spécialité Numérique et sciences informatiques

24 leçons avec exercices corrigés

classe de terminale

Thibaut Balabonski
Sylvain Conchon
Jean-Christophe Filliâtre
Kim Nguyen

préface de Xavier Leroy

À tous les enseignants pionniers de la spécialité NSI,
qui dans les lycées se sont investis sans compter
pour faire vivre cette discipline où tout était encore à créer.

Préface

Enseigner l'informatique au lycée n'est pas une idée nouvelle : les premières expériences pédagogiques remontent aux années 1980. Mais c'est depuis peu qu'un enseignement approfondi d'informatique est offert dans tous les lycées généraux, avec, depuis la rentrée 2019, un enseignement Sciences numériques et technologie (SNT) en seconde et un enseignement de spécialité Numérique et sciences informatiques (NSI) en première et, dès la rentrée 2020, en terminale.

La spécialité NSI est ambitieuse : par le volume horaire qui lui est consacré, mais aussi par la richesse de son programme, qui va bien au-delà de la simple «littératie informatique» et aborde frontalement les notions fondamentales de la science informatique. Algorithmique et programmation — les deux piliers de la pensée informatique — y sont largement présentes, mais on y découvre également les bases de données, l'architecture des ordinateurs, les systèmes d'exploitation et les réseaux.

Un programme d'une telle richesse est une aventure pour l'élève comme pour l'enseignant, ce dernier n'ayant souvent pas suivi de cours d'informatique comparables lors de sa formation universitaire. L'ouvrage de Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre et Kim Nguyen, sobrement intitulé *Spécialité Numérique et sciences informatiques*, est le parfait guide pour les accompagner tout au long de ce voyage à la découverte des bases de l'informatique.

Le premier volume de cet ouvrage, consacré à la classe de première et paru à la rentrée 2019, avait beaucoup impressionné : Gilles Dowek, l'un des initiateurs de la spécialité NSI, a employé à son propos le mot de «miracle». C'est donc avec impatience que je découvre le second volume, consacré à la classe de terminale, et avec plaisir que je constate que nos quatre auteurs ont fait, une fois de plus, un travail admirable.

Deux styles sont (hélas) très répandus dans les manuels pour l'enseignement de l'informatique : la vulgarisation légère, qui effleure plaisamment le sujet mais laisse le lecteur sur sa faim, et le traité universitaire, qui l'étouffe sous l'exhaustivité et la rigueur formelle. Rien de tel dans l'ouvrage de Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre et Kim Nguyen : la présentation des concepts va droit à l'essentiel, mais ne cède jamais à l'approximation. C'est ainsi, dans un bel équilibre entre intuition

et rigueur, que ce livre nous fait parcourir d'un bon pas — *andante* — un joli chemin à travers les bases de l'informatique.

Ce livre sera sans aucun doute une grande aide dans la mise en place de la spécialité NSI. Remercions les auteurs pour cette précieuse contribution, et souhaitons à tous les enseignants et à tous les élèves de la spécialité NSI bien des satisfactions et bien des succès dans leur découverte et leur transmission de la pensée informatique.

Xavier Leroy

Professeur au Collège de France,
chaire de sciences du logiciel

Avant-propos

À qui s'adresse cet ouvrage ? Cet ouvrage s'adresse autant à l'enseignant qu'à l'élève. Un élève de terminale trouvera dans cet ouvrage un rappel du cours, de nombreux exercices pour s'entraîner, ainsi que des encarts pour approfondir certains points. L'enseignant y trouvera un cours structuré pour mener l'enseignement de NSI en classe de terminale, sous la forme de vingt-quatre leçons couvrant tous les points du programme officiel. Chaque leçon prend la forme d'un chapitre, contenant à la fois l'introduction de nouvelles notions et des exercices corrigés. Les leçons peuvent être traitées dans l'ordre, au sens où chacune ne fait appel qu'à des notions introduites dans les leçons précédentes. Il reste possible de traiter beaucoup de leçons dans un ordre différent.

Cet ouvrage fait suite à un premier volume pour l'enseignement de la spécialité NSI en classe de première. Nous y faisons parfois référence (avec la notation [NSI 1re, 9.3] pour un renvoi vers la section 9.3, par exemple) mais ces références peuvent être facilement ignorées.

Style. On adopte un style de programmation en Python le plus idiomatique possible, mais tout en restant relativement simple. En particulier, on s'interdit d'utiliser des concepts et notations introduits dans des chapitres ultérieurs, ce qui rend parfois le code un peu plus lourd qu'il ne pourrait être.

Exercices. Cet ouvrage contient de nombreux exercices, regroupés à chaque fois en fin de chapitre. Les exercices sont tous corrigés, les solutions étant regroupées à la fin de l'ouvrage. Pour chaque exercice, il existe le plus souvent de très nombreuses solutions. Nous n'en donnons qu'une seule, avec seulement parfois une discussion sur des variantes possibles. Certains exercices sont plus longs que d'autres et peuvent constituer des séances de travaux pratiques relativement longues voire de petits projets. Des exemples sont le jeu des lemmings (exercice 28 page 65) ou celui de la course de tortues (exercice 29 page 67), la simulation d'attente à des guichets (exercice 69 page 145) ou encore la méthode de Karatsuba (exercice 117 page 231).

Le site du livre. Le site `https://www.nsi-terminale.fr/` propose des ressources complémentaires. En particulier, il donne accès au code Python de tous les programmes décrits dans cet ouvrage. Ils pourront ainsi être facilement réutilisés, par exemple dans des séances de travaux pratiques visant à les manipuler ou à les modifier.

Remerciements. Nous tenons à remercier très chaleureusement toutes les personnes qui ont contribué à cet ouvrage par leur relecture attentive et leurs suggestions pertinentes, à savoir David Baelde, Lila Boukhatem, Alain Busser, Jérôme Duval, François Fayard, Yann Régis-Gianas, Laurent Sartre. Nous remercions tout particulièrement notre collègue Frédéric Voisin pour sa relecture, de très grande qualité, de presque tous les chapitres de ce livre. De nombreux enseignants de la spécialité NSI de classe de première nous ont fait des retours constructifs et encourageants sur le précédent volume et nous tenons à les en remercier sincèrement. Nous remercions aussi les participants au groupe de discussion national NSI : certaines remarques de ce livre font écho à leurs échanges. Nous sommes reconnaissants à Corinne Baud et Anne Laure Tedesco, des éditions Ellipses, pour la confiance qu'elles nous ont accordée et leur réactivité. Nous remercions également Didier Rémy pour son excellent paquet LaTeX `exercise`. Enfin, nous sommes très honorés que Xavier Leroy ait accepté de préfacer cet ouvrage et nous le remercions vivement.

Table des matières

I Programmation **1**

1 Récursivité **3**
1.1 Problème : la somme des n premiers entiers . . . 3
1.2 Formulations récursives . . . 6
1.3 Programmer avec des fonctions récursives . . . 11
Exercices . . . 16

2 Modularité **19**
2.1 Variations sur les ensembles . . . 19
2.2 Modules, interfaces et encapsulation . . . 26
2.3 Exceptions . . . 34
Exercices . . . 40

3 Programmation objet **45**
3.1 Classes et attributs : structurer les données . . . 46
3.2 Méthodes : manipuler les données . . . 50
3.3 Retour sur l'encapsulation . . . 57
3.4 Héritage . . . 59
Exercices . . . 62

4 Mise au point des programmes **71**
4.1 Types . . . 71
4.2 Tester un programme . . . 79
4.3 Invariant de structure . . . 84
Exercices . . . 86

5 Programmation fonctionnelle **89**
5.1 Fonctions passées en arguments . . . 91
5.2 Fonctions renvoyées comme résultats . . . 95
5.3 Structures de données immuables . . . 97
Exercices . . . 100

II Algorithmique **105**

6 Listes chaînées **107**
6.1 Structure de liste chaînée . . . 108
6.2 Opérations sur les listes . . . 111
6.3 Modification d'une liste . . . 119
6.4 Encapsulation dans un objet . . . 121
Exercices . . . 124

7 Piles et files **127**
7.1 Interface et utilisation d'une pile . . . 128
7.2 Interface et utilisation d'une file . . . 130
7.3 Réaliser une pile avec une liste chaînée . . . 133
7.4 Réaliser une file avec une liste chaînée mutable . . . 135
7.5 Réaliser une file avec deux piles . . . 138
Exercices . . . 141

8 Arbres binaires **147**
8.1 Structures arborescentes . . . 147
8.2 Définition et propriétés des arbres binaires . . . 148
8.3 Représentation en Python . . . 151
8.4 Algorithmique des arbres binaires . . . 153
Exercices . . . 155

9 Arbres binaires de recherche **157**
9.1 Notion . . . 158
9.2 Recherche dans un ABR . . . 159
9.3 Ajout dans un ABR . . . 161
9.4 Suppression dans un ABR . . . 166
9.5 Encapsulation dans un objet . . . 168
9.6 Arbre équilibré . . . 169
Exercices . . . 170

10 Autres structures arborescentes **173**
10.1 Arborescence . . . 173
10.2 Exemples de structures arborescentes . . . 176
Exercices . . . 185

11 Graphes **189**
11.1 Définitions . . . 190
11.2 Exemples de graphes . . . 194
11.3 Représentation d'un graphe en Python . . . 196
11.4 Exemple d'algorithme sur les graphes . . . 202
Exercices . . . 204

12 Parcours en profondeur et en largeur **207**
12.1 Parcours en profondeur 208
12.2 Détecter la présence d'un cycle dans un graphe orienté 212
12.3 Parcours en largeur 214
Exercices 218

13 Diviser pour régner **221**
13.1 Retour sur la recherche dichotomique 222
13.2 Tri fusion 224
Exercices 229

14 Programmation dynamique **233**
14.1 Rendu de monnaie 234
14.2 Alignement de séquences 239
Exercices 246

15 Recherche textuelle **249**
15.1 Un algorithme simple 250
15.2 Accélération de la recherche 252
15.3 Un algorithme plus efficace : Boyer-Moore 254
Exercices 261

16 Calculabilité **263**
16.1 Problème : détecteur d'erreurs en Python 263
16.2 Calculabilité, des mathématiques à l'informatique 268
16.3 Limites de la calculabilité 274
Exercices 280

III Bases de données **283**

17 Modèle relationnel **285**
17.1 Le modèle relationnel 286
17.2 Modélisation relationnelle des données 288
Exercices 295

18 Bases de données relationnelles **297**
18.1 SQL : un langage de définition de données 298
18.2 Types de données en SQL 300
18.3 Spécification des contraintes d'intégrité 302
18.4 Suppression de tables 304
18.5 Insertion dans une table 305
Exercices 306

19 Requêtes SQL et mises à jour **309**
19.1 Sélection de données 310
19.2 Modification des données 319
19.3 Requêtes imbriquées 323
Exercices 325

20 Systèmes de Gestion de Bases de Données **329**
20.1 Historique 330
20.2 Transactions 331
20.3 Interaction entre un SGBD et un programme 337
Exercices 342

IV Architectures matérielles, systèmes d'exploitation et réseaux **345**

21 Circuits intégrés **347**
21.1 Microcontrôleurs 348
21.2 Système sur puce 351
Exercices 356

22 Gestion des processus et des ressources **359**
22.1 L'ordonnanceur 360
22.2 Commandes Unix de gestion des processus 364
22.3 Programmation concurrente 367
Exercices 378

23 Protocoles de routage **383**
23.1 Le protocole RIP 386
23.2 Protocole OSPF 390
23.3 Commandes système 396
Exercices 402

24 Sécurisation des communications **405**
24.1 Cryptographie symétrique 406
24.2 Cryptographie asymétrique 410
24.3 Authentification des participants 415
24.4 Le protocole HTTPS 417
Exercices 424

Solutions des exercices **427**

Index **507**

Première partie

Programmation

Chapitre 1

Récursivité

Notions introduites

- définitions récursives
- programmation avec fonctions récursives
- arbre d'appels
- modèle d'exécution et pile d'appels

On aborde dans ce chapitre la programmation à l'aide de *fonctions récursives*. Il s'agit à la fois d'un style de programmation mais également d'une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.

1.1 Problème : la somme des n premiers entiers

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante [1] :

$$0 + 1 + 2 + \cdots + n \tag{1.1}$$

Bien que cette formule nous paraisse simple et intuitive, il n'est pas si évident de l'utiliser pour écrire en Python une fonction `somme(n)` qui renvoie la somme des `n` premiers entiers. L'une des difficultés est de trouver un moyen de programmer la répétition des calculs qui est représentée par la notation $+\cdots+$.

Une solution pour calculer cette somme consiste à utiliser une boucle **for** pour parcourir tous les entiers `i` entre `0` et `n`, en s'aidant d'une variable locale `r` pour accumuler la somme des entiers de `0` à `i`. On obtient par exemple le code Python suivant :

1. Cet exemple est inspiré du cours *Programmation récursive* de Christian Queinnec.

```
def somme(n):
    r = 0
    for i in range(n + 1):
        r = r + i
    return r
```

S'il n'est pas difficile de se convaincre que la fonction `somme(n)` ci-dessus calcule bien la somme des n premiers entiers, on peut néanmoins remarquer que ce code Python n'est pas *directement* lié à la formule (1.1).

En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable intermédiaire `r` est nécessaire pour calculer cette somme. Certains peuvent y voir l'art subtil de la programmation, d'autres peuvent se demander s'il ne serait pas possible de donner une définition mathématique plus précise à cette somme, à partir de laquelle il serait plus « simple » d'écrire un programme Python.

Il existe en effet une autre manière d'aborder ce problème. Il s'agit de définir une fonction mathématique $somme(n)$ qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0, \\ n + somme(n-1) & \text{si } n > 0. \end{cases}$$

Cette définition nous indique ce que vaut $somme(n)$ pour un entier n quelconque, selon que n soit égal à 0 ou strictement positif. Ainsi, pour $n = 0$, la valeur de $somme(0)$ est simplement 0. Dans le cas où n est strictement positif, la valeur de $somme(n)$ est $n + somme(n-1)$.

Par exemple, voici ci-dessous les valeurs de $somme(n)$, pour n valant 0, 1, 2 et 3.

$$\begin{array}{rcl} somme(0) & = & 0 \\ somme(1) & = & 1 + somme(0) \;=\; 1+0 \;=\; 1 \\ somme(2) & = & 2 + somme(1) \;=\; 2+1 \;=\; 3 \\ somme(3) & = & 3 + somme(2) \;=\; 3+3 \;=\; 6 \end{array}$$

Comme on peut le voir, la définition de $somme(n)$ dépend de la valeur de $somme(n-1)$. Il s'agit là d'une définition *récursive*, c'est-à-dire d'une définition de fonction qui fait appel à elle-même. Ainsi, pour connaître la valeur de $somme(n)$, il faut connaître la valeur de $somme(n-1)$, donc connaître la valeur de $somme(n-2)$, etc. Ceci jusqu'à la valeur de $somme(0)$ qui ne dépend de rien et vaut 0. La valeur de $somme(n)$ s'obtient en ajoutant toutes ces valeurs.

L'intérêt de cette définition récursive de la fonction $somme(n)$ est qu'elle est directement calculable, c'est-à-dire exécutable par un ordinateur. En particulier, cette définition est directement programmable en Python, comme le montre le code ci-dessous.

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

L'analyse par cas de la définition récursive est ici réalisée par une instruction conditionnelle pour tester si l'argument `n` est égal à `0`. Si c'est le cas, la fonction renvoie la valeur `0`, sinon elle renvoie la somme `n + somme(n - 1)`. Cet appel à `somme(n - 1)` dans le corps de la fonction est un *appel récursif*, c'est-à-dire un appel qui fait référence à la fonction que l'on est en train de définir. On dit de toute fonction qui contient un appel récursif que c'est une *fonction récursive*.

Par exemple, l'évaluation de l'appel à `somme(3)` peut se représenter de la manière suivante

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + somme(1)
                                |
                           return 1 + somme(0)
                                          |
                                      return 0
```

où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n == 0` de la conditionnelle. Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée un *arbre d'appels*.

Ainsi, pour calculer la valeur renvoyée par `somme(3)`, il faut tout d'abord appeler `somme(2)`. Cet appel va lui-même déclencher un appel à `somme(1)`, qui à son tour nécessite un appel à `somme(0)`. Ce dernier appel se termine directement en renvoyant la valeur `0`. Le calcul de `somme(3)` se fait donc « à rebours ». Une fois que l'appel à `somme(0)` est terminé, c'est-à-dire que la valeur `0` a été renvoyée, l'arbre d'appels a la forme suivante, où l'appel à `somme(0)` a été remplacé par `0` dans l'expression `return 1 + somme(0)`.

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + somme(1)
                                |
                           return 1 + 0
```

À cet instant, l'appel à `somme(1)` peut alors se terminer et renvoyer le résultat de la somme `1 + 0`. L'arbre d'appels est alors le suivant.

```
somme(3) = return 3 + somme(2)
                        |
                   return 2 + 1
```

Enfin, l'appel à `somme(2)` peut lui-même renvoyer la valeur `2 + 1` comme résultat, ce qui permet à `somme(3)` de se terminer en renvoyant le résultat de `3 + 3`.

`somme(3)` = `return 3 + 3`

On obtient bien au final la valeur `6` attendue.

Une notation ambiguë. La formule (1.1) est non seulement éloignée du programme qui la calcule, elle est également ambiguë quant à la spécification de son résultat. À lire cette formule à la lettre, comment savoir si la somme des premiers entiers pour $n = 2$ est $0 + 1 + 2$ ou $0 + 1 + 2 + 2$? Si la réponse à cette question peut sembler évidente, elle ne l'est que parce que nous avons l'habitude de la signification de $+ \cdots +$ et savons que les entiers (0, 1 et 2) dans cette formule sont déjà des instances de n.
De manière générale, la programmation imposant la précision, nous sommes souvent amenés comme ici à considérer avec une rigueur nouvelle certaines choses « évidentes ».

1.2 Formulations récursives

Une formulation récursive d'une fonction est toujours constituée de plusieurs cas, parmi lesquels on distingue des *cas de base* et des *cas récursifs* du calcul.

Les cas récursifs sont ceux qui renvoient à la fonction en train d'être définie ($somme(n) = n + somme(n - 1)$ si $n > 0$). Les cas de base de la définition sont à l'inverse ceux pour lesquels on peut obtenir le résultat sans avoir recours à la fonction définie elle-même ($somme(n) = 0$ si $n = 0$). Ces cas de base sont habituellement les cas de valeurs particulières pour lesquelles il est facile de déterminer le résultat.

Prenons comme deuxième exemple l'opération de puissance n-ième d'un nombre x, c'est-à-dire la multiplication répétée n fois de x avec lui-même, que l'on écrit habituellement de la manière suivante

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ fois}}$$

avec, par convention, que la puissance de x pour $n = 0$ vaut 1.

Pour écrire une version récursive de x^n, on va définir une fonction $puissance(x, n)$ en commençant par chercher les cas de base à cette opération. Ici, le cas de base évident est celui pour $n = 0$. On écrira donc la définition (partielle) suivante :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0, \\ ? & \text{si } n > 0. \end{cases}$$

Pour définir la valeur de $puissance(x,n)$ pour un entier n strictement positif, on suppose que l'on connaît le résultat de x à la puissance $n-1$, c'est-à-dire la valeur de $puissance(x,n-1)$. Dans ce cas, $puissance(x,n)$ peut simplement être définie par $x \times puissance(x,n-1)$. Au final, on obtient donc la définition suivante :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0, \\ x \times puissance(x,n-1) & \text{si } n > 0. \end{cases}$$

Faire confiance à la récursion. Pour faciliter l'écriture des cas récursifs, il est très important de supposer que les appels récursifs donnent les bons résultats pour les valeurs sur lesquelles ils opèrent, sans chercher à « construire » dans sa tête l'arbre des appels pour se convaincre du bien fondé de la définition.

Définitions récursives plus riches

Toute formulation récursive d'une fonction possède au moins un cas de base et un cas récursif. Ceci étant posé, une grande variété de formes est possible.

Cas de base multiples. La définition de la fonction $puissance(x,n)$ n'est pas unique. On peut par exemple identifier deux cas de base « faciles », celui pour $n = 0$ mais également celui pour $n = 1$ avec $puissance(x,1) = x$. Ce deuxième cas de base a l'avantage d'éviter de faire la multiplication (inutile) $x \times 1$ de la définition précédente. Ainsi, on obtient la définition suivante avec deux cas de base :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ x \times puissance(x,n-1) & \text{si } n > 1. \end{cases}$$

Bien sûr, on pourrait continuer à ajouter des cas de base pour $n = 2$, $n = 3$, etc., mais cela n'apporterait rien à la définition. En particulier, cela ne réduirait pas le nombre de multiplications à effectuer.

Cas récursifs multiples. Il est également possible de définir une fonction avec plusieurs cas récursifs. Par exemple, on peut donner une autre définition pour $puissance(x, n)$ en distinguant deux cas récursifs selon la parité de n. En effet, si n est pair, on a alors $x^n = (x^{n/2})^2$. De même, si n est impair, on a alors $x^n = x \times (x^{(n-1)/2})^2$, où l'opération de division est supposée ici être la division entière. Ceci nous amène à définir la fonction $puissance(x, n)$ de la manière suivante, en supposant que l'on dispose d'une fonction $carre(x) = x \times x$.

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0, \\ carre(puissance(x, n/2)) & \text{si } n \geq 1 \text{ et } n \text{ est pair}, \\ x \times carre(puissance(x, (n-1)/2)) & \text{si } n \geq 1 \text{ et } n \text{ est impair}. \end{cases}$$

Pour des raisons dont nous discuterons un peu plus loin, cette définition va nous permettre d'implémenter en Python une version plus efficace de la fonction *puissance*.

Double récursion. Les expressions qui définissent une fonction peuvent aussi dépendre de *plusieurs* appels à la fonction en cours de définition. Par exemple, la fonction *fibonacci*(n), qui doit son nom au mathématicien Leonardo Fibonacci, est définie récursivement, pour tout entier naturel n, de la manière suivante :

$$fibonacci(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ fibonacci(n-2) + fibonacci(n-1) & \text{si } n > 1. \end{cases}$$

Voici par exemple les premières valeurs de cette fonction.

$$\begin{array}{lcl} fibonacci(0) & = & 0 \\ fibonacci(1) & = & 1 \\ fibonacci(2) & = & fibonacci(0) + fibonacci(1) = 0+1 = 1 \\ fibonacci(3) & = & fibonacci(1) + fibonacci(2) = 1+1 = 2 \\ fibonacci(4) & = & fibonacci(2) + fibonacci(3) = 1+2 = 3 \\ fibonacci(5) & = & fibonacci(3) + fibonacci(4) = 2+3 = 5 \\ \ldots & & \end{array}$$

Récursion imbriquée. Les occurrences de la fonction en cours de définition peuvent également être *imbriquées*. Par exemple, la fonction $f_{91}(n)$ ci-dessous, que l'on doit à John McCarthy (informaticien et lauréat du prix Turing en 1971), est définie avec deux occurrences imbriquées, de la manière suivante :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f_{91}(f_{91}(n+11)) & \text{si } n \leq 100. \end{cases}$$

Voici par exemple la valeur de $f_{91}(99)$:

$$\begin{array}{rcll} f_{91}(99) & = & f_{91}(f_{91}(110)) & \text{puisque } 99 \leq 100, \\ & = & f_{91}(100) & \text{puisque } 110 > 100, \\ & = & f_{91}(f_{91}(111)) & \text{puisque } 100 \leq 100, \\ & = & f_{91}(101) & \text{puisque } 111 > 100, \\ & = & 91 & \text{puisque } 101 > 100. \end{array}$$

Le fait que $f_{91}(99)$ renvoie 91 n'est pas un hasard ! On peut en effet démontrer que $f_{91}(n) = 91$, pour tout entier naturel n inférieur ou égal à 101.

Récursion mutuelle. Il est également possible, et parfois nécessaire, de définir plusieurs fonctions récursives en *même temps*, quand ces fonctions font référence les unes aux autres. On parle alors de définitions *récursives mutuelles*.

Par exemple, les fonctions $a(n)$ et $b(n)$ ci-dessous, inventées par Douglas Hofstadter (il y fait référence dans son ouvrage *Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle*, pour lequel il a obtenu le prix Pulitzer en 1980), sont définies par récursion mutuelle de la manière suivante :

$$a(n) = \begin{cases} 1 & \text{si } n = 0, \\ n - b(a(n-1)) & \text{si } n > 0. \end{cases}$$

$$b(n) = \begin{cases} 0 & \text{si } n = 0, \\ n - a(b(n-1)) & \text{si } n > 0. \end{cases}$$

On peut vérifier que les premières valeurs de ces deux suites sont bien

n :	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$
$a(n)$:	$1, 1, 2, 2, 3, 3, 4, 5, 5, 6, \ldots$
$b(n)$:	$0, 0, 1, 2, 2, 3, 4, 4, 5, 6, \ldots$

D'une manière amusante, on peut montrer que les deux séquences diffèrent à un indice n si et seulement si $n+1$ est un nombre de Fibonacci, c'est-à-dire s'il existe un entier k tel que $fibonacci(k) = n + 1$.

Définitions récursives bien formées

Il est important de respecter quelques règles élémentaires lorsqu'on écrit une définition récursive. Tout d'abord, il faut s'assurer que la récursion va bien se terminer, c'est-à-dire que l'on va finir par « retomber » sur un cas de base de la définition. Ensuite, il faut que les valeurs utilisées pour appeler la fonction soient toujours dans le domaine de la fonction. Enfin, il convient de vérifier qu'il y a bien une définition pour toutes les valeurs du domaine.

Prenons comme premier exemple la fonction $f(n)$ ci-dessous définie de la manière suivante :

$$f(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + f(n+1) & \text{si } n > 0. \end{cases}$$

Cette définition est incorrecte car la valeur de $f(n)$, pour tout n strictement positif, ne permet pas d'atteindre le cas de base pour $n = 0$. Par exemple, la valeur de $f(1)$ est :

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = \ldots$$

Notre deuxième exemple est celui d'une définition récursive pour une fonction $g(n)$ qui s'applique à des entiers naturels.

$$g(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + g(n-2) & \text{si } n > 0. \end{cases}$$

Cette définition n'est pas correcte car, par exemple, la valeur de $g(1)$ vaut

$$g(1) = 1 + g(-1)$$

mais $g(-1)$ n'a pas de sens puisque cette fonction ne s'applique qu'à des entiers naturels. Comme nous le verrons dans la section suivante, cette définition peut conduire à une exécution infinie ou à une erreur, selon la manière dont elle est écrite en Python.

Enfin, notre dernier exemple est celui d'une fonction $h(n)$ qui s'applique également à des entiers naturels et qui est définie de la manière suivante :

$$h(n) = \begin{cases} 1 & \text{si } n = 0, \\ n + h(n-1) & \text{si } n > 1. \end{cases}$$

Cette définition est incorrecte, puisqu'une valeur est oubliée. L'avez-vous repérée ? [2]

Définition récursive de structures de données. Les techniques de définition récursive peuvent s'appliquer à toute une variété d'objet, et pas seulement à la définition de fonctions. Nous verrons en particulier aux chapitres 6 et 8 des structures de données définies récursivement.

2. C'est la valeur de $h(1)$ qui n'est pas définie.

1.3 Programmer avec des fonctions récursives

Une fois que l'on dispose d'une définition récursive pour une fonction, il est en général assez facile de la programmer en Python. Comme nous l'avons montré pour la fonction $somme(n)$, le code Python correspondant s'obtient d'une manière quasi immédiate en utilisant une conditionnelle pour distinguer les cas de base et les cas récursifs.

Il faut néanmoins faire attention à deux points importants (explicités ci-après). Le premier est que le domaine mathématique d'une fonction, c'est-à-dire les valeurs sur lesquelles elle est définie, n'est pas toujours le même que l'ensemble des valeurs du type Python avec lesquelles elle sera appelée. Le deuxième point est que le choix d'une définition récursive plutôt qu'une autre peut dépendre du modèle d'exécution des fonctions récursives, en particulier quand il s'agit de prendre en compte des contraintes d'efficacité.

Domaine mathématique *vs.* type de données

Le code de la fonction `somme(n)` présenté dans la section précédente, et rappelé ci-dessous à droite, ne se comporte pas exactement comme la fonction mathématique $somme(n)$ définie récursivement, ci-dessous à gauche.

$$somme(n) = \begin{cases} 0 & \text{si } n = 0, \\ n + somme(n-1) & \text{si } n > 0. \end{cases}$$

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

La principale différence est que la fonction mathématique est uniquement définie pour des entiers naturels, alors que la fonction `somme(n)` peut être appelée avec un entier Python arbitraire, qui peut être une valeur négative. Par exemple, bien que la fonction mathématique ne soit pas définie pour $n = -1$, l'appel `somme(-1)` ne provoque aucune erreur immédiate, mais il implique un appel à `somme(-2)`, qui déclenche un appel à `somme(-3)`, etc. Comme on le verra un peu plus loin, ce processus infini va finir par provoquer une erreur à l'exécution.

Pour éviter ce comportement, il y a plusieurs possibilités. La première est de changer le test `n == 0` par `n <= 0`. Cette solution a l'avantage de garantir la terminaison de la fonction, mais elle modifie la spécification de la fonction en renvoyant, de manière arbitraire, la valeur `0` pour chaque appel sur un nombre négatif.

Une autre solution est de restreindre les appels à la fonction `somme(n)` aux entiers positifs ou nuls. Pour cela, on peut utiliser une instruction `assert` [NSI 1re, p. 124] comme ci-dessous.

```
def somme(n):
    assert n >= 0
    if n == 0:
        return 0
    else:
        return n + somme(n - 1)
```

De cette manière, une erreur sera déclenchée pour tout appel à `somme(n)` avec `n < 0`. Bien que cette solution soit correcte, elle n'est pas encore complètement satisfaisante. En effet, pour tout appel `somme(n)` avec `n >= 0`, chaque appel récursif commencera par faire le test associé à l'instruction `assert`, alors que chaque valeur de `n` sera nécessairement positive.

Une solution pour éviter ces tests inutiles est de définir deux fonctions. La première, `somme_bis(n)`, implémente la définition récursive de la fonction mathématique $somme(n)$ sans vérifier son argument.

```
def somme_bis(n):
    if n == 0:
        return 0
    else:
        return n + somme_bis(n - 1)
```

La seconde, `somme(n)`, est la fonction « principale » qui sera appelée par l'utilisateur. Cette fonction ne fait que vérifier (une et une seule fois) que son argument `n` est positif puis, si c'est le cas, elle appelle la fonction `somme_bis(n)`.

```
def somme(n):
    assert n >= 0
    return somme_bis(n)
```

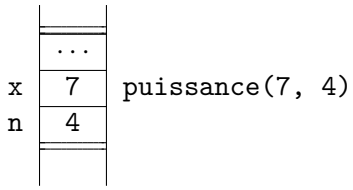
Modèle d'exécution

Comme nous l'avons présenté dans le volume pour la classe de première [NSI 1re, chap. 23], une partie de l'espace mémoire d'un programme est organisée sous forme d'une pile où sont stockés les contextes d'exécution de chaque appel de fonction. Par exemple, pour la fonction `puissance(x,n)` ci-dessous,

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n - 1)
```

l'organisation de la mémoire au début de l'appel à `puissance(7,4)` est représentée par une pile contenant un environnement d'exécution avec, entre

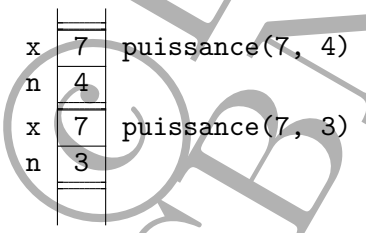
autres, un emplacement pour l'argument `x` initialisé à `7` et un autre pour `n` contenant la valeur `4`.

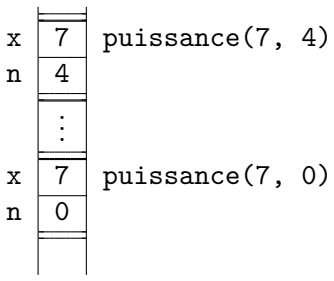


L'environnement contient également d'autres valeurs (comme l'emplacement pour la valeur renvoyée par la fonction, la sauvegarde des registres, etc.) qui sont simplement représentées par des $\cdots$ dans le schéma ci-dessus. Le calcul récursif de `puissance(7,4)` va engendrer une suite d'appels « en cascade » à la fonction `puissance`, que l'on peut représenter par l'arbre d'appels suivant :

```
puissance(7,4) =
 return 7 * puissance(7,3)
             |
          return 7 * puissance(7,2)
                       |
                    return 7 * puissance(7,1)
                                 |
                              return 7 * puissance(7,0)
                                           |
                                        return 1
```

En ce qui concerne l'organisation de la pile mémoire, un environnement d'exécution, similaire à celui décrit ci-dessus, va être alloué sur la pile pour chacun de ces appels. Ainsi, lors de l'évaluation de l'expression `7 * puissance(7,3)`, la pile d'appels contiendra deux environnements, celui pour l'appel à `puissance(7,4)` et, juste en dessous (car l'allocation sur une pile se fait habituellement vers le bas de la mémoire), celui pour l'appel à `puissance(7,3)`, comme décrit dans le schéma de gauche ci-dessous, où nous n'avons indiqué que les emplacements pertinents pour notre propos.





Juste après le dernier appel à `puissance(7,0)`, la pile contient donc les environnements d'exécution pour les cinq appels à la fonction `puissance`, comme représenté par le schéma de droite ci-dessus. Plus généralement, pour un appel initial `puissance(x,n)`, il y aura `n`+1 environnements dans la pile.

Erreurs. Malheureusement, Python limite explicitement le nombre d'appels récursifs dans une fonction. Ainsi, après 1000 appels récursifs, l'interpréteur Python va lever l'exception `RecursionError` et afficher le message d'erreur suivant :

```
RecursionError: maximum recursion depth exceeded.
```

Cette limite, fixée à 1000 appels récursifs, est une valeur par défaut qu'il est possible de modifier en utilisant la fonction `setrecursionlimit` disponible dans le module `sys`. Par exemple, pour passer cette limite à 2000 appels maximum, on exécutera le code Python suivant :

```
import sys
sys.setrecursionlimit(2000)
```

Un tel changement reste cependant dérisoire lorsque l'on a une définition récursive qui, par nature, effectue un très grand nombre d'appels emboîtés.

La récursion dans d'autres langages. Certains langages de programmation, plus spécialisés que Python dans l'écriture de fonctions récursives, savent dans certains cas éviter de placer de trop nombreux environnements d'appel dans la pile. Cela leur permet dans les cas en question de s'affranchir de toute limite relative au nombre d'appels emboîtés. C'est le cas notamment des langages fonctionnels (voir le chapitre 5).

On peut cependant utiliser une autre définition de la fonction mathématique $puissance(x, n)$ qui réduit drastiquement le nombre d'appels récursifs emboîtés. On peut le faire avec deux cas récursifs qui distinguent la parité de n et deux cas de base ($n = 0$ et $n = 1$), comme ci-dessous.

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ carre(puissance(x, n/2)) & \text{si } n > 1 \text{ et } n \text{ est pair}, \\ x \times carre(puissance(x, (n-1)/2)) & \text{si } n > 1 \text{ et } n \text{ est impair}. \end{cases}$$

Une implémentation possible de cette définition en Python est la suivante, où l'appel à la fonction $carre(x)$ est simplement remplacé par la

multiplication `r * r` et où le test de parité est réalisé par un test à zéro du reste de la division entière par 2 (soit `r % 2 == 0`).

```
def puissance(x,n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        r = puissance(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

Cette fonction récursive a en effet l'avantage de diminuer largement le nombre d'appels récursifs. Pour s'en convaincre, prenons l'exemple de l'appel `puissance(7, 28)`. L'arbre des appels (simplifié) représenté ci-dessous montre que seuls quatre appels récursifs sont nécessaires pour effectuer le calcul.

```
puissance(7, 28) =
 r = puissance(7, 14)···return r * r
  r = puissance(7, 7)···return r * r
   r = puissance(7, 3)···return 7 * r * r
    r = puissance(7, 1)···return 7 * r * r
     return 7
```

D'une manière générale, il faut $1 + \lfloor log_2(n) \rfloor$ appels pour calculer `puissance(x, n)` avec cette définition, c'est-à-dire un appel initial et $\lfloor log_2(n) \rfloor$ appels récursifs. Ainsi, le calcul de `puissance(x, 1000)` ne nécessite que $1 + \lfloor log_2(1000) \rfloor = 10$ appels.

À retenir. Un calcul peut être décrit à l'aide d'une **définition récursive**. L'avantage de cette technique est que l'implémentation est souvent plus proche de la définition. L'écriture d'une **fonction récursive** nécessite de distinguer les **cas de base**, pour lesquels on peut donner un résultat facilement, et les **cas récursifs**, qui font appel à la définition en cours. Il faut faire attention à ce que la fonction en Python ne s'applique que sur le **domaine** de la fonction mathématique, par exemple en utilisant l'instruction `assert`. Enfin, il faut comprendre le modèle d'exécution des fonctions récursives pour choisir la définition qui **limite** le nombre d'appels récursifs.

Exercices

Exercice 1 Donner une définition récursive qui correspond au calcul de la fonction factorielle $n!$ définie par $n! = 1 \times 2 \times \cdots \times n$ si $n > 0$ et $0! = 1$, puis le code d'une fonction `fact(n)` qui implémente cette définition.

Solution page 427 □

Exercice 2 Soit u_n la suite d'entiers définie par

$$\begin{array}{rcll} u_{n+1} & = & u_n/2 & \text{si } u_n \text{ est pair,} \\ & = & 3 \times u_n + 1 & \text{sinon.} \end{array}$$

avec u_0 un entier quelconque plus grand que 1.

Écrire une fonction récursive `syracuse(u_n)` qui affiche les valeurs successives de la suite u_n tant que u_n est plus grand que 1.

La conjecture de Syracuse affirme que, quelle que soit la valeur de u_0, il existe un indice n dans la suite tel que $u_n = 1$. Cette conjecture défie toujours les mathématiciens.

Solution page 427 □

Exercice 3 On considère la suite u_n définie par la relation de récurrence suivante, où a et b sont des réels quelconques :

$$u_n = \begin{cases} a \in \mathbb{R} & \text{si } n = 0 \\ b \in \mathbb{R} & \text{si } n = 1 \\ 3u_{n-1} + 2u_{n-2} + 5 & \forall n \geq 2 \end{cases}$$

Écrire une fonction récursive `serie(n, a, b)` qui renvoie le `n`-ème terme de cette suite pour des valeurs `a` et `b` données en paramètres.

Solution page 427 □

Exercice 4 Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers entre `i` et `k`. Par exemple, `boucle(0,3)` doit afficher `0 1 2 3`.

Solution page 428 □

Exercice 5 Écrire une fonction récursive `pgcd(a, b)` qui renvoie le PGCD de deux entiers `a` et `b`.

Solution page 428 □

Exercice 6 Écrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier positif ou nul `n` en argument et renvoie son nombre de chiffres. Par exemple, `nombre_de_chiffres(34126)` doit renvoyer `5`.

Solution page 428 □

Exercice 7 En s'inspirant de l'exercice 6, écrire une fonction récursive `nombre_de_bits_1(n)` qui prend un entier positif ou nul et renvoie le nombre de bits valant `1` dans la représentation binaire de `n`. Par exemple, `nombre_de_bits_1(255)` doit renvoyer `8`.

Solution page 428 □

Exercice 8 Écrire une fonction récursive `appartient(v, t, i)` prenant en paramètres une valeur `v`, un tableau `t` et un entier `i` et renvoyant `True` si `v` apparaît dans `t` entre l'indice `i` (inclus) et **`len`**`(t)` (exclu), et `False` sinon. On supposera que `i` est toujours compris entre `0` et **`len`**`(t)`.

Solution page 428 □

Exercice 9 Le triangle de Pascal (nommé ainsi en l'honneur du mathématicien Blaise Pascal) est une présentation des coefficients binomiaux sous la forme d'un triangle défini ainsi de manière récursive :

$$C(n,p) = \begin{cases} 1 & \text{si } p = 0 \text{ ou } n = p, \\ C(n-1,p-1) + C(n-1,p) & \text{sinon.} \end{cases}$$

Écrire une fonction récursive `C(n,p)` qui renvoie la valeur de $C(n,p)$, puis dessiner le triangle de Pascal à l'aide d'une double boucle **`for`** en faisant varier `n` entre 0 et 10.

Solution page 429 □

Exercice 10 La courbe de Koch est une figure qui s'obtient de manière récursive. Le cas de base à l'ordre 0 de la récurrence est simplement le dessin d'un segment d'une certaine longueur l, comme ci-dessous (figure de gauche).

Le cas récursif d'ordre n s'obtient en divisant ce segment en trois morceaux de même longueur $l/3$, puis en dessinant un triangle équilatéral dont la base est le morceau du milieu, en prenant soin de ne pas dessiner cette base. Cela forme une sorte de chapeau comme dessiné sur la figure de droite ci-dessus. On réitère ce processus à l'ordre $n-1$ pour chaque segment de ce chapeau (qui sont tous de longueur $l/3$). Par exemple, les courbes obtenues à l'ordre 2 et 3 sont données ci-dessous (à gauche et à droite, respectivement).

Écrire une fonction `koch(n, l)` qui dessine avec `Turtle` un flocon de Koch de profondeur `n` à partir d'un segment de longueur `l`.

Solution page 429 □

Erreurs. Considérons la tentative de réponse suivante à l'exercice 8 page 17, à savoir l'écriture d'une fonction récursive testant la présence d'une valeur `v` dans un tableau `t` à un indice supérieur ou égal à `i`.

```
def appartient(v, t, i):
    if i == len(t):
        return False
    elif t[i] == v:
        return True
    else:
        appartient(v, t, i + 1)
```

Si `i` a dépassé le dernier indice du tableau, alors la fonction renvoie `False`. Si au contraire la case d'indice `i` contient la valeur cherchée, alors la fonction renvoie `True`. Dans les autres cas enfin, la fonction procède à un appel récursif poursuivant la recherche à partir de l'indice `i + 1`. Cette fonction donnera parfois les bons résultats.

```
>>> appartient(3, [], 0)
False
>>> appartient(3, [3], 0)
True
```

Elle renverra cependant le plus souvent `None`.

```
>>> appartient(3, [1, 2, 3, 4], 0)
None
>>> appartient(3, [1, 2], 0)
None
```

Ceci tient à l'oubli du mot-clé **return** dans le troisième cas. En effet, en Python toute fonction doit utiliser **return** à chaque endroit où elle renvoie un résultat, et cela vaut y compris lorsque ce résultat n'est que la transmission du résultat d'un appel récursif. Sans **return**, l'appel récursif est effectué mais son résultat est oublié sitôt obtenu et la fonction en définitive ne renvoie rien, ou plutôt elle renvoie `None`, c'est-à-dire la valeur traduisant habituellement l'absence de résultat. Cette erreur peut être délicate à détecter si notre fonction `appartient` n'est pas testée ainsi de manière isolée, puisque dans un branchement tel que

```
if appartient(v, t, i):
```

l'obtention de la valeur `None` serait interprétée comme `False` et ne provoquerait pas d'erreur immédiate.

Chapitre 6

Listes chaînées

Notions introduites

- structure de liste chaînée
- opérations usuelles sur les listes chaînées
- encapsulation dans un objet

La structure de tableau permet de stocker des séquences d'éléments mais n'est pas adaptée à toutes les opérations que l'on pourrait vouloir effectuer sur des séquences. Les tableaux de Python permettent par exemple d'insérer ou de supprimer efficacement des éléments à la fin d'un tableau, avec les opérations `append` et `pop`, mais se prêtent mal à l'insertion ou la suppression d'un élément à une autre position [1]. En effet, les éléments d'un tableau étant contigus et ordonnés en mémoire, insérer un élément dans une séquence demande de déplacer tous les éléments qui le suivent pour lui laisser une place.

Si par exemple on veut insérer une valeur `v` à la première position d'un tableau

1	1	2	3	5	8	13

il faut d'une façon ou d'une autre construire le nouveau tableau

v	1	1	2	3	5	8	13

dans lequel la case d'indice 0 contient maintenant la valeur `v`. On peut le faire facilement en utilisant l'opération `insert` des tableaux de Python.

```
t.insert(0, v)
```

1. Et lorsque l'on s'intéresse à d'autres langages que Python dans lesquels les tableaux ne sont pas *redimensionnables*, même les opérations `append` et `pop` ne sont pas évidentes.

Cette opération est cependant très coûteuse, car elle déplace *tous* les éléments du tableau d'une case vers la droite après avoir agrandi le tableau. C'est exactement comme si nous avions écrit les lignes suivantes :

```
t.append(None)
for i in range(len(t) - 1, 0, -1):
    t[i] = t[i - 1]
t[0] = v
```

Avec une telle opération on commence donc par agrandir le tableau, en ajoutant un nouvel élément à la fin avec `append`.

1	1	2	3	5	8	13	None

Puis on décale tous les éléments d'une case vers la droite, en prenant soin de commencer par le dernier et de terminer par le premier.

1	1	1	2	3	5	8	13

Enfin, on écrit la valeur `v` dans la première case du tableau.

v	1	1	2	3	5	8	13

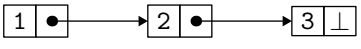
Au total, on a réalisé un nombre d'opérations proportionnel à la taille du tableau. Si par exemple le tableau contient un million d'éléments, on fera un million d'opérations pour ajouter un premier élément. En outre, supprimer le premier élément serait tout aussi coûteux, pour les mêmes raisons.

Dans ce chapitre nous étudions une structure de données, la *liste chaînée*, qui d'une part apporte une meilleure solution au problème de l'insertion et de la suppression au début d'une séquence d'éléments, et d'autre part servira de brique de base à plusieurs autres structures dans les prochains chapitres.

6.1 Structure de liste chaînée

Une *liste chaînée* permet avant tout de représenter une liste, c'est-à-dire une séquence finie de valeurs, par exemple des entiers. Comme le nom le suggère sa structure est en outre caractérisée par le fait que les éléments sont chaînés entre eux, permettant le passage d'un élément à l'élément suivant.

Ainsi, chaque élément est stocké dans un petit bloc alloué quelque part dans la mémoire, que l'on pourra appeler maillon ou *cellule*, et y est accompagné d'une deuxième information : l'adresse mémoire où se trouve la cellule contenant l'élément suivant de la liste.



Ici, on a illustré une liste contenant trois éléments, respectivement 1, 2 et 3. Chaque élément de la liste est matérialisé par un emplacement en mémoire

Programme 18 — Cellule d'une liste chaînée

```
class Cellule:
    """une cellule d'une liste chaînée"""
    def __init__(self, v, s):
        self.valeur = v
        self.suivante = s
```

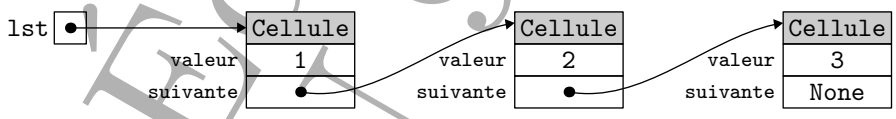
contenant d'une part sa valeur (dans la case de gauche) et d'autre part l'adresse mémoire de la valeur suivante (dans la case de droite). Dans le cas du dernier élément, qui ne possède pas de valeur suivante, on utilise une valeur spéciale désignée ici par le symbole $\perp$ et marquant la fin de la liste [2].

Une façon traditionnelle de représenter une liste chaînée en Python consiste à utiliser une classe décrivant les cellules de la liste, de sorte que chaque élément de la liste est matérialisé par un objet de cette classe. Cette classe est appelée ici `Cellule` et est donnée dans le programme 18. Tout objet de cette classe contient deux attributs : un attribut `valeur` pour la valeur de l'élément (l'entier, dans notre exemple) ; et un attribut `suivante` pour la cellule suivante de la liste. Lorsqu'il n'y a pas de cellule suivante, c'est-à-dire lorsque l'on considère la dernière cellule de la liste, on donne à l'attribut `suivante` la valeur `None`. Dit autrement, `None` est notre représentation du symbole $\perp$.

Pour construire une liste, il suffit d'appliquer le constructeur de la classe `Cellule` autant de fois qu'il y a d'éléments dans la liste. Ainsi, l'instruction

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
```

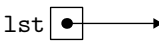
construit la liste 1,2,3 donnée en exemple plus haut et la stocke dans une variable `lst`. Plus précisément, on a ici créé trois objets de la classe `Cellule`, que l'on peut visualiser comme suit.



La valeur contenue dans la variable `lst` est l'adresse mémoire de l'objet contenant la valeur 1, qui lui-même contient dans son attribut `suivante` l'adresse mémoire de l'objet contenant la valeur 2, qui enfin contient dans son attribut `suivante` l'adresse mémoire de l'objet contenant la valeur 3. Ce dernier contient la valeur `None` dans son attribut `suivante`, marquant

2. Le symbole $\perp$, dont le nom officiel est « taquet vers le haut », est utilisé pour désigner plusieurs choses en mathématiques et informatique. Les logiciens, par exemple, l'utilisent pour désigner la contradiction. En anglais, on le désigne parfois sous le nom de *bottom*.

ainsi la fin de la liste. Par la suite, on s'autorisera un dessin simplifié, de la manière suivante.

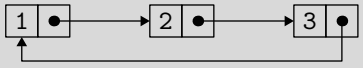


Dans ce dessin, il faut interpréter chaque élément de la liste comme un objet de la classe `Cellule`.

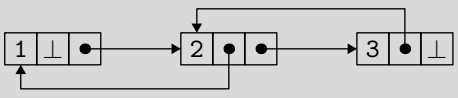
Définition récursive des listes chaînées. Comme on le voit, une liste est soit la valeur `None`, soit un objet de la classe `Cellule` dont l'attribut `suivante` contient une liste. C'est là une *définition récursive* de la notion de liste.

Représentations alternatives. D'autres représentations des listes chaînées sont possibles. Plutôt qu'un objet de la classe `Cellule`, on pourrait utiliser un couple, et dans ce cas écrire `(1,(2,(3,None)))`, ou encore un tableau à deux éléments, et dans ce cas écrire `[1,[2,[3,None]]]`. Cependant, l'utilisation d'une valeur structurée avec des champs nommés (ici les attributs `valeur` et `suivante`) est idiomatique, y compris dans un langage comme Python.

Variantes des listes chaînées. Il existe de nombreuses variantes de la structure de liste chaînée, dont la *liste cyclique*, où le dernier élément est lié au premier,



ou la *liste doublement chaînée*, où chaque élément est lié à l'élément suivant *et* à l'élément précédent dans la liste,



ou encore la liste cyclique doublement chaînée qui combine ces deux variantes.
Dans tout ce chapitre, on ne manipule que des listes simplement chaînées et ne contenant pas de cycles.

Homogénéité. Bien que nous illustrions ce chapitre avec des listes d'entiers, les listes chaînées, au même titre que les tableaux Python, peuvent contenir des valeurs de n'importe quel type. Ainsi, on peut imaginer des listes de chaînes de caractères, de couples, etc. Comme pour les tableaux, nous recommandons une utilisation *homogène* des listes chaînées, où tous les éléments de la liste sont du même type.

6.2 Opérations sur les listes

Dans cette section, nous allons programmer quelques opérations fondamentales sur les listes. D'autres opérations sont proposées en exercices.

Longueur d'une liste

La première opération que nous allons programmer consiste à calculer la longueur d'une liste chaînée, c'est-à-dire le nombre de cellules qu'elle contient. Il s'agit donc de *parcourir* la liste, de la première cellule jusqu'à la dernière, en suivant les liens qui relient les cellules entre elles. On peut réaliser ce parcours, au choix, avec une fonction récursive ou avec une boucle. Nous allons faire les deux. Dans les deux cas, on écrit une fonction `longueur` qui reçoit une liste `lst` en argument et renvoie sa longueur.

```
def longueur(lst):
    """renvoie la longueur de la liste lst"""
```

Commençons par la version récursive. Elle consiste à distinguer le cas de base, c'est-à-dire une liste vide ne contenant aucune cellule, du cas général, c'est-à-dire une liste contenant au moins une cellule. Dans le premier cas, il suffit de renvoyer `0` :

```
    if lst is None:
        return 0
```

Ici, on a testé si la liste `lst` est égale à `None` avec l'opération `is` de Python mais on aurait tout aussi bien pu utiliser `==`, c'est-à-dire écrire `lst == None` (voir encadré).

Dans le second cas, il faut renvoyer `1`, pour la première cellule, plus la longueur du reste de la liste, c'est-à-dire la longueur de la liste `lst.suivante`, que l'on peut calculer récursivement :

```
    else:
        return 1 + longueur(lst.suivante)
```

On se persuade facilement que cette fonction termine, car le nombre de cellules de la liste passée en argument à la fonction `longueur` décroît strictement à chaque appel.

Écrivons maintenant la fonction `longueur` différemment, cette fois avec une boucle. On commence par se donner deux variables : une variable `n` contenant la longueur que l'on calcule et une variable `c` contenant la cellule courante du parcours de la liste.

```
def longueur(lst):
    """renvoie la longueur de la liste lst"""
    n = 0
    c = lst
```

Initialement, `n` vaut 0 et `c` prend la valeur de `lst`, c'est-à-dire `None` si la liste est vide et la première cellule sinon. Le parcours est ensuite réalisé avec une boucle `while`, qui exécute autant d'itérations qu'il y a de cellules dans la liste.

```
    while c is not None:
```

L'opération `is not` est, comme on le devine, la négation de l'opération `is`. On exécuter donc cette boucle tant que `c` n'est pas égale à `None`.

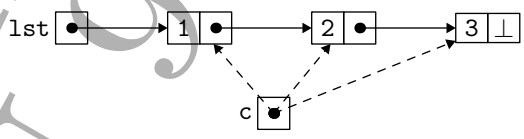
À chaque étape, c'est-à-dire pour chaque cellule de la liste, on incrémente le compteur `n` et on passe à la cellule suivante en donnant à `c` la valeur de `c.suivante`.

```
        n += 1
        c = c.suivante
```

Une fois que l'on sort de la boucle, il suffit de renvoyer la valeur de `n`.

```
    return n
```

Il est important de comprendre que, dans cette version itérative, seule la *variable* `c` est modifiée, pour désigner successivement les différentes cellules de la liste[3] :



L'affectation `c = c.suivante` ne modifie pas le contenu ou la structure de la liste, seulement le contenu de la variable `c`, qui est l'adresse d'une cellule de la liste.

Le code complet de ces deux fonctions `longueur` est donné dans le programme 19.

3. Nous aurions pu également nous servir de la variable `lst` à la place de la variable `c`, car la variable `lst` est locale à la fonction `longueur`. Mais, de manière générale, nous évitons de modifier les variables qui correspondent aux arguments d'une fonction.

Programme 19 — Calcul de la longueur d'une liste

```
# avec une fonction récursive
def longueur(lst):
    """renvoie la longueur de la liste lst"""
    if lst is None:
        return 0
    else:
        return 1 + longueur(lst.suivante)

# avec une boucle
def longueur(lst):
    """renvoie la longueur de la liste lst"""
    n = 0
    c = lst
    while c is not None:
        n += 1
        c = c.suivante
    return n
```

Comparaison avec `None`. A priori, il n'y a pas de différence entre écrire `lst is None` et `lst == None`. Les deux opérations ne sont pas exactement les mêmes : l'opération `is` est une égalité physique (être identiquement le même objet, au même endroit dans la mémoire) et l'opération `==` est une égalité structurelle (être la même valeur, après une comparaison en profondeur). Mais dans le cas particulier de la valeur `None`, ces deux égalités coïncident, car l'objet qui représente `None` est unique. On teste donc bien si la valeur de `lst` est `None` dans les deux cas.
Cependant, une classe peut redéfinir l'égalité représentée par l'opération `==` et, dans ce cas, potentiellement modifier le résultat d'une comparaison avec `None`. Pour cette raison, il est d'usage de tester l'égalité à `None` avec `is` plutôt qu'avec `==`. Bien évidemment, dans le cas précis de notre propre classe `Cellule`, nous savons qu'elle ne redéfinit pas l'opération `==`. Néanmoins, nous choisissons de nous conformer à cette bonne pratique.

Complexité. Il est clair que la complexité du calcul de la longueur est directement proportionnelle à la longueur elle-même, puisqu'on réalise un nombre constant d'opérations pour chaque cellule de la liste. Ainsi, pour

une liste `lst` de mille cellules, `longueur(lst)` va effectuer mille tests, mille appels récursifs et mille additions dans sa version récursive, et mille tests, mille additions et deux mille affectations dans sa version itérative.

N-ième élément d'une liste

Comme deuxième opération sur les listes, écrivons une fonction qui renvoie le n-ième élément d'une liste chaînée. On prend la convention que le premier élément est désigné par $n = 0$, comme pour les tableaux. On cherche donc à écrire une fonction de la forme suivante.

```
def nieme_element(n, lst):
    """renvoie le n-ième élément de la liste lst
       les éléments sont numérotés à partir de 0"""
```

Comme pour la fonction `longueur`, nous avons le choix entre écrire la fonction `nieme_element` comme une fonction récursive ou avec une boucle. Nous faisons ici le choix d'une fonction récursive ; l'exercice 50 page 124 propose d'écrire cette fonction avec une boucle.

Comme pour le calcul de la longueur, nous commençons par traiter le cas d'une liste qui ne contient aucun élément. Dans ce cas, on choisit de lever une exception (voir chapitre 4), en l'occurrence la même exception `IndexError` que celle levée par Python lorsque l'on tente d'accéder à un indice invalide d'un tableau.

```
    if lst is None:
        raise IndexError("indice invalide")
```

La chaîne de caractères passée en argument de l'exception est arbitraire.

Si en revanche la liste `lst` n'est pas vide, il y a deux cas de figure à considérer. Si `n` $= 0$, c'est que l'on demande le premier élément de la liste et il est alors renvoyé.

```
    if n == 0:
        return lst.valeur
```

Sinon, il faut continuer la recherche dans le reste de la liste. Pour cela, on fait un appel récursif à `nieme_element` en diminuant de un la valeur de `n`.

```
    else:
        return nieme_element(n - 1, lst.suivante)
```

Attention à ne pas oublier ici l'instruction `return`, car il faut *renvoyer* le résultat de l'appel récursif et non pas se contenter de *faire* un appel récursif. Ceci achève le code de la fonction `nieme_element`. Le code complet est donné dans le programme 20.

Programme 20 — N-ième élément d'une liste

```
def nieme_element(n, lst):
    """renvoie le n-ième élément de la liste lst
       les éléments sont numérotés à partir de 0"""
    if lst is None:
        raise IndexError("indice invalide")
    if n == 0:
        return lst.valeur
    else:
        return nieme_element(n - 1, lst.suivante)
```

Complexité. La complexité de la fonction `nieme_element` est un peu plus subtile que celle de la fonction `longueur`. Dans certains cas, on effectue exactement `n` appels récursifs pour trouver le `n` $-$ *ième* élément, et donc un nombre d'opérations proportionnel à `n`.

Dans d'autres cas, en revanche, on parcourt toute la liste. Cela se produit clairement lorsque `n` $\geq$ `longueur(lst)`. Il pourrait être tentant de commencer par comparer `n` avec la longueur de la liste, pour ne pas parcourir la liste inutilement, mais c'est inutile car le calcul de la longueur parcourt déjà toute la liste. Pire encore, calculer la longueur de la liste à chaque appel récursif résulterait en un programme de complexité quadratique (proportionnelle au carré de la longueur de la liste).

On peut remarquer que la liste est également intégralement parcourue lorsque `n` < 0. En effet, la valeur de `n` va rester strictement négative, puisqu'on la décrémente à chaque appel, et on finira par atteindre la liste vide. Pour y remédier, il suffit de modifier légèrement le premier test de la fonction, de la manière suivante :

```
if n < 0 or lst is None:
    raise IndexError("indice invalide")
```

On obtient exactement le même comportement qu'auparavant (la levée de l'exception `IndexError`) mais cela se fait maintenant en temps constant, car la liste n'est plus parcourue.

Concaténation de deux listes

Considérons maintenant l'opération consistant à mettre bout à bout les éléments de deux listes données. On appelle cela la *concaténation* de deux listes. Ainsi, si la première liste contient $1, 2, 3$ et la seconde $4, 5$ alors le résultat de la concaténation est la liste $1, 2, 3, 4, 5$. Nous choisissons d'écrire

la concaténation sous la forme d'une fonction `concatener` qui reçoit deux listes en arguments et renvoie une troisième liste contenant la concaténation.

```
def concatener(l1, l2):
    """concatène les listes l1 et l2,
       sous la forme d'une nouvelle liste"""
```

Il est ici aisé de procéder récursivement sur la structure de la liste `l1`. Si elle est vide, la concaténation est identique à la liste `l2`, qu'on se contente donc de renvoyer.

```
    if l1 is None:
        return l2
```

Sinon, le premier élément de la concaténation est le premier élément de `l1` et le reste de la concaténation est obtenu récursivement en concaténant le reste de `l1` avec `l2`.

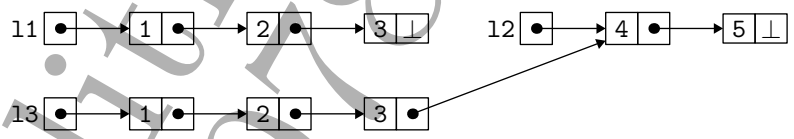
```
    else:
        return Cellule(l1.valeur, concatener(l1.suivante,l2))
```

Le programme 21 contient l'intégralité du code.

Il est important de comprendre ici que les listes passées en argument à la fonction `concatener` ne sont pas modifiées. Plus précisément, les éléments de la liste `l1` sont copiés et ceux de `l2` sont partagés. Illustrons-le avec la concaténation des listes $1, 2, 3$ et $4, 5$. Après les trois instructions

```
l1 = Cellule(1, Cellule(2, Cellule(3, None)))
l2 = Cellule(4, Cellule(5, None))
l3 = concatener(l1, l2)
```

on a la situation suivante, avec huit cellules au total :



On voit que les trois cellules de `l1` ont été dupliquées, pour former le début de la liste $1, 2, 3, 4, 5$, et que les deux cellules de `l2` sont partagées pour former à la fois la liste `l2` et la fin de la liste `l3`. Il n'y a pas de danger à réaliser ainsi un tel partage de cellules, tant qu'on ne cherche pas à modifier les listes. Une alternative consisterait à copier également tous les éléments de `l2`, ce qui pourrait se faire en écrivant une fonction `copie` et en remplaçant **return** `l2` par **return** `copie(l2)`. Mais c'est inutile dès lors qu'on choisit de ne jamais modifier les listes une fois construites.

Dans la section suivante, nous discuterons d'une autre façon de réaliser la concaténation de deux listes `l1` et `l2`, consistant à modifier la dernière cellule de la liste `l1` pour la faire pointer vers la première cellule de la liste `l2`. Mais nous mettrons également en garde contre les dangers que comporte une telle modification.

Programme 21 — Concaténation de deux listes

```
def concatener(l1, l2):
    """concatène les listes l1 et l2,
       sous la forme d'une nouvelle liste"""
    if l1 is None:
        return l2
    else:
        return Cellule(l1.valeur, concatener(l1.suivante,l2))
```

Complexité. Il est clair que le coût de la fonction `concatener` est directement proportionnel à la longueur de la liste `l1`. En revanche, il ne dépend pas de la longueur de la liste `l2`.

Renverser une liste

Comme quatrième et dernière opération sur les listes, considérons le renversement d'une liste, c'est-à-dire une fonction `renverser` qui, recevant en argument une liste comme $1, 2, 3$, renvoie[4] la liste renversée $3, 2, 1$.

Vu que la récursivité a été utilisée avec succès pour écrire les trois opérations précédentes, il semble naturel de chercher une écriture récursive de la fonction `renverser`. Le cas de base est celui d'une liste vide, pour laquelle il suffit de renvoyer la liste vide. Pour le cas récursif, en revanche, c'est plus délicat, car le premier élément doit devenir le dernier élément de la liste renversée. Aussi, il faut renverser la queue de la liste puis *concaténer* à la fin le tout premier élément. Vu que nous venons justement d'écrire une fonction `concatener`, il n'y a qu'à s'en servir. Cela nous donne un code relativement simple pour la fonction `renverser`.

```
def renverser(lst):
    if lst is None:
        return None
    else:
        return concatener(renverser(lst.suivante),
                          Cellule(lst.valeur, None))
```

Un tel code, cependant, est particulièrement inefficace. Si on en mesure le temps d'exécution, on s'aperçoit qu'il est proportionnel au *carré* du nombre d'éléments. Pour renverser une liste de 1000 éléments, il faut près d'un demi-million d'opérations. En effet, il faut commencer par renverser une liste de

4. On comprend ici à quel point l'usage de l'anglicisme « retourner » à la place de « renvoyer » peut être source de confusion [NSI 1re, p. 62].

999 éléments, puis concaténer le résultat avec une liste d'un élément. Comme on l'a vu, cette concaténation coûte 999 opérations. Et pour renverser la liste de 999 éléments, il faut renverser une liste de 998 éléments puis concaténer le résultat avec une liste d'un élément. Et ainsi de suite. On total, on a donc au moins $999 + 998 + \cdots + 1 = 499\,500$ opérations.

Une fois n'est pas coutume, la récursivité nous a mis sur la piste d'une mauvaise solution, du moins en termes de performance. Il se trouve que dans le cas de la fonction `renverser`, une boucle `while` est plus adaptée. En effet, il suffit de parcourir les éléments de la liste `lst` avec une simple boucle, et d'ajouter ses éléments au fur et à mesure *en tête* d'une seconde liste, appelons-la `r`. Ainsi, le premier élément de la liste `lst` se retrouve en dernière position dans la liste `r`, le deuxième élément de `lst` en avant-dernière position dans `r`, etc., jusqu'au dernier élément de `lst` qui se retrouve en première position dans `r`. Une bonne image est celle d'une pile de feuilles de papier sur notre bureau : si on prend successivement chaque feuille au sommet de la pile pour former à côté une seconde pile, alors on aura inversé l'ordre des feuilles au final. Ici, la liste `lst` joue le rôle de la première pile et la liste `r` celle de la seconde.

Le code qui met en œuvre cette idée est relativement simple. On commence par se donner deux variables : une variable `r` pour le résultat et une variable `c` pour parcourir la liste `lst`.

```
def renverser(lst):
    r = None
    c = lst
```

On parcourt ensuite la liste avec une boucle `while`, en ajoutant à chaque étape le premier élément de `c` en tête de la liste `r`, avant de passer à l'élément suivant.

```
    while c is not None:
        r = Cellule(c.valeur, r)
        c = c.suivante
```

Enfin, il ne reste plus qu'à renvoyer la liste `r` une fois sorti de la boucle.

```
    return r
```

Le programme 22 contient l'intégralité du code.

Complexité. Il est clair que cette nouvelle fonction `renverser` a un coût directement proportionnel à la longueur de la liste `lst`, car le code fait un simple parcours de la liste, avec deux opérations élémentaires à chaque étape. Ainsi, renverser une liste de 1000 éléments devient presque instantané, avec un millier d'opérations, là où notre fonction basée sur la concaténation utilisait un demi-million d'opérations. L'exercice 53 propose d'écrire une fonction récursive de même complexité que le programme 22.

Programme 22 — Renverser une liste

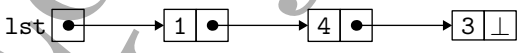
```
def renverser(lst):
    """renvoie une liste contenant les éléments
       de lst dans l'ordre inverse"""
    r = None
    c = lst
    while c is not None:
        r = Cellule(c.valeur, r)
        c = c.suivante
    return r
```

6.3 Modification d'une liste

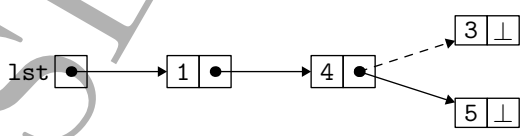
Jusqu'à présent, nous avons délibérément choisi de ne jamais modifier les deux attributs `valeur` et `suivante` d'un objet de la classe `Cellule`. Une fois qu'un tel objet est construit, il n'est plus jamais modifié. Cependant, rien ne nous empêcherait de le faire, intentionnellement ou accidentellement, car il reste toujours possible de modifier la valeur de ces attributs *a posteriori* avec des affectations.

Reprenons l'exemple de la liste 1,2,3 du début de ce chapitre, construite avec `lst = Cellule(1, Cellule(2, Cellule(3, None)))` et que nous représentons ainsi :

Il est très facile de modifier la valeur du deuxième élément de la liste, avec une simple affectation comme `lst.suivante.valeur = 4`. On se retrouve alors avec la situation suivante



c'est-à-dire avec la liste 1,4,3. Ici, on vient de modifier le *contenu* de la liste, en modifiant un attribut `valeur`. Mais on peut également modifier la *structure* de la liste, en modifiant un attribut `suivante`. Si par exemple on réalise maintenant l'affectation `lst.suivante.suivante = Cellule(5, None)` alors on se retrouve avec la situation suivante :



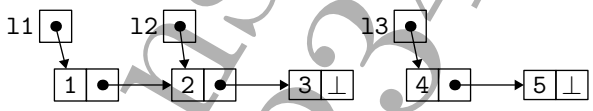
Ici, on a représenté que l'attribut `suivante` du deuxième élément pointait anciennement vers l'élément 3 (en pointillés) et qu'il pointe désormais vers un nouvel élément 5. La variable `lst` contient maintenant la liste 1,4,5 [5].

Du danger des listes mutables

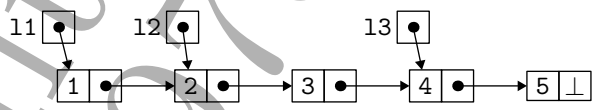
Puisque les listes peuvent être modifiées *a posteriori*, comme nous venons de l'expliquer, il peut être tentant d'en tirer profit pour écrire autrement certaines de nos opérations sur les listes. Ainsi, pour réaliser la concaténation de deux listes, par exemple, il suffit de modifier l'attribut `suivante` du dernier élément de la première liste pour lui donner la valeur de la seconde liste. (Les exercices 58 et 59 proposent de le faire.) Cela semble une bonne idée. Mais il y a un risque. Supposons que l'on construise deux listes 1,2,3 et 4,5 de la manière suivante :

```
l2 = Cellule(2, Cellule(3, None))
l1 = Cellule(1, l2)
l3 = Cellule(4, Cellule(5, None))
```

On note en particulier que la variable `l2` contient toujours la liste 2,3, même si elle a servi depuis à construire la liste 1,2,3 stockée dans la variable `l1`.



S'il nous prend maintenant l'envie de concaténer les listes `l1` et `l3` en reliant le dernier élément de `l1` au premier élément de `l3`, par exemple en appelant une hypothétique fonction `concatener_en_place(l1, l3)` qui ferait cela, alors on se retrouverait dans cette situation :



La variable `l1` contient maintenant la liste 1,2,3,4,5, ce qui était recherché, mais la variable `l2` ne contient plus la liste 2,3 mais la liste 2,3,4,5. C'est là un *effet de bord* qui n'était peut-être pas du tout souhaité. D'une manière générale, pouvoir accéder à une même donnée par deux chemins différents n'est pas un problème en soi, mais modifier ensuite la donnée par l'intermédiaire de l'un de ces chemins (ici `l1`) peut résulter en une modification non souhaitée de la valeur accessible par un autre chemin (ici `l2`). Par ailleurs, que feraient deux appels supplémentaires à `concatener_en_place(l1, l3)` ?

C'est pourquoi nous avons privilégié une approche où la concaténation, et plus généralement les opérations qui construisent des listes, renvoient de

5. L'élément 3 ne fait plus partie de la liste. Il sera récupéré par le GC (voir encadré page 126, et [NSI 1re, p. 296]), s'il n'est pas utilisé dans une autre liste.

nouvelles listes plutôt que de modifier leurs arguments. On peut remarquer que c'est là une approche également suivie par certaines constructions de Python. L'opération `+` de Python, par exemple, ne modifie pas ses arguments mais renvoie une nouvelle valeur, qu'il s'agisse d'entiers, de chaînes de caractères ou encore de tableaux. Ainsi, si `t` est le tableau `[1, 2]`, alors `t + [3]` construit un *nouveau* tableau `[1, 2, 3]`. En ce sens, l'opération `+` se distingue d'autres opérations, comme `.append`, qui modifient leur argument.

Comme nous allons le voir dans la section suivante, cela ne nous empêche pas pour autant d'utiliser nos listes dans un style de programmation impératif.

6.4 Encapsulation dans un objet

Pour terminer ce chapitre sur les listes chaînées, nous allons maintenant montrer comment encapsuler une liste chaînée dans un objet, pour mettre en œuvre les idées présentées dans les chapitres 3 et 2. L'idée consiste à définir une nouvelle classe, `Liste`, qui possède un unique attribut, `tete`, qui contient une liste chaînée. On l'appelle `tete` car il désigne la tête de la liste, lorsque celle-ci n'est pas vide (et `None` sinon). Le constructeur initialise l'attribut `tete` avec la valeur `None`.

```
class Liste:
    """une liste chaînée"""

    def __init__(self):
        self.tete = None
```

Autrement dit, un objet construit avec `Liste()` représente une liste vide. On peut également introduire une méthode `est_vide` qui renvoie un booléen indiquant si la liste est vide.

```
    def est_vide(self):
        return self.tete is None
```

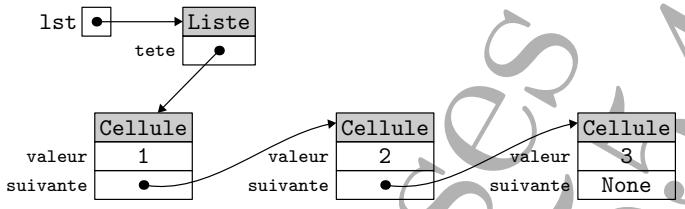
En effet, notre intention est d'encapsuler, c'est-à-dire de cacher, la représentation de la liste derrière cet objet. Pour cette raison, on ne souhaite pas que l'utilisateur de la classe `Liste` teste explicitement si l'attribut `tete` vaut `None`, mais qu'il utilise cette méthode `est_vide`.

On poursuit la construction de la classe `Liste` avec une méthode pour ajouter un élément en tête de la liste.

```
    def ajoute(self, x):
        self.tete = Cellule(x, self.tete)
```

Cette méthode modifie l'attribut `tete` et ne renvoie rien. Si par exemple on exécute les quatre instructions à gauche, on obtient la situation représentée à droite :

```
lst = Liste()
lst.ajoute(3)
lst.ajoute(2)
lst.ajoute(1)
```



On a donc construit ainsi la liste $1, 2, 3$, dans cet ordre.

On peut maintenant reformuler nos opérations, à savoir `longueur`, `nieme_element`, `concatener` ou encore `renverser`, comme autant de méthodes de la classe `Liste`. Ainsi, on peut écrire par exemple

```
def longueur(self):
    return longueur(self.tete)
```

qui ajoute à la classe `Liste` une méthode `longueur`, qui nous permet d'écrire `lst.longueur()` pour obtenir la longueur de la liste `lst`. Il est important de noter qu'il n'y a pas confusion ici entre la *fonction* `longueur` définie précédemment et la *méthode* `longueur`. En particulier, la seconde est définie en appelant la première. Le langage Python est ainsi fait que, lorsqu'on écrit `longueur(self.tete)`, il ne s'agit pas d'un appel récursif à la méthode `longueur`. (Un appel récursif s'écrirait `self.longueur()`.) Si l'on trouve que donner le même nom à la fonction et à la méthode est source de confusion, on peut tout à fait choisir un nom différent pour la méthode, comme par exemple

```
def taille(self):
    return longueur(self.tete)
```

Mieux encore, on peut donner à cette méthode le nom `__len__` et Python nous permet alors d'écrire `len(lst)` comme pour un tableau. En effet, lorsque l'on écrit `len(e)` en Python, ce n'est qu'un synonyme pour l'appel de méthode `e.__len__()`.

De même, on peut ajouter à la classe `Liste` une méthode pour accéder au n-ième élément de la liste, c'est-à-dire une méthode qui va appeler notre fonction `nieme_element` sur `self.tete`. Le nom de la méthode est arbitraire et nous pourrions choisir de conserver le nom `nieme_element`. Mais là encore nous pouvons faire le choix d'un nom idiomatique en Python, à savoir `__getitem__` :

```
def __getitem__(self, n):
    return nieme_element(n, self.tete)
```

Ceci nous permet alors d'écrire `lst[i]` pour accéder au `i`-ième élément de notre liste, exactement comme pour les tableaux. Pour la fonction `renverser`, on fait le choix de nommer la méthode `reverse` car là encore c'est un nom qui existe déjà pour les tableaux de Python.

Programme 23 — Encapsulation d'une liste dans un objet

```
class Liste:
    """une liste chaînée"""
    def __init__(self):
        self.tete = None

    def est_vide(self):
        return self.tete is None

    def ajoute(self, x):
        self.tete = Cellule(x, self.tete)

    def __len__(self):
        return longueur(self.tete)

    def __getitem__(self, n):
        return nieme_element(n, self.tete)

    def reverse(self):
        self.tete = renverser(self.tete)

    def __add__(self, lst):
        r = Liste()
        r.tete = concatener(self.tete, lst.tete)
        return r
```

```
    def reverse(self):
        self.tete = renverser(self.tete)
```

Enfin, le cas de la concaténation est plus subtil, car il s'agit de renvoyer une nouvelle liste, c'est-à-dire un nouvel objet. On choisit d'appeler la méthode `__add__`, qui correspond à la syntaxe `+` de Python.

```
    def __add__(self, lst):
        r = Liste()
        r.tete = concatener(self.tete, lst.tete)
        return r
```

Ainsi, on peut écrire `l+l` pour obtenir la liste $1, 2, 3, 1, 2, 3$. La totalité du code est donnée dans le programme 23.

Intérêt d'une telle encapsulation. Comme expliqué dans les chapitres 3 et 2, l'intérêt de l'encapsulation est multiple. D'une part, il cache la représentation de la structure à l'utilisateur. Ainsi, celui qui utilise notre classe `Liste` n'a plus à manipuler explicitement la classe `Cellule`. Mieux encore, il peut complètement ignorer l'existence de la classe `Cellule`. De même, il ignore que la liste vide est représentée par la valeur `None`. En particulier, la réalisation de la classe `Liste` pourrait être modifiée sans pour autant que le code qui l'utilise n'ait besoin d'être modifié à son tour.

D'autre part, l'utilisation de classes et de méthodes nous permet de donner le même nom à toutes les méthodes qui sont de même nature. Ainsi, on peut avoir plusieurs classes avec des méthodes `est_vide`, `ajoute`, etc. Si nous avions utilisé de simples fonctions, il faudrait distinguer `liste_est_vide`, `pile_est_vide`, `ensemble_est_vide`, etc.

À retenir. Une **liste chaînée** est une structure de données pour représenter une séquence finie d'éléments. Chaque élément est contenu dans une **cellule**, qui fournit par ailleurs un moyen d'accéder à la cellule suivante. Les opérations sur les listes chaînées se programment sous la forme de **parcours** qui suivent ces liaisons, en utilisant une **fonction récursive** ou une **boucle**.

Exercices

Exercice 48 Écrire une fonction `listeN(n)` qui reçoit en argument un entier `n`, supposé positif ou nul, et renvoie la liste des entiers $1, 2, \ldots,$ `n`, dans cet ordre. Si `n` $= 0$, la liste renvoyée est vide. Solution page 452 □

Exercice 49 Écrire une fonction `affiche_liste(lst)` qui affiche, en utilisant la fonction `print`, tous les éléments de la liste `lst`, séparés par des espaces, suivis d'un retour chariot. L'écrire comme une fonction récursive, puis avec une boucle `while`. Solution page 452 □

Exercice 50 Réécrire la fonction `nieme_element` (programme 20 page 114) avec une boucle `while`. Solution page 453 □

Exercice 51 Écrire une fonction `occurrences(x, lst)` qui renvoie le nombre d'occurrences de la valeur `x` dans la liste `lst`. L'écrire comme une fonction récursive, puis avec une boucle `while`. Solution page 453 □

Exercice 52 Écrire une fonction `trouve(x, lst)` qui renvoie le rang de la première occurrence de `x` dans `lst`, le cas échéant, et `None` sinon. L'écrire comme une fonction récursive, puis avec une boucle `while`.
Solution page 454 □

Exercice 53 Écrire une fonction récursive `concatener_inverse(l1, l2)` qui renvoie le même résultat que `concatener(renverser(l1), l2)`, mais sans appeler ces deux fonctions. En déduire une fonction `renverser` qui a la même complexité que le programme 22. Solution page 454 □

Exercice 54 Écrire une fonction `identiques(l1, l2)` qui renvoie un booléen indiquant si les listes `l1` et `l2` sont identiques, c'est-à-dire contiennent exactement les mêmes éléments, dans le même ordre. On suppose que l'on peut comparer les éléments de `l1` et `l2` avec l'égalité `==` de Python.
Solution page 455 □

Exercice 55 Écrire une fonction `inserer(x, lst)` qui prend en arguments un entier `x` et une liste d'entiers `lst`, supposée triée par ordre croissant, et qui renvoie une nouvelle liste dans laquelle `x` a été inséré à sa place. Ainsi, insérer la valeur 3 dans la liste $1, 2, 5, 8$ renvoie la liste $1, 2, 3, 5, 8$. On suggère d'écrire `inserer` comme une fonction récursive. Solution page 455 □

Exercice 56 En se servant de l'exercice précédent, écrire une fonction `tri_par_insertion(lst)` qui prend en argument une liste d'entiers `lst` et renvoie une nouvelle liste, contenant les mêmes éléments et triée par ordre croissant. On suggère de l'écrire comme une fonction récursive.
Solution page 455 □

Exercice 57 Écrire une fonction `liste_de_tableau(t)` qui renvoie une liste qui contient les éléments du tableau `t`, dans le même ordre. On suggère de l'écrire avec une boucle `for`. Solution page 455 □

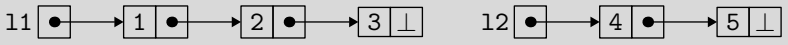
Exercice 58 Écrire une fonction `derniere_cellule(lst)` qui renvoie la dernière cellule de la liste `lst`. On suppose la liste `lst` non vide.
Solution page 456 □

Exercice 59 En utilisant la fonction de l'exercice précédent, écrire une seconde fonction, `concatener_en_place(l1, l2)`, qui réalise une concaténation en place des listes `l1` et `l2`, c'est-à-dire qui relie la dernière cellule de `l1` à la première cellule de `l2`. Cette fonction doit renvoyer la toute première cellule de la concaténation. Solution page 456 □

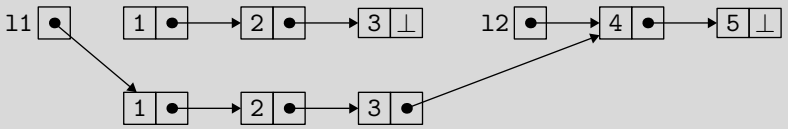
Gestion de la mémoire. L'utilisation de la mémoire faite par une fonction comme la concaténation de deux listes immuables `l1` et `l2` donnée en programme 21 pourrait inquiéter. En effet, cette fonction crée de nouvelles cellules pour dupliquer intégralement la liste `l1` donnée comme premier paramètre. Certes cela permet de préserver cette liste `l1` d'origine si elle doit encore être utilisée, mais n'est-ce pas un gâchis de mémoire si au contraire cette liste d'origine n'est elle-même plus utile ? Notamment, l'instruction

```
l1 = concatener(l1, l2)
```

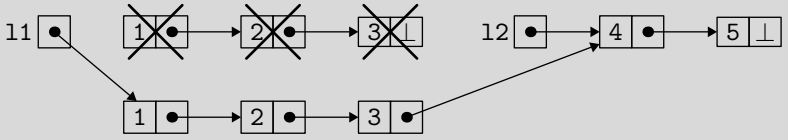
ferait passer d'un état de la mémoire



à l'état



où la variable `l1` permet d'atteindre les nouvelles cellules créées par la concaténation, mais où les cellules d'origine de la liste `l1` sont peut-être définitivement inaccessibles et la mémoire qu'elles utilisent gâchée.
Cette utilisation supplémentaire de mémoire n'est en réalité que temporaire, grâce à l'action du gestionnaire automatique de mémoire (GC). Ce mécanisme, présent en Python comme dans plusieurs autres langages, agit sans intervention du programmeur pour recycler automatiquement la mémoire utilisée par les éléments devenus inutiles.
Le critère utilisé par le GC pour déterminer les éléments utiles ou non à un instant donné est leur accessibilité à partir des variables du programme (variables globales du programme ou variables locales des appels de fonction en cours d'exécution) : un élément en mémoire que l'on ne peut plus atteindre en partant de ces variables peut être considéré comme définitivement perdu, et l'espace mémoire qu'il occupe est alors recyclé. On ne sait pas *quand* cette libération de la mémoire aura lieu, mais on sait qu'elle arrivera tôt ou tard.



Solutions des exercices

Exercice 1, page 16

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \times (n-1)! & \text{si } n > 0. \end{cases}$$

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Exercice 2, page 16

```
def syracuse(u_n):
    print(u_n)
    if u_n > 1:
        if u_n % 2 == 0:
            syracuse (u_n // 2)
        else:
            syracuse (3 * u_n + 1)
```

Exercice 3, page 16

```
def serie(n, a, b):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        r = 3 * serie(n - 1, a, b) + 2 * serie(n - 2, a, b) + 5
        return r
```

Exercice 4, page 16

```
def boucle(i,k):
    if i <= k:
      print(i)
      boucle(i+1, k)
```

Exercice 5, page 16

```
def pgcd(a, b):
    if a == 0:
      return b
    else:
      return pgcd(b % a, a)
```

Exercice 6, page 16

```
def nombre_de_chiffres(n):
    if n <= 9:
        return 1
    else:
        return 1 + nombre_de_chiffres(n // 10)
```

Exercice 7, page 16

```
def nombre_de_bits_1(n):
    if n == 0:
        return 0
    else:
        return (n % 2) + nombre_de_bits_1(n // 2)
```

Remarque : cette fonction est utile dans de nombreux algorithmes bas niveau et porte le nom de `popcount` dans la littérature, (pour l'anglais *population count*). Elle est souvent implémentée par les unité arithmétiques et logiques des processeurs sur des entiers de taille fixe (32 ou 64 bits).

Exercice 8, page 17

```
def appartient(v, t, i):
    if i == len(t):
      return False
    else:
      return t[i] == v or appartient(v, t, i + 1)
```

Exercice 9, page 17

```
def C(n,p):
    if p == 0 or n == p:
        return 1
    else:
        return C(n - 1, p - 1) + C(n - 1, p)

for i in range(10):
    for j in range(i + 1):
        print(C(i, j), ' ', end='')
    print()
```

Exercice 10, page 17

```
from turtle import *

def koch(n, l):
    if n == 0:
        forward(l)
    else:
        koch(n - 1, l/3)
        left(60)
        koch(n - 1, l/3)
        right(120)
        koch(n - 1, l/3)
        left(60)
        koch(n - 1, l/3)
```

Exercice 11, page 40

```
def cree():
    return []

def contient(s, x):
    return x in s

def ajoute(s, x):
    s.append(x)
```

Exercice 12, page 40 Pour permettre la modification de l'entier représentant l'ensemble, celui-ci doit être contenu dans une structure mutable. Par exemple un singleton (« 1-uplet ») nommé.

```
def cree():
    return { 'n': 0 }

def contient(s, x):
    return s['n'] & (1 << x) != 0

def ajoute(s, x):
    s['n'] = s['n'] | (1 << x)
```

Exercice 13, page 40 Comme dans le programme 9, on construit un tableau que l'on agrandit à chaque élément trouvé. On teste alors chaque bit de chacun des six entiers du tableau, et on calcule le nombre correspondant lorsque nécessaire.

```
def enumere(s):
    tab = []
    for paquet in range(6):
        for bit in range(64):
            if s[paquet] & (1 << bit) != 0:
                tab.append(paquet*64 + bit)
    return tab
```

Exercice 14, page 40 Dans la réalisation utilisant un tableau, on crée un nouveau tableau, que l'on remplit avec les éléments concernés. On peut dans le cas de l'union s'assurer qu'on n'introduit pas de doublons.

```
def union(s1, s2):
    t = [x for x in s1]
    for x in s2:
        if x not in s1:
            t.append(x)
    return t

def intersection(s1, s2):
    t = []
    for x in s1:
        if x in s2:
            t.append(x)
    return t
```

Pour la réalisation à base de tableau de bits, on utilise les opérations bit à bit | et &.

```
def union(s1, s2):
    return [(s1[i] | s2[i]) for i in range(6)]
```

Exercice 47, page 103
Partie 1.

```
# Question 1.1
a = point(0, 0)
b = point(1, 2)
c = point(-5, 18)
d = point(42, 37)

# Question 1.2
def deplacer_triangle(t, dx, dy):
    p0 = deplacer(t[0], dx, dy)
    p1 = deplacer(t[1], dx, dy)
    p2 = deplacer(t[2], dx, dy)
    return triangle(p0, p1, p2)

# Question 1.3
t1 = triangle(a,b,c)
t2 = triangle(b,c,d)
# Question 1.4
t3 = deplacer_triangle(t1, -1, -1)
t4 = deplacer_triangle(t2, 2, 3)
```

Partie 2. A priori, si l'on crée un objet `origine = point(0,0)` mutable, alors on pourra modifier l'origine en faisant `origine.deplacer(1,1)`, ce qui n'est pas souhaitable. À chaque fois que l'on voudra utiliser l'origine, il faudra recréer le point avec `point(0,0)`.

```
# Question 2.2
a = point(0, 0)
b = point(1, 2)
c = point(-5, 18)
d = point(42, 37)

# Question 2.3
t1 = triangle(a, b, c)
t2 = triangle(b, c, d)

# Question 2.4
t1.deplacer(-1, -1)
t2.deplacer(2, 3)
```

Après le premier appel de méthode, les points `a` `b` et `c` ont été modifiés en place. On a donc `a`$= (-1, -1)$, `b`$= (0, 1)$, et `c`$= (-6, 17)$. Cette modification a déjà un impact car `t2`, qui contient `b` et `c`, a déjà eu deux de ses points déplacés (mais pas le point `d`). Le résultat final est un triangle `t1` aux points

$(-1,-1),(2,4),(-4,20)$ et `t2` aux points $(2,4),(-4,20),(44,40)$, qui sont différents de `t3`$=(-1,-1),(0,-1),(6,17)$ et `t4`$=(3,5),(-3,21),(44,30)$.

```
#Question 2.5
class point:
   ... # constructeur et deplacer inchangés

    def clone(self):
        return point(self.x, self.y)

class triangle:
    def __init__(self, p1, p2, p3):
        self.p1 = p1.clone()
        self.p2 = p2.clone()
        self.p3 = p3.clone()

        ... # deplacer inchangé
```

Avec les modifications ci-dessus, la classe `triangle` stocke une *copie* des points qui lui sont passés en argument. Elle peut donc les modifier sans impacter les points initiaux. Il faudra cependant être vigilant, à chaque fois que l'on « donne » un tel `point` mutable à une fonction dont on ne connaît pas le code, car il existe un risque que ce point soit modifié. Il conviendra donc de faire des copies au moment opportun. Le style fonctionnel utilisé en Partie 1 permet d'écrire une code plus lisible (il n'est pas encombré d'opérations de copies) et dans lequel les erreurs causées par le partage sont inexistantes.

Exercice 48, page 124

```
def listeN(n):
    lst = None
    while n > 0:
        lst = Cellule(n, lst)
        n -= 1
    return lst
```

Exercice 49, page 124 Pour éviter que `print` n'insère un retour chariot après chaque élément, on utilise `print(..., end= " ")`, puis un unique `print()` à la fin de la liste.

```
def affiche_liste(lst):
    if lst is None:
        print()
    else:
```

```
        print(lst.valeur, end=" ")
        affiche_liste(lst.suivante)

def affiche_liste(lst):
    c = lst
    while c is not None:
        print(c.valeur, end=" ")
        c = c.suivante
    print()
```

Dans ces deux versions, on imprime un espace avant le retour chariot final. On peut faire un petit effort supplémentaire pour ne pas l'imprimer.

Exercice 50, page 124 Comme pour la fonction `longueur`, on se sert d'une variable `c` pour parcourir la liste. On se sert également d'une variable `i` pour mesurer notre avancement dans la liste. Il y a donc une double condition d'arrêt à la boucle.

```
def nieme_element(n, lst):
    i = 0
    c = lst
    while i < n and c is not None:
        i += 1
        c = c.suivante
    if n < 0 or c is None:
        raise IndexError("indice invalide")
    return c.valeur
```

Note : la condition `n < 0` nous préserve d'une valeur négative qui aurait été passée à la fonction `nieme_element`. Bien entendu, on aurait pu effectuer ce test dès l'entrée de la fonction, avant même de parcourir la liste, ou encore supposer que `n` est positif ou nul [NSI 1re, 9.3].

Exercice 51, page 124 Ici les deux versions se valent :

```
def occurrences(x, lst):
    if lst is None:
        return 0
    elif x == lst.valeur:
        return 1 + occurrences(x, lst.suivante)
    else:
        return occurrences(x, lst.suivante)

def occurrences(x, lst):
    n = 0
    c = lst
```

```
    while c is not None:
        if x == c.valeur:
            n += 1
        c = c.suivante
    return n
```

Exercice 52, page 124 Ici la version récursive est assez pénible à écrire, car il faut tester si l'appel récursif a renvoyé `None` :

```
def trouve(x, lst):
    if lst is None:
        return None
    elif lst.valeur == x:
        return 0
    else:
        r = trouve(x, lst.suivante)
        if r == None:
            return None
        else:
            return 1 + r
```

La boucle, en revanche, est plus simple à écrire :

```
def trouve(x, lst):
    i = 0
    c = lst
    while c is not None:
        if c.valeur == x:
            return i
        c = c.suivante
        i += 1
    return None
```

Exercice 53, page 125 On procède récursivement sur la structure de la liste `l1` :

```
def concatener_inverse(l1, l2):
    if l1 is None:
        return l2
    else:
        return concatener_inverse(l1.suivante, \
                                  Cellule(l1.valeur, l2))
```

La fonction `renverser` s'en déduit trivialement, en prenant une liste vide pour `l2` :

```
def renverser(lst):
    return concatener_inverse(lst, None)
```

Exercice 54, page 125 On le fait ici avec une fonction récursive. Il faut prendre soin de bien traiter le cas où l'une ou l'autre des listes est vide :

```
def identiques(l1, l2):
    if l1 is None:
        return l2 is None
    if l2 is None:
        return l1 is None
    return l1.valeur == l2.valeur \
       and identiques(l1.suivante, l2.suivante)
```

Exercice 55, page 125 Le cas d'arrêt est double : soit la liste est vide, soit `x` n'est pas plus grand que la valeur en tête de liste. Dans les deux cas, on ajoute `x` au début de `lst`. Sinon, on conserve le premier élément et on insère `x` récursivement dans la liste `lst.suivante`.

```
def inserer(x, lst):
    if lst is None or x <= lst.valeur:
        return Cellule(x, lst)
    else:
        return Cellule(lst.valeur, inserer(x, lst.suivante))
```

Exercice 56, page 125 Le cas de base correspond à une liste de longueur au plus 1, pour laquelle il suffit de la renvoyer.

```
def tri_par_insertion(lst):
    if lst is None or lst.suivante is None:
        return lst
    else:
        return inserer(lst.valeur, \
                       tri_par_insertion(lst.suivante))
```

Note : on pourrait également écrire `if longueur(lst) <= 1` mais ce serait calculer inutilement la longueur de la liste (calculer la longueur oblige à parcourir toute la liste), ce qui dégraderait les performances de ce programme.

Exercice 57, page 125 Pour préserver l'ordre des éléments, il faut prendre soin de parcourir le tableau de la droite vers la gauche :

```
def liste_de_tableau(t):
    lst = None
    for i in range(len(t) - 1, -1, -1):
```

```
            lst = Cellule(t[i], lst)
        return lst
```

Exercice 58, page 125

```
def derniere_cellule(lst):
    c = lst
    while c.suivante is not None:
        c = c.suivante
    return c
```

Exercice 59, page 125

```
def concatener_en_place(l1, l2):
    if l1 is None:
        return l2
    c = derniere_cellule(l1)
    c.suivante = l2
    return l1
```

Exercice 60, page 141 L'initialisation demande de créer une nouvelle pile, vide à l'origine.

```
adresse_courante = ""
adresses_precedentes = Pile()
adresses_suivantes = Pile()
```

La fonction de navigation principale annule les adresses suivantes en vidant la pile correspondante.

```
def aller_a(adresse):
    adresses_suivantes.vider()
    adresses_precedentes.empile(adresse_courante)
    adresse_courante = adresse
```

Chaque retour en arrière ajoute l'adresse courante à la pile des adresses suivantes, avant de revenir effectivement à la dernière adresse précédente connue.

```
def retour():
    if not adresses_precedentes.est_vide():
        adresses_suivantes.empile(adresse_courante)
        adresse_courante = adresses_precedentes.depile()
```

Symétriquement, `retour_avant` va à la première adresse suivante, après avoir remis la page courant sur la pile des adresses précédentes.

Index

⊥, 109
λ-calcul, 269
* (SQL), 313
+, 72
-, 101
.format, 247
[:], 250
[], 122, 250
&, 101
__add__, 123
__contains__, 54
__eq__, 54
__getitem__, 54, 122
__hash__, 53
__init__, 53
__len__, 54, 122
__lt__, 53
__str__, 53
|, 101

17576, 157

ABR, 158
ABR (classe), 168
abstraction, 29
ACID, 335
acquire (threading.), 375
Ada, 64
add (méthode), 19, 199, 209, 216
adjacence
    dictionnaire, 198
    liste, 200
    matrice, 196
Adleman, Len, 414
adresse IP, 384
AES, 408
ajout dans un ABR, 161
ajoute (fonction), 163
ajouter (méthode), 168
ajouter_arc (méthode), 197, 199
algorithme, 268
    de Boyer-Moore, 254
    de Dijkstra, 395
    du lièvre et de la tortue, 189
    glouton, 202, 234
alias de type, 76
alignement de séquences, 239
alto, 279
âne rouge, 195
angle, 63
annotation de type, 73
appartient (fonction), 160
appel récursif, 5
append, 134
arborescence, 173
arbre, 147, 173, 195, 219
    AVL, 169
    binaire, 147
    binaire de recherche, 157
    binaire parfait, 150
    d'appels, 5
    équilibré, 169
    peigne, 150
    rouge et noir, 169
    vide, 148
arc, 190
arc (méthode), 197, 199
architecture de Harvard, 349

arête d'un graphe, 193
arrêt (problème), 275
AS (SQL), 312, 318
ASC (SQL), 315
assert, 11, 79
AssertionError (exception), 79
attaque par force brute, 409
attribut, 46, 288
AttributeError (exception), 48
authentification, 415
automate, 273
autorité de certification, 418
AVG (SQL), 313
AVL, 169

balise XML, 176
bande passante, 390, 396
base de données, 288
    relationnelle, 297
BEGIN (SQL), 334
bibliothèque
    csv, 343
    Image, 230
    json, 184
    math, 435
    os, 186
    psycopg2, 338
    random, 28, 80, 98
    socket, 426
    ssl, 426
    sys, 14, 211, 226
    threading, 373, 375
    time, 82, 101
    turtle, 429
    typing, 78
    xml, 179
BIGINT (SQL), 300
bio-informatique, 239
bit array, 22
Bizet, Georges, 264
boîte noire, 79
bool (type), 72
BOOLEAN (SQL), 301
Border Gateway Protocol, 398
Borges, Jorge Luis, 157
boucle, 190
Boyer, Robert S., 254
Boyer-Moore (algorithme), 254
bytecode, 266

cache (défaut de), 353
calculabilité, 263
calculable (fonction), 268
canal auxiliaire, 409
carte, 191, 195
casse-tête, 195
Cellule (classe), 109
cellule d'une liste, 108, 109
certificat, 416
César, Jules, 407
ChaCha20, 408
chaîne (dans un graphe), 193
chaîne de caractères, 249
champ, 46
CHAR (SQL), 300
CHECK (SQL), 303
chemin, 191, 210
chiffrement
    par XOR, 407
    symétrique, 406
chmod (commande), 447
Chrono (classe), 46
Church, Alonzo, 269
circuit
    intégré, 347
    logique programmable, 348
class, 46
classe, 46
clé primaire, 291
client, 383
client d'un module, 29
Codd, Edgar F., 286, 331
Coffman, Edward Grady Jr., 371
colonne d'une table SQL, 299
coloriage (fonction), 203
coloriage d'un graphe, 195, 202
commande
    chmod, 447

`ifconfig`, 398
`ip`, 397
`kill`, 366, 372
`man`, 365
`openssl`, 419, 505
`ping`, 399, 403
`ps`, 364
`route`, 399
`top`, 366
`traceroute`, 400, 403
`COMMIT` (SQL), 333
commutation de contexte, 363
concaténation, 115, 120
concurrence, 367
conditions de Coffman, 371
connexité, 192, 218
constructeur, 53
`contient` (méthode), 168
contrainte, 289
contrat, 29
convertisseur, 350
copie de tables SQL, 321
copier-coller, 92
`cos` (`math.`), 435
`COUNT` (SQL), 313
couplage, 31
courbe elliptique, 422
`CREATE TABLE` (SQL), 298
cryptanalise, 409
cryptographie
asymétrique, 410
symétrique, 406
à clé publique, 410
`csv` (bibliothèque), 343
cycle, 192, 212
`cycle` (fonction), 213

`DATE` (SQL), 301
date, 63
décidabilité, 263
`DECIMAL` (SQL), 300
`degre` (méthode), 205
degré d'un sommet, 205
délai de convergence, 390
`DELETE` (SQL), 319
dépendance, 26
`depiler`, 129
DEPS, 127
`dequeue`, 130
`DESC` (SQL), 315
dichotomie, 170, 222
`dict` (type), 72
dictionnaire, 215, 245, 256
d'adjacence, 198
Diffie, Baily W., 412
Diffie-Hellman (protocole), 412
diffusion, 391
Dijkstra, Edsger, 395, 500
dîner des philosophes, 500
distance, 192, 214
`distance` (fonction), 216
`DISTINCT` (SQL), 315
diviser pour régner, 221
document semi-structuré, 176
DOM, 178
`DOUBLE PRECISION` (SQL), 300
`DROP TABLE` (SQL), 304
`dump` (`json.`), 184
dynamique (programmation), 233

EEPROM, 349
elliptique (courbe), 422
embarqué (système), 348
`empiler`, 129
encapsulation, 31, 50, 57, 121, 168
`END` (SQL), 334
`enqueue`, 130
ensemble, 209, 215
entité, 286
Entscheidungsproblem, 268
équilibrage d'un arbre, 169
Erdős, Paul, 194
espace de noms, 54
étiquette, 194
Euler, Leonhard, 189
`except`, 38
exception, 34
`AssertionError`, 79

`AttributeError`, 48
`IndexError`, 35, 64, 114, 115
`IOError`, 448
`KeyError`, 35, 200, 433
`NameError`, 35
`OSError`, 448
`RecursionError`, 14, 160, 169, 211, 224, 226
`TypeError`, 35, 52, 72
`ValueError`, 37, 85
`ZeroDivisionError`, 35, 263
exécutable, 361
exécution concurrente, 360
`existe_chemin` (fonction), 209
extension, 59

f_{91}, 8
factorielle, 16
FAI, 398
feuille d'un arbre, 149, 174
Fibonacci, 8, 9, 236, 246
FIFO, 128
`File` (classe), 137, 140
file, 127, 217
firmware, 355
FLASH, 349
`float` (type), 72
flocon de Koch, 17
fonction
anonyme, 94
calculable, 268
d'agrégation, 313
récursive, 5
forte connexité, 193
fraction, 62

GC, 120, 126, 164, 168, 462
gestionnaire d'interruption, 361
glouton (algorithme), 202, 234
`gnuplot`, 84
Gödel, Kurt, 278
GPS, 194
`Graphe` (classe), 197, 199
graphe, 189
arc, 190
chemin, 191
coloriage, 195, 202
parcours, 207
simple, 190
sommet, 190
`GROUP BY` (SQL), 324

hachage, 33, 200
Hanoï (tours de), 229
`hauteur`, 153
hauteur d'un arbre, 149, 174
`HAVING` (SQL), 324
Hellman, Martin, 412
héritage, 59
Hilbert, David, 268
Hofstadter, Douglas, 9
homme du milieu (attaque), 415
homogénéité, 111, 152
HTTPS (protocole), 417

IBM, 331
`ifconfig` (commande), 398
`Image` (bibliothèque), 230
immuable, 97
implémentation, 29
`IN` (SQL), 324
indécidabilité, 263
`IndexError` (exception), 35, 64, 114, 115
infixe (parcours), 154
injection de code SQL, 341
`INSERT INTO` (SQL), 305
instance, 47
`INT` (SQL), 300
`int` (type), 72
`INTEGER` (SQL), 300
interblocage, 370
interface, 26, 29, 58, 128, 130
réseau, 387
interruption (processeur), 361
intervalle, 62
`INTO` (SQL), 322
invariant, 84

`IOError` (exception), 448
`ip` (commande), 397
    `addr`, 397
    `route`, 398
`is`, 111

JavaScript, 91, 182
jeu, 195
`JOIN` (SQL), 316
jointure, 315
JSON, 182
`json` (bibliothèque), 184

Karatsuba, Anatolii Alexevich, 86, 231
`KeyError` (exception), 35, 200, 433
`kill` (commande), 366, 372
Knuth, Donald, 170, 189
Koch (flocon de), 17

labyrinthe, 195, 207
`lambda`, 94
largeur (parcours en), 214
lemming, 65
`len`, 122, 249
Leroy, Xavier, iv
lever (une exception), 35
lièvre, 189
LIFO, 127
ligne d'une table SQL, 299
`LIKE` (SQL), 312
`list` (type), 72
`Liste` (classe), 123
liste
    chaînée, 107
    concaténation, 115
    cyclique, 110
    d'adjacence, 200
    doublement chaînée, 110
    longueur, 111
    modification, 119
    renverser, 117
`load` (`json.`), 184
`Lock` (`threading.`), 375
logarithme, 228
loi de Moore, 347
longueur
    d'un chemin, 192
    d'une liste, 111

machine de Turing, 271
`man` (commande), 365
Mario Kart, 69
masque de sous-réseau, 385
`math` (bibliothèque), 435
matrice d'adjacence, 196
`MAX` (SQL), 313
McCarthy, John, 8
mémoire
    accès direct, 353
    cache, 353
    EEPROM, 349
    FLASH, 349
    morte, 349
    RAM, 349
    ROM, 349
    vive, 349
mémoïsation, 245
Merkle, Ralphe C., 410
méthode d'une classe, 50
`mex` (fonction), 203
microcontrôleur, 348
micrologiciel, 355
`MIN` (SQL), 313
`minimum` (fonction), 170
miroir, 87
mise à jour (SQL), 297
modèle relationnel, 285
modélisation, 286
modularité, 19
module, 26
monnaie (rendu de), 234
Moore, Gordon E., 347
Moore, J Strother, 254
Morris, 39
multiplication, 86, 231

`NameError` (exception), 35
navigateur, 130

GPS, 194
`nb_arcs` (méthode), 205
`nb_sommets` (méthode), 205
Newton, Isaac, 96
niveau de certification, 420
`Noeud` (classe), 151, 175
nœud
d'un arbre, 148
d'un graphe, 193
nombre d'Erdős, 194
non orienté (graphe), 190
`None`, 109, 113, 151
`NoneType` (type), 72
`NOT NULL` (SQL), 303
notation polonaise inverse, 141
NSI 1re, v, 11, 12, 19, 26, 29, 35, 45, 67, 71, 79, 90–92, 100, 102, 117, 120, 147, 160, 169, 170, 202, 216, 222, 224, 234, 238, 241, 251, 256, 276, 297, 300, 316, 343, 349, 359–361, 364, 371, 385, 399, 405, 408, 447, 453
`NULL` (SQL), 302

objet, 45, 47
`ON` (SQL), 316
`openssl` (commande), 419, 505
`ORDER BY` (SQL), 314
ordonnanceur de processus, 360
orienté (graphe), 190
`os` (bibliothèque), 186
`OSError` (exception), 448

panne, 393
paradigme
fonctionnel, 91
impératif, 91
orienté objets, 91
paradigmes de programmation, 90
paramètre de type, 76
parcours
de graphe, 207
en largeur, 214
en profondeur, 208
infixe, 154
`parcours` (fonction), 209
`parcours_largeur` (fonction), 216
parent (nœud), 152
parenthèse, 142
parfait (arbre binaire), 150
`parse` (fonction), 179
`parseString` (fonction), 179
partage, 116, 155, 163
Pascal (langage), 64
Pascal, Blaise, 17
peigne, 150, 160
PEPS, 128
`perf_counter` (`time.`), 82, 101
performance, 82
PGCD, 16, 434
philosophes (dîner des), 500
`pi` (`math.`), 435
PIL (Python Image Library), 230
`Pile` (classe), 135
pile, 127, 211, 230
d'appels, 12, 36, 131
`ping` (commande), 399, 403
plus court chemin, 395
polonaise inverse (notation), 141
`pop`, 129, 134
`pop` (méthode), 216
population count, 428
Post, Emil, 279
postfixe (parcours), 154
préfixe (parcours), 154, 175
`PRIMARY KEY` (SQL), 302
problème
de correspondance de Post, 279
de l'arrêt, 275
de la décision, 268
du tri, 228
processus, 359, 361
profondeur (parcours en), 208
programmation

concurrente, 367
dynamique, 233
fonctionnelle, 89
objet, 45
programme de Hilbert, 268
propriété, 46
protocole, 383
de routage, 385, 386
OSPF, 390
RIP, 386
protocole ICMP, 399
`ps` (commande), 364
`psycopg2` (bibliothèque), 338
puissance, 6
`push`, 129
puzzle de Merkle, 410

Queinnec, Christian, 3

racine d'un arbre, 148
`raise`, 35
RAM, 349
`randint` (`random.`), 28, 80, 98
`random` (bibliothèque), 28, 80, 98
rattraper une exception, 38
`REAL` (SQL), 300
réalisation, 29
recherche
dans un ABR, 159
dichotomique, 222
textuelle, 249
récursif (appel), 5
`RecursionError` (exception), 14, 160, 169, 211, 224, 226
récursive
définition, 4
fonction, 3, 5
récursivité, 3
mutuelle, 9
réduction de problème, 277
`REFERENCES` (SQL), 303
région française, 191
relation, 287
`release` (`threading.`), 375
`remove` (méthode), 205
rendu de monnaie, 234
renverser une liste, 117
requête
imbriquée, 323
SQL, 309
requête SQL, 297
réseau, 194
ressource, 359
`reverse` (méthode), 472
RISC, 350
Rivest, Ron, 414
`ROLLBACK` (SQL), 333
ROM, 349
rotation (d'un image), 230
routage statique, 397
`route` (commande), 399
routeur, 383
RSA (système cryptographique), 414

schéma, 288
section critique, 375
`SELECT` (SQL), 312, 338
`SELECT FROM WHERE` (SQL), 310
`self`, 47, 52
semi-décidable, 275
séquence (alignement), 239
serveur, 383
`SET` (SQL), 321
`set` (type), 19, 72, 199, 209
`setrecursionlimit` (`sys.`), 14, 211, 226
SGBD, 297
SHA256, 419
Shamir, Adi, 414
`sin` (`math.`), 435
sinus, 96
`SMALLINT` (SQL), 300
`socket` (bibliothèque), 426
solitaire, 195
sommet, 190
sous-arbre, 148
SQL, 297

*, 313
AS, 312, 318
ASC, 315
AVG, 313
BEGIN, 334
BIGINT, 300
BOOLEAN, 301
CHAR, 300
CHECK, 303
COMMIT, 333
COUNT, 313
CREATE TABLE, 298
DATE, 301
DECIMAL, 300
DELETE, 319
DESC, 315
DISTINCT, 315
DOUBLE PRECISION, 300
DROP TABLE, 304
END, 334
GROUP BY, 324
HAVING, 324
IN, 324
INSERT INTO, 305
INT, 300
INTEGER, 300
INTO, 322
JOIN, 316
LIKE, 312
MAX, 313
MIN, 313
NOT NULL, 303
NULL, 302
ON, 316
ORDER BY, 314
PRIMARY KEY, 302
REAL, 300
REFERENCES, 303
ROLLBACK, 333
SELECT, 312, 338
SELECT FROM WHERE, 310
SET, 321
SMALLINT, 300
START TRANSACTION, 333
SUM, 313
TEXT, 300
TIME, 301
TIMESTAMP, 301
UNIQUE, 303
UPDATE, 321
VARCHAR, 300
WHERE, 311
ssl (bibliothèque), 426
start (threading.), 373
START TRANSACTION (SQL), 333
str (type), 72
SUM (SQL), 313
suppression dans un ABR, 166
supprime (fonction), 167
supprimer_arc (méthode), 205
surcharge, 72
Syracuse (conjecture de), 16
sys (bibliothèque), 14, 211, 226
System on Chip (SoC), 348, 351
System R, 331
système
    autonome, 398
    d'information, 285
    de gestion de bases de données (SGBD), 297, 329
    multitâche, 360
    sur puce, 351

table
    de hachage, 33
    SQL, 299
tableau, 64
    de bits, 22
tâche, 362
taille, 153
taille d'un arbre, 148, 174
taquin, 195
temps d'exécution, 82
test, 79
TEXT (SQL), 300
texte (recherche dans un), 249
théorème des quatre couleurs, 195
thèse de Church-Turing, 273

`Thread` (`threading.`), 373
thread, 362
`threading` (bibliothèque), 373, 375
tiers de confiance, 417
`TIME` (SQL), 301
`time` (`time.`), 82
`time` (bibliothèque), 82, 101
timer, 350
`TIMESTAMP` (SQL), 301
TLS (protocole), 420
`top` (commande), 366
tortue, 67, 189
tours de Hanoï, 229
`traceroute` (commande), 400, 403
transaction, 333
tri, 79, 170, 171
  fusion, 224
  par insertion, 91
  rapide, 229
triangle de Pascal, 17
`try`, 38
Turing, Alan, 269
`turtle` (bibliothèque), 429
Twain, Mark, 334
type, 71
`TypeError` (exception), 35, 52, 72
`typing` (bibliothèque), 78

`UNIQUE` (SQL), 303
`UPDATE` (SQL), 321
UTF-8, 183, 447, 505, 506

`ValueError` (exception), 37, 85
`VARCHAR` (SQL), 300
variant, 224
vecteur de distance, 387
verrou, 375
voisin, 190
`voisins` (méthode), 197, 199

W3C, 176
`WHERE` (SQL), 311
Wikipedia, 189

X.509, 419
XML, 176
`xml` (bibliothèque), 179
XOR, 407

`ZeroDivisionError` (exception), 35, 263
zone OSPF, 391