.

Des conditions pour comparer des choses

Une condition est une instruction de programme qui compare des choses et nous indique si le critère de la comparaison est vrai (`True`) ou faux (`False`). Ainsi, `age > 10` est une condition et revient à demander « la valeur de la variable `age` est-elle plus grande que `10` ? ». Voici une autre condition : `couleur_de_cheveux == 'violet'` ; elle revient à dire « la valeur de la variable `couleur_de_cheveux` est-elle égale à `violet` ? ».

Python utilise des symboles, appelés des **opérateurs**, pour créer des conditions, comme égal à, plus grand que ou plus petit que. Le tableau 5-1 énumère quelques opérateurs de comparaison, qui servent dans les conditions.

Tableau 5-1. Symboles pour conditions

Symbol	Définition
<code>==</code>	égal à
<code>!=</code>	non égal à (ou différent de)
<code>></code>	plus grand que
<code><</code>	plus petit que
<code>>=</code>	plus grand ou égal à
<code><=</code>	plus petit ou égal à

Par exemple, si tu as 10 ans, la condition `ton_age == 10` renvoie `True`, sinon, elle renverrait `False`. Et si tu as 12 ans, la condition `ton_age > 10` renvoie `True`.

ATTENTION Assure-toi de bien écrire le double signe égal (`==`) quand tu veux créer une condition « égal à ».

Voyons encore quelques exemples. Dans celui qui suit, nous définissons l'âge comme égal à `10`, puis nous écrivons une instruction conditionnelle qui affiche « Tu es trop vieux pour apprécier mes blagues ! » si l'âge est plus grand que `10` :

```
>>> age = 10
>>> if age > 10:
    print('Tu es trop vieux pour apprécier mes blagues !')
```

Que se passe-t-il quand tu entres ces lignes dans IDLE et que tu appuies sur `Entrée` ?

Rien. Pourquoi ?

Parce que la valeur renvoyée par `age` n'est pas plus grande que `10`, donc Python n'exécute pas le contenu du bloc avec `print`. En revanche, si nous avions défini la variable `age` à `20`, alors ce message aurait été affiché.

Reprendons à présent cet exemple mais avec une condition plus grande ou égale à (`>=`) :

```
>>> age = 10
>>> if age >= 10:
    print('Tu es trop vieux pour apprécier mes blagues !')
```



Cette fois, tu vois le message s'afficher à l'écran, parce que la valeur `d'age` est égale à `10`, donc *elle est* plus grande ou *égale à* `10`.
Et voyons ce que cela donne avec une condition égal à (`==`) :

```
>>> age = 10
>>> if age == 10:
    print("Qu'est-ce qu'une chauve-souris avec une perruque ?")
    print("Une souris !")
```

Comme la valeur de la variable `age` est bien égale à `10`, la condition est vraie et le message peut s'afficher.

Instructions si-alors-sinon

L'instruction `if`, que nous pourrions traduire par « si, alors », fait quelque chose lorsqu'une condition est vraie (`True`). Dans certains cas, on veut prévoir en plus de faire autre chose quand la condition n'est pas vraie. Nous pourrions par exemple afficher un message à l'écran si l'âge vaut `12` et un autre message si l'âge ne vaut pas `12` (`False`).

Lastuce consiste ici à utiliser une instruction « si-alors-sinon » ou plutôt `if-alors-else`, qui signifie « si (`if`) quelque chose est vrai, alors (`:`) faire ceci, sinon (`else`), faire cela ».

Pour essayer l'instruction si-alors-sinon, entre ce qui suit :

```
>>> print("Tu veux entendre une histoire cochonne ?")
Tu veux entendre une histoire cochonne ?
>>> age = 12
>>> if age == 12:
    print("Un cochon est tombé dans la boue.")
else:
    print("Chut. C'est un secret.")
Un cochon est tombé dans la boue.
```

Comme nous avons donné à la variable `age` la valeur `12` et que la condition demande si `age` est égal à `12`, tu dois voir le premier message à l'écran. Maintenant, essaie de changer la valeur `d'age` en un autre nombre, comme ceci :



```
>>> print("Tu veux entendre une histoire cochonne ?")
Tu veux entendre une histoire cochonne ?
>>> age = 8
>>> if age == 12:
    print("Un cochon est tombé dans la boue.")
```

```
else:  
    print("Chut. C'est un secret.")  
Chut. C'est un secret.
```

Cette fois, comme `age` vaut `8`, c'est le second message qui s'affiche.

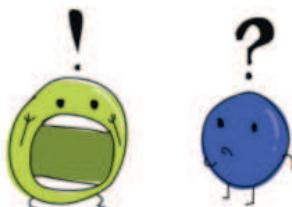
Instructions if et elif

Nous pouvons même aller plus loin avec les `elif` (l'abréviation de `else-if`, « sinon-si »), par exemple si nous voulons vérifier que l'âge d'une personne vaut `10`, `11`, `12` et ainsi de suite, pour que le programme fasse des choses différentes selon l'âge donné en réponse. Ces instructions sont très différentes des instructions `if-alors-else` car il peut y avoir plusieurs `elif` dans la même instruction :

```
>>> age = 12  
❶ >>> if age == 10:  
❷     print("Comment appelle-t-on un lapin sourd ?")  
     print("On ne l'appelle pas, on va le chercher !")  
❸ elif age == 11:  
     print("Que dit un raisin blanc à un raisin noir ?")  
     print("Du beau temps pendant les vacances ?")  
❹ elif age == 12:  
❺     print("Que dit un 0 à un 8 ?")  
     print("Salut, vous deux !")  
elif age == 13:  
    print("Monsieur et madame Enfaillite ont une fille ?")  
    print("Mélusine, bien sûr !")  
else:  
    print("Pardon ?")  
Que dit un 0 à un 8 ? Salut, vous deux !
```

Dans cet exemple, l'instruction `if` de la deuxième ligne vérifie si la variable `age` est égale à `10` en ❶. L'instruction `print` qui suit en ❷ n'est exécutée que si `age` vaut `10`. Or, comme nous avons donné la valeur `12` à `age`, l'ordinateur saute à l'instruction `if` suivante, en ❸, et vérifie si la valeur d'`age` est égale à `11`. Elle ne l'est pas, donc l'ordinateur saute à l'instruction `if` suivante, en ❹, pour voir si `age` est égal à `12`. Il l'est, donc cette fois, l'ordinateur exécute la commande `print` en ❺.

Lorsque tu entres ces lignes de programme dans IDLE, celui-ci leur applique automatiquement le retrait adéquat. Assure-toi donc d'appuyer sur



les touches **Ret.**, **arr** et **Suppr** lorsque tu as entré chaque instruction `print`, pour que les instructions `if`, `elif` et `else` commencent tout au début de la ligne, à la marge gauche. C'est le même emplacement qu'occuperaient ces instructions `if` si les invites (`>>>`) étaient absentes.

Combiner des conditions

Pour combiner des conditions, utilise les mots-clés `and` (et) et `or` (ou), qui raccourcissent et simplifient le code. Voici un exemple d'utilisation :

```
>>> if age == 10 or age == 11 or age == 12 or age == 13:  
    print('Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !')  
else:  
    print('Pardon ?')
```

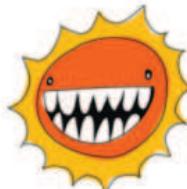
Dans ce code, si au moins une des conditions de la première ligne est vraie (autrement dit, si `age` vaut **10**, **11**, **12 ou 13**), le bloc de code de la ligne suivante, qui commence par `print`, s'exécute.

Si aucune des conditions de la première ligne n'est vraie (« sinon », c'est-à-dire `else`), alors Python va dans le bloc de la dernière ligne et affiche **Pardon ?** à l'écran.

Pour compacter encore un peu le code de cet exemple, nous pouvons utiliser le mot-clé `and` combiné avec les opérateurs supérieur ou égal à (`>=`) et inférieur ou égal à (`<=`), comme ceci :

```
>>> if age >= 10 and age <= 13:  
    print('Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !')  
else:  
    print('Pardon ?')
```

Ici, si `age` est supérieur ou égal à **10 et (and)**, à la fois, inférieur ou égal à **13**, comme l'indique la première ligne, alors le bloc de code du `print` de la ligne suivante s'exécute. Donc, si la valeur d'`age` est **12**, alors **Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !** apparaît à l'écran, car **12** est plus grand que **10** et plus petit que **13**.



Variabes sans valeur : None

De même que nous pouvons donner à une variable une valeur de nombre, de chaîne ou de liste, nous pouvons lui donner une valeur

vide, c'est-à-dire rien. En Python, cette valeur vide s'appelle `None` et elle correspond à l'absence de valeur. Il est très important de bien comprendre que la valeur `None` n'a rien à voir avec la valeur `0`, parce qu'elle ne correspond à aucune valeur, tandis que `0` est un nombre dont la valeur est... le nombre zéro ! La seule valeur qu'une variable ait quand on lui donne (on dit « affecte » ou « attribue ») la valeur vide `None`, c'est rien du tout. Voici un exemple :

```
>>> mavaleur = None  
>>> print(mavaleur)  
None
```

L'affectation de la valeur `None` à une variable est une manière de réinitialiser cette dernière à son état initial, vide. C'est aussi une façon de définir une variable sans lui donner de valeur. Tu peux faire ce genre de chose quand tu sais que tu vas avoir besoin d'une variable plus tard dans un programme, mais que tu préfères définir (ou « déclarer ») toutes tes variables au début du code. Les programmateurs procèdent souvent ainsi, parce que les indiquer là aide à mieux les reconnaître lorsqu'elles apparaissent dans des extraits de code.

Tu peux aussi vérifier la présence de `None` dans une instruction `if`, comme dans cet exemple :

```
>>> mavaleur = None  
>>> if mavaleur == None:  
    print("La variable mavaleur n'a aucune valeur.")  
La variable mavaleur n'a aucune valeur.
```

Ceci peut s'avérer bien utile pour calculer une valeur à partir d'une variable qui n'a pas encore été calculée.

Différence entre chaînes et nombres

Quand une personne tape quelque chose au clavier, ce qu'il produit est appelé « entrée de l'utilisateur » et cela peut correspondre à un caractère, une touche de curseur (les flèches `Haut`, `Bas`, `Gauche` ou `Droite`), la touche `Entrée` ou n'importe quoi d'autre. En Python, une entrée d'utilisateur produit une chaîne ; lorsque tu entres `10` au clavier, par exemple, Python enregistre cela dans une variable sous forme d'une chaîne, pas d'un nombre.

Alors, quelle est la différence entre le nombre `10` et la chaîne `'10'` ? Ils nous apparaissent identiques, avec la seule différence des apos-

trophes verticales qui entourent la chaîne. Pour l'ordinateur, il en va tout autrement : il y a une grande différence entre les deux.

Prenons l'exemple où nous définissons la variable `age` à la valeur `10` et comparons cette valeur à un nombre dans une instruction `if`, comme ceci :

```
>>> age = 10
>>> if age == 10:
    print("Comment parle-t-on à un monstre ?")
    print("D'aussi loin que possible !")
Comment parle-t-on à un monstre ?
D'aussi loin que possible !
```

Comme tu peux le voir, les instructions `print` s'exécutent.

Maintenant, définissons la variable `age` à la chaîne '`10`' (avec les apostrophes), comme suit :

```
>>> age = '10'
>>> if age == 10:
    print("Comment parle-t-on à un monstre ?")
    print("D'aussi loin que possible !")
```

Ici, le code des `print` ne s'exécute plus, parce que Python ne voit aucun nombre entre les apostrophes : il ne l'interprète pas comme un nombre.

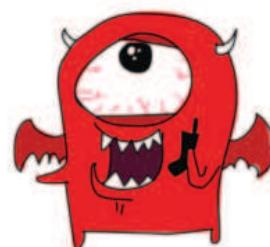
Pour passer outre ce problème, Python possède heureusement quelques fonctions magiques qui servent à transformer, à « convertir » des chaînes en nombres et aussi des nombres en chaînes. Voici par exemple comment convertir la chaîne '`10`' en un nombre, avec `int` :

```
>>> age = '10'
>>> age_converti = int(age)
```

La variable `age_converti` reçoit un nombre, `10`. Et tu peux ensuite utiliser cette variable, qui contient un nombre, dans ton instruction `if`.

À l'inverse, pour convertir un nombre en chaîne, utilise `str` :

```
>>> age = 10
>>> age_converti = str(age)
```



Ici, `age_converti` contient la chaîne '10' au lieu du nombre 10.

Nous avons vu que l'instruction `if age == 10` n'a produit aucun affichage quand la variable était définie en une chaîne (`age = '10'`). En revanche, si tu convertis d'abord la variable, le résultat est tout différent :

```
>>> age = '10'  
>>> age_converti = int(age)  
>>> if age_converti == 10:  
    print("Comment parle-t-on à un monstre ?")  
    print("D'aussi loin que possible !")  
Comment parle-t-on à un monstre ?  
D'aussi loin que possible !
```

Ce n'est pas fini. Si tu essaies de convertir une chaîne qui contient un nombre avec un point décimal, tu obtiens une erreur, parce que la fonction `int` s'attend à rencontrer un nombre entier dans la chaîne de départ :

```
>>> age = '10.5'  
>>> age_converti = int(age)  
Traceback (most recent call last):  
  File "<pyshell#35>", line 1, in <module>  
    age_converti = int(age)  
ValueError: invalid literal for int() with base 10: '10.5'
```

Une `ValueError` (erreur de valeur) est le moyen qu'utilise Python pour te dire que la valeur que tu as essayé de convertir n'est pas correcte. Pour régler ce problème, utilise plutôt la fonction `float` dans ce cas-ci. Cette fonction est capable de gérer des nombres qui ne sont pas entiers.

```
>>> age = '10.5'  
>>> age_converti = float(age)  
>>> print(age_converti)  
10.5
```

Tu recevras aussi une erreur de type `ValueError` si tu essaies de convertir une chaîne qui ne contient pas de chiffre :

```
>>> age = 'dix'  
>>> age_converti = int(age)  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    age_converti = int(age)  
ValueError: invalid literal for int() with base 10: 'dix'
```

Ce que tu as appris

Dans ce chapitre, tu as appris à utiliser les instructions `if` pour créer des blocs de code qui ne s'exécutent que lorsque des conditions particulières sont vraies. Tu as vu comment étendre les instructions `if` à l'aide de `elif`, pour exécuter des portions différentes de code en réponse à des conditions différentes, et comment utiliser le mot-clé `else` pour exécuter du code lorsqu'aucune des conditions précédentes n'est vraie. Tu as appris aussi la combinaison de conditions avec les mots-clés `and` et `or`, pour vérifier qu'un nombre apparaît dans une plage de valeurs. Tu as également vu comment convertir des chaînes en nombres avec `int` et `float`, mais aussi des nombres en chaînes avec `str`. Enfin, tu as découvert que rien du tout (`None`) a une signification en Python et peut servir à réinitialiser des variables à leur état initial, vide.

Puzzles de programmation

Essaie de résoudre les puzzles suivants à l'aide d'instructions `if` et de conditions. Les réponses sont disponibles sur le site d'accompagnement du livre.

1. Es-tu riche ?

À ton avis, que fait le code suivant ? Essaie d'imaginer la réponse sans la taper dans le shell, puis vérifie ta réponse.

```
>>> mes_sous = 2000
>>> if mes_sous > 1000:
...     print("Je suis riche !!")
... else:
...     print("Je ne suis pas riche !!")
...     print("Mais ça viendra...")
```

2. Barres chocolatées

Crée une instruction `if` pour vérifier que le nombre de barres chocolatées dans la variable `barres_choco` est plus petit que `100` ou plus grand que `500`. Le programme doit afficher « Trop peu ou beaucoup trop » quand la condition est vraie.

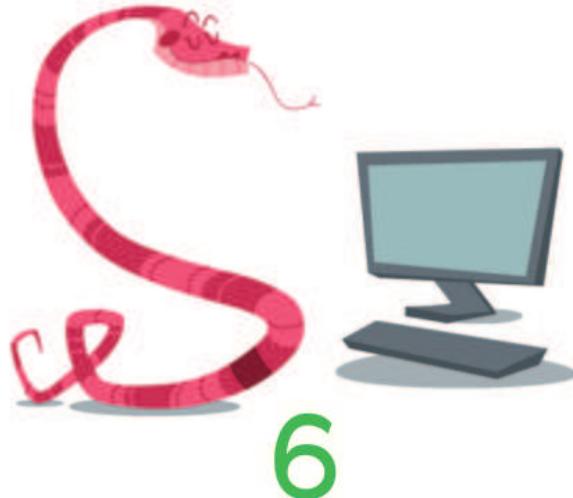
3. Juste le bon nombre

Dans une instruction `if`, vérifie que la somme d'argent contenue dans une variable `somme` est comprise entre `100` et `500` ou entre `1 000` et `5 000`.

4. Affronter des ninjas

Crée une instruction `if` qui affiche « Il y en a trop » si la variable `ninjas` contient un nombre inférieur à `50`, affiche « Je vais devoir me battre mais je peux les avoir » s'il est inférieur à `30` et affiche « Je peux affronter ces ninjas ! » s'il est inférieur à `10`. Essaie ton code avec :

```
>>> ninjas = 5
```



TOURNER EN BOUCLE

Il n'y a sans doute rien de pire que de devoir répéter les mêmes choses, encore et encore. Si les gens comptent des moutons quand ils peinent à s'endormir, il y a une raison et il ne faut pas chercher dans ces mammifères couverts de laine des pouvoirs magiques qui feraient dormir. En fait, cela vient de ce que répéter sans fin les mêmes choses est assommant, donc l'esprit décroche et le corps s'endort plus facilement s'il ne se concentre pas sur quelque chose d'intéressant.

Les programmeurs n'aiment pas non plus se répéter, à moins qu'ils veuillent s'endormir, bien entendu. Heureusement, les langages de programmation possèdent généralement ce que l'on appelle des boucles `for`, qui répètent automatiquement des choses, telles que des instructions de programme et des blocs de code.

Dans ce chapitre, nous allons examiner les boucles `for`, ainsi qu'un autre type de boucle que Python propose : la boucle `while`.



Utiliser les boucles `for`

Pour afficher cinq fois « bonjour » en Python, il y a bien cette solution :

```
>>> print("bonjour")
bonjour
```

Il faut avouer que c'est assez lourd et fastidieux. Au lieu de cela, il y a la solution de la boucle `for`, qui réduit de beaucoup les frappes au clavier et les répétitions, comme ceci :

```
❶ >>> for x in range(0, 5):
❷         print('bonjour')
bonjour
bonjour
bonjour
bonjour
bonjour
```

La fonction `range` en ❶ permet de créer une liste de nombres compris entre un nombre de départ (inclus) et un nombre de fin (exclu). Cela peut sembler un peu compliqué, donc combinons la fonction `range` avec la fonction `list` pour voir exactement comment cela fonctionne. En réalité, la fonction `range` ne crée pas vraiment une liste de nombres : elle renvoie un itérateur, c'est-à-dire un type d'objet de Python spécialement conçu pour travailler avec des boucles. En revanche, si nous combinons `range` avec `list`, nous obtenons une vraie liste de nombres :

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Dans le cas de la boucle `for`, le code en ❶ indique à Python de :

- démarrer un comptage à `0` et de le terminer avant d'atteindre `5` ;
- pour chaque nombre compté, en stocker la valeur dans la variable `x`.

Ensuite, Python exécute le bloc de code en ❷. Remarque les quatre espaces au début de la ligne ❷, par rapport à la ligne ❶. Python a placé automatiquement ce retrait pour toi.

Lorsque tu presses Entrée à la fin de la deuxième ligne, Python affiche cinq fois « bonjour ».

Nous pouvons aussi utiliser `x` dans l'instruction `print` pour compter les « bonjour » :

```
>>> for x in range(0, 5):
...     print('bonjour %s' % x)
bonjour 0
bonjour 1
bonjour 2
bonjour 3
bonjour 4
```

Pour comparer, si nous éliminions la boucle `for`, le code éclaté ressemblerait à ceci :

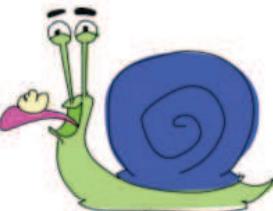
```
>>> x = 0
>>> print('bonjour %s' % x)
bonjour 0
>>> x = 1
>>> print('bonjour %s' % x)
bonjour 1
>>> x = 2
>>> print('bonjour %s' % x)
bonjour 2
>>> x = 3
>>> print('bonjour %s' % x)
bonjour 3
>>> x = 4
>>> print('bonjour %s' % x)
bonjour 4
```

Donc la présence de la boucle nous épargne de devoir entrer huit lignes de code supplémentaires. Les bons programmeurs détestent écrire plusieurs fois les mêmes choses ; par conséquent, la boucle `for` figure parmi leurs instructions favorites.

Tu n'es pas obligé d'utiliser à tout prix les fonctions `range` et `list` quand tu écris des boucles `for`, car tu pourrais aussi utiliser une liste déjà créée, par exemple la liste de courses du sorcier que nous avons créée au chapitre 3 :

```
>>> liste_sorcier = ["patte d'araignée", 'orteil de crapaud', "langue ⚡\n    d'escargot", 'aile de chauve-souris', 'beurre de limace', "rôt ⚡\n    d'ours"]\n>>> for i in liste_sorcier:\n        print(i)\n        patte d'araignée\n        orteil de crapaud\n        langue d'escargot\n        aile de chauve-souris\n        beurre de limace\n        rôt d'ours
```

Ce code revient à dire « pour chaque élément (`for`) dans (`in`) `liste_sorcier`, stocke la valeur dans la variable `i`, puis affiche (`print`) le contenu de cette variable ». Même chose ici, si nous ne passions pas par la boucle `for`, nous devrions écrire tout ce qui suit :



```
>>> liste_sorcier = ["patte d'araignée", 'orteil de crapaud', "langue ⚡\n    d'escargot", 'aile de chauve-souris', 'beurre de limace', "rôt ⚡\n    d'ours"]\n>>> print(liste_sorcier[0])\n        patte d'araignée\n>>> print(liste_sorcier[1])\n        orteil de crapaud\n>>> print(liste_sorcier[2])\n        langue d'escargot\n>>> print(liste_sorcier[3])\n        aile de chauve-souris\n>>> print(liste_sorcier[4])\n        beurre de limace\n>>> print(liste_sorcier[5])\n        rôt d'ours
```

Il faut avouer qu'ici aussi, la boucle permet de s'épargner beaucoup de code.

Essayons une autre boucle. Saisis le code suivant dans le shell. Il doit placer les retraits automatiquement pour toi :

```
❶ >>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
❷ >>> for i in longpantalonpoilu:
❸     print(i)
❹     print(i)
❺
❻ Long
❼ Long
❼ pantalon
❼ pantalon
❼ poilu
❼ poilu
```

À la première ligne ❶, nous créons une liste qui contient 'long', 'pantalon' et 'poilu'. À la deuxième ligne ❷, nous bouclons parmi les éléments de la liste et chaque élément est affecté à la variable *i*. Aux deux lignes suivantes ❸ et ❹, nous affichons deux fois le contenu de la variable. L'appui sur la touche Entrée à la ligne vide suivante ❺ indique à Python de fermer le bloc ; en conséquence, il exécute la totalité du code, pour afficher deux fois chacun des éléments.

Rappelle-toi que si tu tapes un nombre incorrect d'espaces, tu obtiens un message d'erreur. Si, par exemple, tu entres un espace supplémentaire au début de la ligne ❷, Python affiche une erreur de retrait (*indent*) :

```
>>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
>>> for i in longpantalonpoilu:
    print(i)
    print(i)
```

SyntaxError: unexpected indent

Nous avons en effet expliqué au chapitre 5 que Python s'attend à ce que les nombres d'espaces de retrait des blocs soient cohérents. Peu importe le nombre d'espaces que tu choisis, tant que tu conserves le même nombre pour chaque nouvelle ligne. En plus, cela facilite la lecture du code aux êtres humains.

Voici pour suivre un exemple un peu plus complexe de boucle `for`, avec deux niveaux de blocs :



```
>>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
>>> for i in longpantalonpoilu:
    print(i)
        for j in longpantalonpoilu:
            print(j)
```

Quels sont les blocs dans ce code ? Le premier bloc est celui de la première boucle `for` :

```
>>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
>>> for i in longpantalonpoilu:
    print(i) # Ces lignes forment
    for j in longpantalonpoilu: # le PREMIER bloc.
        print(j) #
```

Le deuxième bloc ne contient que la ligne `print` de la deuxième boucle `for` :

```
❶ >>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
      >>> for i in longpantalonpoilu:
          print(i)
❷   ❷   for j in longpantalonpoilu:
❸     print(j) # Voici le DEUXIÈME bloc.
```

Imagines-tu bien ce que fait ce petit bout de code ?

Après la création d'une liste `longpantalonpoilu` en ❶, nous pouvons dire des deux lignes suivantes qu'elles vont boucler parmi les éléments de cette liste et les afficher un à un. Cependant, en ❷, une nouvelle boucle parcourt de nouveau la liste et, cette fois, affecte l'élément à la variable `j`, puis affiche à nouveau chaque élément en ❸. Les lignes de code ❷ et ❸ font toujours partie de la première boucle `for`, du fait de leur retrait, ce qui signifie qu'elles sont exécutées pour chaque élément que la première boucle `for` rencontre.

Donc, lorsque s'exécute ce code, nous devons voir `long`, suivi de `long`, `pantalon` et `poilu`, puis `pantalon`, suivi de `long`, `pantalon` et `poilu`, et ainsi de suite.

Saisis le code dans le shell de Python et vois par toi-même :

```
>>> longpantalonpoilu = ['long', 'pantalon', 'poilu']
>>> for i in longpantalonpoilu:
    print(i)
    for j in longpantalonpoilu:
        print(j)

❶   long
❶   long
❶   pantalon
❶   poilu
❷   pantalon
❷   long
❷   pantalon
❷   poilu
```

```
+ poilu  
long  
pantalon  
poilu
```

Python entre dans la première boucle et affiche un élément de la liste en ❶. Puis, il entre dans la deuxième boucle et affiche tous les éléments de la même liste, en ❷. Ensuite, il continue dans la première boucle avec la commande `print(i)`, puis réaffiche la liste complète avec `print(j)`. En sortie, les lignes marquées de + sont affichées par l'instruction `print(i)`. Les autres sont affichées par `print(j)`.

C'est bien joli tout cela, mais il serait peut-être intéressant d'utiliser les boucles à autre chose qu'à afficher des mots inutiles. Rappelle-toi le calcul que nous avons effectué au chapitre 2 pour connaître le nombre de pièces d'or que tu aurais à la fin d'une année, si tu utilisais l'invention géniale de ton grand-père pour répliquer des pièces. Il avait l'allure suivante :

```
>>> 20 + 10 * 365 - 3 * 52
```

Cela représente 20 pièces trouvées plus 10 pièces magiques multipliées par 365 jours de l'année, moins les 3 pièces volées par semaine par le corbeau.



Il serait intéressant de voir comment augmente ton tas de pièces chaque semaine. Pour cela, nous pouvons utiliser une boucle `for` mais, auparavant, nous devons changer la valeur de notre variable `pieces_magiques` pour qu'elle représente le nombre total de pièces magiques par semaine, soit 10 pièces magiques par jour fois 7 jours dans une semaine, de sorte que `pieces_magiques` vaut 70 :

```
>>> pieces_trouvees = 20  
>>> pieces_magiques = 70  
>>> pieces_volees = 3
```

Ensuite, pour voir le trésor augmenter chaque semaine, il nous faut une autre variable, `pieces`, et une boucle :

```
>>> pieces_trouvees = 20  
>>> pieces_magiques = 70  
>>> pieces_volees = 3  
❶ >>> pieces = pieces_trouvees  
❷ >>> for semaine in range(1, 53):
```

```
❸     pieces = pieces + pieces_magiques - pieces_volees
❹     print('Semaine %s = %s' % (semaine, pieces))
```

En ❶, la variable `pieces` reçoit la valeur de la variable `pieces_trouvees`, pour obtenir le nombre de départ. La ligne suivante ❷ prépare la boucle `for` qui exécutera les commandes du bloc formé par les lignes ❸ et ❹. À chaque passage dans la boucle, la variable `semaine` reçoit le nombre suivant de la plage comprise entre 1 (inclus) et 53 (exclu).

La ligne ❸ est un peu plus compliquée. À la base, nous voulons chaque semaine ajouter le nombre de pièces créées par magie et soustraire le nombre de pièces volées par le corbeau. Imagine la variable `pieces` comme une sorte de coffre au trésor, où chaque semaine viennent s'empiler les pièces. Ainsi, cette ligne signifie « remplacer le contenu de la variable `pieces` par le nombre actuel de pièces, plus celles que j'ai créées cette semaine, moins celles qui ont été volées cette semaine ». Dans le fond, le signe égal (`=`) est une portion de code qui signifie « effectuer ce qui se trouve à droite et le conserver pour un usage par la suite, en utilisant le nom placé à gauche ».

La ligne ❹ contient une instruction `print` avec des espaces réservés pour afficher à l'écran le numéro de la semaine, ainsi que le nombre total de pièces obtenu jusqu'ici. Si ceci te paraît incompréhensible, relis le passage intitulé « Insérer des valeurs dans des chaînes » du chapitre 3. Ensuite, si tu tapes les lignes de ce programme, tu obtiens ceci :

```
Python 3.4.1 |v3.4.1:c0de311e010fc|, May 18 2014, 10:45:13 |MSC v.1600 64 bit
(AMD64)
Type "copyright", "credits" or "license()" for more information.

>>> pieces_trouvees = 20
>>> pieces_magiques = 30
>>> pieces_volees = 3
>>> pieces = pieces_trouvees
>>> for semaine in range(1, 53):
...     pieces = pieces + pieces_magiques - pieces_volees
...     print('Semaine %s = %s' % (semaine, pieces))

Semaine 1 = 57
Semaine 2 = 184
Semaine 3 = 221
Semaine 4 = 258
Semaine 5 = 295
Semaine 6 = 322
Semaine 7 = 349
Semaine 8 = 386
Semaine 9 = 423
Semaine 10 = 460
Semaine 11 = 497
Semaine 12 = 524
Semaine 13 = 551
Semaine 14 = 578
Semaine 15 = 605
Semaine 16 = 632
Semaine 17 = 659
```

Tant que nous parlons de boucles : while

La boucle `for` n'est pas la seule que tu puisses utiliser en Python. Il y a aussi la boucle `while` (qui signifie « tant que »). La boucle `for` est faite pour les cas où tu sais où elle débute et où elle s'arrête, c'est-à-dire que tu en connais en quelque sorte la longueur, tandis que la boucle `while` sert dans les cas où tu ne connais pas cette longueur et ne sais pas quand elle s'arrêtera de boucler.

Imaginons un escalier avec 20 marches. L'escalier est à l'intérieur d'une maison et tu sais facilement monter 20 marches. La boucle `for` fonctionne ainsi.

```
>>> for marche in range(0, 20):
    print(marche)
```

Imaginons à présent un long escalier taillé dans une montagne pour accéder au sommet. La montagne est vraiment haute et tu risques de manquer d'énergie avant d'atteindre le sommet ou la météo peut devenir mauvaise. Donc tu risques de devoir t'arrêter et éventuellement rebrousser chemin, mais quand ? La boucle `while` fonctionne ainsi.



```
marche = 0
while marche < 10000:
    print(marche)
    if fatigue == True:
        break
    elif meteoaidé == True:
        break
    else:
        marche = marche + 1
```

Si tu essaies d'exécuter ce code, tu obtiens une erreur, parce que nous n'avons pas créé les variables `fatigue` et `meteoaidé`. Même s'il ne fonctionne pas tel quel, il suffit pour montrer un exemple simple de boucle `while`.

Nous commençons par créer une variable nommée `marche`, initialisée à `0`. Ensuite, nous commençons une boucle `while` dont la première ligne se lit ainsi : « Tant que `marche` est plus petite que `10000`, faire ce qui suit. » Il s'agit donc de vérifier à chaque passage par cette ligne que la valeur de `marche` est plus petite que `10000`, qui représente le nombre total des marches de l'escalier. Ainsi, tant que cette condition est vraie, Python exécute le bloc qui suit.

L'instruction `print(marche)` affiche la marche atteinte, puis nous vérifions si la valeur de la variable `fatigue` est égale à *vrai* avec `if fatigue == True:` (*True* est une valeur dite « booléenne », que nous verrons au chapitre 8). Si la condition est vraie, alors le mot-clé `break` (qui signifie littéralement : rompre) permet de sortir de la boucle, autrement dit d'arrêter la boucle immédiatement. Ce mot-clé fonctionne également pour les boucles `for`. Ici, il a pour effet de sauter directement hors du bloc, c'est-à-dire juste après la ligne `marche = marche + 1`.

La ligne `elif meteoalade == True:` vérifie si la variable `meteoalade` contient la valeur `True`. Si c'est le cas, le mot-clé `break` fait sortir Python de la boucle. Si aucune des valeurs de `fatigue` ou de `meteoalade` n'est vraie, Python saute au `else:` et exécute le bloc qui le suit, donc nous ajoutons `1` à la variable `marche` et la boucle reprend au début.

Ainsi donc, les étapes d'une boucle `while` sont les suivantes.

1. Vérifier la condition.
2. Exécuter le code du bloc.
3. Répéter.

Plus généralement, une boucle `while` peut être créée avec une paire de conditions au lieu d'une seule, par exemple comme ceci :

```
❶ >>> x = 45
❷ >>> y = 80
❸ >>> while x < 50 and y < 100:
    x = x + 1
    y = y + 1
    print(x, y)
```

Ici, nous créons ❶ une variable `x` de valeur `45`, puis en ❷ une variable `y` de valeur `80`. La boucle ❸ vérifie deux conditions : d'abord que `x` est plus petit que `50`, puis que `y` est plus petit que `100`. Tant que ces deux conditions sont vraies, les lignes suivantes sont exécutées et ajoutent `1` à chacune des deux variables, puis les affichent. Voici les résultats d'exécution du code :

```
46 81
47 82
48 83
49 84
50 85
```

Te représentes-tu bien comment cela fonctionne ?

Nous commençons à compter à partir de `45` pour la variable `x` et de `80` pour la variable `y`, puis nous les incrémentons de `1` (c'est-à-dire que nous leur ajoutons `1`) chaque fois que le code du bloc de la boucle s'exécute. La boucle s'exécute tant que `x` est plus petit que `50` et que `y` est plus petit que `100`. Or, après cinq passages dans la boucle, la valeur de `x` atteint `50`, donc la condition `x < 50` n'est plus vraie ; Python cesse de boucler et sort.

Une utilisation fréquente de `while` concerne ce que l'on appelle des boucles semi-infinies. Il s'agit d'un type de boucle qui ne s'arrête que lorsque quelque chose se produit dans le code qui la fait cesser. En voici un exemple :

```
while True:  
    plein de code ici  
    plein de code ici  
    plein de code ici  
    if une_certaine_valeur == True:  
        break
```

La condition de la boucle `while` contient simplement `True`, qui est toujours vrai, donc le code du bloc s'exécute toujours (c'est pour cela que l'on parle de « boucle infinie »). Python ne quitte la boucle que si la variable `une_certaine_valeur` devient vraie (`True`). Nous verrons un meilleur exemple de ceci dans « Obtenir un nombre au hasard à l'aide de `randint` », mais il vaut mieux attendre d'arriver au chapitre 10 pour l'examiner.

Ce que tu as appris

Dans ce chapitre, nous avons utilisé des boucles pour effectuer des tâches répétitives. Nous avons indiqué à Python ce que nous voulions répéter en écrivant ces tâches à l'intérieur de blocs de code placés dans des boucles. Nous en avons vu deux types : les boucles `for` et les boucles `while`. Nous avons aussi appris à les quitter à l'aide du mot-clé `break`.

Puzzles de programmation

Voici quelques exemples de boucles que tu peux essayer de réaliser toi-même. Les réponses sont disponibles sur le site web d'accompagnement du livre.

1. La boucle Bonjour

Que penses-tu que ce code fait ? Essaie de deviner d'abord ce qui se produit, puis exécute le code dans le shell de Python pour vérifier si tu as raison.

```
>>> for x in range(0, 20):
    print('Bonjour %s' % x)
    if x < 9:
        break
```

2. Nombres pairs

Crée une boucle qui n'affiche que les nombres pairs (2, 4, 6, etc.) jusqu'à atteindre ton âge. Ou, si ton âge est un nombre impair (1, 3, 5, etc.), n'affiche que les nombres impairs jusqu'à ton âge. Par exemple, tu pourrais afficher quelque chose comme ceci :

```
2
4
6
8
10
12
14
```

3. Mes cinq ingrédients préférés

Crée une liste qui contienne cinq ingrédients indispensables pour un bon sandwich, par exemple :

```
>>> ingredients = ['escargots', 'sangues', 'tranche de gorille',
    'sourcils de chenilles', 'orteils de mille-pattes']
```

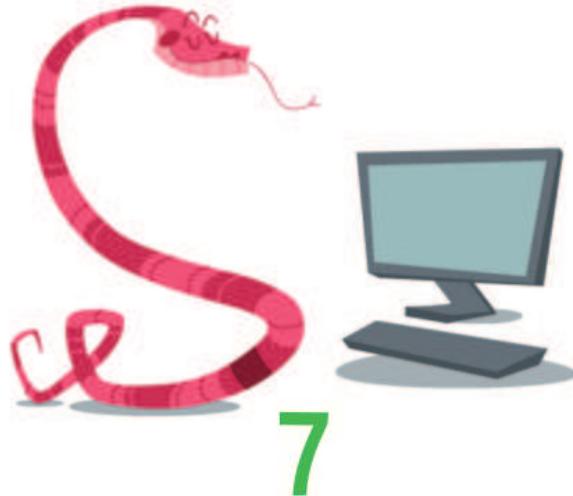
Maintenant, crée une boucle qui affiche le contenu de cette liste, avec les numéros :

```
1 escargots
2 sangues
3 tranche de gorille
4 sourcils de chenilles
5 orteils de mille-pattes
```

4. Ton poids sur la lune

Si tu étais en ce moment sur la lune, ton poids ne représenterait que 16,5 % de celui que tu as sur la terre. Pour le calculer, multiplie ton poids sur terre par 0,165 (attention au point décimal : **0.165** !).

Supposons que tu prennes un kilo de plus tous les ans pendant les 15 années à venir. Utilise une boucle **for** pour afficher ton poids sur la lune pour chacune de ces années.



RECYCLER DU CODE AVEC DES FONCTIONS ET DES MODULES

Imagine deux secondes la quantité de matières que tu jettes chaque jour : des bouteilles, des emballages divers, des restes de nourriture, des sacs en plastique, des journaux, et ainsi de suite. Imagine à présent que tout cela s'empile au bout de ta rue, sans tri sélectif des papiers, des emballages plastiques et métalliques, des verres...

Bien entendu, tu recycles sans doute autant que possible. D'abord parce que personne n'aime enjamber des tas d'immondices en allant

à l'école. Ensuite, au lieu de s'accumuler en une pile énorme, ces bouteilles que tu recycles sont fondues pour être transformées en nouveaux récipients, le papier est recyclé dans du nouveau papier, tandis que les plastiques sont récupérés pour fabriquer de nouveaux emballages, des tubes pour la construction et même des vêtements ! Nous n'avons pas le choix : nous devons recycler les choses qui, sinon, seraient perdues.

Dans le monde de la programmation, la réutilisation est tout aussi importante. Évidemment, tes programmes ne disparaissent pas sous un tas d'immondices mais, si tu ne réutilises pas une partie de ce que tu fais, tu risques d'attraper des boursouflures aux bouts des doigts à force de taper et de retaper les mêmes choses. En plus, la réutilisation de code permet de simplifier tes programmes et d'en faciliter la relecture.

Comme ce chapitre va t'expliquer, Python offre toute une série d'options pour réutiliser du code.



Utiliser des fonctions

Tu as déjà aperçu une des manières dont Python recycle du code. Au chapitre précédent, nous avons utilisé les fonctions `range` et `list` pour compter des éléments.

```
>>> list(range(0, 5))
[0,1,2,3,4]
```

Si tu sais compter, il n'est pas difficile de créer une liste de nombres successifs en les entrant toi-même, mais plus la liste est longue, plus il te faut taper de code. En revanche, si tu utilises des fonctions, tu crées facilement une liste de milliers de nombres.

Voici un exemple qui se sert des fonctions `list` et `range` pour créer une liste de nombres :

```
>>> list(range(0, 1000))
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16...,997,998,999]
```

Les fonctions sont des extraits de code qui indiquent à Python d'effectuer quelque chose. Une méthode permet de réutiliser du code de suite : utilise des fonctions dans tes programmes, aussi vite et aussi souvent que tu peux.

Lorsque tu rédiges des programmes simples, les fonctions sont déjà pratiques mais, quand tu commences à écrire des programmes longs et complexes, comme des jeux, les fonctions deviennent *indispensables* (en supposant bien sûr que tu comptes terminer ton programme dans le courant de ce siècle).

Qu'est-ce qu'une fonction ?

Une fonction possède trois parties : un *nom*, des *paramètres* et un *corps*. Voici un exemple de fonction simple :

```
>>> def foncttest(mon_nom):
    print('Bonjour %s' % mon_nom)
```

Le nom de cette fonction est `foncttest`. Elle possède un seul paramètre, `mon_nom`, et son corps est constitué du bloc de code qui suit immédiatement la ligne commençant par `def` (le diminutif de définition). Un paramètre est une variable qui n'existe que tant que la fonction est utilisée.

Pour exécuter la fonction, il suffit d'appeler son nom, avec des parenthèses autour de la valeur du paramètre :

```
>>> foncttest('Blaise')
Bonjour Blaise
```

Une fonction peut accepter deux, trois ou n'importe quel nombre de paramètres, au lieu d'un seul :

```
>>> def foncttest(pnom, nom):
    print('Bonjour %s %s' % (pnom, nom))
```

Les valeurs de ces deux paramètres sont séparées par une virgule :

```
>>> foncttest('Blaise', 'Pascal')
Bonjour Blaise Pascal
```

Nous pouvons aussi créer des variables pour les utiliser comme paramètres dans l'appel de fonction :

```
>>> prenom = 'Gustave'
>>> nomfam = 'Flaubert'
>>> foncttest(prenom, nomfam)
Bonjour Gustave Flaubert
```

Généralement, une fonction renvoie (*return* en anglais) une valeur à l'aide d'une instruction `return`. Par exemple, tu pourrais écrire une fonction pour calculer l'argent que tu épargnes :

```
>>> def epargne(argent_poche, petits_boulots, depenses):  
    return argent_poche + petits_boulots - depenses
```

Cette fonction prend trois paramètres, additionne les deux premiers (`argent_poche` et `petits_boulots`) et en soustrait le troisième (`depenses`). Le résultat est renvoyé et peut être affiché ou affecté à une variable, de la même manière que les autres valeurs :

```
>>> print(epargne(10, 10, 5))  
15
```

Variabes et portée

Une variable présente dans le corps d'une fonction ne peut plus servir lorsque l'exécution de cette dernière est terminée, parce qu'elle n'existe qu'à l'intérieur de la fonction. Dans le monde de la programmation, cela s'appelle la **portée** de la variable.

Pour bien comprendre ce que cela signifie, prenons l'exemple d'une fonction simple qui utilise deux variables, mais aucun paramètre :

```
❶ >>> def test_variables():  
    premiere_variable = 10  
    seconde_variable = 20  
❷     return premiere_variable * seconde_variable
```

Cet exemple crée une fonction nommée `test_variables` ❶, qui définit deux variables numériques et renvoie ❷ le résultat de leur multiplication. L'appel de la fonction provoque l'affichage suivant :

```
>>> print(test_variables())  
200
```

En revanche, si nous essayons d'afficher ensuite le contenu de la `premiere_variable` (ou de la seconde) en dehors du bloc de code qui forme le corps de la fonction, nous obtenons un message d'erreur :

```
>>> print(premiere_variable)  
Traceback (most recent call last):  
File "<pyshell#50>", line 1, in <module>  
    print(premiere_variable)
```

```
NameError: name 'premiere_variable' is not defined
```

Ce message dit que `premiere_variable` n'est pas définie au moment de la demande d'affichage.

Si nous définissons une variable en dehors de la fonction, sa portée est différente. Par exemple, définissons `autre_variable` avant de créer la fonction, puis essayons de l'utiliser dans le corps de la fonction :

```
❶ >>> autre_variable = 100
      >>> def test_variables2():
              premiere_variable = 10
              seconde_variable = 20
❷       return premiere_variable * seconde_variable * autre_variable
```

Ici, `premiere_variable` et `seconde_variable` ne peuvent servir en dehors de la fonction ; en revanche, `autre_variable`, créée en dehors de `test_variables2()` ❶, peut servir à l'intérieur de la fonction ❷. Voici le résultat de l'appel de cette fonction :

```
>>> print(test_variables2())
20000
```

Imaginons à présent que tu décides de construire un vaisseau spatial à partir d'un matériau économique comme des canettes recyclées. Tu penses pouvoir aplatisir deux canettes par semaine pour fabriquer les parois incurvées du vaisseau, mais il te faut quelque 500 canettes pour finir le fuselage. Nous pouvons écrire une fonction pour t'aider à calculer le temps nécessaire pour aplatisir les 500 canettes.



Créons une fonction pour montrer le nombre de canettes aplatis au fur et à mesure des semaines pendant un an. Elle prend en paramètre le nombre de canettes déjà aplatis :

```
>>> def construction_vaisseau(canettes):
          total_canettes = 0
          for semaine in range(1, 53):
              total_canettes = total_canettes + canettes
              print('Semaine %s = %s canettes' % (semaine, total_canettes))
```

À la première ligne du corps de la fonction, nous créons une variable `total_canettes` initialisée à `0`. Nous débutons ensuite une boucle pour les semaines de l'année, puis nous additionnons le

nombre de canettes aplatis chaque semaine, pour l'afficher. Ce bloc de code forme le corps de la fonction, mais il y a un autre bloc de code dans cette fonction : les deux dernières lignes forment le bloc de la boucle `for`.

Essayons d'entrer cette fonction dans le shell et de l'appeler avec des valeurs différentes du nombre de canettes par semaine :

```
>>> construction_vaisseau(2)
Semaine 1 = 2 canettes
Semaine 2 = 4 canettes
Semaine 3 = 6 canettes
Semaine 4 = 8 canettes
Semaine 5 = 10 canettes
Semaine 6 = 12 canettes
Semaine 7 = 14 canettes
Semaine 8 = 16 canettes
Semaine 9 = 18 canettes
Semaine 10 = 20 canettes
(et ainsi de suite...)

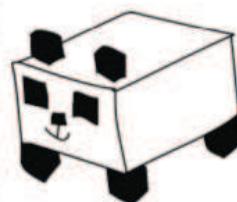
>>> construction_vaisseau(13)
Semaine 1 = 13 canettes
Semaine 2 = 26 canettes
Semaine 3 = 39 canettes
Semaine 4 = 52 canettes
Semaine 5 = 65 canettes
(et ainsi de suite...)
```

Cette fonction peut servir plusieurs fois avec des valeurs différentes pour le nombre de canettes aplatis par semaine, ce qui s'avère bien plus efficace que de retaper toute la boucle `for` chaque fois que nous voulons essayer d'autres nombres.

Il est également possible de regrouper des fonctions dans des modules et c'est là que Python se montre réellement utile.

Utiliser des modules

Un **module** sert à grouper des fonctions, des variables et d'autres choses dans des programmes plus vastes et plus puissants. Certains modules sont intégrés dans Python lui-même, tandis que tu peux en télécharger d'autres de manière séparée. Certains t'aident à écrire des jeux (comme `tkinter`, intégré dans Python, ou `PyGame`, qui



ne l'est pas), d'autres à manipuler des images (comme `PIL`, la bibliothèque d'imagerie de Python) ou à dessiner en trois dimensions (comme `Panda3D`).

Des modules peuvent servir à réaliser toutes sortes de choses utiles. Ainsi, si tu conçois un jeu de simulation et si tu souhaites que le monde du jeu change de façon réaliste, tu peux calculer la date et l'heure actuelles à l'aide du module intégré appelé `time` :

```
>>> import time
```

La commande `import` indique à Python que nous voulons utiliser le module `time`. Ensuite seulement, nous pouvons appeler les fonctions disponibles dans ce module. Au chapitre 4, nous avons utilisé des fonctions du module `turtle`, comme `t.forward(50)`. Par exemple, voici un appel de la fonction `asctime` du module `time`, qui renvoie la date et l'heure actuelles en anglais sous forme d'une chaîne :

```
>>> print(time.asctime())
Tue Sep 9 06:51:47 2014
```

Supposons à présent que tu veuilles demander à la personne utilisant ton programme d'entrer une valeur au clavier, par exemple sa date de naissance. Tu pourrais l'afficher sous forme d'un message dans une instruction `print`. Pour cela, il existe le module `sys` (l'abréviation de système), qui contient des utilitaires pour interagir avec le système Python lui-même. D'abord, importe le module `sys` :



```
>>> import sys
```

Dans le module `sys` existe un **objet** spécial, nommé `stdin` (pour « entrée standard »), qui fournit une fonction plutôt utile, appelée `readline` (*lire ligne*). Cette fonction lit le contenu d'une ligne de texte saisie au clavier, jusqu'à l'appui sur la touche `Entrée`. Nous verrons plus de détails sur les objets au chapitre 8 mais, pour l'instant, voyons comment fonctionne `readline`. Tape le code suivant dans le shell :

```
>>> import sys
>>> print(sys.stdin.readline())
```

Rappelle-toi ce code du chapitre 5, qui contenait une instruction `if` :

```
>>> if age >= 10 and age <= 13:  
    print('Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !')  
else:  
    print('Pardon ?')
```

Au lieu de créer la variable `age` et de lui donner une valeur spécifique avant l'instruction `if`, nous pouvons cette fois demander à quelqu'un d'entrer la valeur. Convertissons d'abord le code en une fonction :

```
>>> def age_blaque_idiote(age):  
    if age >= 10 and age <= 13:  
        print('Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !')  
    else:  
        print('Pardon ?')
```

À partir de là, il est possible d'appeler la fonction par son nom et de lui donner en paramètre entre parenthèses le nombre à utiliser. Cela fonctionne-t-il ?

```
>>> age_blaque_idiote(9)  
Pardon ?  
>>> age_blaque_idiote(10)  
Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !
```

Oui, ça fonctionne ! Voyons à présent comment faire pour que la fonction demande l'âge de la personne. Tu peux ajouter ou modifier une fonction autant de fois que tu le veux. Note que le deuxième `print` dans le code suivant tient sur la même ligne.

```
>>> def age_blaque_idiote():  
    print('Quel âge as-tu ?')  
①    age = int(sys.stdin.readline())  
②    if age >= 10 and age <= 13:  
        print('Que donnent 13 + 49 + 84 + 155 + 97 ? ↵  
              La migraine !')  
    else:  
        print('Pardon ?')
```

As-tu reconnu la fonction `int` ①, qui convertit une chaîne en un nombre ? Nous sommes obligés d'utiliser cette fonction, parce que `readline()` renvoie tout ce que l'utilisateur entre au clavier sous forme d'une chaîne, or nous avons besoin d'un nombre pour le comparer ② aux nombres `10` et `13`.

Pour essayer cela, il nous reste à appeler la fonction sans paramètre, puis à entrer un nombre lorsque s'affiche la question Quel âge as-tu :

```
>>> age_blaque_idiote()
Quel âge as-tu ?
10
Que donnent 13 + 49 + 84 + 155 + 97 ? La migraine !
>>> age_blaque_idiote()
Quel âge as-tu ?
15
Pardon ?
```

Ce que tu as appris

Dans ce chapitre, tu as vu comment rendre réutilisables des extraits de code en Python à l'aide de fonctions et comment utiliser les fonctions présentes dans des modules. Tu as appris aussi que la portée des variables contrôle leur visibilité à l'intérieur et à l'extérieur des fonctions. La création de fonctions repose sur le mot-clé `def`. Enfin, tu as appris à importer des modules pour pouvoir en utiliser le contenu.



Puzzles de programmation

Essaie de programmer toi-même les exemples suivants, pour t'entraîner à la création de tes propres fonctions. Les réponses sont disponibles dans le site web d'accompagnement du livre.

1. Fonction de base du poids sur la lune

Au chapitre 6, un des exercices consistait à créer une boucle `for` pour déterminer ton poids sur la lune sur une période de 15 ans. Tu peux aisément convertir cette boucle en une fonction. Crée une fonction qui accepte le poids de départ sur terre et un incrément de poids annuel, dont le poids de départ est augmenté chaque année. Tu pourrais ensuite appeler cette nouvelle fonction comme ceci :

```
>>> poids_lune(30, 0.25)
```

2. Fonction poids sur la lune avec les années

Prends la fonction que tu viens juste de créer et modifie-la pour traiter les poids sur des périodes différentes, par exemple sur 5 ou 20 années. Assure-toi de modifier la fonction pour qu'elle accepte trois arguments : le poids initial, le poids ajouté chaque année et le nombre d'années au total :

```
>>> poids_lune(30, 0.25, 5)
```

3. Programme de poids sur la lune

Au lieu de n'utiliser qu'une seule fonction à laquelle tu passes les valeurs en paramètres, tu peux réaliser un petit programme qui demande d'entrer les valeurs et les saisit à l'aide de `sys.stdin.readline()`. Dans ce cas-ci, tu appelles la fonction sans paramètre, puisqu'elle les demande à l'utilisateur :

```
>>> poids_lune()
```

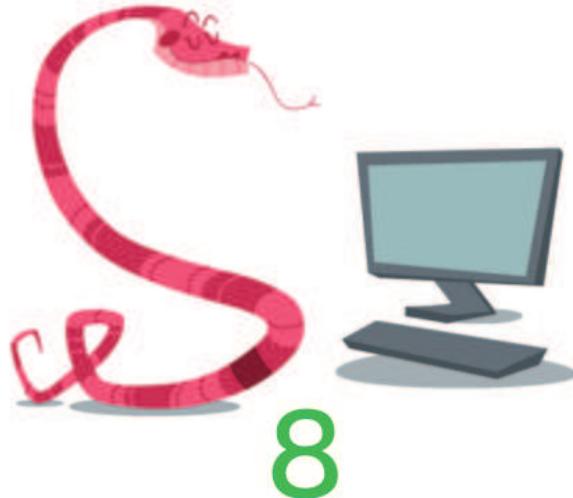
La fonction affiche un premier message pour demander le poids initial, un deuxième pour demander le poids ajouté chaque année et enfin un troisième qui demande le nombre d'années. Le résultat devrait ressembler à ceci :

```
Entre ton poids actuel sur la terre  
45  
Entre le poids que tu prends en plus chaque année  
0.4  
Entre le nombre d'années  
12
```

N'oublie pas d'importer le module `sys` avant de créer la fonction :

```
>>> import sys
```

Comme le deuxième nombre entré est un nombre décimal et non plus un entier, il faut réfléchir aussi à la façon de le convertir, avant de l'insérer dans les calculs. Au chapitre 5, nous avions donné une piste.



CLASSES ET OBJETS

Quel est le rapport entre une girafe et un trottoir ? Ce sont tous deux des choses, désignées dans le langage courant par des noms ; du point de vue de Python, il s'agit d'**objets**.

L'idée des objets est très importante dans le monde informatique. Ils permettent d'organiser du code dans un **programme** et de découper les choses en petits morceaux pour faciliter la réflexion à propos d'idées complexes. Nous avons déjà utilisé un objet au chapitre 4, lorsque nous avons travaillé avec le **module** turtle de la tortue : l'objet `Pen`.

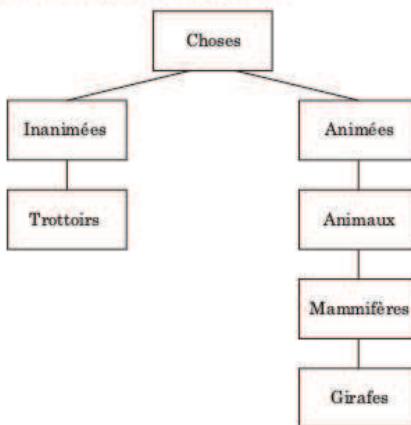
Pour bien comprendre comment fonctionnent les objets en Python, nous devons réfléchir à leurs types. Commençons avec les girafes et les trottoirs.

La girafe est un type de mammifère, qui est lui-même un type d'animal. La girafe est aussi un objet animé, puisqu'elle est vivante. Examinons maintenant le trottoir. Il n'y a pas grand-chose à dire à son sujet, sauf que c'est un objet inanimé, car non vivant. Les termes mammifère, animal, animé et inanimé sont autant de moyens de ranger les choses dans des classes.



Organiser les choses en classes

En Python, les objets sont définis par des **classes**, qui servent à les ordonner dans des groupes. Voici un diagramme en arbre des classes dans lesquelles les girafes et les trottoirs peuvent se ranger selon nos définitions précédentes :



La classe principale est celle des Choses. Sous elle, nous avons les Inanimées et les Animées, que nous pouvons préciser encore avec les Trottoirs pour les premiers et les Animaux pour les seconds. Puis, en descendant encore un peu, nous avons les Animaux, les Mammifères et, enfin, les Girafes pour les Animées.

Utilisons des classes pour organiser un peu le code Python. Par exemple, regardons d'un peu plus près le module `turtle`. Tout ce qu'il est capable de faire, comme avancer, reculer, tourner à gauche et ainsi de suite, est constitué de fonctions de la classe `Pen`. Il nous est possible de voir un objet comme étant un membre d'une classe et de créer autant d'objets de cette classe que nous le voulons, comme nous allons le voir bientôt.

Pour l'heure, réalisons l'ensemble des classes de notre diagramme, à partir du haut (`Choses`). Pour cela, utilise le mot-clé `class`, suivi d'un nom :

```
>>> class Choses:  
    pass
```

Nous nommons la classe `Choses` et utilisons l'instruction `pass` pour indiquer à Python que nous n'ajoutons pas d'autre information à son propos. Celle-ci sert en effet lorsque nous souhaitons créer une classe ou une fonction mais sans en remplir les détails pour le moment.

Ensuite, nous ajoutons les autres classes et établissons quelques relations parmi elles.

Enfants et parents

Si une classe A fait partie d'une classe B, on dit que A est un **enfant** de B et que B est le **parent** de A. Des classes peuvent être à la fois filles (enfants) de certaines classes et parentes d'autres.

Dans le diagramme en arbre, la classe au-dessus d'une autre est son parent, tandis que celle en dessous est son enfant (ou fille). Ainsi, `Inanimées` et `Animées` sont toutes deux des enfants de la classe `Choses`, ce qui implique que celle-ci est leur parent. Pour des raisons pratiques, nous masculiniserons le nom des classes et éviterons les accents : `Inanimés` et `Animés`.

Pour indiquer à Python qu'une classe est fille d'une autre, nous plaçons le nom de la classe parente entre parenthèses, après celui de la nouvelle classe, comme ceci :

```
>>> class Inanimés(Choses): ❶  
    pass  
  
>>> class Animés(Choses): ❷  
    pass
```

Avec la ligne ❶, nous créons une classe nommée `Inanimes` et nous indiquons à Python que sa classe parente est `Choses`. Ensuite, en ❷, c'est au tour de la classe nommée `Animes` ; nous précisons à Python que sa classe parente est aussi `Choses`.

Poursuivons avec la classe `Trottoirs`, dont le parent est `Inanimes` :

```
>>> class Trottoirs(Inanimes):
        pass
```

Et nous pouvons également établir les classes `Animaux`, `Mammifères` et `Girafes` avec leurs parents respectifs :

```
>>> class Animaux(Animes):
        pass

>>> class Mammiferes(Animaux):
        pass

>>> class Girafes(Mammiferes):
        pass
```

Ajouter des objets aux classes

Maintenant que nous avons toute une série de classes, nous allons ajouter un certain nombre de choses. Admettons que nous ayons une girafe nommée Régine. Nous savons qu'elle appartient à la classe `Girafes`, mais qu'utilise-t-on, en termes de programmation, pour décrire une seule girafe nommée Régine ? En fait, on appelle Régine un **objet** de la classe `Girafes`. On dit aussi que c'est une **instance** de cette classe. Pour présenter Régine à Python, nous utilisons le code suivant :

```
>>> regine = Girafes()
```

Cette ligne de code dit à Python de produire un objet de la classe `Girafes` et de l'affecter à la variable `regine`. Comme pour une fonction, le nom de la classe est suivi de parenthèses. Plus loin dans ce chapitre, nous verrons qu'il est possible de créer des objets et de placer des paramètres entre les parenthèses.

Ensuite, voyons ce que peut faire l'objet `regine`. En fait, pas grand-chose pour l'instant car, pour que les objets servent à quelque chose, il faut aussi définir dans leur classe des fonctions qu'ils pourront utiliser. Au lieu de placer le mot-clé `pass` directement après la définition de la classe, nous ajouterons des définitions de fonctions.

Définir des fonctions de classes

Le chapitre 7 présentait les fonctions comme une manière de réutiliser du code. Lorsque nous définissons une fonction associée à une classe, la manière est la même que pour des fonctions normales, sauf que nous appliquons un retrait par rapport à la définition de la classe. Voici, par exemple, une fonction qui ne s'associe pas à une classe :

```
>>> def voici_une_fonction_normale():
    print("Je suis une fonction normale.")
```

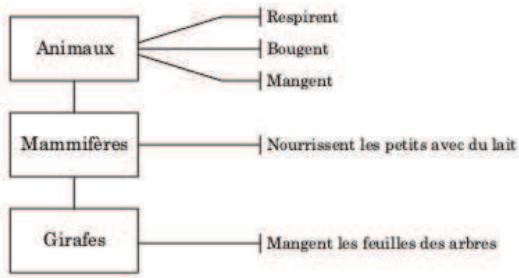
Et voici en comparaison deux fonctions qui appartiennent à une classe :

```
>>> class JeSuisUneClasse:
    def voici_une_fonction_de_Classe():
        print("Je suis une fonction de classe.")
    def voici_une_autre_fonction_de_Classe():
        print("Je suis une autre fonction de classe. Vu ?")
```

Ajouter des caractéristiques à une classe avec des fonctions

Regardons à présent les classes filles de la classe `Animes` que nous avons définies précédemment. Ajoutons des caractéristiques à chaque classe pour décrire ce qu'elle est et ce qu'elle sait faire. Une caractéristique est une particularité, une spécialité que tous les membres de cette classe (et leurs filles) ont en commun.

Ainsi, qu'est-ce que tous les animaux ont en commun ? Bon, pour commencer, ils respirent, ils bougent et ils mangent. Et les mammifères ? Ils nourrissent leurs enfants avec du lait, ils respirent, ils bougent et ils mangent. Nous savons en outre que les girafes mangent les feuilles hautes des arbres et que, comme les mammifères, elles nourrissent leurs petits avec leur lait, qu'elles respirent, bougent et mangent. Bref, lorsque nous ajoutons toutes ces caractéristiques à notre diagramme (dont nous ne montrons ici que la partie intéressante), nous obtenons :



Nous considérons ces caractéristiques comme des actions, ou des **fonctions**, c'est-à-dire des choses qu'un objet de telle classe peut faire. Nous utiliserons de préférence des verbes à l'infinitif pour les nommer.

Pour ajouter une fonction à une classe, nous utilisons le mot-clé `def`. La classe `Animaux` prend donc l'aspect suivant :

```
>>> class Animaux(Animales):
        def respirer(self):
            pass
        def bouger(self):
            pass
        def manger(self):
            pass
```



À la première ligne de ce code, nous définissons la classe `Animaux` comme précédemment mais, au lieu de laisser simplement le mot-clé `pass` à la ligne qui suit, nous établissons une fonction `respirer` avec un seul paramètre : `self`. Celui-ci permet qu'une fonction d'une classe puisse en appeler une autre de la classe (et dans les classes parentes). Nous en saurons plus sur son usage par la suite.

À la ligne suivante, le mot-clé `pass` dit à Python que nous ne donnons pas plus d'informations à propos de la fonction `respirer`, parce qu'elle ne fait rien pour l'instant. Ensuite, nous ajoutons de la même manière les fonctions (et actions de) `bouger` et `manger`, qui ne font rien non plus pour le moment. Nous allons bientôt réécrire nos classes et placer du code dans les différentes fonctions. Cette manière de développer, avec des fonctions et des classes vides, est très habituelle chez les programmeurs. Ils préparent ainsi leurs classes et leurs fonctions qui, au départ, ne font rien du tout. Ce n'est qu'ensuite, lorsqu'ils savent plus précisément ce qu'elles doivent faire, qu'ils se

penchent sur la manière dont elles le font. Il leur reste à rédiger réellement le code des fonctions, l'une après l'autre.

Il est aussi possible d'ajouter des fonctions aux deux autres classes, `Mammifères` et `Girafes`. Chacune pourra utiliser les caractéristiques, donc les fonctions, de son parent. Cela signifie que tu ne dois pas nécessairement créer des classes très compliquées, car tu peux aussi placer certaines fonctions dans le parent le plus élevé auquel s'applique une caractéristique. C'est un bon moyen de faire des classes plus simples et plus faciles à comprendre.

```
>>> class Mammifères(Animaux):
...     def nourrir_petits_avec_du_lait(self):
...         pass
...
>>> class Girafes(Mammifères):
...     def manger_feuilles_des_arbres(self):
...         pass
```

Pourquoi utiliser des classes et des objets ?

Maintenant que nous avons ajouté des fonctions à nos classes, la question suivante se pose : pourquoi utiliser des classes et des objets, alors que nous aurions aussi bien pu écrire leurs fonctions comme des fonctions normales : `respirer`, `bouger`, `manger`, et ainsi de suite ?

Pour répondre à cette question, nous allons reprendre l'exemple de Régine, notre girafe. Nous avons créé l'objet `regine` de la classe `Girafes` comme ceci :

```
>>> regine = Girafes()
```

Comme `regine` est un objet, nous pouvons appeler (c'est-à-dire **exécuter**) les fonctions fournies par sa classe (`Girafes`) mais aussi par ses classes parentes. Pour cela, utilise l'opérateur point (.) et le nom de la fonction. Par exemple, pour dire à la girafe Régine de bouger ou de manger, appelle les fonctions comme ceci :

```
>>> regine = Girafes()
>>> regine.bouger()
>>> regine.manger_feuilles_des_arbres()
```

Supposons que Régine ait un ami, également girafe, qui se nomme Jules. Créeons un autre objet `Girafes` nommé `jules` :

```
>>> jules = Girafes()
```

Vu que nous utilisons des objets et des classes, nous pouvons indiquer à Python précisément de quelle girafe on parle lorsqu'on veut exécuter la fonction bouger. Ainsi, si nous voulons que Jules bouge et laisse Régine où elle est, appelons la fonction `bouger` à partir de notre objet `jules` (dans ce cas-ci, seul Jules se déplace) :

```
>>> jules.bouger()
```

Modifions légèrement nos classes pour que cela devienne plus évident. Ajoutons simplement des instructions `print` à chaque fonction, à la place de `pass` :

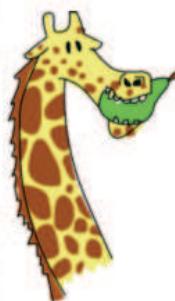
```
>>> class Animaux(Animales):
...     def respirer(self):
...         print("respire")
...     def bouger(self):
...         print("bouge")
...     def manger(self):
...         print("mange")

>>> class Mammiferes(Animaux):
...     def nourrir_petits_avec_du_lait(self):
...         print("nourrit les petits")

>>> class Girafes(Mammiferes):
...     def manger_feuilles_des_arbres(self):
...         print("mange des feuilles")
```

Maintenant, quand nous créons nos deux objets `regine` et `jules`, puis appelons des fonctions au départ d'eux, quelque chose se passe :

```
>>> regine = Girafes()
>>> jules = Girafes()
>>> jules.bouger()
bouge
>>> regine.manger_feuilles_des_arbres()
mange des feuilles
```



Aux deux premières lignes, nous établissons les variables `regine` et `jules`, objets de la classe `Girafes`. Ensuite, nous appelons la fonction `bouger` à partir de l'objet `jules`, et Python affiche `bouge` à la ligne suivante. De la même manière, nous appelons la fonction `manger_feuilles_des_arbres` de `regine`, et Python affiche `mange des feuilles`. Si c'étaient de vraies girafes et non de simples objets dans un ordi-

nateur, l'une d'elles marcherait et l'autre mangerait des feuilles en haut d'un arbre.

Objets et classes en images

Voyons s'il est possible d'adopter une approche graphique des objets et des classes.

Reprendons le module `turtle` avec lequel nous avons joué au chapitre 4. Lorsque nous écrivons `turtle.Pen()`, Python crée un objet de la classe `Pen`, fourni par le module `turtle`, d'une manière semblable à nos objets `regine` et `jules` de la section précédente. Ici aussi, établissons deux objets tortues que nous nommons Aline et Karine, à la manière des deux girafes :

```
>>> import turtle  
>>> aline = turtle.Pen()  
>>> karine = turtle.Pen()
```

Chaque objet tortue est un membre de la classe `Pen`.

C'est ici que nous allons voir toute la puissance des objets. Nos deux objets tortues créés, appelons des fonctions sur chacun d'eux, pour les faire dessiner indépendamment :

```
>>> aline.forward(50)  
>>> aline.right(90)  
>>> aline.forward(20)
```

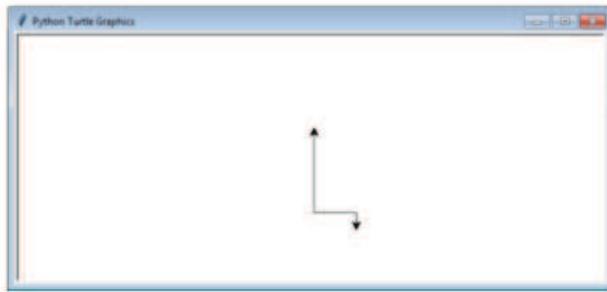
Ces quelques instructions indiquent à Aline d'avancer de 50 pixels, de tourner de 90° vers la droite, puis d'avancer de 20 pixels, pour se retrouver face au bas de l'écran. Rappelle-toi que les tortues démarrent toujours face à la droite de l'écran.

Maintenant, c'est au tour de Karine :

```
>>> karine.left(90)  
>>> karine.forward(100)
```

Nous demandons à Karine de tourner vers la gauche de 90°, puis d'avancer de 100 pixels, de sorte qu'elle finisse face au haut de l'écran.

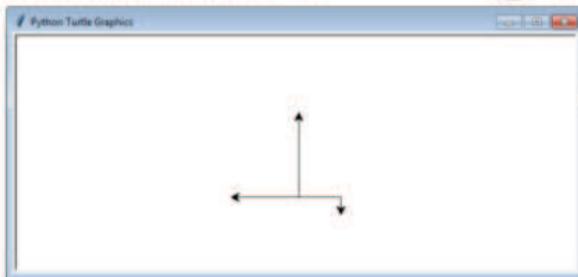
Jusque-là, nous avons une ligne avec deux pointes de flèches (représentant deux objets tortues distincts) qui se déplacent dans des directions différentes : Aline pointe vers le bas et Karine vers le haut.



Ensuite, ajoutons une autre tortue, Jacques, pour la déplacer en dehors des traces d'Aline et de Karine :

```
>>> jacques = turtle.Pen()  
>>> jacques.left(180)  
>>> jacques.forward(80)
```

Nous créons d'abord l'objet `Pen` nommé `jacques`, puis nous le faisons tourner à 180° vers la gauche et nous le déplaçons de 80 pixels vers l'avant. Le dessin prend l'allure suivante, avec les trois tortues :



Retiens donc que, chaque fois que nous appelons `turtle.Pen()` pour créer une tortue, nous ajoutons un nouvel objet indépendant des autres. Chaque objet est une instance de la classe `Pen` et peut utiliser les mêmes fonctions que les autres mais, comme nous utilisons des objets, nous pouvons déplacer chaque tortue de manière indépendante. Tout comme nos deux objets girafes (Régine et Jules), Aline, Karine et Jacques sont des objets tortues indépendants. Si nous créons un objet avec le même nom de variable qu'un autre objet

qui existe déjà, l'ancien ne disparaît pas nécessairement. Essaie par toi-même : crée une nouvelle tortue de nom Karine et essaie de la déplacer à l'écran.

Autres fonctionnalités des objets et des classes

Les classes et les objets simplifient le regroupement des fonctions, mais ils permettent aussi de découper un programme en de plus petits extraits de code.

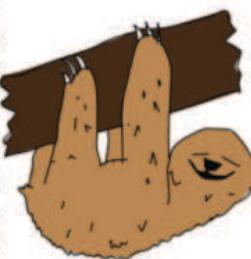
Prenons l'exemple d'une grande application logicielle, telle qu'un traitement de texte ou un jeu en trois dimensions. Il est quasi impossible à une personne normale de comprendre dans leur ensemble de vastes programmes comme ceux-ci, parce qu'ils contiennent trop de code. À partir du moment où l'on découpe ces programmes gigantesques en de petits morceaux, chacun commence à prendre sens à condition de connaître le langage, bien entendu.

Lors de l'écriture d'un vaste programme, une telle découpe divise le travail afin de le répartir entre plusieurs informaticiens. Les programmes les plus complexes que tu utilises, ton navigateur web par exemple, ont été écrits par de nombreuses personnes, qui ont travaillé sur des portions différentes, au même moment et un peu partout dans le monde.

À présent, imagine que tu souhaites étendre certaines des classes que nous avons créées dans ce chapitre ([Animaux](#), [Mammifères](#) et [Girafes](#)). Comme tu as trop de travail, tu désires te faire aider de quelques-uns de tes amis. Pour ce faire, tu peux découper la tâche d'écriture du code pour qu'une personne se penche sur la classe [Animaux](#), une autre sur la classe [Mammifères](#) et la dernière sur la classe [Girafes](#).

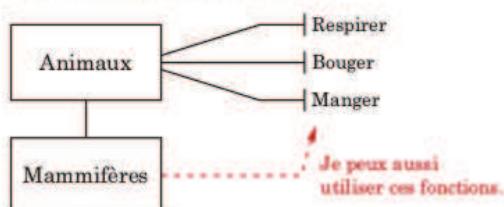
Fonctions héritées

Si tu as été attentif, tu as probablement compris que celui ou celle qui s'occupe de la classe [Girafes](#) est le plus chanceux, car toutes les fonctions créées par les personnes travaillant sur les classes [Animaux](#) et [Mammifères](#) peuvent être réutilisées par la classe [Girafes](#). Celle-ci hérite en effet des fonctions de la classe [Mammifères](#) qui, à son tour, hérite de celles de la classe [Animaux](#). Autrement



dit, avec un objet girafe, nous pouvons utiliser les fonctions définies dans la classe `Girafes`, ainsi que celles des classes `Mammiferes` et `Animaux`. De la même manière, un objet mammifère peut réemployer les fonctions définies dans la classe `Mammiferes`, ainsi que celles de sa classe parente `Animaux`.

Observe une nouvelle fois la relation entre les classes `Animaux`, `Mammiferes` et `Girafes`. La première est le parent de la deuxième, qui elle-même est le parent de la troisième.



Même si `regine` est un objet de la classe `Girafes`, nous pouvons toujours appeler la fonction `bouger`, définie dans la classe `Animaux`, parce que les fonctions déterminées dans une classe parente sont accessibles à ses classes filles :

```
>>> regine = Girafes()
>>> regine.bouger()
bouge
```

En fait, toutes les fonctions définies dans les classes `Animaux` et `Mammiferes` peuvent être appelées par l'objet `regine`, parce qu'il en hérite :

```
>>> regine = Girafes()
>>> regine.respirer()
respire
>>> regine.manger()
mange
>>> regine.nourrir_petits_avec_du_lait()
nourrit les petits
```

Fonctions appelant d'autres fonctions

Lorsque nous appelons des fonctions à partir d'un objet, nous utilisons le nom de la variable. Ainsi, pour appeler la fonction `bouger` sur Régine la girafe, nous écrivons :

```
>>> regine.bouger()
```

Pour qu'une fonction de la classe `Girafes` appelle `bouger`, qui appartient à cette même classe, il faut se servir du paramètre `self` qui sert à appeler une autre fonction de la même classe (ou de ses parents). Ainsi, imaginons que nous ajoutons une fonction nommée `trouver_nourriture` à la classe `Girafes` :

```
>>> class Girafes(Mammiferes):
...     def trouver_nourriture(self):
...         self.bouger()
...         print("J'ai trouvé de la nourriture !")
...         self.manger()
```

Nous venons de créer une fonction qui en combine deux autres, comme cela se fait souvent en programmation. Tu devras fréquemment écrire des fonctions simples de ce type, qui serviront ensuite dans d'autres fonctions plus complexes. C'est ce que nous ferons au chapitre 13 pour réaliser un jeu.

Profitons de `self` pour préciser notre classe `Girafes` :

```
>>> class Girafes(Mammiferes):
...     def trouver_nourriture(self):
...         self.bouger()
...         print("J'ai trouvé de la nourriture !")
...         self.manger()
...     def manger_feuilles_des_arbres(self):
...         self.manger()
...     def danser_une_gigue(self):
...         self.bouger()
...         self.bouger()
...         self.bouger()
...         self.bouger()
```

Nous utilisons les fonctions `manger` et `bouger` de la classe parente `Animaux` pour déterminer `manger_feuilles_des_arbres` et `danser_une_gigue` dans la classe `Girafes`, parce qu'elles sont héritées. L'ajout de fonctions qui en appellent d'autres de cette manière nous permet, lorsque nous créons des objets de ces classes, d'appeler une fonction unique qui effectue plusieurs choses. Ainsi, quand nous appelons `danser_une_gigue`, notre girafe bouge quatre fois (ce qu'indique le texte `bouge` copié quatre fois) :

```
>>> regine = Girafes()
>>> regine.danser_une_gigue()
bouge
```

```
bouge  
bouge  
bouge
```

Initialiser un objet

Quelquefois, lorsque nous créons un objet, il est nécessaire de lui définir certaines valeurs, appelées « propriétés » de l'objet, pour un usage ultérieur. **Initialiser** un objet revient à le préparer pour qu'il soit prêt à l'emploi.

Imaginons, par exemple, que nous devions définir le nombre de taches pour chaque objet girafe au moment de son initialisation. Pour cela, nous créons une fonction `_init_`. Note qu'`_init` est entouré de chaque côté de deux caractères de soulignement (`_`), soit quatre au total.

Il s'agit là d'une fonction d'un type spécial des classes de Python, qui doit porter exactement ce nom-là. Elle sert à définir les propriétés d'un objet au moment de sa toute première création, car Python appelle automatiquement cette fonction lorsque nous créons un objet. Voici comment elle s'écrit :

```
>>> class Girafes:  
...     def __init__(self, taches): ❶  
...         self.taches_girafe = taches ❷
```

D'abord (❶), nous définissons la fonction `init` avec deux paramètres, `self` et `taches`. Comme les autres, elle doit avoir `self` comme premier paramètre. Ensuite (❷), nous affectons le paramètre `taches` à une variable d'objet, autrement dit une propriété, nommée `taches_girafe`. Cette ligne peut se lire comme suit : « Prendre la valeur du paramètre `taches` et la stocker pour un usage ultérieur dans la variable d'objet `taches_girafe`. » De même qu'une fonction d'une classe peut appeler une autre fonction avec le paramètre `self`, l'accès aux variables de la classe, dans la classe, oblige à utiliser `self`.

Ensuite, lorsque nous établissons deux objets girafes, Oscar et Gertrude, et que nous affichons leur nombre de taches, nous voyons la fonction d'initialisation en action :

```
>>> oscar = Girafes(100)  
>>> gertrude = Girafes(150)  
>>> print(oscar.taches_girafe)  
100  
>>> print(gertrude.taches_girafe)  
150
```

Tout d'abord, nous créons une instance de la classe `Girafes` avec la valeur de paramètre `100`. Ceci a pour effet d'appeler automatiquement la fonction `_init_` de la classe et d'utiliser `100` pour la valeur du paramètre `taches`. Ensuite, nous réalisons une seconde instance de la classe `Girafes`, avec cette fois `150` comme valeur de paramètre. Enfin, nous affichons la variable d'objet `taches_girafe` pour chacun des objets, ce qui donne comme résultats `100` et `150`. Parfait !

Retiens bien que, lorsque nous créons un objet d'une classe, comme `oscar`, nous pouvons faire référence à ses variables et fonctions à l'aide de l'opérateur point `(.)`, suivi du nom de la variable ou de la fonction à utiliser (par exemple, `oscar.taches_girafe`). En revanche, avec des fonctions à l'intérieur d'une classe, pour faire référence à ces mêmes variables (et autres fonctions), c'est le paramètre `self` (par exemple `self.taches_girafe`) qui doit être employé.

Ce que tu as appris

Dans ce chapitre, nous avons utilisé des classes pour créer des catégories de choses et construire des objets (instances) de ces classes. Tu as appris que les enfants d'une classe héritent des fonctions de leurs parents et que, même si deux objets sont issus d'une même classe, ce ne sont pas nécessairement des clones. Par exemple, un objet girafe peut posséder son propre nombre de taches.

Tu as vu aussi comment appeler (ou exécuter) des fonctions d'un objet et que les variables d'objet (ou propriétés) permettent de stocker des valeurs dans ces mêmes objets. Enfin, nous avons utilisé le paramètre `self` dans des fonctions pour faire référence à d'autres fonctions et variables. Ces concepts sont fondamentaux dans Python ; tu les reverras donc souvent dans le reste du livre.

Puzzles de programmation

Certaines des idées de ce chapitre commenceront à dévoiler tout leur sens à mesure que tu les manipuleras. Expérimente-les dans les exemples suivants, puis vérifie les réponses sur le site d'accompagnement du livre.

1. Moulinet de girafe

Ajoute des fonctions à la classe `Girafes` pour déplacer les pattes postérieures gauche et droite en avant et en arrière. Voici un exemple de fonction pour bouger la patte postérieure gauche vers l'avant :

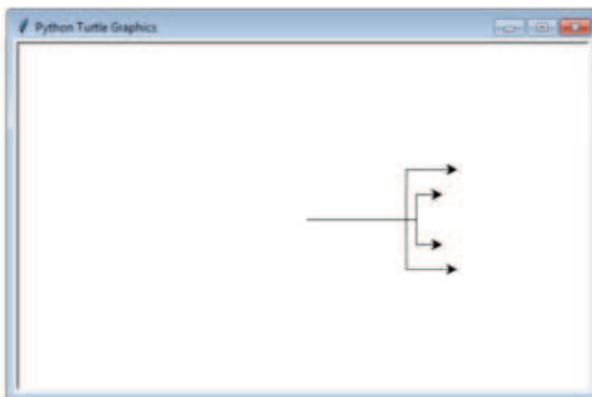
```
>>> def pied_gauche_en_avant(self):
    print("pied gauche en avant")
```

Crée ensuite une fonction pour faire danser Régine. Elle doit appeler les quatre fonctions de pieds que tu auras réalisées. Le résultat de l'appel de cette nouvelle fonction est un simple pas de danse :

```
>>> regine = Girafes()
>>> regine.danser()
pied gauche en avant
pied gauche en arrière
pied droit en avant
pied droit en arrière
pied gauche en arrière
pied droit en arrière
pied droit en avant
pied gauche en avant
```

2. Fourche de tortues

Réalise l'image suivante d'une fourche, à l'aide de quatre objets `Pen` du module `turtle`. Les longueurs des lignes n'ont pas d'importance. N'oublie pas d'importer le module avant !





FONCTIONS INTÉGRÉES DE PYTHON

Python possède une belle boîte pleine d'outils de programmation, avec notamment un grand nombre de fonctions et de modules prêts à l'emploi, totalement à ta disposition. Comme un bon marteau ou une clé universelle pour vélo, ces outils intégrés permettent d'écrire bien plus facilement des programmes et ce sont en réalité de véritables extraits de code.

Au chapitre 7, tu as vu qu'il est nécessaire d'importer un module avant de pouvoir l'utiliser. Les fonctions intégrées de Python ne nécessitent pas d'importation avant leur utilisation car elles sont accessibles dès que le shell démarre. Dans ce chapitre, nous allons examiner quelques-unes des plus utiles, puis nous concentrer sur la fonction `open`, qui sert à ouvrir des fichiers pour en lire ou y écrire des informations.

Utiliser des fonctions intégrées

Nous allons examiner 12 fonctions intégrées habituellement utilisées par les programmeurs en Python, avec leur description, la manière de les employer et des exemples des avantages que tu auras à t'en servir.

La fonction abs

La fonction `abs` renvoie la valeur absolue d'un nombre, c'est-à-dire la valeur de ce nombre sans son signe. Par exemple, la valeur absolue de 10 vaut 10 et la valeur absolue de -10 vaut aussi 10.

Pour utiliser la fonction `abs`, appelle-la simplement avec un nombre ou une variable en paramètre, comme ceci :

```
>>> print(abs(10))
10
>>> print(abs(-10))
10
```

La fonction `abs` sert par exemple à calculer la quantité absolue de mouvement d'un personnage de jeu, quelle que soit la direction dans laquelle il se déplace. Si le personnage fait trois pas sur sa droite (3 positif), puis dix pas sur sa gauche (10 négatif ou -10), et si nous ne tenons pas compte de la direction (positive ou négative), les valeurs absolues de ces nombres sont 3 et 10. Ceci peut intervenir dans un jeu de plateau où tu lances deux dés, puis déplaces le personnage d'un nombre maximal de pas dans n'importe quelle direction, en fonction du total des points des deux dés. Ensuite, si tu stockes le nombre de pas dans une variable, tu seras en mesure de déterminer si le personnage se déplace avec l'extrait de code suivant. Tu afficheras par exemple des informations quand le joueur décide de se déplacer. Ici, nous affichons simplement « Le personnage se déplace » :



```
>>> pas = -3
>>> if abs(pas) > 0:
    print('Le personnage se déplace')
```

Si nous n'utilisions pas `abs`, l'instruction `if` deviendrait ceci :

```
>>> pas = -3
>>> if pas < 0 or pas > 0:
    print('Le personnage se déplace')
```

Tu constates que l'utilisation d'`abs` raccourcit un peu la condition de l'instruction `if` et en facilite la compréhension.

La fonction `bool`

Le nom `bool` est l'abréviation de **booléen**, le terme que les programmeurs utilisent pour décrire un type de donnée qui ne peut avoir qu'une valeur parmi deux possibles, généralement vrai ou faux.

La fonction `bool` accepte un paramètre et renvoie `True` (vrai) ou `False` (faux), selon sa valeur. Appliquée aux nombres, `bool` renvoie `False` quand le nombre est égal à `0` et `True` pour n'importe quel autre nombre. Voici ce que cela donne pour différents nombres :

```
>>> print(bool(0))
False
>>> print(bool(1))
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

Appliquée à d'autres valeurs, comme une chaîne de caractères, `bool` renvoie `False` quand la chaîne ne contient aucune valeur, autrement dit quand elle vaut `None` ou qu'elle est vide (''), sinon elle renvoie `True`, comme ceci :

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
>>> print(bool(' '))
True
>>> print(bool("Qu'est-ce qui est rose et qui fait peur ? Un cochon ↴
karatéka."))
True
```

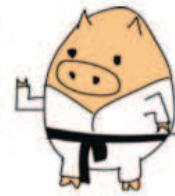
La fonction `bool` renvoie aussi `False` pour les listes, les tuples et les dictionnaires qui ne contiennent aucune valeur, et `True` dans les autres cas :

```
>>> une_liste_stupide = []
>>> print(bool(une_liste_stupide))
False
>>> une_liste_stupide = ['s', 't', 'u', 'p', 'i', 'd', 'e']
>>> print(bool(une_liste_stupide))
True
```

La fonction `bool` sert notamment à déterminer si une valeur a été définie ou non. Par exemple, si nous demandons à des gens d'utiliser notre programme pour entrer l'année de leur naissance, une instruction `if` peut utiliser `bool` pour vérifier qu'ils ont entré une valeur :

```
>>> annee = input('Année de naissance :')
Année de naissance :
>>> if not bool(annee.rstrip()):
    print("Vous devez entrer une valeur pour l'année de naissance")
Vous devez entrer une valeur pour l'année de naissance
```

La première ligne de cet exemple utilise `input` pour mémoriser dans la variable `annee` ce que l'utilisateur entre au clavier. Une pression sur `Entrée` à la ligne suivante, sans rien taper d'autre, stocke la valeur de la touche `Entrée` dans la variable. Au chapitre 7, nous avions utilisé `sys.stdin.readline()` pour aboutir au même résultat.



À la ligne suivante, l'instruction `if` vérifie la valeur booléenne, après avoir appliqué la fonction `rstrip`, qui supprime tous les espaces et tous les caractères `Entrée` de la fin de la chaîne. Comme l'utilisateur n'a saisi aucune donnée, la fonction `bool` renvoie `False`. Du fait que l'instruction `if` emploie le mot-clé `not` (« pas » ou « non »), cela revient à dire « faire ceci si la fonction ne renvoie pas vrai » et donc le code affiche le message à la ligne suivante.

La fonction `dir`

La fonction `dir` (abrégé de *directory*, c'est-à-dire répertoire en anglais) retourne des informations à propos de n'importe quelle valeur. À la base, elle indique en ordre alphabétique les fonctions qui peuvent être utilisées avec la valeur du paramètre.

Ainsi, pour afficher les fonctions disponibles pour une valeur de liste, tape ceci :

```
>>> dir(['une', 'courte', 'liste'])
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__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', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

La fonction `dir` fonctionne sur à peu près n'importe quoi, notamment les chaînes, les nombres, les fonctions, les modules, les objets et les classes. Parfois cependant, les informations qu'elle renvoie ne sont pas très utiles. Ainsi, si tu appelles `dir` sur le nombre `1`, elle affiche un certain nombre de fonctions spéciales, celles qui débutent et se terminent par des caractères de soulignement, que Python utilise lui-même et qui n'ont pas vraiment d'utilité pour nous ; donc tu peux généralement les ignorer.

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
 '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
 '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
 '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__',
 '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__',
 '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__',
 '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
 '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
 'real', 'to_bytes']
```

La fonction s'avère utile quand tu as une variable et que tu veux rapidement savoir ce que tu peux en faire. Par exemple, exécute `dir` avec une variable `popcorn` qui contient une valeur de chaîne de caractères et tu obtiens la liste des fonctions fournies par la classe `string`, car toutes les chaînes de caractères sont membres de cette classe :

```
>>> popcorn = "J'adore le popcorn !"
>>> dir(popcorn)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__',
 '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition']
```

```
tion', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

À partir de là, utilise la fonction `help` pour obtenir une courte description de n'importe quelle fonction de cette liste. Ainsi, pour en savoir plus à propos de la fonction `upper` de la variable, exécute :

```
>>> help(popcorn.upper)
Help on built-in function upper:

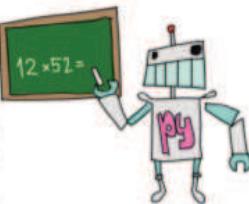
upper(...) method of builtins.str instance
S.upper() -> str

Return a copy of S converted to uppercase.
```

Tout cela est un peu nébuleux, d'autant que tout est en anglais, évidemment ! Donc regardons cela plus en détail. L'ellipse (...) signifie que `upper` est une fonction intégrée de la classe et que, dans ce cas-ci, elle ne prend aucun paramètre. La flèche (->) de la ligne suivante indique que la fonction renvoie une chaîne (`str`). La dernière ligne donne une brève description de ce que fait la fonction, c'est-à-dire qu'elle renvoie une copie de la chaîne `S` convertie en lettres capitales.

La fonction `eval`

Abrégé (en anglais) d'évaluer, la fonction `eval` prend une chaîne de caractères en paramètre et l'exécute comme si c'était une expression en Python. Ainsi, `eval('print("Bravo")')` exécute en fait l'instruction `print("Bravo")`.



La fonction `eval` ne fonctionne qu'avec des expressions simples, comme la suivante :

```
>>> eval('10*5')
50
```

Les expressions qui s'étalent sur plusieurs lignes, telles les instructions `if`, ne peuvent généralement pas être évaluées, comme dans cet exemple :

```
>>> eval('''if True:
...     print("Ceci ne fonctionne pas du tout.")''')
Traceback (most recent call last):
```

```
Fайл "<pyshell#6>", line 2, in <module>
    print("Ceci ne fonctionne pas du tout."'''')
Fайл "<string>", line 1
    if True:
        ^
SyntaxError: invalid syntax
```

La fonction `eval` sert souvent à convertir une entrée de l'utilisateur en expressions Python. Ainsi, tu peux écrire un programme de calculatrice simple, qui lit des équations entrées en Python et les calcule, c'est-à-dire les évalue, pour fournir les réponses.

Comme l'entrée utilisateur est lue sous forme de chaîne, Python doit la convertir en nombres et en opérateurs avant d'effectuer les calculs, mais la fonction `eval` évite et facilite fortement cette conversion :

```
>>> ton_calcul = input('Entre un calcul : ')
Entre un calcul : 12*52
>>> eval(ton_calcul)
624
```

Dans cet exemple, nous utilisons `input` pour lire ce que l'utilisateur entre et le verser dans la variable `ton_calcul`. À la ligne suivante, l'utilisateur saisit l'expression `12*52`, qui correspond par exemple à son âge multiplié par le nombre de semaines dans une année. Nous utilisons `eval` pour effectuer le calcul et le résultat s'affiche à la dernière ligne.

La fonction `exec`

La fonction `exec` fonctionne comme `eval`, mais elle accepte des expressions plus complexes. La grande différence entre les deux se situe dans le fait qu'`eval` renvoie une valeur, qu'il est possible d'affecter à une variable, tandis qu'`exec` ne renvoie aucune valeur. Voici un exemple :

```
>>> mon_petit_programme = '''print('sandwich')
print('au jambon')'''
>>> exec(mon_petit_programme)
sandwich
au jambon
```

Aux deux premières lignes, nous créons une variable avec une chaîne multiligne qui contient deux instructions `print`. À la ligne suivante, `exec` exécute cette chaîne.

La fonction `exec` permet d'exécuter des mini-programmes, que Python peut par exemple lire dans un fichier, ce qui revient en fait à créer un programme qui exécute des programmes ! Cela s'avère parfois utile lors de l'écriture des applications longues et complexes. Imagine un jeu de duel entre deux robots qui évoluent à l'écran et qui essaient de s'attaquer mutuellement. Les joueurs pourraient fournir les instructions à leurs robots sous la forme de mini-programmes (ou scripts) en Python. Le jeu des robots en duel lirait ces petits scripts et les exécuterait avec `exec`.

La fonction `float`

La fonction `float` convertit une chaîne en un « nombre à virgule flottante » (*floating point*), c'est-à-dire un nombre qui possède un point décimal en Python. Cela s'appelle aussi un « nombre réel », par opposition à un « nombre entier » ou simplement entier (*integer*). Ainsi, si `10` est un entier, `10.0`, `10.1` et `10.253` sont tous des nombres à virgule flottante. Il faut bien comprendre que le langage Python est en anglais et qu'un nombre tel que `10,253` en mathématiques s'écrit en Python avec un point au lieu de la virgule, soit `10.253`.

Pour convertir une chaîne en nombre à virgule flottante, appelle `float` :



La chaîne peut aussi contenir un point décimal (mais pas de virgule décimale) :

```
>>> float('123.456789')
123.456789
```

Utilise `float` pour convertir en valeurs appropriées des valeurs entrées par l'utilisateur dans un programme, ce qui s'avère particulièrement utile lorsque tu veux comparer la valeur entrée par une personne avec d'autres valeurs. Ainsi, pour vérifier que l'âge d'une personne est au-dessus d'un nombre déterminé, tu pourrais écrire ceci :

```
>>> ton_age = input('Entre ton âge : ')
Entre ton âge : 20
>>> age = float(ton_age)
>>> if age > 13:
    print('Tu es trop vieux de %s années.' % (age - 13))
Tu es trop vieux de 7.0 années.
```

La fonction int

La fonction `int` convertit une chaîne ou un nombre en un nombre entier (*integer*), ce qui signifie en réalité que tout ce qui se situe à droite du point décimal éventuel est éliminé. Ainsi, pour convertir un nombre à virgule flottante en un entier, écris :

```
>>> int(123.456)  
123
```

Et pour convertir une chaîne en un entier :

```
>>> int('123')  
123
```

En revanche, si tu essaies de convertir en un entier une chaîne qui contient un nombre à virgule flottante, tu provoques une erreur de valeur. Dans l'exemple qui suit, nous essayons de convertir le nombre `123.456` présenté sous forme de chaîne à l'aide la fonction `int` :

```
>>> int('123.456')  
Traceback (most recent call last):  
  File "<pyshell>", line 1, in <module>  
    int('123.456')  
ValueError: invalid literal for int() with base 10: '123.456'
```

Tu constates que le résultat produit est effectivement une erreur de valeur, `ValueError`.

La fonction len

La fonction `len` retourne la longueur d'un objet ou, dans le cas d'une chaîne, le nombre de caractères dans la chaîne. Par exemple, pour connaître la longueur de la chaîne `Ceci est une chaîne de test`, écris :



```
>>> len('Ceci est une chaîne de test')  
27
```

Utilisée avec une liste ou un tuple, `len` renvoie le nombre d'éléments de la liste ou du tuple :

```
>>> liste_creatures = ['licorne', 'cyclope', 'fée', 'elfe', 'dragon', «  
  'troll']  
>>> print(len(liste_creatures))  
6
```

Avec un dictionnaire, `len` retourne aussi le nombre d'éléments :

```
>>> dict_enemis = {'Batman' : 'Joker', 'Superman' : 'Lex Luthor', 'Spiderman' : 'Lutin vert'}
>>> print(len(dict_enemis))
3
```

La fonction `len` est particulièrement utile avec des boucles. Ainsi, pour afficher l'indice des éléments dans une liste, nous pouvons écrire :

```
>>> fruit = ['pomme', 'banane', 'clémentine', 'fruit de la passion']
❶ >>> longueur = len(fruit)
❷ >>> for x in range(0, longueur):
❸ >>>     print("Le fruit à l'indice %s est %s" % (x, fruit[x]))
Le fruit à l'indice 0 est pomme
Le fruit à l'indice 1 est banane
Le fruit à l'indice 2 est clémentine
Le fruit à l'indice 3 est fruit de la passion
```

Ici, nous stockons la taille de la liste dans la variable `longueur` ❶, puis nous utilisons cette variable dans la fonction `range` pour créer notre boucle ❷. Comme nous parcourons en boucle chaque élément de la liste, nous affichons ❸ un message qui montre l'indice et la valeur de l'élément. Bien utilisée, la fonction `len` permet aussi d'afficher un élément sur deux ou trois d'une liste de chaînes.

Les fonctions `max` et `min`

La fonction `max` renvoie le plus grand élément d'une liste, d'un tuple ou d'une chaîne. Par exemple, voici son utilisation sur une liste de nombres :

```
>>> nombres = [5, 4, 10, 30, 22]
>>> print(max(nombres))
30
```

Une chaîne avec ses caractères séparés par des virgules ou des espaces fonctionne également :

```
>>> chaines = 'c,h,a,i,n,e,C,H,A,I,N,E'
>>> print(max(chaines))
N
```



L'exemple montre que les lettres sont triées en ordre alphabétique, avec d'abord les lettres capitales, puis les lettres minuscules, de sorte que n vaut plus que N .

Il n'est pas indispensable d'utiliser des listes, des tuples ou des chaînes car il est possible d'appeler `max` directement et d'entrer les éléments à comparer entre les parenthèses, comme des paramètres :

```
>>> print(max(10, 300, 450, 50, 90))
450
```

La fonction `min` fonctionne comme `max`, mais elle renvoie le plus petit élément de la liste, du tuple ou de la chaîne. Voici le même exemple de liste, auquel nous appliquons `min` :

```
>>> nombres = [5, 4, 10, 30, 22]
>>> print(min(nombres))
4
```

Admettons que tu joues à un jeu de devinette avec une équipe de quatre joueurs, dont chacun doit deviner un nombre plus petit que celui que tu as choisi en secret. Si un des joueurs propose un nombre plus grand que le tien, tous les joueurs perdent, mais s'ils proposent tous un nombre plus petit que le tien, ils gagnent. La fonction `max` indique si toutes les propositions sont plus petites, comme ceci :

```
>>> devine_ce_nombre = 61
>>> propositions = [12, 15, 70, 45]
>>> if max(propositions) >= devine_ce_nombre:
    print('Boum ! Tout le monde perd !')
else:
    print('Tu gagnes')
Boum ! Tout le monde perd !
```

Dans cet exemple, nous stockons le nombre à deviner dans la variable `devine_ce_nombre`. Les membres de l'équipe proposent des valeurs qui viennent se ranger dans la liste `propositions`. L'instruction `if` compare la proposition maximale au nombre à deviner et, si au moins un joueur dépasse ce nombre, nous affichons le message `Boum ! Tout le monde perd !`.

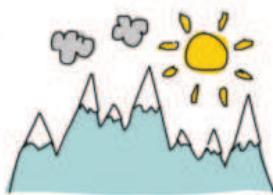
La fonction `range`

Ainsi que nous l'avons vu auparavant, la fonction `range`, qui signifie plage (de valeurs), sert surtout dans des boucles, pour répéter un certain nombre de fois une portion de code. Les deux premiers para-

mètres demandés par `range` sont appelés le « début » et l'« arrêt ». Cette fonction a été mise en œuvre dans l'exemple précédent qui employait `len` au sein d'une boucle.

Les nombres que `range` génère commencent avec la valeur donnée en premier paramètre et se terminent avec le nombre inférieur d'une unité par rapport au second paramètre. L'exemple suivant montre ce qu'il advient quand nous affichons les nombres créés par `range` entre 0 et 5 (non compris) :

```
>>> for x in range(0, 5):
    print(x)
0
1
2
3
4
```



La fonction `range` renvoie en réalité un objet spécial, appelé « itérateur », qui répète une action un certain nombre de fois. Dans ce cas-ci, elle renvoie le numéro plus grand suivant à chaque appel.

Il est possible de convertir l'itérateur en une liste, grâce à la fonction `list`. Si nous affichons la valeur renvoyée lors de l'appel de `range`, nous pouvons voir également les nombres qu'elle contient :

```
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

La fonction `range` accepte aussi un troisième paramètre, appelé « pas ». Si la valeur du pas n'est pas indiquée, le nombre `1` est considéré par défaut. Si nous choisissons la valeur `2` pour le pas, nous aurions le résultat suivant :

```
>>> comptage_par_deux = list(range(0, 30, 2))
>>> print(comptage_par_deux)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Chaque nombre de la liste augmente de deux par rapport au nombre précédent et la liste se termine au nombre `28`, qui correspond au nombre inférieur de `2` par rapport à `30`. Les pas négatifs sont aussi possibles :

```
>>> decompte_par_deux = list(range(40, 10, -2))
>>> print(decompte_par_deux)
[40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12]
```

La fonction sum

La fonction `sum` additionne les éléments d'une liste et en renvoie le total. Voici un exemple :

```
>>> ma_liste_de_nombres = list(range(0, 500, 50))
>>> print(ma_liste_de_nombres)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
>>> print(sum(ma_liste_de_nombres))
2250
```

La première ligne crée une liste de nombres compris entre 0 et 500 par pas de 50, à l'aide de `range` et de `list`. La ligne suivante affiche la liste pour en voir le contenu. Enfin, le passage de `ma_liste_de_nombres` à la fonction `sum` additionne tous les éléments de cette liste, pour donner le total de 2250.

Manipuler des fichiers

En Python, les fichiers sont semblables aux autres fichiers de l'ordinateur, comme les documents, les images, la musique, les jeux et ainsi de suite. En pratique, tout est enregistré dans l'ordinateur sous cette forme.

Nous allons voir comment ouvrir et manipuler des fichiers en Python à l'aide de la fonction intégrée `open`, mais au préalable, nous devons créer un fichier pour pouvoir jouer avec lui.

Créer un fichier de test

Livrons-nous à quelques expériences à partir d'un fichier que nous allons nommer `test.txt`. Suis les étapes qui correspondent à ton système d'exploitation.

Créer un fichier sous Windows

Si tu utilises Windows, suis ces étapes pour créer le fichier `test.txt`.

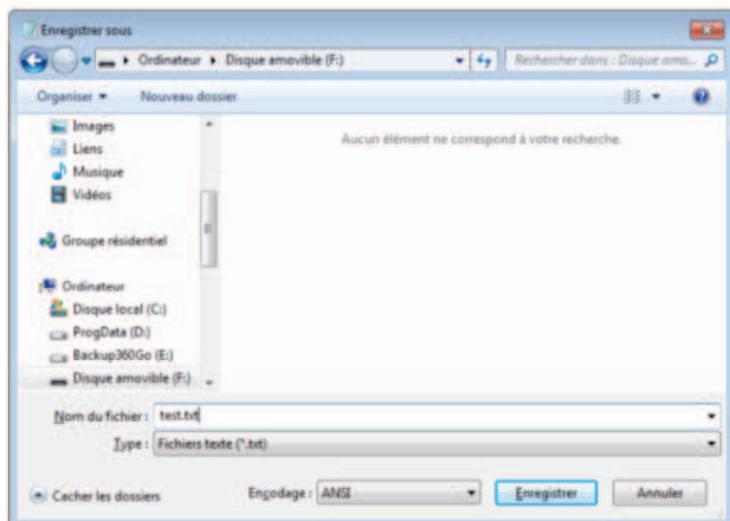
1. Ouvre le menu **Démarrer>Tous les programmes>Accessoires>Bloc-notes**.
2. Tape quelques lignes dans le fichier vide.
3. Clique sur **Ficher>Enregistrer**.
4. Quand la boîte de dialogue **Enregistrer sous** s'affiche, sélectionne le dossier qui contiendra le fichier. Choisis de préférence un disque amovible (clé mémoire USB) où tu peux enregistrer ce fichier.

Clique sur **Ordinateur** dans la liste de gauche, puis double-clique sur **Disque amovible (F:)**, par exemple.

5. Entre **test.txt** dans la zone **Nom du fichier** au bas de la boîte de dialogue.
6. Clique sur le bouton **Enregistrer**.

NOTE

*Le plus facile consiste à utiliser une clé mémoire USB car celle-ci apparaît dans la boîte de dialogue **Enregistrer sous**, sous la rubrique **Ordinateur**. Tu verras parmi ces pages F: mais chez toi, ce peut-être D: ou une autre lettre.*

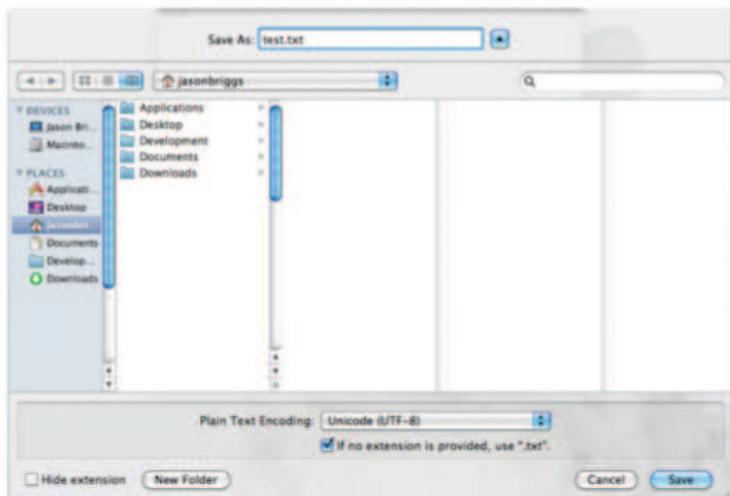


Créer un fichier sous OS X

Si tu utilises un Mac, suis ces étapes pour créer **test.txt**.

1. Clique sur l'icône **Spotlight** dans la barre de menus du haut de l'écran.
2. Tape **TextEdit** dans la zone de recherche qui apparaît.
3. **TextEdit** apparaît dans la section **Applications**. Clique dessus pour ouvrir l'éditeur. **TextEdit** est aussi accessible dans le dossier **Applications** du Finder.

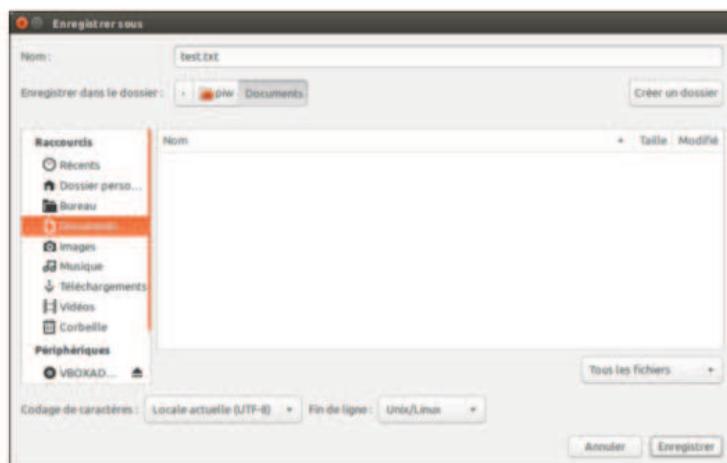
4. Entre quelques lignes de texte dans le fichier vide.
5. Clique sur **Format>Make Plain Text**.
6. Clique sur **Fichier>Enregistrer**.
7. Dans la boîte de dialogue **Enregistrer sous**, entre **test.txt**.
8. Dans la liste des emplacements, clique sur ton nom d'utilisateur, le nom sous lequel tu t'es connecté ou le nom du propriétaire de l'ordinateur.
9. Enfin, clique sur **Enregistrer**.



Créer un fichier sous Linux et Ubuntu

Si tu utilises Linux, notamment Ubuntu, suis ces étapes pour créer le fichier **test.txt**.

1. Ouvre ton Éditeur de texte (ou gedit), accessible, soit dans le menu **Applications**, soit dans le tableau de bord.
2. Tape quelques lignes de texte dans l'éditeur de texte.
3. Clique sur **Fichier>Enregistrer**.
4. Dans la zone de texte **Nom**, entre **test.txt** comme nom de fichier. Ton dossier personnel est peut-être déjà sélectionné dans la zone nommée **Enregistrer dans le dossier**. Sinon, clique dessus dans la liste des emplacements, puis sur le dossier **Documents**.
5. Clique sur le bouton **Enregistrer**.



Ouvrir un fichier en Python

La fonction intégrée `open` ouvre un fichier dans le shell de Python et affiche son contenu. La manière d'indiquer à la fonction le fichier à ouvrir dépend du système d'exploitation. Examine l'exemple pour un fichier Windows, puis consulte la section spécifique à OS X ou à Linux si tu utilises un de ces systèmes.

Ouvrir un fichier sous Windows

Si tu utilises Windows, entre le code suivant pour ouvrir `test.txt` :

```
>>> fichier_test = open('f:\\\\test.txt')
>>> texte = fichier_test.read()
>>> print(texte)
Il était une fois un garçon nommé Marcelo
Qui rêvait qu'il mangeait une guimauve
Il se réveilla en sursaut
Comme son lit s'est effondré
Il s'est découvert bien enrobé
```

À la première ligne, la fonction `open` renvoie un objet fichier doté de fonctions pour l'exploiter. Le paramètre de la fonction `open` est une chaîne qui indique à Python où trouver le fichier. Si tu utilises Windows, tu as enregistré le fichier selon nos conseils dans un lecteur amovible de type clé mémoire USB et la lettre de lecteur dépend

de l'emplacement sous Windows, soit `F:` ici. Le chemin complet du fichier est indiqué sous la forme `f:\\test.txt`.

Les deux barres obliques inverses (`\\\`) dans le nom de fichier sous Windows indiquent à Python que la deuxième barre oblique inverse est le caractère `\` et non une commande. Au chapitre 3, nous avons vu que les barres obliques inverses seules ont une signification spéciale en Python, en particulier dans les chaînes. Nous affectons l'objet fichier à la variable `fichier_test`.

À la deuxième ligne, la fonction `read` fournie par l'objet fichier lit le contenu du fichier et le stocke dans la variable `texte`. La troisième ligne affiche le contenu du fichier.

Ouvrir un fichier sous OS X

Sous l'OS X du Mac, l'emplacement du fichier à ouvrir diffère de celui indiqué à la première ligne du code de l'exemple sous Windows. Indique le nom d'utilisateur sur lequel tu as cliqué pour enregistrer le fichier texte dans la chaîne. Si ton nom d'utilisateur est `pierre`, par exemple, le paramètre d'`open` ressemble à ceci :

```
>>> fichier_test = open('/Users/pierre/test.txt')
```

Ouvrir un fichier sous Linux (Ubuntu)

Sous Ubuntu et plus généralement sous Linux, l'emplacement du fichier est différent de celui de la première ligne de l'exemple de code pour Windows. Indique le nom d'utilisateur sur lequel tu as cliqué pour enregistrer le fichier de test. Si ton nom d'utilisateur est `pierre`, par exemple, le paramètre d'`open` ressemble à ceci :

```
>>> fichier_test = open('/home/pierre/test.txt')
```

Écrire dans des fichiers

L'objet fichier renvoyé par `open` possède d'autres fonctions que `read`. Nous pouvons créer un fichier vide à l'aide du deuxième paramètre :

```
>>> fichier_test = open('f:\\monfichier.txt', 'w')
```

Le paramètre '`w`' (*write*) dit à Python que nous voulons ouvrir l'objet fichier en mode écriture et non plus en mode lecture (*read*). Ensuite seulement, nous ajoutons des informations dans ce nouveau fichier, à l'aide de la fonction `write` :

```
>>> fichier_test = open('f:\\monfichier.txt', 'w')
>>> fichier_test.write('Ceci est mon fichier de test')
```

Enfin, lorsque nous avons fini d'écrire, nous devons l'indiquer à Python à l'aide de la fonction `close`, qui ferme le fichier :

```
>>> fichier_test = open('f:\\monfichier.txt', 'w')
>>> fichier_test.write('Comment fait-on aboyer un chat ?')
>>> fichier_test.write(' On lui donne une soucoupe de lait')
>>> fichier_test.write(' et il la boit !')
>>> fichier_test.close()
```

À présent, si tu ouvres le fichier avec l'éditeur de texte, tu peux voir le texte de la blague. Tu peux aussi demander à Python de le relire :

```
>>> fichier_test = open('f:\\monfichier.txt')
>>> print(fichier_test.read())
Comment fait-on aboyer un chat ? On lui donne une soucoupe de lait et
il la boit !
```

Ce que tu as appris

Dans ce chapitre, tu as découvert quelques fonctions intégrées de Python, telles que `float` et `int`, qui convertissent des nombres à virgule flottante en nombres entiers et vice versa. Tu as vu aussi l'intérêt d'utiliser la fonction `len` dans des boucles. Tu as enfin appris à ouvrir des fichiers pour les lire ou pour y écrire.

Puzzles de programmation

Essaie les exemples suivants pour t'entraîner à utiliser quelques fonctions intégrées de Python. Les réponses sont disponibles sur le site d'accompagnement du livre.

1. Code mystère

Quel est le résultat de l'exécution du code suivant ? Essaie de le deviner, puis seulement, exécute le code pour vérifier si tu as raison.

```
>>> a = abs(10) + abs(-10)
>>> print(a)
>>> b = abs(-10) + -10
>>> print(b)
```

2. Message caché

Cherche dans la documentation fournie par les fonctions `dir` et `help` des informations sur la manière de découper une chaîne en mots, puis crée un petit programme qui affiche un mot sur deux de la chaîne suivante, en commençant par le premier mot (`ceci`) :

"ceci si n'est tu pas peux une lire très ceci bonne alors manière c'est de que cacher tu un t'es message trompé"

CONSEIL

Un bon dictionnaire en ligne ou sur papier est indispensable pour programmer en Python mais tu verras que tu assimileras très rapidement l'anglais technique, parce que les mêmes mots reviennent régulièrement et, à force de rechercher dans la documentation, tu trouveras vite des points de repère. Pour l'heure, en anglais, découper se dit split.

3. Copier un fichier

Crée un petit programme en Python pour copier un fichier. Pour tester ton programme, ajoute les lignes nécessaires pour afficher à l'écran le contenu de la copie du fichier.

CONSEIL

Ouvre le fichier à copier, lis-en le contenu, puis crée un nouveau fichier et écris-y le contenu du précédent. N'oublie pas de fermer les deux fichiers après lecture et écriture.



10 MODULES UTILES DE PYTHON

Au chapitre 7, nous avons vu qu'un **module** Python contient un ensemble de **fonctions**. Au chapitre 8, nous avons découvert que le module `turtle` comporte une classe importante, `Pen` ; un module peut donc aussi comprendre des **classes**. En fait, un module Python contient tout cela, ainsi que des variables, qu'il regroupe par type d'utilisation. Le module `turtle`, par exemple, que nous avons déjà utilisé dans deux chapitres, rassemble des fonctions et des classes afin de concevoir un **canevas** pour que la tortue dessine à l'écran.

Dès que tu as importé un module dans un programme, tout son contenu devient utilisable. Avec l'importation du module `turtle` au chapitre 4, nous avons eu accès à la classe `Pen`, exploitée pour créer un objet représentant le canevas de dessin de la tortue :

```
>>> import turtle  
>>> t = turtle.Pen()
```

Python possède la panoplie complète de modules pour accomplir toutes sortes de tâches différentes et, dans ce chapitre, nous allons examiner les plus utiles, pour essayer quelques-unes de leurs fonctions.

Créer des copies avec le module `copy`

Le module `copy` contient des fonctions pour réaliser des copies d'objets. Lors de l'écriture de programme, il est habituel de réaliser des objets. Sache qu'il s'avère parfois utile, en particulier lorsque le processus de création exige plusieurs étapes, de réaliser une copie de cet objet, puis de s'en servir pour en créer un nouveau.

Prenons l'exemple d'une classe `Animal` avec une fonction `_init_` qui demande les paramètres `espece`, `nombre_de_pattes` et `couleur`.



```
>>> class Animal:  
    def __init__(self, espece, nombre_de_pattes, couleur):  
        self.espece = espece  
        self.nombre_de_pattes = nombre_de_pattes  
        self.couleur = couleur
```

Créons un nouvel objet de la classe `Animal`, de l'espèce hippogriffe, avec six pattes, de couleur rose et nommé `hector` :

```
>>> hector = Animal('hippogriffe', 6, 'rose')
```

Imaginons que nous voulions une horde d'hippogriffes roses à six pattes. Nous pourrions répéter le code précédent encore et encore, ou faire appel à la fonction `copy()`, qui se situe dans le module `copy` :

```
>>> import copy  
>>> hector = Animal('hippogriffe', 6, 'rose')  
>>> henriette = copy.copy(hector)  
>>> print(hector.espece)  
hippogriffe  
>>> print(henriette.espece)  
hippogriffe
```
