Pensez en Python

Comment maîtriser la science de l'informatique

Par Allen B. Downey - Mishulyna (traducteur) - Laurent Rosenfeld (traducteur)

Date de publication : 8 février 2016

Dernière mise à jour : 19 février 2016

Ce livre n'est pas fait pour vous enseigner le langage de programmation Python. Ou du moins pas en priorité. Non, ce livre vise tout d'abord à vous apprendre à programmer, à penser comme un informaticien et à maîtriser la science de l'informatique. Au passage, vous en apprendrez aussi beaucoup sur le langage Python et serez en mesure de l'utiliser pour des tâches réelles assez complexes, mais ce n'est pas le but essentiel.

Ce livre est notamment destiné à l'informaticien débutant qui veut apprendre la programmation. Il ne nécessite donc pas de connaissances préalables dans le domaine du développement informatique et vous pouvez donc vous lancer même si vous n'avez jamais écrit le moindre programme. Cependant, même si vous savez déjà programmer, que ce soit en Python ou dans un ou même plusieurs autres langages, il est probable que vous appreniez beaucoup de choses nouvelles à la lecture de ce livre et, surtout, que vous appreniez à penser comme un informaticien et à maîtriser vraiment la science et l'art de la programmation informatique.

Commentez

Introduction....7
L'étrange histoire de ce livre....7
Remerciements....8
Liste des contributeurs....8
1 - Le chemin du programme....11
1-1 - Qu'est-ce qu'un programme ?....12
1-2 - Exécuter Python....12
1-3 - Le premier programme....13
1-4 - Opérateurs arithmétiques....13
1-5 - Valeurs et types....14
1-6 - Langages naturels et formels....15
1-7 - Débogage....16
1-8 - Glossaire....16
1-9 - Exercices....17
2 - Variables, expressions et instructions....18
2-1 - Instructions d'affectation....18
2-2 - Noms de variables....18
2-3 - Expressions et instructions....19
2-4 - Mode de script....19
2-5 - Ordre des opérations....21
2-6 - Opérations sur des chaînes....21
2-7 - Commentaires....21
2-8 - Débogage....22
2-9 - Glossaire....22
2-10 - Exercices....23
3 - Fonctions....24
3-1 - Appels de fonctions....24
3-2 - Fonctions mathématiques....24
3-3 - Composition....25
3-4 - Créer de nouvelles fonctions....26
3-5 - Définitions et utilisation....27
3-6 - Flux d'exécution....27
3-7 - Paramètres et arguments....28
3-8 - Les variables et les paramètres sont locaux....29
3-9 - Diagrammes de pile....29
3-10 - Fonctions productives et fonctions vides....30
3-11 - Pourquoi des fonctions ?....31
3-12 - Débogage....31
3-13 - Glossaire....31
3-14 - Exercices....32
4 - Étude de cas : conception d'une interface....33
4-1 - Le module turtle....34
4-2 - Répétition simple....35
4-3 - Exercices....36
4-4 - Encapsulation....36
4-5 - Généralisation....36
4-6 - Conception d'une interface....37
4-7 - Réusinage....38
4-8 - Un plan de développement....39
4-9 - docstring....39
4-10 - Débogage....40
4-11 - Glossaire....40
4-12 - Exercices....40
5 - Structures conditionnelles et récursion....42
5-1 - Division entière et modulo....42
5-2 - Expressions booléennes....42
5-3 - Opérateurs logiques....43
5-4 - Exécution conditionnelle....43

5-5 - Exécution alternative.....44
5-6 - Conditions enchaînées.....44
5-7 - Conditions imbriquées.....44
5-8 - Récursion.....45
5-9 - Diagrammes de pile pour les fonctions récursives.....46
5-10 - Récursion infinie.....47
5-11 - Saisie au clavier.....47
5-12 - Débogage.....48
5-13 - Glossaire.....49
5-14 - Exercices.....50
6 - Fonctions productives.....52
6-1 - Valeurs de retour.....52
6-2 - Développement incrémental.....53
6-3 - Composition.....55
6-4 - Fonctions booléennes.....55
6-5 - Plus de récursivité.....56
6-6 - Acte de foi.....58
6-7 - Encore un exemple.....58
6-8 - Vérifier les types.....59
6-9 - Débogage.....60
6-10 - Glossaire.....60
6-11 - Exercices.....61
7 - Itération.....62
7-1 - Réaffectation.....62
7-2 - Mettre à jour les variables.....63
7-3 - L'instruction while.....64
7-4 - break.....65
7-5 - Racines carrées.....65
7-6 - Algorithmes.....67
7-7 - Débogage.....67
7-8 - Glossaire.....68
7-9 - Exercices.....68
8 - Chaînes de caractères.....69
8-1 - Une chaîne de caractères est une séquence.....69
8-2 - len.....70
8-3 - Parcours avec une boucle for.....70
8-4 - Tranches de chaînes de caractères.....71
8-5 - Les chaînes de caractères sont immuables.....72
8-6 - Recherche.....72
8-7 - Boucler et compter.....73
8-8 - Méthodes de chaînes de caractères.....73
8-9 - L'opérateur in.....74
8-10 - Comparaison de chaînes de caractères.....74
8-11 - Débogage.....75
8-12 - Glossaire.....76
8-13 - Exercices.....77
9 - Étude de cas : jouer avec les mots.....78
9-1 - Lire des listes de mots.....79
9-2 - Exercices.....80
9-3 - Recherche.....81
9-4 - Boucler avec des indices.....82
9-5 - Débogage.....83
9-6 - Glossaire.....83
9-7 - Exercices.....83
10 - Listes.....84
10-1 - Une liste est une séquence.....85
10-2 - Les listes sont modifiables.....85
10-3 - Parcourir une liste.....86

10-4 - Opérations sur listes....87
10-5 - Tranches de liste....87
10-6 - Méthodes de listes....88
10-7 - Mapper, filtrer et réduire....88
10-8 - Supprimer des éléments....89
10-9 - Listes et chaînes de caractères....90
10-10 - Objets et valeurs....91
10-11 - Aliasing....91
10-12 - Arguments de type liste....92
10-13 - Débogage....93
10-14 - Glossaire....94
10-15 - Exercices....95
11 - Dictionnaires....97
11-1 - Un dictionnaire est un mappage....97
11-2 - Un dictionnaire comme une collection de compteurs....99
11-3 - Boucles et dictionnaires....100
11-4 - Recherche inversée....100
11-5 - Dictionnaires et listes....101
11-6 - Mémos....103
11-7 - Variables globales....104
11-8 - Débogage....105
11-9 - Glossaire....106
11-10 - Exercices....106
12 - Tuples....107
12-1 - Les tuples sont immuables....107
12-2 - Affectation de tuple....108
12-3 - Tuples comme valeurs de retour....109
12-4 - Arguments tuples à longueur variable....110
12-5 - Listes et tuples....110
12-6 - Dictionnaires et tuples....112
12-7 - Séquences de séquences....113
12-8 - Débogage....114
12-9 - Glossaire....114
12-10 - Exercices....115
13 - Étude de cas : le choix des structures de données....116
13-1 - Analyse de la fréquence des mots....116
13-2 - Nombres aléatoires....117
13-3 - Histogramme de mots....118
13-4 - Les mots les plus fréquents....120
13-5 - Paramètres optionnels....121
13-6 - Soustraction de dictionnaire....121
13-7 - Mots aléatoires....122
13-8 - Analyse de Markov....123
13-9 - Structures de données....124
13-10 - Débogage....125
13-11 - Glossaire....126
13-12 - Exercices....126
14 - Fichiers....127
14-1 - Persistance....127
14-2 - Lecture et écriture....128
14-3 - L'opérateur de formatage....128
14-4 - Noms de fichiers et chemins....129
14-5 - Intercepter les exceptions....130
14-6 - Bases de données....131
14-7 - Sérialiser les données avec pickle....132
14-8 - Pipes....132
14-9 - Écrire des modules....133
14-10 - Débogage....134

14-11 - Glossaire..... 135
14-12 - Exercices..... 135
15 - Classes et objets..... 136
15-1 - Types définis par le programmeur..... 136
15-2 - Attributs..... 137
15-3 - Rectangles..... 138
15-4 - Instances comme valeurs de retour..... 139
15-5 - Les objets sont modifiables..... 139
15-6 - Copier..... 140
15-7 - Débogage..... 141
15-8 - Glossaire..... 142
15-9 - Exercices..... 142
16 - Classes et fonctions..... 143
16-1 - Temps..... 143
16-2 - Fonctions pures..... 143
16-3 - Modificateurs..... 144
16-4 - Prototypage versus planification..... 145
16-5 - Débogage..... 146
16-6 - Glossaire..... 147
16-7 - Exercices..... 147
17 - Classes et méthodes..... 148
17-1 - Fonctionnalités orientées objet..... 148
17-2 - Afficher des objets..... 149
17-3 - Un autre exemple..... 150
17-4 - Un exemple plus compliqué..... 151
17-5 - La méthode init..... 151
17-6 - La méthode __str__..... 152
17-7 - Surcharge d'opérateur..... 152
17-8 - Résolution de méthode basée sur le type..... 153
17-9 - Polymorphisme..... 154
17-10 - Débogage..... 155
17-11 - Interface et implémentation..... 155
17-12 - Glossaire..... 156
17-13 - Exercices..... 156
18 - Héritage..... 157
18-1 - Objets carte de jeu..... 157
18-2 - Attributs de classe..... 158
18-3 - Comparer des cartes..... 159
18-4 - Paquets de cartes..... 160
18-5 - Afficher le paquet..... 160
18-6 - Ajouter, enlever, mélanger et trier..... 160
18-7 - Héritage..... 161
18-8 - Diagrammes de classes..... 162
18-9 - Débogage..... 163
18-10 - Encapsulation de données..... 164
18-11 - Glossaire..... 165
18-12 - Exercices..... 166
19 - Les bonus..... 167
19-1 - Expressions conditionnelles..... 168
19-2 - Les listes en compréhension..... 169
19-3 - Les générateurs..... 169
19-4 - Les fonctions any et all..... 170
19-5 - Les ensembles (sets)..... 171
19-6 - Les compteurs (Counter)..... 172
19-7 - Dictionnaire de type defaultdict..... 173
19-8 - Tuples nommés..... 174
19-9 - Assembler des arguments avec mot-clé..... 175
19-10 - Glossaire..... 176

19-11 - Exercices....176
A - Débogage....177
A-1 - Erreurs de syntaxe....177
A-1-1 - Je n'arrête pas de modifier et rien ne change....178
A-2 - Erreurs d'exécution....178
A-2-1 - Mon programme ne fait absolument rien....178
A-2-2 - Mon programme s'arrête et ne fait plus rien....179
A-2-3 - Lorsque j'exécute le programme, j'obtiens une exception....180
A-2-4 - J'ai ajouté tant d'instructions print, je suis submergé par la sortie....181
A-3 - Erreurs sémantiques....181
A-3-1 - Mon programme ne fonctionne pas....181
A-3-2 - J'ai une grosse expression touffue qui ne fait pas ce que j'attends d'elle....182
A-3-3 - J'ai une fonction qui ne renvoie pas ce que j'attends....182
A-3-4 - Je me retrouve complètement coincé et j'ai besoin d'aide....183
A-3-5 - Mais non, j'ai vraiment besoin d'aide....183
B - Analyse des algorithmes....183
B-1 - Ordre de croissance et complexité....184
B-2 - Analyse des opérations Python de base....186
B-3 - Analyse des algorithmes de recherche....188
B-4 - Tables de hachage....188
B-5 - Glossaire....192
Remerciements Developpez....192

Introduction

L'étrange histoire de ce livre

En janvier 1999, je me préparais à donner un cours d'introduction à la programmation en Java. Je l'avais enseigné trois fois et je devenais frustré. Le taux d'échec dans la classe était trop élevé et, même pour les étudiants ayant réussi, le niveau global était trop faible.

Un des problèmes que je voyais était les livres. Ils étaient trop gros, avec trop de détails inutiles à propos de Java, et ne contenaient pas assez de conseils de haut niveau sur la façon de programmer. Et ils souffraient tous d'un effet piège : ils commençaient doucement, se complexifiaient progressivement, puis, quelque part autour du Chapitre 5, c'était la catastrophe. Les étudiants recevaient trop de notions nouvelles, trop vite, et je devais passer le reste du semestre à ramasser les morceaux.

Deux semaines avant le premier jour de cours, j'ai décidé d'écrire mon propre livre. Mes objectifs étaient :

- être bref. Il est préférable que les étudiants lisent 10 pages, plutôt qu'ils ne lisent pas 50 pages ;
- faire attention au vocabulaire. J'ai essayé de minimiser le jargon et de définir chaque terme lors de la première utilisation ;
- construire progressivement. Pour éviter les pièges, j'ai pris les sujets les plus difficiles et je les ai divisés en une série de petites étapes ;
- se concentrer sur la programmation, et non sur le langage de programmation. J'ai inclus un sous-ensemble minimal de notions utiles de Java et laissé le reste de côté.

Il me fallait un titre, alors je choisis « *Comment maîtriser la science de l'informatique* » sur un caprice.

La première version était un peu brute de fonderie, mais elle a fonctionné. Les étudiants l'ont lue et ils en ont compris assez pour que je puisse aborder en classe juste les sujets difficiles et intéressants et (surtout) pour que je puisse les faire pratiquer.

J'ai publié le livre sous la licence de documentation libre GNU, qui permet aux utilisateurs de copier, modifier et distribuer le livre.

Ce qui est arrivé ensuite est très sympa. Jeff Elkner, un professeur en Virginie, a adopté mon livre et l'a « traduit » en Python. Il m'a envoyé une copie de son adaptation, et j'ai eu l'expérience inédite d'apprendre Python en lisant mon propre livre. Sous le nom d'éditeur *Green Tea Press*, j'ai publié la première version Python en 2001.

En 2003, j'ai commencé à enseigner à l'université Olin et je devais enseigner Python pour la première fois. Le contraste avec Java était frappant. Les étudiants ont eu moins de peine, appris plus, travaillé sur des projets plus intéressants, et généralement ont eu beaucoup plus de plaisir.

Depuis, j'ai continué à développer le livre, en corrigeant les erreurs, en améliorant certains des exemples et en ajoutant du matériel nouveau, en particulier les exercices.

Le résultat est ce livre, maintenant avec le titre moins grandiose « *Pensez en Python* ». Certains de ces changements sont :

- j'ai ajouté une section sur le débogage à la fin de chaque chapitre. Ces sections présentent les techniques générales pour trouver et éviter les bogues et des avertissements sur les pièges de Python ;
- j'ai ajouté une série d'études de cas - exemples plus longs avec des exercices, des solutions et discussions ;
- j'ai élargi la discussion sur les modèles de développement de programmes et sur les patrons de conception (*design patterns*) de base ;
- j'ai ajouté les annexes sur le débogage et l'analyse des algorithmes.

La deuxième édition de « *Pensez Python* » a ces nouvelles fonctionnalités :

- j'ai ajouté plus d'exercices, allant de courts tests de compréhension à quelques projets d'envergure. La plupart des exercices incluent un lien vers ma solution ;

- le livre et tout le code inclus ont été mis à jour pour Python 3 ;
- j'ai ajouté quelques sections et plus de détails sur le web, pour aider les débutants à commencer à exécuter Python dans un navigateur, ce qui vous dispense d'installer Python tant que vous n'en avez pas besoin ;
- pour le chapitre **4.1** je suis passé de mon propre logiciel graphique tortue, appelé Swampy, à un module Python plus standard, turtle, plus facile à installer et plus puissant ;
- j'ai ajouté le nouveau Chapitre 19 intitulé « Les bonus », qui présente quelques fonctionnalités supplémentaires de Python qui ne sont pas strictement nécessaires, mais parfois utiles.

J'espère que vous aimerez travailler avec ce livre, et qu'il vous aidera à apprendre à programmer et à appréhender la science de l'informatique, au moins un peu.
Allen B. Downey
Collège Olin

Remerciements

Un grand merci à Jeff Elkner, qui a adapté mon livre Java à Python, qui a démarré ce projet et m'a présenté ce qui est devenu mon langage préféré.

Merci également à Chris Meyers, qui a contribué avec plusieurs sections de *« Comment maîtriser la science de l'informatique »*.

Merci à la Free Software Foundation d'avoir développé la licence de documentation libre GNU, qui a contribué à rendre possible ma collaboration avec Jeff et Chris, et à la Creative Commons pour la licence que j'utilise maintenant.

Merci aux rédacteurs de chez Lulu, qui ont travaillé sur *« Comment maîtriser la science de l'informatique »*.

Merci aux rédacteurs de chez O'Reilly Media qui ont travaillé sur « *Pensez en Python* ».

Merci à tous les étudiants qui ont travaillé avec les versions antérieures de ce livre et à tous les contributeurs (énumérés ci-dessous) qui ont envoyé des corrections et suggestions.

Liste des contributeurs

Plus de 100 lecteurs réfléchis et ayant l'œil vif ont envoyé des suggestions et corrections au cours des dernières années. Leurs contributions et l'enthousiasme pour ce projet ont été d'une grande aide.

Si vous avez des suggestions ou corrections, veuillez envoyer un courriel à feedback@thinkpython.com. Si je fais un changement basé sur vos commentaires, je vous ajouterai à la liste de contributeurs (sauf si vous demandez à être omis).

Si vous incluez au moins une partie de la phrase où l'erreur apparaît, cela me facilitera la recherche. Des numéros de page et de section sont très bien aussi, mais pas tout à fait aussi faciles à utiliser. Merci !

- Lloyd Hugh Allen a envoyé une correction pour la Section 8.4 ;
- Yvon Boulianne a envoyé une correction pour une erreur sémantique dans le Chapitre 5 ;
- Fred Bremmer a effectué une correction dans la Section 2.1 ;
- Jonah Cohen a écrit les scripts Perl pour convertir la source LaTeX de ce livre en beau HTML ;
- Michael Conlon a envoyé une correction grammaticale pour le Chapitre 2 et une amélioration du style du Chapitre 1, et il a lancé la discussion sur les aspects techniques des interpréteurs ;

- Benoît Girard a envoyé la correction d'une erreur humoristique dans la Section 5.6 ;
- Courtney Gleason et Katherine Smith ont écrit horsebet.py, utilisé comme étude de cas dans la version précédente du livre. Leur programme peut maintenant être trouvé sur le site web ;
- Lee Harr a effectué plus de corrections qu'on n'en peut énumérer ici, et il devrait vraiment être mentionné comme l'un des principaux éditeurs du texte ;
- James Kaylin est un étudiant qui a utilisé ce texte. Il a envoyé de nombreuses corrections ;
- David Kershaw a corrigé la fonction défectueuse concat_deux_fois de la Section 3.10 ;
- Eddie Lam a envoyé de nombreuses corrections pour les Chapitres 1, 2 et 3. Il a également corrigé le Makefile, de façon qu'il crée un index lors de la première exécution et nous a aidé à établir un schéma de gestion de versions ;
- Man-Yong Lee a envoyé une correction de l'exemple de code de la Section 2.4 ;
- David Mayo a fait remarquer que le mot « inconsciemment » dans le Chapitre 1 devait être modifié en « subconsciemment » ;
- Chris McAloon a envoyé plusieurs corrections pour les Sections 3.9 et 3.10 ;
- Matthew J. Moelter a été un contributeur de longue durée, qui a envoyé de nombreuses corrections et suggestions pour le livre ;
- Simon Dicon Montford a signalé une définition de fonction manquante et plusieurs fautes de frappe dans le Chapitre 3. Il a également décelé des erreurs dans la fonction incremente du Chapitre 13 ;
- John Ouzts a corrigé la définition de « return value » dans le Chapitre 3 ;
- Kevin Parks a envoyé de commentaires pertinents et des suggestions pour améliorer la distribution du livre ;
- David Pool a signalé une faute de frappe dans le glossaire du Chapitre 1, et a envoyé des mots gentils d'encouragement ;
- Michael Schmitt a envoyé une correction pour le chapitre sur les fichiers et les exceptions ;
- Robin Shaw a signalé une erreur dans la Section 13.1, où la fonction printTime était utilisée dans un exemple sans avoir été définie ;
- Paul Sleigh a trouvé une erreur dans le Chapitre 7 et un bogue dans un script Perl de Jonah Cohen qui génère du HTML à partir de LaTeX ;
- Craig T. Snydal teste ce livre pour un cours à l'université Drew. Il a contribué avec plusieurs suggestions pertinentes et corrections ;
- Ian Thomas et ses étudiants utilisent le texte pour un cours de programmation. Ils sont les premiers à tester les chapitres de la seconde moitié du livre, et ils ont fait de nombreuses corrections et suggestions ;
- Keith Verheyden a envoyé une correction pour le Chapitre 3 ;
- Peter Winstanley nous a signalé une erreur de longue date dans le Chapitre 3 ;
- Chris Wrobel a apporté des corrections au code du chapitre sur les E/S de fichiers et les exceptions ;
- Moshe Zadka a apporté des contributions inestimables à ce projet. En plus d'écrire le premier brouillon du chapitre sur les Dictionnaires, il nous a guidé de façon continue lors des premières étapes du livre ;
- Christoph Zwerschke a envoyé plusieurs corrections et suggestions pédagogiques, et a expliqué la différence entre *gleich* et *selbe (égal et même, identique, en allemand)* ;
- James Mayer nous a signalé tout un tas de fautes d'orthographe et de frappe, dont deux dans la liste des contributeurs ;
- Hayden McAfee a décelé une incohérence pouvant prêter à confusion entre deux exemples ;
- Angel Arnal fait partie d'une équipe internationale de traducteurs et travaille sur la version en espagnol du texte. Il a également trouvé quelques erreurs dans la version en anglais ;
- Tauhidul Hoque et Lex Berezhny ont créé les illustrations du Chapitre 1 et ont amélioré la plupart des autres illustrations ;
- Le Dr Michele Alzetta a décelé une erreur dans le Chapitre 8 et nous a envoyé quelques commentaires pédagogiques intéressants et des suggestions à propos de Fibonacci ;
- Andy Mitchell a trouvé une faute de frappe dans le Chapitre 1 et un exemple défectueux dans le Chapitre 2 ;
- Kalin Harvey a suggéré une clarification dans le Chapitre 7 et a trouvé quelques fautes de frappe ;
- Christopher P. Smith a trouvé quelques fautes de frappe et nous a aidé à mettre à jour le livre pour Python 2.2 ;
- David Hutchins a trouvé une faute de frappe dans l'Avant-propos ;
- Gregor Lingl enseigne le Python dans un lycée à Vienne, en Autriche. Il travaille à une traduction du livre en allemand et a trouvé deux ou trois erreurs graves dans le Chapitre 5 ;
- Julie Peters a trouvé une faute de frappe dans l'Introduction ;
- Florin Oprina nous a envoyé une amélioration de makeTime, une correction de afficher_temps, et une jolie faute de frappe ;
- D. J. Webre a suggéré une clarification dans le Chapitre 3 ;

- Ken a trouvé une poignée d'erreurs dans les Chapitres 8, 9 et 11 ;
- Ivo Wever a trouvé une faute de frappe dans le Chapitre 5 et a suggéré une clarification dans le Chapitre 3 ;
- Curtis Yanko a suggéré une clarification dans le Chapitre 2 ;
- Ben Logan a signalé un certain nombre de fautes de frappe et des problèmes dans la version HTML du livre ;
- Jason Armstrong a repéré un mot manquant dans le Chapitre 2 ;
- Louis Cordier a remarqué un endroit dans le Chapitre 16 où le code ne concordait pas avec le texte ;
- Brian Cain a proposé plusieurs clarifications dans les Chapitres 2 et 3 ;
- Rob Black nous a envoyé un paquet de corrections, y compris quelques modifications pour Python 2.2 ;
- Jean-Philippe Rey, de l'École Centrale de Paris, nous a envoyé de nombreux patchs, y compris quelques mises à jour pour Python 2.2 et d'autres améliorations judicieuses ;
- Jason Mader de l'Université George Washington a fait de nombreuses suggestions utiles et corrections ;
- Jan Gundtofte-Bruun nous a rappelé qu'« un erreur » est une erreur ;
- Abel David et Alexis Dinno nous ont rappelé que le pluriel de « matrice » est « matrices », et pas « matrixes ». Cette erreur était dans le livre depuis des années, mais deux lecteurs ayant les mêmes initiales l'ont rapportée le même jour. Bizarre ;
- Charles Thayer nous a encouragé à nous débarrasser des points-virgules que nous avions mis à la fin de certaines instructions et à clarifier l'utilisation des mots « argument » et « paramètre » ;
- Roger Sperberg nous a indiqué un raisonnement tordu dans le Chapitre 3 ;
- Sam Bull nous a indiqué un paragraphe confus dans le Chapitre 2 ;
- Andrew Cheung nous a signalé deux occurrences de « use before def » (variable utilisée avant son initialisation) ;
- C. Corey Capel a mis en évidence un mot manquant dans le 3^{e} théorème du débogage et une faute de frappe dans le Chapitre 4 ;
- Alessandra nous a aidé à éliminer quelques confusions de Turtle ;
- Wim Champagne a trouvé une inversion de mots dans un exemple de dictionnaire ;
- Douglas Wright a indiqué un problème de division entière en `arc` ;
- Jared Spindor a trouvé quelques déchets à la fin d'une proposition ;
- Lin Peiheng nous a envoyé un certain nombre de suggestions très utiles ;
- Ray Hagtvedt nous a signalé deux erreurs et pas tout à fait une erreur ;
- Torsten Hübsch nous a indiqué une inconsistance dans Swampy ;
- Inga Petuhhov a corrigé un exemple dans le Chapitre 14 ;
- Arne Babenhauserheide a envoyé plusieurs corrections utiles ;
- Mark E. Casida est est très fort pour repérer les répétitions de mots ;
- Scott Tyler a complété ce qui manquait. Et en plus, il a envoyé tout un tas de corrections ;
- Gordon Shephard a envoyé plusieurs corrections, chacune dans un mail différent ;
- Andrew Turner a indiqué une erreur de `spot` dans le Chapitre 8 ;
- Adam Hobart a résolu un problème de division entière dans `arc` ;
- Daryl Hammond et Sarah Zimmerman ont fait remarquer que j'ai servi `math.pi` trop tôt. And Zim a décelé une faute de frappe ;
- George Sass a trouvé un bogue dans la section Débogage ;
- Brian Bingham a suggéré l'Exercice **5** ;
- Leah Engelbert-Fenton a fait remarquer que j'ai utilisé `tuple` comme nom de variable, à l'encontre de mon propre conseil. En plus, elle a trouvé beaucoup de fautes de frappe et une erreur de type « use before def » ;
- Joe Funke a mis en évidence une faute de frappe ;
- Chao-chao Chen a trouvé une incohérence dans l'exemple Fibonacci ;
- Jeff Paine connaît la différence entre espace et spam ;
- Lubos Pintes a signalé une faute de frappe ;
- Gregg Lind et Abigail Heithoff ont suggéré l'Exercice **3** ;
- Max Hailperin a envoyé de nombreuses corrections et suggestions. Max est l'un des auteurs de l'extraordinaire livre **Concrete Abstractions**, que vous pourriez vouloir lire, après avoir fini de lire celui-ci ;
- Chotipat Pornavalai a trouvé une erreur dans un message d'erreur ;
- Stanislaw Antol a envoyé une liste de suggestions très utiles ;
- Eric Pashman a envoyé de nombreuses corrections pour les Chapitres 4-11 ;
- Miguel Azevedo a trouvé quelques fautes de frappe ;
- Jianhua Liu a envoyé une longue liste de corrections ;
- Nick King a trouvé un mot manquant ;
- Martin Zuther a envoyé une longue liste de suggestions ;

- Adam Zimmerman a trouvé une incohérence dans mon instance d'une « instance » et plusieurs autres erreurs ;
- Ratnakar Tiwari a suggéré une note de bas de page expliquant les triangles dégénérés ;
- Anurag Goel a suggéré une autre solution pour `est_en_ordre_alphabetique` et a envoyé quelques corrections supplémentaires. Et il sait épeler Jane Austen ;
- Kelli Kratzer a signalé une des fautes de frappe ;
- Mark Griffiths a indiqué un exemple prêtant à confusion dans le Chapitre 3 ;
- Roydan Ongie a trouvé une erreur dans ma méthode de Newton ;
- Patryk Wolowiec m'a aidé à résoudre un problème dans la version HTML ;
- Mark Chonofsky m'a parlé d'un nouveau mot-clé dans Python 3 ;
- Russell Coleman m'a aidé pour ma géométrie ;
- Wei Huang a signalé plusieurs fautes de frappe ;
- Karen Barber a repéré la plus ancienne faute de frappe dans le livre ;
- Nam Nguyen a trouvé une faute de frappe et a signalé que j'ai utilisé le modèle Decorator sans mentionner son nom ;
- Stéphane Morin a envoyé plusieurs corrections et suggestions ;
- Paul Stoop a corrigé une faute de frappe dans `utilise_uniquement` ;
- Eric Bronner a signalé une confusion dans la discussion sur l'ordre des opérations ;
- Alexandros Gezerlis a établi un nouveau record en ce qui concerne le nombre et la qualité des suggestions qu'il a envoyées. Nous sommes profondément reconnaissant !
- Gray Thomas distingue la droite de la gauche ;
- Giovanni Escobar Sosa a envoyé une longue liste de corrections et suggestions ;
- Alix Etienne a corrigé une des URL ;
- Kuang He a trouvé une faute de frappe ;
- Daniel Neilson a corrigé une erreur concernant l'ordre des opérations ;
- Will McGinnis a signalé que `polyligne` était définie de façon différente à deux endroits ;
- Swarup Sahoo a signalé un point-virgule manquant ;
- Frank Hecker a indiqué un exercice dont l'énoncé était incomplet, et quelques liens morts ;
- Animesh B m'a aidé à nettoyer un exemple ambigu ;
- Martin Caspersen a trouvé deux erreurs d'arrondi ;
- Gregor Ulm a envoyé plusieurs corrections et suggestions ;
- Dimitrios Tsirigkas m'a suggéré de clarifier un exercice ;
- Carlos Tafur a envoyé une page de corrections et suggestions ;
- Martin Nordsletten a trouvé un bogue dans la solution d'un exercice ;
- Lars O.D. Christensen a trouvé une référence morte ;
- Victor Simeone a trouvé une faute de frappe ;
- Sven Hoexter a signalé qu'une variable nommée `input` occulte une fonction interne ;
- Viet Le a trouvé une faute de frappe ;
- Stephen Gregory a signalé le problème lié à `cmp` en Python 3 ;
- Matthew Shultz m'a signalé un lien mort
- Lokesh Kumar Makani m'a signalé quelques liens morts et quelques modifications dans les messages d'erreur ;
- Ishwar Bhat a corrigé ma description du dernier théorème de Fermat ;
- Brian McGhie a suggéré une clarification ;
- Andrea Zanella a traduit le livre en italien, et a envoyé de nombreuses corrections pendant ce temps ;
- Beaucoup, beaucoup de remerciements à Melissa Lewis et Luciano Ramalho pour les excellents commentaires et suggestions pour la 2^{e} édition ;
- Remerciements à Harry Percival de PythonAnywhere pour aider les gens à commencer à exécuter Python dans un navigateur ;
- Xavier Van Aubel a fait plusieurs corrections utiles dans la 2^{e} édition.

1 - Le chemin du programme

Le but de ce livre est de vous apprendre à penser comme un informaticien. Cette façon de penser combine certaines des meilleures caractéristiques des mathématiques, de l'art de l'ingénieur et des sciences. Comme les mathématiciens, les informaticiens utilisent des langages formels pour désigner des idées (spécifiquement, des

calculs). Comme les ingénieurs, ils conçoivent des choses, en assemblant des composants en systèmes et en évaluant des compromis entre les différentes solutions. Comme les scientifiques, ils observent le comportement des systèmes complexes, formulent des hypothèses et testent des prévisions.

La compétence la plus importante pour un informaticien est la capacité de résoudre des problèmes. La **résolution de problèmes** signifie la capacité de formuler des problèmes, de penser de manière créative aux solutions, et d'exprimer une solution claire et précise. Il se trouve que le processus d'apprentissage de la programmation est une excellente occasion de pratiquer les compétences de résolution de problèmes. Voilà pourquoi ce chapitre est appelé « Le chemin du programme ».

À un certain niveau, vous allez apprendre à programmer, une compétence utile en elle-même. Sur un autre plan, vous pourrez utiliser la programmation comme un moyen pour une fin. Au fur et à mesure que nous nous avançons, cette fin deviendra plus claire.

1-1 - Qu'est-ce qu'un programme ?

Un **programme** est une séquence d'instructions qui spécifie comment effectuer un calcul. Le calcul pourrait être quelque chose de mathématique, comme résoudre un système d'équations ou trouver les racines d'un polynôme, mais il peut aussi être un calcul symbolique tel que la recherche et remplacement de texte dans un document ou quelque chose de graphique, comme le traitement d'une image ou la lecture d'une vidéo.

Les détails diffèrent d'un langage à un autre, mais quelques instructions de base apparaissent dans à peu près tous les langages :

- **entrée :** obtenir des données à partir du clavier, d'un fichier, du réseau ou un autre dispositif ;
- **sortie :** afficher des données à l'écran, les enregistrer dans un fichier, les envoyer sur le réseau, etc. ;
- **maths :** effectuer des opérations mathématiques de base, comme l'addition et la multiplication ;
- **exécution conditionnelle :** vérifier certaines conditions et exécuter le code approprié ;
- **répétition :** effectuer une action à plusieurs reprises, le plus souvent avec une certaine variation.

Croyez-le ou non, c'est à peu près tout ce qu'il y a à faire. Chaque programme que vous avez déjà utilisé, quelle que soit sa complexité, est constitué d'instructions qui ressemblent à peu près à celles-ci. Alors, vous pouvez penser à la programmation comme au processus de division d'une grande tâche complexe en sous-tâches de plus en plus petites, jusqu'à ce que les sous-tâches soient assez simples pour être effectuées en utilisant l'une de ces instructions de base.

1-2 - Exécuter Python

L'un des défis de débuter avec Python est que vous pourriez avoir à installer Python et des logiciels connexes sur votre ordinateur. Si vous connaissez bien votre système d'exploitation, et surtout si vous êtes à l'aise avec l'interface de ligne de commande, vous n'aurez aucune difficulté à installer Python. Mais pour les débutants, il peut être douloureux à apprendre en même temps l'administration système et la programmation.

Pour éviter ce problème, je vous recommande de commencer par exécuter du code Python dans un navigateur. Plus tard, lorsque vous serez à l'aise avec Python, je vous suggérerai de l'installer sur votre ordinateur.

Il y a un certain nombre de pages web que vous pouvez utiliser pour exécuter du Python. Si vous en avez déjà une favorite, utilisez-la. Sinon, je recommande PythonAnywhere. Je fournis des instructions détaillées pour commencer à l'adresse **http://tinyurl.com/thinkpython2e**.

Il existe deux versions de Python, appelées Python 2 et Python 3. Elles sont très semblables, de sorte que si vous apprenez l'une, il est facile de passer à l'autre. En tant que débutant, vous ne rencontrerez que peu de différences. Ce livre est écrit pour Python 3, mais j'y ai inclus quelques notes sur Python 2.

L'**interpréteur** Python est un programme qui lit et exécute du code Python. Selon votre environnement, vous pourriez lancer l'interpréteur en cliquant sur une icône, ou en tapant python sur une ligne de commande. Quand il démarre, vous devriez voir s'afficher quelque chose de ce genre :

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les trois premières lignes contiennent des informations sur l'interpréteur et le système d'exploitation sur lequel il tourne, si bien que les informations affichées en sortie peuvent être différentes dans votre cas. Mais vous devez vérifier que le numéro de version, qui est 3.4.0 dans cet exemple, commence par 3, ce qui indique que vous utilisez Python 3. S'il commence par 2, vous utilisez (vous l'aurez deviné) Python 2.

La dernière ligne est une **invite** qui indique que l'interpréteur est prêt pour la saisie de votre code. Si vous tapez une ligne de code et appuyez sur la touche Entrée, l'interpréteur affiche le résultat :

```
>>> 1 + 1
2
```

Maintenant, vous êtes prêt à commencer. À partir de maintenant, je suppose que vous savez comment démarrer l'interpréteur Python et exécuter du code.

1-3 - Le premier programme

Traditionnellement, le premier programme que vous écrivez dans un nouveau langage est appelé « Hello, World! », parce que tout ce qu'il fait est d'afficher les mots « Hello, World! ». En Python, il ressemble à ceci :

Python 3
```
>>> print('Hello, World!')
```

Ceci est un exemple d'une **instruction d'impression**, même si elle n'imprime rien sur du papier. Elle affiche un résultat sur l'écran. Dans ce cas, le résultat est

```
Hello, World!
```

Les guillemets dans le programme marquent le début et la fin du texte à afficher ; ils ne figurent pas dans le résultat.

Les parenthèses indiquent que l'impression est une fonction. Nous étudierons les fonctions dans le chapitre **3**.

En Python 2, l'instruction print est légèrement différente ; ce n'est pas une fonction, et elle n'utilise donc pas de parenthèses.

Python 2
```
>>> print 'Hello, World!'
```

Bientôt, vous comprendrez mieux cette différence, mais c'est suffisant pour commencer.

1-4 - Opérateurs arithmétiques

Après « Hello, World! », l'étape suivante est l'arithmétique. Python fournit des **opérateurs**, qui sont des symboles spéciaux qui représentent des calculs comme l'addition et la multiplication.

Les opérateurs +, - et * effectuent l'addition, la soustraction et la multiplication, comme dans les exemples suivants :

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

L'opérateur / effectue la division :

```
>>> 84 / 2
42.0
```

Vous pourriez vous demander pourquoi le résultat est 42.0 au lieu de 42. Je vais vous l'expliquer dans la section suivante.

Enfin, l'opérateur ** effectue l'exponentiation, c'est-à-dire qu'il élève un nombre à une puissance :

```
>>> 6**2 + 6
42
```

Dans d'autres langages, c'est l'opérateur ^ qui est utilisé pour l'exponentiation, mais, en Python, ce dernier est un opérateur de bits appelé XOR (*ou* exclusif). Si vous n'êtes pas habitué aux opérateurs de bits, le résultat vous surprendra :

```
>>> 6 ^ 2
4
```

Je ne vais pas traiter les opérateurs de bits dans ce livre, mais vous pouvez lire à leur sujet sur **http://wiki.python.org/moin/BitwiseOperators**.

1-5 - Valeurs et types

Une **valeur** est l'une des choses fondamentales avec lesquelles fonctionne un programme, comme une lettre ou un chiffre. Certaines des valeurs que nous avons vues jusqu'à présent sont 2, 42.0, et 'Hello, World!'.

Ces valeurs appartiennent à différents **types** : 2 est un **entier**, 42.0 est un **nombre à virgule flottante**, et 'Hello, World!' est une **chaîne** ou **chaîne de caractères**, appelée ainsi parce que les lettres qu'elle contient sont enchaînées.

Si vous n'êtes pas sûr du type d'une valeur, l'interpréteur peut vous le dire :

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<class 'str'>
```

Dans ces résultats, le mot « class » est utilisé dans le sens d'une catégorie ; un type est une catégorie de valeurs.

Les entiers (en anglais, *integers*) appartiennent au type int, les chaînes (*strings*) appartiennent au type str et les nombres à virgule flottante (*floating-point numbers*) sont de type float.

Qu'en est-il des valeurs comme '2' et '42.0' ? Ils ressemblent à des nombres, mais ils sont entre guillemets comme les chaînes.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
```

```
<class 'str'>
```

Ce sont des chaînes.

Lorsque vous tapez un grand nombre entier, vous pourriez être tenté d'utiliser des virgules (ou des points) entre les groupes de chiffres, comme dans 1,000,000. Cela n'est pas un *entier* valide en Python, mais c'est une notation autorisée :

```
>>> 1,000,000
(1, 0, 0)
```

Ce n'est pas du tout ce que nous attendions ! Python interprète 1,000,000 comme une séquence de nombres entiers séparés par des virgules. Nous en apprendrons plus sur ce genre de séquences plus tard.

1-6 - Langages naturels et formels

Les **langages naturels** sont les langues que les gens parlent, comme l'anglais, l'espagnol et le français. Ils n'ont pas été inventés par des personnes (bien que les gens essaient d'y imposer un certain ordre) ; ils ont évolué naturellement.

Les **langages formels** sont des langages conçus par des personnes pour des applications spécifiques. Par exemple, la notation utilisée par les mathématiciens est un langage formel qui convient particulièrement bien pour désigner les relations entre les chiffres et les symboles. Les chimistes utilisent un langage formel pour représenter la structure chimique des molécules. Et surtout :

Les langages de programmation sont des langages formels qui ont été conçus pour exprimer des calculs.

Les langages formels ont tendance à avoir des règles de **syntaxe** strictes, qui régissent la structure des instructions. Par exemple, en mathématiques l'instruction 3 + 3 = 6 a une syntaxe correcte, mais 3 + = 3 $ 6 n'en a pas une. En chimie, H_2O est une formule syntaxiquement correcte, mais $_2Zz$ ne l'est pas.

Il y a deux types de règles de syntaxe, se rapportant aux symboles et à la structure. Les symboles sont les éléments de base du langage, tels que les mots, les chiffres et les éléments chimiques. Un des problèmes avec 3 + = 3 $ 6 est que $ n'est pas un symbole valide en mathématiques (du moins autant que je sache). De même, $_2Zz$ n'est pas valide parce qu'il n'y a aucun élément chimique avec le symbole Zz.

Le deuxième type de règle de syntaxe se rapporte à la façon dont les symboles sont combinés. L'équation 3+ = 3 est invalide parce que même si + et = sont des symboles valides, vous ne pouvez pas les avoir l'un après l'autre. De même, dans une formule chimique, l'indice vient après le nom de l'élément, pas avant.

Ceci est une phrase bien $tructurée en fr@nçais, m@is c*ntenant des symboles invalides. Cette phrase tous les symboles valides a, mais est invalide sa structure.

Lorsque vous lisez une phrase en français ou une expression dans un langage formel, vous devez comprendre sa structure (même si, dans un langage naturel, vous faites cela subconsciemment). Ce processus est appelé **analyse** (grammaticale, lexicale, syntaxique, etc.).

Même si les langages formels et naturels ont de nombreuses caractéristiques en commun - symboles, structure et syntaxe - il y a quelques différences :

- **ambiguïté :** les langages naturels sont pleins d'ambiguïté, que les gens traitent en utilisant des indices contextuels et d'autres informations. Les langages formels sont conçus pour être presque ou complètement sans ambiguïté, ce qui signifie que toute instruction a exactement un sens, peu importe le contexte ;

- **redondance :** afin de compenser l'ambiguïté et de réduire les malentendus, les langages naturels emploient beaucoup de redondance. En conséquence, ils sont souvent verbeux. Les langages formels sont moins redondants et plus concis ;
- **littéralité :** les langages naturels sont pleins d'expressions idiomatiques et de métaphores. Si je dis, « Les dés sont jetés », il n'y a probablement aucun dé et rien n'est jeté (cette expression signifie que quelqu'un a pris une décision). Les langages formels veulent dire exactement ce qu'ils disent.

Comme nous grandissons tous en parlant des langages naturels, il est parfois difficile de s'adapter aux langages formels. La différence entre le langage formel et naturel est analogue à la différence entre la poésie et la prose, mais plus encore :

- **poésie :** les mots sont utilisés pour leur sonorité ainsi que pour leur signification, et le poème dans son ensemble crée un effet ou une réaction émotionnelle. L'ambiguïté est non seulement fréquente, mais souvent délibérée ;
- **prose :** le sens littéral des mots est plus important et la structure contribue plus au sens. La prose est plus susceptible à l'analyse que la poésie, mais encore souvent ambiguë ;
- **programmes :** la signification d'un programme d'ordinateur est littérale et sans ambiguïté, et peut être comprise en totalité par l'analyse des symboles et de la structure.

Les langages formels sont plus denses que les langages naturels, il faut donc plus de temps pour les lire. En outre, la structure est importante, si bien qu'il n'est pas toujours préférable de lire de haut en bas et de gauche à droite. Au lieu de cela, apprenez à analyser le programme dans votre tête, en identifiant les symboles et en interprétant la structure. Enfin, les détails comptent. De petites erreurs d'orthographe et ponctuation, que vous pouvez plus ou moins ignorer dans les langages naturels, peuvent faire une grande différence dans un langage formel.

1-7 - Débogage

Les programmeurs font des erreurs. Pour des raisons historiques anecdotiques, les erreurs de programmation sont appelées bogues et le processus de leur traque est appelé **débogage**.

La programmation, surtout le débogage, apporte parfois des émotions fortes. Si vous luttez avec un bogue difficile, vous pourriez vous sentir en colère, déprimé, ou embarrassé.

Il existe des preuves que les gens réagissent naturellement aux ordinateurs comme si ces derniers étaient des personnes. Quand ils fonctionnent bien, nous pensons à eux comme coéquipiers, et quand ils sont têtus ou grossiers, nous leur répondons de la même manière qu'à des gens grossiers, obstinés (Reeves et Nass, *L'équation Média : Comment les gens traitent les ordinateurs, la télévision et les nouveaux médias comme de vraies personnes et lieux*).

Se préparer à ces réactions peut vous aider à y faire face. Une approche est de considérer l'ordinateur comme un employé ayant certains points forts, comme la vitesse et la précision, et des faiblesses particulières, comme le manque d'empathie et l'incapacité de saisir l'image d'ensemble.

Votre travail est d'être un bon manager : trouver des moyens de tirer parti des points forts et atténuer les faiblesses. Et trouvez des façons d'utiliser vos émotions pour résoudre le problème, sans laisser vos réactions interférer avec votre capacité de travailler efficacement.

Apprendre à déboguer peut être frustrant, mais c'est une aptitude précieuse qui est utile pour de nombreuses activités au-delà de la programmation. À la fin de chaque chapitre se trouve une section, comme celle-ci, avec mes suggestions pour le débogage. J'espère que cela sera utile !

1-8 - Glossaire

- **résolution de problème :** le processus de formuler un problème, de trouver une solution et de l'exprimer.
- **langage de haut niveau :** un langage de programmation comme Python, conçu pour être facile à lire et à écrire pour les humains.

- **langage de bas niveau :** un langage de programmation conçu pour être facile à exécuter par un ordinateur ; également appelé « langage machine » ou « langage assembleur ».
- **portabilité :** une propriété d'un programme qui peut fonctionner sur plus d'un type d'ordinateur.
- **interpréteur :** un programme qui lit un autre programme et l'exécute.
- **invite :** caractères affichés par l'interpréteur pour indiquer qu'il est prêt à traiter la saisie de l'utilisateur.
- **programme :** un ensemble d'instructions qui spécifie un calcul.
- **instruction d'impression :** une instruction qui dit à l'interpréteur Python d'afficher une valeur sur l'écran.
- **opérateur :** un symbole spécial qui représente un calcul simple, comme l'addition, la multiplication ou la concaténation de chaînes.
- **valeur :** une des unités élémentaires de données, comme un nombre ou une chaîne, qu'un programme manipule.
- **type :** une catégorie de valeurs. Les types que nous avons vus jusqu'à présent sont des nombres entiers (type `int`), nombres à virgule flottante (type `float`), et chaînes (type `str`).
- **entier :** un type qui représente des nombres entiers.
- **virgule flottante :** un type qui représente des nombres avec des parties fractionnaires.
- **chaîne** ou **chaîne de caractères :** un type qui représente des séquences de caractères.
- **langage naturel :** toute langue parlée par des gens, qui a évolué naturellement.
- **langage formel :** tout langage conçu par des gens à des fins spécifiques, comme la représentation des idées mathématiques ou des programmes informatiques ; tous les langages de programmation sont des langages formels.
- **symbole :** l'un des éléments de base de la structure syntaxique d'un programme, analogue à un mot dans une langue naturelle.
- **syntaxe :** les règles qui régissent la structure d'un programme.
- **analyser :** examiner un programme et en analyser la structure syntaxique.
- **bogue :** une erreur dans un programme.
- **débogage :** le processus de détection et de correction des bogues.

1-9 - Exercices

Exercice 1

C'est une bonne idée de lire ce livre en face d'un ordinateur afin que vous puissiez essayer les exemples au fur et à mesure.

Chaque fois que vous expérimentez une nouvelle fonctionnalité, vous devriez essayer de faire des erreurs. Par exemple, dans le programme « Hello, world! », que se passe-t-il si vous omettez l'un des guillemets ? Ou si vous les omettez tous les deux ? Et si vous écrivez `print` *de façon incorrecte ?*

Ce genre d'expérience permet de vous rappeler ce que vous lisez ; il vous aide également lorsque vous programmez, parce que vous arrivez à apprendre ce que les messages d'erreur veulent dire. Il est préférable de faire des erreurs aujourd'hui et intentionnellement que plus tard et accidentellement.

1. *Dans une instruction* `print` *, que se passe-t-il si vous laissez de côté l'une des parenthèses, ou les deux ?*
2. *Si vous tentez d'afficher une chaîne, que se passe-t-il si vous laissez de côté l'un des guillemets, ou les deux ?*
3. *Vous pouvez utiliser un signe moins pour obtenir un nombre négatif comme* `-2` *. Qu'advient-il si vous mettez un signe plus devant un nombre ? Mais si vous écrivez* `2++2` *?*
4. *Dans la notation mathématique, des zéros devant un nombre ne posent aucun problème, comme dans le cas de* `02` *. Qu'est-ce qu'il se passe si vous essayez ceci en Python ?*
5. *Qu'est-ce qu'il se passe si vous avez deux valeurs avec aucun opérateur entre elles ?*

Exercice 2

Lancez l'interpréteur Python et utilisez-le comme calculatrice.

1. *Combien de secondes y a-t-il dans 42 minutes et 42 secondes ?*

2 *Combien de miles y a-t-il dans 10 kilomètres ? Indice : il y a 1,61 kilomètre dans un mile.*
3 *Si vous exécutez une course de 10 kilomètres en 42 minutes et 42 secondes, quel est votre rythme moyen (durée écoulée par mile en minutes et secondes) ? Quelle est votre vitesse moyenne en miles par heure ?*

2 - Variables, expressions et instructions

Une des fonctionnalités les plus puissantes d'un langage de programmation est la possibilité de manipuler des variables. Une variable est un nom qui fait référence à une valeur.

2-1 - Instructions d'affectation

Une **instruction d'affectation** crée une nouvelle variable et lui donne une valeur :

```
>>> message = 'Et maintenant, quelque chose de complètement différent'
>>> n = 17
>>> pi = 3.141592653589793
```

Cet exemple effectue trois affectations. La première attribue une chaîne à une nouvelle variable nommée message ; la seconde donne la valeur 17 au nombre entier n ; la troisième affecte la valeur (approximative) de π à pi.

Une façon courante de représenter les variables sur le papier est d'écrire le nom avec une flèche pointant à sa valeur. Ce genre de représentation est appelé un **diagramme d'état**, car il montre l'état dans lequel se trouve chacune des variables (considérez-le comme l'état d'esprit de la variable). La figure 2.1 montre le résultat de l'exemple précédent.

Figure 2.1 : Diagramme d'état.

2-2 - Noms de variables

Les programmeurs choisissent généralement des noms significatifs pour leurs variables, qui expliquent à quoi sert la variable.

Les noms de variables peuvent être aussi longs que vous le souhaitez. Ils peuvent contenir des lettres et des chiffres, mais ils ne peuvent pas commencer par un chiffre. L'utilisation des lettres majuscules est autorisée, mais il est habituel d'utiliser seulement des minuscules pour les noms de variables.

Le caractère de soulignement, _, peut apparaître dans un nom. Il est souvent utilisé dans les noms avec plusieurs mots, comme votre_nom ou vitesse_de_l_hirondelle.

Si vous donnez à une variable un nom invalide, vous obtenez une erreur de syntaxe :

```
>>> 76trombones = 'grande parade'
SyntaxError: invalid syntax
>>> plus@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Théorie avancée de la fermentation alcoolique'
SyntaxError: invalid syntax
```

Le nom de variable 76trombones n'est pas valide parce qu'il commence par un nombre. plus@ n'est pas valide parce qu'il contient un caractère non autorisé, @. Mais quel est le problème avec class ?

Il se trouve que class est l'un des **mots-clés** ou **mots réservés** de Python. L'interpréteur utilise des mots-clés pour reconnaître la structure du programme, et ils ne peuvent pas être utilisés comme noms de variables.

Python 3 a les mots-clés suivants :

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

Vous n'avez pas besoin de mémoriser cette liste. Dans la plupart des environnements de développement, les mots-clés sont affichés dans une couleur différente ; si vous essayez d'en utiliser un comme nom de variable, vous le saurez.

2-3 - Expressions et instructions

Une **expression** est une combinaison de valeurs, variables et opérateurs. Une valeur, rien que par elle-même, est considérée comme une expression, et il en va de même avec une variable, si bien que les expressions suivantes sont toutes valides :

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

Lorsque vous tapez une expression, l'interpréteur l'**évalue**, ce qui signifie qu'il trouve la valeur de l'expression. Dans cet exemple, n a la valeur 17 et n + 25 a la valeur 42.

Une **instruction** est une unité de code qui a un effet, comme la création d'une variable ou l'affichage d'une valeur.

```
>>> n = 17
>>> print(n)
```

La première ligne est une instruction d'affectation qui donne une valeur à n. La deuxième ligne est une instruction print qui affiche la valeur de n.

Lorsque vous tapez une instruction, l'interpréteur l'exécute, ce qui signifie qu'il fait tout ce que dit l'instruction. En général, les instructions n'ont pas de valeurs.

2-4 - Mode de script

Jusqu'ici, nous avons exécuté Python en **mode interactif**, ce qui signifie que vous interagissez directement avec l'interpréteur. Le mode interactif est une bonne façon de commencer, mais si vous travaillez avec plus de quelques lignes de code, cela devient peu commode.

L'autre voie est d'enregistrer le code dans un fichier appelé un **script**, puis de lancer l'interpréteur en **mode de script** pour exécuter le script. Par convention, les scripts Python ont des noms qui se terminent par .py.

Si vous savez comment créer et exécuter un script sur votre ordinateur, vous êtes prêt à continuer. Sinon, je recommande d'utiliser à nouveau PythonAnywhere. J'ai posté des instructions pour l'exécution en mode de script à **http://tinyurl.com/thinkpython2e**.

Comme Python fournit les deux modes, vous pouvez tester des morceaux de code en mode interactif avant de les mettre dans un script. Mais il y a des différences entre le mode interactif et le mode de script qui peuvent être source de confusion.

Par exemple, si vous utilisez Python comme une calculatrice, vous pouvez taper :

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

La première ligne assigne une valeur à miles, mais elle n'a aucun effet visible. La deuxième ligne est une expression, alors l'interpréteur l'évalue et affiche le résultat. Il se trouve que le marathon est d'environ 42 km.

Mais si vous tapez le même code dans un script et l'exécutez, vous n'obtenez aucune sortie. En mode de script, une expression en elle-même n'a aucun effet visible. Python évalue effectivement l'expression, mais il ne montre pas sa valeur, sauf si vous lui dites :

```
miles = 26.2
print(miles * 1.61)
```

Ce comportement peut être déroutant au début.

Un script contient généralement une séquence d'instructions. S'il y a plus d'une instruction, les résultats apparaissent au fur et à mesure que les instructions sont exécutées.

Par exemple, le script

```
print(1)
x = 2
print(x)
```

affiche la sortie

```
1
2
```

L'instruction d'affectation ne produit aucune sortie.

Pour vérifier votre compréhension, tapez les instructions suivantes dans l'interpréteur Python et regardez ce qu'elles font :

```
5
x = 5
x + 1
```

Maintenant, mettez les mêmes déclarations dans un script et exécutez-le. Quel est le résultat ? Modifiez le script en transformant chaque expression dans une instruction d'impression et puis exécutez-le à nouveau.

2-5 - Ordre des opérations

Lorsqu'une expression contient plus d'un opérateur, l'ordre d'évaluation dépend de l'**ordre des opérations**. Pour les opérateurs mathématiques, Python suit la convention mathématique. L'acronyme **PEMDAS** est un moyen utile de mémoriser les règles de priorité (ou de précédence) :

- les **P**arenthèses ont la plus haute priorité et peuvent être utilisées pour forcer l'évaluation d'une expression dans l'ordre que vous voulez. Puisque les expressions entre parenthèses sont évaluées en premier, `2 * (3 - 1)` vaut 4, et `(1 + 1) ** (5 - 2)` vaut 8. Vous pouvez également utiliser des parenthèses pour rendre une expression plus facile à lire, comme dans `(minute * 100) / 60`, même si cela ne change pas le résultat ;
- l'**E**xponentiation a la prochaine priorité la plus élevée, donc `1 + 2 ** 3` vaut 9, pas 27, et `2 * 3 ** 2` est égal à 18, et pas à 36 ;
- la **M**ultiplication et la **D**ivision ont une priorité plus élevée que l'**A**ddition et la **S**oustraction. Donc `2 * 3 - 1` vaut 5, pas 4, et `6 + 4/2` vaut 8, pas 5 ;
- les opérateurs ayant la même priorité sont évalués de gauche à droite (sauf l'exponentiation). Ainsi, dans l'expression `degres / 2 * pi`, la division a lieu en premier et le résultat est multiplié par `pi`. Pour diviser par 2 π, vous pouvez utiliser des parenthèses ou écrire `degres / 2 / pi`.

Je ne fais pas trop d'efforts pour me rappeler la priorité des opérateurs. Si j'ai un doute en regardant l'expression, j'utilise des parenthèses pour obtenir l'ordre souhaité.

2-6 - Opérations sur des chaînes

Généralement, vous ne pouvez pas effectuer des opérations arithmétiques sur les chaînes de caractères, même si elles ressemblent à des nombres. Par exemple, les instructions suivantes sont invalides :

`'2'-'1'` `'truc'/'facile'` `'premier'*'un essai'`

Mais il y a deux exceptions, les opérateurs `+` et `*`.

L'opérateur `+` effectue la **concaténation des chaînes**, c'est-à-dire l'assemblage des chaînes bout à bout. Par exemple :

```
>>> premier = 'plate'
>>> second = 'forme'
>>> premier + second
plateforme
```

L'opérateur `*` fonctionne aussi sur les chaînes ; il effectue une répétition. Par exemple, `"Spam" * 3` a pour résultat `'SpamSpamSpam'`. Si l'une des valeurs est une chaîne, l'autre doit être un entier.

Cette utilisation de `+` et `*` a du sens par analogie avec l'addition et la multiplication. Tout comme `4 * 3` est équivalent à `4 + 4 + 4`, nous nous attendons à ce que `'Spam' * 3` soit équivalent à `'Spam' + 'Spam' + 'Spam'`, et c'est bien le cas. D'un autre côté, la concaténation et la répétition des chaînes diffèrent significativement de l'addition et la multiplication des nombres entiers. Pouvez-vous penser à une propriété de l'addition que la concaténation de chaînes n'a pas ?

2-7 - Commentaires

Lorsque les programmes deviennent de plus en plus volumineux et compliqués, ils sont plus difficiles à lire. Les langages formels sont denses, et il est souvent difficile de regarder un bout de code et de comprendre ce qu'il fait, ou pourquoi.

Pour cette raison, c'est une bonne idée d'ajouter à vos programmes des notes expliquant en langage naturel ce que le programme est en train de faire. Ces notes sont appelées **commentaires**, et commencent par le symbole `#` :

```
# calculer le pourcentage d'une heure écoulé
pourcentage = (minute * 100) / 60
```

Dans ce cas, le commentaire apparaît lui seul sur une ligne. Vous pouvez aussi mettre des commentaires à la fin d'une ligne :

```
pourcentage = (minute * 100) / 60     # pourcentage d'une heure
```

Tout, depuis le # jusqu'à la fin de la ligne est ignoré, cela n'a aucun effet sur l'exécution du programme.

Les commentaires sont les plus utiles quand ils documentent des caractéristiques non évidentes du code. Il est raisonnable de supposer que le lecteur comprendra *ce que* fait le code ; il est plus utile d'expliquer *pourquoi*.

Ce commentaire est redondant avec le code et donc inutile :

```
v = 5      # assigner la valeur 5 à v
```

Le commentaire suivant contient des informations utiles qui ne sont pas dans le code :

```
v = 5      # vitesse en mètres/seconde.
```

Le choix des bons noms de variables peut réduire le besoin de commentaires, mais des noms longs peuvent rendre les expressions complexes difficiles à lire, il y a donc un compromis à trouver.

2-8 - Débogage

Trois types d'erreurs peuvent se produire dans un programme : des erreurs de syntaxe, des erreurs d'exécution et les erreurs sémantiques. Il est utile de distinguer ces types d'erreurs afin de les retrouver plus rapidement.

- **Erreur de syntaxe :** la notion de « syntaxe » désigne la structure d'un programme et les règles relatives à cette structure. Par exemple, les parenthèses doivent être en paires, donc (1 + 2) est valide, mais 8) est une erreur de syntaxe.
  S'il y a une erreur de syntaxe quelque part dans votre programme, Python affiche un message d'erreur et s'arrête, et vous ne serez pas en mesure d'exécuter le programme. Pendant les premières semaines de votre carrière en programmation, vous pourriez passer beaucoup de temps à traquer les erreurs de syntaxe. Lorsque vous aurez acquis de l'expérience, vous ferez moins d'erreurs et les trouverez plus rapidement.
- **Erreur d'exécution :** le deuxième type d'erreurs est l'erreur d'exécution, appelé ainsi parce que l'erreur n'apparaît qu'après le lancement du programme. Ces erreurs sont également appelées exceptions, car elles indiquent généralement que quelque chose d'exceptionnel (et mauvais) est arrivé.
  Les erreurs d'exécution étant rares dans les programmes simples que vous verrez dans les premiers chapitres, il pourrait se passer un certain temps avant que vous n'en rencontriez une.
- **Erreur sémantique :** le troisième type d'erreurs est « sémantique », ce qui signifie lié au sens. S'il y a une erreur sémantique dans votre programme, il sera exécuté sans générer des messages d'erreur, mais il ne fera pas ce qu'il faut. Il va faire autre chose. Plus précisément, il va faire ce que vous lui avez dit de faire, mais pas ce que vous vouliez qu'il fasse.
  Identifier les erreurs sémantiques peut être délicat, car cela vous oblige à remonter en arrière en regardant la sortie du programme et en essayant de comprendre ce qu'il fait.

2-9 - Glossaire

- **variable :** un nom qui fait référence à une valeur.
- **affectation :** une instruction qui attribue une valeur à une variable.
- **diagramme d'état :** une représentation graphique d'un ensemble de variables et des valeurs auxquelles il se réfère.

- **mot-clé :** un mot réservé qui est utilisé pour analyser un programme ; vous ne pouvez pas utiliser des mots-clés comme `if`, `def`, et `while` comme noms de variables.
- **opérande :** l'une des valeurs sur lesquelles agit un opérateur.
- **expression :** une combinaison de variables, d'opérateurs et de valeurs, qui représente un résultat unique.
- **évaluer :** simplifier une expression en effectuant des opérations, dans le but de donner une valeur unique.
- **instruction :** une partie de code qui représente une commande ou une action. Jusqu'à présent, les instructions que nous avons vues sont les affectations et l'instruction d'impression.
- **exécuter :** exécuter une instruction et faire ce qu'elle demande.
- **mode interactif :** une façon d'utiliser l'interpréteur Python en tapant le code à l'invite de commande.
- **mode de script :** une façon d'utiliser l'interpréteur Python pour lire le code d'un script stocké dans un fichier et l'exécuter.
- **script :** un programme stocké dans un fichier.
- **ordre des opérations :** les règles de priorité (ou de précédence) régissant l'ordre dans lequel sont évaluées les expressions impliquant de multiples opérateurs et opérandes.
- **concaténer :** joindre deux opérandes bout à bout.
- **commentaire :** informations contenues dans un programme qui sont destinées aux autres programmeurs (ou toute personne lisant le code source) et n'ont aucun effet sur l'exécution du programme.
- **erreur de syntaxe :** une erreur dans un programme qui rend celui-ci impossible à analyser (et donc impossible à interpréter).
- **exception :** une erreur qui est détectée lorsque le programme est en cours d'exécution.
- **sémantique :** la signification d'un programme.
- **erreur sémantique :** une erreur dans un programme qui lui fait faire autre chose que ce que le programmeur veut.

2-10 - Exercices

Exercice 1

Je répète mon conseil du chapitre précédent : chaque fois que vous apprenez une nouvelle fonctionnalité, vous devriez l'essayer en mode interactif et faire des erreurs, afin de voir ce qui va mal.

- *Nous avons vu que* `n = 42` *est autorisé. Qu'en est-il de* `42 = n` *?*
- *Et dans ce cas :* `x = y = 1` *?*
- *Dans certains langages, chaque instruction se termine par un point-virgule,* `;` *. Que se passe-t-il si vous mettez un point-virgule à la fin d'une instruction en Python ?*
- *Que se passe-t-il si vous mettez un point à la fin d'une instruction ?*
- *Dans la notation mathématique, vous pouvez multiplier x et y comme ceci :* `xy` *. Que se passe-t-il si vous essayez cela en Python ?*

Exercice 2

Exercez-vous en utilisant l'interpréteur Python comme calculatrice :

1. *Le volume d'une sphère de rayon r est 4/3 π r^3. Quel est le volume d'une sphère de rayon 5 ?*
2. *Supposons que le prix d'un livre est de 24,95 €, mais les librairies obtiennent un rabais de 40 %. Les frais d'expédition s'élèvent à 3 € pour le premier exemplaire et à 75 centimes pour chaque exemplaire supplémentaire. Quel est le coût total de gros pour 60 copies ?*
3. *Si je pars de chez moi à 6 h 52 et que je cours 1,5 kilomètre à un rythme facile de 5 minutes et 10 secondes par km, puis 5 km au tempo plus soutenu de 4 minutes et 20 secondes par kilomètre et encore 1,5 km au rythme facile du début, à quelle heure serai-je de retour à la maison pour le petit déjeuner ?*

3 - Fonctions

Dans le cadre de la programmation, une **fonction** est une séquence nommée d'instructions qui effectue un calcul. Lorsque vous définissez une fonction, vous spécifiez le nom et la séquence d'instructions. Plus tard, vous pouvez « appeler » la fonction par son nom.

3-1 - Appels de fonctions

Nous avons vu déjà un exemple d'**appel de fonction** :

```
>>> type(42)
<class 'int'>
```

Le nom de la fonction est type. L'expression entre parenthèses est appelée **l'argument de la fonction**. Le résultat, pour cette fonction, est le type de l'argument.

De façon usuelle, on dit que la fonction « prend » un argument et qu'elle « retourne » ou « renvoie » un résultat. Ce résultat est appelé aussi la **valeur de retour**.

Python fournit des fonctions qui convertissent des valeurs d'un type à l'autre. La fonction int prend n'importe quelle valeur et la convertit en un nombre entier, si elle le peut, ou se plaint dans le cas contraire :

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

La fonction int peut convertir les valeurs à virgule flottante en entiers, mais elle n'arrondit pas ; elle en coupe la partie fractionnaire :

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

La fonction float convertit des entiers et des chaînes de caractères en nombres à virgule flottante :

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Enfin, str convertit son argument en une chaîne :

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3-2 - Fonctions mathématiques

Python dispose d'un module math, qui fournit la plupart des fonctions mathématiques usuelles. Un **module** est un fichier qui contient une collection de fonctions apparentées.

Avant que nous puissions utiliser les fonctions d'un module, nous devons l'importer avec une **instruction** import :

```
>>> import math
```

Cette instruction crée un **objet module** nommé math. Si vous affichez l'objet module, vous obtiendrez quelques informations à son sujet :

```
>>> math
<module 'math' (built-in)>
```

L'objet module contient les fonctions et variables définies dans le module. Pour accéder à l'une des fonctions, vous devez spécifier le nom du module et le nom de la fonction, séparés par un point. Ce format est appelé **notation pointée** ou parfois notation objet.

```
>>> ratio = puissance_signal / puissance_bruit
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> hauteur = math.sin(radians)
```

Le premier exemple utilise math.log10 pour calculer un rapport signal / bruit en décibels (en supposant que force_signal et force_bruit sont définis). Le module math fournit également log, qui calcule des logarithmes en base e.

Le second exemple trouve le sinus de radians. Le nom de la variable est une indication que sin et les autres fonctions trigonométriques (cos, tan, etc.) prennent des arguments mesurés en radians. Pour convertir de degrés en radians, divisez par 180 et multipliez par π :

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

L'expression math.pi obtient la variable pi à partir du module math. Sa valeur est une approximation à virgule flottante de π, avec une précision d'environ 15 chiffres.

Si vous avez des notions de trigonométrie, vous pouvez vérifier le résultat précédent en le comparant à la racine carrée de deux divisée par deux :

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3-3 - Composition

Jusqu'ici, nous avons examiné les éléments d'un programme - variables, expressions et instructions - en les traitant de façon isolée, sans parler de la façon de les combiner.

L'une des caractéristiques les plus utiles des langages de programmation est leur capacité de prendre de petites briques de construction et de les **composer**. Par exemple, l'argument d'une fonction peut être tout type d'expression, y compris une expression comprenant des opérateurs arithmétiques :

```
x = math.sin(degres / 360.0 * 2 * math.pi)
```

et même des appels de fonction :

```
x = math.exp(math.log(x+1))
```

Presque partout où vous pouvez mettre une valeur, vous pouvez mettre une expression arbitraire, à une exception près : le côté gauche d'une instruction d'affectation doit être un nom de variable. Toute autre expression sur le côté gauche est une erreur de syntaxe (nous verrons plus tard des exceptions à cette règle).

```
>>> minutes = heures * 60                    # correct
>>> heures * 60 = minutes                    # incorrect !
SyntaxError: can't assign to operator
```

3-4 - Créer de nouvelles fonctions

Jusqu'à présent, nous n'avons utilisé que des fonctions fournies par Python, mais il est également possible d'ajouter de nouvelles fonctions. Une **définition de fonction** spécifie le nom d'une nouvelle fonction et la séquence d'instructions exécutées lorsque la fonction est appelée.

Voici un exemple :

```
def afficher_paroles():
    print("J'abats des arbres, je danse et saute,")
    print("J'aime jouer avec des pâquerettes.")
```

Le mot clé def indique qu'il s'agit d'une définition de fonction. Le nom de la fonction est afficher_paroles. Les règles pour les noms de fonction sont les mêmes que pour les noms de variables : des lettres, des chiffres et des traits de soulignement sont autorisés, mais le premier caractère ne peut pas être un chiffre. Vous ne pouvez pas utiliser un mot-clé comme nom d'une fonction, et vous devriez éviter d'avoir une variable et une fonction du même nom.

Les parenthèses vides après le nom indiquent que cette fonction ne prend aucun argument.

La première ligne de la définition de la fonction est appelée l'**en-tête** ; le reste est appelé le **corps**. L'en-tête doit se terminer par un deux-points et le corps doit être indenté (les lignes commencent par des espaces). Par convention, l'indentation est toujours de quatre espaces. Le corps peut contenir un nombre quelconque d'instructions. La fin de l'indentation (retour au début de la ligne) marque la fin de la définition de la fonction.

Les chaînes dans les instructions d'impression sont entre guillemets doubles. Les guillemets simples et doubles font la même chose ; la plupart des gens utilisent des guillemets simples, sauf dans des cas comme celui-ci, où un guillemet simple (qui est aussi une apostrophe) apparaît dans la chaîne.

Tous les guillemets (simples et doubles) doivent être des « guillemets droits ». Les guillemets français ne sont pas autorisés en Python.

Si vous tapez une définition de fonction en mode interactif, l'interpréteur affiche trois points (...) pour vous faire savoir que la définition est incomplète :

```
>>> def afficher_paroles():
...     print("J'abats des arbres, je danse et saute,")
...     print("J'aime jouer avec des pâquerettes.")
...
```

Pour mettre fin à la fonction, vous devez entrer une ligne vide.

Définir une fonction crée un **objet fonction**, de type function :

```
>>> print(afficher_paroles)
<function afficher_paroles at 0xb7e99e9c>
>>> type(afficher_paroles)
<class 'function'>
```

La syntaxe pour appeler la nouvelle fonction est la même que pour les fonctions internes :

```
>>> afficher_paroles()
J'abats des arbres, je danse et saute,
J'aime jouer avec des pâquerettes.
```

Une fois que vous avez défini une fonction, vous pouvez l'utiliser dans une autre fonction. Par exemple, pour répéter le refrain précédent, nous pourrions écrire une fonction appelée repeter_paroles :

```
def repeter_paroles():
    afficher_paroles()
    afficher_paroles()
```

Puis appeler repeter_paroles :

```
>>> repeter_paroles()
J'abats des arbres, je danse et saute,
J'aime jouer avec des pâquerettes.
J'abats des arbres, je danse et saute,
J'aime jouer avec des pâquerettes.
```

Mais cela ne respecte plus vraiment les paroles de la chanson des Monty Python.

3-5 - Définitions et utilisation

En rassemblant les fragments de code de la section précédente, le programme complet ressemble à ceci :

```
def afficher_paroles():
    print("J'abats des arbres, je danse et saute,")
    print("J'aime jouer avec des pâquerettes.")

def repeter_paroles():
    afficher_paroles()
    afficher_paroles()

repeter_paroles()
```

Ce programme contient deux définitions de fonctions : afficher_paroles et repeter_paroles. Les définitions de fonctions sont exécutées exactement comme les autres instructions, mais l'effet est de créer des objets fonction. Les instructions à l'intérieur de la fonction ne sont pas exécutées jusqu'à ce que la fonction soit appelée, et la définition de la fonction ne génère aucune sortie.

Comme vous vous y attendez, vous devez créer une fonction avant de pouvoir l'exécuter. Autrement dit, la définition de la fonction doit être exécutée avant que la fonction soit appelée.

Comme exercice, déplacez la dernière ligne de ce programme au début, de sorte que l'appel de fonction apparaisse avant les définitions. Exécutez le programme et voyez quel message d'erreur vous obtenez.

Maintenant, remettez l'appel de la fonction en bas et déplacez la définition de afficher_paroles après la définition de repeter_paroles. Qu'advient-il lorsque vous exécutez ce programme ?

3-6 - Flux d'exécution

Pour garantir qu'une fonction soit définie avant sa première utilisation, vous devez connaître l'ordre dans lequel les instructions sont exécutées ; c'est ce que l'on appelle le flux d'exécution.

L'exécution commence toujours à la première instruction du programme. Les instructions sont exécutées une par une, dans l'ordre de haut en bas.

Les définitions de fonctions ne modifient pas le flux d'exécution du programme, mais rappelez-vous que les instructions à l'intérieur de la fonction ne sont pas exécutées avant que la fonction soit appelée.

Un appel de fonction est comme un détour dans le flux d'exécution. Au lieu d'aller à l'instruction suivante, le flux saute dans le corps de la fonction, il en exécute les instructions, puis il revient et reprend là où il s'était interrompu.

Cela semble assez simple, mais souvenez-vous qu'une fonction peut appeler une autre fonction. Lorsqu'il est arrivé au milieu d'une fonction, le programme peut devoir exécuter les instructions d'une autre fonction. Puis, lors de l'exécution de cette nouvelle fonction, le programme pourrait avoir à exécuter une autre fonction !

Heureusement, Python est doué pour garder la trace de l'endroit où il était arrivé, de sorte que chaque fois qu'une fonction se termine, le programme reprend là où il en était dans la fonction qui l'a appelée. Quand il arrive à la fin du programme, il se termine.

En résumé, lorsque vous lisez un programme, vous ne voulez pas toujours lire le code de haut en bas. Parfois, il est plus judicieux de suivre le flux d'exécution.

3-7 - Paramètres et arguments

Certaines des fonctions que nous avons vues exigent des arguments. Par exemple, lorsque vous appelez `math.sin` vous passez un nombre comme un argument. Certaines fonctions prennent plus d'un argument : `math.pow` en prend deux, la base et l'exposant.

Dans la fonction, les arguments sont assignés à des variables appelées **paramètres**. Voici une définition pour une fonction qui prend un argument :

```
def afficher_deux_fois(bruce):
    print(bruce)
    print(bruce)
```

Cette fonction assigne l'argument à un paramètre nommé `bruce`. Lorsque la fonction est appelée, elle affiche la valeur du paramètre (quel qu'il soit) à deux reprises.

Cette fonction marche pour n'importe quelle valeur qui peut être affichée.

```
>>> afficher_deux_fois('Spam')
Spam
Spam
>>> afficher_deux_fois(42)
42
42
>>> afficher_deux_fois(math.pi)
3.14159265359
3.14159265359
```

Les règles de composition applicables aux fonctions internes s'appliquent également aux fonctions définies par le programmeur, donc nous pouvons utiliser tout type d'expression comme argument pour `afficher_deux_fois` :

```
>>> afficher_deux_fois('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> afficher_deux_fois(math.cos(math.pi))
-1.0
-1.0
```

L'argument est évalué avant que la fonction soit appelée, donc dans les exemples précédents, les expressions `'Spam ' * 4` et `math.cos(math.pi)` sont évaluées une seule fois.

Vous pouvez également utiliser une variable comme argument :

```
>>> michael = 'Eric, le demi-abeille.'
```

```
>>> afficher_deux_fois(michael)
Eric, le demi-abeille.
Eric, le demi-abeille.
```

Le nom de la variable que nous passons comme argument (michael) n'a rien à voir avec le nom du paramètre (bruce). Peu importe comment s'appelait la valeur chez elle (dans la fonction appelante) ; ici, dans afficher_deux_fois, nous appelons tout le monde bruce.

3-8 - Les variables et les paramètres sont locaux

Lorsque vous créez une variable dans une fonction, elle est **locale** à cette fonction, ce qui signifie qu'elle existe seulement à l'intérieur de la fonction. Par exemple :

```
def concat_deux_fois(partie1, partie2):
    concat = partie1 + partie2
    afficher_deux_fois(concat)
```

Cette fonction prend deux arguments, les concatène et affiche le résultat deux fois. Voici un exemple qui l'utilise :

```
>>> ligne1 = 'Bing tiddle '
>>> ligne2 = 'tiddle bang.'
>>> concat_deux_fois(ligne1, ligne2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Lorsque concat_deux_fois se termine, la variable concat est détruite. Si nous essayons de l'afficher, nous obtenons une exception :

```
>>> print(concat)
NameError: name 'concat' is not defined
```

Les paramètres aussi sont locaux. Par exemple, en dehors de concat_deux_fois, il n'existe rien qui s'appelle bruce.

3-9 - Diagrammes de pile

Pour garder une trace des variables pouvant être utilisées à tel ou tel endroit, il est parfois utile de dessiner un **diagramme de pile d'exécution**. Comme les diagrammes d'états, les diagrammes de pile montrent la valeur de chaque variable, mais ils montrent également la fonction à laquelle appartient chaque variable.

Chaque fonction est représentée par une **structure de pile**. Une structure de pile est une boîte ayant à côté d'elle un nom de fonction et, à l'intérieur, les paramètres et les variables de la fonction. Le schéma de pile pour l'exemple précédent est illustré sur la figure 3.1.

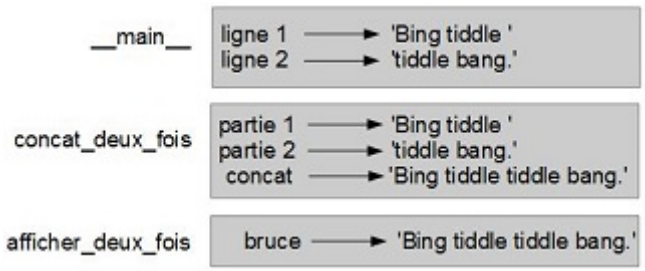


Figure 3.1 : Diagramme de pile.

Ces structures sont disposées en une pile qui indique quelle fonction a appelé quelle fonction, et ainsi de suite. Dans cet exemple, afficher_deux_fois a été appelée par concat_deux_fois et cette dernière a été appelée par __main__,

qui est un nom spécial pour la structure située au sommet de la pile. Lorsque vous créez une variable en dehors de toute fonction, elle appartient à __main__.

Chaque paramètre se réfère à la même valeur que son argument correspondant. Donc, partie1 a la même valeur que ligne1, partie2 a la même valeur que ligne2, et bruce a la même valeur que concat.

Si une erreur se produit lors d'un appel de fonction, Python affiche le nom de la fonction en question, le nom de la fonction appelante, et ainsi de suite, jusqu'à __main__.

Par exemple, si vous essayez d'accéder à concat à l'intérieur de afficher_deux_fois, vous obtenez une NameError :

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    concat_deux_fois(ligne1, ligne2)
  File "test.py", line 5, in concat_deux_fois
    afficher_deux_fois(concat)
  File "test.py", line 9, in afficher_deux_fois
    print(concat)
NameError: name 'concat' is not defined
```

Cette liste de fonctions s'appelle une **trace d'appel**. Elle vous indique dans quel fichier du programme se trouve l'erreur, et à quelle ligne, et quelles fonctions étaient en cours d'exécution au moment où l'erreur est survenue. Elle montre également la ligne de code qui a provoqué l'erreur.

L'ordre des fonctions dans la trace d'appel est le même que l'ordre des structures de pile dans le schéma de la pile. La fonction en cours d'exécution est en bas.

3-10 - Fonctions productives et fonctions vides

Certaines des fonctions que nous avons utilisées, telles que les fonctions du module math, produisent des résultats ; faute d'un meilleur terme, je les appelle **fonctions productives**. D'autres fonctions, comme afficher_deux_fois, effectuent une action, mais ne retournent aucune valeur. On les appelle des **fonctions vides** (*void*, en anglais).

Lorsque vous appelez une fonction productive, vous voulez presque toujours en exploiter le résultat ; par exemple, vous pouvez l'assigner à une variable ou l'utiliser dans le cadre d'une expression :

```
x = math.cos(radians)
nombre_d_or = (math.sqrt(5) + 1) / 2
```

Lorsque vous appelez une fonction en mode interactif, Python affiche le résultat :

```
>>> math.sqrt(5)
2.2360679774997898
```

Mais dans un script, si vous ne faites qu'appeler une fonction productive, la valeur de retour est perdue à jamais !

```
math.sqrt(5)
```

Ce script calcule la racine carrée de 5, mais comme il ne stocke ni n'affiche le résultat, il n'est pas très utile.

Les fonctions *void* peuvent par exemple afficher quelque chose sur l'écran ou avoir un autre effet, mais elles n'ont aucune valeur de retour. Si vous affectez leur résultat à une variable, vous obtenez une valeur spéciale appelée None.

```
>>> resultat = afficher_deux_fois('Bing')
Bing
Bing
>>> print(resultat)
```

```
None
```

La valeur None n'est pas la même chose que la chaîne 'None'. C'est une valeur spéciale, qui a son propre type :

```
>>> print(type(None))
<class 'NoneType'>
```

Les fonctions que nous avons écrites jusqu'à présent sont toutes vides. Nous allons commencer à écrire des fonctions productives d'ici quelques chapitres.

3-11 - Pourquoi des fonctions ?

Les motifs pour lesquels on prend la peine de diviser un programme en fonctions ne sont peut-être pas très clairs. Il existe plusieurs raisons à cela :

- la création d'une nouvelle fonction vous donne la possibilité de nommer un groupe d'instructions, ce qui rend votre programme plus facile à lire et à déboguer ;
- les fonctions peuvent réduire la taille d'un programme en éliminant les répétitions de code. Plus tard, si vous faites une modification, vous aurez à le faire à un seul endroit ;
- diviser un programme long en fonctions vous permet de déboguer les parties une par une, puis de les rassembler en un tout qui fonctionne ;
- les fonctions bien conçues sont souvent utiles pour de nombreux programmes. Une fois que vous en écrivez et déboguez une, vous pouvez la réutiliser.

3-12 - Débogage

L'une des compétences les plus importantes que vous acquerrez est le débogage. Bien qu'il puisse être frustrant, le débogage est une des parties les plus intellectuellement riches, stimulantes et intéressantes de la programmation.

À certains égards, le débogage est comme un travail de détective. Vous vous trouvez en présence d'indices et vous devez déduire les processus et les événements qui ont conduit aux résultats que vous voyez.

Le débogage est aussi une forme de science expérimentale. Une fois que vous avez une idée sur ce qui va mal, vous modifiez votre programme et essayez à nouveau. Si votre hypothèse est correcte, vous pouvez prédire le résultat de la modification et vous faites un pas de plus vers un programme fonctionnel. Si votre hypothèse est fausse, vous devez en formuler une autre. Comme Sherlock Holmes l'a souligné, « Lorsque vous avez éliminé l'impossible, ce qui reste, si improbable soit-il, est nécessairement la vérité. » (A. Conan Doyle, L*e signe des Quatre*)

Pour certaines personnes, la programmation et le débogage sont la même chose. Autrement dit, la programmation est le processus de débogage progressif d'un programme jusqu'à ce qu'il fasse ce que vous voulez. L'idée est que vous devriez commencer avec un programme fonctionnel et lui apporter de petites modifications, en les déboguant au fur et à mesure.

Par exemple, Linux est un système d'exploitation qui contient des millions de lignes de code, mais il a commencé par être un simple programme que Linus Torvalds a utilisé pour explorer la puce Intel 80386. Selon Larry Greenfield, « Un des projets antérieurs de Linus était un programme qui permettrait de basculer entre l'affichage de AAAA et BBBB. Cela a évolué plus tard vers Linux. » (*Le guide des utilisateurs Linux* Version 1 Bêta).

3-13 - Glossaire

- **fonction :** une séquence nommée d'instructions qui effectue une opération utile. Les fonctions peuvent prendre des arguments ou pas et peuvent produire un résultat ou pas.
- **définition de fonction :** une instruction qui crée une nouvelle fonction, en spécifiant son nom, les paramètres et les instructions qu'elle contient.

- **objet fonction :** une valeur créée par une définition de fonction. Le nom de la fonction est une variable qui fait référence à un objet fonction.
- **en-tête :** la première ligne de la définition de fonction.
- **corps :** la séquence d'instructions à l'intérieur de la définition de la fonction.
- **paramètre :** un nom utilisé à l'intérieur d'une fonction pour faire référence à la valeur passée en argument.
- **appel de fonction :** une instruction qui exécute la fonction. Elle se compose du nom de la fonction suivi par une liste éventuelle d'arguments entre parenthèses.
- **argument :** une valeur fournie à une fonction lorsque la fonction est appelée. Cette valeur est affectée au paramètre correspondant à l'intérieur de la fonction.
- **variable locale :** une variable définie à l'intérieur d'une fonction. Une variable locale peut être utilisée uniquement à l'intérieur de la fonction où elle a été définie.
- **valeur de retour :** le résultat retourné par une fonction. Si un appel de fonction est utilisé comme une expression, alors cette expression prend pour valeur la valeur retournée par la fonction.
- **fonction productive :** une fonction qui retourne une valeur.
- **fonction vide :** une fonction qui retourne toujours la valeur `None`.
- `None` **:** une valeur spéciale retournée par les fonctions vides.
- **module :** un fichier qui contient une collection de fonctions apparentées et d'autres définitions.
- **instruction d'importation :** une instruction qui lit un fichier de module et crée un objet module.
- **objet module :** une valeur créée par une instruction `import`, qui permet d'accéder aux valeurs définies dans un module.
- **notation pointée :** la syntaxe permettant d'appeler une fonction définie dans un module, en spécifiant le nom du module suivi par un point et le nom de la fonction.
- **composition :** l'utilisation d'une expression comme partie d'une expression plus grande, ou d'une instruction dans le cadre d'une instruction plus complexe.
- **flux d'exécution :** l'ordre dans lequel les commandes sont exécutées.
- **diagramme de pile :** une représentation graphique d'un empilement de fonctions, leurs variables et les valeurs qu'elles réfèrent.
- **structure de pile :** une boîte dans un diagramme de pile qui représente un appel de fonction. Elle contient les variables locales et les paramètres de la fonction.
- **trace d'appel :** une liste des fonctions en cours d'exécution, affichée lorsqu'une exception se produit.

3-14 - Exercices

Exercice 1

Écrivez une fonction nommée `aligner_a_droite` *qui prend comme paramètre une chaîne nommée* `s` *et affiche cette chaîne avec suffisamment de caractères espace pour que la dernière lettre de la chaîne se trouve dans la colonne 70 de l'écran.*

```
>>> aligner_a_droite('monty')
                                                              monty
```

Indice : utilisez les opérateurs de concaténation et de répétition de chaînes. Notez aussi que Python fournit une fonction interne appelée `len` *, qui renvoie la longueur d'une chaîne, de sorte que la valeur de* `len('monty')` *est 5.*

Exercice 2

Un objet fonction est une valeur que vous pouvez assigner à une variable ou transmettre en tant qu'argument. Par exemple, `appeler_deux_fois` *est une fonction qui prend comme argument un objet fonction et l'appelle deux fois :*

```
def appeler_deux_fois(f):
    f()
    f()
```

Voici un exemple qui utilise `appeler_deux_fois` *pour appeler deux fois une fonction nommée* `afficher_spam` *.*

```
def print_spam():
    print('spam')

appeler_deux_fois(print_spam)
```

1 *Tapez cet exemple dans un script et testez-le.*
2 *Modifiez la fonction* appeler_deux_fois *de sorte qu'elle prenne deux arguments, un objet fonction et une valeur, et appelle la fonction à deux reprises, en passant la valeur comme argument.*
3 *Copiez la définition de la fonction* afficher_deux_fois *utilisée plus tôt dans ce chapitre dans votre script.*
4 *Utilisez la version modifiée de la fonction* appeler_deux_fois *pour appeler à deux reprises la fonction* afficher_deux_fois *deux fois, en passant* 'spam' *comme argument.*
5 *Définissez une nouvelle fonction appelée* appeler_quatre_fois *, qui prend un objet fonction et une valeur et appelle la fonction quatre fois, en passant la valeur comme paramètre. Il ne devrait y avoir que deux instructions dans le corps de cette fonction, et non quatre.*

Solution : **do_four.py**.

Exercice 3

Note : cet exercice devrait être fait en n'utilisant que les instructions et les autres fonctionnalités que nous avons apprises jusqu'ici.

1 *Écrire une fonction qui affiche une grille comme ce qui suit :*

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

Indice : pour imprimer plus d'une valeur sur une ligne, vous pouvez imprimer une séquence de valeurs séparées par des virgules :

```
print('+', '-')
```

Par défaut, print *passe à la ligne suivante, mais vous pouvez modifier ce comportement et mettre un espace à la fin, comme ceci :*

```
print ('+', end = ' ')
print('-')
```

Ces instructions affichent '+ -'.
Une instruction print *sans aucun argument termine la ligne courante et va à la ligne suivante.*
2 *Écrivez une fonction qui affiche une grille similaire avec quatre lignes et quatre colonnes.*

Solution : **grid.py**. *Référence : cet exercice est basé sur un exercice de Oualline,* Programmation pratique en C, troisième édition*, O'Reilly Media 1997.*

4 - Étude de cas : conception d'une interface

Ce chapitre présente une étude de cas qui illustre un processus de conception de fonctions qui travaillent ensemble.

Il présente le module turtle (« tortue »), qui vous permet de créer des images à l'aide des « outils graphiques de type tortue » (tortue graphique de type **Logo** - NdT). Le module turtle est inclus dans la plupart des distributions de Python, mais si vous exécutez Python en utilisant PythonAnywhere, vous ne serez pas en mesure d'exécuter les exemples turtle (ou du moins, ce n'était pas possible au moment où j'ai écrit ce livre).

Si vous avez déjà installé Python sur votre ordinateur, vous devriez être en mesure d'exécuter les exemples. Sinon, c'est le bon moment pour l'installer. J'ai posté des instructions à l'adresse **http://tinyurl.com/thinkpython2e**.

Les exemples de code de ce chapitre sont disponibles sur **polygon.py**.

4-1 - Le module turtle

Pour vérifier si vous disposez du module turtle, ouvrez l'interpréteur Python et tapez

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Créez un fichier nommé monpolygone.py et tapez-y le code suivant :

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

Le module turtle (avec un « t » minuscule) fournit une fonction appelée Turtle (avec un « T » majuscule), qui crée un objet Turtle, que nous affectons à une variable nommée bob. L'affichage de bob donnera quelque chose comme :

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Cela signifie que bob stocke la référence d'un objet de type Turtle, défini dans le module turtle.

mainloop demande à la fenêtre d'attendre que l'utilisateur fasse quelque chose, bien que, dans ce cas, l'utilisateur n'ait pas grand-chose à faire, sinon fermer la fenêtre.

Une fois que vous créez un objet Turtle, vous pouvez appeler une **méthode** pour le déplacer autour de la fenêtre. Une méthode ressemble à une fonction, mais utilise une syntaxe légèrement différente. Par exemple, pour faire avancer la tortue :

```
bob.fd(100)
```

La méthode fd (*forward*) est associée à l'objet Turtle que nous nommons bob. Appeler une méthode, c'est comme faire une demande : vous demandez à bob d'avancer.

L'argument de fd est une distance en pixels, sa taille réelle dépend donc de votre écran.

Vous pouvez appeler d'autres méthodes sur un objet Tortue comme : bk (*backward*) pour reculer, lt (*left turn*) pour tourner à gauche, et rt (*right turn*) pour tourner à droite. L'argument des méthodes lt et rt est un angle en degrés.

En outre, chaque Tortue a un « stylo », qui est descendu ou relevé ; si le stylo est descendu, la Tortue laisse une trace lorsqu'elle se déplace. Les méthodes pu et pd correspondent à *« pen up »* (stylo levé) et *« pen down »* (stylo descendu).

Pour dessiner un angle droit, ajoutez ces lignes au programme (après la création de bob et avant d'appeler mainloop) :

```
bob.fd(100)
```

```
bob.lt(90)
bob.fd(100)
```

Lorsque vous exécutez ce programme, vous devriez voir bob aller vers l'est (la droite), puis vers le nord, laissant derrière deux segments de ligne.

Maintenant, modifiez le programme pour dessiner un carré. Ne poursuivez pas avant d'avoir réussi à le faire !

4-2 - Répétition simple

Vous avez probablement écrit quelque chose comme ceci :

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

Nous pouvons faire la même chose de manière plus concise en utilisant une instruction for. Ajoutez cet exemple à monpolygone.py et exécutez-le à nouveau :

```
for i in range(4):
    print('Hello!')
```

Vous devriez voir quelque chose comme ceci :

```
Hello!
Hello!
Hello!
Hello!
```

C'est l'utilisation la plus simple de l'instruction for ; nous en verrons plus à ce sujet plus tard. Mais cela devrait suffire pour vous permettre de réécrire votre programme de dessin d'un carré. Ne poursuivez pas avant d'avoir réussi à le faire !

Voici une instruction for qui dessine un carré :

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

La syntaxe d'une instruction for ressemble à une définition de fonction. Elle a un en-tête qui se termine par un deux-points et un corps indenté. Le corps peut contenir un nombre quelconque d'instructions.

Une instruction for est également appelée une **boucle**, parce que le flux d'exécution traverse le corps, puis reboucle vers le haut. Dans notre cas, le corps de la boucle est exécuté à quatre reprises.

Cette version du code du dessin du carré est en fait un peu différente de la précédente, car il y a un dernier virage à gauche après avoir dessiné le dernier côté du carré. Ce virage supplémentaire prend plus de temps, mais faire à chaque fois la même chose dans la boucle simplifie le code. Un autre effet de cette version est de laisser la tortue dans la position de départ, orientée dans la direction de départ.

4-3 - Exercices

Ce qui suit est une série d'exercices utilisant TurtleWorld. Ils sont censés être amusants, mais ils ont aussi une raison. Pendant que vous les effectuez, réfléchissez à leur raison.

Les solutions aux exercices se trouvent dans les sections suivantes, alors ne poursuivez pas avant d'avoir fini (ou au moins essayé).

1. Écrivez une fonction appelée carre qui prend un paramètre nommé t, qui est une tortue. Elle doit utiliser la tortue pour dessiner un carré.
   Écrivez un appel de fonction qui passe bob comme argument à carre, puis exécutez à nouveau le programme.
2. Ajoutez à carre un deuxième paramètre, nommé longueur. Modifiez le corps de la fonction afin que la longueur des côtés soit longueur, puis modifiez l'appel de la fonction pour fournir un deuxième argument. Exécutez à nouveau le programme. Testez votre programme avec une plage de valeurs pour longueur.
3. Faites une copie de carre et changez son nom en polygone. Ajoutez-lui un autre paramètre, nommé n, et modifiez-en le corps de sorte qu'il dessine un polygone régulier à n côtés. Indice : Les angles extérieurs d'un polygone régulier à n côtés ont 360/n degrés.
4. Écrivez une fonction appelée cercle, qui prend comme paramètres une tortue t, et le rayon r, et qui dessine un cercle approximatif en appelant polygone avec une longueur et un nombre de côtés approprié. Testez votre fonction avec une gamme de valeurs de r.
   Indice : calculez la circonférence du cercle et assurez-vous que longueur * n = circonference.
5. Faites une version plus générale de cercle, appelée arc, qui prend un paramètre supplémentaire angle, qui détermine quelle fraction d'un cercle dessiner. angle est mesuré en degrés, donc lorsque angle = 360, arc doit dessiner un cercle complet.

4-4 - Encapsulation

Le premier exercice vous demande de mettre votre code qui dessine un carré dans une définition de fonction et ensuite appeler la fonction, en passant la tortue comme paramètre. Voici une solution :

```
def carre(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

carre(bob)
```

Les instructions les plus intérieures, fd et lt, sont indentées deux fois pour montrer qu'elles sont à l'intérieur de la boucle for, qui se trouve à l'intérieur de la définition de fonction. La ligne suivante, carre(bob), est alignée sur la marge gauche, ce qui indique à la fois la fin de la boucle et de la définition de fonction.

À l'intérieur de la fonction, t est la référence de la même tortue bob, de sorte que t.lt(90) a le même effet que bob.lt(90). Dans ce cas, pourquoi ne pas appeler ce paramètre bob ? L'idée est que t peut être n'importe quelle tortue, pas seulement bob, donc vous pouvez créer une deuxième tortue et la passer comme argument à carre :

```
alice = Turtle()
carre(alice)
```

Envelopper un morceau de code dans une fonction s'appelle l'**encapsulation**. L'un des avantages de l'encapsulation est qu'elle attache un nom au code, qui sert comme une sorte de documentation. Un autre avantage est que si vous réutilisez le code, il est plus concis d'appeler une fonction deux fois que de copier et coller le corps !

4-5 - Généralisation

L'étape suivante consiste à ajouter un paramètre longueur à carre. Voici une solution :

```
def carre(t, longueur):
    for i in range(4):
        t.fd(longueur)
        t.lt(90)

carre(bob, 100)
```

L'ajout d'un paramètre à une fonction s'appelle une **généralisation**, car il rend la fonction plus générale : dans la version précédente, le carré a toujours la même taille ; dans cette nouvelle version, il peut avoir n'importe quelle taille.

L'étape suivante est également une généralisation. Au lieu de dessiner des carrés, polygone dessine des polygones réguliers ayant un nombre quelconque de côtés. Voici une solution :

```
def polygone(t, n, longueur):
    angle = 360 / n
    for i in range(n):
        t.fd(longueur)
        t.lt(angle)

polygone(bob, 7, 70)
```

Cet exemple dessine un polygone à sept côtés, avec une longueur de côté de 70.

Si vous utilisez Python 2, la valeur de angle peut être incorrecte à cause de la division entière. Une solution simple consiste à calculer angle = 360.0 / n. Comme le numérateur est un nombre à virgule flottante, le résultat est en virgule flottante.

Lorsqu'une fonction a plus que quelques arguments numériques, il est facile d'oublier ce qu'ils sont, ou l'ordre dans lequel ils devraient être. Dans ce cas, c'est souvent une bonne idée d'inclure les noms des paramètres dans la liste d'arguments :

```
polygone(bob, n=7, longueur=70)
```

Ceux-ci sont appelés **arguments nommés** ou **arguments mot-clé**, car ils incluent les noms de paramètres comme « mots-clés » (à ne pas les confondre avec les mots-clés Python comme while et def).

Cette syntaxe rend le programme plus lisible. Elle représente aussi un rappel de la façon dont les arguments et les paramètres fonctionnent : lorsque vous appelez une fonction, les arguments sont affectés aux paramètres.

4-6 - Conception d'une interface

L'étape suivante consiste à écrire cercle, qui prend comme un paramètre un rayon r. Voici une solution simple, qui utilise polygone pour dessiner un polygone à 50 côtés :

```
import math

def cercle(t, r):
    circonference = 2 * math.pi * r
    n = 50
    longueur = circonference / n
    polygone(t, n, longueur)
```

La première ligne calcule la circonférence d'un cercle de rayon r en utilisant la formule 2 π r. Puisque nous utilisons math.pi, nous devons importer math. Par convention, les instructions import sont généralement au début du script.

n est le nombre de segments de ligne dans notre approximation d'un cercle, donc longueur est la longueur de chaque segment. Ainsi, polygone dessine un polygone à 50 côtés qui sera la représentation approximative d'un cercle de rayon r.

Une limitation de cette solution est que n est une constante, ce qui signifie que pour de très grands cercles, les segments de ligne sont trop longs, et pour les petits cercles, nous gaspillons du temps à dessiner de très petits segments. Une solution serait de généraliser la fonction en prenant comme paramètre n. Cela donnerait à l'utilisateur (celui qui appelle cercle) plus de contrôle, mais l'interface serait moins propre.

L'**interface** d'une fonction est un résumé de la façon dont elle est utilisée : quels sont les paramètres ? Qu'est-ce que la fonction fait ? Et quelle est la valeur de retour ? Une interface est « propre » si elle permet à l'appelant de faire ce qu'il veut, sans se préoccuper des détails inutiles.

Dans cet exemple, r appartient à l'interface, car il spécifie le rayon du cercle à dessiner. n est moins approprié, car il concerne les détails de la *façon* dont le cercle doit être dessiné.

Plutôt que d'encombrer l'interface, il est préférable de choisir une valeur appropriée de n en fonction de circonference :

```
def cercle(t, r):
    circonference = 2 * math.pi * r
    n = int(circonference / 3) + 1
    longueur = circonference / n
    polygone(t, n, longueur)
```

Maintenant, le nombre de segments est un nombre entier près de circonference / 3, donc la longueur de chaque segment est d'environ 3, assez petite pour que les cercles aient l'air bien circulaires, mais assez grande pour être efficace et acceptable pour des cercles de toutes tailles.

4-7 - Réusinage

Lorsque je l'ai écrit cercle, je pouvais réutiliser polygone, car un polygone à nombreux côtés est une bonne approximation d'un cercle. Mais arc est plus récalcitrant : nous ne pouvons pas utiliser polygone ou cercle pour dessiner un arc.

Une solution possible est de commencer avec une copie de polygone et la transformer en arc. Le résultat pourrait ressembler à ceci :

```
def arc(t, r, angle):
    longueur_arc = 2 * math.pi * r * angle / 360
    n = int(longueur_arc / 3) + 1
    longueur_etape = longueur_arc / n
    angle_etape = angle / n

    for i in range(n):
        t.fd(longueur_etape)
        t.lt(angle_etape)
```

La seconde moitié de cette fonction ressemble à polygone, mais nous ne pouvons pas réutiliser polygone sans modifier l'interface. Nous pourrions généraliser polygone pour qu'elle prenne un angle comme troisième argument, mais alors polygone ne serait plus un nom approprié ! Au lieu de cela, nous allons appeler cette fonction plus générale polyligne :

```
def polyligne(t, n, longueur, angle):
    for i in range(n):
        t.fd(longueur)
        t.lt(angle)
```

Maintenant, nous pouvons réécrire polygone et arc pour utiliser polyligne :

```
def polygone(t, n, longueur):
    angle = 360.0 / n
    polyligne(t, n, longueur, angle)
```

```
def arc(t, r, angle):
    longueur_arc = 2 * math.pi * r * angle / 360
    n = int(longueur_arc / 3) + 1
    longueur_etape = longueur_arc / n
    angle_etape = float(angle) / n
    polyligne(t, n, longueur_etape, angle_etape)
```

Enfin, nous pouvons réécrire cercle pour utiliser arc :

```
def cercle(t, r):
    arc(t, r, 360)
```

Ce processus - le réarrangement d'un programme visant à améliorer les interfaces et à faciliter la réutilisation du code - est appelé réusinage (ou parfois refactorisation). Dans ce cas, nous avons remarqué qu'il y avait un code similaire tant en arc qu'en polygone, alors nous l'avons déplacé dans polyligne.

Si nous avions planifié dès le début, nous aurions peut-être écrit d'abord polyligne et évité le réusinage, mais souvent vous n'en savez pas assez au début d'un projet pour concevoir toutes les interfaces. Une fois que vous commencez à coder, vous comprenez mieux le problème. Parfois, le réusinage est un signe que vous avez appris quelque chose.

4-8 - Un plan de développement

Un **plan de développement** est un processus pour écrire des programmes. Le processus que nous avons utilisé dans cette étude de cas est « l'encapsulation et la généralisation ». Les étapes de ce processus sont :

1. Commencer par écrire un petit programme sans aucune définition de fonction ;
2. Une fois que votre programme fonctionne, identifier un composant cohérent, l'encapsuler dans une fonction et lui donner un nom ;
3. Généraliser la fonction en ajoutant des paramètres appropriés ;
4. Répéter les étapes 1-3 jusqu'au moment où vous avez un ensemble de fonctions qui font ce que vous souhaitez. Copier et coller le code fonctionnel pour éviter de retaper (et de déboguer à nouveau) ;
5. Chercher des opportunités d'améliorer le programme par réusinage. Par exemple, si vous avez un code similaire à plusieurs endroits, envisagez de le factoriser dans une fonction générale appropriée.

Ce processus a quelques inconvénients - nous allons voir d'autres options plus tard -, mais il peut être utile si vous ne savez pas à l'avance comment diviser le programme en fonctions. Cette approche vous permet de concevoir pendant que vous avancez.

4-9 - docstring

Une **docstring** est un texte au début d'une fonction qui explique l'interface (« doc » est l'abréviation de « documentation »). Voici un exemple :

```
def polyligne(t, n, longueur, angle):
    """Dessine n segments de ligne ayant la longueur et l'angle
    entre eux (en degrés) donnés. t est une tortue.
    """
    for i in range(n):
        t.fd(longueur)
        t.lt(angle)
```

Par convention, toutes les docstrings sont des chaînes entourées par des guillemets triples, également connues comme des chaînes multilignes, parce que les guillemets triples permettent l'écriture de la chaîne sur plusieurs lignes.

La docstring ci-dessus est laconique, mais contient les informations essentielles dont quelqu'un aurait besoin pour utiliser cette fonction. Elle explique de façon concise ce que fait la fonction (sans entrer dans les détails de comment

elle le fait). Elle explique l'effet de chaque paramètre sur le comportement de la fonction et de quel type doit être chaque paramètre (si cela n'est pas évident).

La rédaction de ce type de documentation est une partie importante de la conception de l'interface. Une interface bien conçue doit être simple à expliquer ; si vous avez du mal à expliquer l'une de vos fonctions, peut-être l'interface devrait-elle être améliorée.

4-10 - Débogage

Une interface est comme un contrat entre une fonction et un appelant. L'appelant accepte de fournir certains paramètres et la fonction accepte de faire un certain travail.

Par exemple, polyligne nécessite quatre arguments : t doit être une tortue, n doit être un nombre entier, longueur doit être un nombre positif et angle doit être un nombre qui peut être compris comme étant en degrés.

Ces exigences sont appelées **préconditions**, parce qu'elles sont censées être vraies avant que la fonction commence à être exécutée. Inversement, les conditions à la fin de la fonction sont des **postconditions**. Les postconditions incluent l'effet escompté de la fonction (comme le dessin des segments de ligne) et des effets secondaires (comme déplacer la tortue ou d'autres changements).

Les préconditions ou conditions préalables sont à la charge de l'appelant. Si l'appelant viole une précondition (correctement documentée !) et la fonction ne marche pas correctement, le bogue est dans l'appelant, et pas dans la fonction.

Si les préconditions sont satisfaites et les postconditions ne le sont pas, le bogue est dans la fonction. Si vos préconditions et vos postconditions sont claires, elles peuvent aider au débogage.

4-11 - Glossaire

- **méthode :** une fonction qui est associée à un objet et appelée en utilisant la notation pointée.
- **boucle :** une partie d'un programme qui peut être exécutée à plusieurs reprises.
- **encapsulation :** le processus de transformation d'une séquence d'instructions en une définition de fonction.
- **généralisation :** le processus qui consiste à remplacer quelque chose d'inutilement spécifique (comme un nombre) par quelque chose de convenablement général (comme une variable ou un paramètre).
- **argument mot-clé :** un argument qui inclut le nom du paramètre comme un « mot-clé ».
- **interface :** une description de la façon d'utiliser une fonction, y compris les noms et les descriptions des arguments et la valeur de retour.
- **réusinage :** le processus de modification d'un programme qui fonctionne pour améliorer les interfaces des fonctions et d'autres qualités du code.
- **plan de développement :** un procédé pour l'écriture de programmes.
- **docstring :** une chaîne qui apparaît au début d'une définition de fonction afin de documenter l'interface de la fonction.
- **précondition (condition préalable) :** une exigence qui doit être satisfaite par l'appelant avant l'exécution d'une fonction.
- **postcondition :** une exigence qui doit être satisfaite par la fonction avant sa fin.

4-12 - Exercices

Exercice 1

Téléchargez le code de ce chapitre à partir de **polygon.py**.

1 *Dessinez un diagramme de pile qui montre l'état du programme lors de l'exécution de* `circle(bob, radius)`. *Vous pouvez faire le calcul à la main ou ajouter des instructions* `print` *au code.*

2 *La version de* `arc` *dans la Section* **4.7.** *n'est pas très précise parce que l'approximation linéaire du cercle est toujours en dehors du vrai cercle. En conséquence, la tortue s'arrête à une distance de quelques pixels de la destination correcte. Ma solution montre un moyen de réduire l'effet de cette erreur. Lisez le code et voyez si cela a du sens pour vous. Si vous dessinez un diagramme, cela pourra vous aider à voir comment cela fonctionne.*

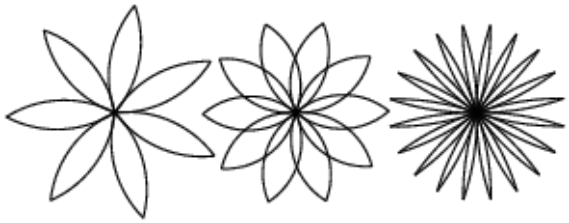


Figure 4.1 : Des fleurs dessinées par une tortue.

Exercice 2

Écrivez un ensemble convenablement général de fonctions qui peuvent dessiner des fleurs comme dans la Figure 4.1.

Solution: **flower.py**, *exige également* **polygon.py**.

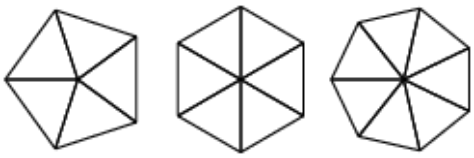


Figure 4.2 : Des tartes dessinées par une tortue.

Exercice 3

Écrivez un ensemble convenablement général de fonctions qui peuvent dessiner des formes comme dans la Figure 4.2.

Exercice 4

Les lettres de l'alphabet peuvent être construites à partir d'un nombre modéré d'éléments de base, comme les lignes verticales et horizontales et quelques courbes. Concevez un alphabet qui peut être dessiné avec un nombre minimal d'éléments de base et ensuite écrivez les fonctions qui dessinent les lettres.

Vous devriez écrire une fonction pour chaque lettre, ayant les noms `dessiner_a` , `dessiner_b` *, etc., et mettre vos fonctions dans un fichier nommé* `letters.py` *. Vous pouvez télécharger une « tortue - machine à écrire » à* **typewriter.py** *pour vous aider à tester votre code.*

Vous pouvez obtenir une solution à l'adresse **letters.py** *; elle exige aussi* **polygon.py**.

Exercice 5

Lisez à propos de spirales à l'adresse **https://fr.wikipedia.org/wiki/Spirale** (voir aussi **http://en.wikipedia.org/wiki/Spiral**) *; ensuite, écrivez un programme qui dessine une spirale d'Archimède (ou l'un des autres types de spirales). Solution:* **spiral.py**.

5 - Structures conditionnelles et récursion

Le sujet principal de ce chapitre est l'instruction if, qui exécute un code différent en fonction de l'état du programme. Mais d'abord, je veux vous présenter deux nouveaux opérateurs : la division entière et le modulo.

5-1 - Division entière et modulo

L'opérateur de **division entière**, //, divise un nombre par un autre et arrondit le résultat à l'entier juste en dessous. Par exemple, supposons que la durée d'un film est de 105 minutes. Vous voudrez peut-être savoir combien de temps cela fait en heures. La division classique retourne un nombre à virgule flottante :

```
>>> minutes = 105
>>> minutes / 60
1.75
```

Mais, généralement nous n'écrivons pas les heures avec décimales. La division entière renvoie le nombre entier d'heures, tronquant la partie fractionnaire :

```
>>> minutes = 105
>>> heures = minutes // 60
>>> heures
1
```

Pour obtenir le reste, vous devez soustraire une heure du total des minutes :

```
>>> reste = minutes - heures * 60
>>> reste
45
```

Une autre possibilité consiste à utiliser l'**opérateur modulo**, %, qui effectue la division et renvoie le reste.

```
>>> reste = minutes % 60
>>> reste
45
```

L'opérateur modulo est plus utile qu'il n'y paraît. Par exemple, vous pouvez vérifier si un nombre est divisible par un autre : si x % y est égal à zéro, alors x est divisible par y.

Vous pouvez aussi extraire les chiffres ou le chiffre le plus à droite d'un nombre. Par exemple, x % 10 donne le chiffre le plus à droite de x (en base 10). De même, x % 100 donne les deux derniers chiffres.

Si vous utilisez Python 2, la division fonctionne différemment. L'opérateur de division, /, effectue la division entière si les deux opérandes sont des nombres entiers, et la division décimale si l'un au moins des opérandes est un float.

5-2 - Expressions booléennes

Une **expression booléenne** est une expression qui est soit vraie, soit fausse. Les exemples suivants utilisent l'opérateur ==, qui compare deux opérandes et produit True (« vrai ») s'ils sont égaux et False (« faux ») sinon :

```
>>> 5 == 5
True
>>> 5 == 6
False
```

True et False sont des valeurs particulières qui appartiennent au type bool ; ce ne sont pas des chaînes de caractères :

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

L'opérateur == est l'un des **opérateurs relationnels** ; les autres sont :

```
x != y               # x n'est pas égal à y
x > y                # x est supérieur à y
x < y                # x est inférieur à y
x >= y               # x est supérieur ou égal à y
x <= y               # x est inférieur ou égal à y
```

Bien que ces opérations vous soient probablement familières, les symboles de Python sont différents des symboles mathématiques. Une erreur courante consiste à utiliser un seul signe égal (=) au lieu d'un signe égal double (==). Rappelez-vous que = est un opérateur d'affectation et == est un opérateur relationnel. Il n'existe pas d'opérateurs tels que =< ou =>.

5-3 - Opérateurs logiques

Il y a trois **opérateurs logiques** : and, or, et not. La sémantique (le sens) de ces opérateurs est similaire à leur signification en anglais : « et », « ou » et respectivement « non ». Par exemple, l'expression x > 0 and x < 10 est vraie seulement si x est supérieur à 0 *et* inférieur à 10.

L'expression n % 2 == 0 or n % 3 == 0 est vraie si *l'une ou les deux* conditions est remplie, à savoir, si le nombre n est divisible par 2 ou par 3.

Enfin, l'opérateur not nie une expression booléenne, donc l'expression not (x > y) est vraie si x > y est fausse, c'est-à-dire, si x est inférieur ou égal à y.

Au sens strict, les opérandes des opérateurs logiques devraient être des expressions booléennes, mais Python n'est pas très strict. Tout nombre différent de zéro est interprété comme True :

```
>>> 42 and True
True
```

Cette souplesse peut être utile, mais il y a quelques subtilités qui pourraient être source de confusion. Vous avez peut-être intérêt à l'éviter (sauf si vous savez ce que vous faites).

5-4 - Exécution conditionnelle

Pour pouvoir écrire des programmes utiles, nous devons presque toujours vérifier les conditions et modifier le comportement du programme en conséquence. Les **instructions conditionnelles** nous donnent cette possibilité. La forme la plus simple est l'instruction if :

```
if x > 0:
    print('x est positif')
```

L'expression booléenne après if est appelée **condition**. Si elle est vraie, l'instruction indentée est exécutée. Sinon, rien ne se passe.

Les instructions if ont la même structure que les définitions de fonctions : un en-tête, suivi d'un corps indenté. Ce genre d'instructions sont appelées **instructions composées**.

Il n'y a aucune limite sur le nombre d'instructions qui peuvent apparaître dans le corps, mais il doit y en avoir au moins une. Parfois, il est utile d'avoir un corps sans instructions (généralement, pour réserver de la place pour du code que vous n'avez pas encore écrit). Dans ce cas, vous pouvez utiliser l'instruction pass, qui ne fait rien.

```
if x < 0:
    pass          # TODO: il faut gérer les valeurs négatives !
```

5-5 - Exécution alternative

Une deuxième forme de l'instruction if est l'« exécution alternative », dans laquelle il y a deux possibilités et la condition détermine laquelle est exécutée. La syntaxe ressemble à ceci :

```
if x % 2 == 0:
    print('x est pair')
else:
    print('x est impair')
```

Si le reste de la division de x par 2 est égal à 0, alors nous savons que x est pair, et le programme affiche un message approprié. Si la condition est fausse, la deuxième série d'instructions est exécutée. Puisque la condition doit être soit vraie, soit fausse, une seule des branches de l'alternative sera exécutée. Les alternatives sont appelées **branchement**, parce qu'elles sont des branchements dans le flux d'exécution.

5-6 - Conditions enchaînées

Parfois, il y a plus de deux possibilités et nous avons besoin de plus de deux branches. Une façon d'exprimer ce genre de calcul est un **enchaînement de conditions** :

```
if x < y:
    print('x plus petit que y')
elif x > y:
    print('x plus grand que y')
else:
    print('x et y sont égaux')
```

elif est une abréviation de « else if ». Ici encore, exactement une seule branche sera exécutée. Il n'y a aucune limite sur le nombre d'instructions elif. S'il existe une clause else, elle doit être à la fin, mais elle n'est pas obligatoire.

```
if choix == 'a':
    dessiner_a()
elif choix == 'b':
    dessiner_b()
elif choix == 'c':
    dessiner_c()
```

Chaque condition est vérifiée dans l'ordre. Si la première est fausse, la prochaine est vérifiée, et ainsi de suite. Si l'une d'entre elles est vraie, la branche correspondante est exécutée et l'instruction se termine. Même si plusieurs conditions sont vraies, seule la branche correspondant à la première condition vraie est exécutée.

5-7 - Conditions imbriquées

Les structures conditionnelles peuvent également être imbriquées les unes dans les autres. Nous aurions pu écrire l'exemple de la section précédente comme ceci :

```
if x == y:
    print('x et y sont égaux')
else:
    if x < y:
        print('x est plus petit que y')
```

```
    else:
        print('x est plus grand que y')
```

La structure conditionnelle externe contient deux branches. La première branche contient une simple instruction. La seconde branche contient une autre instruction if, qui a deux branches à son tour. Ces deux branches sont de simples instructions, même si elles aussi auraient pu être des instructions conditionnelles.

Bien que l'indentation des instructions rende la structure apparente, les **conditions imbriquées** deviennent très rapidement difficiles à lire. Il est judicieux de les éviter quand c'est possible.

Les opérateurs logiques offrent souvent un moyen de simplifier les instructions conditionnelles imbriquées. Par exemple, le code suivant :

```
if 0 < x:
    if x < 10:
        print("x est un nombre positif d'un seul chiffre.")
```

peut se réécrire en utilisant une seule condition. L'instruction print est exécutée uniquement si les deux conditions sont vraies. Donc, nous pouvons obtenir le même effet en utilisant l'opérateur and :

```
if 0 < x and x < 10:
    print("x est un nombre positif d'un seul chiffre.")
```

Pour ce genre de condition, Python fournit une option plus concise :

```
if 0 < x < 10:
    print("x est un nombre positif d'un seul chiffre.")
```

5-8 - Récursion

Une fonction peut en appeler une autre ; il est également autorisé qu'une fonction s'appelle elle-même. Les avantages de ce genre de construction ne sont peut-être pas évidents, mais cela se révèle être l'une des choses les plus magiques qu'un programme puisse faire. Par exemple, considérez la fonction suivante :

```
def compte_a_rebours(n):
    if n <= 0:
        print('Décollez !')
    else:
        print(n)
        compte_a_rebours(n-1)
```

Si n est 0 ou négatif, elle affiche le mot « Décollez ! ». Sinon, elle affiche n et ensuite appelle une fonction nommée compte_a_rebours, elle-même, en passant n-1 comme argument.

Qu'advient-il si nous appelons cette fonction ainsi ?

```
>>> compte_a_rebours(3)
```

L'exécution de compte_a_rebours commence avec n = 3. Puisque n est supérieur à 0, elle affiche la valeur 3, puis s'appelle elle-même...

- L'exécution de compte_a_rebours commence avec n = 2, et puisque n est supérieur à 0, elle affiche la valeur 2, puis s'appelle elle-même...
    - L'exécution de compte_a_rebours commence avec n = 1, et puisque n est supérieur à 0, elle affiche la valeur 1, puis s'appelle elle-même...

- L'exécution de compte_a_rebours commence avec n = 0, et puisque n n'est pas supérieur à 0, elle affiche le mot « Décollez ! », puis retourne.
- La fonction compte_a_rebours qui a reçu la valeur n = 1 retourne.
- La fonction compte_a_rebours qui a reçu la valeur n = 2 retourne.

La fonction compte_a_rebours qui a reçu la valeur n = 3 retourne.

Vous êtes maintenant de retour dans __main__. En définitive, l'affichage ressemble à ceci :

```
3
2
1
Décollez !
```

Une fonction qui s'appelle elle-même est dite **récursive** ; le processus de son exécution est appelé **récursivité**.

Voici un autre exemple d'une fonction qui affiche une chaîne de caractères n fois.

```
def afficher_n(chaine, n):
    if n <= 0:
        return
    print(chaine)
    afficher_n(chaine, n-1)
```

Si n <= 0, l'**instruction return** fait sortir de la fonction. Le flux d'exécution retourne immédiatement à l'appelant, et les lignes suivantes de la fonction ne sont pas exécutées.

Le reste de la fonction est similaire à compte_a_rebours : elle affiche chaine et s'appelle elle-même pour afficher chaine encore n-1 fois. Ainsi, le nombre de lignes en sortie est 1 + (n - 1), ce qui donne n.

Pour des exemples simples de ce genre, il est probablement plus facile d'utiliser une boucle for. Mais nous verrons plus loin des exemples qui sont difficiles à écrire avec une boucle for et faciles à écrire avec la récursivité, il est donc bien de commencer tôt.

5-9 - Diagrammes de pile pour les fonctions récursives

Dans la section **3.9**, nous avons utilisé un diagramme de pile pour représenter l'état d'un programme au cours d'un appel de fonction. Le même type de diagramme peut aider à interpréter une fonction récursive.

Chaque fois qu'une fonction est appelée, Python crée une structure de pile pour contenir les variables et les paramètres locaux de la fonction. Pour une fonction récursive, il se peut qu'il y ait plus d'une structure sur la pile en même temps.

La figure 5.1 représente un diagramme de pile pour compte_a_rebours appelée avec n = 3.

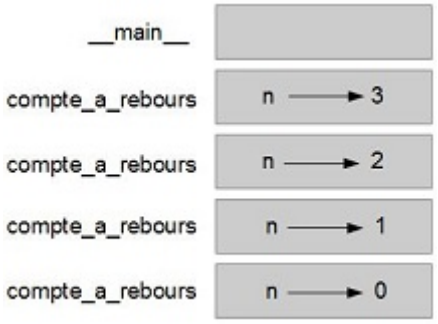


Figure 5.1 : Diagramme de pile.

Comme d'habitude, le haut de la pile est la structure pour `__main__`. Il est vide parce que nous n'avons créé aucune variable dans `__main__` et ne lui avons passé aucun argument.

Les quatre structures de `compte_a_rebours` ont des valeurs différentes pour le paramètre `n`. Le bas de la pile, là où `n = 0`, est appelé le **cas de base**. Il ne fait aucun appel récursif, donc il n'y a aucune autre structure après lui.

À titre d'exercice, dessinez un diagramme de pile pour `afficher_n`, appelée avec `chaine = 'Hello'` et `n = 2`. Ensuite, écrivez une fonction appelée `faire_n`, qui prend comme arguments un objet fonction et un nombre `n`, et qui appelle `n` fois la fonction passée en argument.

5-10 - Récursion infinie

Si une cascade d'appels récursifs n'atteint jamais un cas de base, les appels récursifs se poursuivent pour l'éternité, et le programme ne se termine jamais. C'est ce que l'on appelle une **récursion infinie**, et ce n'est généralement pas une bonne idée. Voici un programme minimal avec une récursion infinie :

```
def recurse():
    recurse()
```

Dans la plupart des environnements de programmation, un programme avec une récursion infinie ne s'exécutera pas tout à fait éternellement. Python affiche un message d'erreur lorsque la profondeur maximale de récursivité est atteinte :

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                .
                .
                .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

Cette trace d'appel est un peu plus longue que celle que nous avons vue dans le chapitre précédent. Lorsque l'erreur se produit, il y a 1000 structures récursives sur la pile !

Si votre code génère par accident une récursion infinie, examinez votre fonction pour confirmer qu'il y a un scénario de base qui ne fait aucun appel récursif. Et s'il y a un cas de base, vérifiez si vous avez la garantie de l'atteindre.

5-11 - Saisie au clavier

Les programmes que nous avons écrits jusqu'ici n'acceptent aucune intervention de l'utilisateur. Ils font juste la même chose chaque fois.

Python fournit une fonction interne appelée input qui suspend l'exécution d'un programme et attend que l'utilisateur tape quelque chose au clavier. Lorsque l'utilisateur appuie sur la touche Retour ou Entrée, l'exécution du programme reprend et input renvoie une chaîne contenant ce que l'utilisateur a tapé. En Python 2, la même fonction est appelée raw_input.

```
>>> text = input()
Qu'est-ce que tu attends ?
>>> text
Qu'est-ce que tu attends ?
```

Avant d'obtenir les données saisies par l'utilisateur, il est souhaitable d'afficher une invite indiquant à l'utilisateur que taper. input peut prendre comme argument une invite :

```
>>> nom = input('Quel...est votre nom ?\n')
Quel...est votre nom ?
Arthur, Roi des Bretons !
>>> nom
Arthur, Roi des Bretons !
```

La séquence \n à la fin de l'invite représente un **saut de ligne**, qui est un caractère spécial qui provoque un passage à la ligne. Voilà pourquoi l'entrée de l'utilisateur apparaît à la ligne, en dessous de l'invite.

Si vous vous attendez à ce que l'utilisateur tape un nombre entier, vous pouvez essayer de convertir la valeur de retour en int :

```
>>> invite = 'À quelle vitesse vole une hirondelle égarée ?\n'
>>> vitesse = input(invite)
À quelle vitesse vole une hirondelle égarée ?
42
>>> int(vitesse)
42
```

Mais si l'utilisateur tape autre chose qu'une chaîne de chiffres, vous obtenez une erreur :

```
>>> vitesse = input(invite)
À quelle vitesse vole une hirondelle égarée ?
Vous voulez dire une hirondelle africaine ou européenne ?
>>> int(vitesse)
ValueError: invalid literal for int() with base 10
```

Nous verrons plus loin comment gérer ce genre d'erreur.

5-12 - Débogage

Lorsqu'une erreur de syntaxe ou d'exécution se produit, le message d'erreur contient beaucoup d'informations, mais on peut se retrouver dépassé. Les parties les plus utiles sont généralement :

- le genre d'erreur et
- l'endroit où elle est survenue.

Les erreurs de syntaxe sont généralement faciles à trouver, mais il y a quelques astuces. Les erreurs d'espace blanc peuvent être difficiles à repérer parce que les espaces et tabulations sont invisibles et nous sommes habitués à les ignorer.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

Dans cet exemple, le problème est que la deuxième ligne est en retrait d'un espace. Mais le message d'erreur indique y, ce qui est trompeur. En général, les messages d'erreur indiquent où le problème a été découvert, mais l'erreur réelle peut parfois se trouver plus haut dans le code, parfois sur une ligne précédente.

La même chose s'applique aux anomalies d'exécution. Supposons que vous essayez de calculer un rapport signal / bruit en décibels. La formule est *Rapport signal/bruit = 10 log_{10} (Psignal / Pbruit)*. En Python, vous pourriez essayer d'écrire quelque chose comme ceci :

```
import math
puissance_signal = 9
puissance_bruit = 10
ratio = puissance_signal // puissance_bruit
decibels = 10 * math.log10(ratio)
print(decibels)
```

Lorsque vous exécutez ce programme, vous obtenez une erreur :

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

Le message indique une erreur ligne 5, mais il n'y a aucune erreur sur cette ligne. Pour trouver l'erreur réelle, il pourrait être utile d'afficher la valeur de ratio, qui se révèle être 0. Le problème est sur la ligne 4, où l'on utilise la division entière au lieu de la division en virgule flottante. Du coup, ratio est nul (au lieu de valoir 0,9), et l'erreur survient parce que la fonction logarithme n'est pas définie pour la valeur 0.

Vous devez prendre le temps de lire attentivement les messages d'erreur, mais ne présumez pas que tout ce qu'ils disent est correct.

5-13 - Glossaire

- **division entière :** un opérateur, noté //, qui divise un nombre par un autre et arrondit le résultat vers le bas (vers zéro) à un nombre entier.
- **opérateur modulo :** un opérateur, noté avec un signe pour cent (%), qui fonctionne sur les nombres entiers et renvoie le reste de la division d'un nombre par un autre.
- **expression booléenne :** une expression dont la valeur est soit True, soit False.
- **opérateur relationnel :** un opérateur qui compare ses opérandes, ==, !=, >, <, >= et <=.
- **opérateur logique :** un opérateur qui combine des expressions booléennes and, or, et not.
- **instruction conditionnelle :** une instruction qui contrôle le flux d'exécution en fonction de certaines conditions.
- **condition :** l'expression booléenne dans une instruction conditionnelle, qui détermine quelle branche s'exécute.
- **instruction composée :** une instruction qui se compose d'une tête et d'un corps. L'en-tête se termine par deux points (:). Le corps est en retrait par rapport à l'en-tête.
- **branche :** une des séquences successives d'instructions d'une instruction conditionnelle.
- **enchaînement de conditions :** une instruction conditionnelle avec une série de branches successives.
- **instruction conditionnelle imbriquée :** une instruction conditionnelle qui apparaît dans l'une des branches d'une autre instruction conditionnelle.
- **instruction return :** une instruction qui provoque l'arrêt immédiat d'une fonction et le retour à la fonction appelante.
- **récursivité :** le fait d'appeler la fonction qui est en cours d'exécution.
- **cas de base :** une branche conditionnelle dans une fonction récursive qui ne fait pas un appel récursif.
- **récursion infinie :** une récursion qui ne dispose pas d'un cas de base, ou ne l'atteint jamais. En fin de compte, une récursion infinie provoque une erreur d'exécution.

5-14 - Exercices

Exercice 1

Le module `time` *fournit une fonction, appelée également* `time` *, qui renvoie l'heure GMT par rapport à « l'époque » (epoch), qui est une date arbitraire utilisée comme un point de référence. Sur les systèmes UNIX, epoch est le 1er janvier 1970. Autrement dit, la fonction* `time` *renvoie donc à un instant donné le nombre de secondes écoulées depuis l'époque.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

Écrivez un script qui lit l'heure actuelle et la convertit en un moment de la journée en heures, minutes et secondes, plus le nombre de jours depuis l'époque.

Exercice 2

Le dernier théorème de Fermat dit qu'il n'existe pas de nombres entiers positifs a, b, *et* c *tels que*

$$a^n + b^n = c^n$$

pour toute valeur de `n` *supérieure à 2.*

1. *Écrivez une fonction nommée* `verifie_fermat` *qui prend quatre paramètres -* `a` *,* `b` *,* `c` *et* `n` *- et vérifie si le théorème de Fermat se vérifie pour ces valeurs. Si* `n` *est supérieur à 2 et*

   $$a^n + b^n = c^n$$

   le programme doit afficher, « Bon sang, Fermat avait tort ! ». Sinon, le programme doit afficher « Non, cela ne marche pas. »
2. *Écrivez une fonction qui invite l'utilisateur à saisir les valeurs de* `a` *,* `b` *,* `c` *et* `n` *, les convertit en nombres entiers, et utilise* `verifie_fermat` *pour vérifier si elles enfreignent le théorème de Fermat.*

Exercice 3

Si l'on vous donne trois bâtons, vous pouvez être en mesure de les organiser dans un triangle, ou pas. Par exemple, si l'un des bâtons a une longueur de 12 centimètres et les deux autres ont un centimètre de long, vous ne serez pas en mesure de faire en sorte que les bâtons courts se rencontrent au milieu. Pour trois longueurs quelconques, il existe un test simple pour savoir s'il est possible de former un triangle :

Si l'une des trois longueurs est supérieure à la somme des deux autres, alors vous ne pouvez pas former un triangle. Sinon, vous pouvez. (Si la somme des deux longueurs est égale à la troisième, elles forment ce qu'on appelle un triangle « aplati » ou « dégénéré ».)

1. *Écrivez une fonction nommée* `is_triangle` *qui prend comme arguments trois entiers, et qui imprime « Oui » ou « Non », selon que vous pouvez ou ne pouvez pas former un triangle avec des bâtons ayant les longueurs données.*
2. *Écrivez une fonction qui demande à l'utilisateur de saisir trois longueurs, les convertit en nombres entiers, et utilise* `is_triangle` *pour vérifier si les bâtons de longueurs données peuvent former un triangle.*

Exercice 4

Quelle est la sortie du programme suivant ? Dessinez un diagramme de pile qui montre l'état du programme lorsqu'il affiche le résultat.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1 *Qu'arriverait-il si vous avez appelé cette fonction comme ceci :* `recurse(-1, 0)` *?*
2 *Écrivez une docstring qui explique tout ce que quelqu'un devrait savoir pour utiliser cette fonction (et rien d'autre).*

Les exercices suivants utilisent le module `turtle`, décrit dans le chapitre **4** :

Exercice 5

Lisez la fonction suivante et voyez si vous pouvez comprendre ce qu'elle fait. Puis exécutez-la (voir les exemples dans le chapitre **4** *).*

```
def dessiner(t, longueur, n):
    if n == 0:
        return
    angle = 50
    t.fd(longueur * n)
    t.lt(angle)
    dessiner(t, longueur, n-1)
    t.rt(2 * angle)
    dessiner(t, longueur, n-1)
    t.lt(angle)
    t.bk(longueur * n)
```

Figure 5.2 : Une courbe de Koch.

Exercice 6

La courbe de Koch est une fractale qui ressemble à la figure 5.2. Pour dessiner une courbe de Koch de longueur x*, tout ce que vous avez à faire est de*

1 *Dessiner une courbe de Koch avec une longueur* x */ 3.*
2 *Tourner à gauche 60 degrés.*
3 *Dessiner une courbe de Koch avec une longueur* x */ 3.*
4 *Tourner à droite 120 degrés.*
5 *Dessiner une courbe de Koch avec une longueur* x */ 3.*
6 *Tourner à gauche 60 degrés.*
7 *Dessiner une courbe de Koch avec une longueur* x */ 3.*

L'exception est si x *est inférieur à 3 : dans ce cas, vous pouvez uniquement tracer une ligne droite d'une longueur* x*.*

1 *Écrivez une fonction appelée* `koch` *qui prend comme paramètres une tortue et une longueur, et qui utilise la tortue pour dessiner une courbe de Koch avec la longueur donnée.*

2 *Écrivez une fonction appelée* snowflake *qui dessine trois courbes de Koch pour faire l'ébauche d'un flocon de neige.*
Solution : **koch.py**.

3 *La courbe de Koch peut être généralisée à plusieurs égards. Voir* **https://fr.wikipedia.org/wiki/Flocon_de_Koch** *(voir aussi* **https://en.wikipedia.org/wiki/Koch_snowflake** *) pour des exemples et mettez en œuvre votre favori.*

6 - Fonctions productives

Beaucoup des fonctions Python que nous avons utilisées, comme les fonctions mathématiques, produisent des valeurs de retour. Mais les fonctions que nous avons écrites jusqu'à présent sont toutes « vides » : elles ont un effet, comme l'affichage d'une valeur ou le déplacement d'une tortue, mais elles ne renvoient aucune valeur de retour. Dans ce chapitre, vous allez apprendre à écrire des fonctions productives (renvoyant des valeurs).

6-1 - Valeurs de retour

L'appel d'une fonction génère une valeur de retour. Habituellement, nous attribuons cette valeur à une variable, ou l'utilisons comme partie d'une expression.

```
e = math.exp(1.0)
hauteur = rayon * math.sin(radians)
```

Les fonctions que nous avons écrites jusqu'à présent sont vides, c'est-à-dire qu'elles ne renvoient aucune valeur ; plus précisément, leur valeur de retour est None.

Dans ce chapitre, nous allons (enfin) écrire des fonctions productives. Le premier exemple est aire, qui retourne la surface d'un cercle dont le rayon est donné :

```
def aire(rayon):
    a = math.pi * rayon**2
    return a
```

Nous avons rencontré l'instruction return précédemment, mais, dans une fonction productive, l'instruction return inclut une expression. Cette instruction signifie : « sortir immédiatement de cette fonction et utiliser l'expression qui suit le return comme valeur de retour ». L'expression peut être arbitrairement compliquée, donc nous aurions pu écrire cette fonction de façon plus concise :

```
def aire(rayon):
    return math.pi * rayon**2
```

D'un autre côté, les **variables temporaires** comme a peuvent rendre le débogage plus facile.

Parfois, il est utile d'avoir plusieurs instructions de retour, une dans chaque branche d'une instruction conditionnelle :

```
def valeur_absolue(x):
    if x < 0:
        return -x
    else:
        return x
```

Puisque ces instructions de retour sont dans une alternative, une seule d'entre elles est atteinte et exécutée.

Dès qu'une instruction de retour est exécutée, la fonction se termine sans exécuter aucune des lignes de code qui suivent. Le code qui apparaît après une instruction de retour ou tout autre endroit que le flux d'exécution ne peut jamais atteindre s'appelle du **code mort**.

Dans une fonction productive, il est souhaitable de faire en sorte que tous les chemins d'exécution possibles à travers le programme atteignent une instruction de retour. Par exemple :

```
def valeur_absolue(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Cette fonction est incorrecte parce que s'il arrive que x soit égal à 0, aucune des deux conditions n'est vraie, et la fonction se termine sans atteindre d'instruction de retour. Si le flux d'exécution arrive à la fin d'une fonction, la valeur de retour par défaut est None, ce qui n'est pas la valeur absolue de 0.

```
>>> valeur_absolue(0)
None
```

Au fait, Python fournit une fonction interne appelée abs qui calcule des valeurs absolues.

À titre d'exercice, écrivez une fonction comparer qui prend deux valeurs, x et y, et retourne 1 si x> y, 0 si x == y, et -1 si x < y.

6-2 - Développement incrémental

Si vous écrivez des fonctions plus volumineuses, il se peut que vous dépensiez plus de temps à déboguer.

Pour gérer le développement de programmes de plus en plus complexes, vous pourriez vouloir essayer un processus appelé **développement incrémental**. L'objectif de celui-ci est d'éviter de longues sessions de débogage et de test en ajoutant seulement une petite quantité de code à la fois.

À titre d'exemple, supposons que vous vouliez trouver la distance entre deux points spécifiés par leurs coordonnées (x1, y1) et (x2, y2). Le théorème de Pythagore nous dit que la distance est :

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

La première étape est de considérer à quoi devrait ressembler une fonction distance en Python. Autrement dit, quelles sont les données en entrée (paramètres) et quelle en la donnée en sortie (la valeur de retour) ?

Dans ce cas, les entrées sont les deux points, que vous pouvez représenter en utilisant quatre nombres. La valeur de retour est la distance représentée par une valeur en virgule flottante.

Immédiatement, vous pouvez écrire un début de fonction :

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Évidemment, cette version ne calcule pas des distances, elle renvoie toujours zéro. Mais elle est syntaxiquement correcte et elle s'exécute sans erreur, ce qui signifie que vous pouvez la tester avant de la rendre plus complexe.

Pour tester la nouvelle fonction, appelez-la avec des exemples d'arguments :

```
>>> distance(1, 2, 4, 6)
0.0
```

J'ai choisi ces valeurs de sorte que la distance horizontale soit égale à 3 et la distance verticale soit 4 ; de cette façon, le résultat est $\sqrt{9+16}=5$, l'hypoténuse d'un triangle 3-4-5. Lorsque l'on teste une fonction, il est utile de connaître la bonne réponse attendue.

À ce stade, nous avons confirmé que la fonction est syntaxiquement correcte et nous pouvons commencer à ajouter du code au corps. Une prochaine étape raisonnable est de trouver les différences x_2 - x_1 et y_2 - y_1. La version suivante stocke ces valeurs dans des variables temporaires et les affiche :

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx est', dx)
    print('dy est', dy)
    return 0.0
```

Si la fonction est correcte, elle devrait afficher dx est 3 et dy est 4. Dans l'affirmative, nous savons que la fonction reçoit les bons arguments et effectue correctement le premier calcul. Sinon, il y a seulement quelques lignes à vérifier.

Ensuite, nous calculons la somme des carrés de dx et dy :

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    somme_carres = dx**2 + dy**2
    print('somme_carres est : ', somme_carres)
    return 0.0
```

Encore une fois, vous devez exécuter le programme à ce stade et vérifier la sortie (qui devrait être 25). Enfin, vous pouvez utiliser math.sqrt pour calculer et renvoyer le résultat :

Si cela fonctionne correctement, vous avez terminé. Sinon, vous voudrez peut-être afficher la valeur de resultat avant l'instruction de retour.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    somme_carres = dx**2 + dy**2
    resultat = math.sqrt(somme_carres)
    return resultat
```

La version définitive de la fonction n'affiche rien quand elle est exécutée ; elle ne fait que renvoyer une valeur. Les instructions print que nous avons écrites sont utiles pour le débogage, mais une fois que votre fonction marche, vous devez les supprimer. Ce genre de code s'appelle parfois un **échafaudage**, car il est utile pour la construction du programme, mais ne fait pas partie du produit final.

Lorsque vous débutez, vous devriez ajouter seulement une ou deux lignes de code à la fois. Au fur et à mesure que vous accumulez de l'expérience, vous en arriverez sans doute à écrire et déboguer des fragments de code de plus en plus longs. En tout cas, le développement incrémental peut vous faire économiser beaucoup de temps de débogage.

Les aspects clés du processus sont :

1. Commencez avec un programme qui fonctionne et faites de petits changements progressifs. À tout moment, s'il y a une erreur, vous devriez deviner à peu près l'endroit où elle se trouve ;
2. Utilisez des variables pour stocker les valeurs intermédiaires, afin de pouvoir les afficher et vérifier leur contenu ;
3. Une fois que le programme fonctionne, vous voudrez peut-être supprimer certains éléments de l'échafaudage ou consolider plusieurs instructions dans des expressions composées, mais seulement si cela ne rend pas le programme difficile à lire.

À titre d'exercice, utilisez le développement incrémental pour écrire une fonction appelée hypotenuse qui retourne la longueur de l'hypoténuse d'un triangle rectangle à partir de la longueur des deux autres côtés passés en arguments. Enregistrez chaque étape du processus de développement au fur et à mesure.

6-3 - Composition

Comme vous devriez vous en douter depuis un moment, vous pouvez appeler une fonction au sein d'une autre. À titre d'exemple, nous allons écrire une fonction qui prend en entrée deux points, le centre du cercle et un point sur le périmètre, et calcule l'aire du cercle.

Supposons que le point central est stocké dans les variables xc et yc, et les coordonnées du point sur le périmètre sont xp et yp. La première étape consiste à trouver le rayon du cercle, qui est la distance entre les deux points. Nous venons d'écrire une fonction, distance, qui fait exactement cela :

```
rayon = distance(xc, yc, xp, yp)
```

La prochaine étape est de trouver l'aire d'un cercle avec ce rayon ; nous pouvons écrire aussi :

```
resultat = aire(rayon)
```

En encapsulant ces étapes dans une fonction, nous obtenons :

```
def circle_area(xc, yc, xp, yp):
    rayon = distance(xc, yc, xp, yp)
    resultat = aire(rayon)
    return resultat
```

Les variables temporaires rayon et resultat sont utiles pour le développement et le débogage, mais une fois que le programme fonctionne, nous pouvons rendre la fonction plus concise en *composant* les appels de fonction :

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6-4 - Fonctions booléennes

Les fonctions peuvent retourner des valeurs booléennes, ce qui est souvent pratique pour cacher des tests compliqués dans des fonctions. Par exemple :

```
def divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Il est courant de donner aux fonctions booléennes des noms qui ressemblent à des questions oui/non ; is_divisible retourne soit True, soit False pour indiquer si x est divisible par y.

Voici un exemple :

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

Le résultat de l'opérateur == est lui-même un booléen, donc nous pouvons écrire la fonction de manière plus concise en renvoyant ce booléen directement :

```
def divisible(x, y):
    return x % y == 0
```

Les fonctions booléennes sont souvent utilisées dans des instructions conditionnelles :

```
if divisible(x, y):
    print('x est divisible par y')
```

Il pourrait être tentant d'écrire quelque chose comme :

```
if divisible(x, y) == True:
    print('x est divisible par y')
```

Cela fonctionne, mais la comparaison supplémentaire est inutile : on a déjà un booléen, on peut l'utiliser directement dans la condition.

À titre d'exercice, écrivez une fonction compris_entre(x, y, z) qui retourne True si x ≤ y ≤ z et False sinon.

6-5 - Plus de récursivité

Nous avons couvert seulement un petit sous-ensemble de Python, mais vous pourriez être intéressés de savoir que ce sous-ensemble est un langage de programmation *complet*, ce qui signifie que tout ce qui peut être calculé peut être exprimé dans ce langage. Tout programme jamais écrit pourrait être réécrit en utilisant uniquement les fonctionnalités du langage que vous avez apprises jusqu'ici (en fait, vous auriez besoin de quelques commandes pour contrôler des périphériques comme la souris, les disques, etc., mais c'est tout).

Prouver cette affirmation est un exercice non trivial, accompli pour la première fois par Alan Turing, l'un des premiers chercheurs en informatique (certains diront qu'il était un mathématicien, mais beaucoup des premiers informaticiens ont fait leurs débuts comme mathématiciens). Par conséquent, cela est connu comme la thèse de Turing. Pour une discussion plus complète (et précise) sur la thèse de Turing, je recommande le livre de Michael Sipser, *Introduction to the Theory of Calculation*. (Le lecteur francophone pourra se faire une idée de la problématique sur **la page de Wikipédia consacrée à ce sujet**. NdT).

Pour vous donner une idée de ce que vous pouvez faire avec les outils que vous avez étudiés jusqu'ici, nous allons évaluer quelques fonctions mathématiques définies de manière récursive. Une définition récursive est semblable à une définition circulaire, en ce sens que la définition contient une référence à la chose en train d'être définie. Une définition véritablement circulaire n'est pas très utile :

vorpal : *adjectif utilisé pour décrire quelque chose qui est vorpal.*

Si vous lisiez cette définition dans le dictionnaire, vous seriez sans doute bien ennuyé. D'un autre côté, si vous regardiez la définition de la fonction factorielle, notée par le symbole « ! », vous obtiendriez sans doute quelque chose comme ceci :

$$0! = 1$$
$$n! = n(n-1)!$$

Cette définition indique que la factorielle de 0 est 1, et la factorielle de toute autre valeur *n* est *n* multiplié par la factorielle de *n* - 1.

Donc 3! est 3 fois 2!, qui est 2 fois 1!, qui est 1 fois 0!. En mettant le tout ensemble, 3! est égal à 3 fois 2 fois 1 fois 1, ce qui donne 6.

Si vous pouvez écrire une définition récursive de quelque chose, vous pouvez écrire un programme Python pour l'évaluer. La première étape consiste à décider ce que les paramètres devraient être. Dans ce cas, il devrait être clair que factorielle prend un entier naturel :

```
def factorielle(n):
```

S'il se trouve que l'argument est 0, tout ce que nous avons à faire est de renvoyer 1 :

```
def factorielle(n):
    if n == 0:
        return 1
```

Sinon, et là, cela devient intéressant, nous devons faire un appel récursif pour trouver la factorielle de *n* - 1 et la multiplier par *n* :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        recurse = factorielle(n - 1)
        resultat = n * recurse
        return result
```

Le flux d'exécution de ce programme est similaire à celui de compte_a_rebours dans la section **5.8**. Si nous appelons factorielle avec la valeur 3 :

Puisque 3 est différent de 0, nous prenons la deuxième branche et calculons la factorielle de n - 1...

- Puisque 2 est différent de 0, nous prenons la deuxième branche et calculons la factorielle de n - 1...
    - Puisque 1 est différent de 0, nous prenons la deuxième branche et calculons la factorielle de n - 1...
        - Puisque 0 est égal à 0, nous prenons la première branche et retournons 1, sans faire d'appel récursif supplémentaire.
    - La valeur de retour, 1, est multipliée par *n*, qui vaut 1, et le résultat est retourné.
- La valeur de retour, 1, est multipliée par *n*, qui vaut 2, et le résultat est retourné.

La valeur de retour (2) est multipliée par n, qui est 3, et le résultat, 6, devient la valeur de retour de l'appel de fonction qui a démarré l'ensemble du processus.

La figure 6.1 montre à quoi ressemble le diagramme de pile pour cette séquence d'appels de fonction.

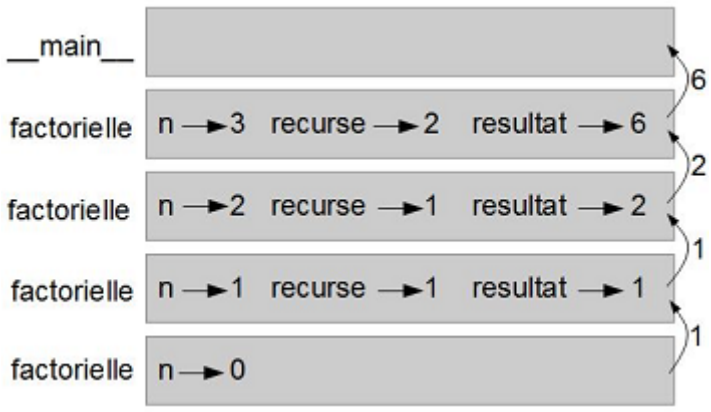


Figure 6.1 : Diagramme de pile.

Dans la dernière structure de pile, les variables locales recurse et resultat n'existent pas, parce que la branche qui les crée n'est pas exécutée.

6-6 - Acte de foi

Suivre le flux d'exécution est une façon de lire les programmes, mais on peut rapidement se trouver dépassé. Une autre possibilité est ce que j'appelle « l'acte de foi ». Quand vous arrivez à un appel de fonction, au lieu de suivre le flux d'exécution, vous *supposez* que la fonction s'exécute correctement et renvoie le bon résultat.

En fait, vous pratiquez déjà cet acte de foi lorsque vous utilisez les fonctions internes. Quand vous appelez math.cos ou math.exp, vous n'examinez pas le corps de ces fonctions. Vous ne faites que supposer qu'elles fonctionnent parce que les gens qui ont écrit les fonctions internes étaient de bons programmeurs.

Il en va de même lorsque vous appelez une de vos propres fonctions. Par exemple, dans la section **6.4**, nous avons écrit une fonction appelée divisible qui détermine si un nombre est divisible par un autre. Une fois que nous nous sommes convaincus que cette fonction est correcte - en examinant son code et en la testant - , nous pouvons utiliser la fonction sans regarder à nouveau le corps.

Même chose en ce qui concerne les programmes récursifs. Lorsque vous arrivez à l'appel récursif, au lieu de suivre le flux d'exécution, vous devez supposer que les appels récursifs fonctionnent (renvoient le résultat correct), puis vous poser la question, « En supposant que je puisse trouver la factorielle de n - 1, puis-je calculer la factorielle de n ? » Il est clair que vous pouvez, en multipliant la factorielle de n - 1 par *n*.

Bien sûr, il est un peu étrange de penser que la fonction s'exécute correctement alors que vous ne l'avez pas fini de l'écrire, mais c'est pourquoi on appelle cela un acte de foi !

6-7 - Encore un exemple

Après factorielle, l'exemple le plus courant d'une fonction mathématique définie de manière récursive est fibonacci, qui a la définition suivante (voir **https://fr.wikipedia.org/wiki/Suite_de_Fibonacci** et **http://en.wikipedia.org/wiki/Fibonacci_number**) :

$$fibonacci(0) = 0$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$$

Adapté en Python, cela ressemblerait à ceci :

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Si vous essayez de suivre le flux d'exécution dans ce cas-ci, même pour des valeurs assez petites de *n*, votre tête risque d'exploser. Mais, conformément à l'acte de foi, si vous supposez que les deux appels récursifs fonctionnent correctement, alors il est clair que vous obtenez le bon résultat en les ajoutant.

6-8 - Vérifier les types

Que se passe-t-il si nous appelons factorielle en lui donnant 1,5 comme argument ?

```
>>> factorielle(1.5)
RuntimeError: Maximum recursion depth exceeded
```

Cela ressemble à une récursion infinie. Comment est-ce possible ? La fonction a un cas de base, pour le cas où n == 0. Mais si n n'est pas un nombre entier, nous risquons de *ne pas atteindre* le cas de base et d'effectuer des récursions à jamais.

Dans le premier appel récursif, la valeur de n est de 0.5. Dans l'appel suivant, elle est de - 0.5. À partir de là, elle devient de plus en plus petite (plus négative), mais elle ne sera jamais 0.

Nous avons deux choix. Nous pouvons essayer de généraliser la fonction factorielle pour travailler avec des nombres à virgule flottante, ou nous pouvons modifier factorielle pour faire en sorte qu'elle vérifie le type de son argument. La première option est appelée la fonction gamma et elle est un peu au-delà du périmètre de ce livre. Nous allons donc choisir la seconde.

Nous pouvons utiliser la fonction interne isinstance pour vérifier le type de l'argument. Tant que nous y sommes, nous pouvons également nous assurer que l'argument est positif :

```
def factorielle (n):
    if not isinstance(n, int):
        print('Factorielle est définie uniquement pour des entiers.')
        return None
    elif n < 0:
        print("Factorielle n'est pas définie pour des entiers négatifs.")
        return None
    elif n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

Le premier cas de base gère les valeurs qui ne sont pas des entiers ; le deuxième gère les entiers négatifs. Dans les deux cas, le programme affiche un message d'erreur et retourne None pour indiquer que quelque chose s'est mal passé :

```
>>> factorielle('fred')
Factorielle est définie uniquement pour des entiers.
None
>>> factorielle(-2)
Factorielle n'est pas définie pour des entiers négatifs.
None
```

Si nous passons les deux contrôles, nous savons que *n* est entier et qu'il est positif ou nul, donc nous pouvons prouver que la récursion ne sera pas infinie.

Ce programme illustre un mécanisme que l'on appelle parfois une **garde**. Les deux premières conditionnelles agissent comme des gardes, en protégeant le code qui suit des valeurs qui pourraient causer une erreur. Les gardes permettent de prouver l'exactitude du code.

Dans la section **11.4**, nous verrons une autre option, plus souple que l'impression d'un message d'erreur : signaler une exception.

6-9 - Débogage

Découper un vaste programme en fonctions plus petites crée des points de contrôle naturels pour le débogage. Si une fonction ne fonctionne pas, il y a trois possibilités à envisager :

- il y a un problème avec les arguments que la fonction reçoit ; une condition préalable est enfreinte ;
- il y a un problème avec la fonction ; une postcondition est enfreinte ;
- il y a un problème avec la valeur de retour ou la façon dont elle est utilisée.

Pour exclure la première possibilité, vous pouvez ajouter une instruction d'impression au début de la fonction et afficher la valeur des paramètres (et peut-être leur type). Ou vous pouvez écrire du code qui vérifie explicitement les conditions préalables.

Si les paramètres semblent bons, ajoutez une instruction d'impression avant chaque déclaration de retour et affichez la valeur de retour. Si possible, vérifiez le résultat à la main. Envisagez d'appeler la fonction avec des valeurs qui donnent un résultat facile à vérifier (comme dans la section **6.2**).

Si la fonction semble marcher, regardez l'appel de fonction pour vous assurer que la valeur de retour est utilisée correctement (ou qu'elle est utilisée tout court !).

L'ajout d'instructions d'impression au début et à la fin d'une fonction peut aider à rendre le flux d'exécution plus visible. Par exemple, voici une version de `factorielle` avec des instructions d'impression :

```
def factorielle(n):
    espace = ' ' * (4 * n)
    print(espace, 'factorielle', n)
    if n == 0:
        print(espace, 'retourner 1')
        return 1
    else:
        recurse = factorielle(n - 1)
        resultat = n * recurse
        print(espace, 'retourner', resultat)
        return resultat
```

`espace` est une chaîne de caractères espace qui contrôle l'indentation de la sortie. Voici le résultat de `factorielle(4)` :

```
                factorielle 4
            factorielle 3
        factorielle 2
    factorielle 1
factorielle 0
retourner 1
    retourner 1
        retourner 2
            retourner 6
                retourner 24
```

Si vous le déroulement de l'exécution n'est pas clair pour vous, ce genre de sortie peut être utile. Il faut un certain temps pour développer un échafaudage efficace, mais un peu d'échafaudage peut économiser beaucoup de débogage.

6-10 - Glossaire

- **variable temporaire :** une variable utilisée pour stocker une valeur intermédiaire dans un calcul complexe.
- **code mort :** partie d'un programme qui ne peut jamais être exécutée, souvent parce qu'elle apparaît après une instruction `return`.
- **développement incrémental :** un système de développement d'un programme, qui vise à éviter du débogage en ajoutant et testant seulement une petite quantité de code à la fois.

- **échafaudage :** code qui est utilisé pendant le développement du programme, mais ne fait pas partie de la version finale.
- **garde :** un mécanisme de programmation qui utilise une instruction conditionnelle pour vérifier et gérer les circonstances qui pourraient causer une erreur.

6-11 - Exercices

Exercice 1

Dessinez un schéma de pile pour le programme suivant. Qu'affiche le programme ?

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Exercice 2

La fonction d'Ackermann, A(m, n), est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

Voir **https://fr.wikipedia.org/wiki/Fonction_d%27Ackermann** *. Écrivez une fonction nommée* `ack` *qui calcule la fonction d'Ackermann. Utilisez votre fonction pour évaluer* `ack(3, 4)` *, qui devrait être 125. Que se passe-t-il dans le cas des valeurs plus grandes de* `m` *et* `n` *?*

Solution : **ackermann.py**.

Exercice 3

Un palindrome est un mot que l'on peut lire indifféremment de gauche à droite ou de droite à gauche, comme « kayak » et « ressasser ». De façon récursive, un mot est un palindrome si sa première et sa dernière lettre sont identiques et le reste des lettres au milieu du mot est un palindrome.

Les fonctions suivantes prennent comme argument une chaîne de caractères et retournent respectivement la première lettre, la dernière et le milieu du mot (*word*, en anglais) :

```
def first(word):
    return word[0]

def last(word):
    return word[-1]
```

```
def middle(word):
    return word[1:-1]
```

Nous verrons leur fonctionnement dans le chapitre **8** *.*

1 *Tapez ces fonctions dans un fichier nommé* palindrome.py *et testez-les. Qu'advient-il si vous appelez* middle *avec une chaîne ayant deux lettres ? Mais dans le cas d'une chaîne d'une lettre ? Qu'en est-il de la chaîne vide, qui s'écrit ' ' et ne contient aucune lettre ?*
2 *Écrivez une fonction appelée* is_palindrome *qui prend un argument chaîne de caractères et retourne* True *si celle-ci est un palindrome et* False *sinon. Rappelez-vous que vous pouvez utiliser la fonction interne* len *pour déterminer la longueur d'une chaîne.*

Solution : **palindrome_soln.py**.

Exercice 4

Un nombre quelconque, a, *est une puissance de* b *s'il est divisible par* b *et* a / b *est une puissance de* b. *Écrivez une fonction appelée* is_power *qui prend les paramètres* a *et* b *et retourne* True *si* a *est une puissance de* b. *Remarque : vous devrez penser au cas de base.*

Exercice 5

Le plus grand diviseur commun (PGDC) de a *et* b *est le plus grand nombre qui divise les deux sans aucun reste.*

Une manière de trouver le PGDC de deux nombres est basée sur l'observation que si r *est le reste de la division de* a *par* b, *alors* pgdc(a, b) = pgdc(b, r). *Comme cas de base, nous pouvons utiliser* pgdc(a, 0) = a.

Écrivez une fonction appelée pgdc *qui prend les paramètres* a *et* b *et retourne leur plus grand diviseur commun.*

Référence : cet exercice est basé sur un exemple du livre « Structure et interprétation des programmes informatiques » (*InterÉd. 1989) de Abelson et Sussman.*

7 - Itération

Ce chapitre traite de l'itération, c'est-à-dire la capacité d'exécuter à plusieurs reprises un bloc d'instructions. Nous avons vu un type d'itération, utilisant la récursivité, dans la section **5.8**. Nous en avons vu un autre type, utilisant une boucle for, dans la section **4.2**. Dans ce chapitre, nous verrons encore un autre type, en utilisant une instruction while. Mais d'abord, je tiens à en dire un peu plus sur l'affectation des variables.

7-1 - Réaffectation

Comme vous avez pu le découvrir, il est permis d'assigner plus d'une valeur à la même variable. Une nouvelle affectation permet à une variable existante de faire référence à une nouvelle valeur (et de cesser de faire référence à l'ancienne valeur).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

Lorsque nous affichons x la première fois, sa valeur est 5 ; la seconde fois, sa valeur est 7.

La figure 7.1 montre à quoi ressemble une **réaffectation** dans un diagramme d'état.

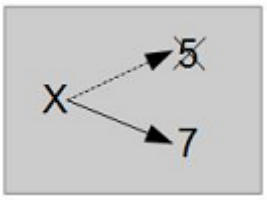


Figure 7.1 : Diagramme d'état.

À ce stade, je voudrais aborder une source de confusion commune. Comme Python utilise le signe égal (=) pour l'affectation, il est tentant d'interpréter une instruction du genre `a = b` comme une proposition mathématique d'égalité ; à savoir l'affirmation selon laquelle `a` et `b` sont égaux. Mais cette interprétation est erronée.

Premièrement, l'égalité est une relation symétrique et l'affectation ne l'est pas. Par exemple, en mathématiques si *a* = 7, alors 7 = *a*. Mais en Python, l'instruction `a = 7` est autorisée et `7 = a` ne l'est pas.

En plus, en mathématiques une proposition d'égalité est toujours soit vraie, soit fausse. Si *a* = *b* maintenant, alors *a* sera toujours égal à *b*. En Python, une instruction d'affectation peut rendre égales deux variables, mais rien ne les oblige à rester égales :

```
>>> a = 5
>>> b = a     # a et b sont égales maintenant
>>> a = 3     # a et b ne sont plus égales
>>> b
5
```

La troisième ligne modifie la valeur de `a`, mais ne modifie pas la valeur de b, et les deux variables ne sont donc plus égales.

La réaffectation des variables est souvent utile, mais vous devez l'utiliser avec prudence. Si la valeur des variables change fréquemment, cela peut rendre le code difficile à lire et à déboguer.

7-2 - Mettre à jour les variables

Un genre commun de réaffectation est une **mise à jour**, où la nouvelle valeur de la variable dépend de l'ancienne.

```
>>> x = x + 1
```

Cela signifie « prenez la valeur actuelle de x, ajoutez-y 1, puis mettez à jour x avec la nouvelle valeur ».

Si vous essayez de mettre à jour une variable qui n'existe pas, vous obtenez une erreur, parce que Python évalue le côté droit avant d'attribuer une valeur à x :

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Avant de pouvoir mettre à jour une variable, vous devez l'**initialiser**, habituellement avec une instruction simple :

```
>>> x = 0
>>> x = x + 1
```

La mise à jour d'une variable en ajoutant 1 à sa valeur s'appelle une **incrémentation** ; la soustraction de 1 est appelée **décrémentation**.

7-3 - L'instruction while

Les ordinateurs sont souvent utilisés pour automatiser des tâches répétitives. Répéter des tâches identiques ou similaires sans faire d'erreur est quelque chose que les ordinateurs font bien et que les gens font mal. Dans un programme d'ordinateur, la répétition s'appelle aussi **itération**.

Nous avons vu déjà deux fonctions, compte_a_rebours et print_n, qui itèrent en utilisant la récursivité. Comme l'itération est une tâche très commune, Python fournit des fonctionnalités de langage pour la rendre plus facile. L'une d'entre elles est l'instruction for, que nous avons vue dans la section **4.2**. Nous y reviendrons plus tard.

Une autre est l'instruction while *(tant que)*. Voici une version du compte_a_rebours qui utilise une instruction while :

```
def compte_a_rebours(n):
    while n > 0:
        print(n)
        n = n - 1
    print('Décollez !')
```

Vous pouvez presque lire l'instruction while comme si c'était une phrase écrite en anglais. Cela signifie « Tant que n est supérieur à 0, affichez la valeur de n, puis décrémentez n. Lorsque vous arrivez à 0, affichez le mot Décollez ! ».

Plus formellement, voici le flux d'exécution pour une instruction while :

1 Déterminer si la condition est vraie ou fausse ;
2 Si elle est fausse, sortir de l'instruction while et poursuivre l'exécution à l'instruction suivante ;
3 Si la condition est vraie, exécutez le corps, puis retournez à l'étape 1.

Ce type de flux s'appelle une boucle parce que la troisième étape reboucle vers le haut.

Le corps de la boucle doit modifier la valeur d'une ou plusieurs variables de sorte que la condition devienne finalement fausse et que la boucle se termine. Sinon, la boucle se répète pour toujours et s'appelle une **boucle infinie**. Une source inépuisable de divertissement pour les informaticiens est l'observation que les instructions sur le shampooing, « Faire mousser, rincer, répéter », sont une boucle infinie.

Dans le cas de compte_a_rebours, nous pouvons prouver que la boucle se termine : si n est zéro ou négatif, la boucle n'est jamais exécutée. Sinon, n devient plus petit à chaque itération de la boucle, donc, finalement, nous devons arriver à 0.

Pour certaines autres boucles, ce n'est pas si facile à dire. Par exemple :

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n est pair
            n = n / 2
        else:                 # n est impair
            n = n * 3 + 1
```

La condition de cette boucle est n != 1, donc la boucle continuera jusqu'à ce que n soit 1, ce qui rend la condition fausse.

À chaque passage dans la boucle, le programme affiche la valeur de la variable n, puis vérifie si celle-ci est paire ou impaire. Si n est pair, il est divisé par 2. Si n est impair, sa valeur est remplacée par n * 3 + 1. Par exemple, si l'argument passé à sequence est 3, les valeurs de n résultantes sont 3, 10, 5, 16, 8, 4, 2, 1.

Puisque n tantôt augmente, tantôt diminue, il n'y a aucune preuve évidente qu'un jour n sera 1, ou que le programme se terminera. Pour certaines valeurs particulières de n, nous pouvons prouver la fin du programme. Par exemple, si

la valeur de départ est une puissance de deux, n sera pair à chaque passage dans la boucle jusqu'à ce qu'il atteigne 1. L'exemple précédent se termine par une telle séquence, en commençant par 16.

La question difficile est de savoir si nous pouvons prouver que ce programme se termine pour *toutes* les valeurs positives de n. Jusqu'à présent, personne n'a été en mesure de le prouver *ou* de le réfuter ! (Voir **https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse**.)

À titre d'exercice, réécrivez la fonction afficher_n de la section **5.8** en utilisant l'itération au lieu de la récursivité.

7-4 - break

Parfois, vous ne savez pas qu'il est temps de mettre fin à une boucle avant d'arriver à mi-chemin du corps de la boucle au cours d'une itération donnée. Dans ce cas, vous pouvez utiliser l'instruction break pour sortir de la boucle.

Par exemple, supposons que vous vouliez prendre une saisie au clavier d'un utilisateur jusqu'à ce qu'il tape fini. Vous pourriez écrire :

```
while True:
    ligne = input('> ')
    if ligne == 'fini':
        break
    print(ligne)

print('Fini !')
```

La condition de la boucle est True, qui est toujours vraie, donc la boucle s'exécute jusqu'à ce qu'elle atteigne l'instruction break.

À chaque itération, elle affiche à l'utilisateur l'invite > . Si l'utilisateur tape fini, l'instruction break provoque la sortie de la boucle. Sinon, le programme affiche à l'écran tout ce qu'écrit l'utilisateur et remonte au début de la boucle. Voici un exemple d'exécution :

```
> pas fini
pas fini
> fini
Fini !
```

Cette façon d'écrire des boucles while est commune parce que vous pouvez vérifier la condition n'importe où dans la boucle (et pas seulement en début) et vous pouvez exprimer la condition d'arrêt affirmativement (« arrêter quand cela arrive ») plutôt que négativement (« continuer jusqu'à ce que cela se passe »).

7-5 - Racines carrées

Les boucles sont souvent utilisées dans des programmes qui calculent des résultats numériques en commençant avec une réponse approximative et l'améliorant de manière itérative.

Par exemple, une façon de calculer des racines carrées est la méthode de Newton. Supposons que vous vouliez trouver la racine carrée de *a*. Si vous commencez avec une estimation quelconque, x, vous pouvez calculer une meilleure estimation en utilisant la formule suivante :

$$y = \frac{x + a/x}{2}$$

Par exemple, si *a* est 4 et *x* est 3 :

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

Le résultat est plus proche de la bonne réponse (√4 = 2). Si nous répétons le processus avec la nouvelle estimation, le résultat se rapproche encore plus :

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

Après encore quelques itérations, l'estimation est presque exacte :

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

En général, nous ne savons pas à l'avance combien d'étapes il faudra pour arriver à la bonne réponse, mais nous savons quand nous y arrivons, car l'estimation cesse de changer :

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

Lorsque `y == x`, nous pouvons arrêter. Voici une boucle qui commence par une première estimation, `x`, et améliore celle-ci jusqu'à ce qu'elle arrête de changer :

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Pour la plupart des valeurs de `a`, ceci fonctionne très bien, mais en général, il est dangereux de tester l'égalité des `float`. Les valeurs à virgule flottante ne sont qu'approximativement exactes : la majorité des nombres rationnels, comme 1/3, et irrationnels, comme √2, ne peuvent pas être représentés exactement avec un `float`.

Plutôt que de vérifier si `x` et `y` sont exactement égaux, il est plus sûr d'utiliser la fonction interne `abs` pour calculer la valeur absolue, ou l'ampleur, de la différence entre eux :

```
    if abs(y-x) < epsilon:
        break
```

Où `epsilon` a une valeur comme `0,0000001` qui détermine à partir de quand on estime être suffisamment proche de l'égalité.

7-6 - Algorithmes

La méthode de Newton est un exemple d'un **algorithme** : un processus mécanique pour résoudre une catégorie de problèmes (dans ce cas, le calcul des racines carrées).

Pour comprendre que c'est un algorithme, cela pourrait aider de commencer par quelque chose qui n'est pas un algorithme. Lorsque vous avez appris à multiplier les nombres à un chiffre, vous avez probablement mémorisé la table de multiplication. En effet, vous avez mémorisé 100 solutions spécifiques. Ce genre de connaissances n'est pas de l'algorithmique.

Mais si vous étiez « paresseux », vous pourriez avoir appris quelques trucs. Par exemple, pour trouver le produit de *n* et 9, vous pouvez écrire *n* - 1 comme premier chiffre et 10 - *n* comme deuxième chiffre. Cette astuce est une solution générale pour multiplier un nombre à un chiffre par 9. C'est un algorithme !

De même, les techniques que vous avez apprises pour l'addition, la soustraction ou la division avec retenue sont toutes des algorithmes. L'une des caractéristiques des algorithmes est qu'ils ne nécessitent aucune intelligence particulière pour les effectuer. Ce sont des mécanismes dans lesquels chaque étape découle de la précédente selon un ensemble de règles simples.

L'exécution des algorithmes est ennuyeuse, mais leur conception est intéressante, intellectuellement stimulante, et représente une partie centrale de la science informatique.

Certaines des choses que les gens font naturellement, sans aucune difficulté ou pensée consciente, sont les plus difficiles à exprimer en algorithmique. Comprendre le langage naturel humain est un bon exemple. Nous le faisons tous, mais jusqu'à présent, personne n'a été en mesure d'expliquer comment nous le faisons ni de le formaliser sous la forme d'un algorithme.

7-7 - Débogage

Comme vous commencez à écrire des programmes plus volumineux, il se peut que vous arriviez à passer plus de temps à déboguer. Plus de code signifie plus de chances de faire une erreur et plus d'endroits où les bogues peuvent se cacher.

Une façon de réduire votre temps de débogage est le « débogage par dichotomie ». Par exemple, s'il y a 100 lignes de code dans votre programme et vous les vérifiez une par une, il faudra 100 étapes.

Au lieu de cela, essayez de diviser le problème en deux. Recherchez au milieu du programme, ou aux alentours, une valeur intermédiaire que vous pouvez vérifier. Ajoutez une instruction d'affichage (ou autre chose dont l'effet est vérifiable) et exécutez le programme.

Si le résultat de la vérification à mi-chemin est incorrect, il doit y avoir un problème dans la première moitié du programme. S'il est correct, le problème est dans la seconde moitié.

Chaque fois que vous effectuez une vérification de ce genre, vous réduisez de moitié le nombre de lignes que vous devez examiner. Après six étapes (ce qui est nettement moins de 100), vous arriverez à une ou deux lignes de code, du moins en théorie.

En pratique, trouver le « milieu du programme » n'est pas toujours simple et il n'est pas toujours possible de le vérifier. Compter les lignes et trouver le point médian exact n'a pas de sens. Au lieu de cela, pensez à des endroits dans le programme où il pourrait y avoir des erreurs et des endroits où il est facile de faire une vérification. Ensuite, choisissez un endroit où vous pensez qu'il y a des chances approximativement égales que le bogue soit avant ou après la vérification.

7-8 - Glossaire

- **réaffectation :** assignation d'une nouvelle valeur à une variable qui existe déjà.
- **mise à jour :** une affectation où la nouvelle valeur d'une variable dépend de l'ancienne.
- **initialisation :** une assignation qui donne une valeur initiale à une variable qui sera mise à jour.
- **incrémentation :** une mise à jour qui augmente la valeur d'une variable (souvent par pas de 1).
- **décrémentation :** une mise à jour qui diminue la valeur d'une variable.
- **itération :** exécution répétée d'une série d'instructions en utilisant soit un appel de fonction récursive, soit une boucle.
- **boucle infinie :** une boucle dans laquelle la condition d'arrêt n'est jamais satisfaite.
- **algorithme :** un processus général pour résoudre une catégorie de problèmes.

7-9 - Exercices

Exercice 1

Copiez la boucle de la section **7.5** *et encapsulez-la dans une fonction appelée* `mysqrt` *qui prend* `a` *comme paramètre, choisit une valeur raisonnable de* `x`*, et renvoie une estimation de la racine carrée de* `a`*.*

Pour la tester, écrivez une fonction nommée `test_racine_carree` *qui imprime un tableau comme celui-ci :*

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

Sur la première colonne se trouve un nombre, `a` *; la deuxième colonne est la racine carrée de* `a` *calculée avec* `mysqrt` *; la troisième colonne est la racine carrée calculée avec* `math.sqrt` *; la quatrième colonne est la valeur absolue de la différence entre les deux estimations.*

Exercice 2

La fonction interne `eval` *prend une chaîne de caractères et l'évalue comme du code Python en utilisant l'interpréteur Python. Par exemple :*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Écrivez une fonction appelée `eval_boucle` *qui invite l'utilisateur de manière itérative à saisir une chaîne de caractères, prend la saisie résultante, l'évalue en utilisant* `eval` *et affiche le résultat.*

Cela devrait continuer jusqu'à ce que l'utilisateur entre « fini », puis la valeur de la dernière expression évaluée est retournée.

Exercice 3

Le mathématicien Srinivasa Ramanujan a trouvé une série infinie qui peut être utilisée pour générer une approximation numérique de 1 / π *:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Écrivez une fonction appelée estimate_pi *qui utilise cette formule pour calculer et retourner une estimation de* π*. Elle doit utiliser une boucle* while *pour calculer les termes de la somme jusqu'à ce que le dernier terme calculé de la somme soit plus petit que* 1e-15 *(qui est la notation Python pour* 10^{-15}*). Vous pouvez vérifier le résultat en le comparant à* math.pi*.*

Solution : **pi.py**.

8 - Chaînes de caractères

Les chaînes de caractères ne sont pas comme les entiers, les flottants et les booléens. Une chaîne de caractères est une **séquence**, ce qui signifie que c'est une collection ordonnée d'autres valeurs. Dans ce chapitre, vous verrez comment accéder aux caractères qui composent une chaîne, et vous apprendrez à utiliser certaines des méthodes que les chaînes de caractères fournissent.

8-1 - Une chaîne de caractères est une séquence

Une chaîne de caractères est une séquence de caractères. Vous pouvez accéder aux caractères un par un, en utilisant l'opérateur [] d'indexation :

```
>>> fruit = 'banane'
>>> lettre = fruit[1]
```

La seconde instruction sélectionne le caractère numéroté par 1 de fruit et l'affecte à lettre.

L'expression entre crochets s'appelle un **indice**. L'indice indique quel caractère de la séquence vous voulez (d'où son nom).

Mais il se peut que vous n'obteniez pas ce que vous attendez :

```
>>> lettre
'a'
```

Pour la plupart des gens, la première lettre du mot 'banane' est b, et non a. Mais pour les informaticiens, l'indice est un décalage à partir du début de la chaîne, et le décalage de la première lettre est zéro.

```
>>> lettre = fruit[0]
>>> lettre
'b'
```

Donc b est la lettre numéro 0 (la « zéro-ième ») de 'banane', a est la lettre numéro 1 (« un-ième »), et n est la lettre numéro 2 (« deux-ième »).

Comme indice, vous pouvez utiliser une expression qui contient des variables et des opérateurs :

```
>>> i = 1
>>> fruit[i]
'a'
```

```
>>> fruit[i+1]
'n'
```

Mais la valeur de l'indice doit être un entier. Sinon, vous obtenez l'erreur suivante :

```
>>> lettre = fruit[1.5]
TypeError: string indices must be integers
```

8-2 - len

len est une fonction interne qui retourne le nombre des caractères d'une chaîne :

```
>>> fruit = 'banane'
>>> len(fruit)
6
```

Pour obtenir la dernière lettre d'une chaîne de caractères, vous pourriez être tentés à écrire quelque chose comme ceci :

```
>>> longueur = len(fruit)
>>> dernier = fruit[longueur]
IndexError: string index out of range
```

La raison de IndexError est qu'il n'y a aucune lettre dans 'banane' ayant l'indice 6. Puisque nous avons commencé à compter à partir de zéro, les six lettres sont numérotées de 0 à 5. Pour obtenir le dernier caractère, vous devez soustraire 1 de longueur :

```
>>> dernier = fruit[longueur - 1]
>>> dernier
'e'
```

Ou vous pouvez utiliser des indices négatifs, qui comptent à l'envers à partir de la fin de la chaîne. L'expression fruit[-1] donne la dernière lettre, fruit[-2] donne l'avant-dernière, et ainsi de suite.

8-3 - Parcours avec une boucle for

Beaucoup de calculs impliquent le traitement d'une chaîne caractère par caractère. Souvent, ils commencent au début, sélectionnent chaque caractère à tour de rôle, lui font quelque chose et continuent jusqu'à la fin. Ce modèle de traitement est appelé un **parcours**. Une façon d'écrire un tel parcours est une boucle while :

```
index = 0
while index < len(fruit):
    lettre = fruit[index]
    print(lettre)
    index = index + 1
```

Cette boucle parcourt la chaîne et affiche chaque lettre sur une ligne individuelle. La condition de la boucle est index < len(fruit), donc lorsque l'indice devient égal à la longueur de la chaîne, la condition devient fausse, et le corps de la boucle ne s'exécute plus. Le dernier caractère accédé est celui d'indice len(fruit) -1, qui est le dernier caractère de la chaîne.

À titre d'exercice, écrivez une fonction qui prend une chaîne comme argument et affiche les lettres à partir de la fin, une par ligne.

Une autre façon d'écrire un parcours est d'utiliser une boucle for :

```
for lettre in fruit:
```

```
    print(lettre)
```

À chaque passage dans la boucle, le caractère suivant dans la chaîne est affecté à la variable lettre. La boucle continue jusqu'à ce qu'il n'y ait aucun caractère restant.

L'exemple suivant montre comment utiliser la concaténation (addition de chaînes de caractères) et une boucle pour générer une série dans l'ordre alphabétique. Dans le livre de Robert McCloskey, *Faire place aux canetons*, les noms des canetons sont Jack, Kack, Lack, Mack, Nack, Ouack, Pack, et Quack. Cette boucle affiche ces noms dans l'ordre :

```
prefixes = 'JKLMNOPQ'
suffixe = 'ack'

for lettre in prefixes:
    print(lettre + suffixe)
```

L'affichage sera :

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Bien sûr, ce n'est pas tout à fait correct parce que « Ouack » et « Quack » sont mal orthographiés. À titre d'exercice, modifiez le programme pour corriger cette erreur.

8-4 - Tranches de chaînes de caractères

Un segment de chaîne de caractères s'appelle une **tranche**. La sélection d'une tranche est similaire à la sélection d'un caractère :

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

L'opérateur [n:m] retourne la partie de la chaîne allant du « n-ième » caractère, y compris celui-ci, au « m-ième » caractère, en excluant ce dernier. Ce comportement est contre-intuitif, mais si vous imaginez les indices comme pointant *entre* les caractères, comme sur la Figure 8.1, cela peut vous aider.

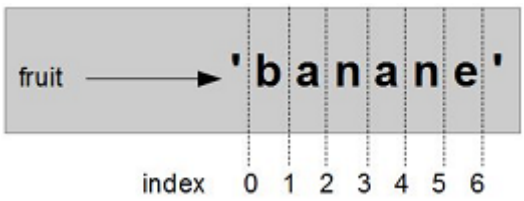


Figure 8.1 : Indices de tranches.

Si vous omettez le premier indice (avant le deux-points), la tranche commence au début de la chaîne. Si vous omettez le second indice, la tranche s'étend jusqu'à la fin de la chaîne :

```
>>> fruit = 'banane'
>>> fruit[:3]
'ban'
>>> fruit[3:]
```

```
'ane'
```

Si le premier indice est supérieur ou égal au second, le résultat est une **chaîne vide**, représentée par deux guillemets simples :

```
>>> fruit = 'banane'
>>> fruit[3:3]
''
```

Une chaîne vide ne contient aucun caractère et a une longueur de 0, mais à part ça, c'est une chaîne comme toutes les autres.

Poursuivant cet exemple, que pensez-vous que fruit[:] représente ? Essayez et vous verrez.

8-5 - Les chaînes de caractères sont immuables

Il est tentant d'utiliser l'opérateur [] du côté gauche de l'affectation, avec l'intention de modifier un caractère dans une chaîne. Par exemple :

```
>>> salutation = 'Hello, world!'
>>> salutation[0] = 'J'
TypeError: 'str' object does not support item assignment
```

Dans ce cas-ci, l'« objet » est la chaîne de caractères et l'« élément » (*item*) est le caractère que vous avez essayé d'affecter. Pour l'instant, considérez qu'un objet est la même chose qu'une valeur, mais nous allons affiner cette définition plus tard (section **10.10**).

La raison de l'erreur est que les chaînes de caractères sont **immuables** (on dit aussi non mutables), ce qui signifie que vous ne pouvez pas modifier une chaîne existante. Le mieux que vous puissiez faire est de créer une nouvelle chaîne qui est une variation sur l'original :

```
>>> salutation = 'Hello, world!'
>>> nouvelle_salutation = 'J' + salutation[1:]
>>> nouvelle_salutation
'Jello, world!'
```

Cet exemple concatène une nouvelle première lettre sur une tranche de salutation. Il n'a aucun effet sur la chaîne d'origine.

8-6 - Recherche

Que fait la fonction suivante ?

```
def trouver(mot, lettre):
    index = 0
    while index < len(mot):
        if mot[index] == lettre:
            return index
        index = index + 1
    return -1
```

Dans un sens, trouver est l'inverse de l'opérateur []. Au lieu de prendre un indice et d'extraire le caractère correspondant, elle prend un caractère et trouve l'indice où le caractère apparaît. Si le caractère n'est pas trouvé, la fonction retourne -1.

Celui-ci est le premier exemple d'une instruction return dans une boucle. Si mot[index] == lettre, la fonction sort de la boucle et retourne immédiatement.

Si le caractère n'existe pas dans la chaîne, le programme termine la boucle normalement et retourne -1.

Ce modèle de calcul - parcourir une séquence et retourner quand nous trouvons ce que nous cherchons - s'appelle une **recherche**.

À titre d'exercice, modifiez trouver de sorte qu'elle prenne un troisième paramètre, l'indice dans mot où elle devrait commencer la recherche.

8-7 - Boucler et compter

Le programme suivant compte les occurrences de la lettre a dans une chaîne de caractères :

```
mot = 'banane'
compteur = 0
for lettre in mot:
    if lettre == 'a':
        compteur = compteur + 1
print(compteur)
```

Ce programme montre un autre modèle de calcul appelé un **compteur**. La variable compteur est initialisée à 0 et ensuite elle est incrémentée chaque fois qu'un a est trouvé. Lorsque la boucle se termine, compteur contient le résultat - le nombre total de lettres a.

À titre d'exercice, encapsulez ce code dans une fonction nommée compteur et généralisez-le pour qu'il accepte la chaîne de caractères et la lettre comme arguments.

Puis réécrivez la fonction de sorte qu'au lieu de parcourir la chaîne, elle utilise la version à trois paramètres de trouver de la section précédente.

8-8 - Méthodes de chaînes de caractères

Les chaînes de caractères fournissent des méthodes qui effectuent une variété d'opérations utiles. Une méthode est semblable à une fonction - elle prend des arguments et renvoie une valeur -, mais la syntaxe d'appel est différente. Par exemple, la méthode upper prend une chaîne de caractères et retourne une nouvelle chaîne avec toutes les lettres en majuscule.

Au lieu de la syntaxe de fonction upper(mot), elle utilise la syntaxe de méthode, mot.upper().

```
>>> mot = 'banane'
>>> nouveau_mot = mot.upper()
>>> nouveau_mot
'BANANE'
```

Cette forme de notation pointée indique le nom de la méthode, upper, et le nom de la chaîne sur laquelle on applique la méthode, mot. Les parenthèses vides indiquent que cette méthode ne prend aucun argument.

Un appel de méthode est appelé une **invocation** ; dans ce cas, nous dirions que nous invoquons upper sur mot.

Il se trouve qu'il existe une méthode de chaîne de caractères nommée find, qui est remarquablement similaire à la fonction trouver que nous avons écrite :

```
>>> mot = 'banane'
>>> index = mot.find('a')
>>> index
1
```

Dans cet exemple, nous invoquons find sur mot et passons la lettre que nous recherchons en tant que paramètre.

En fait, la méthode find est plus générale que notre fonction ; elle peut trouver des sous-chaînes, et pas seulement des caractères :

```
>>> mot.find('na')
2
```

Par défaut, find commence sa recherche au début de la chaîne, mais elle peut prendre un deuxième argument, l'indice où elle devrait commencer à chercher :

```
>>> mot.find('ne', 3)
4
```

Cela est un exemple d'un **argument optionnel** ; find peut également prendre un troisième argument, l'indice où elle doit s'arrêter :

```
>>> nom = 'bob'
>>> nom.find('b', 1, 2)
-1
```

Cette recherche échoue parce que b n'apparaît pas dans la plage d'indices de 1 à 2, la lettre correspondant à l'indice 2 étant non comprise. En recherchant jusqu'à cette valeur d'indice, mais sans inclure le caractère correspondant, le second indice permet à find de rester cohérente avec l'opérateur de tranche.

8-9 - L'opérateur in

Le mot in est un opérateur booléen qui prend deux chaînes de caractères et retourne True si la première apparaît comme une sous-chaîne dans la seconde :

```
>>> 'a' in 'banane'
True
>>> 'graine' in 'banane'
False
```

Par exemple, la fonction suivante affiche toutes les lettres de mot1 qui apparaissent aussi dans mot2 :

```
def lettres_communes(mot1, mot2):
    for lettre in mot1:
        if lettre in mot2:
            print(lettre)
```

Avec des noms de variables bien choisis, Python peut parfois être lu comme si c'était une phrase écrite en anglais. Vous pouvez lire cette boucle ainsi : « for (chaque) lettre in (le premier) mot, if (la) lettre (apparaît) in (le deuxième) mot, print (la) lettre ».

Voici ce que vous obtenez si vous comparez des pommes et des oranges :

```
>>> lettres_communes('pommes', 'oranges')
o
e
s
```

8-10 - Comparaison de chaînes de caractères

Les opérateurs relationnels peuvent être utilisés sur les chaînes de caractères. Pour vérifier si deux chaînes sont égales :

```
if mot == 'banane':
    print("C'est bon, des bananes.")
```

D'autres opérations relationnelles sont utiles pour mettre des mots dans l'ordre alphabétique :

```
if mot < 'banane':
    print('Votre mot, ' + mot + ', vient avant banane.')
elif mot > 'banane':
    print('Votre mot, ' + mot + ', vient après banane.')
else:
    print("C'est bon, des bananes.")
```

Python ne gère pas les lettres majuscules et minuscules de la même façon que la plupart des gens. Toutes les lettres majuscules viennent avant toutes les lettres minuscules, donc :

```
Votre mot, Pomme, vient avant banane.
```

Une façon courante de résoudre ce problème est de convertir les chaînes dans un format standard, par exemple des minuscules, avant d'effectuer la comparaison. Gardez cela à l'esprit pour le cas où l'on désire vous faire croquer la Pomme.

8-11 - Débogage

Lorsque vous utilisez les indices pour parcourir les valeurs dans une séquence, il est délicat d'obtenir correctement le début et la fin du parcours. Voici une fonction qui est censée comparer deux mots et retourner True si l'un des mots est l'inverse de l'autre, mais elle contient deux erreurs :

```
def is_inverse(mot1, mot2):
    if len(mot1) != len(mot2):
        return False

    i = 0
    j = len(mot2)

    while j > 0:
        if mot1[i] != mot2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

La première instruction if vérifie si les mots ont la même longueur. Si ce n'est pas le cas, nous pouvons renvoyer False immédiatement. Autrement, pour le reste de la fonction, on peut supposer que les mots ont la même longueur. Ceci est un exemple du mécanisme de garde vu dans la section **6.8**.

i et j sont des indices : i parcourt mot1 à partir du début tandis que j parcourt mot2 à partir de la fin. Si nous trouvons deux lettres qui ne correspondent pas, nous pouvons renvoyer False immédiatement. Si nous arrivons à la fin de la boucle et toutes les lettres correspondent, nous renvoyons True.

Si nous testons cette fonction avec les mots « tracé » et « écart », nous nous attendons à ce que la valeur de retour soit True, mais nous obtenons une IndexError :

```
>>> is_inverse('tracé', 'écart')
...
  File "reverse.py", line 15, in is_inverse
    if mot1[i] != mot2[j]:
IndexError: string index out of range
```

Pour déboguer ce genre d'erreur, mon premier réflexe est d'imprimer les valeurs des indices immédiatement avant la ligne où l'erreur apparaît.

```
    while j > 0:
        print(i, j)        # affichage ici

        if mot1[i] != mot2[j]:
            return False
        i = i+1
        j = j-1
```

Maintenant, quand j'exécute à nouveau le programme, je reçois plus d'informations :

```
>>> is_inverse('tracé', 'écart')
0 5
...
IndexError: string index out of range
```

Au premier passage dans la boucle, la valeur de j est 5, ce qui dépasse la taille de la chaîne 'tracé'. L'indice du dernier caractère est 4, donc la valeur initiale de j devrait être len(mot2) - 1.

Si je corrige cette erreur et j'exécute à nouveau le programme, j'obtiens :

```
>>> is_inverse(tracé', 'écart')
0 4
1 3
2 2
3 1
True
```

Cette fois, nous obtenons la bonne réponse, mais il semble que la boucle n'a été exécutée que quatre fois, ce qui est suspect. Pour avoir une meilleure idée de ce qui se passe, il est utile de dessiner un diagramme d'état. Au cours de la première itération, la structure de pile de is_inverse est représentée sur la figure 8.2.

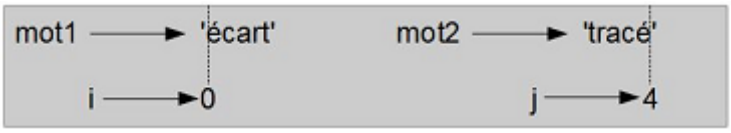


Figure 8.2 : Diagramme d'état.

J'ai arrangé les variables dans la structure et ajouté des lignes pointillées pour montrer que les valeurs de i et j indiquent des caractères dans mot1 et mot2.

Avec ce schéma comme point de départ, simulez sur papier l'exécution de ce programme, en modifiant les valeurs de i et j lors de chaque itération. Trouvez et corrigez la seconde erreur dans cette fonction.

8-12 - Glossaire

- **objet :** quelque chose auquel une variable peut se référer. Pour l'instant, vous pouvez utiliser « objet » et « valeur » de façon interchangeable.
- **séquence :** une collection ordonnée de valeurs où chaque valeur est identifiée par un indice entier.
- **élément :** l'une des valeurs dans une séquence.
- **indice :** une valeur entière utilisée pour sélectionner un élément dans une séquence, comme un caractère dans une chaîne. En Python, les indices commencent à 0.
- **tranche :** une partie d'une chaîne spécifiée par une gamme d'indices.
- **chaîne vide :** une chaîne sans aucun caractère et de longueur 0, représentée par deux guillemets simples.
- **immuable :** la propriété d'une séquence dont les éléments ne peuvent être modifiés.

- **parcourir :** itérer à travers les éléments dans une séquence, en exécutant une opération du même type sur chacun.
- **recherche :** un modèle de parcours qui s'arrête quand il trouve ce qu'il recherche.
- **compteur :** une variable utilisée pour compter quelque chose, généralement initialisée à zéro, puis incrémentée.
- **invocation :** une instruction qui appelle une méthode.
- **argument optionnel :** un argument de fonction ou de méthode qui n'est pas requis.

8-13 - Exercices

Exercice 1

Lisez la documentation des méthodes de chaînes de caractères sur **http://docs.python.org/3/library/stdtypes.html#string-methods** *. Vous voudrez peut-être expérimenter certaines d'entre elles pour vous assurer que vous comprenez comment elles fonctionnent.* `strip` *et* `replace` *sont particulièrement utiles.*

La documentation utilise une syntaxe qui pourrait être source de confusion. Par exemple, dans `find (sub[, start[, end]])` *, les parenthèses indiquent des arguments optionnels. Donc* `sub` *est requis, mais* `start` *est optionnel, et si vous incluez* `start` *, alors* `end` *est optionnel.*

Exercice 2

Il existe une méthode de chaîne de caractères appelée `count` *qui est similaire à la fonction de la section* **8.7** *. Lisez la documentation de cette méthode et écrivez une invocation qui compte le nombre des* `a` *dans* `'banane'` *.*

Exercice 3

Une tranche de chaîne peut prendre un troisième indice qui spécifie la « taille du pas » ; autrement dit, le nombre d'espaces entre caractères successifs. Une taille de pas de 2 signifie un caractère sur deux ; 3 signifie un caractère sur trois, etc.

```
>>> fruit = 'banane'
>>> fruit[0:5:2]
'bnn'
```

Une taille de pas de -1 parcourt le mot à l'envers, donc la tranche `[:: - 1]` *génère une chaîne inversée.*

Utilisez cette syntaxe pour écrire une version d'une ligne de la méthode `is_palindrome` *de l'exercice* **3** *du chapitre 6.*

Exercice 4

Les fonctions suivantes sont toutes censées *vérifier si une chaîne contient au moins une lettre minuscule quelconque, mais au moins certaines d'entre elles sont erronées. Pour chaque fonction, décrivez ce que fait réellement la fonction (en supposant que le paramètre est une chaîne).*

```
def trouver_minuscules1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def trouver_minuscules2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'
```

```
def trouver_minuscules3(s):
    for c in s:
        drapeau = c.islower()
    return drapeau

def trouver_minuscules4(s):
    drapeau = False
    for c in s:
        drapeau = drapeau or c.islower()
    return drapeau

def trouver_minuscules5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercice 5

Un chiffre de César est une forme faible de chiffrement qui implique la « rotation » de chaque lettre d'un nombre fixe de places. La rotation d'une lettre signifie de décaler sa place dans l'alphabet, en repassant par le début si nécessaire, de sorte qu'après la rotation par 3, 'A' devient 'D' *et* 'Z' *décalé de* 1 *est* 'A'.

Pour effectuer la rotation d'un mot, décalez-en chaque lettre par le même nombre. Par exemple, les mots « JAPPA » et « ZAPPA », décalés de quatre lettres, donnent respectivement « DETTE » et « NETTE », et le mot « RAVIER », décalé de 13 crans, donne « ENIVRE » (et réciproquement). De même, le mot « OUI » décalé de 10 crans devient « YES ». Dans le film 2001 : l'Odyssée de l'espace*, l'ordinateur du vaisseau s'appelle HAL, qui est IBM décalé de* -1.

Écrivez une fonction appelée `rotate_word` *qui prend comme paramètres une chaîne de caractères et un entier, et renvoie une nouvelle chaîne qui contient les lettres de la chaîne d'origine décalées selon le nombre donné.*

Vous voudrez peut-être utiliser la fonction interne `ord` *, qui convertit un caractère en un code numérique, et* `chr` *, qui convertit des codes numériques en caractères. Les lettres de l'alphabet sont codées dans l'ordre alphabétique, par exemple :*

```
>>> ord('c') - ord('a')
2
```

parce que 'c' *est la « deux-ième » lettre de l'alphabet. Mais attention : les codes numériques pour les lettres majuscules et minuscules sont différents. Et évitez les mots contenant des caractères accentués, cédilles, apostrophes et autres traits d'union (NdT).*

Des blagues potentiellement offensantes sur l'Internet sont parfois encodées en ROT13, qui est un chiffre de César avec un décalage de 13. (Comme l'alphabet latin comprend 26 lettres, appliquer à deux reprises un décalage de 13 lettres redonne le texte d'origine, ce qui fait que le même programme peut servir à chiffrer un texte en clair et à déchiffrer le cryptogramme résultant - NdT). Si vous ne vous offusquez pas facilement, trouvez et décodez certains d'entre eux. Solution : **rotate.py .**

9 - Étude de cas : jouer avec les mots

Ce chapitre présente la deuxième étude de cas, qui consiste à résoudre des problèmes de mots en recherchant des mots qui ont certaines propriétés. Par exemple, nous trouverons les plus longs palindromes en français et rechercherons des mots dont les lettres apparaissent dans l'ordre alphabétique. Et je vais vous présenter une autre méthode de développement d'un programme : la réduction à un problème déjà résolu.

9-1 - Lire des listes de mots

Pour les exercices de ce chapitre, nous avons besoin d'une liste de mots français. Il y a beaucoup de listes de mots disponibles sur le Web, et vous pouvez en principe utiliser celle que vous préférerez. Nous allons toutefois utiliser la liste des mots pour le Scrabble de Jean-Philippe Durand, disponible à l'adresse **http://jph.durand.free.fr/scrabble.htm** (ou en format texte à l'adresse **http://jph.durand.free.fr/scrabble.txt**). Il vous sera peut-être plus facile de suivre si vous utilisez la même liste.

> ***Récupérer une liste de mots français***
>
> *Vous pouvez utiliser la liste de mots qui vous plaira, mais pour récupérer la liste de mots autorisés au Scrabble de Jean-Philippe Durand mentionnée ci-dessus, vous pouvez utiliser deux méthodes :*
>
> - *Vous connecter à l'adresse **http://jph.durand.free.fr/scrabble.txt** ; sélectionner toute la page (CTRL-A sous Windows) et faire un copier-coller vers un éditeur de texte (mais non un traitement de texte) ; puis sauvegarder le document ainsi créé ;*
> - *Vous connecter sur la page d'accueil de son site (**http://jph.durand.free.fr**) ; dans la rubrique « Jeux de mots », première ligne consacrée aux mots pour le Scrabble, faire un clic droit sur le lien txt ; dans le menu déroulant, choisir l'option « Enregistrer la cible du lien sous... » ou l'équivalent dans votre navigateur ; puis sauvegarder le fichier à l'endroit voulu.*
>
> *Il s'agit d'une liste de 130 557 mots en lettres capitales et sans accents ni cédilles, ce qui élimine pour ces exercices des difficultés propres aux accents et autres signes diacritiques en français (et les éventuels problèmes d'encodage). S'agissant d'une liste de mots pour le Scrabble, il n'y a que des mots d'une longueur maximale de 15 lettres (la taille du plateau de Scrabble).*

Ce fichier est en format texte brut, donc vous pouvez l'ouvrir avec un éditeur de texte, mais vous pouvez aussi le lire à partir de Python. La fonction interne `open` prend comme paramètre le nom du fichier et retourne un **objet fichier** que vous pouvez utiliser pour lire le fichier.

```
>>> fin = open('mots.txt')
```

`fin` (pour *file in*) est un nom usuel pour un objet fichier utilisé en lecture. L'objet fichier fournit plusieurs méthodes de lecture, dont `readline`, qui lit les caractères du fichier en entrée jusqu'à ce qu'elle arrive à un passage à la ligne et renvoie le résultat sous forme de chaîne de caractères :

```
>>> fin.readline()
'AA\r\n'
```

Le premier mot de cette liste particulière est « AA », qui est une sorte de lave. La séquence `\r\n` représente deux caractères non imprimables, un retour chariot et un saut de ligne, qui séparent ce mot du mot suivant, du moins dans un environnement Windows. Selon la façon dont vous avez récupéré la liste et si vous utilisez un environnement Mac, Unix ou Linux, il se peut que vous trouviez des passages à la ligne constitués du seul caractère `\n` de saut de ligne.

L'objet fichier garde en mémoire l'endroit où il se trouve dans le fichier, donc si vous appelez `readline` à nouveau, vous obtenez le mot de la ligne suivante :

```
>>> fin.readline()
'AH\r\n'
```

Le mot suivant est « AH », qui est un mot parfaitement légitime, n'ayez pas l'air aussi étonné. Ou, si ce sont les caractères non imprimables qui vous préoccupent, nous pouvons nous en débarrasser avec la méthode de chaîne de caractères `strip` :

```
>>> ligne = fin.readline()
>>> mot = ligne.strip()
>>> mot
'AI'
```

Vous pouvez également utiliser un objet fichier avec une boucle for. Ce programme lit mots.txt et affiche chaque mot, un par ligne :

```
fin = open('mots.txt')
for ligne in fin:
    mot = ligne.strip()
    print(mot)
```

9-2 - Exercices

Les solutions à ces exercices se trouvent dans la section suivante. Vous devriez au moins essayer de les résoudre avant de lire les solutions.

Exercice 1

Écrivez un programme qui lit mots.txt *et affiche uniquement les mots de plus de 14 caractères (sans compter les caractères non imprimables). Pour l'instant, n'affichez que les premiers 20 ou 30 mots trouvés (il y en a plus de 2000 dans la liste que nous avons suggéré d'utiliser).*

Exercice 2

En 1969, Georges Perec a publié le roman appelé La Disparition *qui ne comporte pas une seule fois la lettre « e ». (Voir* **https://fr.wikipedia.org/wiki/La_Disparition_%28roman%29***) Comme « e » est de loin la lettre la plus commune en français, c'est presque un tour de force.*

En fait, il est difficile de construire même une seule idée, sans l'aide de ce symbole le plus commun. Cela va lentement au début, mais avec de la prudence et des heures d'entraînement, vous pouvez vous y habituer progressivement.

Très bien, je vais arrêter maintenant.

Écrivez une fonction appelée n_a_aucun_E *qui renvoie* True *si le mot donné ne contient aucune lettre « E ».*

Modifiez votre programme de l'exercice précédent pour afficher seulement les mots de plus de 14 lettres qui ne contiennent aucun « e » et calculer le pourcentage de mots de la liste qui ne contiennent aucun « e ». Si vous utilisez la liste suggérée, n'oubliez pas que les mots sont écrits en lettres capitales et que c'est dont la lettre capitale « E » qu'il vous faudra détecter. À titre d'information, sur l'ensemble des 130 557 mots de notre liste en entrée, seuls un peu plus de 18 000 n'ont aucune fois la lettre E.

Exercice 3

Écrivez une fonction nommée evite *qui prend un mot et une liste de lettres interdites, et qui renvoie* True *si le mot ne contient aucune des lettres interdites.*

Modifiez votre programme pour inviter l'utilisateur à saisir une chaîne de lettres interdites et ensuite afficher le nombre de mots de la liste en entrée qui ne contient aucune de celles-ci. Pouvez-vous trouver une combinaison de 5 lettres interdites qui exclut le plus petit nombre de mots ?

Exercice 4

Écrivez une fonction nommée utilise_uniquement *qui prend un mot et une chaîne de lettres, et qui renvoie* True *si le mot ne contient que les lettres de la liste. Pouvez-vous faire une phrase en utilisant seulement les lettres* acefhlo *?*

Exercice 5

Écrivez une fonction nommée utilise_toutes *qui prend un mot et une chaîne de lettres requises, et qui renvoie* True *si le mot utilise toutes les lettres requises au moins une fois. Combien y a-t-il de mots qui utilisent toutes les voyelles* aeiou *? Et* aeiouy *?*

Exercice 6

Écrivez une fonction appelée est_en_ordre_alphabetique *qui renvoie* True *si les lettres d'un mot apparaissent dans l'ordre alphabétique (les lettres doublées sont autorisées). Combien y a-t-il de mots de ce genre ?*

9-3 - Recherche

Tous les exercices de la section précédente ont quelque chose en commun : ils peuvent être résolus avec le modèle de recherche que nous avons vu dans la section **8.6**. L'exemple le plus simple est :

```
def n_a_aucun_E(mot):
    for lettre in mot:
        if lettre == 'E':
            return False
    return True
```

La boucle for parcourt les caractères de mot. Si nous trouvons la lettre « E », nous pouvons immédiatement renvoyer False ; sinon, nous devons aller à la lettre suivante. Si nous sortons de la boucle normalement, cela signifie que nous ne trouvons aucun « E », alors nous renvoyons True.

Vous pouviez écrire cette fonction de manière plus concise en utilisant l'opérateur in, mais j'ai commencé par cette version, car elle démontre la logique du modèle de recherche.

eviteest une version plus générale den_a_aucun_E, mais elle a la même structure :

```
def evite(mot, interdites):
    for lettre in mot:
        if lettre in interdites:
            return False
    return True
```

Au lieu d'une liste de lettres interdites, nous avons une liste de lettres disponibles. Si nous trouvons dans le mot une lettre qui ne fait pas partie des disponibles, nous pouvons renvoyer False.

utilise_toutes est semblable, sauf que nous inversons le rôle du mot et de la chaîne de lettres :

```
def utilise_toutes(mot, requises):
    for lettre in requises:
        if lettre not in mot:
            return False
    return True
```

Au lieu de parcourir les lettres de mot, la boucle parcourt les lettres requises. Si l'une des lettres requises ne figure pas dans le mot, nous pouvons renvoyer False.

Si vous pensiez vraiment comme un informaticien, vous auriez remarqué que utilise_toutes était une variante du problème résolu précédemment, et vous auriez écrit :

```
def utilise_toutes(mot, requises):
    return utilise_toutes(requises, mot)
```

Cela représente un exemple de méthode de développement d'un programme appelée **réduction à un problème déjà résolu**, qui signifie que vous reconnaissez le problème que vous essayez de résoudre comme une variante d'un problème résolu et appliquez une solution existante.

9-4 - Boucler avec des indices

J'ai utilisé des boucles `for` pour écrire les fonctions de la section précédente parce que j'avais besoin uniquement des caractères dans les chaînes ; je n'avais pas besoin d'utiliser les indices.

Pour `est_en_ordre_alphabetique`, nous devons comparer des lettres adjacentes, ce qui est un peu difficile avec une boucle `for` :

```
def est_en_ordre_alphabetique(mot):
    precedent = mot[0]
    for c in mot:
        if c < precedent:
            return False
        precedent = c
    return True
```

Une autre possibilité est d'utiliser la récursivité :

```
def est_en_ordre_alphabetique(mot):
    if len(mot) <= 1:
        return True
    if mot[0] > mot[1]:
        return False
    return est_en_ordre_alphabetique(mot[1:])
```

Une autre option serait d'utiliser une boucle `while`:

```
def est_en_ordre_alphabetique(mot):
    i = 0
    while i < len(mot) - 1:
        if mot[i+1] < mot[i]:
            return False
        i = i + 1
    return True
```

La boucle commence à `i = 0` et se termine lorsque `i = len(mot) - 1`. À chaque itération, la boucle compare le *i*ème caractère (que vous pouvez voir comme le caractère actuel) au *i + 1*ème caractère (que vous pouvez voir comme le caractère suivant).

Si le caractère suivant est inférieur au (alphabétiquement avant le) caractère actuel, alors nous avons découvert une interruption dans la suite alphabétique, et nous renvoyons `False`.

Si nous arrivons à la fin de la boucle sans rencontrer une telle situation, alors le mot passe le test. Pour vous convaincre que la boucle se termine correctement, prenons un exemple comme `'EFFORT'`. La longueur du mot est de 6, donc la dernière exécution de la boucle a lieu lorsque `i` est 4, qui est l'indice de l'avant-dernier caractère. Lors de la dernière itération, on compare l'avant-dernier caractère (R) au dernier (T), ce qui est ce que nous voulons.

Voici une version de `is_palindrome` (voir Exercice **3** du chapitre 6) qui utilise deux indices ; l'un commence au début et augmente ; l'autre commence à la fin et diminue.

```
def is_palindrome(mot):
    i = 0
    j = len(mot) - 1

    while i < j:
        if mot[i] != mot[j]:
```

```
            return False
        i = i + 1
        j = j - 1

    return True
```

Ou nous pourrions réduire à un problème déjà résolu et écrire :

```
def is_palindrome(mot):
    return is_inverse(mot, mot)
```

en utilisant la fonction is_inverse de la section **8.11**.

9-5 - Débogage

Il est difficile de tester des programmes. Les fonctions de ce chapitre sont relativement faciles à tester, parce que vous pouvez vérifier les résultats à la main. Même ainsi, il est difficile sinon impossible de choisir un ensemble de mots pour tester toutes les erreurs possibles.

Si nous prenons n_a_aucun_E comme exemple, il y a deux cas évidents à vérifier : les mots qui ont un 'E' devraient retourner False et les mots qui n'en ont pas devraient retourner True. Vous ne devriez avoir aucune difficulté à trouver un mot de chaque type.

Dans chaque cas, il y a des sous-cas moins évidents. Parmi les mots qui contiennent un « E », vous devez tester mots avec un « E » au début, à la fin, et quelque part au milieu. Vous devez tester des mots longs, des mots courts, et des mots très courts, comme la chaîne vide. La chaîne vide est un exemple d'un cas particulier, qui est l'un des cas non évidents où les erreurs se cachent souvent.

En plus des cas de test que vous générez, vous pouvez également tester votre programme avec une liste de mots comme ceux dans le fichier mots.txt. En examinant la sortie, vous pourriez être en mesure de remarquer les erreurs, mais attention : vous pourriez déceler un type d'erreur (les mots qui ne devraient pas être inclus, mais le sont), mais pas l'autre (les mots qui devraient être inclus, mais ne le sont pas).

En général, les tests peuvent vous aider à trouver des bogues, mais il n'est pas facile de générer un bon jeu de cas de tests, et, même si vous le faites, vous ne pouvez pas être sûr que votre programme est correct. Selon un légendaire chercheur en informatique :

Tester un programme peut démontrer la présence de bogues, jamais leur absence.
— Edsger W. Dijkstra

9-6 - Glossaire

- **objet fichier :** une valeur qui représente un fichier ouvert.
- **réduction à un problème déjà résolu :** une façon de résoudre un problème en l'exprimant comme une variante d'un problème déjà résolu.
- **cas spécial :** un cas de test qui est atypique ou non évident (et moins susceptible d'être géré correctement).

9-7 - Exercices

Exercice 7

Cette question est basée sur un casse-tête diffusé sur le programme de radio Car Talk *(***http://www.cartalk.com/content/puzzlers***) :*

- *Donnez-moi un mot contenant trois lettres doubles consécutives. Je vais vous donner quelques mots qui se rapprochent, mais ne remplissent pas cette condition. Par exemple, le mot* ASSOMMEE A-s-s-o-m-m-é-e. *Ce serait super, sans le* 'o' *qui s'y faufile. Ou* Mississippi : M-i-s-s-i-s-s-i-p-p-i. *Si vous pouviez enlever ces* i, *cela fonctionnerait. Mais il y a un mot qui a trois paires consécutives de lettres et à ma connaissance, il pourrait en être le seul (avec ce même mot mis au pluriel). Bien sûr, il y en a peut-être d'autres, mais je ne peux penser qu'à un seul. Quel est le mot ?*

Écrivez un programme pour le trouver. Solution : **cartalk1.py**.

Exercice 8

Voici un autre casse-tête diffusé sur Car Talk *(***http://www.cartalk.com/content/puzzlers***) :*

- *Je roulais sur l'autoroute l'autre jour et mon regard a été attiré par mon compteur kilométrique. Comme la plupart de ces compteurs, il affiche six chiffres, seulement en kilomètres entiers. Donc, si ma voiture avait 300 000 kilomètres, par exemple, j'aurais vu 3-0-0-0-0-0.*
  Maintenant, ce que je voyais ce jour-là était très intéressant. Je remarquai que les 4 derniers chiffres étaient palindromiques ; c'est-à-dire ils composent le même nombre si on les lit de gauche à droite ou de droite à gauche. Par exemple, 5-4-4-5 est un palindrome, donc mon compteur pouvait indiquer 3-1-5-4-4-5.
  Un kilomètre plus tard, les 5 derniers chiffres étaient un palindrome. Par exemple, on aurait pu lire 3-6-5-4-5-6. Un kilomètre après, sur les 6 chiffres, les 4 du milieu composaient un palindrome. Et accrochez-vous ! Un kilomètre plus tard, tous les 6 composaient un palindrome !
  La question est qu'affichait le compteur kilométrique quand j'ai regardé la première fois ?

Écrivez un programme Python qui teste tous les numéros à six chiffres et affiche les chiffres qui satisfont à ces exigences. Solution : **cartalk2.py**.

Exercice 9

Voici un autre casse-tête Car Talk *que vous pouvez résoudre par une recherche (***http://www.cartalk.com/content/puzzlers***) :*

- *Récemment, j'étais en visite chez ma mère et nous avons réalisé que si l'on inverse les deux chiffres qui composent mon âge, nous obtenons son âge. Par exemple, si elle a 73 ans, moi j'en ai 37. Nous nous demandions combien de fois cela est arrivé au fil du temps, mais nous nous sommes détourné l'attention avec d'autres sujets et nous n'avons pas trouvé une réponse.*
  Lorsque je suis rentré chez moi, j'avais découvert que les chiffres de nos âges ont été réversibles six fois jusqu'à présent. Je me suis dit aussi que si nous sommes chanceux cela arriverait de nouveau dans quelques années, et si nous sommes vraiment chanceux cela se produirait encore une fois par la suite. Autrement dit, cela serait arrivé 8 fois en total. Donc la question est, quel âge ai-je maintenant ?

Écrivez un programme Python qui recherche des solutions à ce casse-tête. Indice : vous pourriez trouver utile la méthode de chaîne de caractères zfill.

Solution : **cartalk3.py**.

10 - Listes

Ce chapitre présente l'un des types internes les plus utiles de Python, les listes. Vous pourrez également en apprendre plus sur les objets et ce qui peut arriver lorsque vous avez plus d'un nom pour le même objet.

10-1 - Une liste est une séquence

Tout comme une chaîne, une **liste** est une séquence de valeurs. Dans une chaîne, les valeurs sont des caractères ; dans une liste, elles peuvent être de n'importe quel type. Les valeurs dans la liste sont appelées **éléments**.

Il existe plusieurs façons de créer une nouvelle liste ; la plus simple est d'entourer les éléments par des crochets ([et]) :

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

Le premier exemple est une liste de quatre entiers. Le second est une liste de trois chaînes de caractères (décrivant les ingrédients appétissants entrant dans la fabrication de confiseries chocolatées dans un sketch des Monty Python : grenouille croquante, vessie de bélier et vomi d'alouette - NdT). Les éléments de la liste ne doivent pas nécessairement être du même type. La liste suivante contient une chaîne de caractères, un flottant, un entier, et (tiens !) une autre liste :

```
['spam', 2.0, 5, [10, 20]]
```

Une liste à l'intérieur d'une autre liste est dite **imbriquée**.

Une liste qui ne contient aucun élément s'appelle une liste vide ; vous pouvez en créer une avec des crochets vides, [].

Comme vous pourriez vous en douter, vous pouvez affecter des valeurs aux variables de la liste :

```
>>> fromages = ['Cheddar', 'Edam', 'Gouda']
>>> nombres = [42, 123]
>>> vide = []
>>> print(fromages, nombres, vide)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

10-2 - Les listes sont modifiables

La syntaxe pour accéder aux éléments de la liste est la même que pour accéder aux caractères d'une chaîne - l'opérateur [] d'indexation. L'expression placée entre crochets spécifie l'indice.Rappelez-vous que les indices commencent à 0 :

```
>>> fromages[0]
'Cheddar'
```

Contrairement aux chaînes de caractères, les listes sont modifiables. Lorsque l'opérateur d'indexation apparaît du côté gauche de l'affectation, il identifie l'élément de la liste auquel la valeur sera affectée.

```
>>> nombres = [42, 123]
>>> nombres[1] = 5
>>> nombres
[42, 5]
```

L'un-ième élément de nombres, qui avait la valeur 123, a maintenant la valeur 5.

La Figure 10.1 montre le diagramme d'état pour fromages, nombres et vide :

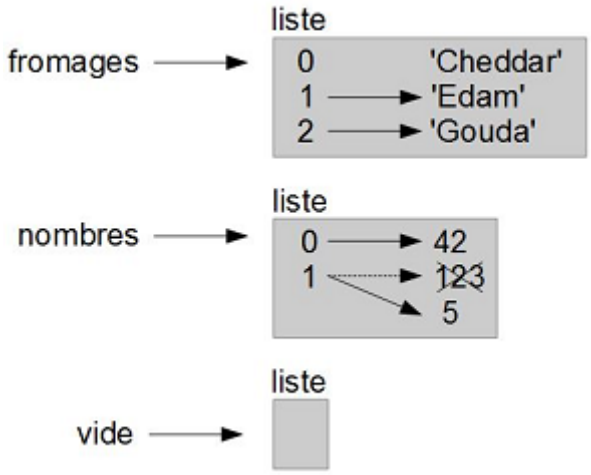


Figure 10.1 : Diagramme d'état.

Les listes y sont représentées par des structures avec le mot « liste » à l'extérieur et les éléments de la liste à l'intérieur. fromages se réfère à une liste avec trois éléments indexés 0, 1 et 2. nombres contient deux éléments ; le diagramme montre que la valeur du second élément a été réaffectée de 123 à 5. vide se réfère à une liste sans aucun élément.

Les indices de liste fonctionnent de la même manière que les indices des chaînes de caractères :

- toute expression entière peut être utilisée comme indice ;
- si vous essayez de lire ou d'écrire un élément qui n'existe pas, vous obtenez une IndexError ;
- si un index a une valeur négative, il compte en sens inverse, à partir de la fin de la liste.

L'opérateur inpeut également être utilisé sur des listes.

```
>>> fromages = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in fromages
True
>>> 'Brie' in fromages
False
```

10-3 - Parcourir une liste

La façon la plus courante de parcourir les éléments d'une liste est avec une boucle for. La syntaxe est la même que pour les chaînes de caractères :

```
for fromage in fromages:
    print(fromage)
```

Cela fonctionne bien si vous voulez juste lire les éléments de la liste. Mais si vous voulez écrire ou mettre à jour les éléments, vous avez besoin d'indices. Une façon courante de faire cela est de combiner les fonctions internes range et len :

```
for i in range(len(nombres)):
    nombres[i] = nombres[i] * 2
```

Cette boucle parcourt la liste et met à jour chaque élément. len renvoie le nombre d'éléments dans la liste. range renvoie une liste d'indices allant de 0 à *n* - 1, où *n* est la longueur de la liste. À chaque passage dans la boucle, i prend la valeur de l'indice de l'élément suivant. L'instruction d'affectation dans le corps utilise i pour lire l'ancienne valeur de l'élément et pour lui attribuer la nouvelle valeur.

Le corps d'une boucle for sur une liste vide n'est jamais exécuté :

```
for x in []:
    print("Ceci ne s'exécutera jamais.")
```

Bien que la liste puisse contenir une autre liste, la liste imbriquée compte toujours comme un seul élément. La longueur de cette liste est quatre :

```
['spam', 1, ['Brie', 'Roquefort', 'Cheddar'], [1, 2, 3]]
```

10-4 - Opérations sur listes

L'opérateur + concatène des listes :

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

L'opérateur * répète une liste un nombre donné de fois :

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Le premier exemple répète [0] quatre fois. Le second exemple répète la liste [1, 2, 3] trois fois.

10-5 - Tranches de liste

L'opérateur de tranche [m:n] peut également être utilisé sur les listes :

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Si vous omettez le premier indice, la tranche commence au début de la liste. Si vous omettez le second, la tranche s'étend jusqu'à la fin de la liste. Donc, si vous omettez les deux, la tranche est une copie de la liste entière.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Puisque les listes sont modifiables, il est souvent utile de faire une copie avant d'effectuer les opérations qui modifient des listes.

Un opérateur de tranche du côté gauche de l'affectation peut mettre à jour plusieurs éléments :

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

Rappelons que l'opérateur de tranche de Python inclut l'élément de départ, mais exclut l'élément de fin de la tranche, si bien que la tranche [1:5] correspond à [1,2,3,4] (NdT).

10-6 - Méthodes de listes

Python fournit des méthodes qui agissent sur les listes. Par exemple, append ajoute un nouvel élément à la fin d'une liste :

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

extend prend comme argument une liste et ajoute tous les éléments de celle-ci :

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

L'exemple ci-dessus ne modifie pas t2.

sortarrange les éléments d'une liste en ordre croissant :

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

La majorité des méthodes de liste sont « vides » : elles modifient la liste et renvoient None. Si vous écrivez accidentellement t = t.sort(), vous serez déçu par le résultat.

10-7 - Mapper, filtrer et réduire

Pour additionner tous les nombres d'une liste, vous pouvez utiliser une boucle de cette façon :

```
def additionner_tout(t):
    total = 0
    for x in t:
        total += x
    return total
```

total est initialisée à 0. À chaque passage dans la boucle, la valeur d'un élément de la liste est additionnée à la valeur de x. L'opérateur += fournit un raccourci pour mettre à jour une variable. Cette **instruction d'affectation augmentée** :

```
total += x
```

est équivalente à :

```
total = total + x
```

Lors de l'exécution de la boucle, total accumule la somme des éléments ; une variable utilisée de cette façon s'appelle parfois un **accumulateur**.

L'addition des éléments d'une liste est une opération si commune que Python offre comme une fonction interne, sum :

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Une opération de ce genre qui combine une séquence d'éléments en une seule valeur s'appelle parfois une **réduction**.

Parfois, vous voulez parcourir une liste en construisant une autre. Par exemple, la fonction suivante prend une liste de chaînes de caractères et renvoie une nouvelle liste qui contient ces chaînes en majuscules :

```
def mettre_tout_en_majuscules(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res est initialisée à une liste vide ; à chaque tour de boucle, nous lui ajoutons l'élément suivant. Donc res est un autre genre d'accumulateur.

Une opération comme mettre_tout_en_majuscules est appelée parfois un **map**, car elle « mappe » (ou applique) une fonction (dans ce cas, la méthode capitalize) sur chacun des éléments d'une séquence.

Une autre opération commune est de sélectionner certains des éléments d'une liste et de renvoyer une sous-liste. Par exemple, la fonction suivante prend une liste de chaînes de caractères et retourne une liste qui contient seulement les chaînes en majuscules :

```
def seulement_majuscules(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper est une méthode de chaîne de caractères qui renvoie True si la chaîne ne contient que des lettres majuscules.

Une opération comme seulement_majuscules s'appelle un **filtre**, car elle sélectionne certains des éléments et filtre les autres.

La plupart des opérations communes de liste peuvent être exprimées comme une combinaison de mappage, filtrage et réduction.

10-8 - Supprimer des éléments

Il existe plusieurs façons de supprimer des éléments d'une liste. Si vous connaissez l'indice de l'élément que vous voulez supprimer, vous pouvez utiliser pop :

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

pop modifie la liste et renvoie l'élément qui a été supprimé. Si vous ne fournissez aucun indice, elle supprime le dernier élément et renvoie sa valeur.

Si vous n'avez pas besoin de la valeur supprimée, vous pouvez utiliser l'opérateur del :

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

Si vous connaissez l'élément que vous souhaitez supprimer (mais pas son indice), vous pouvez utiliser remove :

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

La valeur de retour de remove est None.

Pour supprimer plus d'un élément, vous pouvez utiliser del avec un indice de tranche :

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

Comme d'habitude, la tranche sélectionne tous les éléments jusqu'au second indice, mais sans l'inclure.

10-9 - Listes et chaînes de caractères

Une chaîne est une séquence de caractères et une liste est une séquence de valeurs, mais une liste de caractères n'est pas la même chose qu'une chaîne de caractères. Pour convertir une chaîne en une liste de caractères, vous pouvez utiliser la fonction list :

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Comme list est le nom d'une fonction interne, vous devriez éviter de l'utiliser comme nom de variable. J'évite également d'utiliser la lettre l, car elle ressemble trop au chiffre 1. Voilà pourquoi j'utilise t.

La fonction list décompose une chaîne de caractères en lettres individuelles. Si vous voulez découper une chaîne en mots, vous pouvez utiliser la méthode split :

```
>>> s = 'se languir de fjords'
>>> t = s.split()
>>> t
['se', 'languir', 'de', 'fjords']
```

Un argument optionnel appelé **délimiteur** ou parfois séparateur spécifie quels caractères utiliser comme limites de mots. L'exemple suivant utilise un trait d'union comme séparateur :

```
>>> s = 'spam-spam-spam'
>>> delimiteur = '-'
>>> t = s.split(delimiteur)
>>> t
['spam', 'spam', 'spam']
```

join est l'inverse de split. Elle prend une liste de chaînes de caractères et concatène les éléments. join est une méthode de chaîne de caractères, donc vous devez l'invoquer sur le délimiteur et passer la liste en paramètre :

```
>>> t = ['se', 'languir', 'de', 'fjords']
>>> delimiteur = ' '
>>> s = delimiteur.join(t)
>>> s
'se languir de fjords'
```

Dans ce cas, le séparateur est un caractère espace, donc join insère un espace entre les mots. Pour concaténer des chaînes de caractères sans espaces, vous pouvez utiliser la chaîne vide, '', comme délimiteur.

10-10 - Objets et valeurs

Si nous exécutons ces instructions d'affectation :

```
a = 'banane'
b = 'banane'
```

Nous savons que a et b se réfèrent tous les deux à une chaîne de caractères, mais nous ignorons s'il s'agit de *la même* chaîne. Il existe deux états possibles, présentés dans la figure 10.2.

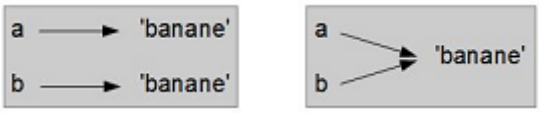


Figure 10.2 : Diagramme d'état.

Dans l'un des cas, a et b se réfèrent à deux objets différents qui ont la même valeur. Dans le second cas, ils se réfèrent au même objet.

Pour vérifier si deux variables font référence au même objet, vous pouvez utiliser l'opérateur is.

```
>>> a = 'banane'
>>> b = 'banane'
>>> a is b
True
```

Dans cet exemple, Python a créé un seul objet chaîne de caractères et tant a que b se réfèrent à lui. Mais lorsque vous créez deux listes, vous obtenez deux objets :

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Ainsi, le diagramme d'état ressemble à la figure 10.3.

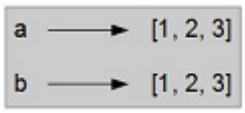


Figure 10.3 : Diagramme d'état.

Dans ce cas, nous dirions que les deux listes sont **équivalentes**, parce qu'elles ont les mêmes éléments, mais pas **identiques**, car il ne s'agit pas d'un même objet. Si deux objets sont identiques, ils sont aussi équivalents, mais s'ils sont équivalents, ils ne sont pas nécessairement identiques.

Jusqu'à présent, nous avons utilisé « objet » et « valeur » de façon interchangeable, mais dire qu'un objet a une valeur est plus exact. Si vous évaluez [1, 2, 3], vous obtenez un objet liste dont la valeur est une séquence d'entiers. Si une autre liste a les mêmes éléments, on dit qu'elle a la même valeur, mais ce n'est pas le même objet.

10-11 - Aliasing

Si a se réfère à un objet et si vous écrivez b = a, alors les deux variables font référence au même objet :

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Le diagramme d'état ressemble à la figure 10.4.

Figure 10.4 : Diagramme d'état.

L'association entre une variable et un objet s'appelle une **référence**. Dans cet exemple, il existe deux références vers le même objet.

Un objet ayant plus d'une référence a plus d'un nom, donc nous disons que l'objet est un **alias**.

Si l'alias de l'objet est mutable, les modifications apportées à un alias affectent l'autre :

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Bien que ce comportement puisse être utile, il est source d'erreurs. En général, il est plus sûr d'éviter l'aliasing lorsque vous travaillez avec des objets modifiables.

Pour les objets immuables comme les chaînes, l'aliasing ne pose pas autant de problèmes. Dans cet exemple :

```
a = 'banane'
b = 'banane'
```

cela ne change presque rien si a et b se réfèrent à la même chaîne de caractères ou non.

10-12 - Arguments de type liste

Lorsque vous passez une liste à une fonction, la fonction reçoit une référence vers la liste. Si la fonction modifie la liste, l'appelant voit la modification. Par exemple, supprimer_premier supprime le premier élément d'une liste :

```
def supprimer_premier(t):
    del t[0]
```

Voici comment elle est utilisée :

```
>>> lettres = ['a', 'b', 'c']
>>> supprimer_premier(lettres)
>>> lettres
['b', 'c']
```

Le paramètre t et la variable lettres sont des alias pour le même objet. Le diagramme de pile ressemble à la figure 10.5.

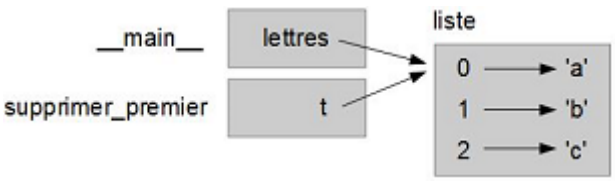


Figure 10.5 : Diagramme de pile.

Comme la liste est partagée par deux structures, je l'ai représentée entre les deux.

Il est important de faire la distinction entre les opérations qui modifient des listes et les opérations qui créent de nouvelles listes. Par exemple, la méthode append modifie une liste, mais l'opérateur + crée une nouvelle liste :

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append modifie la liste et renvoie None.

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

L'opérateur + crée une nouvelle liste et laisse la liste initiale inchangée.

Cette différence est importante lorsque vous écrivez des fonctions qui sont censées modifier des listes. Par exemple, cette fonction *ne supprime pas* le premier élément d'une liste :

```
def mauvaise_supprimer_premier(t):
    t = t[1:]              # ERRONÉ !
```

L'opérateur de tranche crée une nouvelle liste et l'affectation fait de t la référence de cette liste, mais cela n'a aucun impact sur l'appelant.

```
>>> t4 = [1, 2, 3]
>>> mauvaise_supprimer_premier(t4)
>>> t4
[1, 2, 3]
```

Au début de mauvaise_supprimer_premier, t et t4 se réfèrent à la même liste. À la fin, t se réfère à une nouvelle liste, mais t4 fait toujours référence à la liste originale non modifiée.

Une autre possibilité consiste à écrire une fonction qui crée et retourne une nouvelle liste. Par exemple, tail renvoie tous les éléments d'une liste, sauf le premier :

```
def tail(t):
    return t[1:]
```

Cette fonction laisse la liste originale non modifiée. Voici comment l'utiliser :

```
>>> lettres = ['a', 'b', 'c']
>>> reste = tail(lettres)
>>> reste
['b', 'c']
```

10-13 - Débogage

L'utilisation négligente des listes (et d'autres objets mutables) peut conduire à de longues heures de débogage. Voici quelques pièges courants et des moyens de les éviter.

1 La majorité des méthodes de liste modifient l'argument et retournent None. Les méthodes de chaînes, en revanche, renvoient une nouvelle chaîne et laissent l'original inchangé.
Si vous êtes habitué à écrire pour les chaînes de caractères du code comme ceci :

```
mot = mot.strip ()
```

Comme sort renvoie None, il est très probable que la prochaine opération que vous effectuerez avec t échouera. Avant d'utiliser des méthodes et des opérateurs de listes, vous devez lire attentivement la documentation et ensuite les tester en mode interactif.

2 Choisissez une syntaxe et tenez-vous-y.
Une partie du problème avec les listes est qu'il y a trop de façons de faire les choses. Par exemple, pour supprimer un élément d'une liste, vous pouvez utiliser pop, remove, del, ou même une affectation de tranche. Pour ajouter un élément, vous pouvez utiliser la méthode append ou l'opérateur +. En supposant que t est une liste et x est un élément de la liste, ces instructions sont correctes :

```
t.append(x)
t = t + [x]
t += [x]
```

Et celles-ci sont erronées :

```
t.append([x])           # ERRONÉ !
t = t.append(x)         # ERRONÉ !
t + [x]                 # ERRONÉ !
t = t + x               # ERRONÉ !
```

Essayez chacun de ces exemples en mode interactif pour vous assurer que vous comprenez ce qu'ils font. Notez que seule la dernière provoque une erreur d'exécution ; les trois autres sont permises, mais elles ne font pas ce qui est recherché ici.

3 Faites des copies pour éviter aliasing.
Si vous souhaitez utiliser une méthode qui modifie l'argument, telle que sort, mais vous devez garder également la liste originale, vous pouvez faire une copie.

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

Dans cet exemple, vous pouvez également utiliser la fonction interne sort, qui retourne une nouvelle liste triée et laisse l'originale inchangée.

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

10-14 - Glossaire

- **liste :** une séquence de valeurs.
- **élément :** une des valeurs d'une liste (ou une autre séquence), également appelées items.
- **liste imbriquée :** une liste qui est un élément d'une autre liste.
- **accumulateur :** une variable utilisée dans une boucle pour additionner ou accumuler un résultat.
- **affectation augmentée :** une instruction qui met à jour la valeur d'une variable en utilisant un opérateur comme +=.

- **réduire :** un modèle de traitement qui parcourt une séquence et accumule les éléments dans un seul résultat.
- **mapper :** un modèle de traitement qui parcourt une séquence et effectue une opération sur chaque élément.
- **filtrer :** un modèle de traitement qui parcourt une liste et sélectionne les éléments qui satisfont certains critères.
- **objet :** quelque chose auquel une variable peut se référer. Un objet a un type et une valeur.
- **équivalent :** ayant la même valeur.
- **identique :** être le même objet (ce qui implique l'équivalence).
- **référence :** l'association entre une variable et sa valeur.
- **aliasing :** une circonstance où deux ou plusieurs variables font référence au même objet.
- **délimiteur :** un caractère ou une chaîne indiquant où une chaîne devrait être scindée.

10-15 - Exercices

Vous pouvez télécharger les solutions de ces exercices à l'adresse **list_exercises.py**.

Exercice 1

Écrivez une fonction appelée `nested_sum` *qui prend une liste de listes d'entiers et additionne les éléments de toutes les listes imbriquées. Par exemple :*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercice 2

Écrivez une fonction appelée `cumsum` *qui prend une liste de nombres et renvoie la somme cumulative ; c'est-à-dire une nouvelle liste où le n-ième élément est la somme des premiers* n + *1 éléments de la liste originale. Par exemple :*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Exercice 3

Écrivez une fonction appelée `middle` *qui prend une liste et renvoie une nouvelle liste qui contient tous les éléments, sauf le premier et le dernier. Par exemple :*

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercice 4

Écrivez une fonction appelée `chop` *qui prend une liste, la modifie en supprimant le premier et le dernier élément, et retourne* `None`*. Par exemple :*

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercice 5

Écrivez une fonction appelée `is_sorted` *qui prend une liste comme paramètre et renvoie* `True` *si la liste est triée par ordre croissant et* `False` *sinon. Par exemple :*

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercice 6

Deux mots sont des anagrammes si vous pouvez réarranger les lettres de l'un pour en former l'autre (par exemple ALEVIN et NIVELA sont des anagrammes). Écrivez une fonction appelée is_anagram *qui prend deux chaînes et renvoie* True *si ce sont des anagrammes.*

Exercice 7

Écrivez une fonction appelée has_duplicates *qui prend une liste et renvoie* True *s'il y a au moins un élément qui apparaît plus d'une fois. La méthode ne devrait pas modifier la liste originale.*

Exercice 8

Cet exercice est relatif à ce que l'on appelle le paradoxe des anniversaires, au sujet duquel vous pouvez lire sur **https://fr.wikipedia.org/wiki/Paradoxe_des_anniversaires** .

S'il y a 23 étudiants dans votre classe, quelles sont les chances que deux d'entre vous aient le même anniversaire ? Vous pouvez estimer cette probabilité en générant des échantillons aléatoires de 23 anniversaires et en vérifiant les correspondances. Indice : vous pouvez générer des anniversaires aléatoires avec la fonction randint *du module* random.

Vous pouvez télécharger ma solution à l'adresse **birthday.py**.

Exercice 9

Écrivez une fonction qui lit le fichier mots.txt *du chapitre précédent et construit une liste avec un élément par mot. Écrivez deux versions de cette fonction, l'une qui utilise la méthode* append *et l'autre en utilisant la syntaxe* t = t + [x]. *Laquelle prend plus de temps pour s'exécuter ? Pourquoi ? Solution :* **wordlist.py**.

Exercice 10

Pour vérifier si un mot se trouve dans la liste de mots, vous pouvez utiliser l'opérateur in *, mais cela serait lent, car il vérifie les mots un par un dans l'ordre de leur apparition.*

Si les mots sont dans l'ordre alphabétique, nous pouvons accélérer les choses avec une recherche dichotomique (aussi connue comme recherche binaire), qui est similaire à ce que vous faites quand vous recherchez un mot dans le dictionnaire. Vous commencez au milieu et vérifiez si le mot que vous recherchez vient avant le mot du milieu de la liste. Si c'est le cas, vous recherchez de la même façon dans la première moitié de la liste. Sinon, vous regardez dans la seconde moitié.

Dans les deux cas, vous divisez en deux l'espace de recherche restant. Si la liste de mots a 130 557 mots, il faudra environ 17 étapes pour trouver le mot ou conclure qu'il n'y est pas.

Écrivez une fonction appelée in_bisect *qui prend une liste triée et une valeur cible et renvoie l'index de la valeur dans la liste si elle s'y trouve, ou* None *si elle n'y est pas. N'oubliez pas qu'il faut préalablement trier la liste par ordre alphabétique pour que cet algorithme puisse fonctionner ; vous gagnerez du temps si vous commencez par trier la liste en entrée et la stockez dans un nouveau fichier (vous pouvez utiliser la fonction* sort *de votre système d'exploitation si elle existe, ou sinon le faire en Python), vous n'aurez ainsi besoin de le faire qu'une seule fois.*

Ou alors vous pouvez lire la documentation du module bisect *et l'utiliser ! Solution :* **inlist.py**.

Exercice 11

Un mot est anacyclique si l'on peut le lire à l'envers ou à l'endroit, généralement la signification étant différente selon le sens de lecture. (Par exemple, « NEZ » et « ZEN », « REGAGNER » et « RENGAGER », ou « LAMINA » et « ANIMAL ») Écrivez un programme qui trouve tous les **anacycliques** *dans la liste de mots. Indice 1: utilisez la recherche dichotomique étudiée à l'exercice précédent pour recherche un mot dans une liste. Indice 2 : il y a près de 300 mots de plus de deux lettres de ce type dans notre liste. Solution :* **reverse_pair.py**.

Exercice 12

Deux mots s'« entrelacent » si vous pouvez former un nouveau mot en prenant des lettres alternées de chacun. Par exemple, les mots « FADE » et « RUES » s' « entrelacent » pour former « FRAUDEES ». Indice: les mots « composants » n'ont pas nécessairement le même nombre de lettres, le premier peut avoir une lettre de plus que le second. Ainsi, les mots « ACRES » et « CODE » peuvent former « ACCORDEES ». Solution : **interlock.py**. *Référence : Cet exercice est inspiré par un exemple sur* **http://puzzlers.org**.

1. *Écrivez un programme qui trouve toutes les paires de mots qui s'« entrelacent » . Indice : ne pas énumérer toutes les paires, il y en a plus de 400 en n'acceptant que des « composants » d'au moins 3 lettres !*
2. *Pouvez-vous trouver tous les trios de mots qui s'« entrelacent » par trois ; c'est-à-dire que, en partant de la première, deuxième ou troisième lettre du mot résultat, on obtient trois mots en prenant une lettre toutes les trois lettres ? Indice 1 : la formulation employée ici est susceptible de faire penser à un algorithme bien plus efficace : n'essayez pas toutes les combinaisons de mots courts pour trouver des mots longs, ce pourrait être très lent ; partez plutôt des mots longs et voyez si vous pouvez les décomposer trois mots appartenant à la liste. Indice 2 : contrairement à ce que l'on pourrait penser de prime abord, il y en a beaucoup (plus de 700).*

11 - Dictionnaires

Ce chapitre présente un autre type interne appelé dictionnaire. Les dictionnaires sont une des meilleures fonctionnalités de Python ; ce sont les blocs de construction de nombreux algorithmes efficaces et élégants.

11-1 - Un dictionnaire est un mappage

Un dictionnaire ressemble à une liste, mais il est plus général. Dans une liste, les indices doivent être des nombres entiers ; dans un dictionnaire, ils peuvent être de (presque) n'importe quel type.

Un dictionnaire contient une collection d'indices, qui sont appelés **clés**, et une collection de valeurs. Chaque clé est associée à une valeur unique. L'association entre une clé et une valeur est appelée une **paire clé-valeur** ou parfois un **item**.

En langage mathématique, un dictionnaire représente un **mappage** de clés vers des valeurs, donc vous pouvez également dire que chaque clé « correspond à » une valeur. À titre d'exemple, nous allons construire un dictionnaire qui mappe des mots français vers des mots espagnols, si bien que tant les clés que les valeurs seront des chaînes de caractères.

La fonction dict crée un nouveau dictionnaire sans aucun élément. Comme dict est le nom d'une fonction interne, vous devez éviter de l'utiliser comme nom de variable.

```
>>> fr_vers_es = dict()
>>> fr_vers_es
{}
```

Les accolades {} représentent un dictionnaire vide. Pour ajouter des éléments au dictionnaire, vous pouvez utiliser des crochets :

```
>>> fr_vers_es['un'] = 'uno'
```

Cette ligne crée un élément qui fait correspondre la clé 'un' à la valeur 'uno'. Si nous affichons à nouveau le dictionnaire, nous voyons une paire clé-valeur avec un caractère deux-points entre la clé et la valeur :

```
>>> fr_vers_es
{'un': 'uno'}
```

Ce format en sortie est également un format en entrée possible. Par exemple, vous pouvez créer un nouveau dictionnaire avec trois éléments :

```
>>> fr_vers_es = {'un': 'uno', 'deux': 'dos', 'trois': 'tres'}
```

Mais si vous affichez fr_vers_es, vous pourriez être surpris :

```
>>> fr_vers_es
{'un': 'uno', 'trois': 'tres', 'deux': 'dos'}
```

L'ordre des paires clé-valeur pourrait ne pas être le même. Si vous tapez le même exemple sur votre ordinateur, vous obtiendrez peut-être un résultat différent. En général, l'ordre des éléments dans un dictionnaire est imprévisible.

Mais cela ne pose aucun problème parce que les éléments d'un dictionnaire ne sont jamais indexés avec des indices entiers. Au lieu de cela, vous utilisez les clés pour rechercher les valeurs correspondantes :

```
>>> fr_vers_es['deux']
'dos'
```

La clé 'deux' mappe toujours vers la valeur 'dos', et l'ordre des éléments n'a donc aucune importance.

Si la clé n'existe pas dans le dictionnaire, vous obtenez une exception :

```
>>> fr_vers_es['quatre']
KeyError: 'quatre'
```

La fonction len peut être utilisée sur les dictionnaires ; elle renvoie le nombre de paires clé-valeur :

```
>>> len(fr_vers_es)
3
```

L'opérateur in aussi fonctionne sur des dictionnaires ; il vous indique si quelque chose apparaît comme une *clé* dans le dictionnaire (mais pas si cette chose figure parmi les valeurs).

```
>>> 'un' in fr_vers_es
True
>>> 'uno' in fr_vers_es
False
```

Pour voir si quelque chose apparaît comme une valeur dans un dictionnaire, vous pouvez utiliser la méthode values, qui retourne une collection de valeurs, et ensuite utiliser l'opérateur in :

```
>>> valeurs = fr_vers_es.values()
>>> 'uno' in valeurs
True
```

L'opérateur in utilise des algorithmes différents pour les listes et les dictionnaires. Pour les listes, il recherche les éléments de la liste dans l'ordre, comme nous l'avons vu dans la section **8.6**. Au fur et à mesure que la liste s'allonge, le temps de recherche augmente proportionnellement à sa taille.

Pour les dictionnaires, Python utilise un algorithme appelé **table de hachage** qui a une propriété remarquable : l'opérateur in prend à peu près le même temps, quel que soit le nombre d'éléments dans le dictionnaire. J'explique comment c'est possible dans la section **B.4** (annexe à la fin de ce livre), mais l'explication pourrait ne pas être compréhensible avant que vous n'ayez lu encore quelques chapitres.

11-2 - Un dictionnaire comme une collection de compteurs

Supposons que vous ayez une chaîne de caractères et que vous vouliez compter le nombre de fois où apparaît chaque lettre. Il y a plusieurs façons de faire cela :

1. Vous pourriez créer 26 variables, une pour chaque lettre de l'alphabet. Ensuite, vous pourriez parcourir la chaîne et, pour chaque caractère, incrémenter le compteur correspondant, probablement en utilisant des conditions enchaînées ;
2. Vous pourriez créer une liste de 26 éléments. Ensuite, vous pouvez convertir chaque caractère en un nombre (en utilisant la fonction interne ord), utiliser ce nombre comme un index de liste, et incrémenter le compteur approprié ;
3. Vous pourriez créer un dictionnaire avec des caractères comme clés et des compteurs comme valeurs correspondantes. La première fois que vous trouvez un caractère, vous ajouteriez un élément au dictionnaire. Ensuite, vous incrémenteriez la valeur d'un élément existant.

Chacune de ces options effectue le même calcul, mais chacune d'entre elles met en œuvre ce calcul d'une manière différente.

Une **implémentation** ou **mise en œuvre** est une façon pratique d'effectuer un calcul ; certaines implémentations sont meilleures qu'autres. Par exemple, un avantage de la mise en œuvre du dictionnaire est que nous n'avons pas besoin de savoir à l'avance quelles lettres apparaissent dans la chaîne de caractères et nous ne devons réserver de la place mémoire que pour les lettres qui apparaissent réellement.

Voici à quoi pourrait ressembler le code :

```
def histogramme(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Le nom de la fonction est histogramme, qui est un terme statistique pour une collection de compteurs (ou de fréquences).

La première ligne de la fonction crée un dictionnaire vide. La boucle for parcourt la chaîne. À chaque passage dans la boucle, si le caractère c n'est pas dans le dictionnaire, nous allons créer un nouvel élément ayant la clé c et la valeur initiale 1 (puisque nous avons trouvé cette lettre une fois). Si c existe déjà dans le dictionnaire, alors on incrémente d[c].

Voici comment cela fonctionne :

```
>>> h = histogramme('brontosaure')
>>> h
{'a': 1, 'b': 1, 'e': 1, 'o': 2, 'n': 1, 's': 1, 'r': 2, 'u': 1, 't': 1}
```

L'histogramme indique que les lettres 'a', 'b' et 'e' apparaissent une fois ; 'o' apparaît deux fois, et ainsi de suite.

Les dictionnaires ont une méthode appelée get qui prend une clé et une valeur par défaut. Si la clé apparaît dans le dictionnaire, get renvoie la valeur correspondante ; sinon, elle renvoie la valeur par défaut. Par exemple :

```
>>> h = histogramme('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

À titre d'exercice, utilisez la méthode get pour écrire histogramme de façon plus concise. Vous devriez être en mesure d'éliminer l'instruction if.

11-3 - Boucles et dictionnaires

Si vous utilisez un dictionnaire dans une instruction for, elle parcourt les clés du dictionnaire. Par exemple, print_hist affiche chaque clé et la valeur correspondante :

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Voici à quoi ressemble l'affichage :

```
>>> h = histogramme('perroquet')
>>> print_hist(h)
e 2
o 1
q 1
p 1
r 2
u 1
t 1
```

Encore une fois, les clés ne sont dans aucun ordre particulier. Pour parcourir les clés dans l'ordre trié, vous pouvez utiliser la fonction interne sorted :

```
>>> for key in sorted(h):
...     print(key, h[key])
'e', 2
'o', 1
'p', 1
'q', 1
'r', 2
't', 1
'u', 1
```

11-4 - Recherche inversée

Étant donné un dictionnaire d et une clé k, il est facile de trouver la valeur correspondante v = d[k]. Cette opération s'appelle une **recherche**.

Mais que faire si vous avez v et que vous voulez trouver k ? Vous avez deux problèmes : d'abord, il pourrait y avoir plus d'une clé qui correspond à la valeur v. En fonction de l'application, vous pourriez être en mesure d'en choisir une, ou vous pourriez avoir à faire une liste qui les contient toutes. Deuxièmement, il n'y a aucune syntaxe simple pour faire une **recherche inversée** ; vous devez faire une recherche en parcourant les éléments un par un.

Voici une fonction qui prend une valeur et renvoie la première clé qui correspond à cette valeur :

```
def recherche_inverse(d, v):
    for k in d:
        if d[k] == v:
```

```
            return k
    raise LookupError()
```

Cette fonction est encore un autre exemple du modèle de recherche, mais elle utilise une fonctionnalité que nous n'avons pas vue avant, raise. L'**instruction** raise provoque une exception ; dans ce cas, elle génère une LookupError, qui est une exception interne utilisée pour indiquer qu'une opération de recherche a échoué.

Si nous arrivons à la fin de la boucle, cela signifie que v ne figure pas comme une valeur dans le dictionnaire, donc nous générons une exception.

Voici un exemple d'une recherche inversée réussie :

```
>>> h = histogramme('perroquet')
>>> key = recherche_inverse(h, 2)
>>> key
'e'
```

Et une qui a échoué :

```
>>> key = recherche_inverse(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in recherche_inverse
LookupError
```

L'effet lorsque vous générez une exception est le même que lorsque Python en génère une : une trace et un message d'erreur sont affichés.

L'instruction raise peut prendre un message d'erreur détaillé comme un argument optionnel. Par exemple :

```
>>> raise LookupError('valeur inexistante dans le dictionnaire')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: valeur inexistante dans le dictionnaire
```

Une recherche inversée est beaucoup plus lente qu'une recherche directe ; si vous devez le faire souvent, ou si le dictionnaire devient grand, la performance de votre programme se détériorera.

11-5 - Dictionnaires et listes

Des listes peuvent apparaître comme valeurs dans un dictionnaire. Par exemple, si vous avez un dictionnaire qui fait correspondre des lettres à des fréquences, vous voudrez peut-être l'inverser ; autrement dit, créer un dictionnaire qui fait correspondre des fréquences aux lettres. Comme il se peut qu'il y ait plusieurs lettres ayant la même fréquence, chaque valeur dans le dictionnaire inversé doit être une liste de lettres.

Voici une fonction qui inverse un dictionnaire :

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

À chaque passage dans la boucle, key reçoit une clé du dictionnaire d et val reçoit la valeur correspondante. Si val ne se trouve pas dans inverse, cela signifie que nous ne l'avons pas encore rencontrée, donc nous créons un nouvel

élément et l'initialisons avec un **singleton** (une liste qui contient un seul élément). Sinon, nous avons déjà rencontré cette valeur, alors nous ajoutons la clé correspondante à la fin de la liste.

Voici un exemple :

```
>>> hist = histogramme('perroquet')
>>> hist
{'e': 2, 'o': 1, 'q': 1, 'p': 1, 'r': 2, 'u': 1, 't': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['o', 'q', 'p', 'u', 't'], 2: ['e', 'r']}
```

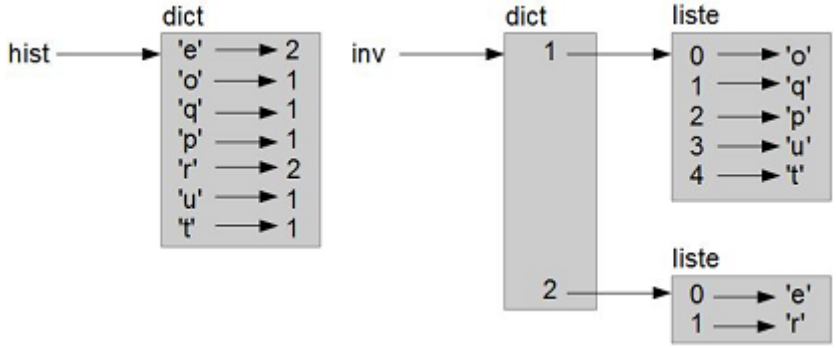


Figure 11.1 : Diagramme d'état.

La Figure 11.1 est un diagramme d'état montrant hist et inverse. Un dictionnaire est représenté comme un rectangle avec le type dict au-dessus et les paires clé-valeur à l'intérieur. Si les valeurs sont des nombres entiers, des flottants ou des chaînes de caractères, je les dessine à l'intérieur du rectangle, mais généralement je dessine les listes en dehors du rectangle, juste pour garder le schéma simple.

Les listes peuvent être des valeurs dans un dictionnaire, comme le montre cet exemple, mais elles ne peuvent pas être des clés. Voici ce qui se passe si vous essayez :

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oups'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

J'ai mentionné plus tôt qu'un dictionnaire est mis en œuvre en utilisant une table de hachage et cela signifie que les clés doivent être **hachables**.

Une fonction de **hachage** est une fonction qui prend une valeur (de n'importe quel type) et retourne un entier. Les dictionnaires utilisent ces entiers, appelés valeurs de hachage, pour stocker et rechercher des paires clé-valeur.

Ce système fonctionne très bien si les clés sont immuables. Mais si les clés sont modifiables, comme les listes, de gros problèmes se posent. Par exemple, lorsque vous créez une paire clé-valeur, Python hache la clé et la stocke dans l'emplacement correspondant. Si vous modifiez la clé, puis la hachez à nouveau, elle ira à un endroit différent. Dans ce cas, vous pourriez avoir deux entrées pour la même clé, ou vous pourriez ne pas être en mesure de trouver une clé. Quoi qu'il arrive, le dictionnaire ne fonctionnera pas correctement.

Voilà pourquoi les clés doivent être « hachables », et pourquoi les types modifiables comme les listes ne le sont pas. La façon la plus simple de contourner cette limitation est d'utiliser des tuples, que nous verrons dans le chapitre suivant.

Puisque les dictionnaires sont modifiables, ils ne peuvent pas servir de clés, mais ils *peuvent* être utilisés comme valeurs.

11-6 - Mémos

Si vous avez joué un peu avec la fonction fibonacci de la section **6.7**, vous avez peut-être remarqué que plus l'argument que vous fournissez est grand, plus la durée d'exécution de la fonction augmente. En outre, la durée d'exécution augmente très rapidement :

Exemples de durée d'exécution de la recherche de nombre de Fibonacci		Nombre de Fibonacci	Durée d'exécution
20	0m0.109s		
30	0m1.295s		
40	2m28.965s		
100 (estimation)	Milliards d'années		

Pour comprendre pourquoi, examinons la figure 11.2, qui montre le **graphe des appels** pour fibonacci avec n = 4 :

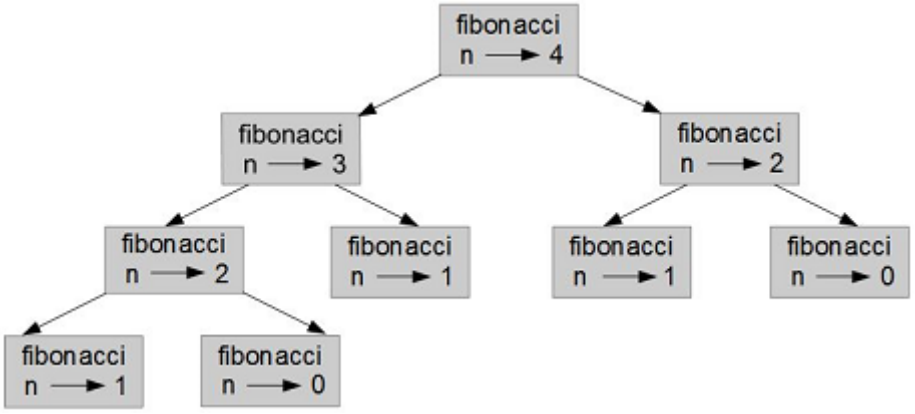


Figure 11.2 :Graphe des appels.

Un graphe d'appels montre un ensemble de structures de fonction, avec des lignes reliant chaque structure aux structures des fonctions qu'elle appelle. Au sommet du graphe, fibonacci avec n = 4 appelle fibonacci avec n = 3 et avec n = 2. À son tour, fibonacci avec n = 3 appelle fibonacci avec n = 2 et avec n = 1. Et ainsi de suite.

Comptez le nombre de fois où fibonacci(0) et fibonacci(1) sont appelées. C'est une solution inefficace du problème, et cela devient pire au fur et à mesure que l'argument devient plus grand.

Une solution consiste à garder la trace des valeurs qui ont déjà été calculées en les stockant dans un dictionnaire. Une valeur calculée précédemment qui est stockée pour une utilisation ultérieure est appelée un **mémo** (ou un *cache*). Voici une version « mémoïsée » ou avec mise en cache de fibonacci :

```
connu = {0:0, 1:1}

def fibonacci(n):
    if n in connu:
        return connu[n]

    res = fibonacci(n - 1) + fibonacci(n - 2)
    connu[n] = res
    return res
```

connu est un dictionnaire qui permet de garder la trace des nombres de Fibonacci que nous connaissons déjà. Il commence par deux éléments : 0 mappe vers 0 et 1 mappe vers 1.

Chaque fois que fibonacci est appelée, elle vérifie connu. Si le résultat s'y trouve déjà, elle peut le renvoyer immédiatement. Sinon, elle doit calculer la nouvelle valeur, l'ajouter au dictionnaire, et la renvoyer.

Si vous exécutez cette version de fibonacci et la comparez avec l'original, vous trouverez qu'elle est beaucoup plus rapide.

11-7 - Variables globales

Dans l'exemple précédent, connu est créé en dehors de la fonction, de sorte qu'il appartient à la structure spéciale appelée __main__. Les variables dans __main__ sont parfois appelées **globales**, car elles peuvent être accédées à partir de n'importe quelle fonction. Contrairement aux variables locales, qui disparaissent lorsque leur fonction se termine, les variables globales persistent d'un appel de fonction à l'autre.

Il est courant d'utiliser des variables globales pour des **drapeaux** ou *flags* ; c'est-à-dire des variables booléennes qui indiquent (ou « signalent ») si une condition est vraie. Par exemple, certains programmes utilisent un drapeau nommé verbose (« bavard » ou « verbeux ») pour contrôler le niveau de détail dans la sortie :

```
verbose = True

def exemple1():
    if verbose:
        print('Exécuter exemple1')
```

Si vous essayez de réaffecter une variable globale, vous pourriez être surpris. L'exemple suivant est censé garder une trace afin de savoir si la fonction a été appelée :

```
est_appelee = False

def exemple2():
    est_appelee = True          # ERRONÉ
```

Mais si vous l'exécutez, vous verrez que la valeur de est_appelee ne change pas. Le problème est que exemple2 crée une nouvelle variable locale nommée est_appelee. La variable locale disparaît lorsque la fonction se termine, et n'a aucun effet sur la variable globale.

Pour réaffecter une variable globale dans une fonction, vous devez **déclarer** la variable comme étant globale avant de l'utiliser :

```
est_appelee = False

def exemple2():
    global est_appelee
    est_appelee = True
```

L'**instruction** global dit à peu près ceci à l'interpréteur : « Dans cette fonction, quand je dis est_appelee, je parle de la variable globale ; n'en crée pas une version locale. »
Voici un exemple qui tente de mettre à jour une variable globale :

```
compteur = 0

def exemple3():
    compteur = compteur + 1     # ERRONÉ
```

Si vous l'exécutez, vous obtenez :

```
UnboundLocalError: local variable 'compteur' referenced before assignment
```

Python suppose que compteur est une variable locale, et, selon cette hypothèse, que vous lisez celle-ci avant de l'écrire. La solution, encore une fois, est de déclarer la variable compteur comme étant globale.

```
def exemple3():
```

```
    global compteur
    compteur += 1
```

Si une variable globale fait référence à une valeur modifiable (par exemple l'élément d'une liste ou d'un dictionnaire), vous pouvez modifier la valeur sans déclarer la variable :

```
connu = {0:0, 1:1}

def exemple4():
    connu[2] = 1
```

Ainsi, vous pouvez ajouter, supprimer et remplacer des éléments d'une liste ou un dictionnaire global, mais si vous voulez réaffecter la variable, vous devez déclarer cela :

```
def exemple5():
    global connu
    connu = dict()
```

Les variables globales peuvent être utiles, mais si vous en avez beaucoup, et que vous les modifiez fréquemment, elles peuvent rendre les programmes difficiles à déboguer.

11-8 - Débogage

Lorsque vous travaillez avec de plus grands ensembles de données, il peut devenir difficile de déboguer en affichant et en vérifiant la sortie à la main. Voici quelques suggestions pour déboguer de grands ensembles de données.
Réduire la taille des données en entrée

- Si possible, réduisez la taille de l'ensemble de données. Par exemple, si le programme lit un fichier texte, commencez avec seulement les 10 premières lignes, ou avec le plus petit exemple que vous puissiez trouver. Vous pouvez soit éditer les fichiers mêmes, ou (mieux) modifier le programme pour qu'il ne lise que les n premières lignes.
- Si une erreur survient, vous pouvez réduire n à la plus petite valeur qui provoque l'erreur. Ensuite, vous pouvez augmenter n progressivement au fur et à mesure que vous trouvez et corrigez les erreurs.

Vérifier les résumés et les types

- Au lieu d'afficher et de vérifier l'ensemble des données, pensez à afficher des résumés des données : par exemple, le nombre d'éléments dans un dictionnaire ou le total d'une liste de nombres.
- Une cause fréquente d'erreurs d'exécution est une valeur qui n'est pas du bon type. Pour déboguer ce genre d'erreur, souvent il suffit d'afficher le type d'une valeur.

Écrire des contrôles automatisés

- Parfois, vous pouvez écrire du code pour vérifier automatiquement les erreurs. Par exemple, si vous calculez la moyenne d'une liste de nombres, vous pourriez vérifier que le résultat n'est pas plus grand que le plus grand élément de la liste ou plus petit que le plus petit élément. Cela est appelé une « vérification de vraisemblance », car on détecte des résultats « invraisemblables ».
- Un autre type de vérification compare les résultats de deux calculs différents pour voir s'ils sont compatibles. Cela s'appelle un « contrôle de cohérence ».

Formater la sortie

- Le formatage de la sortie du débogage peut faciliter le repérage d'une erreur. Nous avons vu un exemple dans la section **6.9**. Le module pprint fournit une fonction pprint qui affiche les types internes dans un format plus lisible (pprint signifie « pretty print» - affichage joli).

Encore une fois, le temps que vous passez à construire des échafaudages peut réduire le temps que vous passez à déboguer.

11-9 - Glossaire

- **mappage :** une relation dans laquelle chaque élément d'un ensemble est associé à un élément d'un autre ensemble.
- **dictionnaire :** un mappage de clés vers leurs valeurs correspondantes.
- **paire clé-valeur :** la représentation du mappage d'une clé vers une valeur.
- **clé :** un objet qui apparaît dans un dictionnaire comme la première partie d'une paire clé-valeur.
- **valeur :** un objet qui apparaît dans un dictionnaire comme la seconde partie d'une paire clé-valeur. Il s'agit ici d'une utilisation plus spécifique que précédemment du mot « valeur ».
- **implémentation (mise en œuvre) :** une façon d'effectuer un calcul dans le cadre d'un programme informatique.
- **hashtable (table de hachage) :** l'algorithme utilisé pour mettre en œuvre des dictionnaires en Python.
- **fonction de hachage :** une fonction utilisée par une table de hachage pour calculer l'emplacement d'une clé.
- **hachable :** un type qui a une fonction de hachage. Les types immuables comme les entiers, les flottants et les chaînes de caractères sont hachables ; les types modifiables comme les listes et les dictionnaires ne le sont pas.
- **recherche :** une opération de dictionnaire qui prend une clé et trouve la valeur correspondante.
- **recherche inversée :** une opération de dictionnaire qui prend une valeur et trouve une ou plusieurs clés qui lui sont associées.
- **instruction** `raise` **:** une instruction qui génère (volontairement) une exception.
- **singleton :** une liste (ou une autre séquence) avec un seul élément.
- **graphe des appels :** un diagramme qui montre chaque structure créée pendant l'exécution d'un programme, avec une flèche de chaque appelant à chaque appelé.
- **mémo** ou **cache :** une valeur calculée stockée pour éviter de refaire inutilement le même calcul.
- **variable globale :** une variable définie en dehors d'une fonction. Les variables globales peuvent être accessibles à partir de toute fonction.
- **instruction** `global` **:** une instruction qui déclare un nom variable comme étant global.
- **drapeau** ou **flag :** une variable booléenne utilisée pour indiquer si une condition est vraie.
- **déclaration :** une instruction, comme `global`, qui transmet à l'interpréteur une information au sujet d'une variable.

11-10 - Exercices

Exercice 1

Écrivez une fonction qui lit les mots dans `mots.txt` *et les stocke comme clés dans un dictionnaire. Peu importent les valeurs. Ensuite, vous pouvez utiliser l'opérateur* `in` *comme un moyen rapide pour vérifier si une chaîne se trouve dans le dictionnaire.*

Si vous avez fait l'exercice **10** du chapitre précédent*, vous pouvez comparer la durée d'exécution de cette mise en œuvre avec celle utilisant l'opérateur de liste* `in` *et celle de la recherche dichotomique.*

Exercice 2

Lisez la documentation de la méthode de dictionnaire `setdefault` *et utilisez-la pour écrire une version plus concise de* `invert_dict`*. Solution :* **invert_dict.py**.

Exercice 3

Transformez la fonction d'Ackermann de l'exercice **2** *du chapitre 6 en une fonction « mémoïsée » et voyez si la « mémoïsation » permet d'évaluer la fonction avec des arguments plus grands. Indice : aucun. Solution :* **ackermann_memo.py**.

Exercice 4

Si vous avez fait l'exercice 7 du chapitre 10, vous avez déjà une fonction nommée has_duplicates *qui prend comme paramètre une liste et renvoie* True *s'il existe un objet qui apparaît plus d'une fois dans la liste.*

Utilisez un dictionnaire pour écrire une version plus rapide et plus simple de has_duplicates. *Solution :* **has_duplicates.py**.

Exercice 5

Deux mots forment « une paire par rotation » si vous pouvez effectuer la rotation d'un d'entre eux avec un chiffre de César pour en obtenir l'autre (voir rotate_word *dans l'exercice 5 du chapitre 8).*

Écrivez un programme qui lit une liste de mots et trouve toutes les paires par rotation. Solution : **rotate_pairs.py**.

Exercice 6

Trouvez, dans notre liste de mots, des mots de cinq lettres tels que, si vous retirez la première lettre, le nouveau mot de quatre lettres obtenu figure également dans notre liste, et que si vous retirez la nouvelle première lettre, le nouveau mot de trois lettres obtenu soit également un mot de notre liste. Par exemple, le mot EPILE donne PILE puis ILE, qui sont deux mots appartenant à notre liste. Utilisez un dictionnaire pour stocker les mots de la liste et faire les recherches nécessaires. Indice : il y a plus de 250 mots dans ce cas.

12 - Tuples

Ce chapitre présente un autre type interne, le tuple, et ensuite montre comment les listes, les dictionnaires et les tuples fonctionnent ensemble. Je présente également une fonctionnalité utile pour les listes d'arguments de longueur variable, les opérateurs de regroupement et de dispersion.

12-1 - Les tuples sont immuables

Un tuple est une séquence de valeurs. Les valeurs peuvent être de tout type, et elles sont indexées par des entiers ; à cet égard, les tuples ressemblent donc beaucoup aux listes. La différence importante est que les tuples sont immuables.

Syntaxiquement, un tuple est une liste de valeurs séparées par des virgules :

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Bien que ce ne soit pas nécessaire, il est courant de mettre les tuples entre des parenthèses :

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Pour créer un tuple avec un seul élément, vous devez inclure une virgule finale :

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

Une valeur entre parenthèses n'est pas un tuple :

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Une autre façon de créer un tuple est d'utiliser la fonction interne tuple. Sans aucun argument, elle crée un tuple vide :

```
>>> t = tuple()
>>> t
()
```

Si l'argument est une séquence (chaîne de caractères, liste ou tuple), le résultat est un tuple contenant les éléments de la séquence :

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Le mot tuple étant le nom d'une fonction interne, vous devez éviter de l'utiliser comme nom de variable.

La majorité des opérateurs de liste fonctionnent également sur des tuples. L'opérateur [] d'indexation indexe un élément :

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

Et l'opérateur de tranche sélectionne un ensemble d'éléments.

```
>>> t[1:3]
('b', 'c')
```

Mais si vous essayez de modifier un des éléments du tuple, vous obtenez une erreur :

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Comme les tuples sont immuables, vous ne pouvez pas modifier les éléments. Mais vous pouvez remplacer un tuple par un autre :

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

Cette instruction crée un nouveau tuple, puis fait de t sa référence.

Les opérateurs relationnels fonctionnent sur les tuples et d'autres séquences ; Python commence par comparer le premier élément de chaque séquence. S'ils sont égaux, il passe à l'élément suivant, et ainsi de suite, jusqu'à ce qu'il trouve des éléments qui diffèrent. Les éléments ultérieurs ne sont pas considérés (même s'ils sont très grands).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12-2 - Affectation de tuple

Il est souvent utile d'échanger les valeurs de deux variables. Avec des affectations classiques, vous devez utiliser une variable temporaire. Par exemple, pour intervertir a et b :

```
>>> temp = a
>>> a = b
>>> b = temp
```

Cette solution est lourde ; l'**affectation de tuple** est plus élégante :

```
>>> a, b = b, a
```

Le côté gauche de l'affectation est un tuple de variables ; le côté droit est un tuple d'expressions. Chaque valeur est affectée à sa variable respective. Toutes les expressions du côté droit sont évaluées avant toute affectation.

Le nombre de variables du côté gauche doit être identique au nombre de valeurs du côté droit :

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

De façon plus générale, le côté droit peut être tout type de séquence (chaîne de caractères, liste ou tuple). Par exemple, pour diviser une adresse de courriel dans un nom d'utilisateur et un domaine, vous pourriez écrire :

```
>>> adresse = 'monty@python.org'
>>> nom_utilisateur, domaine = adresse.split('@')
```

La valeur de retour de split est une liste de deux éléments ; le premier élément est assigné à nom_utilisateur et le second à domaine.

```
>>> nom_utilisateur
'monty'
>>> domaine
'python.org'
```

12-3 - Tuples comme valeurs de retour

Au sens strict, une fonction ne peut renvoyer qu'une seule valeur, mais si la valeur est un tuple, l'effet est le même que de renvoyer des valeurs multiples. Par exemple, si vous voulez effectuer une division entière entre deux nombres entiers et calculer le quotient et le reste, il est inefficace de calculer x / y puis x % y. Il est préférable de calculer les deux en même temps.

La fonction interne divmod prend deux arguments et retourne un tuple de deux valeurs, le quotient et le reste. Vous pouvez stocker le résultat comme un tuple :

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

Ou utiliser l'affectation de tuple pour stocker les éléments séparément :

```
>>> quot, res = divmod(7, 3)
>>> quot
2
>>> res
1
```

Voici un exemple de fonction qui retourne un tuple :

```
def min_max(t):
    return min(t), max(t)
```

max et min sont des fonctions internes qui trouvent le plus grand et le plus petit éléments d'une séquence. min_max calcule les deux et retourne un tuple de deux valeurs.

12-4 - Arguments tuples à longueur variable

Les fonctions peuvent prendre un nombre variable d'arguments. Un nom de paramètre qui commence par un symbole * **assemble** les arguments dans un tuple. Par exemple, printall prend un nombre quelconque d'arguments et les affiche :

```
def printall(*args):
    print(args)
```

Le paramètre d'assemblage peut avoir un nom quelconque, mais par convention on l'appelle souvent args. Voici comment cela fonctionne :

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Le complément de l'assemblage est la **dispersion**. Si vous avez une séquence de valeurs et que vous souhaitez la passer à une fonction sous forme de plusieurs arguments, vous pouvez utiliser l'opérateur *. Par exemple, divmod prend exactement deux arguments ; elle ne fonctionne pas sur un tuple :

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Mais si vous dispersez le tuple, cela fonctionne :

```
>>> divmod(*t)
(2, 1)
```

Beaucoup de fonctions internes utilisent des arguments tuples à longueur variable. Par exemple, max et min peuvent prendre un nombre quelconque d'arguments :

```
>>> max(1, 2, 3)
3
```

Mais sum ne le fait pas.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

À titre d'exercice, écrivez une fonction appelée sumall qui prend un nombre quelconque d'arguments et renvoie leur somme.

12-5 - Listes et tuples

zip est une fonction interne qui prend deux ou plusieurs séquences et retourne une liste de tuples où chaque tuple contient un élément de chaque séquence. Le nom de la fonction fait allusion à une fermeture-éclair (*zip* ou *zipper* en anglais), qui joint et entrelace deux rangées de dents.

Cet exemple zippe une chaîne et une liste :

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

Le résultat est un **objet zip** qui sait comment parcourir les paires. L'utilisation la plus courante de zip est dans une boucle for :

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

Un objet zip est une sorte d'**itérateur**, c'est-à-dire un objet qui parcourt une séquence. Les itérateurs ressemblent aux listes à certains égards, mais à la différence des listes, vous ne pouvez pas utiliser un index pour sélectionner un élément d'un itérateur.

Si vous souhaitez utiliser des opérateurs et méthodes de liste, vous pouvez utiliser un objet zip pour faire une liste :

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

Le résultat est une liste de tuples ; dans cet exemple, chaque tuple contient un caractère de la chaîne de caractères s et l'élément correspondant de la liste t.

Si les séquences n'ont pas la même longueur, le résultat a la longueur de la séquence la plus courte.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

Vous pouvez utiliser l'affectation de tuple dans une boucle for pour parcourir une liste de tuples :

```
t = [('a', 0), ('b', 1), ('c', 2)]
for lettre, nombre in t:
    print(nombre, lettre)
```

À chaque passage dans la boucle, Python sélectionne le prochain tuple dans la liste et affecte les éléments à lettre et nombre. Cette boucle affiche :

```
0 a
1 b
2 c
```

Si vous combinez zip, for et l'affectation de tuple, vous obtenez une syntaxe utile pour parcourir deux (ou plusieurs) séquences en même temps. Par exemple, a_correspondant prend deux séquences, t1 et t2, et renvoie True s'il existe un indice i tel que t1[i] == t2[i] :

```
def a_correspondant(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Si vous devez parcourir les éléments d'une séquence et de leurs indices, vous pouvez utiliser la fonction interne enumerate :

```
for index, element in enumerate('abc'):
    print(index, element)
```

Le résultat de enumerate est un objet énumération, qui parcourt une séquence de paires ; chaque paire contient un indice (numéroté à partir de 0) et un élément de la séquence donnée. Cet exemple affiche à nouveau :

```
0 a
1 b
2 c
```

12-6 - Dictionnaires et tuples

Les dictionnaires ont une méthode appelée items qui renvoie une séquence de tuples, où chaque tuple est une paire clé-valeur.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

Le résultat est un objet dict_items, un itérateur qui parcourt les paires clé-valeur. Vous pouvez l'utiliser dans une boucle for comme ceci :

```
>>> for key, value in d.items():
...     print(key, value)
...
c 2
a 0
b 1
```

Comme vous devriez vous attendre de la part d'un dictionnaire, les éléments ne sont dans aucun ordre particulier.

Dans l'autre sens, vous pouvez utiliser une liste de tuples pour initialiser un nouveau dictionnaire :

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

La combinaison entre dict et zip donne une manière concise de créer un dictionnaire :

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

La méthode de dictionnaire update prend également une liste de tuples et les ajoute, en tant que paires clé-valeur, à un dictionnaire existant.

Il est courant d'utiliser des tuples comme clés dans les dictionnaires (principalement parce que vous ne pouvez pas utiliser des listes). Par exemple, un annuaire téléphonique pourrait établir une correspondance entre des paires nom - prénom et des numéros de téléphone. En supposant que nous avons défini nom, prenom et numero, nous pourrions écrire :

```
annuaire[nom, prenom] = numero
```

L'expression entre crochets est un tuple. Nous pourrions utiliser l'affectation de tuple pour parcourir ce dictionnaire.

```
for nom, prenom in annuaire:
    print(nom, prenom, annuaire[nom,prenom])
```

Cette boucle parcourt les clés de annuaire, qui sont des tuples. Il attribue les éléments de chaque tuple à nom et prenom, puis affiche le nom et le numéro de téléphone correspondant.

Il existe deux façons de représenter les tuples sur un diagramme d'état. La version plus détaillée montre les indices et les éléments de la même façon qu'ils apparaissent dans une liste. Par exemple, le tuple ('Cleese', 'John') apparaît comme dans la figure 12.1.

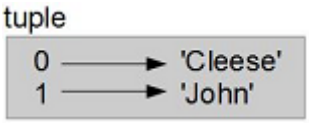


Figure 12.1 : Diagramme d'état.

Mais sur un diagramme plus grand, vous pourriez vouloir laisser de côté les détails. Par exemple, un diagramme d'annuaire téléphonique des membres de la troupe des Monty Python pourrait apparaître comme sur la figure 12.2.

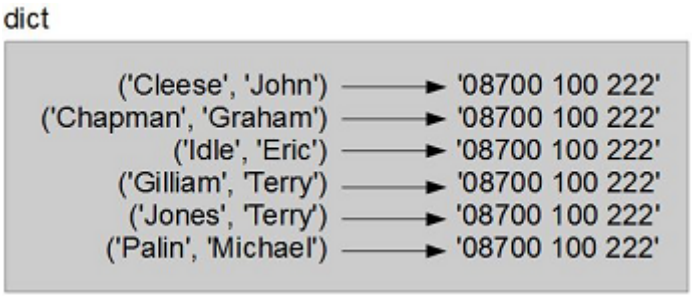


Figure 12.2 : Diagramme d'état.

Ici, les tuples sont représentés en utilisant la syntaxe Python comme un raccourci graphique. Le numéro de téléphone dans le diagramme est le numéro des réclamations de la BBC, inutile de les appeler.

12-7 - Séquences de séquences

Je me suis concentré sur les listes de tuples, mais presque tous les exemples de ce chapitre fonctionnent également avec des listes de listes, des tuples de tuples et des tuples de listes. Pour éviter d'énumérer toutes les combinaisons possibles, il est parfois plus facile de parler des séquences de séquences.

Dans de nombreux contextes, les différents types de séquences (chaînes de caractères, listes et tuples) peuvent être utilisés de manière interchangeable. Alors, comment en choisir une par rapport aux autres ?

Pour commencer par le plus évident, les chaînes de caractères sont plus limitées que les autres séquences parce que leurs éléments doivent être des caractères. Elles sont également immuables. Si vous avez besoin de la capacité de modifier les caractères d'une chaîne (par opposition à la création d'une nouvelle chaîne de caractères), vous pouvez utiliser plutôt une liste de caractères.

Les listes sont plus utilisées que les tuples, principalement parce qu'elles sont modifiables. Mais il existe quelques cas où vous pourriez préférer les tuples :

1. Dans certains contextes, comme une instruction `return`, il est syntaxiquement plus simple de créer un tuple qu'une liste ;
2. Si vous souhaitez utiliser une séquence comme une clé de dictionnaire, vous devez utiliser un type immuable comme un tuple ou une chaîne de caractères ;
3. Si vous passez une séquence comme argument à une fonction, l'utilisation des tuples réduit le risque de comportement inattendu à cause de l'aliasing.

Comme les tuples sont immuables, ils ne fournissent pas de méthodes telles que `sort` et `reverse`, qui modifient les listes existantes. Mais Python fournit la fonction interne `sorted`, qui prend une séquence quelconque et renvoie une nouvelle liste avec les mêmes éléments triés, et `reversed`, qui prend une séquence et retourne un itérateur qui parcourt la liste dans l'ordre inverse.

12-8 - Débogage

Les listes, les dictionnaires et les tuples sont des exemples de **structures de données** ; dans ce chapitre, nous commençons à découvrir les structures composées de données, comme les listes de tuples, ou les dictionnaires qui contiennent des tuples comme clés et des listes comme valeurs. Les structures composées de données sont utiles, mais elles sont sujettes à ce que j'appelle des **erreurs de forme**, c'est-à-dire des erreurs provenant du fait qu'une structure de données n'a pas le bon type, la bonne taille ou la bonne constitution. Par exemple, si vous vous attendez à une liste contenant un nombre entier et je vous donne un simple entier (qui ne se trouve pas dans une liste), cela ne fonctionnera pas.

Pour vous aider à déboguer ces types d'erreurs, j'ai écrit un module appelé structshape qui fournit une fonction, appelée également structshape, qui prend comme argument un type quelconque de structure de données et renvoie une chaîne de caractères qui résume son format. Vous pouvez le télécharger à l'adresse **structshape.py**.

Voici un exemple de résultat pour une liste simple :

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

Un programme plus élégant pourrait écrire « liste de 3 ints », mais il était plus facile de ne pas traiter les pluriels. Voici une liste de listes :

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

Si les éléments de la liste ne sont pas du même type, structshape les groupe, dans l'ordre, par type :

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Voici une liste de tuples :

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

Et voici un dictionnaire avec trois éléments qui font correspondre des entiers à des chaînes de caractères :

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

Si vous éprouvez des difficultés à vous y retrouver dans vos structures de données, structshape peut vous aider.

12-9 - Glossaire

- **tuple :** une séquence immuable d'éléments.
- **affectation de tuple :** une affectation ayant une séquence du côté droit et un tuple de variables du côté gauche. Le côté droit est évalué, puis ses éléments sont affectés aux variables du côté gauche.
- **assembler :** l'opération d'assemblage d'un tuple de longueur variable passé comme argument.
- **disperser :** l'opération de traitement d'une séquence comme une liste d'arguments.
- **objet zip :** le résultat de l'appel de la fonction interne zip ; un objet qui parcourt une séquence de tuples.

- **itérateur :** un objet qui peut parcourir une séquence, mais qui ne fournit pas des opérateurs et des méthodes de liste.
- **structure de données :** une collection de valeurs apparentées, souvent organisées en listes, dictionnaires, tuples, etc.
- **erreur de forme :** une erreur provoquée parce qu'une valeur a la mauvaise forme, c'est-à-dire le mauvais type ou la mauvaise taille.

12-10 - Exercices

Exercice 1

Écrivez une fonction appelée most_frequent *qui prend une chaîne de caractères et affiche les lettres dans l'ordre décroissant de leur fréquence. Trouvez des échantillons de texte dans plusieurs langues différentes et analysez comment varie la fréquence des lettres d'une langue à l'autre. Comparez vos résultats aux tables à l'adresse* **https://fr.wikipedia.org/wiki/Fr%C3%A9quence_d%27apparition_des_lettres_en_fran%C3%A7ais**.

Solution : **most_frequent.py**.

Exercice 2

Encore des anagrammes !

1 *Écrivez un programme qui lit le fichier* mots.txt *et imprime tous les groupes de mots qui forment des anagrammes.*
Voici à quoi pourrait ressembler la sortie :

```
['ARGENT', 'GERANT', 'GRENAT', 'GARENT', 'RAGENT', 'GANTER']
['CRANE', 'ECRAN', 'NACRE', 'CARNE', 'RANCE', 'ANCRE', 'ENCRA', 'CANER', 'CERNA']
['PLATINE', 'PATELIN', 'PLAINTE', 'EPILANT', 'PLIANTE']
```

Indice : vous pourriez vouloir construire un dictionnaire qui établit des correspondances entre une collection de lettres et une liste de mots contenant ces lettres. La question est : comment pouvez-vous représenter la collection de lettres de façon qu'elle puisse être utilisée comme une clé ? Notre liste mots.txt *renferme plus de 2000 groupes de lettres formant des anagrammes (NdT).*

2 *Modifiez le programme précédent de sorte qu'il imprime la plus longue liste d'anagrammes en premier suivie par la deuxième la plus longue, et ainsi de suite. Indice : les listes les plus longues sont quatre groupes de lettres formant chacun neuf anagrammes (NdT).*

3 *Au jeu de Scrabble, vous réalisez un « scrabble » quand vous jouez les sept jetons de votre chevalet, avec généralement une lettre du plateau, pour former un mot de huit lettres. Quelle collection de huit lettres forme le plus de scrabbles possibles ? Indice : il y a deux collections de huit lettres permettant de former chacune huit scrabbles (NdT).*

Solution : **anagram_sets.py**.

Exercice 3

Deux mots forment une « paire par métathèse » si vous pouvez transformer l'un dans l'autre en inversant la position d'un groupe de deux lettres ; par exemple, « CONVERSER » *et* « CONSERVER ». *Écrivez un programme qui trouve toutes les paires par métathèse du document* mots.txt. *Indice : ne testez pas toutes les paires de mots, et ne testez pas toutes les inversions possibles. Indice 2 : ces mots sont aussi des anagrammes, vous pourriez partir des solutions de l'exercice 1. Solution :* **metathesis.py**. *Référence : Cet exercice est inspiré par un exemple sur* **http://puzzlers.org**.

Exercice 4

Voici un autre casse-tête provenant de Car Talk (**http://www.cartalk.com/content/puzzlers**) *:*

- *Quel est le mot français le plus long, qui reste toujours un mot français valide au fur et à mesure que vous retirez ses lettres une par une ?*
  Les lettres peuvent être retirées de chaque extrémité, ou du milieu, mais vous ne pouvez pas réarranger l'ordre des lettres. Chaque fois que vous supprimez une lettre, vous obtenez un autre mot français. Si vous faites cela, finalement il vous reste une seule lettre qui sera également un mot français que l'on retrouve dans le dictionnaire (même si notre liste de mots pour le Scrabble ne contient pas de mots d'une seule lettre). Je veux savoir : quel est le mot le plus long et combien de lettres a-t-il ?
  Prenons un petit exemple modeste : « VIEILLE ». *D'accord ? Vous commencez avec* « VIEILLE », *vous supprimez une lettre de l'intérieur du mot, le premier* I, *et nous nous retrouvons avec le mot* « VEILLE », *puis nous enlevons le premier* E, *nous nous retrouvons avec* « VILLE », *nous supprimons* un L, *il nous reste* « VILE », *puis* « ILE » (ou « VIL »), « IL » *(ou* « LE » *si on a pris* « ILE »*)) et* « I » ou « L ».

Écrivez un programme pour trouver tous les mots qui peuvent être raccourcis de cette manière, et ensuite trouvez-en le plus long.

Cet exercice est un peu plus difficile que les précédents, alors voici quelques suggestions :

1. *Vous pourriez écrire une fonction qui prend un mot et calcule une liste de tous les mots qui peuvent être formés en supprimant une lettre. Ce sont les « enfants » du mot.*
2. *De façon récursive, un mot est réductible si l'un de ses enfants est réductible. Comme un cas de base, vous pouvez considérer la chaîne vide comme étant réductible.*
3. *La liste de mots fournis,* mots.txt *, ne contient aucun mot d'une lettre. Donc, vous voudrez peut-être y ajouter toutes les lettres de l'alphabet (en capitales) et la chaîne vide, que l'on considérera pour les besoins de la recherche comme autant de mots valides.*
4. *Pour améliorer les performances de votre programme, vous voudrez peut-être « mémoïser » les mots qui sont déjà connus comme réductibles.*

Indice : dans notre liste de mots (de 15 lettres au maximum), trois mots de 13 lettres peuvent être réduits ainsi à une seule lettre (NdT).

Solution : **reducible.py**.

13 - Étude de cas : le choix des structures de données

Jusqu'ici, vous avez appris à employer les structures de données standard de Python, et vous avez vu certains des algorithmes qui les utilisent. Si vous souhaitez en savoir plus sur des algorithmes, il pourrait être le bon moment pour lire l'annexe **B**. Mais il n'est pas indispensable de la lire avant de poursuivre ; vous pouvez la lire quand vous voulez.

Ce chapitre présente une étude de cas avec des exercices qui vous permettent de réfléchir au choix des structures de données et de vous entraîner à les utiliser.

13-1 - Analyse de la fréquence des mots

Comme d'habitude, vous devriez au moins essayer de résoudre les exercices avant de lire mes solutions.

Exercice 1

Écrivez un programme qui lit un fichier, divise chaque ligne en mots, enlève les espaces et la ponctuation, remplace les lettres accentuées par leurs équivalentes sans accent et convertit les lettres en majuscules.

Indice : le module `string` *fournit une chaîne de caractères nommée* `whitespace`*, qui contient l'espace, la tabulation, le passage à la ligne, etc., et* `punctuation` *qui contient les caractères de ponctuation. Voyons si nous pouvons faire Python jurer :*

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Vous pouvez aussi envisager d'utiliser les méthodes de chaînes de caractères `strip`*,* `replace` *et* `translate`*. Cette dernière méthode permet de convertir des lettres en d'autres autant de fois que nécessaire. Par exemple, pour remplacer les lettres minuscules accentuées dans une liste hétéroclite de mots :*

```
>>> table_traduction = str.maketrans('àâéèêëïîôùûç', 'aaeeeeiiouuc')
>>> "Leçon; mène; à; été; où; paraître.".translate(table_traduction)
'Lecon; mene; a; ete; ou; paraitre.'
```

L'objet de cet exercice est de trouver un moyen de normaliser les données en entrée pour faciliter leur traitement par la suite. Notre liste de mots de référence est en lettres capitales et sans accents, nous adaptons le texte que nous utilisons en entrée.

Exercice 2

Allez sur la page des livres en français du projet Gutenberg (**http://www.gutenberg.org/browse/languages/fr** *) et téléchargez votre livre favori libre de droits en format texte brut.*

> *Remarque : certains livres en français peuvent poser des problèmes d'encodage de certains caractères accentués ou de ponctuation, peut-être parfois en raison d'une typographie inhabituelle ou fantaisiste. Nous n'avons pas rencontré ce genre de problème avec* ***Germinal*** *(1885), d'Émile Zola, que nous utiliserons plus loin. Vous éviterez peut-être des difficultés en choisissant le même titre (NdT).*

Modifiez votre programme de l'exercice précédent pour lire le livre que vous avez téléchargé, omettre les informations de l'en-tête du début du fichier (et le verbiage juridique en anglais à la fin du livre, le cas échéant), et traiter le reste des mots comme précédemment.

Ensuite, modifiez le programme pour compter le nombre total de mots dans le livre et le nombre de fois que chaque mot est utilisé.

Affichez le nombre de mots différents utilisés dans le livre. Comparez des livres différents par des auteurs différents, écrits à des époques différentes. Quel auteur utilise le vocabulaire le plus vaste ?

Exercice 3

Modifiez le programme de l'exercice précédent pour imprimer les 20 mots les plus fréquemment utilisés dans le livre.

Exercice 4

Modifiez le programme précédent pour lire le fichier `mots.txt` *et ensuite afficher tous les mots du livre qui ne sont pas dans la liste de mots. Combien d'entre eux contiennent des fautes de frappe ? Combien d'entre eux sont des mots communs, qui* devraient *se trouver sur la liste de mots, et combien d'entre eux sont vraiment obscurs ?*

13-2 - Nombres aléatoires

S'ils reçoivent les mêmes données en entrée, la plupart des programmes informatiques génèrent à chaque fois les mêmes résultats, donc ils sont censés être **déterministes**. Le déterminisme est généralement une bonne

chose, puisque nous nous attendons à ce que le même calcul donne le même résultat. Pour certaines applications, cependant, nous voulons que l'ordinateur soit imprévisible. Les jeux sont un exemple évident, mais il y en a d'autres.

Rendre un programme vraiment non déterministe se révèle être difficile, mais il existe des façons de le faire au moins paraître non déterministe. L'une d'elles est d'utiliser des algorithmes qui génèrent des nombres **pseudoaléatoires**. Les nombres pseudoaléatoires ne sont pas vraiment aléatoires, car ils sont générés par un calcul déterministe, mais il est presque impossible de les distinguer des nombres aléatoires uniquement en les regardant.

Le module random fournit des fonctions qui génèrent des nombres pseudoaléatoires (que j'appellerai tout simplement « aléatoires » à partir de maintenant).

La fonction random renvoie un nombre en virgule flottante aléatoire compris entre 0.0 et 1.0 (y compris 0.0, mais pas 1.0). Chaque fois que vous appelez random, vous obtenez le nombre suivant d'une longue série. Pour voir un exemple, exécutez cette boucle :

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

La fonction randint prend en paramètres une limite basse et une limite haute et renvoie un nombre entier compris entre les deux (limites comprises).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Pour choisir de façon aléatoire un élément d'une séquence, vous pouvez utiliser choice :

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Le module random fournit également des fonctions pour générer des valeurs aléatoires de distributions continues comme les distributions de Gauss, exponentielle, gamma, et quelques autres.

Exercice 5

Écrivez une fonction nommée choose_from_hist *, qui prend un histogramme tel que défini dans la section* **11.2** *et renvoie une valeur aléatoire de l'histogramme, choisie avec une probabilité proportionnelle à la fréquence. Par exemple, pour cet histogramme :*

```
>>> t = ['a', 'a', 'b']
>>> hist = histogramme(t)
>>> hist
{'a': 2, 'b': 1}
```

votre fonction devrait retourner 'a' avec une probabilité de 2/3 et 'b' avec une probabilité de 1/3.

13-3 - Histogramme de mots

Vous devriez essayer de résoudre les exercices précédents avant de poursuivre. Vous pouvez télécharger ma solution à l'adresse **analyze_book1.py**. Vous aurez besoin également de **emma.txt** ou plutôt du fichier texte du livre en français que vous aurez choisi.

Nous avons dû modifier légèrement les exercices pour la traduction française. Du coup, le programme proposé ci-dessus ne correspond pas exactement au libellé des exercices. Vous devrez certainement faire quelques adaptations pour que cela fonctionne comme prévu. En particulier, comme la liste mots.txt contient des mots en capitales et sans accents, il faut légèrement adapter le traitement des lignes du livre choisi. Voici notre version adaptée de la fonction process_line :

```
def process_line(line, hist):
    line = line.replace('-', ' ')
    line = line.replace("'", ' ')
    strippables = string.punctuation + string.whitespace

    for word in line.split():
                                                    translation_table       =
 {192: 97, 194: 97, 199: 99, 200: 101, 201: 101, 202: 101, 203: 101, 206: 105, 207: 105, 212: 111, 217: 117, 
        word = word.strip(strippables)
        word = word.translate(translation_table)
        word = word.upper()

        # mise à jour de l'histogramme
        hist[word] = hist.get(word, 0) + 1
```

Quelques adaptations supplémentaires seront peut-être nécessaires, mais vous devriez être largement en mesure de les faire à ce stade de votre lecture de ce livre. (NdT)

Voici un programme qui lit un fichier et construit un histogramme des mots présents dans le fichier :

```
import string

def process_file(nom_fichier):
    hist = dict()
    fp = open(nom_fichier)
    for ligne in fp:
        process_line(ligne, hist)
    return hist

def process_line(ligne, hist):
    ligne = ligne.replace('-', ' ')

    for mot in ligne.split():
        mot = mot.strip(string.punctuation + string.whitespace)
        mot = mot.upper()
        hist[mot] = hist.get(mot, 0) + 1

hist = process_file('germinal.txt')
```

Ce programme lit le fichier germinal.txt, qui contient le texte du roman *Germinal* d'Émile Zola.

process_file parcourt les lignes du fichier dans une boucle, en les passant une par une à process_line. L'histogramme hist est utilisé comme un accumulateur.

process_line utilise la méthode de chaîne de caractères replace pour remplacer les tirets par des espaces avant d'utiliser split pour diviser la ligne en une liste de chaînes de caractères. Elle parcourt la liste des mots et utilise strip et lower pour enlever les signes de ponctuation et convertir les lettres en capitales. (Dire que les chaînes de caractères sont « converties » représente un raccourci linguistique ; souvenez-vous qu'elles sont immuables, donc des méthodes comme strip et lower renvoient de nouvelles chaînes.)

Enfin, process_line met à jour l'histogramme en créant un nouvel élément ou en incrémentant un élément existant.

Pour compter le nombre total de mots dans le fichier, nous pouvons additionner les fréquences reprises dans l'histogramme :

```
def total_mots(hist):
    return sum(hist.values())
```

Le nombre de mots différents est tout simplement le nombre d'éléments dans le dictionnaire :

```
def different_mots(hist):
    return len(hist)
```

Voici un code pour afficher les résultats :

```
print('Nombre total de mots :', total_mots(hist))
print('Nombre de mots différents :', different_mots(hist))
```

Et le résultat :

```
Nombre total de mots : 177195
Nombre de mots différents : 13421
```

13-4 - Les mots les plus fréquents

Pour trouver les mots les plus fréquents, nous pouvons faire une liste de tuples, où chaque tuple contient un mot et sa fréquence, et la trier.

La fonction suivante prend un histogramme et renvoie une liste de tuples mot-fréquence :

```
def les_plus_frequents(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

Dans chaque tuple, la fréquence apparaît en premier, si bien que la liste résultante est triée par fréquence. Voici une boucle qui affiche les dix mots les plus fréquents :

```
t = les_plus_frequents(hist)
print('Les mots les plus fréquents sont :')
for freq, mot in t[:10]:
    print(mot, freq, sep='\t')
```

Je l'utilise l'argument mot-clé `sep` pour indiquer à `print` d'utiliser comme « séparateur » un caractère de tabulation plutôt qu'un espace, afin que la deuxième colonne soit alignée. Voici les résultats obtenus avec le roman *Germinal* :

```
Les mots les plus fréquents sont :

DE      7254
LA      6090
LES     3975
LE      3937
IL      3828
A       3272
ET      3172
L       2635
UN      2509
DES     2467
D       2434
UNE     2166
EN      2008
SE      1721
S       1646
ELLE    1613
```

```
QUE       1549
QU        1497
DU        1411
ETAIT     1363
```

Ce code peut être simplifié en utilisant le paramètre key de la fonction sort. Si vous êtes curieux, vous pouvez lire à ce sujet sur **https://wiki.python.org/moin/HowTo/Sorting**.

13-5 - Paramètres optionnels

Nous avons vu des fonctions et des méthodes internes qui prennent des arguments optionnels. Il est aussi possible d'écrire des fonctions définies par le programmeur qui prennent des arguments optionnels. Par exemple, voici une fonction qui affiche les mots les plus fréquents d'un histogramme :

```
def afficher_les_plus_frequents(hist, num=10):
    t = les_plus_frequents(hist)
    print('Les mots les plus fréquents sont :')
    for freq, mot in t[:num]:
        print(mot, freq, sep='\t')
```

Le premier paramètre est requis ; le second est optionnel. La **valeur par défaut** de num est 10.

Si vous fournissez un seul argument :

```
afficher_les_plus_frequents(hist)
```

num prend la valeur par défaut. Si vous fournissez deux arguments :

```
afficher_les_plus_frequents(hist, 20)
```

num prendra la valeur de l'argument. Autrement dit, l'argument optionnel **remplace** la valeur par défaut.

Si une fonction a en même temps des paramètres obligatoires et des paramètres optionnels, tous les paramètres requis doivent venir en premier, suivis par ceux qui sont optionnels.

13-6 - Soustraction de dictionnaire

Trouver les mots du livre qui ne sont pas dans la liste de mots mots.txt est un problème que vous pourriez identifier comme une soustraction d'ensembles ; autrement dit, nous voulons trouver tous les mots d'un ensemble (les mots dans le livre) qui ne sont pas dans l'autre (les mots dans la liste).

soustraire prend les dictionnaires d1 et d2 et renvoie un nouveau dictionnaire qui contient toutes les clés de d1 qui ne sont pas en d2. Comme nous ne nous soucions pas vraiment des valeurs, nous les initialisons toutes à None.

```
def soustraire(d1, d2):
    resultat = dict()
    for clef in d1:
        if clef not in d2:
            resultat[clef] = None
    return resultat
```

Pour trouver les mots présents dans le livre et pas dans words.txt, nous pouvons utiliser process_file afin de créer un histogramme pour words.text, puis effectuer la soustraction :

```
mots = process_file('mots.txt')
diff = soustraire(hist, mots)
```

```
print('Mots dans le livre qui ne sont pas dans la liste des mots :')
for mot in diff:
    print(mot, end=' ')
```

Voici quelques-uns des résultats pour le roman *Germinal* de Zola :

```
Mots dans le livre qui ne sont pas dans la liste des mots :
HERSCHEUR DEFIAT ENFANTERAIT PERFECTIONNEMENTS VANDERHAGHEN FLANQUERAI BOISAIENT MOUQUE FUTAINE
 LIGNARDS VINSSENT DEROUILLAIENT RAGAILLARDISSAIT
```

Certains de ces mots sont des noms propres (*Vanderhaghen* ou *Mouque*). D'autres (comme *perfectionnements* ou *ragaillardissait*) ne figurent pas dans la liste mots.txt parce qu'elle ne contient pas de mots d'une longueur supérieure à 15 lettres (longueur maximale d'un mot au Scrabble). D'autres ne figurent pas dans la liste parce que celle-ci ne contient apparemment pas les verbes conjugués à l'imparfait du subjonctif (*défiât*, *vinssent*). Certains sont des termes techniques (*herscheur*), régionaux ou d'usage peu courant (*futaine*). Mais quelques-uns sont des mots courants (*enfanterait*, *boisaient*, *flanquerai*) qui mériteraient vraiment de figurer dans la liste !

Exercice 6

Python fournit une structure de données appelée ensemble (set *) qui offre de nombreuses opérations courantes sur des ensembles. Vous pouvez lire à leur sujet dans la section* **19.5** *, ou en lire la documentation à l'adresse* **http://docs.python.org/3/library/stdtypes.html#types-set** *.*

Écrivez un programme qui utilise la soustraction des ensembles pour trouver les mots du livre qui ne sont pas dans la liste de mots. Solution : **analyze_book2.py**.

13-7 - Mots aléatoires

Pour choisir un mot de l'histogramme de façon aléatoire, l'algorithme le plus simple consiste à construire une liste avec plusieurs copies de chaque mot, en fonction de la fréquence observée, puis choisir dans la liste :

```
def random_word(h):
    t = []
    for mot, freq in h.items():
        t.extend([mot] * freq)

    return random.choice(t)
```

L'expression [mot] * freq crée une liste de freq copies de la chaîne de caractères mot. La méthode extend est similaire à append, sauf que l'argument est une séquence.

Cet algorithme fonctionne, mais il n'est pas très efficace ; chaque fois que vous choisissez un mot au hasard, il reconstruit la liste, qui est aussi grande que le livre original. Une amélioration évidente est de construire la liste une seule fois, puis faire des sélections multiples, mais la liste est néanmoins grande.

Une autre possibilité :

1 Utilisez keys pour obtenir une liste des mots dans le livre.
2 Créez une liste qui contient la somme cumulée des fréquences des mots (voir l'exercice **2** du chapitre 10). Le dernier élément de cette liste est le nombre total de mots dans le livre, *n*.
3 Choisissez un nombre aléatoire de 1 à *n*. Utilisez une recherche dichotomique (voir l'exercice **10** du chapitre 10) pour trouver l'indice où le nombre aléatoire serait inséré dans la somme cumulée.
4 Utilisez l'indice pour trouver le mot correspondant dans la liste de mots.

Exercice 7

Écrivez un programme qui utilise cet algorithme pour choisir de façon aléatoire un mot du livre. Solution : **analyze_book3.py**.

13-8 - Analyse de Markov

Si vous choisissez au hasard des mots du livre, vous pouvez vous faire une idée sur le vocabulaire employé, mais vous n'obtiendrez probablement pas une phrase :

```
DEPUIS INVITES INQUIETANTE LUI VOUS LUI PASSE IL PAR MALHEUREUSEMENT
```

Une série de mots au hasard a rarement du sens, car il n'y a aucune relation entre les mots successifs. Par exemple, dans une vraie phrase, vous vous attendez qu'un article comme « une » ou « un » soit suivi par un adjectif ou un nom, et probablement pas par un verbe ou un adverbe.

Une façon de mesurer ces types de relations est l'analyse de Markov, qui caractérise, pour une séquence donnée de mots, la probabilité des mots qui pourraient suivre. Par exemple, le chapitre deux du livre *Le Petit Prince* (1943), d'Antoine de Saint-Exupéry, commence ainsi :

> Le premier soir je me suis donc endormi sur le sable à mille milles de toute terre habitée. J'étais bien plus isolé qu'un naufragé sur un radeau au milieu de l'Océan. Alors vous imaginez ma surprise, au lever du jour, quand une drôle de petite voix m'a réveillé. Elle disait :
> - S'il vous plaît... dessine-moi un mouton !
> - Hein !
> - Dessine-moi un mouton...
> J'ai sauté sur mes pieds comme si j'avais été frappé par la foudre. J'ai bien frotté mes yeux. J'ai bien regardé. Et j'ai vu un petit bonhomme tout à fait extraordinaire qui me considérait gravement. [...]
> Je regardai donc cette apparition avec des yeux tout ronds d'étonnement. N'oubliez pas que je me trouvais à mille milles de toute région habitée. Or mon petit bonhomme ne me semblait ni égaré, ni mort de fatigue, ni mort de faim, ni mort de soif, ni mort de peur. Il n'avait en rien l'apparence d'un enfant perdu au milieu du désert, à mille milles de toute région habitée. Quand je réussis enfin à parler, je lui dis :
> - Mais... qu'est-ce que tu fais là ?
> Et il me répéta alors, tout doucement, comme une chose très sérieuse :
> - S'il vous plaît... dessine-moi un mouton...
> Quand le mystère est trop impressionnant, on n'ose pas désobéir. Aussi absurde que cela me semblât à mille milles de tous les endroits habités et en danger de mort, je sortis de ma poche une feuille de papier et un stylographe. Mais je me rappelai alors que j'avais surtout étudié la géographie, l'histoire, le calcul et la grammaire et je dis au petit bonhomme (avec un peu de mauvaise humeur) que je ne savais pas dessiner. Il me répondit :
> - Ça ne fait rien. Dessine-moi un mouton.
> Comme je n'avais jamais dessiné un mouton je refis, pour lui, l'un des deux seuls dessins dont j'étais capable. Celui du boa fermé. Et je fus stupéfait d'entendre le petit bonhomme me répondre :
> - Non ! Non ! Je ne veux pas d'un éléphant dans un boa. Un boa c'est très dangereux, et un éléphant c'est très encombrant. Chez moi c'est tout petit. J'ai besoin d'un mouton. Dessine-moi un mouton.

Dans ce texte, le mot « dessine » est toujours suivi par « moi », et l'expression « dessine-moi » toujours suivie par l'expression « un mouton ». Le mot « éléphant » est généralement suivi par « dans un boa », ou une fois par l'expression « c'est très encombrant ». Le mot « mille » est toujours suivi du mot « milles ». Et ainsi de suite.

Le résultat de l'analyse de Markov est une correspondance entre chaque préfixe (comme « éléphant ») et tous les suffixes possibles (comme « dans un boa » et « c'est très encombrant »).

Étant donné cette correspondance, vous pouvez générer un texte aléatoire en commençant par un préfixe quelconque et en choisissant au hasard parmi les suffixes possibles. Ensuite, vous pouvez combiner la fin du préfixe et le nouveau suffixe pour former le préfixe suivant, et recommencer.

Par exemple, si vous commencez avec le préfixe « Dessine-moi », alors les mots suivants doivent être « un mouton »

Dans cet exemple, la longueur du préfixe est deux, mais vous pouvez faire une analyse de Markov avec une longueur quelconque du préfixe et du suffixe. Le résultat sera sans doute très différent selon les longueurs choisies.

Exercice 8

Analyse de Markov :

1 *Écrivez un programme qui lit un texte à partir d'un fichier et effectue une analyse de Markov. Le résultat devra être un dictionnaire qui établit des correspondances entre chaque préfixe et une collection de suffixes possibles. La collection peut être une liste, un tuple ou un dictionnaire ; c'est à vous de faire le choix approprié. Vous pouvez tester votre programme avec une longueur du préfixe de deux, mais vous devriez écrire le programme d'une manière qui facilitera l'essai d'autres longueurs.*
2 *Ajoutez au programme précédent une fonction pour générer du texte aléatoire basé sur l'analyse de Markov. Voici un exemple du roman* Germinal *avec une longueur de préfixe de 2 :*
Une seule chose lui causait un malaise, parce qu'il avait vu sa baïonnette tordue comme une tombe.
L'injustice devenait trop grande, il cita les usines qui fermaient, les ouvriers entraient dans la maturité superbe de la caisse de prévoyance l'inquiétait, devenait une menace d'éboulement et d'inondation, la fosse voisine.
Pour cet exemple, j'ai gardé la ponctuation attachée aux mots. Le résultat est presque syntaxiquement correct, mais pas tout à fait. Sémantiquement, il a un certain sens, mais pas tout à fait.
Que se passe-t-il si vous augmentez la longueur du préfixe ? Est-ce que le texte aléatoire a plus de sens ?
3 *Une fois que votre programme fonctionne, vous pourriez vouloir essayer un mélange : si vous combinez du texte à partir de deux ou plusieurs livres, le texte aléatoire que vous générez mélangera le vocabulaire et les phrases des sources de façons intéressantes.*

Référence : cette étude de cas est basée sur un exemple tiré de La pratique de la programmation *par Kernighan et Pike, Addison-Wesley, 1999.*

Vous devriez essayer cet exercice avant de poursuivre ; ensuite, vous pouvez télécharger ma solution sur **markov.py**.

13-9 - Structures de données

L'utilisation de l'analyse de Markov pour générer du texte aléatoire est amusante, mais cet exercice a également une raison : le choix des structures de données. Dans votre solution pour les exercices précédents, vous deviez choisir :

- la façon de représenter les préfixes ;
- la façon de représenter la collection de suffixes possibles ;
- la façon de représenter la correspondance de chaque préfixe à la collection de suffixes possibles.

Le dernier choix est facile : un dictionnaire est le choix évident pour faire correspondre des clés à des valeurs.

Pour les préfixes, les options les plus évidentes sont une chaîne de caractères, une liste de chaînes de caractères ou un tuple de chaînes de caractères.

Pour les suffixes, une option est une liste ; l'autre est un histogramme (dictionnaire).

Comment choisir ? La première étape est de réfléchir aux opérations que vous devrez mettre en œuvre pour chaque structure de données. Pour les préfixes, nous devons être en mesure de supprimer des mots du début et les ajouter

à la fin. Par exemple, si le préfixe actuel est « dessine-moi un », et le mot suivant est « mouton », vous devez être en mesure de former le préfixe suivant, « un mouton ».

Votre premier choix serait peut-être une liste, car il est facile d'y ajouter et supprimer des éléments, mais nous devons également être en mesure d'utiliser les préfixes comme clés d'un dictionnaire, donc cela exclut les listes. Avec des tuples, vous ne pouvez pas ajouter ou supprimer, mais vous pouvez utiliser l'opérateur d'addition pour former un nouveau tuple :

```
def shift(prefixe, mot):
    return prefixe[1:] + (mot,)
```

shift prend un tuple de mots, prefixe, et une chaîne de caractères, mot, et forme un nouveau tuple, qui contient tous les mots de prefixe sauf le premier et mot ajouté à la fin.

Pour la collection de suffixes, les opérations que nous devons effectuer comprennent l'ajout d'un nouveau suffixe (ou l'incrémentation de la fréquence d'un suffixe existant) et le choix d'un suffixe aléatoire.

L'ajout d'un nouveau suffixe est tout aussi facile par l'implémentation d'une liste ou d'un histogramme. Le choix d'un élément aléatoire d'une liste est facile ; le choix d'un élément d'un histogramme est plus difficile à faire efficacement (voir l'exercice **7** plus haut).

Jusqu'à présent, nous avons parlé surtout de la facilité d'implémentation, mais il y a d'autres facteurs à considérer dans le choix des structures de données. L'un est la durée d'exécution. Parfois, il existe une raison théorique pour s'attendre qu'une structure de données soit plus rapide que les autres ; par exemple, j'ai mentionné que l'opérateur in est plus rapide pour les dictionnaires que pour les listes, du moins lorsque le nombre d'éléments est grand.

Mais souvent, vous ne savez pas à l'avance quelle mise en œuvre sera la plus rapide. Une option consiste à mettre en œuvre les deux et voir quelle est la meilleure. Cette approche est appelée **test de performance**, banc d'essai ou *benchmark*. Une autre option pratique est de choisir la structure de données la plus facile à mettre en œuvre, et vérifier ensuite si elle est assez rapide pour l'application visée. Si oui, il n'y a pas besoin d'aller chercher plus loin. Sinon, il existe des outils comme le module profile, qui peut identifier les endroits les plus chronophages dans un programme.

L'autre facteur à prendre en considération est l'espace mémoire. Par exemple, l'utilisation d'un histogramme pour la collection des suffixes pourrait nécessiter moins d'espace parce que vous devez stocker chaque mot une seule fois, quel que soit le nombre de fois où il apparaît dans le texte. Dans certains cas, économiser l'espace peut aussi rendre plus rapide l'exécution de votre programme, et à l'autre extrême, celui-ci pourrait ne pas s'exécuter du tout s'il manque de mémoire. Mais pour de nombreuses applications, l'espace mémoire est une considération secondaire après la durée d'exécution.

Une dernière réflexion : dans cette discussion, j'ai sous-entendu qu'il fallait utiliser une seule structure de données tant pour l'analyse que pour la génération. Mais comme ce sont des phases distinctes, il serait également possible d'utiliser une structure pour l'analyse et ensuite la convertir en une autre structure pour la génération. Cela en vaut la peine si le gain de temps lors de la génération est supérieur au temps passé à effectuer la conversion.

13-10 - Débogage

Lorsque vous déboguez un programme, et surtout si vous travaillez sur un bogue coriace, voici cinq choses à essayer :

- lire : examinez votre code, relisez-le et vérifiez qu'il dit ce que vous vouliez dire ;
- exécuter : expérimentez en apportant des changements et en exécutant différentes versions. Souvent, si vous affichez la bonne chose au bon endroit dans le programme, le problème devient évident, mais parfois vous avez à construire des échafaudages ;
- ruminer : prenez le temps de réfléchir ! De quel genre d'erreur s'agit-il : de syntaxe, d'exécution, ou sémantique ? Quelles informations les messages d'erreur ou la sortie du programme vous offrent-ils ?

Quel genre d'erreur peut provoquer le problème que vous rencontrez ? Qu'est-ce que vous avez modifié récemment, avant que le problème soit apparu ?
- méthode du canard en plastique : si vous expliquez le problème à quelqu'un d'autre, vous trouverez parfois la réponse avant d'avoir fini de poser la question. Souvent, vous n'avez pas besoin d'une autre personne ; vous pourriez parler à un canard de baignoire en plastique. Et cela est à l'origine de la stratégie bien connue appelée **méthode du canard en plastique**. Je ne l'invente pas ; voir **https://fr.wikipedia.org/wiki/M%C3%A9thode_du_canard_en_plastique** ;
- revenir en arrière : à un certain moment, la meilleure chose à faire peut être de reculer, de défaire les changements récents, jusqu'au moment où vous vous retrouvez avec un programme qui fonctionne et que vous comprenez. Ensuite, vous pouvez commencer à reconstruire.

Les programmeurs débutants restent parfois bloqués sur l'une de ces activités et oublient les autres. Chaque activité a son propre mode de défaillance.

Par exemple, la lecture de votre code peut aider si le problème est une faute de frappe ou un mot pour un autre, mais pas si le problème est un malentendu conceptuel. Si vous ne comprenez pas ce que fait votre programme, vous pouvez le lire 100 fois et ne jamais voir l'erreur, parce que l'erreur est dans votre tête.

Tester des bouts de code peut aider, surtout si ce sont des petits bouts de code simples. Mais si vous expérimentez des changements sans penser à relire votre code, vous pouvez tomber dans un syndrome que j'appellerais « la programmation errante aléatoire », qui consiste à faire des changements aléatoires jusqu'à ce que le programme fasse ce que vous voulez. Inutile de dire que ce genre de méthode peut prendre beaucoup de temps.

Vous devez prendre le temps de réfléchir. Le débogage est comme une science expérimentale. Vous devez avoir au moins une hypothèse sur la nature du problème. S'il existe deux ou plusieurs possibilités, essayez d'imaginer un test qui permettrait d'éliminer l'une d'elles.

Mais même les meilleures techniques de débogage échoueront s'il y a trop d'erreurs, ou si le code que vous essayez de corriger est trop grand et complexe. Parfois, la meilleure option est de battre en retraite, de simplifier le programme jusqu'au moment où vous arrivez à quelque chose qui fonctionne et que vous comprenez.

Les programmeurs débutants sont souvent réticents à reculer parce qu'ils ne supportent pas l'idée de supprimer une ligne de code (même si elle est erronée). Si cela vous aide à vous sentir mieux, faites une copie votre programme dans un autre fichier avant de commencer l'élagage. Ensuite, vous pourrez rajouter les morceaux un par un.

Trouver un bogue coriace nécessite relecture, exécution, rumination, et parfois du recul. Si vous vous retrouvez coincé sur l'une de ces activités, essayez les autres.

13-11 - Glossaire

- **déterministe :** relatif à un programme qui fait la même chose à chaque exécution, avec les mêmes données en entrée.
- **pseudoaléatoire :** relatif à une suite de nombres qui semble être aléatoire, mais est générée par un programme déterministe.
- **valeur par défaut :** la valeur donnée à un paramètre optionnel si aucun argument n'est fourni.
- **remplacer :** le remplacement d'une valeur par défaut par un argument.
- **test de performances :** le processus permettant de choisir une structure de données en mettant en œuvre différentes solutions possibles et en les testant sur un échantillon de données en entrée.
- **méthode du canard en plastique :** débogage en expliquant votre problème à un objet inanimé tel qu'un canard en plastique. Formuler le problème peut vous aider à le résoudre, même si le canard ne connaît pas Python.

13-12 - Exercices

Exercice 9

Le « rang » d'un mot est sa position dans une liste de mots classés par fréquence : le mot le plus commun est de rang 1, le deuxième le plus courant est de rang 2, etc.

*La loi de Zipf décrit une relation entre les rangs et les fréquences des mots dans les langues naturelles (***https://fr.wikipedia.org/wiki/Loi_de_Zipf***). Plus précisément, elle prévoit que la fréquence,* f*, du mot de rang* r *est :*

$$f = c\, r^{-s}$$

où s *et* c *sont des paramètres qui dépendent de la langue et du texte. Si vous prenez le logarithme des deux côtés de cette équation, vous obtenez :*

$$\log f = \log c - s \log r$$

Donc, si vous tracez log f *en fonction de* log r*, vous devriez obtenir une ligne droite de pente* -s *et intersectant l'axe des ordonnées en* log c*.*

Écrivez un programme qui lit un texte à partir d'un fichier, mesure la fréquence des mots, et affiche une ligne pour chaque mot, en ordre décroissant de fréquence, avec log f et log r. Utilisez le programme graphique de votre choix pour tracer les résultats et vérifier s'ils forment une ligne droite. Pouvez-vous estimer la valeur de s ?

Solution : **zipf.py**. *Pour exécuter ma solution, vous avez besoin du module de traçage* matplotlib*. Si vous avez installé Anaconda, vous avez déjà* matplotlib *; sinon il se peut que vous deviez l'installer.*

14 - Fichiers

Ce chapitre introduit l'idée de programmes « persistants » qui gardent les données dans la mémoire permanente, et montre comment utiliser de différents types de stockage permanent, comme les fichiers et les bases de données.

14-1 - Persistance

La majorité des programmes que nous avons vu jusqu'à présent sont transitoires dans le sens où ils s'exécutent pour un court laps de temps et affichent quelque chose, mais quand ils finissent, leurs données disparaissent. Si vous exécutez le programme à nouveau, il reprend à zéro.

D'autres programmes sont **persistants** : ils s'exécutent longtemps (ou tout le temps) ; ils gardent au moins une partie de leurs données en stockage permanent (un disque dur, par exemple) et s'ils s'arrêtent et redémarrent, ils reprennent d'où ils s'étaient arrêtés.

Des exemples de programmes persistants sont les systèmes d'exploitation, qui s'exécutent à peu près chaque fois qu'un ordinateur est mis en route, et les serveurs Web, qui s'exécutent en permanence, en attendant que des demandes arrivent sur le réseau.

Une des façons les plus simples de conserver les données des programmes est par la lecture et l'écriture de fichiers texte. Nous avons déjà vu des programmes qui lisent des fichiers texte ; dans ce chapitre, nous allons voir des programmes qui les écrivent.

Un autre cas classique consiste à stocker l'état du programme dans une base de données. Dans ce chapitre, je vais présenter une base de données simple et un module, pickle, qui facilite le stockage des données du programme.

14-2 - Lecture et écriture

Un fichier texte est une séquence de caractères stockée sur un support permanent comme un disque dur, une mémoire flash ou un CD-ROM. Nous avons vu comment ouvrir et lire un fichier dans la section **9.1**.

Pour écrire un fichier, vous devez l'ouvrir avec le mode 'w' comme second paramètre :

```
>>> fout = open('sortie.txt', 'w')
```

Si le fichier existe déjà, l'ouverture en mode d'écriture efface les anciennes données et reprend à vide, donc soyez prudent ! Si le fichier n'existe pas, un nouveau fichier est créé.

open renvoie un objet fichier qui fournit des méthodes pour travailler avec le fichier. La méthode write écrit des données dans le fichier.

```
>>> line1 = "Voici l'acacia,\n"
>>> fout.write(line1)
16
```

La valeur de retour est le nombre de caractères qui ont été écrits. L'objet fichier garde la trace de l'endroit où il est arrivé, donc si vous appelez la méthode write à nouveau, elle ajoute les nouvelles données à la suite de ce qui a déjà été écrit, donc à la fin du fichier.

```
>>> line2 = "l'emblème de notre pays.\n"
>>> fout.write(line2)
25
```

Quand vous avez fini d'écrire, vous devez fermer le fichier.

```
>>> fout.close()
```

Si vous ne fermez pas le fichier, il sera fermé à la fin du programme.

14-3 - L'opérateur de formatage

L'argument de write doit être une chaîne de caractères, donc si nous voulons mettre d'autres valeurs que des caractères dans un fichier, nous devons les convertir en chaînes de caractères. La façon la plus simple de le faire est en utilisant str :

```
>>> x = 52
>>> fout.write(str(x))
```

Une autre possibilité consiste à utiliser l'**opérateur de formatage**, %. Lorsqu'il est appliqué à des nombres entiers, % est l'opérateur modulo (reste de la division entière). Mais lorsque le premier opérande est une chaîne, % est l'opérateur de formatage.

Le premier opérande est la **chaîne de formatage**, qui contient une ou plusieurs **séquences de formatage**, qui précisent comment est formaté le deuxième opérande. Le résultat est une chaîne.

Par exemple, la séquence de formatage '%d' signifie que le second opérande doit être formaté en tant que nombre entier en base 10 :

```
>>> chameaux = 42
>>> '%d' % chameaux
'42'
```

Le résultat est la chaîne '42', qui ne doit pas être confondue avec la valeur entière 42.

Une séquence de formatage peut apparaître n'importe où dans la chaîne, donc vous pouvez intercaler une valeur dans une phrase :

```
>>> "J'ai repéré %d chameaux." % chameaux
"J'ai repéré 42 chameaux."
```

S'il y a plus d'une séquence de formatage dans la chaîne de caractères, le second argument doit être un tuple. Chaque séquence de formatage correspond à un élément du tuple, dans l'ordre.

L'exemple suivant utilise '%d' pour formater un nombre entier, '%g' pour formater un nombre à virgule flottante et '%s' pour formater une chaîne de caractères :

```
>>> "En %d ans j'ai repéré %g %s." % (3, 0.1, 'chameaux')
"En 3 ans j'ai repéré 0.1 chameaux."
```

Le nombre d'éléments dans le tuple doit correspondre au nombre de séquences de formatage dans la chaîne de caractères. En outre, les types des éléments doivent correspondre aux séquences de formatage :

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

Dans le premier exemple, il n'y a pas suffisamment d'éléments ; dans le second, l'élément n'a pas le bon type.

Pour plus d'informations sur l'opérateur de formatage, consultez **https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting**. Un autre choix, plus puissant, est la méthode format de formatage de chaîne de caractères, au sujet de laquelle vous pouvez lire sur **https://docs.python.org/3/library/stdtypes.html#str.format**.

14-4 - Noms de fichiers et chemins

Les fichiers sont organisés en **répertoires** (également appelés « dossiers »). Chaque programme en cours d'exécution a un « répertoire courant », qui est le répertoire par défaut pour la majorité des opérations. Par exemple, lorsque vous ouvrez un fichier en lecture, Python le recherche dans le répertoire courant.

Le module os fournit des fonctions pour travailler avec des fichiers et des répertoires (« os » signifie « système d'exploitation »). os.getcwd renvoie le nom du répertoire courant :

```
>>> import os
>>> chemin = os.getcwd()
>>> chemin
'/home/dinsdale'
```

cwd signifie « répertoire de travail courant ». Dans cet exemple, le résultat est /home/dinsdale, qui est le répertoire personnel d'un d'utilisateur nommé dinsdale.

Une chaîne de caractères comme '/home/dinsdale' qui identifie un fichier ou un répertoire s'appelle un **chemin**.

Un simple nom de fichier, comme memo.txt est également considéré comme un chemin, mais il s'agit d'un **chemin relatif**, car il est relatif au répertoire courant. Si le répertoire courant est /home/dinsdale, le nom de fichier memo.txt se référerait à /home/dinsdale/memo.txt.

Un chemin qui commence par / ne dépend pas du répertoire courant ; c'est un **chemin absolu**. Pour trouver le chemin absolu d'un fichier, vous pouvez utiliser os.path.abspath :

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path offre d'autres fonctions pour travailler avec les noms et les chemins de fichiers. Par exemple, os.path.exists vérifie si un fichier ou un répertoire existe :

```
>>> os.path.exists('memo.txt')
True
```

S'il existe, os.path.isdir vérifie s'il est un répertoire :

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

De façon similaire, os.path.isfile vérifie si c'est un fichier.

os.listdir renvoie une liste des fichiers (et de sous-répertoires) du répertoire donné :

```
>>> os.listdir(chemin)
['musique', 'photos', 'memo.txt']
```

Pour illustrer ces fonctions, l'exemple suivant « parcourt » un répertoire, affiche les noms de tous les fichiers, et s'appelle lui-même de manière récursive sur tous les sous-répertoires.

```
def parcourir(nom_repertoire):
    for nom in os.listdir(nom_repertoire):
        chemin = os.path.join(nom_repertoire, nom)

        if os.path.isfile(chemin):
            print(chemin)
        else:
            parcourir(chemin)
```

os.path.join prend un répertoire et un nom de fichier et les concatène pour former un chemin d'accès complet.

Le module os fournit une fonction appelée walk qui est similaire à celle-ci, mais plus polyvalente. À titre d'exercice, lisez la documentation de cette fonction et utilisez-la pour afficher les noms des fichiers d'un répertoire donné et de ses sous-répertoires. Vous pouvez télécharger ma solution à l'adresse **walk.py**.

14-5 - Intercepter les exceptions

Beaucoup de choses peuvent mal se passer lorsque vous essayez de lire et écrire des fichiers. Si vous essayez d'ouvrir un fichier qui n'existe pas, vous obtenez une IOError :

```
>>> fin = open('mauvais_fichier')
IOError: [Errno 2] No such file or directory: 'mauvais_fichier'
```

Si vous n'êtes pas autorisé à accéder à un fichier :

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

Et si vous essayez d'ouvrir un répertoire pour la lecture, vous obtenez

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

Pour éviter ces erreurs, vous pouvez utiliser des fonctions comme os.path.exists et os.path.isfile, mais la vérification de toutes les possibilités nécessiterait beaucoup de temps et de code (si le code d'erreur « Errno 21 » constitue un indice fiable, il existe au moins 21 choses qui peuvent mal se passer).

Il est préférable d'aller de l'avant et d'essayer - et de faire face aux problèmes s'ils se produisent - ce qui est exactement ce que fait l'instruction try (« essaie »). La syntaxe ressemble à une déclaration if...else :

```
try:
    fin = open('mauvais_fichier')
except:
    print("Quelque chose s'est mal passé.")
```

Python commence par exécuter la clause try. Si tout se passe bien, il ignore la clause except et poursuit. Si une exception se produit, il sort de la clause try et exécute la clause except.

La gestion d'une exception par une instruction try s'appelle **intercepter** une exception. Dans cet exemple, la clause except affiche un message d'erreur qui n'est pas très utile. En général, intercepter une exception vous donne une chance de résoudre le problème, ou de réessayer, ou au moins de terminer le programme avec élégance.

14-6 - Bases de données

Une **base de données** est un fichier (ou parfois un groupe de fichiers) qui est organisé pour stocker des données. Beaucoup de bases de données sont organisées comme un dictionnaire en ce sens qu'elles établissent une correspondance entre des clés et des valeurs. La principale différence entre une base de données et un dictionnaire est que la base de données est sur le disque (ou un autre support de stockage permanent), si bien qu'elle continue à exister après la fin de l'exécution du programme.

Le module dbm fournit une interface pour créer et mettre à jour des fichiers de base de données. À titre d'exemple, je vais créer une base de données qui contient des légendes pour fichiers image.

L'ouverture d'une base de données ressemble à l'ouverture d'autres fichiers :

```
>>> import dbm
>>> db = dbm.open('legendes', 'c')
```

Le mode 'c' signifie que la base de données doit être créée à vide si elle n'existe pas déjà. Le résultat est un objet base de données qui peut être utilisé (pour la majorité des opérations) comme un dictionnaire.

Lorsque vous créez un nouvel élément, dbm met à jour le fichier de base de données.

```
>>> db['cleese.png'] = 'Photo de John Cleese.'
```

Lorsque vous accédez à l'un des éléments, dbm lit le fichier :

```
>>> db['cleese.png']
b'Photo de John Cleese.'
```

Le résultat est un **objet octets** *(byte object)*, c'est pour cela qu'il commence par b. Un objet octets ressemble à une chaîne de caractères à bien des égards. Lorsque vous avancerez dans l'étude de Python, vous verrez qu'il existe des différences importantes, mais pour l'instant nous pouvons l'ignorer.

Si vous affectez une autre valeur à une clé existante, dbm remplace l'ancienne valeur :

```
>>> db['cleese.png'] = 'Photo de John Cleese marchant drôlement.'
>>> db['cleese.png']
b'Photo de John Cleese marchant drôlement.'
```

Certaines méthodes de dictionnaire, comme keys et items, ne fonctionnent pas avec des objets base de données. Mais l'itération dans une boucle for fonctionne :

```
for clef in db:
    print(clef, db[clef])
```

Comme dans le cas d'autres fichiers, vous devez fermer la base de données lorsque vous avez terminé :

```
>>> db.close()
```

14-7 - Sérialiser les données avec pickle

Une limitation de dbm est que les clés et les valeurs doivent être des chaînes de caractères ou des octets. Si vous essayez d'utiliser un autre type, vous obtenez une erreur.

Le module pickle peut vous aider. Il traduit presque tout type d'objet en une chaîne de caractères appropriée pour le stockage dans une base de données, puis peut retraduire les chaînes de caractères en objets.

pickle.dumps prend en entrée un objet et renvoie sa représentation en chaîne de caractères (dumps est un raccourci de *dump string* - « copier sous forme de chaîne ») :

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

Le format n'est pas compréhensible pour les lecteurs humains ; il est conçu pour être facile à interpréter pour pickle. pickle.loads (*load string* - « charger la chaîne ») reconstitue l'objet :

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Même si le nouvel objet a la même valeur que l'ancien, ce n'est pas (en général) le même objet :

```
>>> t1 == t2
True
>>> t1 is t2
False
```

Autrement dit, utiliser pickle pour sérialiser puis désérialiser un objet a le même effet que copier l'objet.

Vous pouvez utiliser pickle pour stocker dans une base de données des données qui ne sont pas des chaînes de caractères. En fait, cette combinaison est si commune qu'elle a été encapsulée dans un module appelé shelve.

14-8 - Pipes

Les systèmes d'exploitation fournissent pour la plupart une interface de ligne de commande, souvent appelée **shell**. Un shell fournit habituellement des commandes pour naviguer dans le système de fichiers et lancer des applications. Par exemple, sous Unix ou Linux, vous pouvez changer de répertoire avec cd, afficher le contenu d'un répertoire avec ls, et lancer un navigateur Web en tapant (par exemple) firefox.

Tout programme que vous pouvez lancer à partir du shell peut être lancé également à partir de Python en utilisant un **objet pipe**, qui représente un programme en cours d'exécution.

Par exemple, la commande Unix ls -l affiche généralement le contenu du répertoire courant en format long (détaillé). Vous pouvez lancer ls avec la commande os.popen (1) :

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

L'argument est une chaîne de caractères qui contient une commande shell. La valeur de retour est un objet qui se comporte comme un fichier ouvert. Vous pouvez lire la sortie du processus ls ligne par ligne avec readline ou en obtenir tout le contenu avec read :

```
>>> res = fp.read()
```

Lorsque vous aurez terminé, vous fermez le pipe comme un fichier :

```
>>> stat = fp.close()
>>> print(stat)
None
```

La valeur de retour est le statut final du processus ls ; None signifie qu'il s'est terminé normalement (sans erreur).

Par exemple, la plupart des systèmes Unix fournissent une commande appelée md5sum qui lit le contenu d'un fichier et calcule une « somme de contrôle » ou *checksum*. Vous trouverez des informations complémentaires au sujet de MD5 sur **https://fr.wikipedia.org/wiki/MD5**. Cette commande fournit un moyen efficace pour vérifier si deux fichiers ont le même contenu. La probabilité que des contenus différents donnent la même somme de contrôle est très faible (autrement dit, il y a peu de chances que cela se produise avant que l'univers s'effondre).

Vous pouvez utiliser un pipe pour exécuter md5sum à partir d'un programme Python et en obtenir le résultat :

```
>>> nomfichier = 'livre.tex'
>>> cmd = 'md5sum ' + nomfichier
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  livre.tex
>>> print(stat)
None
```

14-9 - Écrire des modules

Tout fichier qui contient du code Python peut être importé en tant que module. Par exemple, supposons que vous ayez un fichier nommé wc.py (le nom, *wc*, pour *word coun*t, est celui d'un petit utilitaire Unix qui compte notamment le nombre de mots et de lignes d'un fichier texte) avec le code suivant :

```
def compteur_lignes(nomfichier):
    compteur = 0
    for ligne in open(nomfichier):
        compteur += 1
    return compteur

print(compteur_lignes('wc.py'))
```

Si vous exécutez ce programme, il lit son propre contenu et imprime le nombre de lignes du fichier, qui est 7. Vous pouvez également l'importer comme ceci :

```
>>> import wc
```

```
7
```

Maintenant, vous avez un objet module wc :

```
>>> wc
<module 'wc' from 'wc.py'>
```

L'objet module fournit compteur_lignes :

```
>>> wc.compteur_lignes('wc.py')
7
```

Voilà donc comment vous écrivez des modules en Python.

Le seul problème avec cet exemple est que, lorsque vous importez le module, il exécute le code de test de la dernière ligne. Normalement, lorsque vous importez un module, il définit de nouvelles fonctions, mais il ne les exécute pas.

Les programmes qui seront importés en tant que modules utilisent souvent la tournure suivante :

```
if __name__ == '__main__':
    print(compteur_lignes('wc.py'))
```

__name__ est une variable interne définie lorsque le programme démarre. Si le programme s'exécute en tant que script, __name__ a la valeur '__main__' ; dans ce cas, le code de test est exécuté. Sinon, si le module est importé, le code de test est ignoré.

À titre d'exercice, tapez cet exemple dans un fichier nommé wc.py et exécutez-le comme un script. Ensuite, exécutez l'interpréteur Python et import wc. Quelle est la valeur de __name__ lorsque le module est importé ?

Attention : Si vous tentez d'importer un module qui a déjà été importé, Python ne fait rien. Il ne relit pas le fichier, même s'il a été modifié.

Si vous voulez recharger un module, vous pouvez utiliser la fonction interne reload, mais il peut y avoir de subtiles difficultés ; par conséquent, la chose la plus sûre à faire est de redémarrer l'interpréteur et d'importer à nouveau le module.

14-10 - Débogage

Lorsque vous lisez et écrivez des fichiers, il peut arriver que vous rencontriez des problèmes dus aux caractères non imprimables. Ces erreurs peuvent être difficiles à déboguer parce que les espaces, les tabulations et les sauts de ligne sont normalement invisibles :

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

La fonction interne repr peut vous aider. Elle prend comme argument un objet quelconque et renvoie la représentation en chaîne de caractères de celui-ci. Pour les chaînes de caractères, elle représente les caractères non imprimables précédés par des barres obliques inverses :

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

Cela peut être utile pour le débogage.

Un autre problème que vous pourriez rencontrer est que les différents systèmes d'exploitation utilisent différents caractères pour indiquer la fin d'une ligne. Certains systèmes, comme Unix ou Linux, utilisent un saut de ligne, représenté comme \n. D'autres utilisent un caractère de retour chariot, représenté comme \r. Certains, comme Windows, utilisent les deux. Si vous déplacez des fichiers entre les différents systèmes, ces incohérences peuvent causer des problèmes.

Pour la majorité des systèmes, il existe des applications pour conversion d'un format à un autre. Vous pouvez les trouver (et en savoir plus à ce sujet) à l'adresse **https://fr.wikipedia.org/wiki/Fin_de_ligne**. Ou, bien sûr, vous pouvez en écrire une vous-même.

14-11 - Glossaire

- **persistant :** se rapportant à un programme qui fonctionne indéfiniment ou maintient au moins une partie de ses données dans un support de stockage permanent.
- **opérateur de formatage :** un opérateur, %, qui prend une chaîne de formatage et un tuple et génère une chaîne qui contient les éléments du tuple formatés comme spécifié par la chaîne de formatage.
- **chaîne de formatage :** une chaîne, utilisée avec l'opérateur de formatage, qui contient des séquences de formatage.
- **séquence de formatage :** une séquence de caractères dans une chaîne de formatage, comme %d, qui spécifie comment une valeur doit être formatée.
- **fichier texte :** une séquence de caractères stockée dans un support permanent tel qu'un disque dur.
- **répertoire :** une collection de fichiers nommée, parfois aussi appelée un dossier.
- **chemin :** une chaîne de caractères qui identifie un fichier.
- **chemin relatif :** un chemin qui commence à partir du répertoire courant.
- **chemin absolu :** un chemin qui commence à partir du répertoire le plus élevé dans le système de fichiers.
- **intercepter :** éviter qu'une exception mette fin à un programme, en utilisant les instructions try et except.
- **base de données :** un fichier dont le contenu est organisé comme un dictionnaire avec des clés qui correspondent à des valeurs.
- **objet octets :** un objet similaire à une chaîne.
- **shell :** un programme qui permet aux utilisateurs de taper des commandes et les exécuter pour lancer d'autres programmes.
- **objet pipe :** un objet qui représente un programme en cours d'exécution, permettant à un programme Python d'exécuter des commandes et d'en lire les résultats.

14-12 - Exercices

Exercice 1

Écrivez une fonction appelée sed *qui prend pour arguments un motif en tant que chaîne de caractères, une chaîne de caractères de remplacement et deux noms de fichiers ; elle devra lire le premier fichier et écrire son contenu dans le second fichier (en le créant si nécessaire). Si la chaîne de motif apparaît à un endroit quelconque dans le fichier, elle devra être remplacée par la chaîne de remplacement.*

Si une erreur se produit lors de l'ouverture, la lecture, l'écriture ou de fermeture des fichiers, votre programme doit intercepter l'exception, afficher un message d'erreur et se terminer. Solution : **sed.py**.

Exercice 2

Si vous téléchargez ma solution à l'exercice **2** *du chapitre 12 à l'adresse* **anagram_sets.py** *, vous verrez qu'elle crée un dictionnaire qui établit une correspondance entre une chaîne de caractères triée et une liste des mots qui peuvent être orthographiés avec ces lettres. Par exemple,* 'AEILNPT' *mappe vers la liste* ['PLATINE', 'PATELIN', 'PLAINTE', 'EPILANT', 'PLIANTE'] *.*

Écrivez un module qui importe anagram_sets *et fournit deux nouvelles fonctions :* store_anagrams *devra stocker le dictionnaire d'anagrammes dans une « étagère »* shelve *; read_anagrams devra rechercher un mot et renvoyer une liste de ses anagrammes. Solution :* **anagram_db.py**.

Exercice 3

Dans une grande collection de fichiers MP3, il peut y avoir plusieurs copies de la même chanson, stockées dans des répertoires différents ou avec des noms de fichiers différents. Le but de cet exercice est de rechercher les doublons.

1 *Écrivez un programme qui recherche dans un répertoire et tous ses sous-répertoires, de manière récursive, et renvoie une liste de chemins complets pour tous les fichiers avec un suffixe donné (comme* .mp3 *). Indice :* os.path *offre plusieurs fonctions utiles pour la manipulation des noms de fichier et de chemin.*
2 *Pour reconnaître les doublons, vous pouvez utiliser* md5sum *pour calculer une somme de contrôle pour chaque fichier. Si deux fichiers ont la même somme de contrôle, elles ont probablement le même contenu.*
3 *Pour une vérification supplémentaire, vous pouvez utiliser la commande Unix* diff *.*

Solution : **find_duplicates.py**.

15 - Classes et objets

À présent, vous savez comment utiliser les fonctions pour organiser le code et les types internes pour organiser les données. La prochaine étape est d'apprendre « la programmation orientée objet », qui utilise des types définis par le programmeur pour organiser tant le code que les données. La programmation orientée objet est un sujet vaste, il faudra quelques chapitres pour en faire un tour d'horizon.

Les exemples de code de ce chapitre sont disponibles à l'adresse **Point1.py** ; des solutions aux exercices sont disponibles sur **Point1_soln.py**.

15-1 - Types définis par le programmeur

Nous avons utilisé de nombreux types internes de Python ; maintenant, nous allons définir un nouveau type. À titre d'exemple, nous allons créer un type appelé Point qui représente un point dans l'espace bidimensionnel.

En notation mathématique, les points sont souvent écrits entre parenthèses avec une virgule séparant les coordonnées cartésiennes. Par exemple, (0,0) représente l'origine, et *(x, y)* représente le point à *x* unités vers la droite et à *y* unités vers le haut par rapport à l'origine.

Il existe plusieurs façons de représenter des points en Python :

- nous pourrions stocker les coordonnées séparément dans deux variables x et y ;
- nous pourrions stocker les coordonnées comme éléments d'une liste ou d'un tuple ;
- nous pourrions créer un nouveau type pour représenter les points comme des objets.

La création d'un nouveau type est plus compliquée que les autres options, mais elle présente des avantages qui seront bientôt évidents.

Un type défini par le programmeur s'appelle également une **classe**. Une définition de classe ressemble à ceci :

```
class Point:
    """Représente un point dans l'espace 2-D."""
```

L'en-tête indique que la nouvelle classe s'appelle Point. Le corps est une *docstring* qui explique à quoi sert la classe. Vous pouvez définir des variables et des méthodes dans une définition de classe, mais nous y reviendrons plus tard.

La définition d'une classe nommée Point crée un **objet classe**.

```
>>> Point
<class '__main__.Point'>
```

Comme la classe Point est définie au plus haut niveau, son « nom complet » est :

```
__main__.Point.
```

L'objet classe est une espèce d'usine à créer des objets. Pour créer un Point, vous appelez Point comme si c'était une fonction.

```
>>> pt = Point()
>>> pt
<__main__.Point object at 0xb7e9d3ac>
```

La valeur de retour est une référence à un objet Point, que nous attribuons à pt.

La création d'un nouvel objet s'appelle **instanciation**, et l'objet est une **instance** de la classe.

Lorsque vous affichez une instance, Python vous indique à quelle classe elle appartient et où elle est stockée dans la mémoire (le préfixe 0x signifie que le nombre qui suit est en hexadécimal).

Chaque objet est une instance d'une classe, donc « objet » et « instance » sont des termes interchangeables. Mais, dans ce chapitre, j'utilise « instance » pour indiquer que je parle d'un type défini par le programmeur.

15-2 - Attributs

Vous pouvez attribuer des valeurs à une instance en utilisant la notation pointée :

```
>>> pt.x = 3.0
>>> pt.y = 4.0
```

Cette syntaxe est similaire à la syntaxe pour sélectionner une variable d'un module, telle que math.pi ou string.whitespace. Dans ce cas, cependant, nous affectons des valeurs aux éléments nommés d'un objet. Ces éléments s'appellent des attributs.

Le schéma suivant montre le résultat de ces affectations. Un diagramme d'état qui montre un objet et ses attributs s'appelle un **diagramme d'objets** ; voir Figure 15.1.

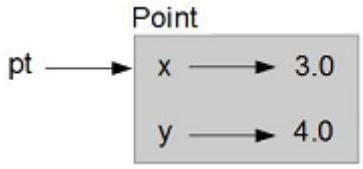


Figure 15.1 : Diagramme d'objets.

La variable pt fait référence à un objet Point, qui contient deux attributs. Chaque attribut se réfère à un nombre à virgule flottante.

Vous pouvez lire la valeur d'un attribut en utilisant la même syntaxe :

```
>>> pt.y
4.0
>>> x = pt.x
>>> x
```

```
3.0
```

L'expression pt.x signifie, « Va à l'objet référencé par pt et amène la valeur de x. » Dans l'exemple, nous attribuons cette valeur à une variable nommée x. Il n'y a aucun conflit entre la variable x et l'attribut x.

Vous pouvez utiliser la notation pointée dans le cadre d'une expression quelconque. Par exemple :

```
>>> '(%g, %g)' % (pt.x, pt.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(pt.x**2 + pt.y**2)
>>> distance
5.0
```

Vous pouvez passer une instance comme argument de la manière habituelle. Par exemple :

```
def afficher_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

afficher_point prend un point comme argument et l'affiche dans la notation mathématique. Pour l'appeler, vous pouvez passer pt comme argument :

```
>>> afficher_point(pt)
(3.0, 4.0)
```

Dans la fonction, p est un alias pour pt, donc si la fonction modifie p, pt est modifié.

À titre d'exercice, écrivez une fonction appelée distance_entre_points qui prend deux Points comme arguments et renvoie la distance entre eux.

15-3 - Rectangles

Parfois, ce que les attributs d'un objet devraient être est évident, mais d'autres fois vous devez prendre des décisions. Par exemple, imaginez que vous concevez une classe pour représenter des rectangles. Quels sont les attributs que vous voudriez utiliser pour spécifier l'emplacement et la taille d'un rectangle ? Vous pouvez ignorer l'angle ; pour faire simple, supposons que le rectangle est soit vertical, soit horizontal.

Il existe au moins deux possibilités :

- vous pourriez spécifier un coin du rectangle (ou le centre), la largeur et la hauteur ;
- vous pourriez spécifier deux coins opposés.

À ce stade, il est difficile de dire si l'une est meilleure que l'autre, donc nous allons mettre en œuvre la première, juste à titre d'exemple.

Voici la définition de la classe :

```
class Rectangle:
    """Représente un rectangle.

    attributs: largeur, hauteur, coin.
    """
```

La *docstring* répertorie les attributs : largeur et hauteur sont des nombres ; coin est un objet Point qui spécifie le coin inférieur gauche.

Pour représenter un rectangle, vous devez instancier un objet Rectangle et attribuer des valeurs aux attributs :

```
rect = Rectangle()
rect.largeur = 100.0
rect.hauteur = 200.0
rect.coin = Point()
rect.coin.x = 0.0
rect.coin.y = 0.0
```

L'expression rect.coin.x signifie, « Va à l'objet auquel se réfère rect et sélectionne l'attribut nommé coin ; puis va à cet objet et sélectionne l'attribut nommé x. »

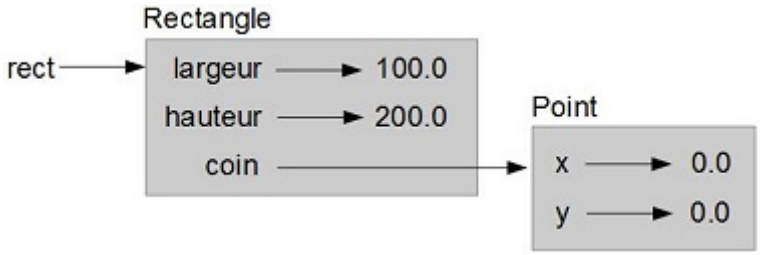


Figure 15.2 : Diagramme d'objets.

La figure 15.2 montre l'état de cet objet. Un objet qui est un attribut d'un autre objet est **inclus**.

15-4 - Instances comme valeurs de retour

Les fonctions peuvent renvoyer des instances. Par exemple, trouver_centre prend un Rectangle comme argument et renvoie un Point qui contient les coordonnées du centre du Rectangle :

```
def trouver_centre(rectangle):
    p = Point()
    p.x = rectangle.coin.x + rectangle.largeur/2
    p.y = rectangle.coin.y + rectangle.hauteur/2
    return p
```

Voici un exemple qui passe rect comme argument et attribue le Point résultant à centre :

```
>>> centre = trouver_centre(rect)
>>> afficher_point(centre)
(50, 100)
```

15-5 - Les objets sont modifiables

Vous pouvez modifier l'état d'un objet en faisant une affectation à l'un de ses attributs. Par exemple, pour modifier la taille d'un rectangle sans modifier sa position, vous pouvez modifier les valeurs de largeur et de hauteur :

```
rect.largeur = rect.largeur + 50
rect.hauteur = rect.hauteur + 100
```

Vous pouvez également écrire des fonctions qui modifient les objets. Par exemple, agrandir_rectangle prend un objet Rectangle et deux nombres, dLargeur et dHauteur, et ajoute les nombres à la largeur et la hauteur du rectangle :

```
def agrandir_rectangle(rectangle, dLargeur, dHauteur):
    rectangle.largeur += dLargeur
    rectangle.hauteur += dHauteur
```

Voici un exemple qui illustre l'effet :

```
>>> rect.largeur, rect.hauteur
(150.0, 300.0)
```

```
>>> agrandir_rectangle(rect, 50, 100)
>>> rect.largeur, rect.hauteur
(200.0, 400.0)
```

À l'intérieur de la fonction, rectangle est un alias pour rect, donc quand la fonction modifie rectangle, rect est modifié.

À titre d'exercice, écrivez une fonction nommée deplacer_rectangle qui prend un Rectangle et deux nombres nommés dx et dy. Elle devrait modifier l'emplacement du rectangle en ajoutant dx à la coordonnée x de coin et dy à la coordonnée y de coin.

15-6 - Copier

L'aliasing peut rendre un programme difficile à lire parce que les changements à un seul endroit pourraient avoir des effets inattendus dans un autre endroit. Il est difficile de garder une trace de toutes les variables qui pourraient se référer à un objet donné.

Copier un objet est souvent une solution de rechange à l'aliasing. Le module copy contient une fonction appelée copy qui peut dupliquer un objet quelconque :

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 et p2 contiennent les mêmes données, mais ils ne sont pas le même Point.

```
>>> afficher_point(p1)
(3, 4)
>>> afficher_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

L'opérateur is indique que p1 et p2 ne sont pas le même objet, comme nous nous y attendions. Mais vous vous attendiez peut-être que l'égalité == soit vraie, parce que ces points contiennent les mêmes données. Dans ce cas, vous serez déçu d'apprendre que pour les instances, le comportement par défaut de l'opérateur == est le même que pour l'opérateur is ; il vérifie l'identité des objets, pas leur équivalence. Cela arrive parce que, pour les types définis par le programmeur, Python ne sait pas ce qui devrait être considéré comme équivalent. Du moins, pas encore.

Si vous utilisez copy.copy pour dupliquer un Rectangle, vous verrez qu'il copie l'objet Rectangle, mais pas le Point inclus.

```
>>> rect2 = copy.copy(rect)
>>> rect2 is rect
False
>>> rect2.coin is rect.coin
True
```

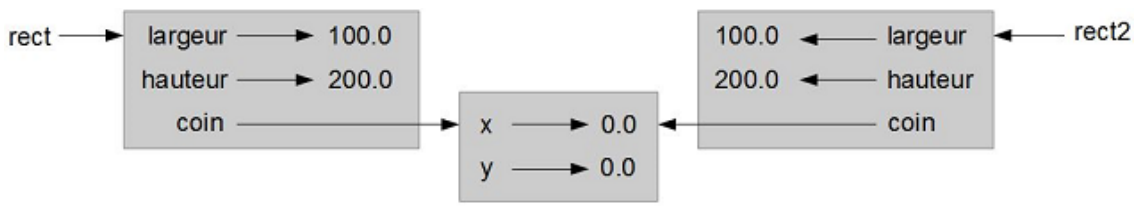


Figure 15.3 : Diagramme d'objets.

La figure 15.3 montre à quoi ressemble le diagramme d'objet. Cette opération s'appelle une **copie superficielle**, car elle copie l'objet et toutes les références qu'il contient, mais pas les objets inclus.

Pour la majorité des applications, ce n'est pas ce que vous voulez. Dans cet exemple, l'invocation de agrandir_rectangle sur l'un des Rectangles n'affecterait pas l'autre, mais l'invocation de deplacer_rectangle sur l'un d'eux affecterait tous les deux ! Ce comportement est source de confusion et d'erreurs.

Heureusement, le module copy fournit une méthode nommée deepcopy qui copie non seulement l'objet, mais aussi les objets auxquels il se réfère, et les objets auxquels ces derniers se réfèrent, et ainsi de suite. Vous ne serez pas surpris d'apprendre que cette opération s'appelle une **copie en profondeur**.

```
>>> rect3 = copy.deepcopy(rect)
>>> rect3 is rect
False
>>> rect3.coin is rect.coin
False
```

rect3 et rect sont des objets complètement séparés.

À titre d'exercice, écrivez une version de deplacer_rectangle qui crée et renvoie un nouveau Rectangle au lieu de modifier l'ancien.

15-7 - Débogage

Lorsque vous commencez à travailler avec des objets, vous pourriez rencontrer de nouvelles exceptions. Si vous essayez d'accéder à un attribut qui n'existe pas, vous obtenez une AttributeError :

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

Si vous n'êtes pas sûr de quel type est un objet, vous pouvez demander :

```
>>> type(p)
<class '__main__.Point'>
```

Vous pouvez également utiliser isinstance pour vérifier si un objet est une instance d'une classe :

```
>>> isinstance(p, Point)
True
```

Si vous n'êtes pas sûr si un objet possède un attribut particulier, vous pouvez utiliser la fonction interne hasattr :

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Le premier argument peut être un objet quelconque ; le second argument est une chaîne qui contient le nom de l'attribut.

Vous pouvez également utiliser une instruction try pour voir si l'objet possède les attributs dont vous avez besoin :

```
try:
    x = p.x
except AttributeError:
```

```
x = 0
```

Cette approche peut faciliter l'écriture des fonctions qui travaillent avec différents types. Nous reviendrons sur ce sujet dans la section **17.9**.

15-8 - Glossaire

- **classe :** un type défini par le programmeur. Une définition de classe crée un nouvel objet classe.
- **objet classe :** un objet qui contient des informations sur un type défini par le programmeur. L'objet classe peut être utilisé pour créer des instances du type.
- **instance :** un objet qui appartient à une classe.
- **instancier :** créer un nouvel objet.
- **attribut :** une des valeurs nommées associées à un objet.
- **objet inclus :** un objet qui est stocké comme un attribut d'un autre objet.
- **copie superficielle :** copier le contenu d'un objet, y compris les références à des objets inclus ; mise en œuvre par la fonction copy du module copy.
- **copie en profondeur :** copier le contenu d'un objet ainsi que tous les objets inclus, et les objets inclus dans ces derniers, et ainsi de suite ; mise en œuvre par la fonction deepcopy du module copy.
- **diagramme d'objets :** un diagramme qui montre les objets, leurs attributs et les valeurs des attributs.

15-9 - Exercices

Exercice 1

Écrivez une définition pour une classe nommée Cercle *ayant les attributs* centre *et* rayon*, où le centre est un objet* Point *et le rayon est un nombre.*

Instanciez un objet Cercle qui représente un cercle ayant son centre à (150, 100) *et un rayon de* 75.

Écrivez une fonction nommée point_du_cercle *qui prend un Cercle et un Point et renvoie Vrai si le Point se trouve dans le cercle ou sur sa circonférence.*

Écrivez une fonction nommée rect_du_cercle *qui prend un Cercle et un Rectangle et renvoie Vrai si le rectangle se trouve entièrement dans le cercle ou sur sa circonférence.*

Écrivez une fonction nommée rect_chevauche_cercle *qui prend un Cercle et un Rectangle et renvoie Vrai si l'un des coins du Rectangle se trouve à l'intérieur du cercle. Ou, version plus difficile, retourne Vrai si une partie quelconque du Rectangle se trouve à l'intérieur du cercle.*

Solution : **Circle.py**.

Exercice 2

Écrivez une fonction appelée dessine_rect *qui prend un objet Tortue et un Rectangle et utilise la Tortue pour dessiner le Rectangle. Voir le chapitre* **4** *pour des exemples utilisant des objets Tortue.*

Écrivez une fonction appelée dessine_cercle *qui prend une tortue et un cercle et dessine le cercle.*

Solution : **draw.py**.

16 - Classes et fonctions

Maintenant que nous savons comment créer de nouveaux types, l'étape suivante consiste à écrire des fonctions qui prennent comme paramètres et renvoient comme résultats des objets définis par le programmeur. Dans ce chapitre, je présente également le « style fonctionnel de programmation » et deux nouveaux modèles de développement de programmes.

Les exemples de code de ce chapitre sont disponibles à l'adresse **Time1.py**. Les solutions aux exercices sont à l'adresse **Time1_soln.py**.

16-1 - Temps

Nous allons définir une classe appelée Temps, qui enregistre le moment de la journée et constituera un nouvel exemple de type défini par le programmeur. La définition de la classe ressemble à ceci :

```
class Temps:
    """Représente le moment de la journée.

    attributs : heure, minute, seconde
    """
```

Nous pouvons créer un nouvel objet de type Temps et attribuer des valeurs à heures, minutes et secondes :

```
moment = Temps()
moment.heure = 11
moment.minute = 59
moment.seconde = 30
```

Le diagramme d'état pour l'objet Temps ressemble à la figure 16.1.

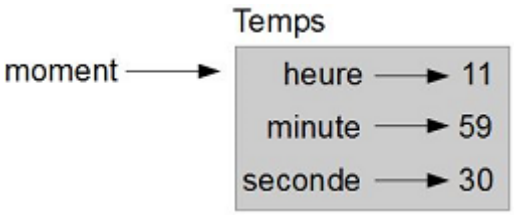


Figure 16.1 : Diagramme d'objets.

À titre d'exercice, écrivez une fonction appelée afficher_temps qui prend un objet Temps et l'affiche dans la forme heure:minute:seconde. Indice : la séquence de formatage '%.2d' affiche un entier en utilisant au moins deux chiffres, en ajoutant un zéro en début si nécessaire.

Écrivez une fonction booléenne nommée est_apres qui prend deux objets Temps, t1 et t2, et retourne True si t1 suit chronologiquement t2 et False sinon. Défi : n'utilisez pas d'instruction if.

16-2 - Fonctions pures

Dans les prochaines sections, nous écrirons deux fonctions qui additionnent des valeurs temporelles. Elles exemplifient deux types de fonctions : les fonctions pures et les modificateurs. Elles illustrent également un modèle de développement que j'appellerai **prototypage et correction**, dont l'objectif est d'aborder un problème complexe en commençant par un simple prototype et en traitant les difficultés de façon incrémentielle.

Voici un prototype simple de ajouter_temps :

```
def ajouter_temps(t1, t2):
    somme = Temps()
    somme.heure = t1.heure + t2.heure
    somme.minute = t1.minute + t2.minute
    somme.seconde = t1.seconde + t2.seconde
    return somme
```

La fonction crée un nouvel objet Temps, initialise ses attributs et renvoie une référence au nouvel objet. Cela s'appelle une **fonction pure**, car elle ne modifie aucun des objets qui lui sont passés comme arguments et elle n'a aucun effet, comme l'affichage d'une valeur ou l'obtention des données saisies par l'utilisateur, autre que de renvoyer une valeur.

Pour tester cette fonction, je vais créer deux objets Temps : debut contient l'heure de début d'un film, comme *Monty Python : Sacré Graal !*, et duree contient la durée du film, qui est d'une heure 35 minutes.

ajouter_temps calcule quand le film sera fini.

```
>>> debut = Temps()
>>> debut.heure = 9
>>> debut.minute = 45
>>> debut.seconde = 0

>>> duree = Temps()
>>> duree.heure = 1
>>> duree.minute = 35
>>> duree.seconde = 0

>>> fini = ajouter_temps(debut, duree)
>>> afficher_temps(fini)
10:80:00
```

Le résultat, 10:80:00 n'est peut-être pas ce que vous espériez. Le problème est que cette fonction ne traite pas les cas où le nombre de secondes ou de minutes additionnées dépasse soixante. Lorsque cela se produit, nous devons « retenir » ou « reporter » les secondes supplémentaires dans la colonne des minutes ou les minutes supplémentaires dans la colonne des heures.

Voici une version améliorée :

```
def ajouter_temps(t1, t2):
    somme = Temps()
    somme.heure = t1.heure + t2.heure
    somme.minute = t1.minute + t2.minute
    somme.seconde = t1.seconde + t2.seconde

    if somme.seconde >= 60:
        somme.seconde -= 60
        somme.minute += 1

    if somme.minute >= 60:
        somme.minute -= 60
        somme.heure += 1

    return somme
```

Bien que cette fonction soit correcte, elle commence à beaucoup grossir. Nous verrons plus tard une version plus courte.

16-3 - Modificateurs

Parfois, il est utile qu'une fonction puisse modifier les objets qu'elle reçoit comme paramètres. Dans ce cas, les changements sont visibles par la procédure appelante. Ce genre de fonctions s'appellent **modificateurs**.

incremente, qui ajoute un nombre donné de secondes à un objet Temps, peut être écrite naturellement comme un modificateur. Voici un premier jet :

```
def incremente(temps, secondes):
    temps.seconde += secondes

    if temps.seconde >= 60:
        temps.seconde -= 60
        temps.minute += 1

    if temps.minute >= 60:
        temps.minute -= 60
        temps.heure += 1
```

La première ligne effectue l'opération d'addition ; le reste traite les cas particuliers que nous avons vus dans la section précédente.

Cette fonction est-elle correcte ? Qu'advient-il si secondes est beaucoup plus élevé que soixante ?

Dans ce cas, il ne suffit pas de faire la retenue une seule fois ; nous devons continuer à la faire jusqu'à ce que la valeur temps.seconde soit inférieure à soixante. Une solution consiste à remplacer les instructions if par des instructions while. Cela rendrait la fonction correcte, mais pas très efficace. À titre d'exercice, écrivez une version correcte de incremente ne contenant aucune boucle.

Tout ce qui peut être fait avec des modificateurs peut également être fait avec des fonctions pures. En fait, certains langages de programmation dits « fonctionnels » autorisent uniquement les fonctions pures. Il existe certaines indications que les programmes qui utilisent des fonctions pures sont plus rapides à développer et moins sujets aux erreurs que les programmes qui utilisent des modificateurs. Mais les modificateurs sont commodes parfois, et les programmes fonctionnels ont tendance à être moins efficaces.

En général, je vous conseille d'écrire des fonctions pures chaque fois que cela est raisonnable et de recourir à des modificateurs uniquement si cela présente un avantage convaincant. Cette approche pourrait s'appeler un **style fonctionnel de programmation**.

À titre d'exercice, écrivez une version « pure » de incremente, qui crée et retourne un nouvel objet Temps plutôt que de modifier le paramètre.

16-4 - Prototypage versus planification

Le modèle de développement que j'illustre ici s'appelle « prototypage et correction ». Pour chaque fonction, j'ai écrit un prototype qui effectue le calcul de base et ensuite je l'ai testé, en corrigeant les erreurs en chemin.

Cette approche peut être efficace, surtout si vous n'avez pas encore une compréhension profonde du problème. Mais les corrections incrémentales peuvent générer du code qui est inutilement compliqué - car il traite de nombreux cas spéciaux - et non fiable - car il est difficile de savoir si vous avez trouvé toutes les erreurs.

Une autre possibilité est le **développement par conception**, dans lequel une compréhension de haut niveau du problème peut rendre la programmation beaucoup plus facile. Dans ce cas, il s'agit de comprendre qu'un objet Temps est en fait un nombre à trois chiffres exprimés en base 60 (voir **https://fr.wikipedia.org/wiki/Syst%C3%A8me_sexag%C3%A9simal**) ! L'attribut seconde est la « colonne des unités », l'attribut minute est « la colonne des soixantaines », et l'attribut heure est « la colonne des trois mille six cents ».

Lorsque nous avons écrit ajouter_temps et increment, nous étions effectivement en train de faire des additions en base 60, ce qui explique pourquoi nous avons dû faire des retenues d'une colonne à l'autre.

Cette observation suggère une autre approche de l'ensemble du problème - nous pouvons convertir des objets Temps en entiers et profiter du fait que l'ordinateur sait comment faire l'arithmétique avec des entiers.

Voici une fonction qui convertit des Temps en entiers :

```
def temps_vers_int(temps):
    minutes = temps.heure * 60 + temps.minute
    secondes = minutes * 60 + temps.seconde
    return secondes
```

Et voici une fonction qui convertit un nombre entier vers un Temps (rappelez-vous que divmod effectue une division entière du premier argument par le second et renvoie le quotient et le reste sous la forme d'un tuple).

```
def int_vers_temps(secondes):
    temps = Temps()
    minutes, temps.seconde = divmod(secondes, 60)
    temps.heure, temps.minute = divmod(minutes, 60)
    return temps
```

Vous avez peut-être dû réfléchir un peu et exécuter quelques tests pour vous convaincre que ces fonctions sont correctes. Une façon de les tester est de vérifier que temps_vers_int(int_vers_temps(x)) == x pour de nombreuses valeurs de x. Cela représente un exemple d'un contrôle de cohérence.

Une fois que vous êtes convaincu qu'elles sont correctes, vous pouvez les utiliser pour réécrire ajouter_temps :

```
def ajouter_temps(t1, t2):
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(seconds)
```

Cette version est plus courte que l'original, et plus facile à vérifier. À titre d'exercice, réécrivez incremente en utilisant temps_vers_int et int_vers_temps.

À certains égards, les conversions de base 60 à base 10 et vice-versa sont plus difficiles que le simple traitement des temps. La conversion de base est plus abstraite ; notre intuition pour traiter les valeurs de temps est meilleure.

Mais si nous avons l'intuition de traiter le temps comme un nombre en base 60 et nous faisons l'investissement d'écrire les fonctions de conversion (temps_vers_int et int_vers_temps), nous obtenons un programme qui est plus court, plus facile à lire et à déboguer, et plus fiable.

Il est également plus facile d'ajouter des fonctionnalités plus tard. Par exemple, imaginez la soustraction de deux Temps pour trouver la durée écoulée entre eux. L'approche naïve serait de mettre en œuvre la soustraction avec retenue. L'utilisation des fonctions de conversion serait plus facile et plus susceptible d'être correcte.

Ironie du sort, parfois le fait de rendre un problème plus difficile (ou plus général) le rend plus facile (car il y a moins de cas particuliers et moins de possibilités d'erreur).

16-5 - Débogage

Un objet Temps est bien formé si les valeurs de minute et seconde sont entre 0 et 60 (0 compris, mais 60 non compris) et si heure est positive. heure et minute doivent être des valeurs entières, mais nous pourrions permettre à seconde d'avoir une partie fractionnaire.

De telles exigences s'appellent des **invariants**, parce qu'elles doivent toujours être satisfaites. Autrement dit, si elles ne sont pas vraies, quelque chose a mal tourné.

Écrire du code pour vérifier les invariants peut aider à détecter les erreurs et à trouver leurs causes. Par exemple, vous pourriez avoir une fonction comme valider_temps qui prend un objet Temps et renvoie False si elle enfreint un invariant :

```
def valide_temps(temps):
```

```
    if temps.heure < 0 or temps.minute < 0 or temps.seconde < 0:
        return False
    if temps.minute >= 60 or temps.seconde >= 60:
        return False
    return True
```

Au début de chaque fonction, vous pourriez vérifier les arguments pour vous assurer qu'ils sont valides :

```
def ajouter_temps(t1, t2):
    if not valide_temps(t1) or not valide_temps(t2):
        raise ValueError('objet Temps invalide dans ajouter_temps')
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(secondes)
```

Ou vous pouvez utiliser une **instruction assert**, qui vérifie un invariant donné et déclenche une exception si elle échoue :

```
def ajouter_temps(t1, t2):
    assert valide_temps(t1) and valide_temps(t2)
    secondes = temps_vers_int(t1) + temps_vers_int(t2)
    return int_vers_temps(secondes)
```

Les instructions assert sont utiles, car elles font la distinction entre un code qui traite des conditions normales et un code qui détecte les erreurs.

16-6 - Glossaire

- **prototypage et correction :** un modèle de développement qui consiste à écrire un brouillon d'un programme, à tester et à corriger les erreurs trouvées.
- **développement par conception :** un modèle de développement qui implique une compréhension de haut niveau du problème et plus de planification que du développement incrémental ou du développement de prototypage.
- **fonction pure :** une fonction qui ne modifie pas les objets qu'elle reçoit comme arguments. La plupart des fonctions pures sont productives.
- **modificateur :** une fonction qui modifie un ou plusieurs objets qu'elle reçoit comme arguments. La plupart des modificateurs sont vides ; c'est-à-dire ils renvoient None.
- **style fonctionnel de programmation :** un style de conception de programmes dans lequel la majorité des fonctions sont pures.
- **invariant :** une condition qui doit toujours être vraie pendant l'exécution d'un programme.
- **instruction assert :** une instruction que vérifie une condition et déclenche une exception si elle échoue.

16-7 - Exercices

Exercice 1

Écrivez une fonction appelée mul_time *qui prend un objet* Temps *et un nombre et retourne un nouvel objet* Temps *, qui contient le produit entre le* Temps *d'origine et le nombre.*

Ensuite, utilisez mul_time *pour écrire une fonction qui prend un objet* Temps *qui représente le temps de l'arrivée dans une course, et un nombre qui représente la distance, et retourne un objet* Temps *qui représente le rythme moyen (temps par kilomètre).*

Exercice 2

Le module datetime *fournit des objets* time *qui sont similaires aux objets* Temps *de ce chapitre, mais ils fournissent un riche ensemble de méthodes et d'opérateurs. Lisez-en la documentation à l'adresse* **http://docs.python.org/3/library/datetime.html**.

1 *Utilisez le module* `datetime` *pour écrire un programme qui prend la date actuelle et affiche le jour de la semaine.*
2 *Écrivez un programme qui prend en entrée un anniversaire et affiche l'âge de l'utilisateur et le nombre de jours, heures, minutes et secondes jusqu'à son prochain anniversaire.*
3 *Si deux personnes sont nées deux jours différents, il existe un jour où l'une d'entre elles est deux fois plus âgée que l'autre. C'est leur Jour double. Écrivez un programme qui prend deux dates de naissance et calcule leur Jour double.*
4 *Pour pimenter un peu le défi, écrivez la version plus générale qui calcule le jour où l'une des personnes est* n *fois plus âgée que l'autre.*

Solution à l'adresse **Time1_soln.py**. *Pour fonctionner, le code proposé ici nécessite la présence du fichier* `Time1.py` *contenant le code des programmes de ce chapitre (lien fourni en début de chapitre)*.

17 - Classes et méthodes

Bien que nous y ayons utilisé certaines fonctionnalités orientées objet de Python, les programmes des deux derniers chapitres ne sont pas vraiment orientés objet parce qu'ils ne représentent pas les relations entre les types définis par le programmeur et les fonctions qui opèrent sur eux. La prochaine étape est de transformer ces fonctions en des méthodes qui rendent les relations explicites.

Les exemples de code de ce chapitre sont disponibles sur **Time2.py**, et les solutions aux exercices se trouvent à l'adresse **Point2_soln.py**.

17-1 - Fonctionnalités orientées objet

Python est un **langage de programmation orienté objet**, ce qui signifie qu'il offre des fonctionnalités qui prennent en charge la programmation orientée objet, laquelle présente ces caractéristiques déterminantes :

- les programmes incluent des définitions de classes et de méthodes ;
- la plus grande partie du calcul est exprimé en tant qu'opérations sur des objets ;
- les objets représentent souvent des choses dans le monde réel, et les méthodes correspondent souvent à la manière dont les choses du monde réel interagissent.

Par exemple, la classe `Temps` définie au chapitre **16** correspond à la façon dont on enregistre habituellement le moment de la journée, et les fonctions que nous avons définies correspondent aux types de choses que l'on fait avec des temps. De même, les classes `Point` et `Rectangle` du chapitre **15** correspondent aux concepts mathématiques de point et de rectangle.

Jusqu'à présent, nous n'avons pas profité des fonctionnalités que Python fournit pour soutenir la programmation orientée objet. Ces fonctionnalités ne sont pas strictement nécessaires ; la plupart d'entre elles fournissent une syntaxe de rechange pour les choses que nous avons déjà faites. Mais dans de nombreux cas, cette nouvelle syntaxe est plus concise et transmet la structure du programme de façon plus précise.

Par exemple, dans `Time1.py`, il n'y a aucun lien évident entre la définition de la classe et les définitions de fonctions qui suivent. Après un bref examen, il est évident que chaque fonction prend comme argument au moins un objet `Temps`.

Cette observation est la motivation pour les **méthodes** ; une méthode est une fonction associée à une classe particulière. Nous avons vu des méthodes relatives aux chaînes de caractères, listes, dictionnaires et tuples. Dans ce chapitre, nous allons définir des méthodes pour les types définis par le programmeur.

Les méthodes sont sémantiquement la même chose que des fonctions, mais il y a deux différences syntaxiques :

- les méthodes sont définies à l'intérieur d'une définition de classe afin de rendre explicite la relation entre la classe et ses méthodes ;

- la syntaxe d'invocation d'une méthode est différente de la syntaxe d'appel d'une fonction.

Dans les prochaines sections, nous allons prendre les fonctions des deux chapitres précédents et les transformer en méthodes. Cette transformation est purement mécanique ; vous pouvez le faire en suivant une séquence d'étapes. Si vous êtes à l'aise avec la conversion d'une forme à une autre, vous serez en mesure de choisir la meilleure forme pour tout ce que vous faites.

17-2 - Afficher des objets

Dans le chapitre **16**, nous avons défini une classe nommée Temps et dans la section **16.1**, vous avez écrit une fonction nommée afficher_temps :

```
class Temps:
    """Représente le moment de la journée."""

def afficher_temps(temps):
    print('%.2d:%.2d:%.2d' % (temps.heure, temps.minute, temps.seconde))
```

Pour appeler cette fonction, vous devez passer comme argument un objet Temps.

```
>>> debut = Temps()
>>> debut.heure = 9
>>> debut.minute = 45
>>> debut.seconde = 00
>>> afficher_temps(debut)
09:45:00
```

Pour faire de la fonction afficher_temps une méthode, tout ce que nous devons faire est de déplacer la définition de la fonction à l'intérieur de la définition de classe.

```
class Temps:
    def afficher_temps(temps):
        print('%.2d:%.2d:%.2d' % (temps.heure, temps.minute, temps.seconde))
```

Maintenant, il y a deux façons d'appeler afficher_temps. La première (et la moins courante) est d'utiliser la syntaxe de fonction :

```
>>> Temps.afficher_temps(debut)
09:45:00
```

Dans cette utilisation de la notation pointée, Temps est le nom de la classe et afficher_temps est le nom de la méthode. debut est passé en paramètre.

La seconde façon, plus concise, est d'utiliser la syntaxe de méthode :

```
>>> debut.afficher_temps()
09:45:00
```

Dans cette utilisation de la notation pointée, afficher_temps est (toujours) le nom de la méthode et debut est l'objet sur lequel la méthode est invoquée, qui s'appelle le **sujet**. Tout comme le sujet d'une phrase est ce dont parle la phrase, le sujet d'un appel de méthode est ce à quoi s'applique la méthode.

À l'intérieur de la méthode, le sujet est affecté au premier paramètre, donc dans ce cas debut est affecté à temps.

Par convention, le premier paramètre d'une méthode s'appelle self, donc il serait plus usuel d'écrire afficher_temps comme ceci :

```
class Temps:
```

```
    def afficher_temps(self):
        print('%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde))
```

La raison de cette convention est une métaphore implicite :

- la syntaxe d'un appel de fonction, afficher_temps(debut), suggère que la fonction est l'agent actif. Il dit quelque chose comme « Hé, afficher_temps ! Voici un objet à afficher » ;
- dans la programmation orientée objet, les objets sont les agents actifs. Un appel de méthode comme debut.afficher_temps() dit « Hé, debut ! S'il te plaît, affiche toi-même ».

Ce changement de perspective est peut-être plus poli, mais il n'est pas évident que cela soit très utile. Dans les exemples que nous avons vus jusqu'à présent, cela pourrait ne pas l'être. Mais parfois, le déplacement de la responsabilité des fonctions sur les objets rend possible l'écriture des fonctions (ou méthodes) plus polyvalentes, et rend le code plus facile à maintenir et à réutiliser.

À titre d'exercice, réécrivez temps_vers_int (de la section **16.4**) comme une méthode. Vous pourriez être tenté à réécrire également int_vers_temps comme une méthode, mais cela n'a pas vraiment de sens, car il n'y a aucun objet sur lequel elle serait invoquée.

17-3 - Un autre exemple

Voici une version de la fonction incremente (de la section **16.3**) réécrite comme une méthode :

```
# à l'intérieur de la classe Temps :

    def incremente(self, secondes):
        secondes += self.temps_vers_int()
        return int_vers_temps(secondes)
```

Cette version suppose que temps_vers_int soit écrite comme une méthode. En outre, notez que c'est une fonction pure, pas un modificateur.

Voici comment vous invoquez incremente :

```
>>> debut.afficher_temps()
09:45:00
>>> fin = debut.incremente(1337)
>>> fin.afficher_temps()
10:07:17
```

Le sujet, debut, est attribué au premier paramètre, self. L'argument, 1337, est attribué au second paramètre, secondes.

Ce mécanisme peut être source de confusion, surtout si vous faites une erreur. Par exemple, si vous invoquez incremente avec deux arguments, vous obtenez :

```
>>> end = start.incremente(1337, 460)
TypeError: incremente() takes 2 positional arguments but 3 were given
```

Le message d'erreur est au départ déroutant, car il y a seulement deux arguments entre les parenthèses. Mais le sujet est également considéré comme un argument, si bien qu'il y en a trois en tout.

Au fait, un argument positionnel est un argument qui ne possède aucun nom de paramètre ; autrement dit, ce n'est pas un argument mot-clé. Dans cet appel de fonction :

```
sketch(perroquet, cage, mort=True)
```

perroquet et cage sont des arguments positionnels et mort est un argument mot-clé.

17-4 - Un exemple plus compliqué

La réécriture de est_apres (de la section **16.1**) est un peu plus compliquée, car elle prend en paramètres deux objets Temps. Dans ce cas, la convention est de nommer le premier paramètre self et le second paramètre other :

```
# à l'intérieur de la classe Temps:

    def est_apres(self, other):
        return self.temps_vers_int() > other.temps_vers_int()
```

Pour utiliser cette méthode, vous devez l'invoquer sur un objet et passer l'autre comme argument :

```
>>> fin.est_apres(debut)
True
```

Une bonne chose à propos de cette syntaxe est qu'elle se lit presque comme en français : « fin est(-elle) après début ? »

17-5 - La méthode init

La méthode init (raccourci de « initialisation ») est une méthode spéciale qui est appelée automatiquement lorsqu'un objet est instancié. Son nom complet est __init__ (deux caractères de soulignement, suivis par init, puis deux autres caractères de soulignement). Une méthode init pour la classe Temps pourrait ressembler à ceci :

```
# à l'intérieur de la classe Temps :

    def __init__(self, heure = 0, minute = 0, seconde = 0):
        self.heure = heure
        self.minute = minute
        self.seconde = seconde
```

Il est courant que les paramètres de __init__ aient les mêmes noms que les attributs. L'instruction

```
self.heure = heure
```

stocke la valeur du paramètre heure en tant qu'attribut de self.

Les paramètres sont facultatifs, donc si vous appelez Temps sans arguments, vous obtenez les valeurs par défaut.

```
>>> temps = Temps()
>>> temps.afficher_temps()
00:00:00
```

Si vous passez un argument, il remplace heure :

```
>>> temps = Temps(9)
>>> temps.afficher_temps()
09:00:00
```

Si vous fournissez deux arguments, ils remplacent heure et minute.

```
>>> temps = Temps(9, 45)
>>> temps.afficher_temps()
09:45:00
```

Et si vous fournissez trois arguments, ils remplacent les trois valeurs par défaut.

À titre d'exercice, écrivez une méthode init pour la classe Point, qui prend x et y comme paramètres optionnels et les assigne aux attributs correspondants.

17-6 - La méthode __str__

__str__ est une méthode spéciale, comme __init__, qui est censée renvoyer une représentation sous forme de chaîne de caractères d'un objet.

Par exemple, voici une méthode str pour les objets Temps :

```
# à l'intérieur de la classe Temps :

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.heure, self.minute, self.seconde)
```

Lorsque vous affichez un objet, Python invoque la méthode str :

```
>>> temps = Temps(9, 45)
>>> print(temps)
09:45:00
```

Quand j'écris une nouvelle classe, je commence presque toujours par écrire __init__, qui rend plus facile l'instanciation des objets, et __str__, qui est utile pour le débogage.

À titre d'exercice, écrivez une méthode str pour la classe Point. Créez un objet Point et affichez-le.

17-7 - Surcharge d'opérateur

En définissant d'autres méthodes spéciales, vous pouvez spécifier le comportement des opérateurs sur les types définis par le programmeur. Par exemple, si vous définissez une méthode nommée __add__ pour la classe Temps, vous pouvez utiliser l'opérateur + sur les objets de type Temps.

Voici à quoi pourrait ressembler la définition :

```
# à l'intérieur de la classe Temps :

    def __add__(self, other):
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(seconds)
```

Et voici comment vous pouvez l'utiliser :

```
>>> debut = Temps(9, 45)
>>> duree = Temps(1, 35)
>>> print(debut + duree)
11:20:00
```

Lorsque vous appliquez l'opérateur + sur des objets de type Temps, Python invoque automatiquement __add__. Lorsque vous affichez le résultat, Python invoque __str__. Donc, il y a beaucoup de choses qui se passent dans les coulisses !

La modification du comportement d'un opérateur afin qu'il fonctionne avec des types définis par le programmeur s'appelle la **surcharge d'opérateur**. Pour chaque opérateur de Python, il existe une méthode

spéciale correspondante, comme __add__. Pour plus de détails, voir **http://docs.python.org/3/reference/datamodel.html#specialnames**.

À titre d'exercice, écrivez une méthode add pour la classe Point.

17-8 - Résolution de méthode basée sur le type

Dans la section précédente, nous avons additionné deux objets Temps, mais vous pouvez également additionner un nombre entier à un objet Temps. Ce qui suit est une version de __add__ qui vérifie le type de other et invoque soit ajouter_temps, soit incremente :

```
# à l'intérieur de la classe Temps :

    def __add__(self, other):
        if isinstance(other, Temps):
            return self.ajouter_temps(other)
        else:
            return self.incremente(other)

    def ajouter_temps(self, other):
        secondes = self.temps_vers_int() + other.temps_vers_int()
        return int_vers_temps(secondes)

    def incremente(self, secondes):
        secondes += self.temps_vers_int()
        return int_vers_temps(secondes)
```

La fonction interne isinstance prend une valeur et un objet classe, et retourne True si la valeur est une instance de la classe.

Si other est un objet de type Temps, __add__ invoque ajouter_temps. Sinon, il suppose que le paramètre est un nombre et invoque incremente. Cette opération s'appelle **résolution de méthode basée sur le type**, car elle choisit la méthode à employer sur la base du type des arguments.

Voici des exemples qui utilisent l'opérateur + avec différents types :

```
>>> debut = Time(9, 45)
>>> duree = Time(1, 35)
>>> print(debut + duree)
11:20:00
>>> print(debut + 1337)
10:07:17
```

Malheureusement, cette mise en œuvre de l'addition n'est pas commutative. Si le nombre entier est le premier opérande, vous obtenez

```
>>> print(1337 + debut)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Le problème est qu'au lieu de demander à l'objet de type Temps d'additionner un nombre entier, Python demande au nombre entier d'additionner un objet de type Temps, et il ne sait pas comment le faire. Mais il existe une solution intelligente pour ce problème : la méthode spéciale __radd__, qui signifie *right-side add*, « additionner à droite ». Cette méthode est invoquée quand un objet de type Temps apparaît du côté droit de l'opérateur +. Voici la définition :

```
# à l'intérieur de la classe Temps :

    def __radd__(self, other):
        return self.__add__(other)
```

Et voici comment l'utiliser :

```
>>> print(1337 + debut)
10:07:17
```

À titre d'exercice, écrivez une méthode add pour des objets Point qui fonctionne sur un objet de type Point ou un tuple :

- si le second opérande est un Point, la méthode doit retourner un nouveau Point dont l'abscisse x est la somme des abscisses x des opérandes, et dont l'ordonnée y est la somme des ordonnées y ;
- si le second opérande est un tuple, la méthode doit additionner le premier élément du tuple à l'abscisse x et le second élément à l'ordonnée y, et renvoyer un nouveau Point dont les coordonnées représentent le résultat.

17-9 - Polymorphisme

La résolution de méthode basée sur le type est utile quand elle est nécessaire, mais elle n'est (heureusement) pas toujours nécessaire. Souvent, vous pouvez l'éviter en écrivant des fonctions qui s'exécutent correctement pour des arguments de types différents.

La plupart des fonctions que nous avons écrites pour les chaînes de caractères acceptent aussi d'autres types de séquences. Par exemple, dans la section 11.2, nous utilisions un histogramme pour compter le nombre de fois où chaque lettre apparaît dans un mot.

```
def histogramme(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

Cette fonction peut être utilisée également avec des listes, tuples, et même des dictionnaires, à condition que les éléments de s soient hachables, de sorte qu'ils puissent être utilisés comme clés dans d.

```
>>> t = ['spam', 'omelette', 'spam', 'spam', 'bacon', 'spam']
>>> histogramme(t)
{'bacon': 1, 'omelette': 1, 'spam': 4}
```

Les fonctions qui acceptent plusieurs types sont dites **polymorphes**. Le polymorphisme peut faciliter la réutilisation du code. Par exemple, la fonction interne sum, qui ajoute des éléments à une séquence, fonctionne tant que les éléments de la séquence supportent l'addition.

Comme les objets de type Temps fournissent une méthode add, ils peuvent être passés en argument à sum :

```
>>> t1 = Temps(7, 43)
>>> t2 = Temps(7, 41)
>>> t3 = Temps(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

En général, si toutes les opérations à l'intérieur d'une fonction peuvent être effectuées sur un type donné, la fonction peut être utilisée avec ce type.

Le meilleur type de polymorphisme est celui involontaire, où vous découvrez que vous avez déjà écrit une fonction qui peut être appliquée à un type pour lequel elle n'était pas prévue.

17-10 - Débogage

Il est permis d'ajouter des attributs à des objets à tout moment de l'exécution d'un programme, mais si vous avez des objets du même type qui ne possèdent pas les mêmes attributs, il est facile de faire des erreurs. Il est souhaitable d'initialiser tous les attributs d'un objet dans la méthode init.

Si vous n'êtes pas sûr si un objet possède un attribut particulier, vous pouvez utiliser la fonction interne hasattr (voir la section **15.7**).

Une autre façon d'accéder à des attributs est la fonction intégrée vars, qui prend en paramètre un objet et renvoie un dictionnaire qui fait correspondre des noms des attributs (sous forme de chaînes de caractères) à leurs valeurs :

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

Aux fins de débogage, vous trouverez peut-être utile de retenir cette fonction très pratique :

```
def afficher_attributs(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

afficher_attributs parcourt le dictionnaire et affiche le nom de chaque attribut et sa valeur correspondante.

La fonction interne getattr prend un objet et un nom d'attribut (sous forme de chaîne de caractères) et renvoie la valeur de l'attribut.

17-11 - Interface et implémentation

L'un des objectifs de la conception orientée objet est de rendre le logiciel plus facile à maintenir, ce qui signifie que vous pouvez garder le programme en état de fonctionnement lorsque d'autres parties du système sont modifiées, et modifier le programme pour répondre à de nouveaux besoins.

Un principe de conception qui permet d'atteindre cet objectif est de garder les interfaces séparées des implémentations. Pour les objets, cela signifie que les méthodes fournies par une classe ne doivent pas dépendre de la façon dont les attributs sont représentés.

Par exemple, dans ce chapitre nous avons développé une classe qui représente un moment de la journée. Cette classe fournit notamment les méthodes temps_vers_int, est_apres et ajouter_temps.

Nous pouvons mettre en œuvre ces méthodes de plusieurs façons. Les détails de la mise en œuvre dépendent de la façon dont nous représentons le temps. Dans ce chapitre, les attributs d'un objet Temps sont heure, minute et seconde.

Une autre solution serait de remplacer ces attributs par un seul entier représentant le nombre de secondes depuis minuit. Cette mise en œuvre rendrait certaines méthodes, comme est_apres, plus facile à écrire, mais elle rend plus difficile l'écriture d'autres méthodes.

Après avoir déployé une nouvelle classe, vous découvrirez peut-être une meilleure mise en œuvre. Si d'autres parties du programme utilisent votre classe, modifier l'interface peut s'avérer fastidieux et être source d'erreurs.

Mais si vous avez soigneusement conçu l'interface, vous pouvez modifier la mise en œuvre interne sans modifier l'interface, ce qui signifie qu'il n'y aura pas besoin de modifier d'autres parties du programme.

17-12 - Glossaire

- **langage orienté objet :** un langage qui fournit des fonctionnalités, telles que les types et les méthodes définies par le programmeur, qui facilitent la programmation orientée objet.
- **programmation orientée objet :** un style de programmation dans lequel les données et les opérations qui les manipulent sont organisées en classes et méthodes.
- **méthode :** une fonction qui est définie à l'intérieur d'une définition de classe et est invoquée sur les instances de cette classe.
- **sujet :** l'objet sur lequel une méthode est invoquée.
- **argument positionnel :** un argument qui n'inclut pas un nom de paramètre, donc il n'est pas un argument mot-clé.
- **surcharge d'opérateur :** modification du comportement d'un opérateur comme `+` de sorte qu'il prenne en charge un type défini par le programmeur.
- **résolution de méthode basée sur le type :** un modèle de programmation qui vérifie le type d'un opérande et invoque des fonctions différentes pour des types différents.
- **polymorphe :** relatif à une fonction qui peut être utilisée pour plus d'un type.
- **dissimulation d'information :** le principe selon lequel l'interface fournie par un objet ne doit pas dépendre de sa mise en œuvre, en particulier de la représentation de ses attributs.

17-13 - Exercices

Exercice 1

Téléchargez le code de ce chapitre à partir de **Time2.py**. *Remplacez les attributs de* `Time` *par un seul entier représentant le nombre de secondes écoulées depuis minuit. Puis, modifiez les méthodes (et la fonction* `int_vers_temps`*) pour les adapter à la nouvelle mise en œuvre. Vous ne devriez pas avoir à modifier le code de test dans* `main`*. Lorsque vous avez terminé, le programme devrait afficher la même chose que précédemment. Solution :* **Time2_soln.py**.

Exercice 2

Cet exercice est un conte pédagogique sur l'une des erreurs les plus courantes et difficiles à trouver en Python. Écrivez une définition pour une classe nommée `Kangaroo` *(*kangaroo *est le mot anglais pour le kangourou), avec les méthodes suivantes :*

1. *Une méthode* `__init__` *qui initialise un attribut nommé* `pouch_contents` *à une liste vide (le mot anglais* pouch *désigne généralement un sac ou une bourse, mais plus précisément ici la poche ventrale dans laquelle la femelle du kangourou abrite son petit) ;*
2. *Une méthode nommée* `put_in_pouch` *qui prend un objet d'un type quelconque et l'ajoute à* `pouch_contents` *;*
3. *Une méthode* `__str__` *qui renvoie une représentation sous forme de chaîne de caractères de l'objet* `Kangaroo` *et du contenu de la poche.*

Testez votre code en créant deux objets `Kangaroo`*, en les attribuant à des variables nommées* `kanga` *et* `roo`*, puis en ajoutant* `roo` *au contenu de la poche de* `kanga`*.*

Téléchargez **BadKangaroo.py**. *Il contient une solution au problème précédent avec un gros bogue bien vicieux. Trouvez et corrigez le bogue.*

Si vous vous retrouvez coincé, vous pouvez télécharger **GoodKangaroo.py**, *qui explique le problème et montre une solution.*

18 - Héritage

La fonctionnalité la plus emblématique de la programmation orientée objet est l'héritage. L'héritage est la possibilité de définir une nouvelle classe, qui est une version modifiée d'une classe existante. Dans ce chapitre, j'illustre l'héritage en utilisant des classes qui représentent des cartes à jouer, des paquets de cartes et des mains de poker.

Si vous ne jouez pas au poker, vous pouvez lire à ce sujet sur **https://fr.wikipedia.org/wiki/Poker**, mais ce n'est pas obligatoire ; je vous dirai tout ce que vous devez savoir pour les exercices.

Les exemples de code de ce chapitre sont disponibles à l'adresse **Card.py**.

18-1 - Objets carte de jeu

Il y a cinquante-deux cartes dans un paquet, dont chacune appartient à une des quatre couleurs (ou enseignes) et à l'une des treize valeurs (ou rangs). Les couleurs sont pique, cœur, carreau, et trèfle (dans l'ordre décroissant au jeu de bridge). Les valeurs sont as, 2, 3, 4, 5, 6, 7, 8, 9, 10, valet, dame (ou reine) et roi. Selon le jeu auquel vous jouez, un as peut être plus fort que le roi ou plus faible que le 2.

Si nous voulons définir un nouvel objet pour représenter une carte à jouer, il est évident que les attributs doivent être la `couleur` et la `valeur`. Le type des attributs n'est pas si évident. Une possibilité est d'utiliser des chaînes contenant des mots comme `'pique'` pour les couleurs et `'dame'` pour les valeurs. Un problème avec cette modélisation est qu'il ne serait pas facile de comparer les cartes pour voir laquelle a une valeur ou une couleur supérieure.

Une autre possibilité est d'utiliser des entiers pour **encoder** les valeurs et les couleurs. Dans ce contexte, « encoder » signifie que nous allons définir une correspondance entre nombres et couleurs, ou entre nombres et valeurs. Ce type d'encodage n'est pas censé être secret (ce serait du « cryptage » ou du chiffrement).

Par exemple, ce tableau montre les couleurs et les valeurs entières correspondantes :

Pique	#	3
Cœur	#	2
Carreau	#	1
Trèfle	#	0

Ce code facilite la comparaison des cartes ; parce que les couleurs les plus élevées correspondent aux nombres plus élevés, nous pouvons comparer les couleurs en comparant leurs codes.

Le codage des valeurs est assez évident ; chacune des valeurs numériques des cartes correspond à l'entier correspondant, et pour les honneurs :

valet	#	11
dame	#	12
roi	#	13

J'utilise le symbole # pour qu'il soit clair que ces correspondances ne font pas partie du programme Python. Elles font partie de la conception du programme, mais elles n'apparaissent pas explicitement dans le code.

La définition de la classe `Carte` ressemble à ceci :

```
class Carte:
    """Représente une carte à jouer standard."""

    def __init__(self, couleur = 0, valeur = 2):
        self.couleur = couleur
```

```
        self.valeur = valeur
```

Comme d'habitude, la méthode init prend un paramètre optionnel pour chaque attribut. La carte par défaut est le 2 de trèfle.

Pour créer une carte, vous appelez Carte avec la couleur et la valeur de la carte souhaitée.

```
dame_de_carreau = Carte(1, 12)
```

18-2 - Attributs de classe

Pour afficher des objets de type Carte d'une manière lisible facilement pour les humains, nous avons besoin d'une correspondance entre les codes nombres entiers et les couleurs et les valeurs correspondantes. Une façon naturelle de le faire est d'utiliser des listes de chaînes de caractères. Nous attribuons ces listes aux **attributs de classe** :

```
# à l'intérieur de la classe Carte :

    noms_couleurs = ['trèfle', 'carreau', 'cœur', 'pique']
    noms_valeurs = [None, 'as', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'valet', 'dame', 'roi']

    def __str__(self):
        return '%s de %s' % (Carte.noms_valeurs[self.valeur],
                             Carte.noms_couleurs[self.couleur])
```

Les variables comme noms_couleurs et noms_valeurs, qui sont définies dans une classe, mais en dehors de toute méthode, s'appellent attributs de classe parce qu'elles sont associées à l'objet classe Carte.

Ce terme les distingue des variables telles que couleur et valeur, qui s'appellent **attributs d'instance** parce qu'elles sont associées à une instance particulière.

Les deux types d'attributs sont accessibles en utilisant la notation pointée. Par exemple, à l'intérieur de __str__, self est un objet Carte et self.couleur est sa couleur. De même, Carte est un objet classe, et Carte.noms_valeurs est une liste de chaînes de caractères associée à la classe.

Chaque carte a sa propre couleur et sa propre valeur, mais il n'y a qu'une seule copie de noms_couleurs et noms_valeurs.

En mettant le tout ensemble, l'expression Carte.noms_valeurs[self.valeur] signifie « utilise l'attribut valeur de l'objet self comme un index de la liste noms_valeurs de la classe Carte, et sélectionne la chaîne de caractères appropriée. »

Le premier élément de noms_valeurs est None, car il n'y existe aucune carte de rang zéro. En incluant None comme un espace réservé, nous obtenons une correspondance ayant comme belle propriété le fait que l'indice 2 corresponde à la chaîne de caractères '2', et ainsi de suite. Pour éviter de devoir faire cet ajustement, nous aurions pu utiliser un dictionnaire à la place d'une liste.

Avec les méthodes que nous avons jusqu'ici, nous pouvons créer et afficher des cartes :

```
>>> carte1 = Carte(2, 11)
>>> print(carte1)
valet de cœur
```

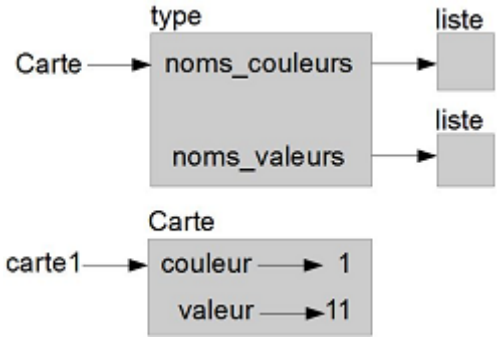


Figure 18.1 : Diagramme d'objets.

La figure 18.1 est un diagramme de l'objet classe Carte et d'une instance de Carte. Carte est un objet classe ; son type est type. L'objet carte1 est une instance de Carte, donc son type est Carte. Pour économiser l'espace, je n'ai pas dessiné le contenu de noms_couleurs et noms_valeurs.

18-3 - Comparer des cartes

Pour les types internes, il existe des opérateurs relationnels (<, >, ==, etc.) qui comparent des valeurs et déterminent si l'un est supérieur, inférieur ou égal à un autre. Pour les types définis par le programmeur, nous pouvons remplacer le comportement des opérateurs internes en fournissant une méthode nommée __lt__, qui signifie *less than*, « inférieur à ».

__lt__ prend deux paramètres, self et other, et renvoie True si self est strictement inférieur à other.

L'ordre correct des cartes n'est pas évident. Par exemple, qu'est-ce qui est mieux, le 3 de trèfle ou le 2 de carreau ? L'une a une valeur plus élevée, mais l'autre a une couleur plus élevée. Afin de comparer les cartes, vous devez décider si la valeur ou la couleur est plus importante.

La réponse pourrait dépendre du jeu auquel vous jouez, mais pour ne pas compliquer les choses, nous faisons le choix arbitraire que c'est la couleur qui primera, donc tous les piques surclassent tous les carreaux, et ainsi de suite.

Une fois cette décision prise, nous pouvons écrire __lt__ :

```
# à l'intérieur de la classe Carte :

    def __lt__(self, other):
        # vérifier les couleurs
        if self.couleur < other.couleur: return True
        if self.couleur > other.couleur: return False

        # les couleurs sont identiques... vérifier les valeurs
        return self.valeur < other.valeur
```

Vous pouvez réécrire cela d'une façon plus concise, en utilisant la comparaison de tuple :

```
# à l'intérieur de la classe Carte :

    def __lt__(self, other):
        t1 = self.couleur, self.valeur
        t2 = other.couleur, other.valeur
        return t1 < t2
```

À titre d'exercice, écrivez une méthode __lt__ pour des objets de type Temps. Vous pouvez utiliser la comparaison de tuple, mais vous pourriez aussi envisager la comparaison des entiers.

18-4 - Paquets de cartes

Maintenant que nous avons les cartes, la prochaine étape est de définir les Paquets de cartes. Comme un paquet est composé de cartes, il est naturel que chaque Paquet contienne comme attribut une liste de cartes.

Ce qui suit est une définition de classe pour Paquet. La méthode init crée l'attribut cartes et génère l'ensemble standard de cinquante-deux cartes :

```
class Paquet:

    def __init__(self):
        self.cartes = []
        for couleur in range(4):
            for valeur in range(1, 14):
                carte = Card(couleur, valeur)
                self.cartes.append(carte)
```

La meilleure façon de constituer le paquet est avec une boucle imbriquée. La boucle externe énumère les couleurs de 0 à 3. La boucle interne énumère les valeurs de 1 à 13. Chaque itération crée une nouvelle carte ayant la couleur et la valeur courantes, et l'ajoute à self.cartes.

18-5 - Afficher le paquet

Voici une méthode __str__ pour Paquet :

```
# à l'intérieur de la classe Paquet :

    def __str__(self):
        res = []
        for carte in self.cartes:
            res.append(str(carte))
        return '\n'.join(res)
```

Cette méthode montre un moyen efficace d'accumuler une longue chaîne de caractères : en construisant une liste de chaînes de caractères, puis en utilisant la méthode de chaîne de caractères join. La fonction interne str invoque la méthode __str__ sur chaque carte et renvoie sa représentation sous forme de chaîne de caractères.

Comme nous invoquons join sur un caractère de fin de ligne, les cartes sont séparées par des caractères de fin de ligne. Voici à quoi ressemble le résultat :

```
>>> paquet = Paquet()
>>> print(paquet)
as de trèfle
2 de trèfle
3 de trèfle
...
10 de pique
valet de pique
dame de pique
roi de pique
```

Même si le résultat apparaît sur 52 lignes, c'est une longue chaîne qui contient des caractères de fin de ligne.

18-6 - Ajouter, enlever, mélanger et trier

Pour distribuer des cartes, nous voudrions une méthode qui enlève une carte du paquet et la renvoie. La méthode de liste pop offre un moyen pratique de le faire :

```
# à l'intérieur de la classe Paquet :
```

```
    def pop_carte(self):
        return self.cartes.pop()
```

Comme pop retire la *dernière* carte dans la liste, nous distribuons les cartes à partir de la fin du paquet.

Pour ajouter une carte, nous pouvons utiliser la méthode de liste append :

```
# à l'intérieur de la classe Paquet :

    def ajouter_carte(self, carte):
        self.cartes.append(carte)
```

Une méthode comme celle-ci, qui utilise une autre méthode sans faire beaucoup de travail s'appelle parfois un **placage**. La métaphore vient du travail en bois, où un placage est une mince couche de bois d'essence noble collée à la surface d'une pièce en bois moins cher, pour améliorer l'apparence.

Dans ce cas, ajouter_carte est une méthode « mince » qui exprime une opération de liste en termes appropriés pour les paquets. Elle améliore l'apparence, ou l'interface, de la mise en œuvre.

Nous pouvons également écrire une méthode de Paquet nommée battre en utilisant la fonction shuffle du module random :

```
# à l'intérieur de la classe Paquet :

    def battre(self):
        random.shuffle(self.cartes)
```

N'oubliez pas d'importer random.

À titre d'exercice, écrivez une méthode de Paquet appelée trier, qui utilise la méthode de liste sort pour trier les cartes d'un Paquet. La méthode trier utilise la méthode __lt__ que nous avons définie pour déterminer l'ordre.

18-7 - Héritage

L'héritage est la capacité de définir une nouvelle classe qui est une version modifiée d'une classe existante. À titre d'exemple, disons que nous voulons une classe pour représenter une « main », c'est-à-dire les cartes détenues par un seul joueur. Une main est semblable à un paquet : les deux sont constitués d'une collection de cartes, et les deux nécessitent des opérations comme l'ajout et le retrait de cartes.

En même temps, une main est différente d'un paquet ; il existe des opérations que nous voulons pour les « mains » qui n'ont pas de sens pour un paquet. Par exemple, au poker, nous pourrions comparer deux mains pour voir qui gagne. Au bridge, nous pourrions calculer le nombre de points d'une main afin de faire une enchère.

Cette relation entre classes - similaires, mais différentes - se prête bien à l'héritage. Pour définir une nouvelle classe qui hérite d'une classe existante en Python, vous mettez le nom de la classe existante entre parenthèses :

```
class Main(Paquet):
    """Représente une main au jeu de cartes."""
```

Cette définition indique que Main hérite de Paquet ; cela signifie que nous pouvons utiliser des méthodes comme pop_carte et ajouter_carte tant pour les Mains que pour les Paquets.

Lorsqu'une nouvelle classe hérite d'une classe existante, la classe existante est appelée **classe mère** ou **classe parente** et la nouvelle classe est appelée **classe fille** ou **classe enfant**.

Dans cet exemple, Main hérite __init__ de Paquet, mais celle-ci ne fait pas vraiment ce que nous voulons : au lieu d'alimenter la main avec 52 nouvelles cartes, la méthode init pour Mains doit initialiser cartes à une liste vide.

Si nous fournissons une méthode d'initialisation à la classe Main, elle remplace celle de la classe Paquet :

```
# à l'intérieur de la classe Main :

    def __init__(self, etiquette = ''):
        self.cartes = []
        self.etiquette = etiquette
```

Lorsque vous créez une Main, Python appelle cette méthode init, pas celle de Paquet.

```
>>> main = Main('nouvelle main')
>>> main.cartes
[]
>>> main.etiquette
'nouvelle main'
```

Les autres méthodes sont héritées de Paquet, donc nous pouvons utiliser pop_carte et ajouter_carte pour distribuer une carte :

```
>>> paquet = Paquet()
>>> carte = paquet.pop_carte()
>>> main.add_carte(carte)
>>> print(main)
roi de pique
```

Une prochaine étape naturelle consiste à encapsuler ce code dans une méthode appelée deplacer_cartes :

```
# à l'intérieur de la classe Paquet :

    def deplacer_cartes(self, main, nombre):
        for i in range(nombre):
            main.ajouter_carte(self.pop_carte())
```

deplacer_cartes prend deux arguments, un objet Main et le nombre de cartes à distribuer. Elle modifie tant self (le paquet) que main, et renvoie None.

Dans certains jeux, les cartes sont déplacées d'une main à l'autre, ou remises d'une main vers le paquet. Vous pouvez utiliser deplacer_cartes pour les deux opérations : self peut être soit un Paquet, soit une Main, et main, malgré le nom, peut aussi être un Paquet.

L'héritage est une fonctionnalité utile. Certains programmes qui seraient répétitifs sans héritage peuvent être écrits plus élégamment en l'utilisant. L'héritage peut faciliter la réutilisation du code, puisque vous pouvez personnaliser le comportement des classes parentes sans devoir les modifier. Dans certains cas, la structure de l'héritage reflète la structure naturelle du problème, ce qui rend la conception plus facile à comprendre.

D'un autre côté, l'héritage peut rendre les programmes difficiles à lire. Quand une méthode est invoquée, parfois on ne sait pas trop où trouver sa définition. Le code en question peut être réparti sur plusieurs modules. De plus, beaucoup de choses qui peuvent être faites en utilisant l'héritage peuvent être faites aussi bien ou mieux sans lui.

18-8 - Diagrammes de classes

Jusqu'à présent, nous avons vu des diagrammes de pile, qui montrent l'état d'un programme, et les diagrammes d'objets, qui montrent les attributs d'un objet et leurs valeurs. Ces diagrammes représentent un instantané dans l'exécution d'un programme, donc ils changent pendant l'exécution du programme.

Ils sont aussi très détaillés ; à certaines fins, trop détaillés. Un diagramme de classe est une représentation plus abstraite de la structure d'un programme. Au lieu de montrer des objets individuels, il montre les classes et les relations entre elles.

Il existe plusieurs types de relations entre les classes :

- les objets d'une classe peuvent contenir des références vers des objets d'une autre classe. Par exemple, chaque Rectangle contient une référence vers un Point, et chaque Paquet contient des références vers plusieurs Cartes. Ce type de relation est appelé **HAS-A**, « a-un(e) », comme dans « un Rectangle *a un* Point » ;
- une classe peut hériter d'une autre. Cette relation est appelée **IS-A**, « est-un(e) », comme dans « une Main *est une* sorte de Paquet. » ;
- une classe peut dépendre d'une autre dans le sens où les objets d'une classe prennent comme paramètres des objets de la seconde classe, ou utilisent des objets de la seconde classe dans le cadre d'un calcul. Ce type de relation est appelée une **dépendance**.

Un **diagramme de classes** est une représentation graphique de ces relations. Par exemple, la figure 18.2 montre les relations entre Carte, Paquet et Main.

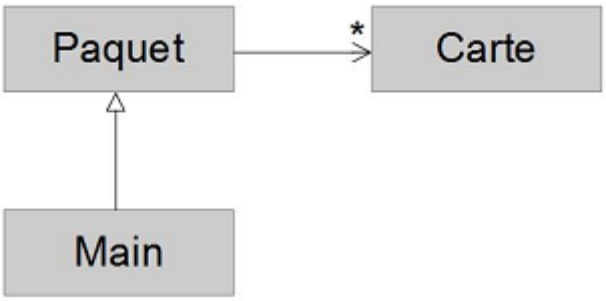


Figure 18.2 : Diagramme de classes.

La flèche à pointe triangulaire creuse représente une relation IS-A ; dans ce cas, elle indique que Main hérite de Paquet.

La flèche à pointe normale représente une relation HAS-A ; dans ce cas, un Paquet a des références vers des objets Carte.

L'astérisque (*) près de la pointe de la flèche est une **multiplicité** ou **cardinalité** ; il indique combien de Cartes a un Paquet. Une multiplicité peut être un simple nombre, comme 52, une plage de valeurs, comme 5..7 ou une étoile, qui indique qu'un Paquet peut avoir un nombre quelconque de Cartes.

Il n'y a aucune dépendance dans ce schéma. Elles seraient normalement représentées par une flèche en pointillé. Ou s'il y a beaucoup de dépendances, elles sont parfois omises.

Un schéma plus détaillé pourrait montrer qu'un Paquet contient en fait une *liste* de Cartes, mais les types internes comme les listes et les dictionnaires ne sont généralement pas inclus dans les diagrammes de classes.

18-9 - Débogage

L'héritage peut rendre le débogage difficile parce que lorsque vous appelez une méthode sur un objet, il peut être difficile de comprendre quelle méthode sera invoquée.

Supposons que vous écriviez une fonction qui travaille avec des objets Main. Vous souhaiterez qu'elle fonctionne avec toutes sortes de Mains, comme MainsAuPoker, MainsAuBridge, etc. Si vous invoquez une méthode comme battre, vous pourriez obtenir celle définie dans Paquet, mais si l'une des sous-classes remplace cette méthode, vous obtiendrez la nouvelle version. Ce comportement est généralement une bonne chose, mais il peut devenir déroutant.

Chaque fois que vous n'êtes pas sûr du flux d'exécution de votre programme, la solution la plus simple consiste à ajouter des instructions d'affichage au début des méthodes pertinentes. Si Paquet.battre affiche un message qui dit quelque chose comme méthode Paquet.battre en cours d'exécution, alors le programme retrace au fur et à mesure le flux de son exécution.

Une autre possibilité serait d'utiliser cette fonction, qui prend un objet et un nom de méthode (sous forme de chaîne de caractères) et renvoie la classe qui fournit la définition de la méthode :

```
def trouver_classe_qui_definit(objet, nom_methode):
    for ty in type(objet).mro():
        if nom_methode in ty.__dict__:
            return ty
```

Voici un exemple :

```
>>> main = Main()
>>> trouver_classe_qui_definit(main, 'battre')
<class '__main__.Paquet'>
```

Donc, la méthode battre pour cette main est celle définie dans Paquet.

trouver_classe_qui_definit utilise la méthode mro pour obtenir la liste des objets classe (types) où rechercher des méthodes. « MRO » signifie *method resolution order* « ordre de résolution des méthodes », qui est la séquence de classes que Python recherche pour « résoudre » un nom de méthode.

Voici une suggestion de conception : lorsque vous substituez une méthode, l'interface de la nouvelle méthode devrait être identique à l'ancienne. Elle devrait prendre les mêmes paramètres, retourner le même type et obéir aux mêmes préconditions et postconditions. Si vous suivez cette règle, vous découvrirez que toute fonction conçue pour travailler avec une instance d'une classe mère, comme un Paquet, va fonctionner également avec des instances des classes filles, comme Main et MainAuPoker.

Si vous ne respectez pas cette règle, qui s'appelle le « principe de substitution de Liskov », votre code va s'effondrer comme (sans jeu de mots) un château de cartes.

18-10 - Encapsulation de données

Les chapitres précédents montrent un modèle de développement que nous pourrions appeler « conception orientée objet ». Nous avons identifié les objets dont nous avions besoin - comme Point, Rectangle et Temps - et défini des classes pour les représenter. Dans chaque cas, il y a une correspondance évidente entre l'objet et une entité dans le monde réel (ou du moins un monde mathématique).

Mais, parfois, il est moins évident de déterminer quels sont les objets dont vous avez besoin et comment ils doivent interagir. Dans ce cas, vous avez besoin d'un modèle de développement différent. De la même manière que nous avons découvert des interfaces de fonction par encapsulation et généralisation, nous pouvons découvrir des interfaces de classe par **encapsulation de données**.

L'analyse de Markov, de la section **13.8**, fournit un bon exemple. Si vous téléchargez mon code à partir de l'adresse **markov.py**, vous verrez qu'il utilise deux variables globales - suffix_map et prefix - qui sont lues et écrites par plusieurs fonctions.

```
suffix_map = {}
prefix = ()
```

Comme ces variables sont globales, nous ne pouvons exécuter qu'une seule analyse à la fois. Si nous lisons deux textes, leurs préfixes et suffixes seront ajoutés aux mêmes structures de données (ce qui peut conduire à la génération de textes quelque peu éclectiques).

Pour exécuter plusieurs analyses et les garder séparées, nous pouvons encapsuler l'état de chaque analyse dans un objet. Voilà à quoi cela ressemble :

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Ensuite, nous transformons les fonctions en méthodes. Par exemple, voici la méthode process_word :

```
def process_word(self, word, order = 2):
    if len(self.prefix) < order:
        self.prefix += (word,)
        return

    try:
        self.suffix_map[self.prefix].append(word)
    except KeyError:
        # s'il n'existe aucune entrée pour ce préfixe, en créer une
        self.suffix_map[self.prefix] = [word]

    self.prefix = shift(self.prefix, word)
```

Transformer un programme de cette façon - modifier la conception sans modifier le comportement - est un autre exemple de réusinage (voir la section **4.7**).

Cet exemple suggère un modèle de développement pour la conception des objets et méthodes :

1 Commencez par écrire des fonctions qui lisent et écrivent des variables globales (si nécessaire) ;
2 Une fois que votre programme fonctionne, recherchez des associations entre les variables globales et les fonctions qui les utilisent ;
3 Encapsulez les variables apparentées dans des attributs d'un objet ;
4 Transformez les fonctions associées en méthodes de la nouvelle classe.

À titre d'exercice, téléchargez mon code Markov à l'adresse **markov.py** et suivez les étapes décrites ci-dessus pour encapsuler les variables globales comme attributs d'une nouvelle classe appelée Markov. Solution : **Markov.py** (remarquez la M capitale).

18-11 - Glossaire

- **encoder :** représenter un ensemble de valeurs en utilisant un autre ensemble de valeurs en construisant une correspondance entre eux.
- **attribut de classe :** un attribut associé à un objet classe. Les attributs de classe sont définis dans une définition de classe, mais en dehors de toute méthode.
- **attribut d'instance :** un attribut associé à une instance d'une classe.
- **placage :** un procédé ou une fonction qui fournit une interface à une autre fonction sans faire beaucoup de calculs.
- **héritage :** la possibilité de définir une nouvelle classe qui est une version modifiée d'une classe définie préalablement.
- **classe mère :** la classe dont hérite une classe fille ou enfant.
- **classe fille :** une nouvelle classe créée en héritant d'une classe existante ; également appelée » classe enfant » ou « sous-classe ».
- **relation IS-A :** une relation entre une classe fille et sa classe mère.
- **relation HAS-A :** une relation entre deux classes dans laquelle les instances d'une classe contiennent des références aux instances de l'autre.
- **dépendance :** une relation entre deux classes où les instances d'une classe utilisent les instances de l'autre classe, mais ne les stockent pas comme attributs.
- **diagramme de classes :** un diagramme qui montre les classes d'un programme et les relations entre elles.

- **multiplicité** ou **cardinalité :** une notation dans un diagramme de classes qui montre, pour une relation HAS-A, combien il y a de références à des instances d'une autre classe.
- **encapsulation de données :** un modèle de développement d'un programme qui implique au départ un prototype utilisant des variables globales et une version finale qui transforme les variables globales en attributs d'instance.

18-12 - Exercices

Exercice 1

Pour le programme suivant, dessinez un diagramme de classes UML qui montre ces classes et les relations entre elles.

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Exercice 2

Écrivez une méthode de Paquet appelée `distribue_mains` *qui prend deux paramètres, le nombre de mains à distribuer et le nombre de cartes par main. Elle doit créer le nombre voulu d'objets Main, distribuer le nombre approprié de cartes par main et renvoyer une liste de Mains.*

Exercice 3

La liste suivante reprend les combinaisons possibles au poker, par ordre croissant de la valeur et ordre décroissant de la probabilité :

- **Paire :** *deux cartes ayant la même valeur ;*
- **Double paire :** *deux paires de cartes ayant la même valeur ;*
- **Brelan :** *trois cartes ayant la même valeur ;*
- **Suite ou quinte :** *cinq cartes ayant les valeurs dans l'ordre (l'as peut être en première ou en dernière position, donc as-2-3-4-5 ou 10-valet-dame-roi-as sont des suites valides, mais dame-roi-as-2-3 ne l'est pas.) ;*
- **Couleur :** *cinq cartes ayant la même couleur ;*
- **Full ou main pleine :** *trois cartes d'une valeur, deux cartes d'une autre ;*
- **Carré :** *quatre cartes de même valeur ;*
- **Quinte flush ou suite couleur :** *cinq cartes dans l'ordre (tel que défini ci-dessus) ayant la même couleur.*

Le but de ces exercices est d'estimer la probabilité de tirer ces différentes combinaisons.

1. *Téléchargez les fichiers suivants à l'adresse* **http://allen-downey.developpez.com/livres/python/pensez-python/fichiers/** *:*

   - `Card.py` *: une version complète des classes Carte, Paquet et Main de ce chapitre.*
   - `PokerHand.py` *: une implémentation incomplète d'une classe qui représente une main au poker, et un code qui la teste.*

2. *Si vous exécutez* `PokerHand.py` *, elle distribue sept mains de 7 cartes chacune et les contrôle pour voir si l'une d'elles contient une quinte flush. Lisez attentivement ce code avant de poursuivre.*
3. *Ajoutez à* `PokerHand.py` *des méthodes nommées* `has_pair` *,* `has_twopair` *, etc. qui renvoient* `True` *ou* `False` *selon que la main satisfait ou non aux critères pertinents. Votre code devrait fonctionner correctement pour des « mains » qui contiennent un nombre quelconque de cartes (bien que 5 et 7 soient les formats les plus courants).*
4. *Écrivez une méthode nommée* `classify` *, qui calcule la combinaison de la plus haute valeur pour une main, et définit l'attribut* `label` *en conséquence. Par exemple, une main de 7 cartes peut contenir une quinte flush et une paire ; elle doit être étiquetée « quinte flush ».*
5. *Lorsque vous êtes convaincu que vos méthodes de classification fonctionnent, l'étape suivante consiste à estimer les probabilités des différentes mains. Écrivez une fonction dans* `PokerHand.py` *qui bat un paquet de cartes, le divise en mains, classifie les mains et compte le nombre de fois où les différentes combinaisons apparaissent.*
6. *Affichez une table des combinaisons et leurs probabilités. Exécutez votre programme avec des nombres de plus en plus grands de mains jusqu'à ce que les valeurs de sortie convergent vers un degré raisonnable de précision. Comparez vos résultats aux valeurs théoriques du tableau ci-après :*

Solution : **PokerHandSoln.py**.

Probabilités en % des différentes combinaisons du poker (mains de 5 et 7 cartes)			**Combinaison**	**Main de 5 cartes**	**Main de 7 cartes**
Quinte flush ou **suite couleur**	0,00154 %	0.0311 %			
Carré	0,0240 %	0,168 %			
Full ou **main pleine**	0,144 %	2,60 %			
Couleur	0,196 %	3,03 %			
Suite ou **quinte**	0,392 %	4,62 %			
Brelan	2,11 %	4,83 %			
Double paire	4,75 %	23,5 %			
Paire	42,3 %	43,8 %			

Source : **https://en.wikipedia.org/wiki/List_of_poker_hands** .

19 - Les bonus

L'un de mes objectifs dans ce livre a été de vous en apprendre aussi peu que possible sur Python. Lorsqu'il y avait deux façons différentes de faire quelque chose, j'en ai choisi une et évité de parler de l'autre. Ou alors, parfois, j'ai mentionné la seconde dans un exercice.

Je voudrais maintenant revenir à quelques-unes des pépites que j'ai laissées de côté. Python offre un bon nombre de fonctionnalités qui ne sont pas réellement nécessaires - vous pouvez écrire de bons programmes sans elles -, mais elles peuvent parfois vous permettre d'écrire du code plus concis, ou plus lisible, ou plus efficace, et parfois même les trois en même temps.

19-1 - Expressions conditionnelles

Nous avons étudié les instructions conditionnelles au chapitre **5.4**. Ces conditions sont souvent utilisées pour choisir entre deux valeurs. Par exemple :

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

Ce fragment de code vérifie si x est un nombre positif. Si c'est le cas, il calcule le logarithme de ce nombre. Sinon, la fonction math.log générerait une exception entraînant l'arrêt du programme. Pour éviter cet arrêt intempestif du programme, nous générons un « NaN » (*Not a Number*), une valeur spéciale en virgule flottante qui représente autre chose qu'un nombre.

Nous pouvons écrire ces instructions de façon plus concise en utilisant une **expression conditionnelle** :

```
y = math.log(x) if x > 0 else float('nan')
```

Que l'on peut lire comme suit : « y prend la valeur log(x) si x est plus grand que 0 ; sinon, ce n'est pas un nombre, il prend la valeur spéciale NaN. »

Les fonctions récursives se prêtent parfois à l'utilisation d'expressions conditionnelles. Par exemple, voici une version récursive de la fonction factorielle :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n-1)
```

Nous pouvons la réécrire comme suit :

```
def factorielle(n):
    return 1 if n == 0 else n * factorielle(n-1)
```

Une autre utilisation des expressions conditionnelles est la gestion des arguments optionnels. Par exemple, voici la méthode init de la solution GoodKangaroo (voir exercice **2** du chapitre 17) :

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
        self.pouch_contents = contents
```

Nous pouvons la réécrire ainsi :

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

D'une façon générale, vous pouvez remplacer une instruction conditionnelle par une expression conditionnelle si les deux branches contiennent des expressions simples qui sont soit renvoyées à une fonction appelante, soit affectées à la même variable.

19-2 - Les listes en compréhension

Nous avons abordé dans la section **10.7** les mécanismes de mappage et de filtration. Par exemple, la fonction suivante prend en entrée une liste de chaînes de caractères, applique la méthode de chaîne capitalize à chacun des éléments et renvoie une nouvelle liste de chaînes :

```
def mettre_tout_en_capitales(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

Nous pouvons réécrire cette fonction de façon plus concise en utilisant une **liste en compréhension** :

```
def mettre_tout_en_capitales(t):
    return [s.capitalize() for s in t]
```

Les opérateurs crochets indiquent que nous construisons une nouvelle liste. L'expression à l'intérieur des crochets spécifie les éléments de la liste, et la clause for indique quelle séquence nous parcourons.

La syntaxe d'une liste en compréhension est un peu étrange parce que la variable de boucle, s dans notre exemple, apparaît dans l'expression (s.capitalize) avant qu'elle ne soit définie (for s in t).

Les listes en compréhension s'appellent parfois aussi *listes en intension* (avec un *s* et non un *t*, car le mot, issu de la philosophie et des mathématiques, s'oppose ici à *extension*).

Les listes en compréhension permettent également de filtrer des données. Par exemple, cette fonction sélectionne les éléments de t qui sont en lettres capitales et renvoie la nouvelle liste :

```
def capitales_seulement(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

Nous pouvons la réécrire en utilisant une liste en compréhension :

```
def capitales_seulement(t):
    return [s for s in t if s.isupper()]
```

Les listes en compréhension sont concises et faciles à lire, tout au moins pour les expressions simples. Et elles sont généralement plus rapides que des boucles for équivalentes, parfois beaucoup plus rapides.

Donc, si vous m'en voulez de ne pas les avoir mentionnées plus tôt, je vous comprends. Pour ma défense, je signalerais cependant que les listes en compréhension sont plus difficiles à déboguer parce que vous ne pouvez pas ajouter une instruction d'impression à l'intérieur de la boucle. Et je vous recommanderais de ne les utiliser que si le calcul est suffisamment simple pour que vous ayez une bonne chance de l'écrire correctement du premier coup. Ce qui, pour de purs débutants, signifie jamais.

19-3 - Les générateurs

Les **générateurs** ressemblent aux listes en compréhension, mais avec des parenthèses à la place des crochets :

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

Le résultat est un objet générateur qui sait comment itérer sur une séquence de valeurs. Mais, à la différence d'une liste en compréhension, il ne s'empresse pas de calculer toutes les valeurs immédiatement : il attend qu'on lui demande d'en calculer une. La fonction interne next obtient et renvoie la valeur suivante du générateur :

```
>>> next(g)
0
>>> next(g)
1
```

Quand vous arrivez à la fin de la séquence, next signale une exception StoptIteration. Vous pouvez également utiliser une boucle for pour itérer sur les valeurs :

```
>>> for valeur in g:
... print(valeur)
4
9
16
```

Le générateur sait où il était arrivé dans la séquence, c'est pourquoi la boucle for ci-dessus a repris là où le dernier next était arrivé. La boucle for s'arrête silencieusement à la fin de la séquence, mais, si on l'appelle une nouvelle fois, il générera à nouveau une exception :

```
>>> next(g)
StopIteration
```

Les générateurs sont fréquemment utilisés avec des fonctions comme sum, max et min :

```
>>> sum(x**2 for x in range(5))
30
```

19-4 - Les fonctions any et all

La fonction interne any de Python prend en entrée une série de valeurs booléennes et renvoie True si l'une au moins des valeurs est vraie. Cela fonctionne sur des listes :

```
>>> any([False, False, True])
True
```

Mais on s'en sert souvent avec des générateurs :

```
>>> any(lettre == 't' for lettre in 'monty')
True
```

Cet exemple ne paraît pas très utile, car il fait la même chose que l'opérateur in. Mais nous pourrions utiliser any pour réécrire certaines des fonctions de recherche que nous avons écrites dans la section **9.3**.

Par exemple, la fonction evite :

```
def evite(mot, interdites):
    for lettre in mot:
        if lettre in interdites:
            return False
    return True
```

peut se réécrire comme suit :

```
def evite(mot, interdites):
    return not any(lettre in interdites for lettre in mot)
```

Ce qui se lit en presque bon français : « mot évite interdites s'il n'y a pas any lettre interdite dans mot ».

Utiliser any avec un générateur est efficace parce que fonction s'arrête (sans aller jusqu'au bout) dès qu'elle a trouvé une valeur satisfaisant la condition recherchée.

Il y a également une fonction interne, all, qui renvoie True si tous les éléments de la séquence sont vrais. À titre d'exercice, utilisez all pour réécrire la fonction utilise_toutes de la section **9.3**.

19-5 - Les ensembles (sets)

Dans la section **13.6**, j'ai utilisé des dictionnaires pour trouver des mots figurant dans un document, mais pas dans une liste de mots. La fonction soustraire utilisait d1, contenant les mots du document sous la forme de clés, et d2, contenant la liste de mots. Elle renvoyait un dictionnaire contenant les clés de d1 ne figurant pas dans d2.

```
def soustraire(d1, d2):
    resultat = dict()
    for clef in d1:
        if clef not in d2:
            resultat[clef] = None
    return resultat
```

Dans chacun de ces trois dictionnaires, les valeurs associées à toutes les clés étaient None, car nous n'utilisions jamais ces valeurs. Ceci a pour effet de gaspiller l'espace mémoire.

Il existe en Python un autre type de donnée interne, l'ensemble (set), qui fonctionne comme une collection de clés d'un dictionnaire sans valeurs. Il est rapide d'ajouter un élément à un ensemble, de même que de vérifier l'appartenance d'un élément à un ensemble. Et les ensembles fournissent des méthodes et des opérateurs pour effectuer les opérations ensemblistes communes.

Par exemple, la soustraction d'ensembles réalisée ci-dessus peut se faire grâce à une méthode de différence ensembliste, difference, ou à l'opérateur - entre deux ensembles. Nous pouvons donc réécrire soustraire comme suit :

```
def soustraire(d1, d2):
    return set(d1) - set(d2)
```

Le résultat est un ensemble au lieu d'un dictionnaire, mais pour les opérations qui nous intéressent, comme l'itération ou le test d'appartenance, le comportement est identique.

Quelques-uns des exercices de ce livre peuvent se faire de façon concise et efficace en utilisant des ensembles. Voici par exemple une solution utilisant un dictionnaire de l'exercice **7** du chapitre 10 :

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

Quand un élément apparaît pour la première fois, il est ajouté au dictionnaire. Si le même élément se présente à nouveau, la fonction renvoie True. À la fin, si aucun doublon n'a été rencontré, la fonction renvoie False.

En utilisant les ensembles, nous pouvons réécrire cette fonction ainsi :

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

Dans un ensemble, un élément ne peut être présent qu'une seule fois. Par conséquent, si un élément existe plusieurs fois dans t, l'ensemble sera plus petit que la liste d'origine. Inversement, s'il n'y a pas de doublon, l'ensemble et la liste d'origine auront la même taille.

Nous pouvons aussi utiliser des ensembles pour résoudre certains des exercices du chapitre **9**. Par exemple, voici notre version de utilise_uniquement avec une boucle :

```
def utilise_uniquement(mot, disponibles):
    for lettre in mot:
        if lettre not in disponibles:
            return False
    return True
```

La fonction utilise_uniquement vérifie si toutes les lettres du mot sont disponibles. Nous pouvons la réécrire en employant un ensemble :

```
def utilise_uniquement(mot, disponibles):
    return set(mot) <= set(disponibles)
```

L'opérateur <=, qui s'écrirait mathématiquement # (est inclus dans ou est égal), vérifie si un ensemble est un sous-ensemble d'un autre (ou est égal à l'autre), ce qui se vérifie si toutes les lettres de mot appartiennent à disponibles.

À titre d'exercice, réécrivez la fonction evite du chapitre **9** en employant des ensembles.

19-6 - Les compteurs (Counter)

Un compteur (Counter) ressemble à un ensemble, sauf que si un élément est ajouté plus d'une fois, il n'est pas dédoublonné, mais le compteur enregistre le nombre de ses occurrences. Si le concept mathématique de **multiensemble** (parfois également appelé « sac ») vous est familier, alors sachez qu'un compteur est une façon naturelle de représenter un multiensemble.

Les compteurs ne sont pas un type de donnée interne de Python, mais sont définis dans un module standard nommé collections, que vous devez donc importer pour pouvoir utiliser les compteurs. Vous pouvez initialiser un compteur avec une chaîne de caractères, une liste ou toute autre chose supportant l'itération :

```
>>> from collections import Counter
>>> decompte = Counter('perroquet')
>>> decompte
Counter({'e': 2, 'r': 2, 'o': 1, 'q': 1, 'p': 1, 'u': 1, 't': 1})
```

Les compteurs se comportent à bien des égards comme des dictionnaires. Ils établissent une correspondance entre chaque clé et le nombre d'occurrences de cette clé. Comme les dictionnaires, il faut que les clés soient hachables.

À la différence des dictionnaires, les compteurs ne déclenchent aucune exception si on tente d'accéder à un élément qui ne s'y trouve pas : dans ce cas, ils renvoient simplement 0 :

```
>>> decompte['d']
0
```

Nous pouvons utiliser des compteurs pour réécrire la fonction is_anagram de l'exercice **6** du chapitre 10 :

```
def is_anagram(mot1, mot2):
    return Counter(mot1) == Counter(mot2)
```

Si deux mots sont des anagrammes, alors ils contiennent les mêmes lettres et chaque lettre le même nombre de fois, si bien que leurs compteurs sont équivalents.

Les compteurs fournissent des méthodes et opérateurs pour effectuer des opérations de type ensembliste, en particulier l'addition, la soustraction, l'union et l'intersection. Une méthode souvent utile est most_common, qui renvoie une liste de paires valeur-fréquence, triées de la plus fréquente à la plus rare :

```
>>> decompte = Counter('perroquet')
>>> for val, freq in decompte.most_common(3):
...     print(val, freq)
...
e 2
r 2
o 1
```

Le nombre passé en entrée à la fonction most_common (ici, 3) indique le nombre maximal de paires clé-valeur à parcourir. Si l'argument est manquant, la fonction most_common itérera sur l'ensemble des paires :

```
>>> for val, freq in count.most_common():
...     print(val, freq)
...
e 2
r 2
o 1
q 1
p 1
u 1
t 1
```

19-7 - Dictionnaire de type defaultdict

Le module collections propose aussi la fonctionnalité defaultdict, qui ressemble à un dictionnaire si ce n'est que, si vous accédez à une clé qui n'existe pas, il peut générer une nouvelle valeur à la volée.

Quand vous créez un dictionnaire defaultdict, vous fournissez une fonction qui est utilisée pour créer de nouvelles valeurs. Une fonction utilisée pour créer des objets s'appelle parfois une **usine** (*factory*). Les fonctions internes de création de listes, d'ensembles et d'autres types peuvent servir d'usines :

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notez que l'argument est list, qui est un objet de classe, et non list() qui serait une nouvelle liste. La fonction que vous fournissez n'est pas appelée tant que vous ne cherchez pas à accéder à une clé qui n'existe pas.

```
>>> t = d['nouvelle clé']
>>> t
[]
```

La nouvelle liste, que nous appelons ici t, est également ajoutée au dictionnaire. Il en résulte que si nous modifions t, le changement se répercutera dans le dictionnaire d :

```
>>> t.append('nouvelle valeur')
>>> d
defaultdict(<class 'list'>, {'nouvelle clé': ['nouvelle valeur']})
```

Si vous constituez un dictionnaire de listes, vous pouvez souvent écrire du code plus simple en utilisant des dictionnaires de type defaultdict. Dans ma solution à l'exercice **2** du chapitre 12, que vous pouvez télécharger à l'adresse **anagram_sets.py**, j'ai construit un dictionnaire établissant une correspondance entre une chaîne de caractères triés par ordre alphabétique et une liste de mots pouvant être écrits avec ces lettres. Par exemple, la chaîne de caractères 'ADEIR' pointe sur la liste de mots ['AIDER', 'ARIDE', 'RADIE', 'RAIDE']. Voici le code d'origine :

```
def all_anagrams(nomfichier):
    d = {}
```

```
    for ligne in open(nomfichier):
        mot = ligne.strip().lower()
        t = signature(mot)
        if t not in d:
            d[t] = [mot]
        else:
            d[t].append(mot)
    return d
```

Il est possible de simplifier cette fonction avec la fonctionnalité setdefault que vous avez peut-être utilisée dans l'exercice **2** du chapitre 11 :

```
def all_anagrams(nomfichier):
    d = {}
    for ligne in open(nomfichier):
        mot = ligne.strip().lower()
        t = signature(mot)
        d.setdefault(t, []).append(mot)
    return d
```

Cette solution présente cependant l'inconvénient de créer une nouvelle liste à chaque fois, même si elle n'est pas nécessaire. Pour des listes, ce n'est pas trop gênant. Mais si la fonction usine est complexe, ça peut le devenir.

Nous pouvons éviter cet écueil et simplifier le code en utilisant un defaultdict :

```
def all_anagrams(nomfichier):
    d = defaultdict(list)
    for ligne in open(nomfichier):
        mot = ligne.strip().lower()
        t = signature(mot)
        d[t].append(mot)
    return d
```

Ma solution à l'exercice **3** du chapitre 18, que vous pouvez télécharger à l'adresse **PokerHandSoln.py** utilise setdefault dans la fonction has_straightflush. Cette solution a le désavantage de créer un objet de type Hand à chaque itération dans la boucle, qu'il soit utile ou non. À titre d'exercice, réécrivez cette fonction en utilisant un defaultdict.

19-8 - Tuples nommés

Beaucoup d'objets simples sont essentiellement des collections de valeurs apparentées. Par exemple, l'objet Point défini au chapitre **15** contient deux nombres, x et y. Quand vous définissez une classe de ce genre, vous commencez souvent avec une méthode init et une méthode str :

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

Cela représente beaucoup de code pour ne pas dire grand-chose. Il existe en Python une façon plus concise de dire la même chose :

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

Le premier argument est le nom de la classe que vous désirez créer et le second une liste d'attributs dont les objets Point ont besoin, sous la forme de chaînes de caractères. La valeur de retour de la fonction namedtuple est un objet classe :

```
>>> Point
<class '__main__.Point'>
```

Point possède automatiquement des méthodes comme __init__ et __str__, si bien que vous n'avez pas besoin de les écrire.

Pour créer un objet Point, il suffit d'utiliser la classe Point comme une fonction :

```
>>> p = Point(1, 2)
>>> p
Point(x = 1, y = 2)
```

La méthode init affecte les arguments aux attributs en utilisant les noms que vous avez fournis. La méthode str affiche une représentation de l'objet de type Point et de ses attributs.

Vous pouvez également accéder aux éléments du tuple nommé par leur nom :

```
>>> p.x, p.y
(1, 2)
```

Mais vous pouvez aussi manipuler un tuple nommé comme un tuple :

```
>>> p[0], p[1]
(1, 2)
>>> x, y = p
>>> x, y
(1, 2)
```

Les tuples nommés fournissent une façon rapide de définir des classes simples. Le problème est que les classes simples ne restent pas toujours simples. Il se peut que vous vouliez ultérieurement ajouter des méthodes à un tuple nommé. Dans ce cas, vous pourriez définir une nouvelle classe héritant du tuple nommé :

```
class Point_plus_riche(Point):
# ajoutez de nouvelles méthodes ici
```

Ou alors vous pourriez décider de passer à une définition de classe conventionnelle.

19-9 - Assembler des arguments avec mot-clé

Nous avons vu à la section **12.4** comment écrire une fonction qui assemble ses arguments dans un tuple :

```
def affiche_tout(*args):
    print(args)
```

Vous pouvez appeler cette fonction avec un nombre quelconque d'arguments positionnels (c'est-à-dire d'arguments n'utilisant pas de mots-clés).

```
>>> affiche_tout(1, 2.0, '3')
(1, 2.0, '3')
```

Mais l'opérateur * ne peut pas assembler des arguments dotés de mots-clés :

```
>>> affiche_tout(1, 2.0, troisieme='3')
TypeError: printall() got an unexpected keyword argument 'troisieme'
```

Pour assembler des arguments avec mots-clés, vous pouvez utiliser l'opérateur ** :

```
def affiche_tout(*args, **kwargs):
    print(args, kwargs)
```

Vous pouvez choisir le nom qu'il vous plaira pour le paramètre d'assemblage des arguments avec mots-clés, mais il est usuel de l'appeler kwargs. Le résultat est un dictionnaire établissant une correspondance entre les mots-clés et les valeurs.

```
>>> affiche_tout(1, 2.0, troisieme='3')
(1, 2.0) {'troisieme': '3'}
```

Si vous avez un dictionnaire de mots-clés et de valeurs, vous pouvez utiliser l'opérateur de dispersion ** pour appeler une fonction :

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Sans cet opérateur, la fonction traiterait d comme un argument positionnel unique, et affecterait donc d à x, et se plaindrait ensuite qu'il n'y ait rien à affecter à y :

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

Quand vous utilisez des fonctions qui ont un nombre élevé de paramètres, il est souvent utile de créer et de passer en paramètre des dictionnaires qui spécifient les options fréquemment utilisées.

19-10 - Glossaire

- **expression conditionnelle** : une expression qui peut prendre une valeur parmi deux possibles, en fonction d'une condition.
- **liste en compréhension** : une expression avec une boucle for entre crochets qui produit une nouvelle liste.
- **générateur** : une expression avec une boucle for entre parenthèses qui produit un objet générateur.
- **multiensemble** : une entité mathématique qui établit une correspondance entre les éléments d'un ensemble et le nombre d'occurrences de chacun de ces éléments.
- **usine** : une fonction, habituellement passée en paramètre, utilisée pour créer des objets.

19-11 - Exercices

Exercice 1

La fonction suivante calcule récursivement les coefficients binomiaux :

```
def coeff_binomial(n, k):
    """Calcule le coefficient binomial "k parmi n".
    n: nombre d'essais
    k: nombre de succès
    renvoie : int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0
    resultat = coeff_binomial(n-1, k) + coeff_binomial(n-1, k-1)
    return resultat
```

Réécrivez le corps de la fonction en utilisant des expressions conditionnelles imbriquées.

Remarque : cette fonction n'est pas très efficace parce qu'elle finit par recalculer encore et encore les mêmes valeurs. Vous pourriez la rendre plus efficace en la mémoïsant (mise en cache de ces valeurs, voir section 11.6). Mais vous découvrirez sans doute qu'elle est plus difficile à mémoïser si vous l'écrivez avec des expressions conditionnelles.

A - Débogage

Lorsque vous déboguez, vous devez faire la distinction entre les différents types d'erreurs afin de les trouver plus rapidement.

- Les erreurs de syntaxe sont découvertes par l'interpréteur lorsqu'il traduit le code source en code binaire. Elles indiquent qu'il y a un souci dans la structure du programme. Exemple : l'omission du caractère deux-points à la fin d'une instruction def génère le message quelque peu redondant SyntaxError: invalid syntax.
- Les erreurs d'exécution sont produites par l'interpréteur si quelque chose se passe mal pendant l'exécution du programme. La plupart des messages d'erreur d'exécution comprennent des informations sur l'endroit où l'erreur est survenue et quelles fonctions étaient exécutées. Exemple : une récursion infinie provoque finalement l'erreur d'exécution « maximum recursion depth exceeded » (profondeur de récursivité maximale dépassée).
- Les erreurs sémantiques sont des problèmes avec un programme qui s'exécute sans produire des messages d'erreur, mais qui ne fait pas ce qu'il devrait. Exemple : une expression peut ne pas être évaluée dans l'ordre que vous attendiez, ce qui donne un résultat incorrect.

La première étape dans le débogage est de déterminer de quel type d'erreur il s'agit. Bien que les sections suivantes soient organisées par type d'erreurs, certaines techniques sont applicables à plusieurs situations.

A-1 - Erreurs de syntaxe

Les erreurs de syntaxe sont généralement faciles à corriger une fois que vous avez compris leur nature. Malheureusement, les messages d'erreur ne sont souvent pas très explicites. Les messages les plus courants sont SyntaxError: invalid syntax et SyntaxError: invalid token, aucun des deux n'offrant assez d'informations.

D'un autre côté, le message vous dit à quel endroit du programme est survenu le problème. En fait, il vous indique l'endroit où Python a remarqué un problème, qui n'est pas nécessairement l'endroit où se trouve l'erreur. Parfois, l'erreur se trouve dans le code précédant l'emplacement indiqué par le message d'erreur, souvent sur la ligne précédente.

Si vous construisez le programme de façon incrémentielle, vous devriez avoir une bonne idée de l'endroit où est l'erreur. Elle sera sur la ou les dernière(s) ligne(s) que vous avez ajoutée(s).

Si vous copiez le code à partir d'un livre, commencez par comparer très attentivement votre code à celui du livre. Vérifiez chaque caractère. En même temps, rappelez-vous que le livre peut contenir des erreurs, donc si vous voyez quelque chose qui ressemble à une erreur de syntaxe, c'est peut-être le cas.

Voici quelques façons d'éviter les erreurs de syntaxe les plus courantes.

1. Assurez-vous que vous n'utilisez pas un mot-clé de Python comme nom de variable.
2. Vérifiez qu'il y a un caractère deux-points à la fin de l'en-tête de chaque instruction composée, notamment les instructions for, while, if et def.
3. Assurez-vous que toutes les chaînes de caractères dans le code sont entre guillemets appariés. Assurez-vous que tous les guillemets sont « droits », pas des virgules « culbutées » ou des guillemets « français » en forme de chevrons.
4. Si vous avez des chaînes de caractères multilignes entre guillemets triples (simples ou doubles), assurez-vous que vous avez terminé correctement la chaîne de caractères. Une chaîne non terminée peut provoquer une erreur invalid token (symbole invalide) à la fin de votre programme, ou le code qui la suit peut être traité comme une chaîne de caractères jusqu'aux guillemets de la chaîne suivante. Dans le second cas, cela pourrait ne produire aucun message d'erreur !

5 Un opérateur ouvert et non fermé, comme (, {, ou [, laisse Python continuer à la ligne suivante comme faisant partie de l'instruction en cours. Généralement, une erreur se produit presque immédiatement dans la ligne suivante.
6 Vérifiez si vous n'avez pas fait l'erreur classique d'utiliser = au lieu de == à l'intérieur d'une instruction conditionnelle.
7 Vérifiez l'indentation pour vous assurer qu'elle s'aligne comme il faut. Python peut gérer des espaces et tabulations, mais si vous les mélangez, cela peut causer des problèmes. La meilleure façon d'éviter ce problème est d'utiliser un éditeur de texte qui connaît Python et génère une indentation cohérente.
8 Si vous avez des caractères non ASCII dans le code (y compris dans les chaînes de caractères et les commentaires), cela peut causer un problème, même si Python 3 gère habituellement les caractères non ASCII. Soyez prudent si vous collez du texte provenant d'une page web ou d'une autre source.

Si rien de tout cela ne fonctionne, passez à la section suivante…

A-1-1 - Je n'arrête pas de modifier et rien ne change

Si l'interpréteur dit qu'il y a une erreur et vous ne la voyez pas, il se peut que vous et l'interpréteur n'analysiez pas le même code. Vérifiez votre environnement de programmation pour vous assurer que le programme que vous éditez est celui que Python essaie d'exécuter.

Si vous n'êtes pas sûr, essayez de mettre une erreur de syntaxe évidente et délibérée au début du programme. Maintenant, lancez à nouveau. Si l'interpréteur ne trouve pas la nouvelle erreur, vous n'exécutez pas le nouveau code.

Il y a quelques causes possibles.

- Vous avez modifié le fichier et oublié d'enregistrer les modifications avant de l'exécuter à nouveau. Certains environnements de programmation font cela pour vous, mais tous ne le font pas.
- Vous avez changé le nom du fichier, mais vous exécutez toujours l'ancien nom.
- Quelque chose dans votre environnement de développement est configuré de manière incorrecte.
- Si vous écrivez un module et utilisez import, assurez-vous de ne pas donner à votre module le nom d'un des modules standard de Python.
- Si vous utilisez import pour lire un module, rappelez-vous que vous devez redémarrer l'interpréteur ou utiliser reload pour lire un fichier modifié. Si vous importez à nouveau le module, il ne fait rien.

Si vous vous retrouvez coincé et vous ne pouvez pas comprendre ce qui se passe, une approche consiste à recommencer avec un nouveau programme comme « Hello World ! », et vous assurer que vous pouvez exécuter un programme connu. Puis, ajoutez progressivement les morceaux du programme original au nouveau programme.

A-2 - Erreurs d'exécution

Une fois que votre programme est syntaxiquement correct, Python peut le lire et au moins commencer l'exécution. Qu'est-ce qui pourrait mal tourner ?

A-2-1 - Mon programme ne fait absolument rien

Ce problème est plus courant lorsque votre fichier est constitué de fonctions et de classes, mais en fait aucune fonction n'est invoquée pour démarrer l'exécution. Cela peut être intentionnel si vous prévoyez juste d'importer ce module pour fournir des classes et des fonctions.

Si cela n'est pas intentionnel, assurez-vous qu'il existe un appel de fonction dans le programme, et assurez-vous que le flux d'exécution l'atteint (voir « Flux d'exécution » ci-dessous).

A-2-2 - Mon programme s'arrête et ne fait plus rien

Si un programme s'arrête et semble ne rien faire, il est « suspendu ». Souvent, cela signifie qu'il est pris dans une boucle infinie ou une récursion infinie.

- Si vous soupçonnez qu'une boucle particulière de votre programme est peut-être à l'origine du problème, ajoutez immédiatement avant la boucle une instruction d'affichage disant « entrer dans la boucle » et une autre immédiatement après, disant « sortir de la boucle ». Exécutez le programme. Si vous voyez le premier message et pas le second, vous avez une boucle infinie. Allez à la section « Boucle infinie » ci-dessous.
- Presque toujours, une récursion infinie va laisser le programme s'exécuter pendant un certain temps, puis une erreur « RuntimeError: Maximum recursion depth exceeded » se produira. Si cela arrive, allez à la section « Boucle infinie » ci-dessous. Si vous n'avez pas cette erreur, mais pensez qu'il y a un problème avec une méthode ou une fonction récursive, vous pouvez toujours utiliser les techniques dans la section « Récursion infinie ».
- Si aucune de ces étapes ne fonctionne, commencez à tester d'autres boucles et d'autres fonctions et méthodes récursives.
- Si cela ne fonctionne pas, alors il est possible que vous ne compreniez pas le flux d'exécution de votre programme. Allez à la section « Flux d'exécution » ci-dessous.

Boucle infinie

Si vous pensez que vous avez une boucle infinie et vous pensez que vous savez quelle boucle est la cause du problème, ajoutez à la fin de la boucle une instruction print qui affiche les valeurs des variables dans la condition et la valeur de la condition.

Par exemple :

```
while x > 0 and y < 0 :
    # faire quelque chose avec x
    # faire quelque chose avec y

    print('x : ', x)
    print('y : ', y)
    print("condition : ", (x > 0 and y < 0))
```

Maintenant, lorsque vous exécutez le programme, vous verrez trois lignes affichées à chaque passage dans la boucle. Au dernier tour de la boucle, la condition doit être fausse. Si la boucle continue, vous serez en mesure de voir les valeurs de x et y, et vous pourrez comprendre pourquoi ces variables ne sont pas mises à jour correctement.

Récursion infinie

La plupart du temps, une récursion infinie laisse le programme s'exécuter pendant un certain temps, puis produit une erreur « profondeur maximale de récursivité dépassée ».

Si vous soupçonnez qu'une fonction est à l'origine d'une récursion infinie, assurez-vous qu'il y a un cas de base. Il doit y avoir quelque part une condition qui provoque le retour de la fonction sans faire un appel récursif. Si ce n'est pas le cas, vous devez repenser l'algorithme et identifier un cas de base.

S'il y a un cas de base, mais le programme ne semble pas l'atteindre, ajoutez au début de la fonction une instruction print qui imprime les paramètres. Maintenant, lorsque vous exécutez le programme, vous verrez quelques lignes s'afficher chaque fois que la fonction est invoquée, et vous verrez les valeurs des paramètres. Si les paramètres ne tendent pas à se rapprocher du cas de base, vous aurez quelques idées sur l'origine du problème.

Flux d'exécution

Si vous n'êtes pas sûr de savoir comment le flux d'exécution chemine à travers votre programme, ajoutez des instructions print au début de chaque fonction avec un message comme « entrer dans la fonction toto », où toto est le nom de la fonction.

Maintenant, lorsque vous exécutez le programme, il imprime une trace de chaque fonction invoquée.

A-2-3 - Lorsque j'exécute le programme, j'obtiens une exception

Si quelque chose se passe mal lors de l'exécution, Python affiche un message qui inclut le nom de l'exception, la ligne du programme où le problème est survenu, et une trace d'appels.

La trace d'appels identifie la fonction qui est en cours d'exécution, puis la fonction qui l'a appelée, et ensuite la fonction qui a appelé cette dernière, et ainsi de suite. Autrement dit, elle reconstitue la séquence d'appels de fonction qui vous a conduit à l'endroit où vous êtes, y compris le numéro de ligne dans votre fichier où chaque appel apparaît.

La première étape consiste à examiner l'endroit dans le programme où l'erreur est survenue et voir si vous pouvez comprendre ce qui est arrivé. Voici quelques-unes des erreurs d'exécution les plus courantes.

NameError : vous essayez d'utiliser une variable qui n'existe pas dans le contexte actuel. Vérifiez si le nom est orthographié correctement, ou au moins de manière cohérente. Et rappelez-vous que les variables locales sont locales ; vous ne pouvez pas les référencer à l'extérieur de la fonction où elles sont définies.

TypeError : il y a plusieurs causes possibles :

- vous essayez d'utiliser une valeur de façon incorrecte. Exemple : l'indexation d'une chaîne, d'une liste ou d'un tuple par autre chose qu'un entier ;
- il y a un décalage entre les éléments d'une chaîne de formatage et les éléments passés pour conversion. Cela peut se produire si le nombre d'éléments ne correspond pas ou si vous tentez une conversion invalide ;
- vous passez le mauvais nombre d'arguments à une fonction. Pour les méthodes, regardez la définition de la méthode et vérifiez si le premier paramètre est self. Ensuite, regardez l'invocation de la méthode ; assurez-vous que vous invoquez la méthode sur un objet avec le bon type et en fournissant correctement les autres arguments.

KeyError : vous tentez d'accéder à un élément d'un dictionnaire à l'aide d'une clé que le dictionnaire ne contient pas. Si les clés sont des chaînes de caractères, rappelez-vous que les majuscules/minuscules sont importantes.

AttributeError : vous tentez d'accéder à un attribut ou une méthode qui n'existe pas. Vérifiez l'orthographe ! Vous pouvez utiliser la fonction interne vars pour lister les attributs qui existent.

Si une AttributeError indique qu'un objet a NoneType, cela signifie qu'il est None. Donc le problème n'est pas le nom de l'attribut, mais l'objet.

La raison pour laquelle l'objet est None pourrait être le fait que vous avez oublié de renvoyer une valeur d'une fonction ; si vous arrivez à la fin d'une fonction sans rencontrer une instruction return, elle retourne None. Une autre cause fréquente est l'utilisation du résultat d'une méthode de liste, comme sort, qui renvoie None.

IndexError : l'indice que vous utilisez pour accéder à une liste, à une chaîne de caractères ou à un tuple est supérieur à la longueur de la collection moins un. Immédiatement avant l'endroit de l'erreur, ajoutez une instruction print pour afficher la valeur de l'indice et de la longueur du tableau. Le tableau a-t-il la bonne taille ? L'indice a-t-il la bonne valeur ?

Le débogueur de Python (pdb) est utile pour traquer les exceptions, car il vous permet d'examiner l'état du programme immédiatement avant l'erreur. Vous pouvez lire au sujet de pdb sur **https://docs.python.org/3/library/pdb.html**.

A-2-4 - J'ai ajouté tant d'instructions print, je suis submergé par la sortie

Un des problèmes de l'utilisation des instructions d'impression pour débogage est que vous pouvez vous retrouver submergé par les affichages. Il existe deux façons de procéder : simplifier la sortie ou simplifier le programme.

Pour simplifier la sortie, vous pouvez supprimer ou mettre en commentaires les instructions print qui ne sont pas utiles, ou les combiner, ou formater la sortie de sorte qu'elle soit plus facile à comprendre.

Pour simplifier le programme, il existe plusieurs choses que vous pouvez faire. Tout d'abord, réduire le problème sur lequel travaille le programme. Par exemple, si vous faites une recherche dans une liste, recherchez dans une *petite* liste. Si le programme prend des données en entrée de l'utilisateur, donnez-lui la saisie la plus simple qui provoque le problème.

Deuxièmement, nettoyez le programme. Supprimez le code mort et organisez le programme pour le rendre facile à lire autant que possible. Par exemple, si vous soupçonnez que le problème est dans une partie profondément imbriquée du programme, essayez de réécrire cette partie avec une structure plus simple. Si vous soupçonnez une grosse fonction, essayez de la diviser en fonctions plus petites et les tester séparément.

Souvent, le processus de recherche d'un cas de test minimal vous mène au bogue. Si vous remarquez qu'un programme fonctionne dans une situation, mais pas dans une autre, cela vous donne une idée de ce qui se passe.

De la même façon, la réécriture d'un bloc de code peut vous aider à trouver des bogues subtils. Si vous faites un changement et que vous pensez qu'il ne devrait pas affecter le programme, et il le fait, cela peut vous avertir.

A-3 - Erreurs sémantiques

À certains égards, les erreurs sémantiques sont les plus difficiles à déboguer, parce que l'interpréteur ne fournit aucune information à propos de ce qui ne va pas. Il n'y a que vous qui sachiez ce que le programme est censé faire.

La première étape est de faire un lien entre le texte du programme et le comportement que vous observez. Vous avez besoin d'une hypothèse sur ce que le programme est en train de faire. Une des choses qui rend cela difficile est que les ordinateurs exécutent le code si vite.

Vous voudrez souvent pouvoir ralentir le programme à une vitesse humaine, et avec quelques débogueurs vous le pouvez. Mais le temps qu'il faut pour insérer quelques instructions print bien placées est souvent court par rapport à la mise en place du débogueur, à l'insertion et à la suppression des points d'arrêt et au parcours du programme « pas à pas » vers l'endroit où l'erreur se produit.

A-3-1 - Mon programme ne fonctionne pas

Vous devriez vous poser ces questions.

- Y a-t-il quelque chose que le programme était censé faire, mais qui ne semble pas se faire ? Trouvez la section du code qui exécute cette fonction et assurez-vous qu'elle s'exécute quand vous pensez qu'elle le devrait.
- Se passe-t-il quelque chose qui ne devrait pas arriver ? Trouvez dans votre programme le code qui appelle cette fonction et regardez si elle s'exécute quand elle ne le devrait pas.
- Est-ce qu'une section de code produit un effet qui n'est pas celui que vous attendiez ? Assurez-vous que vous comprenez le code en question, surtout s'il implique des fonctions ou des méthodes dans d'autres modules Python. Lisez la documentation des fonctions que vous appelez. Essayez-les en écrivant des cas de tests simples et en vérifiant les résultats.

Pour programmer, vous avez besoin d'un modèle mental du fonctionnement des programmes. Si vous écrivez un programme qui ne fait pas ce que vous attendez, souvent le problème ne réside pas dans le programme ; il est dans votre modèle mental.

La meilleure façon de corriger votre modèle mental est de découper le programme en composants individuels (généralement les fonctions et les méthodes) et de tester chaque composant indépendamment. Une fois que vous trouvez la différence entre votre modèle et la réalité, vous pouvez résoudre le problème.

Bien sûr, vous devriez construire et tester les composants au fur et à mesure que vous développez le programme. Si vous rencontrez un problème, il devrait y avoir seulement une petite quantité de nouveau code qui n'est pas encore testé et connu comme correct.

A-3-2 - J'ai une grosse expression touffue qui ne fait pas ce que j'attends d'elle

Écrire des expressions complexes est très bien tant qu'elles restent lisibles, mais elles peuvent être difficiles à déboguer. Il est souhaitable de décomposer une expression complexe en une série d'affectations à des variables temporaires.

Par exemple, l'expression :

```
self.mains[i].ajouterCarte(self.mains[self.trouverVoisin(i)].popCarte())
```

peut être réécrite ainsi :

```
voisin = self.trouverVoisin(i)
carteChoisie = self.mains[voisin].popCarte()
self.mains[i].ajouterCarte(carteChoisie)
```

La version explicite est plus facile à lire parce que les noms de variables fournissent des renseignements additionnels, et elle est plus facile à déboguer parce que vous pouvez vérifier les types des variables intermédiaires et afficher leurs valeurs.

Un autre problème qui peut arriver avec de grosses expressions est que l'ordre d'évaluation n'est peut-être pas celui que vous attendiez. Par exemple, si vous traduisez l'expression $x / 2 \pi$ en Python, vous pourriez écrire :

```
y = x / 2 * math.pi
```

Cela n'est pas correct parce que la multiplication et la division ont la même priorité et sont évaluées de gauche à droite. Donc, cette expression calcule $x \pi / 2$.

Une bonne façon de déboguer des expressions est d'ajouter des parenthèses pour rendre explicite l'ordre d'évaluation :

```
y = x / (2 * math.pi)
```

Chaque fois que vous n'êtes pas sûr de l'ordre de l'évaluation, utilisez des parenthèses. Non seulement le programme va être correct (dans le sens de faire ce que vous aviez l'intention de lui demander), il sera également plus lisible pour d'autres personnes qui n'ont pas mémorisé l'ordre de précédence des opérateurs.

A-3-3 - J'ai une fonction qui ne renvoie pas ce que j'attends

Si vous avez comme valeur de retour une expression complexe, vous n'avez pas la possibilité d'afficher le résultat avant de le renvoyer. Encore une fois, vous pouvez utiliser une variable temporaire. Par exemple, au lieu de :

```
return self.mains[i].enleverEgales()
```

vous pourriez écrire :

```
compter = self.mains[i].enleverEgales()
return compter
```

Maintenant, vous avez la possibilité d'afficher la valeur de compter avant de la renvoyer.

A-3-4 - Je me retrouve complètement coincé et j'ai besoin d'aide

Tout d'abord, essayez de vous éloigner de l'ordinateur pendant quelques minutes. Les ordinateurs émettent des ondes maléfiques qui affectent le cerveau, ce qui provoque les symptômes suivants :

- frustration et rage ;
- croyances superstitieuses (« l'ordinateur me déteste ») et pensées magiques (« le programme ne fonctionne que quand je porte ma casquette à l'envers ») ;
- la « programmation errante aléatoire » (la tentative de programmer en écrivant chaque programme possible et de choisir celui qui fait ce qu'on souhaite).

Si vous vous retrouvez à souffrir d'un de ces symptômes, levez-vous et allez faire une petite promenade. Lorsque vous vous êtes calmé, réfléchissez au programme. Qu'est-ce qu'il fait ? Quelles sont les causes possibles de ce comportement ? À quand remonte la dernière fois que votre programme fonctionnait, et qu'avez-vous modifié depuis ?

Parfois, il faut juste du temps pour trouver un bogue. Je trouve souvent des bogues quand je suis loin de l'ordinateur et je laisse mon esprit vagabonder. Certains des meilleurs endroits pour trouver des bogues sont dans les trains, sous la douche et dans le lit, juste avant de vous endormir.

A-3-5 - Mais non, j'ai vraiment besoin d'aide

Cela arrive. Même les meilleurs programmeurs se retrouvent parfois coincés. Parfois, vous avez travaillé sur un programme si longtemps que vous ne pouvez pas voir l'erreur. Vous avez besoin d'une nouvelle paire d'yeux.

Avant de demander l'aide de quelqu'un d'autre, assurez-vous que vous êtes prêt. Votre programme doit être aussi simple que possible, et vous devriez travailler avec la plus petite quantité de données en entrée qui provoque l'erreur. Vous devriez avoir des instructions print aux bons endroits (et la sortie qu'elles produisent doit être compréhensible). Vous devez comprendre le problème assez bien pour le décrire de façon concise.

Lorsque vous demandez à quelqu'un de vous aider, assurez-vous de pouvoir lui donner l'information dont il a besoin :

- s'il y a un message d'erreur, quel est-il et quelle partie du programme indique-t-il ?
- quelle est la dernière chose que vous avez faite avant cette erreur ? Quelles sont les dernières lignes de code que vous avez écrites, ou quel est le nouveau cas de test qui échoue ?
- qu'avez-vous essayé jusqu'à présent, et qu'avez-vous appris ?

Lorsque vous trouvez le bogue, prenez une seconde pour réfléchir à ce que vous auriez pu faire pour le trouver plus vite. La prochaine fois que vous verrez quelque chose de similaire, vous serez en mesure de trouver le bogue plus rapidement.
Rappelez-vous, l'objectif n'est pas seulement de faire fonctionner le programme. L'objectif est d'apprendre comment faire fonctionner le programme.

B - Analyse des algorithmes

*Cette annexe est un extrait modifié du livre Think Complexity (**http://www.greenteapress.com/compmod/thinkcomplexity.pdf**), d'Allen B. Downey, publié par*

O'Reilly Media (2012). Quand vous aurez fini le présent livre, peut-être voudrez-vous lire cet autre livre.

L'**analyse algorithmique** ou analyse de la complexité des algorithmes est une branche de l'informatique qui étudie les performances des algorithmes, en particulier leur durée d'exécution et leur consommation mémoire. Voir **https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes**.

L'objectif pratique de l'analyse algorithmique est de prévoir la performance des différents algorithmes afin de guider les prises de décisions de conception.

Pendant la campagne présidentielle américaine de 2008, on a demandé au candidat Barack Obama de se livrer à une analyse improvisée lors d'une visite dans les locaux de Google. Le PDG de Google, Eric Schmidt, lui demanda en plaisantant « la meilleure façon de trier un million d'entiers de 32 bits ». Quelqu'un avait dû refiler le tuyau à Obama, car il répondit sans hésiter : « Je pense que le tri à bulles ne serait pas la bonne façon de procéder » (voir **http://www.youtube.com/watch?v=k4RRi_ntQc8**).

Il avait raison : le tri à bulle est conceptuellement simple, mais il est lent quand il s'agit de trier des volumes importants de données. La réponse qu'attendait Schmidt était sans doute le tri par base (**https://fr.wikipedia.org/wiki/Tri_par_base**) ou *radix sort*. (2)

L'objectif de l'analyse algorithmique est de comparer de façon significative différents algorithmes, mais il y a quelques problèmes.

- Les performances relatives des algorithmes peuvent dépendre des caractéristiques matérielles des machines : un algorithme peut être plus rapide sur une machine A et un autre faire mieux sur une machine B. La solution générale de ce problème est de spécifier un **modèle de machine** et d'analyser le nombre d'étapes, ou d'opérations, dont a besoin un algorithme pour arriver à ses fins sur un modèle donné.
- Les performances relatives peuvent dépendre des détails relatifs aux données à trier. Par exemple, certains algorithmes de tri sont plus rapides si les données sont déjà partiellement triées ; d'autres sont au contraire plus lents dans ce cas. Il est assez courant, pour remédier à ce genre de difficulté, d'étudier le pire des cas. Il est parfois utile d'analyser les performances du cas moyen, mais l'analyse est souvent plus difficile, et il n'est pas forcément facile de déterminer sur quel ensemble de cas faire une moyenne.
- Les performances relatives dépendent aussi de la taille du problème. Un algorithme de tri qui est rapide pour de petites listes peut devenir lent pour de grandes listes. On résout habituellement ce problème en exprimant la durée d'exécution (ou le nombre d'opérations) sous la forme d'une fonction de la taille du problème, et en comparant comment ces fonctions croissent quand la taille du problème augmente.

La bonne chose est que ce genre de comparaison se prête bien à une classification simple des algorithmes. Par exemple, si je sais que l'Algorithme A tend à être proportionnel à la taille n des données en entrée, et que l'Algorithme B tend à être proportionnel à n^2, alors j'attends que A soit plus rapide que B, du moins pour des valeurs élevées de n.

Ce type d'analyse peut réserver des surprises et nécessite donc certaines précautions, mais nous y reviendrons plus loin.

B-1 - Ordre de croissance et complexité

Supposons que vous ayez analysé deux algorithmes et estimé leur durée d'exécution en fonction de la taille n des données en entrée : l'Algorithme A nécessite $100\,n + 1$ étapes, et l'Algorithme B $n^2 + n + 1$ étapes.

Le tableau suivant illustre la durée d'exécution de ces algorithmes pour différentes tailles de données en entrée :

Taille des données	Algorithme A	Algorithme B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	100 010 001
100 000	10 000 001	$> 10^{10}$

Pour une taille des données de *n = 10*, l'Algorithme A paraît très mauvais : il prend presque dix fois plus de temps que l'Algorithme B. Mais pour *n = 100*, les deux algorithmes se valent à peu près, et, pour des tailles plus grandes, A est bien meilleur que B. La raison fondamentale est que pour des valeurs élevées de *n*, toute fonction qui contient un terme en n^2 croîtra plus rapidement qu'une fonction dont le terme dominant est *n*. Le **terme dominant** est le terme de degré le plus élevé (c'est-à-dire celui qui contient l'exposant le plus grand).

Pour l'Algorithme A, le terme dominant a un coefficient assez élevé, 100, ce qui explique pourquoi B est meilleur que A pour des valeurs faibles de *n*. Cependant, quels que soient les coefficients *a* et *b* choisis, il y aura toujours des valeurs de *n* suffisamment grandes pour lesquelles $an^2 > bn$.

Il en va de même avec les termes non dominants. Même si le temps d'exécution de l'Algorithme A était de *n + 1 000 000*, celui-ci serait tout de même meilleur que l'Algorithme B pour des valeurs suffisamment grandes de n.

En termes mathématiques, on s'intéresse ici en quelque sorte au comportement *asymptotique* de la fonction représentant le temps d'exécution d'un algorithme quand *n* prend des valeurs très grandes ; ainsi, le tableau ci-dessus reflète le fait que la limite d'une fonction polynomiale, quand sa variable tend vers l'infini, dépend du terme dominant. D'une façon générale, nous attendons que l'algorithme ayant le plus petit terme dominant soit le meilleur pour les grands problèmes, mais, pour les problèmes plus petits, il se peut qu'il y ait un **point de croisement** (ou point de changement de méthode) en dessous duquel un autre algorithme est meilleur. L'endroit où se situe ce point de croisement dépend des détails des algorithmes, des données en entrée et du matériel, si bien qu'on l'ignore généralement dans le cadre de l'analyse algorithmique. Mais cela ne signifie pas que vous deviez l'oublier complètement.

Si deux algorithmes ont des termes dominants du même ordre (même exposant), alors il est difficile de dire lequel est le meilleur ; une nouvelle fois, la réponse dépendra des détails. Du coup, du point de vue de l'analyse algorithmique, des fonctions ayant des termes dominants du même ordre sont considérées comme équivalentes, même s'ils ont des coefficients différents.

Un **ordre de croissance** est un ensemble de fonctions ayant un type de croissance équivalent ou considéré comme tel. Par exemple, *2n*, *100n* et *n+1* appartiennent au même ordre de croissance, que l'on note *O(n)* dans la « **notation asymptotique en O** » et que l'on appelle souvent **linéaire**, parce que toutes les fonctions de cet ensemble croissent linéairement en fonction de *n*. (À noter que le symbole « O » employé dans cette notation est la lettre « o » majuscule - on dit parfois aussi que c'est la lettre grecque *omicron* majuscule, qui s'écrit généralement de la même façon, « O » - mais pas le chiffre zéro 0 ; et *O(n)* se prononce habituellement « *o de n* » ou « *grand o de n* » [NdT].)

Toutes les fonctions dont le terme dominant est en n^2 appartiennent à $O(n^2)$. On les appelle **quadratiques**.

Le tableau suivant énumère quelques-uns des ordres de croissance les plus communément employés en analyse algorithmique, classés des meilleurs vers les pires :

Ordre de croissance	Nom
O(1)	Constante
O($\log_b n$)	Logarithmique (pour tout b)
O(n)	Linéaire
O(n $\log_b n$)	« Linéarithmique » (ou « quasi linéaire »)
O(n^2)	Quadratique
O(n^3)	Cubique
O(n^c)	Polynomiale (c entier positif)
O(c^n)	Exponentielle (pour tout *c*) ou parfois « géométrique »
n!	Factorielle

Pour les termes logarithmiques, la base du logarithme n'importe pas : changer de base équivaut à multiplier par une constante, ce qui ne change pas l'ordre de croissance. De la même façon, toutes les fonctions exponentielles appartiennent au même ordre de croissance, indépendamment de la base de l'exposant. Les fonctions exponentielles (et factorielles) croissent extrêmement rapidement, si bien que ces fonctions ne sont utiles dans la pratique que pour des problèmes très petits.

Exercice 1.

Lisez la page Wikipédia sur la Comparaison algorithmique (**https://fr.wikipedia.org/wiki/Comparaison_asymptotique** *), en complétant si possible avec la page équivalente en anglais (* **https://en.wikipedia.org/wiki/Big_O_notation** *), et répondez aux questions suivantes :*

1. *Quel est l'ordre de croissance de $n^3 + n^2$? Et de* 1 000 000 *$n^3 + n^2$? Et de n^3 +1 000 000 n^2 ?*
2. *Quel est l'ordre de croissance de $(n^2 + n) . (n + 1)$? Avant de commencer à faire les multiplications, rappelez-vous que vous n'avez besoin que du terme dominant.*
3. *Si la fonction f est en O(g), pour une fonction g non spécifiée, que pouvez-vous dire de l'ordre de croissance de f + b (b étant une constante) ?*
4. *Si les fonctions f1 et f2 sont en O(g), que pouvez-vous dire de f1 + f2 ?*
5. *Si f1 est en O(g) et f2 en O(h), que pouvez-vous dire de f1 + f2 ?*
6. *Si f1 est en O(g) et f2 en O(h), que pouvez-vous dire de f1 . f2 ?*

Les programmeurs qui s'intéressent aux performances trouvent souvent ce genre d'analyse difficile à avaler. Ils n'ont pas complètement tort : parfois, les coefficients et les termes non dominants font une vraie différence. Parfois, les détails du matériel utilisé, le langage de programmation employé et les caractéristiques des données en entrée font une grosse différence. Et, pour les petits problèmes, le comportement asymptotique est sans importance.

Mais si vous gardez à l'esprit ces mises en garde relatives aux cas particuliers, l'analyse algorithmique est un outil utile. Au moins en ce qui concerne les problèmes de grande dimension, les algorithmes « meilleurs » du point de vue de cette analyse sont effectivement meilleurs, et parfois très largement meilleurs. La différence entre deux algorithmes ayant le même ordre de croissance est généralement un facteur à peu près constant, mais la différence entre un bon algorithme et un mauvais algorithme est sans limites !

B-2 - Analyse des opérations Python de base

En Python, la plupart des opérations arithmétiques prennent un temps constant ; la multiplication prend plus de temps que l'addition et la soustraction, et la division en prend encore un peu plus, mais ces durées d'exécution ne dépendent pas de la taille des opérandes. Il y a cependant une exception pour les très très grands entiers : dans ce cas, la durée d'exécution croît avec le nombre de chiffres.

Les opérations d'indexation - lire ou écrire des éléments d'une séquence ou d'un dictionnaire - prennent également un temps constant, indépendant de la taille de la structure de donnée.

Une boucle for qui parcourt une séquence ou un dictionnaire a une complexité linéaire, à condition que toutes les opérations effectuées dans le corps de la boucle soient en complexité constante. Par exemple, accumuler les éléments d'une liste est une opération linéaire :

```
total = 0
for x in t:
    total += x
```

La fonction interne sum est également linéaire, car elle fait la même chose, mais elle tend à être plus rapide parce qu'elle bénéficie d'une implémentation plus efficace ; en termes d'analyse algorithmique, le coefficient de son terme dominant est plus petit.

En règle générale, si le corps d'une boucle est en $O(n^a)$, dans le cas par exemple d'une boucle imbriquée, alors l'ensemble de la boucle est en $O(n^{a+1})$. Il y a une exception si vous pouvez établir que la boucle sort après un nombre constant d'itérations. Si la boucle s'exécute *k* fois indépendamment de la valeur de *n*, alors la boucle restera en $O(n^a)$, même si *k* est grand.

Multiplier par *k* ne modifie par l'ordre de croissance, mais diviser par *k* ne le fait pas non plus. Donc, si le corps d'une fonction est en $O(n^a)$ et qu'elle s'exécute *n/k* fois, alors l'ensemble de la boucle est en $O(n^{a+1})$, même si *k* est grand.

La plupart des opérations sur les chaînes de caractères et les tuples sont linéaires, à l'exception des opérations d'indexation et de len, qui sont en temps constant. Les fonctions internes min et max sont linéaires. La durée d'exécution d'une opération de tranche est proportionnelle au nombre des valeurs renvoyées, mais indépendante de la taille des données en entrée.

Les concaténations de chaînes sont linéaires ; la durée d'exécution dépend de la somme des longueurs des opérandes.

Toutes les méthodes de chaînes sont linéaires, mais, si les longueurs des chaînes sont plafonnées par une constante - par exemple si les opérations portent sur des caractères uniques - alors elles sont considérées comme étant en temps constant. La méthode de chaîne join est linéaire ; la durée d'exécution dépend de la longueur totale des chaînes.

La plupart des méthodes de listes sont linéaires, mais il y a des exceptions :

- ajouter un élément à la fin d'une liste est en temps constant en moyenne ; quand l'opération manque de place, la liste est occasionnellement copiée dans un emplacement plus grand, mais, globalement, la durée d'exécution pour *n* ajouts est en *O(n)*, si bien que la durée moyenne est en *O(1) ;*
- retirer un élément de la fin d'une liste est en temps constant ;
- trier une liste avec les fonctions de tri Python est en *O(n log n)*.

La plupart des opérations et méthodes relatives aux dictionnaires sont en temps constant, mais il y a quelques exceptions :

- la durée d'exécution d'une mise à jour est proportionnelle au volume des données à mettre à jour, mais indépendante de la taille du dictionnaire à mettre à jour ;
- les fonctions keys, values et items sont en temps constant, car elles renvoient des itérateurs. Mais si vous bouclez sur ces itérateurs, la boucle sera linéaire.

Les performances affichées par les dictionnaires sont l'un des petits miracles de l'informatique. Nous étudierons comment ils fonctionnent dans la section **B.4**.

Exercice 2

Lisez la page Wikipédia sur les algorithmes de tri (**https://fr.wikipedia.org/wiki/Algorithme_de_tri#Classification** *) et répondez aux questions suivantes :*

1. *Qu'est-ce qu'un « tri par comparaisons » ? Quel est le meilleur ordre de croissance dans le pire des cas pour un tri par comparaison ?*
2. *Quel est l'ordre de croissance du tri à bulles, et pourquoi Barak Obama pense-t-il que c'est un mauvais choix ?*
3. *Quel est l'ordre de croissance du tri par base (*radix sort*) ? Quelles sont les conditions préalables à son utilisation ?*
4. *Qu'est-ce qu'un tri stable et pourquoi la stabilité peut-elle avoir de l'importance en pratique ?*
5. *Quel est le pire algorithme de tri (parmi ceux énumérés) ?*
6. *Quel algorithme de tri utilise la bibliothèque C ? Quel est l'algorithme de tri utilisé par Python ? Ces algorithmes sont-ils stables ? Il se peut que vous deviez faire quelques recherches sur Google pour répondre à ces questions.*
7. *Beaucoup des algorithmes qui ne sont pas des tris par comparaisons sont linéaires. Mais alors, pourquoi Python utilise-t-il un algorithme de tri par comparaisons en* O(n log n*) ?*

B-3 - Analyse des algorithmes de recherche

Un **algorithme de recherche** est un algorithme qui prend en entrée une liste d'éléments et un élément cible à rechercher dans cette collection, et renvoie souvent la position ou l'indice de la cible.

L'algorithme de recherche le plus simple est celui de la « recherche linéaire », qui parcourt les éléments de la liste en entrée dans l'ordre et s'arrête s'il trouve la cible. Dans le pire des cas, il doit parcourir toute la liste, et la durée d'exécution est donc linéaire.

L'opérateur de séquences `in` emploie une recherche linéaire, de même que les méthodes `find` et `count`.

Si les éléments de la séquence sont ordonnés, alors vous pouvez utiliser une **recherche dichotomique**, qui est en *O(log n)*. Une recherche par dichotomie est analogue à la méthode que vous utilisez probablement très naturellement pour trouver un numéro de téléphone dans un annuaire téléphonique : au lieu de commencer à lire les éléments un par un depuis le début, vous commencez à peu près au milieu et vérifiez si le nom que vous cherchez est avant ou après. S'il vient avant, vous cherchez dans la première moitié, sinon dans la seconde moitié. Dans un cas comme dans l'autre, vous divisez de moitié le nombre d'éléments restants. Et vous recommencez à diviser en deux la moitié que vous avez sélectionnée, et ainsi de suite jusqu'au succès (ou jusqu'à l'échec si l'élément recherché ne figure pas dans la liste).

Si la séquence possède 1 000 000 éléments, il faudra environ 20 étapes (dans le pire des cas) pour trouver l'élément recherché ou conclure qu'il n'est pas là. Cette méthode est donc, pour un million d'éléments, environ 50 000 fois plus rapide qu'une recherche linéaire.

La recherche dichotomique peut être beaucoup plus rapide qu'une recherche linéaire, mais il faut que la séquence soit ordonnée, ce qui peut nécessiter du travail supplémentaire.

Il existe une autre structure de données, nommée **table de hachage**, qui est plus rapide encore - elle peut effectuer une recherche en temps constant - et ne nécessite pas de trier les données. Les dictionnaires de Python sont implémentés avec des tables de hachage, et c'est la raison pour laquelle la plupart des opérations sur les dictionnaires, notamment l'opérateur `in`, sont en temps constant.

B-4 - Tables de hachage

Pour expliquer comment fonctionnent les tables de hachage et pourquoi leurs performances sont si bonnes, je vais commencer à coder une simple table de correspondance et l'améliorer progressivement jusqu'à arriver à une table de hachage.

J'utilise Python pour illustrer cette implémentation, mais, dans la vie réelle, vous n'auriez pas besoin d'écrire ce genre de code en Python : vous utiliseriez simplement un dictionnaire ! Donc, pour le reste de cette section, imaginez que les dictionnaires n'existent pas en Python et que vous vouliez mettre en œuvre une structure de données établissant une correspondance entre des clés et des valeurs. Les opérations que vous avez besoin de développer sont les suivantes :

- ajoute(c, v) : ajoute un nouvel élément faisant correspondre une clé c à une valeur v. Avec un dictionnaire Python nommé dict, cette opération s'écrirait dict[c] = v.
- cherche(c) : fait une recherche et renvoie la valeur correspondant à la clé c. Avec un dictionnaire Python nommé dict, cette opération s'écrirait dict[k] ou dict.get(c).

Supposons pour l'instant que chaque clé n'apparaît qu'une seule fois. La façon la plus simple de réaliser une telle table de correspondance est d'utiliser une liste de tuples, dans laquelle chaque tuple est une paire clé-valeur.

```
class CorrespondanceLineaire:

    def __init__(self):
        self.items = []

    def ajoute(self, c, v):
        self.items.append((c, v))

    def cherche(self, c):
        for clef, valeur in self.items:
            if clef == c:
                return valeur
        raise KeyError
```

La méthode ajoute insère un tuple clé-valeur à la fin de la liste d'éléments.

La méthode cherche utilise une boucle for pour parcourir la liste ; si elle trouve la clé recherchée, elle renvoie la valeur correspondante ; sinon, elle renvoie une exception KeyError. Cette méthode de recherche est linéaire, en *O(n)*.

Une autre solution serait de maintenir la liste triée par clé. Nous pourrions alors employer une recherche dichotomique, qui est en *O(log n)*. La recherche serait plus rapide, mais insérer un nouvel élément au milieu d'une liste est linéaire, ce n'est donc sans doute pas la meilleure solution. Il y a d'autres structures de données qui peuvent implémenter les opérations ajoute et cherche en des temps logarithmiques, mais ce n'est toujours pas aussi bien qu'un temps constant, donc cherchons d'autres solutions.

Une façon d'améliorer CorrespondanceLineaire est de diviser la liste de paires clé-valeur en listes plus petites. Voici une implémentation baptisée MeilleureCorrespondance, qui est une liste de 100 exemplaires de CorrespondanceLinéaire que nous appellerons *godets*. Comme nous le verrons dans un instant, l'ordre de croissance de cherche est toujours linéaire, mais MeilleureCorrespondance est un grand pas en direction des tables de hachage.

```
class MeilleureCorrespondance:

    def __init__(self, n=100):
        self.godets = []
        for i in range(n):
            self.godets.append(CorrespondanceLineaire())

    def trouve_godet(self, c):
        indice = hash(c) % len(self.godets)
        return self.godets[indice]

    def ajoute(self, c, v):
        m = self.trouve_godet(c)
        m.ajoute(k, v)

    def cherche(self, c):
        m = self.trouve_godet(c)
        return m.get(c)
```

La méthode __init__ crée une liste de *n* godets de type CorrespondancesLineaire.

La méthode trouve_godet est utilisée par ajoute et cherche pour déterminer dans quel godet mettre le nouvel élément, ou dans quel godet aller chercher un élément.

La méthode trouve_godet utilise la fonction interne hash de Python, une fonction de hachage qui prend en entrée un objet Python presque quelconque et renvoie un entier. Cette implémentation présente un défaut : elle ne fonctionne qu'avec des clés hachables. Les types mutables comme les listes et les dictionnaires ne sont pas hachables.

Les objets hachables qui sont considérés comme équivalents renvoient la même valeur de hachage, mais l'inverse n'est pas nécessairement vrai : deux objets ayant des valeurs différentes peuvent renvoyer la même valeur de hachage.

La méthode trouve_godet utilise l'opérateur modulo pour réduire les valeurs de hachage à un intervalle compris entre 0 et len(self.godets), en sorte que le résultat soit un indice valide de la liste. Bien entendu, ceci signifie que nombreuses valeurs de hachage seront réduites au même indice. Mais si la fonction de hachage répartit les choses de façon à peu près uniforme (les fonctions de hachages sont conçues pour cela), alors nous pouvons nous attendre à environ *n*/100 éléments par CorrespondanceLineaire.

Puisque la durée d'exécution de CorrespondanceLineaire.cherche est proportionnelle au nombre d'éléments, nous pouvons espérer que MeilleureCorrespondance sera environ 100 fois plus rapide que CorrespondanceLineaire. L'ordre de croissance est toujours linéaire, mais les coefficients dominants sont beaucoup plus petits. C'est bien, l'amélioration est spectaculaire, mais ce n'est toujours pas aussi bien qu'une table de hachage.

Voici (finalement) l'idée cruciale qui rend si rapides les tables de hachages : si vous parvenez à maintenir la longueur des godets CorrespondanceLinéaire en dessous d'une certaine limite maximale, alors la méthode CorrespondanceLineaire.cherche est en temps constant. Il vous suffit de garder en mémoire le nombre d'éléments et, quand ce nombre par godet CorrespondanceLinéaire dépasse un certain seuil, de redimensionner la table de hachage en ajoutant de nouveaux godets.

Voici une mise en œuvre d'une table de hachage :

```
class Hachage:

    def __init__(self):
        self.godets = MeilleureCorrespondance(2)
        self.num = 0

    def trouve(self, c):
        return self.godets.trouve(c)

    def ajoute(self, c, v):
        if self.num == len(self.maps.maps):
            self.resize()
        self.godets.ajoute(c, v)
        self.num += 1

    def redimensionne(self):
        nouveaux_godets = MeilleureCorrespondance(self.num * 2)
        for m in self.godets.godets:
            for c, v in m.items:
                nouveaux_godets.ajoute(c, v)
        self.godets = nouveaux_godets
```

Chaque Hachage contient un MeilleureCorrespondance ; __init__ commence avec deux MeilleureCorrespondance et initialise num qui garde le décompte du nombre d'éléments.

La méthode get retransmet les invocations à MeilleureCorrespondance. Le vrai travail a lieu dans ajoute, qui vérifie le nombre d'éléments et la taille de MeilleureCorrespondance ; s'ils sont égaux, le nombre moyen d'éléments par CorrespondanceLinéaire est 1, elle appelle redimensionne.

La méthode redimensionne crée un nouvel objet MeilleureCorrespondance, deux fois plus grand que le précédent, et « *rehache* » ensuite les éléments du vieux godet dans le nouveau.

Le rehachage est nécessaire parce que changer le nombre de CorrespondanceLinéaire modifie le dénominateur de l'opérateur modulo dans trouve_godet. Ce qui veut dire que certains objets qui étaient hachés dans le même CorrespondanceLinéaire vont être répartis ailleurs (et c'est exactement le but recherché, n'est-ce pas?).

Le rehachage est en temps linéaire, si bien que redimensionne sera linéaire, ce qui peut paraître mauvais, puisque j'ai promis que la méthode ajoute serait en temps constant. Mais rappelez-vous que nous ne devons pas rehacher à chaque fois, si bien qu'ajoute est habituellement en temps constant, et seulement occasionnellement linéaire. La quantité totale de travail requise pour ajouter *n* fois est proportionnelle à *n*, si bien que la durée moyenne de chaque ajout est en temps constant !

Pour voir comment ça marche, considérons que nous commençons avec un hachage vide et y ajoutons ensuite une séquence d'éléments. Nous commençons avec deux godets CorrespondanceLinéaire, les deux premiers ajouts sont donc rapides (pas de redimensionnement). Disons qu'ils consomment chacun une unité de travail. L'ajout suivant nécessite un redimensionnement, donc nous devons rehacher les deux premiers éléments (disons que cela représente deux unités de travail) et ajouter ensuite le troisième élément (une unité de travail de plus). Ajouter l'élément suivant coûte une unité, donc nous en sommes pour l'instant à 6 unités de travail pour 4 éléments.

L'ajout suivant coûte 5 unités, mais les trois ajouts suivants ne coûtent qu'une unité chacun, donc nous en arrivons à 14 unités pour les 8 premiers ajouts.

L'ajout suivant coûte 9 unités, mais nous pouvons maintenant ajouter 7 éléments supplémentaires au prix d'une unité chacun ; au total, nous en sommes à 30 unités pour les 16 premiers ajouts.

Après 32 ajouts, le coût total est de 62 unités, et j'espère que vous commencez à voir ce qui se passe. Après *n* ajouts, *n* étant une puissance de deux, le coût total est de *2n - 2* unités, si bien que le travail moyen par ajout est d'un peu moins de 2 unités. Le cas est optimal quand *n* est une puissance de deux, et le travail moyen est un peu plus élevé pour d'autres valeurs de *n*, mais cela importe peu. La chose importante est que nous sommes en *O(1)*.

La figure B.1 illustre ce processus graphiquement. Chaque bloc représente une unité de travail. Les colonnes représentent le travail total pour chaque ajout, dans l'ordre, de gauche à droite : les deux premiers ajouts coûtent une unité, le troisième 3 unités, et ainsi de suite.

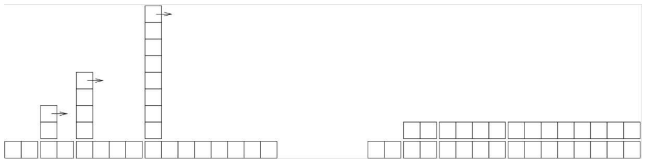


Figure B.1 : Le coût d'ajout dans un hachage.

Le travail de rehachage se présente comme une suite de « tours » de plus en plus hautes, avec des intervalles de plus en plus grands entre elles. Mais si vous faites basculer les tours vers la droite, afin de répartir le coût des redimensionnements sur l'ensemble des ajouts s'y rapportant, vous pouvez voir graphiquement que le coût total après *n* ajouts est de *2n - 2*.

Une caractéristique essentielle de cet algorithme est que, quand on redimensionne le hachage, il croît géométriquement, ce qui veut dire que l'on multiplie la taille par une constante (2 dans notre exemple) à chaque étape. Si on avait décidé de faire croître la taille arithmétiquement, c'est-à-dire d'ajouter un nombre constant d'emplacements à chaque fois, alors la durée moyenne par ajout aurait été linéaire. C'est le choix de la croissance géométrique de la taille qui permet d'obtenir un ajout en temps constant.

Vous pouvez télécharger mon implémentation de ce hachage à l'adresse suivante : **Map.py**. Mais souvenez-vous qu'il n'y a aucune raison de l'utiliser à d'autres fins que l'expérimentation pédagogique ; si vous avez besoin de ce genre d'outils, utilisez tout simplement un dictionnaire Python.

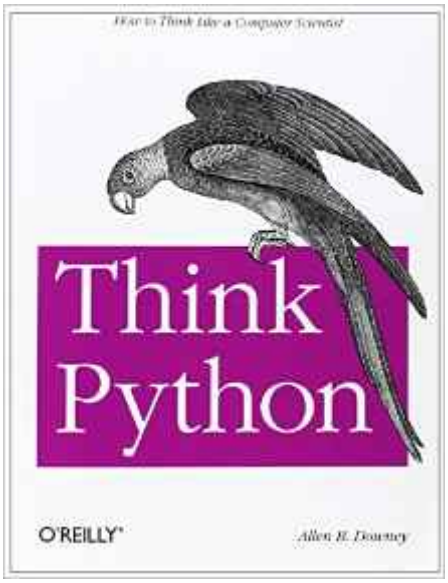
B-5 - Glossaire

- **analyse algorithmique :** une façon de comparer les algorithmes du point de vue de leur durée d'exécution et de leur besoin mémoire.
- **modèle de machine :** une représentation simplifiée d'un ordinateur utilisée pour décrire des algorithmes.
- **terme dominant :** dans un polynôme, le terme de plus haut degré (ayant l'exposant le plus grand).
- **point de croisement :** la taille du problème pour laquelle deux algorithmes nécessitent la même durée d'exécution ou la même empreinte mémoire.
- **ordre de croissance :** un ensemble de fonctions qui croissent toutes d'une façon considérée comme équivalente du point de vue de l'analyse algorithmique. Par exemple, toutes les fonctions qui croissent linéairement appartiennent au même ordre de croissance.
- **notation asymptotique en O :** notation permettant de représenter un ordre de croissance ; par exemple *O(n)* représente l'ensemble des fonctions qui croissent linéairement.
- **quadratique :** un algorithme dont la durée d'exécution est proportionnelle à n^2, n étant une mesure de la taille du problème.
- **recherche :** le problème consistant à trouver l'emplacement d'un élément dans une collection (par exemple une liste ou un dictionnaire) ou à conclure qu'il n'est pas présent.
- **recherche par dichotomie :** une méthode de recherche dans une liste ordonnée consistant à recherche un élément au milieu de la liste et à poursuivre la recherche dans la moitié de la liste susceptible de contenir l'élément cible, et ainsi de suite.
- **table de hachage :** une structure de données qui représente une collection de paires clé-valeur et permet d'effectuer des recherches en temps constant.

Remerciements Developpez

Nous remercions **Allen B. Downey** de nous avoir aimablement autorisés à publier ce livre, dont le texte original peut être trouvé sur **http://greenteapress.com/thinkpython2/html/index.html**. Nous remercions aussi **Mishulyna** et **Laurent Rosenfeld** pour la traduction, **Laurent Rosenfeld** pour sa relecture technique ainsi que **ClaudeLELOUP** pour sa relecture orthographique.

Les commentaires et les suggestions d'amélioration sont les bienvenus. Aussi, après votre lecture, n'hésitez pas : commentez !



1 : popen est maintenant obsolète, ce qui signifie que nous ne sommes plus censés l'utiliser ; à la place, nous devrions commencer à employer le module subprocess. Mais pour les cas simples, je trouve subprocess plus compliqué que nécessaire. Par conséquent, je vais continuer à utiliser popen jusqu'à ce qu'ils le suppriment.
2 : Mais si on vous pose cette question dans un entretien d'embauche, il serait à mon avis plus judicieux de répondre : « La manière la plus rapide de trier un million d'entiers est sans doute d'utiliser la fonction de tri, quelle qu'elle soit, qui est disponible dans le langage que j'utilise. Ses performances sont suffisantes pour la grande majorité des utilisations. Mais s'il s'avérait que mon application est trop lente, alors j'utiliserais un logiciel de profilage pour déterminer où elle passe beaucoup de temps. Et s'il apparaissait qu'un algorithme de tri plus rapide aurait un effet significatif sur les performances de l'application, alors j'essaierais de dénicher une bonne implémentation du tri par base. »