spécialité Numérique et sciences informatiques

30 leçons avec exercices corrigés

classe de première

Thibaut Balabonski
Sylvain Conchon
Jean-Christophe Filliâtre
Kim Nguyen

préface de Gilles Dowek

©Éditions Ellipses
ISBN 978-2340033641

Préface

La rentrée 2019 marque une nouvelle étape dans l'histoire de l'enseignement de l'informatique au lycée qui, après une phase d'expérimentation, passe à une phase de développement, avec les enseignements Sciences Numériques et Technologie en seconde et surtout Numérique et Sciences Informatiques en première et en terminale. La différence avec les expérimentations précédentes est énorme, par le volume horaire de cet enseignement : quatre heures en première et six en terminale, par le nombre d'élèves potentiels auxquels il s'adresse : tous les élèves du lycée général, mais aussi parce que ces élèves auront déjà acquis les éléments fondamentaux de l'informatique à l'école et au collège. Il devient donc possible de donner aux élèves qui choisiront cette spécialité, quel que soit le métier auquel ils se destinent, une solide culture en informatique, qui leur permettra de faire des choix éclairés, dans leur vie professionnelle et comme citoyens.

Ces nouveaux enseignements posent naturellement de nombreux défis aux enseignants. La création d'une nouvelle discipline s'accompagne toujours d'une phase de transition – sans doute d'une décennie, au moins –, pendant laquelle les élèves sont déjà là, mais non encore les enseignants formés à cette discipline sur les bancs de l'université et recrutés par les concours habituels. Il devient alors nécessaire de recourir à un outil très courant ailleurs, mais paradoxalement trop peu utilisé par les enseignants : la formation permanente, afin que ceux qui le souhaitent puissent ajouter une corde à leur arc et devenir, également, professeurs d'informatique.

Il y a un point commun entre la pédagogie de l'informatique et l'informatique elle-même : on l'apprend en faisant. Mais il y a aussi un autre point commun : on ne peut faire et apprendre qu'en s'appuyant sur des documents solides, car on ne peut naturellement pas tout redécouvrir soi-même. C'est pour cela que les manuels destinés aux élèves et les documents destinés à la formation permanente des enseignants sont essentiels pour le succès de cette entreprise.

Rédiger un manuel demande un talent d'équilibriste : il faut en dire ni trop ni trop peu, traiter toutes les branches de l'informatique de manière égale, trouver la bonne proportion de cours et d'exercices… Plus que tout, il faut mettre en lumière les notions fondamentales, et la manière dont elles s'articulent, en s'abstrayant des détails contingents de leur incarnation

© Ellipses
ISBN 978-2-340-03641

dans tel ou tel objet. C'est le miracle qu'accomplissent Thibaut Balabonski, Sylvain Conchon, Jean-Christophe Filliâtre et Kim Nguyen dans ce livre destiné aux enseignants, qui y trouveront les éléments essentiels pour progresser et ainsi faire progresser leurs élèves.

Ce livre, sans nul doute, jouera un rôle clé dans le succès de la spécialité Numérique et Sciences Informatiques, et par delà, dans le succès de l'enseignement des sciences et des techniques au lycée.

Gilles Dowek
Chercheur à Inria
Professeur attaché à l'École Normale Supérieure de Paris-Saclay

©Éditions Ellipses
ISBN 978-2340036411

Avant-propos

À qui s'adresse cet ouvrage ? Cet ouvrage s'adresse autant à l'enseignant qu'à l'élève. Un élève de première 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 première, sous la forme d'une trentaine de 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, mais il est fortement conseillé de commencer par la partie I et de la traiter dans l'ordre proposé.

Cet ouvrage peut également être utilisé comme une introduction au langage Python en particulier et à l'informatique en général.

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 dans la partie 30.3. 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 allumettes (exercice 108 page 101) ou encore le puissance 4 (exercice 128 page 119).

Le site du livre. Le site `http://www.nsi-premiere.fr/` propose des ressources complémentaires. L'enseignant comme l'élève trouveront sur ce site

©Editions Ellipses
ISBN 978-2-340-03641

des sujets de projet pluridisciplinaires. Ces derniers pourront mobiliser les connaissances de plusieurs chapitres et d'autres champs disciplinaires (mathématiques, physique, sciences économiques et sociales, biologie, etc.) sur plusieurs séances de travaux pratiques ou comme devoir à la maison.

Le site contient également des informations sur la prise en main de l'environnement de développement Idle, dont on préconise l'utilisation. Néanmoins, tout cet ouvrage peut être lu et assimilé, et les exercices faits, dans n'importe quel autre environnement de développement en Python.

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 Xavier Blanc, Sylvie Boldo, Alain Busser, Christine Froidevaux, François Fayard, Ignacy Gawędzki, Julien Narboux, Yann Régis-Gianas, Laurent Sartre. Nous sommes reconnaissants à Corinne Baud et Sylvie Cioflan, 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 Gilles Dowek ait accepté de préfacer cet ouvrage et nous le remercions vivement.

©Éditions Ellipses
ISBN 978-2340033641

Table des matières

I Introduction à la programmation avec Python **1**

1 Arithmétique, variables, instructions **3**
1.1 Mode interactif 3
1.2 Mode programme 10
1.3 Bibliothèque Turtle 16
Exercices 21

2 Boucle `for` **25**
2.1 Problème : dessiner une spirale 25
2.2 Boucles bornées simples 26
2.3 Utilisation de l'indice de boucle 28
2.4 Utilisation d'un accumulateur 30
Exercices 36

3 Comparaisons, booléens, tests **39**
3.1 Problème : compter les points au mölkky 39
3.2 Conditions et branchements 40
3.3 Après le branchement : jonction 43
3.4 Expressions booléennes 45
3.5 Conditionnelles imbriquées 49
Exercices 53

4 Fonctions **57**
4.1 Problème : dessiner une face d'un dé 57
4.2 Définir une fonction 58
4.3 Renvoyer un résultat 62
4.4 Variables locales à une fonction 67
4.5 Sortie anticipée 70
Exercices 71

5 Tableaux **73**
5.1 Problème : la pyramide des âges 73
5.2 Notion de tableau 74
5.3 Parcours d'un tableau 76

© Editions Ellipses
ISBN 978-2340033641

5.4 Construire de grands tableaux . . . 79
5.5 Tableaux et variables . . . 80
Exercices . . . 84

6 Boucle `while` 87
6.1 Problème : approximation de la racine carrée . . . 87
6.2 Boucles « tant que » . . . 88
6.3 Sortir d'une boucle avec `break` . . . 92
6.4 Problème : chercher dans un tableau . . . 94
6.5 Terminaison . . . 96
Exercices . . . 97

7 Utilisation avancée des boucles 103
7.1 Problème : recherche de doublons . . . 103
7.2 Boucles imbriquées . . . 104
7.3 Boucles imbriquées dépendantes . . . 106
7.4 Estimation de la complexité . . . 107
7.5 Instruction `continue` . . . 107
Exercices . . . 110

8 Utilisation avancée des tableaux 113
8.1 Itérer sur les éléments d'un tableau . . . 113
8.2 Construire un tableau par compréhension . . . 114
8.3 Tableaux à plusieurs dimensions . . . 115
Exercices . . . 118

9 Spécification et mise au point 121
9.1 Que fait ce programme ? . . . 121
9.2 Documenter ses programmes . . . 122
9.3 Programmation défensive . . . 123
9.4 Tester ses programmes . . . 125
9.5 Corriger les erreurs . . . 129
9.6 Invariant de boucle . . . 135
Exercices . . . 138

II Algorithmique 141

10 Algorithmes de tri 143
10.1 Problème : comparer deux tableaux . . . 143
10.2 Tri par sélection . . . 144
10.3 Tri par insertion . . . 147
10.4 Les tris fournis par Python . . . 150
Exercices . . . 151

© Éditions Eyrolles
ISBN 978-2-340-03364-1

11 Recherche dichotomique dans un tableau trié 153
11.1 Mise en œuvre en Python 153
11.2 Correction 156
11.3 Efficacité 157
Exercices 158

12 Algorithmes gloutons 161
12.1 Problème d'optimisation : le voyageur 161
12.2 Algorithmes gloutons 163
12.3 Problème : rendu de monnaie 167
Exercices 170

13 Apprentissage et algorithme des plus proches voisins 175
13.1 Problème : deviner la carte scolaire en sondant ses voisins . . . 176
13.2 Classification avec l'algorithme des voisins 176
13.3 Notion de distance 180
13.4 Limites de l'approche 181
13.5 Problème : estimation de prix immobiliers 183
Exercices 184

III Traitement de données en tables 187

14 Ensembles, n-uplets et dictionnaires 189
14.1 Le paradoxe des anniversaires 189
14.2 Les n-uplets 193
14.3 Les dictionnaires 196
Exercices 201

15 Indexation de tables 203
15.1 Notion de table de données 203
15.2 Lire un fichier au format CSV 204
15.3 Validation des données 207
15.4 Écrire un fichier au format CSV 208
Exercices 210

16 Recherche dans une table 213
16.1 Recherche 213
16.2 Agrégation 215
16.3 Sélection de lignes 216
16.4 Sélection de lignes et colonnes 219
Exercices 220

©Editions Ellipses
ISBN 978-2340033641

17 Tri d'une table **225**
17.1 Trier des données en fonction d'une clé . . . 226
17.2 Ordre lexicographique et stabilité . . . 227
17.3 Trier en place . . . 228
17.4 Application : recherche des plus proches voisins . . . 228
Exercices . . . 229

18 Fusion de tables **231**
18.1 Réunion de tables . . . 231
18.2 Opération de jointure . . . 233
18.3 Utilisation d'un identifiant unique . . . 234
Exercices . . . 237

IV Architecture matérielle et représentation des données **239**

19 Représentation des entiers **241**
19.1 Encodage des entiers naturels . . . 242
19.2 Boutisme . . . 246
19.3 Encodage des entiers relatifs . . . 247
Exercices . . . 249

20 Représentation approximative des nombres réels **251**
20.1 Norme IEEE 754 . . . 252
20.2 Les flottants en Python . . . 256
Exercices . . . 259

21 Représentation des textes **261**
21.1 Codage ASCII . . . 261
21.2 Normes ISO 8859 . . . 265
21.3 Codage Unicode . . . 265
Exercices . . . 272

22 Circuits et logique booléenne **275**
22.1 Portes logiques . . . 275
22.2 Fonctions booléennes . . . 278
22.3 Circuits combinatoires . . . 280
Exercices . . . 284

23 Modèle de von Neumann **287**
23.1 Composants d'un ordinateur . . . 287
23.2 Organisation de la mémoire . . . 291
23.3 Langage machine . . . 296
Exercices . . . 302

© Éditions Ellipses
ISBN 978-2-340-03641

V Interaction et communication 305

24 Systèmes d'exploitation 307
24.1 Principes généraux 308
24.2 Le standard POSIX 310
24.3 L'interface système ou *Shell* 311
24.4 Redirections 320
24.5 Commandes utiles 323
Exercices 324

25 Interaction avec l'utilisateur 329
25.1 Entrées et sorties en Python 330
25.2 La bibliothèque `tkinter` 333
25.3 Programmation évènementielle 336
Exercices 340

26 Réseaux et Internet 343
26.1 Terminologie et généralités 344
26.2 Modèle OSI 345
26.3 Modèle Internet 346
26.4 Programmation réseau en Python 355
Exercices 359

27 HTML et CSS 361
27.1 HTML 362
27.2 CSS 369
Exercices 378

28 Requêtes HTTP et formulaires 381
28.1 HTTP, le protocole du Web 381
28.2 Formulaires et passage de paramètres 387
Exercices 393

29 Le Web, côté serveur 395
29.1 Concepts fondamentaux du Web côté serveur 395
29.2 Survol du langage PHP 400
Exercices 405

30 Le Web, côté client 407
30.1 Premier contact avec JavaScript 407
30.2 Évaluation de code JavaScript 410
30.3 Les applications Web modernes 412
Exercices 414

Solutions des exercices 417

©Editions Ellipses
ISBN 978-2340033641

Première partie

Introduction à la programmation avec Python

©Éditions Ellipses
ISBN 978-2340033641

©Éditions Ellipses
ISBN 978-2340033641

Chapitre 1

Arithmétique, variables, instructions

Notions introduites

- les modes interactif et programme de Python
- expressions arithmétiques
- messages d'erreur
- manipuler des variables
- séquence d'instructions
- lire et écrire des chaînes de caractères

Le langage de programmation Python permet d'interagir avec la machine à l'aide d'un programme appelé *interprète Python*[1]. On peut l'utiliser de deux façons différentes. La première méthode consiste en un dialogue avec l'interprète. C'est le *mode interactif*. La seconde consiste à écrire un programme dans un fichier source puis à le faire exécuter par l'interprète. C'est le *mode programme*.

1.1 Mode interactif

En première approximation, le mode interactif de l'interprète Python se présente comme une calculatrice[2]. Les trois chevrons `>>>` constituent l'invite de commandes de Python, qui indique qu'il attend des instructions. Si par

1. Le site `https://www.nsi-premiere.fr` qui accompagne ce livre présente plusieurs environnements pour travailler avec Python.

2. On suppose ici que l'interprète Python vient d'être lancé, quelle que soit la solution retenue.

© Éditions ELLIPSES
ISBN 978-2-7298-0033641

exemple on tape `1+2` puis la touche Entrée, l'interprète Python calcule et affiche le résultat.

```
>>> 1+2
3
>>>
```

Comme on le voit, les chevrons apparaissent de nouveau. L'interprète est prêt à recevoir de nouvelles instructions.

Arithmétique

En Python, on peut saisir des combinaisons arbitraires d'opérations arithmétiques, en utilisant notamment les quatre opérations les plus communes.

```
>>> 2 + 5 * (10 - 1 / 2)
49.5
>>>
```

L'addition est notée avec le symbole `+`, la soustraction avec le symbole `-`, la multiplication avec le symbole `*` et la division avec le symbole `/`. La priorité des opérations est usuelle et on peut utiliser des parenthèses. Dans l'exemple ci-dessus, on a utilisé des espaces pour améliorer la lisibilité. Ces espaces sont purement décoratifs et ne modifient pas la façon dont l'expression est calculée. En particulier, elles n'agissent pas sur la priorité des opérations.

```
>>> 1+2 * 3
7
```

Erreurs. L'interprète n'accepte que des expressions arithmétiques complètes et bien formées. Sinon, l'interprète indique la présence d'une erreur.

```
>>> 1 + * 2
  File "<stdin>", line 1
    1 + * 2
        ^
SyntaxError: invalid syntax
```

Ici, le message `SyntaxError: invalid syntax` indique une erreur de syntaxe, c'est-à-dire une instruction mal formée. Nous verrons plus loin d'autres catégories d'erreur. On peut ignorer pour l'instant la ligne `File "<stdin>", line 1`. Les deux lignes suivantes montrent à l'utilisateur l'endroit exact de l'erreur de syntaxe avec le symbole `^`, ici le caractère `*`.

© Éditions Ellipses
ISBN 978-2-340-03641

Erreurs. Un autre type d'erreur se manifeste lorsque l'on donne à l'interprète une expression qui est correcte du point de vue de l'écriture mais dont le résultat n'a pas de sens. Ainsi toute tentative de division par zéro produit un message spécifique.

```
>>> 2 / (3 - 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Attention, les nombres « à virgule » s'écrivent à l'anglo-saxonne avec un point, et non avec une virgule qui a un autre sens en Python. En essayant d'appliquer des opérations arithmétiques à des nombres écrits avec des virgules on obtient toute une panoplie de comportements inattendus.

```
>>> 1,2 + 3,4
1, 5, 4
```

```
>>> 1,2 * 3
1, 6
```

En Python, la virgule sert à séparer deux valeurs. On en verra différentes utilisations aux chapitres 4, 5 et 14.

Les nombres de Python. Les nombres de Python sont soit des entiers relatifs, simplement appelés entiers, soit des nombres décimaux, appelés flottants.
Les entiers peuvent être de taille arbitraire (ce qui n'est pas le cas dans de nombreux langages de programmation). Ces entiers ne sont limités que par la mémoire disponible pour les stocker. L'exercice 6 page 22 propose une expérience pour observer que les entiers de Python peuvent être très grands. Le chapitre 19 reviendra sur la représentation des entiers dans un ordinateur.
Les nombres flottants, en revanche, ont une capacité limitée et ne peuvent représenter qu'une partie des nombres décimaux. Ainsi, des nombres décimaux trop grands ou trop petits ne sont pas représentables. Si on saisit un nombre décimal en tapant `1` suivi de cinq cents `0` et d'un point, on obtient la valeur `inf`, qui représente une valeur trop grande pour être représentée de cette manière. Des nombres comme π, $\sqrt{2}$, qui ne sont pas des nombres décimaux, ne peuvent être représentés que de manière approximative en Python. Ces approximations ne seront cependant pas problématiques dans les situations que nous rencontrerons. Le chapitre 20 reviendra sur la représentation des nombres réels dans un ordinateur.

ISBN : 978-2-340-03641

À propos de la division. Si on effectue la division de deux entiers avec l'opération `/`, on obtient un nombre décimal. Ainsi, `7 / 2` donne le nombre `3.5`. (Le chapitre 20 expliquera comment de tels nombres décimaux sont représentés dans la machine.) Si on veut effectuer en revanche une division entière, alors il faut utiliser l'opération `//`. Ainsi, `7 // 2` donne le nombre `3`, qui est cette fois un entier, à savoir le quotient de 7 par 2 dans la division euclidienne. On peut également obtenir le reste d'une telle division euclidienne avec l'opération `%`. Ainsi, `7 % 2` donne l'entier `1`. (La division euclidienne est celle que l'on a apprise à l'école primaire.)
Attention : les opérations `//` et `%` de Python ne coïncident avec la division euclidienne que lorsque le diviseur est positif. Lorsqu'il est négatif, le reste est alors également négatif. Voici une illustration des quatre cas de figure :

	$a = 7$ $b = 3$	$a = -7$ $b = 3$	$a = 7$ $b = -3$	$a = -7$ $b = -3$
a `//` b	2	-3	-3	2
a `%` b	1	2	-2	-1

Dans tous les cas, on a l'égalité $a = (a \texttt{ // } b) \times b + a \texttt{ \% } b$ et l'inégalité $|a \texttt{ \% } b| < |b|$. Une autre façon de le voir, sans doute plus simple, consiste à définir $a \texttt{ // } b$ comme la partie entière (par défaut) du nombre réel a/b, c'est-à-dire le plus grand entier inférieur ou égal à a/b. Cela étant, il est rare que l'on divise par un nombre négatif. On peut même le décourager purement et simplement pour éviter toute différence accidentelle avec la division euclidienne.

Variables

Les résultats calculés peuvent être mémorisés par l'interprète, afin d'être réutilisés plus tard dans d'autres calculs.

```
>>> a = 1 + 1
>>>
```

La notation `a =` permet de donner un nom à la valeur à mémoriser. L'interprète calcule le résultat de `1+1` et le mémorise dans la *variable* `a`. Aucun résultat n'est affiché. On accède à la valeur mémorisée en utilisant le nom `a` [3].

```
>>> a
2
```

Plus généralement, la variable peut être réutilisée dans la suite des calculs.

3. On suppose que toutes les commandes tapées dans cette section sont tapées à la suite sans relancer à chaque fois l'interprète Python.

©Éditions ENI ISBN 978

```
>>> a * (1 + a)
6
```

Le symbole = utilisé pour introduire la variable a désigne une opération d'*affectation*. Il attend à sa gauche un nom de variable et à sa droite une expression. On peut donner une nouvelle valeur à la variable a avec une nouvelle affectation. Cette valeur remplace la précédente.

```
>>> a = 3
>>> a * (1 + a)
12
```

Le calcul de la nouvelle valeur de a peut utiliser la valeur courante de a.

```
>>> a = a + 1
>>> a
4
```

Un nom de variable peut être formé de plusieurs caractères (lettres, chiffres et souligné). Il est recommandé de ne pas utiliser de caractères accentués et l'usage veut que l'on se limite aux caractères minuscules.

```
>>> cube = a * a * a
>>> ma_variable = 42
>>> ma_variable2 = 2019
```

Erreurs. Un nom de variable ne doit pas commencer par un chiffre et certains noms sont interdits (car ils sont des mots réservés du langage).

```
>>> 4x = 2
  File "<stdin>", line 1
    4x = 2
     ^
SyntaxError: invalid syntax
>>> def = 3
  File "<stdin>", line 1
    def = 3
        ^
SyntaxError: invalid syntax
```

Il n'est pas possible d'utiliser une variable qui n'a pas encore été définie.

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

©Editions ENI
ISBN 978-2340033641

Erreurs. Seul un nom de variable peut se trouver à la gauche d'une affectation.

```
>>> 1 = 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Une variable peut être imaginée comme une petite boîte portant un nom (ou une étiquette) et contenant une valeur. Ainsi, on peut se représenter une variable déclarée avec `x = 1` par une boîte appelée `x` contenant la valeur 1.

Lorsque l'on modifie la valeur de la variable `x`, par exemple avec l'affectation `x = x + 1`, la valeur 1 est remplacée par la valeur 2.

État

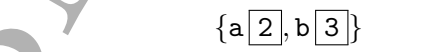
À chaque étape de l'interaction, chacune des variables introduites contient une valeur. L'ensemble des associations entre des noms de variables et des valeurs constitue l'*état* de l'interprète Python. Cet état évolue en fonction des instructions exécutées, et notamment en raison des affectations de nouvelles valeurs. On dit de ces instructions qui modifient l'état qu'elles ont un *effet de bord*.

On peut représenter l'état de l'interprète par un ensemble d'associations entre des noms de variables et des valeurs. Par exemple, l'ensemble {`a` 1, `b` 2, `c` 3} représente l'état dans lequel les variables `a`, `b` et `c` valent respectivement `1`, `2` et `3` et où aucune autre variable n'est définie.

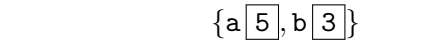
Pour simuler à la main une étape d'interaction, on indiquera l'état avant et après l'exécution de l'instruction. Considérons par exemple l'instruction suivante.

```
>>> a = a + b
```

Cette instruction modifie la valeur de `a` en fonction des valeurs de `a` et `b`. Par exemple, en partant de l'état



on obtient après exécution de l'instruction un état où l'association `a` 5 a remplacé `a` 2.



© Éditions Eyrolles
ISBN 978-2-340-03641

Différences entre mathématiques et informatique. En informatique, le terme de *variable* est utilisé pour indiquer le fait que la valeur associée peut *varier* au fur et à mesure de l'exécution du programme. Ce n'est donc pas la même notion que celle de variable en mathématiques, désignant elle une unique valeur (qui est, ne serait-ce que provisoirement, indéterminée).
Par ailleurs, on a parfois des manières différentes d'écrire les choses dans ces deux domaines, et le même symbole peut ne pas avoir la même signification en informatique et en mathématiques. C'est le cas en particulier du symbole d'égalité, dont l'utilisation en informatique peut paraître étrange si on la confond avec sa signification mathématique. La suite de symboles `a = a + 1` par exemple n'a pas de sens si on la voit comme une égalité. Il faut la voir comme une instruction, qui donne comme nouvelle valeur à la variable `a` le résultat de l'expression `a + 1` calculé avec la valeur courante de `a`. En l'occurrence, on vient donc d'augmenter la valeur de `a` d'une unité.
En conséquence de cette nature de l'instruction d'affectation, même la désignation de la variable par `a` peut avoir deux sens différents en informatique. La plupart du temps `a` désigne la valeur de la variable, c'est-à-dire le contenu de la boîte associée, mais à gauche du symbole d'affectation `a` est une *référence* à la boîte elle-même, dont on va modifier le contenu.
Comme il est courant de modifier la valeur d'une variable en lui ajoutant une certaine quantité, il existe en outre une instruction spécifique pour cela, notée `+=`. Ainsi, on peut, en informatique, écrire `a += 1` au lieu de `a = a + 1`.

Si d'autres variables sont définies dans l'état de départ et ne sont pas affectées par l'instruction, leur valeur reste inchangée. Ainsi, partant de

$$\{\texttt{a}\,\boxed{\texttt{0}}, \texttt{b}\,\boxed{\texttt{1}}, \texttt{d}\,\boxed{\texttt{-12}}, \texttt{x}\,\boxed{\texttt{3}}\}$$

on obtient l'état suivant.

$$\{\texttt{a}\,\boxed{\texttt{1}}, \texttt{b}\,\boxed{\texttt{1}}, \texttt{d}\,\boxed{\texttt{-12}}, \texttt{x}\,\boxed{\texttt{3}}\}$$

Lorsqu'une variable est affectée pour la première fois, l'association correspondante apparaît dans l'état. Ainsi, pour une instruction `a = b + c` et en partant de l'état

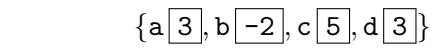
$$\{\texttt{b}\,\boxed{\texttt{-2}}, \texttt{c}\,\boxed{\texttt{5}}\}$$

on obtient l'état suivant.

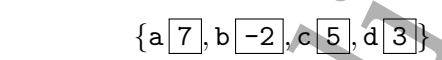
$$\{\texttt{a}\,\boxed{\texttt{3}}, \texttt{b}\,\boxed{\texttt{-2}}, \texttt{c}\,\boxed{\texttt{5}}\}$$

©Éditions ISBN 978-2-7598-0641

Dans le cas particulier où on affecte une nouvelle variable avec la valeur d'une variable qui existe déjà, par exemple avec `d = a`, il est important de comprendre que `d` et `a` ne sont pas deux noms pour la même boîte, mais deux boîtes différentes, la boîte `d` recevant la valeur contenue dans la boîte `a`.



On peut observer que les deux variables sont effectivement indépendantes : modifier la variable `a`, par exemple avec `a = 7`, n'a pas d'effet sur la variable `d`.



1.2 Mode programme

Le mode programme de Python consiste à écrire une suite d'instructions dans un fichier et à les faire exécuter par l'interprète Python. Cette suite d'instructions s'appelle un *programme*, ou encore un *code source*. On évite ainsi de ressaisir à chaque fois les instructions dans le mode interactif. Par ailleurs, cela permet de distinguer le rôle de programmeur du rôle d'utilisateur d'un programme.

Affichage

En mode programme, les résultats des expressions calculées ne sont plus affichés à l'écran. Il faut utiliser pour ceci une instruction explicite d'affichage. En Python, elle s'appelle `print`. Par exemple, dans un fichier portant le nom `test.py`, on peut écrire l'instruction suivante.

```
print(3)
```

On peut alors faire exécuter ce programme par l'interprète Python [4], ce qui affiche `3` à l'écran. L'instruction `print` accepte une expression arbitraire. Elle commence par calculer le résultat de cette expression puis l'affiche à l'écran. Ainsi, l'instruction

```
print(1+3)
```

calcule la valeur de l'expression `1+3` puis affiche `4` à l'écran.

Affichage de textes

L'instruction `print` n'est pas limitée à l'affichage de nombres. On peut lui donner un message à afficher, écrit entre guillemets. Par exemple, l'instruction

4. Voir le site `https://www.nsi-premiere.fr` pour les différentes manières d'appeler l'interprète Python sur un fichier.

© Éditions Ellipses
ISBN 978-2340033641

```
print("Salut tout le monde !")
```

affiche le message `Salut tout le monde !` à l'écran. Le texte écrit entre guillemets est appelé une *chaîne de caractères*. Les caractères accentués sont autorisés, tout comme les caractères provenant d'autres langues, les smileys et plus généralement tous les caractères Unicode. On note que les guillemets englobants ne sont pas affichés.

Il est important de comprendre que le contenu d'une chaîne est arbitraire et n'est pas interprété par Python. Pour s'en convaincre, on peut observer la différence entre l'expression arithmétique `1+3` et la chaîne de caractères `"1+3"` en exécutant l'instruction suivante

```
print("1+3")
```

qui affiche simplement `1+3` à l'écran.

Erreurs. Que la valeur à afficher soit donnée directement ou soit le résultat d'un calcul, les parenthèses font partie de la syntaxe de l'instruction `print`. Si on omet les parenthèses, on obtient une erreur.

```
File "test.py", line 1
  print 3
        ^
SyntaxError: Missing parentheses in call to 'print'
```

La première ligne du message d'erreur indique le nom du fichier et la ligne où se situe l'erreur.

Erreurs. Des guillemets ouverts doivent impérativement être fermés explicitement. On obtiendra sinon une erreur.

```
File "test.py", line 1
  print("1)
        ^
SyntaxError: EOL while scanning string literal
```

On peut aussi noter que les chaînes de caractères et les nombres sont des entités de natures différentes. Leur combinaison par une opération telle que l'addition n'a pas de sens, et provoque une erreur.

```
>>> 1 + "2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
 'int' and 'str'
```

©Éditions Ellipses
ISBN 978-2-340-03641

Séquence d'instructions

Un programme est généralement constitué de plusieurs instructions. Chaque instruction est écrite sur une ligne. L'interprète Python les exécute l'une après l'autre, dans l'ordre du fichier. Ainsi, l'exécution du programme

```
a = 34
b = 21 + a
print(a)
print(b)
```

affiche deux entiers à l'écran, sur deux lignes successives.

```
34
55
```

Si on souhaite afficher les deux entiers sur une même ligne, on peut remplacer les deux instructions `print` par une seule, en séparant les deux éléments à afficher par une virgule.

```
print(a, b)
```

Les deux éléments sont affichés sur la même ligne et séparés par un caractère espace.

```
34 55
```

Plus généralement, on peut utiliser `print` avec un nombre arbitraire d'éléments, qui peuvent être des nombres ou des chaînes de caractères.

```
a = 34
b = 55
print("la somme de", a, "et de", b, "vaut", a+b)
```

L'exécution de ce programme affiche le message suivant.

```
la somme de 34 et de 55 vaut 89
```

Interagir avec l'utilisateur

Pour permettre l'interaction du programme avec l'utilisateur, par exemple la saisie de la valeur d'une variable, il faut procéder en deux temps : d'abord utiliser l'instruction `input` pour récupérer des caractères tapés au clavier par l'utilisateur, puis utiliser l'instruction `int` pour convertir cette chaîne de caractères en un nombre entier.

```
s = input()
a = int(s)
print("le nombre suivant est ", a + 1)
```

© Éditions Ellipses
ISBN 978-2340033641

Instruction print et retour à la ligne. Par défaut, l'instruction **print** provoque un retour à la ligne après l'affichage. Ainsi deux instructions **print** successives affichent deux lignes. On peut changer ce comportement en fournissant une autre chaîne de caractères à accoler à l'affichage principal, par exemple une espace ou même rien. Ainsi le programme

```
print("abc", end=" ")
print("def", end="")
print("gh")
```

affiche la ligne suivante

```
abc defgh
```

puis revient à la ligne avec la troisième instruction **print** .

L'instruction `input` interrompt l'exécution du programme et attend que l'utilisateur saisisse des caractères au clavier. La saisie se termine lorsqu'il appuie sur la touche Entrée. Ici, la suite des caractères saisis par l'utilisateur est stockée dans une variable `s`. Dans un second temps, la variable `a` reçoit le nombre représenté par cette suite de caractères. Ainsi, si on exécute ce programme et qu'on saisit successivement les caractères `2`, `7` et la touche Entrée, le programme affiche la ligne suivante.

```
le nombre suivant est 28
```

Erreurs. Si la suite des caractères saisis par l'utilisateur ne représente pas un nombre, on obtiendra une erreur. Par exemple, si on saisit par accident le caractère `è` au lieu de `7`, on obtient le message suivant.

```
File "test.py", line 2
ValueError: invalid literal for int() with base 10: '2è'
```

En mode programme, lorsqu'une instruction provoque une erreur, l'interprète s'arrête, affiche le message d'erreur et n'exécute pas le reste du programme.

Un programme complet

Nous avons maintenant tous les ingrédients pour écrire un programme complet (voir encadré Programme 1). Il s'agit d'un programme qui demande son année de naissance à l'utilisateur, puis calcule et affiche son âge en 2048. La première ligne est un *commentaire*. C'est une séquence de caractères qui est ignorée par l'interprète et dont le but est uniquement de documenter le

ISBN 978-2-340-03641 © Editions ENI

Programme 1 — calcul de l'âge

```
1 # calcul de l'âge
2 saisie = input("Entrez votre année de naissance : ")
3 annee = int(saisie)
4 age = 2048 - annee   # on calcule l'âge par soustraction
5 print("Vous aurez", age, "ans en 2048.")
```

programme. En Python, un commentaire commence par un symbole `#` et se poursuit jusqu'à la fin de la ligne. Il peut contenir n'importe quelle suite de caractères. La deuxième ligne utilise l'instruction `input` pour demander son année de naissance à l'utilisateur. Comme on le voit, on peut ajouter à l'instruction `input` une chaîne de caractères, qui sera affichée juste avant la saisie. Le résultat de cette saisie est une chaîne de caractères stockée dans la variable `saisie`. La troisième ligne convertit la saisie en entier avec l'instruction `int`, et l'entier représentant l'année de naissance est stocké dans la variable `annee`. Le programme calcule ensuite l'âge de l'utilisateur en 2048 par une soustraction et le stocke dans la variable `age`. On note ici la présence d'un second commentaire, écrit sur la même ligne qu'une instruction. Là encore, le commentaire débute avec le symbole `#` et se poursuit jusqu'à la fin de la ligne. Enfin, la dernière ligne du programme affiche l'âge de l'utilisateur avec l'instruction `print`.

Représentation de l'exécution

On peut représenter l'exécution d'un programme complet par la séquence des états correspondant à chaque instruction exécutée. On présente une telle exécution dans un tableau où chaque ligne contient un numéro désignant l'instruction exécutée, l'état de l'interprète après l'exécution de l'instruction, et les éventuelles interactions avec l'utilisateur au cours de l'exécution.

Ligne	État	Interactions
2	`saisie` `"1985"`	affichage : `Entrez votre année de ...` saisie : `1985`
3	`saisie` `"1985"` `annee` `1985`	

4	`saisie` `"1985"`
	`annee` `1985`
	`age` `63`	

5	`saisie` `"1985"`
	`annee` `1985`
	`age` `63`
		affichage : `Vous aurez 63 ans en 2048.`

© Éditions ENI
ISBN 978-2-409-03...

À propos de l'opération `int`. L'opération `int` convertit une chaîne de caractères en un entier. La chaîne doit être uniquement composée de chiffres et éventuellement du symbole `-` en tête, sans quoi une erreur est levée. Par défaut, l'entier est lu en base 10 mais une autre base peut être spécifiée, avec la syntaxe `int(`*chaîne*`,` *base*`)`. Ainsi, `int("101010", 2)` interprète l'entier `101010` comme étant écrit en base 2, c'est-à-dire $2^5 + 2^3 + 2^1 = 42$. La base doit être comprise entre 2 et 36 et les lettres de `A` à `Z` sont utilisées pour représenter respectivement les chiffres de 10 à 35. Ainsi, `int("-2A", 16)` interprète l'entier `-2A` comme étant écrit en base 16, c'est-à-dire $-(2 \times 16 + 10) = -42$.
Si on souhaite interpréter la chaîne de caractères comme un nombre décimal plutôt que comme un nombre entier, il faut utiliser l'opération `float` à la place de l'opération `int`. Ainsi, `float("3.5")` est accepté là où `int("3.5")` provoque une erreur.

Composition. Deux instructions

```
s = input("Entrer un nombre :")
a = int(s)
```

peuvent être *composées* en une seule de la manière suivante.

```
a = int(input("Entrer un nombre :"))
```

L'effet obtenu est quasiment identique : cette version composée récupère de même une saisie de l'utilisateur sous la forme d'une chaîne de caractères et la convertit en un nombre entier, le nombre obtenu étant stocké dans la variable `a`. Dans cette dernière version cependant la chaîne de caractères intermédiaire n'est pas stockée dans une variable et n'est donc plus accessible par ailleurs.

Mode interactif et mode programme. Même si le mode programme est le mode d'utilisation standard, le mode interactif reste utile pour mettre au point un programme ou encore tester un élément du langage Python sur lequel on a un doute. Si on utilise l'environnement de développement Idle (voir `https://www.nsi-premiere.fr`), on dispose à la fois d'une fenêtre dans laquelle on écrit le texte du programme et d'une seconde fenêtre correspondant au mode interactif. Les deux modes peuvent donc être utilisés simultanément.

1.3 Bibliothèque Turtle

Python propose un certain nombres d'instructions primitives pour les besoins les plus courants, comme **print** et **input**. Le langage contient également de très nombreuses *bibliothèques*, qui apportent des collections d'instructions plus spécialisées permettant d'effectuer de nouvelles tâches.

Par exemple, il existe une bibliothèque `random` donnant accès à différentes instructions produisant des nombres aléatoires. Elle offre en particulier une instruction `randint` pouvant être utilisée sous la forme `n = random.randint(1, 6)` pour tirer au hasard un nombre entier entre `1` et `6` inclus, et stocker ce nombre dans la variable `n`, ou encore une instruction `random` pouvant être utilisée sous la forme `x = random.random()` pour tirer au hasard un nombre décimal entre `0` inclus et `1` exclu, et stocker ce nombre dans la variable `x`. L'utilisation de telles instructions demande une déclaration préalable, faite avec la ligne suivante.

```
import random
```

Cette ligne n'a besoin d'apparaître qu'une seule fois, en général au début du programme, pour chaque bibliothèque dont on voudra utiliser des instructions. Elle permet d'accéder à toutes les instructions de la bibliothèque.

Nous découvrirons différentes bibliothèques de Python à mesure des besoins de ce cours. Commençons ici par la bibliothèque `turtle`, qui permet de reproduire les fonctionnalités de base du langage de programmation éducatif Logo. Les instructions de ce langage font se déplacer une tortue munie d'un crayon à la surface d'une feuille virtuelle. On peut alors observer l'exécution du programme à travers les mouvements de la tortue et le tracé qu'elle laisse derrière elle.

Les instructions de `turtle` comprennent en premier lieu des moyens d'orienter et déplacer la tortue dans le plan cartésien à deux dimensions (en utilisant le repère standard des mathématiques, avec abscisses sur un axe horizontal croissant vers la droite et ordonnées avec axe vertical croissant vers le haut).

instruction	description
`goto(`x`, `y`)`	aller au point de coordonnées (x, y)
`forward(`d`)`	avancer de la distance d
`backward(`d`)`	reculer de la distance d
`left(`a`)`	pivoter à gauche de l'angle a
`right(`a`)`	pivoter à droite de l'angle a
`circle(`r`, `a`)`	tracer un arc de cercle d'angle a et de rayon r
`dot(`r`)`	tracer un point de rayon r

La tortue commence au point de coordonnées $(0, 0)$, situé au centre de l'écran, et est orientée par l'axe des abscisses. L'axe des ordonnées est orienté vers le haut. Les coordonnées et distances sont mesurées en pixels et les

© Éditions Ellipses
ISBN 978-2-340-03641

Variantes sur l'utilisation des bibliothèques. Il existe plusieurs moyen d'éviter d'écrire entièrement le nom de la bibliothèque dans chaque instruction, comme le `random.` de `random.randint` ou `random.random`.

- On peut associer un nom plus court à la bibliothèque au moment de la charger avec le mot clé `as`.

```
>>> import random as r
>>> r.randint(1, 6)
4
```

- On peut charger explicitement certaines des instructions avec la combinaison **`from / import`**. Par exemple ici, les instructions `sqrt` et `cos` de la bibliothèque `math`, qui calculent une racine carrée et un cosinus.

```
>>> from math import sqrt, cos
>>> sqrt(2)
1.4142135623730951
```

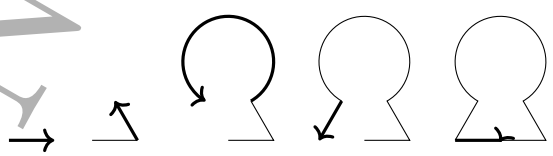
- On peut charger l'intégralité des instructions d'une bibliothèque en une seule fois avec une étoile `*`..

```
>>> from turtle import *
```

La dernière solution peut paraître pratique mais est plus délicate qu'il n'y paraît : deux bibliothèques peuvent contenir des instructions avec le même nom qui entreraient alors en conflit. Aussi on n'utilisera jamais cette possibilité sur plus d'une bibliothèque à la fois.

angles en degrés. Les arcs de cercles sont parcourus dans le sens trigonométrique si le rayon est positif, et sens horaire si le rayon est négatif. Le programme ci-dessous à gauche trace donc un dessin en suivant les étapes détaillées à droite.

```
from turtle import *
forward(60)
left(120)
forward(60)
right(90)
circle(60, 300)
right(90)
forward(60)
goto(0, 0)
```



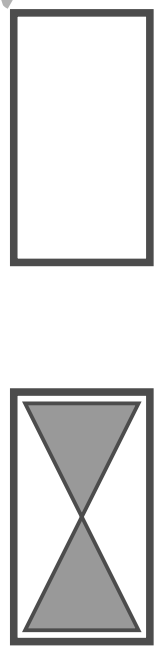
©Éditions ENI
ISBN 978-2-409-03641

S'ajoutent à cette base une série d'instructions permettant de modifier les dessins produits par chacun des déplacements.

instruction	description
`up()`	relever le crayon (et interrompre le dessin)
`down()`	redescendre le crayon (et reprendre le dessin)
`width(`e`)`	fixer à e l'épaisseur du trait
`color(`c`)`	sélectionner la couleur c pour les traits
`begin_fill()`	activer le mode remplissage
`end_fill()`	désactiver le mode remplissage
`fillcolor(`c`)`	sélectionner la couleur c pour le remplissage

Par défaut les tracés sont faits en noir avec une épaisseur d'un pixel. Les couleurs peuvent être désignées par leur nom, sous la forme de chaînes de caractères, avec notamment : `"black"`, `"white"`, `"grey"`, `"pink"`, `"purple"`, `"blue"`, `"green"`, `"yellow"`, `"orange"`, `"red"`, `"brown"`. Pour plus de souplesse on peut également fournir un triplet de nombres compris entre 0 et 1 indiquant les niveaux respectifs de rouge, vert et bleu composant la couleur : on obtient du rouge avec `color(1, 0, 0)`, un gris foncé avec `color(0.3, 0.3, 0.3)`, une certaine teinte de violet avec `color(0.5, 0, 0.6)`, etc. Toute l'aire contenue à l'intérieur de la trajectoire de la tortue pendant une période de temps où le mode remplissage est activé prend la couleur choisie pour le remplissage (noir par défaut).

```
from turtle import *
# Rectangle épais
width(6)
color(0.2, 0.2, 0.2)
goto(60, 0)
goto(60, 110)
goto(0, 110)
goto(0, 0)
# Déplacement
up()
goto(5, 5)
down()
# Sablier gris clair
width(1)
fillcolor("grey")
begin_fill()
goto(55, 5)
goto(5, 105)
goto(55, 105)
goto(5, 5)
end_fill()
```



© Éditions ENSSS
ISBN 978-2-3406-3641

Enfin, quelques instructions permettent de configurer l'action de la bibliothèque Turtle en général.

instruction	description
`reset()`	tout effacer et recommencer à zéro
`speed(`s`)`	définir la vitesse de déplacement de la tortue
`title(`t`)`	donner le titre t à la fenêtre de dessin
`ht()`	ne montre plus la tortue (seulement le dessin)

Les vitesses possibles sont à choisir parmi les chaînes de caractères `"slowest"`, `"slow"`, `"normal"`, `"fast"`, `"fastest"`, ou parmi les nombres entiers entre 0 et 10 : 1 est le déplacement le plus lent, 10 le déplacement le plus rapide, et 0 le déplacement instantané.

Erreurs. Utiliser des instructions Turtle avec des valeurs invalides peut produire des effets variés, souhaitables ou non.

- Utiliser une mauvaise chaîne de caractères ou une mauvaise valeur avec `color` ou `speed` peut déclencher une erreur immédiate. Le message d'erreur est assez massif et la majeure partie, éludée ici, fait référence aux mécanismes internes de Turtle qu'on ne détaillera pas. La dernière ligne contient en revanche un bilan du problème.

```
>>> color("rouge")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 8, in color
  ...
turtle.TurtleGraphicsError: bad color string: rouge
```

- Plus délicat, Python ne relève pas d'erreur lors de la définition d'une valeur négative pour l'épaisseur du trait.

```
width(-5)
```

L'erreur survient au moment de tracer un trait avec une telle épaisseur invalide. À nouveau le message est imposant, mais sa dernière ligne contient l'information essentielle pour nous.

```
>>> forward(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
_tkinter.TclError: bad screen distance "-5"
```

© Editions Ellipses
ISBN 978-2-340-03641

À retenir. Un **langage de programmation** permet d'écrire des programmes en vue de leur exécution par un ordinateur. En **mode interactif**, l'utilisateur fournit des instructions une à une et peut **observer** immédiatement les résultats, ce qui est intéressant notamment pour expérimenter. En **mode programme**, le programmeur rassemble toutes les instructions dans un **fichier source**. La séquence d'instructions est ensuite exécutée d'une seule traite, des opérations de **lecture** permettant de demander des informations à l'utilisateur et des opérations d'**affichage** permettant de faire un compte-rendu.
Une **variable** est un nom associé à une valeur. La valeur d'une variable peut être consultée, mais aussi mise à jour grâce à l'instruction d'**affectation**.

Activité : erreurs en situation contrôlée. Un programmeur débutant peut passer beaucoup de temps à traquer les erreurs dans ses programmes. Une bonne manière de se préparer à cela consiste à s'entraîner dans des situations contrôlées. Voici trois exercices à réaliser, et à reprendre régulièrement à toutes les étapes de l'apprentissage de la programmation.

1. Prendre un programme correct, c'est-à-dire un programme qui s'exécute sans erreur et produit le bon résultat, qui peut être issu de la leçon comme d'un exercice. Le modifier. Exécuter la version modifiée et observer quelles erreurs apparaissent ou en quoi le résultat change. Voici par exemple une version du programme 1 avec plusieurs modifications (à expérimenter une par une).

```
saisie = in("Entrez votre année de naissance: ")
annee = int(sasie)
age += 2048 - annee
print("Vous aurez", age, " en 2048)
```

2. Procéder comme au point précédent, mais essayer de prédire ce qui va se passer avant d'exécuter le programme modifié.
3. À deux personnes. La première prend le rôle de programmeur. Elle choisit un programme correct, le modifie pour introduire une erreur (ou plusieurs pour complexifier l'exercice), et donne le code modifié à la deuxième personne en lui précisant le résultat qui était attendu. La deuxième personne, dans le rôle de correcteur, doit trouver un moyen de corriger le programme pour obtenir le résultat voulu.

© Éditions ENI
ISBN 978-2340-0

Activité : suivi de l'exécution. L'apprentissage de la programmation nécessite d'assimiler l'articulation entre le programme qui n'est lui-même qu'un texte écrit inerte, statique, et son exécution qui est un processus actif, dynamique. L'une des difficultés est que le processus d'exécution contient généralement un grand nombre d'étapes, mais que nous n'en observons souvent que le résultat final avec par exemple l'affichage d'un résultat. Il est utile de savoir reproduire mentalement le cheminement intégral d'un programme, et les exercices suivants, à réaliser régulièrement, peuvent y aider.

1. Prendre un programme correct issu de la leçon ou d'un exercice, et tracer le tableau représentant chacune des étapes de son exécution. Commencer par exemple avec le programme 1 en choisissant pour la saisie son année de naissance au lieu de `1985`.
2. Procéder de même avec un programme incorrect, c'est-à-dire un programme produisant une erreur ou le mauvais résultat, et observer l'état au moment où le problème se manifeste et la suite des évènements ayant mené à cet état.
3. Prendre un programme, correct ou non, et y insérer des instructions **print** pour observer les valeurs de certaines variables à différentes étapes de son exécution.

Exercices

Exercice 1 Observer les résultats suivants donnés par l'interprète Python.

```
>>> 5 - 3 - 2
0
>>> 1 / 2 / 2
0.25
```

Qu'en déduire sur la manière dont sont interprétées les soustractions et les divisions enchaînées ? Écrire des expressions permettant d'observer la manière dont Python interprète les enchaînements mélangeant additions et soustractions, ou multiplications et divisions. Solution page 417 □

Exercice 2 Réécrire les expressions suivantes en explicitant toutes les parenthèses :

1. `1 + 2 * 3 - 4`
2. `1+2 / 4*3`
3. `1-a+a*a/2-a*a*a/6+a*a*a*a/24`

Solution page 417 □

©Éditions ENI Publishes
ISBN 978-2340053641

Exercice 3 Réécrire les expressions suivantes en utilisant aussi peu de parenthèses que possible sans changer le résultat.

1. `1+(2*(3-4))`
2. `(1+2)+((5*3)+4)`
3. `(1-((2-3)+4))+(((5-6)+((7-8)/2)))`

Solution page 417 □

Exercice 4 Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
>>> a = 3
>>> a = 4
>>> a = a+2
>>> a
```

Solution page 417 □

Exercice 5 Quelle est la valeur affichée par l'interprète après la séquence d'instructions suivante ?

```
>>> a = 2
>>> b = a*a
>>> b = a*b
>>> b = b*b
>>> b
```

Solution page 417 □

Exercice 6 Dans le mode interactif, initialiser une variable `a` avec la valeur 2, puis répéter dix fois l'instruction `a = a * a`. Observer le résultat. Quelle puissance de 2 a-t-on ainsi calculée ? Solution page 417 □

Exercice 7 Qu'affichent les instructions suivantes ?

1. `print ("1+")`
2. `print (1+)`

Solution page 417 □

Exercice 8 Que se passe-t-il quand on exécute le programme suivant ?

```
a = input("saisir un nombre : ")
print("le nombre suivant est ", a+1)
```

Le rectifier si nécessaire. Solution page 418 □

Exercice 9 Que fait la séquence d'instructions suivante ? On supposera qu'à l'origine les variables `a` et `b` contiennent chacune un nombre entier.

```
tmp = a
a = b
b = tmp
```

Solution page 418 □

© Éditions Ellipses
ISBN 978-2340033641

Exercice 10 On met deux entiers dans deux boîtes a et b, par exemple 55 et 89. On remplace le contenu de a par la somme de celui de a et de b. Puis on remplace le contenu de b par le contenu de a moins le contenu de b. Enfin, on remplace le contenu de a par son contenu moins celui de b. Que contiennent a et b à la fin de ces opérations ? Programme cet algorithme en Python.

Solution page 418 □

Exercice 11 Écrire un programme qui demande à l'utilisateur les longueurs (entières) des deux côtés d'un rectangle et affiche son aire.

Solution page 418 □

Exercice 12 Écrire un programme qui demande à l'utilisateur d'entrer une base (entre 2 et 36) et un nombre dans cette base et qui affiche ce nombre en base 10. La notation `int(`*chaîne*`, `*base*`)` permet de convertir une chaîne représentant un entier dans une base donnée en un entier Python.

Solution page 418 □

Exercice 13 Écrire un programme qui demande à l'utilisateur d'entrer un nombre de secondes et qui l'affiche sous la forme d'heures/minutes/secondes.

Solution page 418 □

Exercice 14 On souhaite écrire un programme qui demande à l'utilisateur un nombre d'œufs et affiche le nombre de boîtes de 6 œufs nécessaires à leur transport. On considère ce programme, qui utilise la division euclidienne.

```
n = int(input("combien d'oeufs : "))
print(n // 6)
```

Tester ce programme sur différentes entrées.

1. Sur quelles valeurs de `n` ce programme est-il correct ?
2. Pourquoi n'est-il pas correct de remplacer `n // 6` par `n // 6 + 1` ?
3. Proposer une solution correcte.

Solution page 419 □

Exercice 15 On souhaite calculer les coordonnées du point d'intersection de deux droites données sous la forme

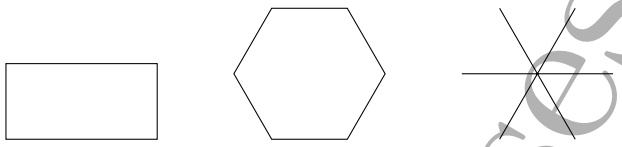
$$\begin{cases} y = ax + b \\ y = cx + d \end{cases}$$

On suppose ici que a, b, c et d sont des entiers. Écrire un programme qui demande ces quatre valeurs à l'utilisateur puis affiche les coordonnées du point d'intersection. Que se passe-t-il si les droites sont parallèles ?

Solution page 419 □

©Editions ELLIPSES
ISBN 978-2-340-03641

Exercice 16 Reproduire avec Turtle les dessins suivants.



Solution page 419 □

Exercice 17 Que traçe le programme Turtle suivant ?

```
goto(20, 0)
goto(0, 20)
goto(20, 20)
goto(0, 0)
goto(0, 20)
goto(10, 30)
goto(20, 20)
goto(20, 0)
```

Solution page 420 □

Exercice 18 Reproduire les dessins suivants avec Turtle, et en particulier les instructions `begin_fill()` et `end_fill()`.

Solution page 420 □

Exercice 19 Reproduire les dessins suivants avec Turtle, et en particulier l'instruction `width`.

Solution page 421 □

Exercice 20 Utiliser Turtle, et notamment les instructions `fill` et `fillcolor`, pour dessiner un drapeau français. Faire de même avec d'autres drapeaux au choix. Solution page 422 □

Exercice 21 Utiliser Turtle, et notamment les instructions `circle`, `color` et `width` pour dessiner un arc en ciel. Solution page 422 □

©Éditions Ellipses
ISBN 978-2-340-03641

Chapitre 5

Tableaux

Notions introduites

- notion de tableau
- construction d'un tableau
- accès à un élément et modification d'un élément
- taille d'un tableau
- parcours d'un tableau avec une boucle `for`

La mémoire de nos ordinateurs est vaste. Dans la mémoire d'un seul ordinateur, on peut stocker sans difficulté les noms et prénoms de tous les français ou encore l'intégralité des œuvres de Jules Verne. Pour donner un ordre de grandeur, le texte du *Tour du monde en quatre-vingts jours* contient un demi-million de caractères alors que la mémoire d'un ordinateur d'aujourd'hui peut en contenir plusieurs milliards et donc plusieurs milliers de romans.

Pour autant, nous avons jusqu'à présent utilisé une infime partie de cette mémoire gigantesque, avec seulement quelques variables. Bien sûr, rien ne nous empêche d'écrire des programmes avec un très grand nombre de variables mais cela atteint vite ses limites. Pour stocker de grandes quantités d'information, il faut se tourner vers d'autres solutions et le *tableau* est la plus simple d'entre elles.

5.1 Problème : la pyramide des âges

Considérons un programme qui stocke la pyramide des âges des français, c'est-à-dire la répartition de la population française par âge, et permet à l'utilisateur de la consulter. Par exemple, on peut saisir un âge et le programme affiche le nombre de français ayant cet âge-là, ou on saisit deux âges

© Editions Eyrolles
ISBN : 978-2-212-03364-1

et le programme affiche le nombre de français ayant un âge compris entre ces deux valeurs, etc. Pour stocker la pyramide des âges dans le programme, on pourrait utiliser autant de variables qu'il y a d'âges différents dans la pyramide [1] :

```
age0 = 691165  # moins de 1 an
age1 = 710534  # entre 1 et 2 ans
age2 = 728579  # entre 2 et 3 ans
...
```

et ainsi de suite jusqu'à l'âge maximum. Après tout, cela ne fait qu'un peu plus de cent variables. Là où les choses deviennent vraiment pénibles, c'est lorsque l'on veut demander à l'utilisateur du programme de saisir un âge, pour afficher ensuite le nombre de français ayant cet âge-là. On peut le faire, mais au prix d'une interminable succession de comparaisons.

```
a = int(input("quel âge : "))
if a == 0:
  print(age0)
elif a == 1:
  print(age1)
elif ...
```

Ce n'est pas raisonnable. D'une part, il faudrait recommencer une telle série de comparaisons pour tout autre calcul (par exemple, pour calculer le nombre de français ayant un âge compris entre deux valeurs données par l'utilisateur). D'autre part, si on avait maintenant plusieurs milliers, voire plusieurs millions d'informations, il deviendrait humainement impossible de procéder ainsi. Il nous faut une meilleure solution.

5.2 Notion de tableau

Un *tableau* permet de stocker plusieurs valeurs dans une seule variable et d'y accéder ensuite facilement. En Python, on construit un tableau en énumérant ses valeurs entre crochets et séparées par des virgules.

```
>>> t = [2, 3, 5]
```

Ici, on a déclaré une variable `t` contenant un tableau. Ce tableau contient trois entiers. Les valeurs contenues dans un tableau sont *ordonnées*. Ici, la première valeur contenue dans le tableau est `2`, la deuxième `3` et la troisième `5`. On peut se représenter un tableau comme des *cases* consécutives contenant des valeurs.

2	3	5

1. Ces chiffres correspondent à janvier 2018 et ont été téléchargés depuis le site `https://www.ined.fr/` de l'Institut National d'Études Démographiques.

© Éditions Ellipses
ISBN 978-2-340-03641

C'est en effet ainsi qu'un tableau est organisé dans la mémoire de l'ordinateur : ses valeurs y sont rangées consécutivement. Pour accéder à une valeur contenue dans le tableau `t`, il faut utiliser la notation `t[`i`]` où i désigne le numéro de la case à laquelle on veut accéder. Les cases sont *numérotées à partir de zéro*. Ainsi, la valeur contenue dans la première case est `t[0]`.

```
>>> t[0]
2
```

On dit que 0 est l'*indice* de la première case, 1 l'indice de la deuxième case, etc. La *taille* d'un tableau est son nombre de cases. On obtient la taille du tableau `t` avec l'opération **len**`(t)`.

```
>>> len(t)
3
```

Les indices d'un tableau `t` prennent donc des valeurs entre 0 et **len**`(t)`-1. On peut se représenter le tableau mentalement avec l'indice de chaque case indiqué juste au-dessus.

0	1	2
2	3	5

Mais il faut comprendre que seules les valeurs sont stockées dans la mémoire de l'ordinateur. Les indices, eux, n'ont pas besoin d'être matérialisés [2].

Erreurs. Si on cherche à accéder à une case en dehors des limites du tableau, on obtient une erreur.

```
>>> t[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Si on reprend notre exemple de la pyramide des âges, on peut la représenter très agréablement par un tableau `pda`.

```
>>> pda = [691165, 710534, 728579, ...]
```

Dans la case i du tableau `pda`, on trouve le nombre de français ayant exactement l'âge i. Maintenant, il devient très facile de demander un âge à l'utilisateur et d'afficher le nombre de français ayant cet âge-là.

2. L'idée est que chaque case du tableau occupe une taille identique et on peut donc calculer facilement où se trouve la case i dans la mémoire par une simple opération arithmétique.

©Éditions Ellipses
ISBN 978-2-340-03641

```
>>> age = int(input("quel âge : "))
quel âge : 42
>>> print("il y a", pda[age], "personnes ayant", age, "ans")
il y a 793019 personnes ayant 42 ans
```

Modification du contenu d'un tableau

Le contenu d'un tableau peut être modifié. Pour cela, on utilise une affectation, exactement comme on le ferait avec une variable. Ainsi, on peut modifier le contenu de la seconde case du tableau `t` pour y remplacer la valeur 3 par la valeur 17.

```
>>> t[1] = 17
```

On peut observer que cette modification a bien été effectuée en demandant à Python d'afficher le tableau `t`.

```
>>> t
[1, 17, 3]
```

Comme on le voit, un tableau est affiché avec la même forme que celle qui permet de le définir.

Si on reprend notre exemple de la pyramide des âges, on peut traduire une naissance en ajoutant 1 à la première case du tableau, c'est-à-dire à la case d'indice 0.

```
>>> pda[0] = pda[0] + 1
>>> pda
[691166, 710534, 728579, ...]
```

En première approximation, on peut donc voir les cases d'un tableau comme autant de variables qu'on peut modifier à loisir, qui seraient appelées ici `pda[0]`, `pda[1]`, etc. Mais avec une différence essentielle : l'indice peut être le résultat d'un calcul.

5.3 Parcours d'un tableau

Toujours sur notre exemple de la pyramide des âges, supposons que nous voulions calculer le nombre de français ayant entre 10 et 20 ans. On obtient ce nombre en additionnant dix valeurs contenues dans le tableau.

```
>>> pda[10] + pda[11] + pda[12] + ... + pda[18] + pda[19]
8035093
```

C'est un peu long à écrire, mais cela reste faisable. Mais cela deviendrait vraiment pénible si la plage était plus grande ou encore saisie interactivement par l'utilisateur.

© Éditions Ellipses
ISBN 978-2-340-03641

Python et les autres langages. Les tableaux de Python diffèrent des tableaux que l'on trouve dans les autres langages de programmation par plusieurs aspects. Tout d'abord, ils sont appelés *listes* dans la documentation de Python, ce qui est assez malheureux car une liste désigne en général une structure de données différente du tableau.
Ensuite, les tableaux de Python peuvent être agrandis ou rétrécis du côté droit, c'est-à-dire du côté des indices les plus grands (avec des opérations `append` et `pop` que nous n'avons pas présentées). Cela les distingue des tableaux usuels, où la taille est fixée une fois pour toutes à la création.
Enfin, accéder à un tableau Python avec un indice négatif ne provoque pas nécessairement une erreur, contrairement à ce que l'on pourrait imaginer. Python permet en effet d'accéder au dernier élément du tableau `t` avec `t[-1]`, à son avant-dernier élément avec `t[-2]`, etc. De manière générale, les indices -1 à $-n$ peuvent être utilisés pour accéder à partir de la droite à un tableau de taille n. C'est parfois utile mais aussi dangereux : il suffit d'une petite erreur de calcul dans un programme pour se retrouver avec un indice -1 plutôt que 0, par exemple, et l'exécution du programme va alors se poursuivre sans signaler l'erreur. Pour un tableau de taille n, seul un indice en dehors de l'intervalle $[-n, n-1]$ provoquera une erreur.
Nous avons délibérément choisi de nous en tenir uniquement au vocabulaire et notions usuels des tableaux tels qu'on les trouve dans tous les langages.

Une meilleur solution consiste à utiliser *une boucle* pour parcourir les cases du tableau concernées, tout en accumulant le nombre total dans une variable. On commence par introduire cette variable, en l'initialisant à zéro.

```
>>> n = 0
```

Puis on effectue une boucle **for** donnant successivement à la variable `i` toutes les valeurs entre 10 et 19, et on ajoute à chaque fois la valeur `pda[i]` à la variable `n`.

```
>>> for i in range(10, 20):
        n += pda[i]
```

Le nom `i` donné à la variable importe peu. On aurait pu l'appeler `age` par exemple. Mais il est courant d'utiliser les noms `i`, `j` ou `k` pour des variables qui vont être utilisées comme indices dans des tableaux. On peut vérifier qu'après la boucle on a bien calculé dans la variable `n` la même valeur que celle obtenue plus haut avec notre addition de dix valeurs du tableau.

```
>>> n
8035093
```

©Edition ISBN 978-2-[illegible]641

Erreurs. La possibilité d'accéder aux cases d'un tableau par des indices négatifs rend certaines erreurs difficiles à analyser. Imaginons une fonction

```
def nb_occurrences(v, a, b, t):
    nb = 0
    for i in range(a, b):
        if t[i] == v:
            nb += 1
    return nb
```

renvoyant le nombre d'occurrences de la valeur `v` entre les indices `a` (inclus) et `b` (exclu) du tableau `t`. Supposons qu'un programmeur écrive alors la séquence suivante.

```
t = [2, 1, 3, 1]
debut = -1
fin = len(t)
n = nb_occurrences(1, debut, fin, t)
```

Que se passe-t-il ? Ici, le dernier élément sera compté deux fois, une pour l'indice `-1` et une pour l'indice `3`, et on obtiendra le résultat fantasque de trois occurrences de la valeur `1` dans le tableau `[2, 1, 3, 1]`. Cette valeur erronée pour `n` est susceptible de provoquer une erreur plus tard dont il peut être compliqué de retrouver l'origine, à savoir ici une mauvaise définition de la variable `debut`.

Programme 8 — pyramide des âges

```
pda = [691165, 710534, 728579, ..., 2160]   # 106 valeurs
age_min = int(input("âge minimum (inclus) : "))
age_max = int(input("âge maximum (exclu) : "))
n = 0
for age in range(age_min, age_max):
    n += pda[age]
print("il y a", n, "personnes qui ont entre", \
      age_min, "et", age_max, "ans")
```

Le programme 8 contient une version plus générale de ce calcul, où la plage d'âge est donnée par l'utilisateur. (L'exercice 69 page 84 propose d'améliorer un peu ce programme.)

ISBN : 978-2-340-03641 © Ellipses

Un cas particulier consiste à parcourir *toutes* les cases d'un tableau `t` avec une boucle **for** allant de 0 inclus à **len**`(t)` exclu. On peut calculer ainsi la population française en janvier 2018 en parcourant tout notre tableau `pda`.

```
>>> popu = 0
>>> for i in range(0, len(pda)):
      popu += pda[i]
>>> popu
65018096
```

5.4 Construire de grands tableaux

Si on doit construire un tableau vraiment grand, par exemple de plusieurs centaines d'éléments, il devient difficile de le faire en énumérant tous ses éléments. Fort heureusement, il existe une opération dans le langage Python pour construire un tableau d'une taille arbitraire. Elle s'utilise ainsi :

```
>>> t = [0] * 1000
```

On donne la taille du tableau après le symbole `*`, ici 1000, et entre crochets une valeur qui sera donnée à toutes les cases du tableau, ici 0. On obtient donc ici un tableau `t` de taille 1000 dont toutes les cases contiennent pour l'instant la valeur 0. On peut notamment vérifier que le tableau ainsi construit a bien la taille 1000.

```
>>> len(t)
1000
```

Il faut bien comprendre que l'opération `[0]*1000` n'implique pas une multiplication, même si elle utilise le même symbole `*`. Il s'agit là d'une opération spécifique aux tableaux. Sa syntaxe n'est cependant pas liée au hasard ; elle est là pour suggérer l'idée que l'on « multiplie par 1000 » un tableau d'une case contenant 0.

Une fois le tableau ainsi construit, on peut maintenant le *remplir* avec des valeurs de son choix, en utilisant des affectations. Si par exemple on veut y stocker les carrés des 1000 premiers entiers, on peut le faire avec une boucle.

```
>>> for i in range(0, 1000):
      t[i] = i * i
```

Dans le chapitre 8, nous verrons une autre façon de construire un tel tableau.

Enfin, mentionnons qu'il est possible de *concaténer* deux tableaux, c'est-à-dire de construire un nouveau tableau contenant, bout à bout, les éléments des deux tableaux. On le fait avec l'opération `+`.

```
>>> [8, 13, 21] + [34, 55]
[8, 13, 21, 34, 55]
```

©Editions Ellipses
ISBN 978-2340033641

Comme pour l'opération * utilisée pour la construction de tableaux, on réutilise ici un symbole arithmétique, mais il n'y a pas d'addition à proprement parler.

Tableaux et chaînes de caractères. Les chaînes de caractères offrent une certaine ressemblance avec les tableaux. En particulier, on peut obtenir la taille d'une chaîne de caractères, c'est-à-dire son nombre de caractères, avec l'opération **len** et accéder au i-ième caractère d'une chaîne avec les crochets.

```
>>> ch = "bonjour"
>>> len(ch)
7
>>> ch[2]
'n'
```

Comme on le voit sur cet exemple, les caractères sont numérotés à partir de zéro, comme dans un tableau. On peut également concaténer deux chaînes avec `+` ou encore construire une chaîne contenant 10 répétitions de `ab` avec `"ab" * 10`.
En revanche, contrairement aux tableaux, les caractères d'une chaîne ne peuvent pas être modifiés.

```
>>> ch[2] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

5.5 Tableaux et variables

Comme on l'a expliqué au chapitre 1, une variable déclarée avec `x = 1` peut être représentée par une boîte appelée `x` contenant la valeur 1.

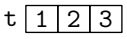
Lorsque l'on modifie la valeur de la variable `x`, par exemple avec l'affectation `x = x + 1`, la valeur 1 a été remplacée par la valeur 2.

Lorsque l'on affecte à une nouvelle variable `y` la valeur de `x`, avec l'instruction `y = x`, une seconde boîte est créée, qui reçoit la valeur de `x`, c'est-à-dire 2.

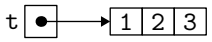
©Éditions ENI
ISBN 978-2-409-00241-1

On a maintenant deux variables indépendantes. En particulier, modifier la variable `y`, par exemple avec `y = 3`, n'a pas d'effet sur la variable `x`.

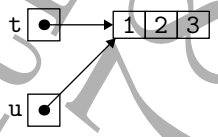
Si nous en reparlons ici, c'est parce la situation devient un peu plus subtile lorsque les variables contiennent des tableaux. Naïvement, on pourrait penser qu'une variable `t` contenant un tableau, par exemple initialisée avec `t = [1, 2, 3]`, désigne trois boîtes plutôt qu'une seule, ce que l'on pourrait se représenter ainsi.



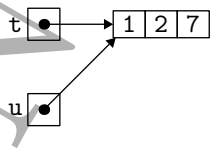
Malheureusement, cette vision des choses est incorrecte. En réalité, la valeur affectée à la variable `t` est l'*adresse mémoire* de l'espace alloué au tableau. La valeur précise de cette adresse importe peu et on peut donc se le représenter ainsi :



La flèche symbolise ici le fait que la variable `t` contient une valeur qui désigne l'emplacement mémoire où se trouve le tableau. En première approximation, cette distinction que nous venons de faire paraît inutile. Après tout, on visualise aussi bien l'accès à `t[1]` ou encore l'affectation `t[2] = 7` avec notre premier schéma. Mais considérons maintenant la création d'un nouveau tableau avec `u = t`. Comme avec les variables entières `x` et `y` plus haut, la variable `u` reçoit *la valeur* de la variable `t`. Mais comme la valeur est ici une adresse mémoire, on se retrouve avec la situation suivante :



Autrement dit, les *deux* variables `t` et `u` désignent *le même tableau*[3]. En particulier, toute modification du contenu du tableau `t` sera visible dans le tableau `u`. Ainsi, si on exécute l'instruction `t[2] = 7`, on se retrouve avec la situation suivante.



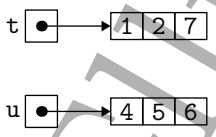
On a donc également modifié la valeur de `u[2]`. Le lecteur sceptique peut s'en convaincre avec quelques lignes dans l'interprète Python.

3. En informatique, on parle d'*aliasing* pour désigner ce phénomène où plusieurs noms permettent d'accéder à une même donnée.

© Editions Ellipses
ISBN 978-2-340-03641

```
>>> t = [1, 2, 3]
>>> u = t
>>> t[2] = 7
>>> u
[1, 2, 7]
```

Les tableaux `t` et `u` ne sont pas pour autant destinés à rester identiques éternellement. Rien ne nous empêche par exemple d'affecter un nouveau tableau à `u`, par exemple avec l'instruction `u = [4, 5, 6]`, pour se retrouver alors dans la situation suivante.



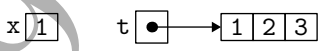
Même s'il est peu fréquent, voire peu recommandé, de se retrouver ainsi avec deux variables qui désignent le même tableau, il est fondamental d'avoir compris cet aspect-là du fonctionnement de Python. C'est en particulier nécessaire lorsque l'on utilise des fonctions qui reçoivent des tableaux en argument, ce que nous allons expliquer maintenant.

Tableaux et fonctions

Illustrons le passage d'un tableau en argument d'une fonction à l'aide d'un petit exemple. Considérons une variable entière `x` et un tableau `t`, initialisées ainsi :

```
x = 1
t = [1, 2, 3]
```

Conformément à ce que nous avons expliqué plus haut, nous pouvons l'illustrer de la manière suivante :

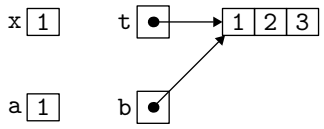


Définissons maintenant une fonction `f` prenant deux arguments, appelés `a` et `b`. La variable `a` est supposée recevoir un entier et la variable `b` un tableau.

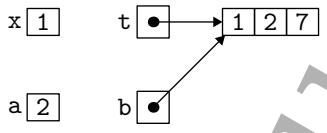
```
def f(a, b):
    a = a + 1
    b[a] = 7
```

Comme on le voit, la fonction `f` commence par incrémenter la variable `a` puis modifie le tableau `b` à l'indice `a`. Considérons maintenant l'appel de fonction `f(x, t)`. Juste après l'appel, on a la situation suivante où deux nouvelles variables `a` et `b` viennent de recevoir les valeurs des variables `x` et `t`.

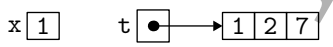
© Éditions Ellipses
ISBN 978-2340033641



Ceci est conforme à ce qui a été expliqué dans le chapitre 4 sur les fonctions, ainsi qu'à ce qui a été expliqué plus haut sur la valeur d'une variable désignant un tableau. Une fois que les deux instructions qui constituent la fonction `f` ont été exécutées, on se retrouve donc dans la situation suivante :



La variable `a` a été incrémentée, et contient maintenant la valeur 2, et la valeur 7 a été affectée à la case d'indice 2 du tableau. Une fois l'appel de fonction terminé, les variables `a` et `b` disparaissent et on se retrouve avec les valeurs suivantes pour les variables `x` et `t`.



Comme on le constate, le contenu du tableau `t` a été modifié par la fonction, mais pas le contenu de la variable `x`. Nous venons d'illustrer ici quelque chose d'important :

> Une fonction peut modifier le contenu d'un tableau qui lui est passé en argument.

Par la suite, nous allons écrire de nombreuses fonctions opérant ainsi sur les tableaux. Un exemple significatif est celui d'une fonction qui trie le contenu d'un tableau, ce que nous ferons au chapitre 10. D'autres exemples plus simples sont proposés en exercice dans ce chapitre (par exemple l'exercice 80).

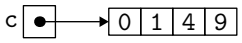
De même qu'une fonction peut recevoir un tableau en argument, une fonction peut renvoyer un tableau comme résultat. Là encore, il convient de bien comprendre ce qui se passe, en l'illustrant de façon précise. Considérons la fonction suivante qui reçoit un entier `n` en argument et renvoie un tableau de taille `n` contenant les carrés des `n` premiers entiers.

```
def carres(n):
    t = [0] * n
    for i in range(n):
        t[i] = i * i
    return t
```

Supposons que l'on appelle cette fonction avec 4 comme argument et que l'on stocke le résultat dans une variable `c`, c'est-à-dire que l'on exécute l'instruction `c = carres(4)`. Au moment où l'exécution de la fonction atteint l'instruction `return`, on est dans la situation suivante :

©Editions Ellipses ISBN 978-2-340-03641

La valeur renvoyée par la fonction est la valeur de la variable `t`, c'est-à-dire l'adresse du tableau contenant les quatre carrés. C'est cette valeur qui est stockée dans la variable `c` juste après l'appel.



Autrement dit, la variable locale `t` qui a servi à la construction du tableau n'a pas survécu à l'appel mais en revanche le tableau lui-même, comme zone mémoire contenant ici quatre entiers, a survécu à l'appel. Plusieurs exercices de ce chapitre proposent d'écrire des fonctions qui renvoient ainsi des tableaux (par exemple l'exercice 78).

À retenir. Un **tableau** permet de regrouper plusieurs valeurs sous la forme d'une **séquence ordonnée** dans laquelle chaque valeur est associée à un **indice** ou numéro. Les indices permettent d'accéder aux valeurs contenues dans un tableau, pour les **consulter** ou les **modifier**. On peut utiliser une **boucle** pour examiner tour à tour les différentes valeurs contenues dans un tableau.

Exercices

Exercice 69 Le programme 8 page 78 n'est pas très robuste. En effet, si l'utilisateur donne pour `age_max` une valeur plus grande que 106 (la taille du tableau `pda`), alors le programme va échouer suite à un accès en dehors des limites du tableau. Améliorer ce programme pour que l'utilisateur puisse spécifier une valeur de `age_max` arbitrairement grande, sans pour autant que le programme n'échoue. Solution page 437 □

Exercice 70 Écrire une fonction `occurrences(v, t)` qui renvoie le nombre d'occurrences de la valeur `v` dans le tableau `t`. Solution page 437 □

Exercice 71 Écrire un programme qui construit un tableau de 100 entiers tirés au hasard entre 1 et 1000, puis l'affiche. Solution page 437 □

Exercice 72 Compléter le programme précédent pour calculer et afficher l'élément maximum de ce tableau. Solution page 437 □

Exercice 73 Écrire un programme qui tire au hasard mille entiers entre 1 et 10 et affiche ensuite le nombre de fois que chaque nombre a été tiré. Relancer le programme plusieurs fois. Solution page 438 □

© Ellipses
ISBN 978-2-340-03641

Exercice 74 En mathématiques, la très célèbre *suite de Fibonacci* est une séquence infinie d'entiers définie de la façon suivante : on part des deux entiers 0 et 1 puis on construit à chaque fois l'entier suivant comme la somme des deux entiers précédents.

$$0,\ 1,\ 1,\ 2,\ 3,\ 5, \ldots$$

Écrire un programme qui construit puis imprime un tableau contenant les 30 premiers termes de la suite. Le dernier élément de ce tableau doit être 514 229. Solution page 438 □

Exercice 75 Écrire une fonction `copie(t)` qui prend en paramètre un tableau `t` et renvoie une copie de ce tableau. Quelle expérience peut-on faire pour s'assurer qu'on ne s'est pas trompé ? Solution page 438 □

Exercice 76 Écrire une fonction `ajout(v, t)` qui crée un nouveau tableau contenant d'abord tous les éléments de `t` puis `v`. Solution page 439 □

Exercice 77 Écrire une fonction `concatenation(t1, t2)` qui crée un nouveau tableau contenant, dans l'ordre, tous les éléments de `t1` puis tous les éléments de `t2`. Solution page 439 □

Exercice 78 Écrire une fonction `tableau_aleatoire(n, a, b)` qui renvoie un tableau de taille `n` contenant des entiers tirés au hasard entre `a` et `b`. Solution page 439 □

Exercice 79 Écrire une fonction `tableau_croissant(n)` qui renvoie un tableau de taille `n` contenant des entiers tirés au hasard et ayant la propriété d'être trié par ordre croissant. Pour faire cela, on pourrait utiliser l'exercice précédent pour construire un tableau aléatoire, puis le trier avec un algorithme de tri (ce que nous verrons dans le chapitre 10). Il y a néanmoins une façon plus simple de procéder, consistant à remplir le tableau de gauche à droite en ajoutant à chaque fois un entier positif ou nul à l'élément précédent. Ainsi, le tableau est trié par construction. Solution page 439 □

Exercice 80 Écrire une fonction `echange(tab, i, j)` qui échange dans le tableau `tab` les éléments aux indices `i` et `j`. Solution page 440 □

Exercice 81 Écrire une fonction `somme(tab)` qui calcule et renvoie la somme des éléments d'un tableau d'entiers. En déduire une fonction `moyenne(tab)` qui calcule et renvoie la moyenne des éléments du tableau `tab`, supposé non vide. Solution page 440 □

Exercice 82 Écrire une fonction `produit(tab)` qui calcule et renvoie le produit des éléments d'un tableau d'entiers. Si le tableau contient 0, la fonction devra renvoyer 0 sans terminer le calcul. Solution page 440 □

©Éditions Eyrolles
ISBN 978-2-340-03641

Exercice 83 Écrire une fonction `miroir(tab)` qui reçoit un tableau en argument et le modifie pour échanger le premier élément avec le dernier, le second avec l'avant-dernier, etc. Dit autrement, on remplace le tableau par son image miroir. On pourra se servir de la fonction `echange` de l'exercice précédent. Solution page 440 □

Exercice 84 Pour mélanger les éléments d'un tableau aléatoirement, il existe un algorithme très simple qui procède ainsi : on parcourt le tableau de la gauche vers la droite et, pour chaque élément à l'indice i, on l'échange avec un élément situé à un indice tiré aléatoirement entre 0 et i (inclus). Écrire une fonction `melange(tab)` qui réalise cet algorithme. On pourra se resservir de la fonction `echange` de l'exercice 80. (Cet algorithme s'appelle le *mélange de Knuth*.) Solution page 441 □

Exercice 85 Écrire une fonction `prefixe(tab1, tab2)` qui renvoie `True` si le tableau `tab1` est un préfixe du tableau `tab2`, c'est-à-dire si le tableau `tab2` commence par les éléments du tableau `tab1` dans le même ordre. Solution page 441 □

Exercice 86 Écrire une fonction `suffixe(tab1, tab2)` qui renvoie `True` si le tableau `tab1` est un suffixe du tableau `tab2`, c'est-à-dire si le tableau `tab2` termine par les éléments du tableau `tab1` dans le même ordre. Solution page 441 □

Exercice 87 Écrire une fonction `hamming(tab1, tab2)` qui prend en paramètres deux tableaux, que l'on supposera de la même taille, et qui renvoie le nombre d'indices auxquels les deux tableaux diffèrent. Solution page 441 □

Exercice 88 Reprendre l'exercice précédent, sans supposer que les tableaux ont la même taille. On considérera qu'un indice auquel seul l'un des tableaux est défini compte pour une différence. Solution page 442 □

Exercice 89 Reprendre l'exercice 65 page 72 en utilisant un tableau donnant le nombre de jours de chaque mois. Solution page 442 □

©Éditions ENSSAT
ISBN 978-2-3403-3641

Solutions des exercices

Exercice 1, page 21 Les opérations arithmétiques `+`, `-`, `*` et `/` sont associatives à gauche : les expressions `1+2+3`, `5-3-2` et `1/2/2` sont comprises respectivement comme `(1+2)+3`, `(5-3)-2` et `(1/2)/2`.

Pour observer les interactions entre `+` et `-` on peut prendre `1-2+3` (donnerait `-4` si `+` était plus prioritaire que `-`). Pour observer les interactions entre `*` et `/` on peut prendre `6/3*2` (donnerait `1.0` si `*` était plus prioritaire que `/`).

Exercice 2, page 21

1. `(1+(2*3))-4`
2. `1+((2/4)*3)`
3. `(((1-a)+((a*a)/2))-(((a*a)*a)/6))+(((a*a)*a)*a)/24)`

Exercice 3, page 22

1. `1+2*(3-4)`
2. `1+2+5*3+4`
3. `1-(2-3+4)+5-6+(7-8)/2`

Exercice 4, page 22 La valeur affichée est `6`.

Exercice 5, page 22 La valeur affichée est `64`.

Exercice 6, page 22 Le résultat est un entier de 309 chiffres. On a calculé successivement 2^1, 2^2, 2^4, 2^8, etc., jusqu'à 2^{1024}.

Exercice 7, page 22

1. Affiche `1+`.
2. Erreur de syntaxe.

©Éditions Ellipses
ISBN 978-2340033641

Exercice 8, page 22 Tenter d'ajouter comme ici un chaîne de caractères et un entier provoque une erreur. Pour corriger l'erreur, il faut convertir la chaîne en entier avec l'instruction `int`.

```
a = input("saisir un nombre : ")
b = int(a)
print("le nombre suivant est ", b+1)
```

Exercice 9, page 22 Ce code échange les valeurs des variables `a` et `b`, en se servant d'une troisième variable comme zone de stockage temporaire. Le nom de cette variable, `tmp`, évoque son caractère temporaire. Il y a d'autres façons de le faire. Une syntaxe particulière de Python, que nous n'avons pas expliquée, permet des affectations multiples comme `a, b = 55, 89`. En particulier, on peut écrire `a, b = b, a` pour échanger le contenu de `a` et `b`.

Exercice 10, page 22 En déroulant cet algorithme sur un exemple, on observe qu'il échange le contenu des deux boîtes. Le code Python est immédiat :

```
a = a + b
b = a - b
a = a - b
```

Exercice 11, page 23

```
longueur = int(input("longueur (en mètres) : "))
largeur  = int(input("largeur  (en mètres) : "))
aire = longueur * largeur
print("L'aire du rectangle vaut", aire, "mètres carrés.")
```

Exercice 12, page 23

```
b = int(input("entrer la base (entre 2 et 36) : "))
s = input("entrer le nombre : ")
n = int(s,b)
print("Le nombre", s, "en base", b, "s'écrit", n, "en base 10.")
```

Exercice 13, page 23

```
n = int(input("entrer le nombre de secondes : "))
h = n // 3600  # nombre d'heures (entières)
n = n % 3600   # nombre de secondes restantes
m = n // 60    # nombre de minutes entières
s = n % 60     # nombre de secondes restantes
print(h, "heures", m, "minutes", s, "secondes")
```

© Éditions Ellipses
ISBN 978-2-340-03641

Exercice 14, page 23

1. Ce programme n'est correct que lorsque `n` est multiple de 6. Par exemple, il répond incorrectement 1 pour $\texttt{n} = 7$.
2. Avec cette modification, le programme est maintenant correct pour les valeurs de `n` qui ne sont pas multiples de 6, mais devient en revanche incorrect lorsque `n` est multiple de 6. Par exemple, il répond maintenant 3 pour $\texttt{n} = 12$.
3. Un programme correct est

```
n = int(input("combien d'oeufs : "))
print((n + 5) // 6)
```

   On pourra s'en convaincre empiriquement ou par un raisonnement arithmétique, en distinguant les cas où n est multiple de 6 ou non. On note que le programme reste correct pour $\texttt{n} = 0$.

Exercice 15, page 23 C'est de l'algèbre élémentaire, consistant par exemple à isoler x dans l'identité $ax + b = cx + d$.

```
a = int(input("valeur de a : "))
b = int(input("valeur de b : "))
c = int(input("valeur de c : "))
d = int(input("valeur de d : "))
x = (b - d) / (c - a)
y = a * x + b
print(x, ",", y)
```

Si les droites sont parallèles, cela veut dire que les coefficients `a` et `c` sont égaux et donc que le programme tente une division par zéro. On obtient alors une erreur :

```
ZeroDivisionError: division by zero
```

Exercice 16, page 24 Pour un rectangle on peut facilement utiliser les coordonnées.

```
goto(120,0)
goto(120,60)
goto(0,60)
goto(0,0)
```

Pour l'hexagone on évite de faire les calculs à la main en se basant sur les angles et les distances. Il suffit alors d'écrire six fois les deux lignes suivantes.

```
forward(40)
left(60)
```

©Editions Ellipses
ISBN 978-2340033641

De même pour le flocon, mais il faut en plus relever le crayon ou revenir au centre entre chaque trait. Chaque trait est dessiné avec

```
forward(80)
```

et à la fin d'un trait on peut se replacer et se réorienter avec

```
up()
left(120)
forward(40)
down()
left(120)
```

avant de dessiner le suivant.

Exercice 17, page 24 Une enveloppe dessinée sans lever le crayon ni repasser deux fois sur le même trait.

Exercice 18, page 24 Papillon : faisable en traçant séparément chaque triangle ou en regroupant les lignes droites.

```
begin_fill()
goto(100, 100)
goto(50, 0)
goto(-50, 100)
goto(0, 0)
end_fill()
```

Trois triangles reliés par un sommet. On peut écrire trois fois les lignes suivantes entre `begin_fill()` et `end_fill()`.

```
forward(100)
left(120)
forward(50)
left(120)
```

Trois triangles en pyramide, solution sans lever le crayon.

```
begin_fill()
left(60)
forward(50)
right(120)
forward(50)
left(120)
```

```
forward(50)
left(120)
forward(50)
right(120)
forward(50)
right(120)
```

```
forward(100)
right(120)
forward(100)
end_fill()
```

© Éditions Ellipses
ISBN 978-2-340-03364-1

Yin et yang : à base de demi-cercles.

```
# Forme principale
left(180)
begin_fill()
circle(-100, 180)
circle(-50, 180)
circle(50, 180)
end_fill()
circle(100, 180)
```

```
# Point blanc
up()
goto(0, 65)
down()
begin_fill()
circle(15)
end_fill()
```

```
# Point noir
up()
goto(0, 165)
fillcolor("white")
begin_fill()
circle(15)
end_fill()
```

Exercice 19, page 24 Croix : le trait épais donne les bouts arrondis.

```
# Carré
begin_fill()
goto(200, 0)
goto(200, 200)
goto(0, 200)
goto(0, 0)
end_fill()
```

```
# Trait horizontal
color("white")
width(10)
up()
goto(50, 100)
down()
goto(150, 100)
```

```
# Trait vertical
up()
goto(100, 50)
down()
goto(100, 150)
```

Point d'interrogation : arcs de cercles.

```
# Forme principale
width(10)
left(90)
circle(-30, 240)
circle(30, 60)
forward(20)
```

```
# Point
up()
forward(20)
down()
dot(10)
```

Tétraèdre : version sans calculer explicitement les coordonnées. On peut copier trois fois la séquence suivante.

```
# Trait extérieur
width(6)
forward(100)
left(150)
```

```
# Trait intérieur
width(2)
forward(58)
backward(58)
right(30)
```

©Éditions Eyrolles
ISBN 978-2340033641

Exercice 20, page 24 Drapeau français.

```
# Cadre
goto(150, 0)
goto(150, 100)
goto(0, 100)
goto(0, 0)
```

```
# Rectangle bleu
fillcolor("blue")
begin_fill()
goto(50, 0)
goto(50, 100)
goto(0, 100)
goto(0, 0)
end_fill()
```

```
# Rectangle rouge
goto(100, 0)
fillcolor("red")
begin_fill()
goto(150, 0)
goto(150, 100)
goto(100, 100)
goto(100, 0)
end_fill()
```

Exercice 21, page 24 Une fois la largeur du trait fixée, on affiche des arcs de plus en plus petits en faisant attention à l'orientation.

```
# Épaisseur, départ
width(10)
left(90)
# Premier arc
color("violet")
circle(200, 180)
# Replacement
up()
goto(-10, 0)
left(180)
down()
# Deuxième arc
color("blue")
circle(190, 180)
```

```
# Replacements
# et arcs 3 et 4
up()
goto(-20, 0)
left(180)
down()
color("green")
circle(180, 180)
up()
goto(-30, 0)
left(180)
down()
color("yellow")
circle(170, 180)
```

```
# Replacements
# et arcs 5 et 6
up()
goto(-40, 0)
left(180)
down()
color("orange")
circle(160, 180)
up()
goto(-50, 0)
left(180)
down()
color("red")
circle(150, 180)
```

Exercice 22, page 36 On calcule cela dans une variable `p` que l'on initialise avec 1.

```
p = 1
for _ in range(n):
p = 2 * p
print(p)
```

Exercice 23, page 36 On calcule cela dans une variable `f` que l'on initialise avec 1.

©Éditions Ellipses
ISBN 978-2-340-03641

2. Trois triangles, avec replacement entre chaque (sans `up` ni `down` car on ne traverse que des zones déjà couvertes).

```
triangle(20)
forward(20)
triangle(20)
left(120)
forward(20)
right(120)
triangle(20)
```

3. Trois triangles, de haut en bas.

```
right(120)
triangle(30)
left(30)
forward(10)
right(30)
triangle(50)
left(30)
forward(15)
right(30)
triangle(80)
```

Exercice 69, page 84 Il suffit de stopper le parcours au minimum de `age_max` et **len**`(pda)`.

```
for age in range(age_min, min(age_max, len(pda))):
```

Le reste du programme est inchangé.

Exercice 70, page 84 On utilise une variable `occ` comme accumulateur qu'on incrémente à chaque occurrence de `v` trouvée.

```
def occurrences(v, t):
    occ = 0
    for i in range(len(t)):
        if t[i] == v:
            occ += 1
    return occ
```

Exercice 71, page 84 Le tableau est créé avec une valeur quelconque dans ses cases, ici 0.

```
tab = [0] * 100   # un tableau de taille 100
for i in range(0, 100):
    tab[i] = randint(1, 1000)
print(tab)
```

Exercice 72, page 84 On se sert d'une variable `maximum` et d'une boucle **for**.

```
maximum = 0
for i in range(0, 100):
    if tab[i] > maximum:
        maximum = tab[i]
print("le maximum est", maximum)
```

©Editions Ellipses ISBN 978-2340033641

On peut initialiser la variable `maximum` à 0, car les éléments du tableau valent au moins 1 et la variable `maximum` se fera donc donner la valeur du premier élément du tableau au premier tour de boucle.

Exercice 73, page 84 On utilise un tableau de taille 11 dans lequel on stocke le nombre de fois que chaque nombre a été tiré. La case 0 de ce tableau n'est pas utilisée. Initialement, tous les compteurs sont à zéro.

```
hist = [0] * 11
```

Puis on répète mille fois le tirage d'un entier `n` entre 1 et 10 (inclus). À chaque fois, on augmente son compteur d'une unité.

```
for k in range(0, 1000):
    n = randint(1, 10) # entre 1 et 10 inclus
    hist[n] = hist[n] + 1
```

Enfin, on affiche les résultats.

```
for v in range(1, 11):
    print(v, "apparaît", hist[v], "fois")
```

Exercice 74, page 85

```
n = 30
fib = [0] * n
fib[1] = 1
for i in range(2, n):
    fib[i] = fib[i-2] + fib[i-1]
print(fib)
```

Exercice 75, page 85 On crée un nouveau tableau de même longueur dans lequel on place un à un les éléments de `t`.

```
def copie(t):
    r = [0] * len(t)
    for i in range(len(t)):
        r[i] = t[i]
    return t
```

On peut vérifier ensuite qu'un tableau et sa copie son bien indépendants : aucune modification de l'un n'est répercuté sur l'autre.

```
>>> t = [0, 1, 2, 3]
>>> t2 = copie(t)
>>> t[0] = 4
>>> t2[1] = 5
>>> t
```

© Éditions Eyrolles
ISBN 978-2-340-03641

```
[4, 1, 2, 3]
>>> t2
[0, 5, 2, 3]
```

Cette expérience n'aurait pas fonctionné avec la définition `t2 = t`.

Exercice 76, page 85

```
def ajoute(v, t):
    r = [v] * (len(t) + 1)
    for i in range(len(t)):
        r[i] = t[i]
    return r
```

Exercice 77, page 85

```
def concatenation(t1, t2):
    r = [0] * (len(t1) + lent(t2))
    for i in range(len(t1)):
        r[i] = t1[i]
    for i in range(len(t2)):
        r[len(t1) + i] = t2[i]
    return r
```

Exercice 78, page 85 C'est très semblable à l'exercice 71 page 84. Le tableau est créé avec la valeur 0 dans ses cases puis rempli avec une boucle `for` avant d'être renvoyé.

```
def tableau_aleatoire(n, a, b):
    tab = [0] * n
    for i in range(0, n):
        tab[i] = randint(a, b)
    return tab
```

Exercice 79, page 85 On suit l'indication, en tirant à chaque fois un nombre aléatoire (entre 0 et 10) pour l'ajouter à l'élément précédent.

```
def tableau_croissant(n):
    tab = [0] * n
    for i in range(1, n):
        tab[i] = tab[i-1] + randint(0, 10)
    return tab
```

Noter que la boucle commence à l'indice 1. Ici, on a arbitrairement commencé avec la valeur 0. On pourrait rendre cette fonction un peu plus générale en lui passant en arguments la valeur de la première case du tableau et l'amplitude de l'intervalle passé à `randint`.

©Editions Ellipses
ISBN 978-2340033641

Exercice 80, page 85 On utilise pour cela une variable temporaire, appelée ici `tmp`.

```
def echange(tab, i, j):
    tmp = tab[i]
    tab[i] = tab[j]
    tab[j] = tmp
```

Exercice 81, page 85

```
def somme(tab):
    s = 0
    for i in range(len(tab)):
        s += tab[i]
    return s

def moyenne(tab):
    s = 0
    for i in range(len(tab)):
        s += tab[i]
    return s / len(tab)
```

Exercice 82, page 85

```
def produit(tab):
    p = 1
    for i in range(len(tab)):
        p *= tab[i]
        if p == 0: return 0
    return p
```

Exercice 83, page 85 En se servant de la fonction `echange` de l'exercice précédent, comme suggéré, le code est assez court, mais pas complètement évident pour autant.

```
def miroir(tab):
    n = len(tab)
    for i in range(0, n // 2):
        echange(tab, i, n - 1 - i)
```

D'une part, il faut bien identifier `n-1-i` comme étant l'élément miroir de l'élément `i`. D'autre part, il faut se persuader que la boucle `for` parcourt le bon intervalle. Si le tableau a une taille paire, c'est-à-dire `n` $= 2k$, alors la boucle `for` va de 0 inclus à k exclu. Elle parcourt donc bien la première moitié du tableau. Si en revanche le tableau a une taille impaire, c'est-à-dire

©Éditions Ellipses ISBN 978-2340033641

`n` $= 2k + 1$, la boucle `for` va toujours de 0 inclus à k exclu. Elle parcourt donc la première moitié du tableau, à l'exclusion de l'élément central. Mais comme il n'y a pas lieu de modifier l'élément central, c'est correct.

Exercice 84, page 86 On suit l'algorithme donné :

```
def melange(tab):
    for i in range(1, len(tab)):
        echange(tab, i, randint(0, i))
```

On note qu'on démarre à l'indice 1, car il est inutile d'échanger le premier élément avec lui-même.

Exercice 85, page 86 On parcourt tous les indices de `tab1`, en renvoyant `False` si on arrive à un indice auquel les deux tableaux diffèrent ou dépassant la capacité de `tab2`. Ce deuxième critère est testé en premier pour éviter un accès hors limites. Le code se repose sur la sémantique paresseuse de **or**.

```
def prefixe(tab1, tab2):
    for i in range(len(tab1)):
        if i >= len(tab2) or tab1[i] != tab2[i]:
            return False
    return True
```

Exercice 86, page 86 Même stratégie qu'à l'exercice précédent, en partant de la fin de chacun des tableaux.

```
def suffixe(tab1, tab2):
    n1 = len(tab1) - 1
    n2 = len(tab2) - 1
    for i in range(len(tab1)):
        if i > n2 or tab1[n1 - i] != tab2[n2 - i]:
            return False
    return True
```

Exercice 87, page 86

```
def hamming(tab1, tab2):
    d = 0
    for i in range(len(tab1)):
        if tab1[i] != tab2[i]:
            d += 1
    return d
```

©Editions Ellipses
ISBN 978-2340033641

Exercice 88, page 86 On définit l'ampleur de la boucle avec la plus petite des deux longueurs, et on ajoute la différence des deux longueurs au résultat obtenu.

```
def hamming(tab1, tab2):
    if len(tab1) <= len(tab2):
        min = len(tab1)
        d = len(tab2) - len(tab1)
    else:
        min = len(tab2)
        d = len(tab1) - len(tab2)
    for i in range(min):
        if tab1[i] != tab2[i]:
            d += 1
    return d
```

Exercice 89, page 86 On fait un cas particulier pour le mois de février, puis on utilise un tableau comme suggéré.

```
def nbjoursmois(a, m):
    if m == 2 and bissextile(a): return 29
    t = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    return t[m - 1]
```

On prend soin de décaler le numéro du mois, le tableau étant indexé à partir de 0 mais les mois à partir de 1.

Exercice 90, page 96 L'idée est simple : tant que le nombre est supérieur ou égal à 10, on le divise par 10, tout en comptant le nombre d'itérations.

```
def nombre_de_chiffres(n):
    c = 1
    while n >= 10:
        n = n // 10
        c = c + 1
    return c
```

Une fois n'est pas coutume, on se permet de modifier l'argument de la fonction. C'est plus simple. Une autre solution consisterait à calculer les puissances de dix successives, dans une variable, jusqu'à dépasser le nombre `n` donné.

Exercice 91, page 97

```
n = int(input("nombre de départ : "))
while n != 1:
```

© Éditions Ellipses
ISBN 978-2-340-03364-1

Index

∞, 254
(), 194
*
  import général, 17
  multiplication, 4
  sélecteur CSS, 374
  tableaux, 79
+
  addition, 4
  tableaux, 79, 232
+=, 9
,
  séparateur de champs CSV, 205
  séparateur de paramètres, 12, 61
  séparateur de valeurs, 5, 66, 74
-
  différence d'ensembles, 191
  soustraction, 4
/
  division, 4
  séparateur de chemin, 315
//, 6
:
  association clé-valeur, 196
  début de bloc, 27, 40, 58, 88
<, 40
<=, 40
<<, 284
=, 7
==, 39
>
  redirection (*shell*), 320
  supérieur, 40
>=, 40
>>
  décalage, 284
  redirection (*shell*), 321
[], 75, 196
%
  (unité CSS), 373
  modulo, 6
&
  et logique, 283
  intersection d'ensembles, 191
^
  ou exclusif d'ensembles, 191
  ou exclusif logique, 284
_, 27
|
  ou logique, 284
  redirection (*shell*), 321
  union d'ensembles, 191
!=, 40
"
  délimiteur de champ CSV, 205
  délimiteur de chaînes, 10
\N, 270
\u, 270
{}, 196
~
  complément à 1, 284
  répertoire personnel (*shell*), 320
127.0.0.1, 349
2>, 320
2>>, 321
404 (erreur), 384

©Editions Eyrolles
ISBN 978-2-340-03364-1

`abs`, 434
accumulateur, 30, 31
`add`, 190
additionneur (circuit), 281
affectation, 7
agrégation, 215
algèbre de Boole, 279
algorithme
    de tri, 143
    des plus proches voisins, 175
    glouton, 161
aliasing, 81
allumettes (jeu des), 101
ALU, 288
`and`, 48
Android, 310
année bissextile, 55
anniversaire, 189
appel d'une fonction, 58
`append`, 77, 217
Apple, 310
apprentissage, 175
architecture
    MIMD, 291
    parallèle, 291
    SIMD, 291
    SISD, 291
`argv` (`sys.argv`), 332
arithmétique, 4
ASCII, 261
assembleur, 297
`assert`, 124, 126
attribut
    d'une table de données, 204
    HTML, 363

`background` (propriété CSS), 373
`backward` (`turtle.backward`), 16
balise
    fermante, 363
    HTML, 362
    ouvrante, 363
    vide, 363
base 2, 242
Berners-Lee, Tim, 363
big endian, 246
`bin`, 245
bin packing (problème), 171
binary32, 252
binary64, 252
bit, 241
    de parité, 264
bloc de code, 28
boîte CSS, 372
BOM, 270
Boole, Georges, 279
booléen, 39
`border` (propriété CSS), 372
boucle, 103
    bornée, 25
    imbriquée, 104
    infinie, 91
    tant que, 87
boutisme, 246, 270
branchement, 39
`break`, 92
BSD, 309
bus, 290

C (langage), 309
`cd` (commande), 315
chaîne
    d'octets, 270
    de caractères, 11, 80
chargement de données, 204
chemin
    (système de fichier), 315
    absolu, 315
    relatif, 315
chiffre booléen, 279
chiffrer, 385
`chr`, 264
`circle` (`turtle.circle`), 16
circuit
    additionneur, 281
    demi-additionneur, 281
    décodeur, 281
    électronique, 275

© Éditions Eyrolles
ISBN 978-2340033641

circuit combinatoire, 280
classification (problème), 176
clé, 196
client, 345
client-serveur, 345
`close`, 199
code de sortie, 126
colonne (d'une table de données), 204
`color` (propriété CSS), 373
commentaire, 13, 123
comparaison, 40
complément à 2, 247
complétude d'un algorithme, 156
complexité, 107
composant graphique, 334
compréhension, 115, 191, 198
    double, 234
compte utilisateur, 313
compteur de boucle, 28
conjonction, 48
console JavaScript, 409
`continue`, 107
cookie, 397
corps
    de boucle, 27, 88
    de fonction, 58
correction d'un algorithme, 156
CPU, 288
crible, 112
CSS (Cascading Style Sheets), 369
CSV, 204, 205
`csv` (bibliothèque), 206, 208
curseur d'un fichier, 331

datagramme, 353
De Morgan, Augustus, 280
déclaration de type HTML, 363
`decode`, 271
`def`, 58
`del`, 197
dessert, 279
devine le nombre (jeu), 100, 159
dichotomique (recherche), 153
dictionnaire, 196
    ordonné, 206
`DictReader` (`csv.DictReader`), 206
`DictReader` (bibliothèque), 209
`DictWriter` (`csv.DictWriter`), 208
dimension, 116
directory, 314
disjonction, 48
`display` (propriété CSS), 372
distance
    d'édition, 181
    de Hamming, 180, 184
    de Manhattan, 180
    euclidienne, 180
distribution logicielle, 311
divergence, 91
diviser pour régner, 153
division, 4
    euclidienne, 6, 135
DMA (Direct Memory Access), 290
DNS, 355
`do`, 93
DOCTYPE, 363
documentation d'une fonction, 122
domaines de premier niveau, 355
données, 175
dossier, 313
double boucle, 234
doublons, 103, 130, 192, 327
drapeau, 299

échange, 85, 145, 418
échappement, 264
effet de bord, 8
Eich, Brendan, 407
élément HTML, 363
`elif`, 39
`else`, 39
émulateur de terminal, 312
`encode`, 270

©Editions Eyrolles
ISBN 978-2-409-03641

endianness, 246
ENIAC, 309
ensemble, 190
entête, 384
entier
    naturel, 242
    relatif, 247
entrée standard, 320
Ératosthène, 112
erreur 404, 384
état d'un programme, 8
Euclide, 135
Euler, Leonhard, 63
`exit`, 124
expansion (ligne de commande), 319
exposant, 251

factorisation, 64
`False`, 45
faux, 45
Fibonacci, 37, 85, 98
fichier, 204, 307
filtrage, 216
fizz-buzz, 55
flag, 299
`float`, 15
flottant (nombre), 251
fonction, 57
`for`, 25, 113
formulaire, 382
`forward` (`turtle.forward`), 16
`from`, 17
fromage, 279

garde, 88
GC (Glaneur de Cellules / Garbage Collector), 296
gestionnaire d'évènements, 336, 412
`get` (méthode HTTP), 389
GID, 313
`global`, 338
glouton (algorithme), 161
`goto` (`turtle.goto`), 16
graphe de flot de contrôle, 44
gros boutisme, 246
groupe d'utilisateur, 313
Gutenberg (Projet), 199

hachage, 200
`help`, 122
Héron (méthode de), 87
`hex`, 245
hexadécimal (système), 244
horloge, 296
Horner (méthode de), 484
HTML (HyperText Markup Language), 362
HTTP (HyperText Transfert Protocol), 381
`https`, 382
HTTPS (HyperText Transfer Protocol Secure), 385

IBM, 309
`id` (commande), 313
identifiant de connexion, 313
IEEE 754 (norme), 252
`if`, 39
`import`, 16
`in`, 190, 196
indentation, 28
index, 204
indice
    de boucle, 28
    de tableau, 75
`inf`, 256
`input`, 12
instruction conditionnelle, 40
Instruction Pointer, 289
`int`, 12
Intel, 309
interface, 344
    système, 311
    utilisateur, 311, 329
interprète Python, 3
interruption, 290

© Éditions Eyrolles
ISBN 978-2-212-13364-1

invariant de boucle, 135
invite de commandes
    du *shell*, 311
    Python, 3
IP (Internet Protocol), 346
ISO, 265, 345
isolation, 64
`items`, 198

jointure, 233

`key`, 226
`KeyError`, 197
`keys`, 197
knapsack (problème du sac à dos), 171
Knuth, Donald, 86

LAN, 347
langage machine, 296
latin-1, 265
`left` (`turtle.left`), 16
`len`, 75, 190
Lerdorf, Rasmus, 400
lexicographique (ordre), 42, 227
lien, 344, 382
    hypertexte, 361
ligne (d'une table de données), 204
Linux, 310
`list`, 115, 197
little endian, 246
`localhost`, 349
logarithme, 150, 157
logiciel libre, 311
login, 313
`ls` (commande), 315

MAC (adresse), 347
macOS, 310
`malloc`, 295
mantisse, 251
`margin` (propriété CSS), 372
masque de sous-réseau, 351
`math` (bibliothèque), 17, 257, 486
matrice, 116
mélange (d'un tableau), 86
mémoire
    cache, 291
    non volatile, 288
    vive, 288
    volatile, 288
microprocesseur, 288
`mkdir` (commande), 314
mölkky, 39
mot, 291
    machine, 241
motif *glob*, 319
MS-DOS, 309

n-uplet, 66, 193
    nommé, 194
`nan`, 256
navigateur Web, 382
négation, 48
nœud, 344
nombre
    entier, 5
    flottant, 251
    premier, 112
`None`, 64, 124, 154
`not`, 48
noyau, 311
NP-complétude, 164
numéro
    de port, 352
    de séquence, 353

octet, 241
`open`, 199, 330
opérande, 289, 297
opérateur
    bit-à-bit, 283
    bitwise, 283
    booléens, 279
    conjonction, 278
    disjonction, 278
    décalage, 284
    logique, 284
    négation, 278

©Editions Eyrolles
ISBN 9782340033641

ou exclusif, 279
optimisation (problème), 163
option (de ligne de commande), 316
`or`, 48
oracle, 131
`ord`, 263
`OrderedDict`, 207
ordinogramme, 45
ordre lexicographique, 42, 227
OSI (Open Systems Interconnection), 345

`padding` (propriété CSS), 372
page Web, 362
paradoxe des anniversaires, 189
paramètre
d'une fonction, 59
d'une requête, 385
paresse, 48
passage par valeur, 68
passerelle, 350
petit boutisme, 246
`pi` (`math.pi`), 486
pile, 292
pilote de périphérique, 309
`ping`, 349
point de code, 266
pointeur d'instruction, 289
`pop`, 77
port d'entrée-sortie, 290
porte
complète, 277
ET, 277
logique, 276
NAND, 276
NOR, 277
NOT, 276
OU, 277
XOR, 279
POSIX, 310
`post` (méthode HTTP), 390
postcondition, 123
précondition, 123
`print`, 10
problème de classification, 176
procédure, 64
processeur, 288
processus, 292
programmation défensive, 124
programmation évènementielle, 336
programme, 10
projection, 219
projet Euler, 63
propriétaire (d'un fichier), 316
propriété CSS, 370
Protocol Data Unit, 345
protocole, 344, 382
Puissance 4 (jeu), 119
`pwd` (commande), 314
`pt` (unité CSS), 373
`px` (unité CSS), 373
Pythagore, 71, 112

racine d'un système de fichiers, 314
RAM, 288
`random` (bibliothèque), 16
`range`, 26
`read`, 199
`reader` (`csv.reader`), 206
recherche
dans une table, 213
dichotomique, 153
redirection (terminal), 320
réel (nombre), 251
registre, 289
régression (problème), 183
`remove`, 190
rendu de monnaie (problème), 167
répertoire, 313
`replace`, 211
réponse, 383
requête, 213, 383
réseau, 344
informatique, 344
`return`, 62, 70

© Éditions Eyrolles
ISBN 978-2-212-13361-3

`reverse`, 227
`right` (`turtle.right`), 16
ROM, 288
routeur, 350
référence, 9

sac à dos (problème), 171
segment de mémoire, 292
sélecteur CSS, 370, 374
sélection, 216
serveur, 344
    Web, 382
service réseau, 344
session HTTP, 398
`set`, 191
shell, 311
site Web, 382
`socket` (bibliothèque), 355
`sort`, 150, 228
`sorted`, 150, 226
sortie
    d'erreur, 320
    standard, 320
sous-domaine, 355
sous-répertoire, 314
sous-réseau, 350, 351
spécification, 121
`split`, 199
SQL, 220
stabilité d'un tri, 227
`stderr` (`sys.stderr`), 333
`stdin` (`sys.stdin`), 333
`stdout` (`sys.stdout`), 333
Syracuse, 443
`sys` (bibliothèque), 332, 333
système
    d'exploitation, 307
    de fichier, 309
    de résolution de noms, 355

table, 203
    ASCII, 262
    de hachage, 200
    de routage, 350
    logique, 276
tableau, 73
    associatif, 196
    à plusieurs dimensions, 115
taille
    d'un ensemble, 190
    d'un tableau, 75
    d'une chaîne de caractères, 80
Tanenbaum, Andrew S., 310
tas, 292
TCP (Transmission Control Protocol), 346, 353
terminaison, 96
terminal, 312
test, 125
`time` (bibliothèque), 110
`tkinter` (bibliothèque), 333
Torvalds, Linus, 310
tour de boucle, 27
transistor, 275
tri
    d'un tableau, 143
    d'une table, 225
    de trois entiers, 55
    en place, 228
    fusion, 150
    par dénombrement, 150, 152
    par insertion, 147
    par sélection, 144
    par tas, 150
    stable, 227
`True`, 45
TTL (Time To Live), 351
`turtle` (bibliothèque), 16

UDP (User Datagram Protocol), 352
UID, 313
Unicode, 11, 265
Unité Centrale de Traitement, 288
Unix, 309
URL, 381
UTF-16, 268
UTF-8, 267

©Editions Eyrolles
ISBN 978-2-340-03641

validation croisée, 179
`values`, 198
variable, 6
    globale, 68
    locale, 67
variant, 96, 157
Verne, Jules, 73, 198
vie privée, 182, 399
voisins (algorithme), 175
von Neumann, John, 287
voyageur de commerce, 161
vrai, 45

Web, 361
`while`, 87
widget, 334
Windows, 310
`writeheaders`
        (`DictReader.writeheaders`),
        209
`writerows`
        (`DictReader.writerows`),
        209

©Éditions Ellipses
ISBN 978-2340033641