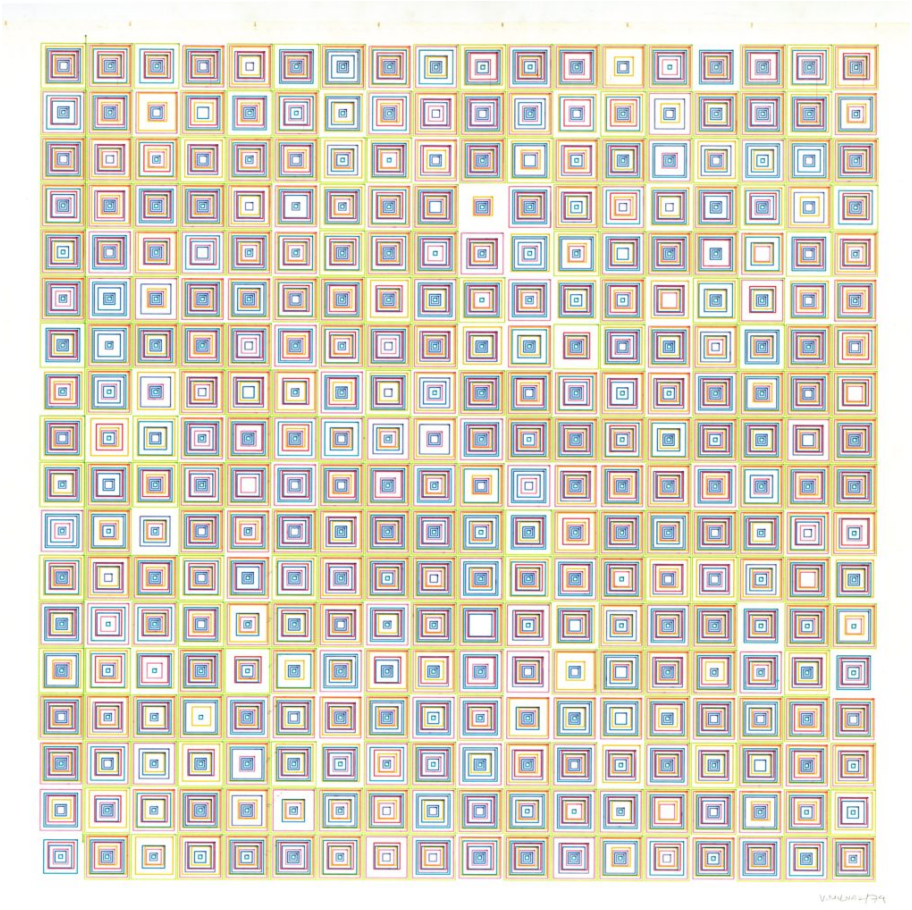
Cours de BCPST 1 et 2

Informatique

BCPST 2A (*Marcelin Berthelot*)



Vera Molnár, « (Des)Ordres » (1974)

Année 2025–2026

Mise en forme et relecture assistées par Claude Code (Anthropic). Contenu pédagogique conçu et vérifié par l'auteur.

Table des matières

Introduction **5**

1 Types de base, variables et opérations **9**
1.1 Types de base . . . 9
1.2 Variables et affectation . . . 10
1.3 Opérations sur les entiers et les flottants . . . 11
1.4 Opérations booléennes et comparaisons . . . 12
1.5 Opérations sur les chaînes de caractères . . . 13
1.6 Entrées et sorties . . . 13
1.7 Fonctions natives . . . 14

2 Instructions conditionnelles **17**
2.1 L'instruction `if` . . . 17
2.2 Conditions composées . . . 18
2.3 Imbrication de conditions . . . 18
2.4 Exemples algorithmiques . . . 19

3 Boucles **21**
3.1 La boucle `while` . . . 21
3.2 La boucle `for` . . . 22
3.3 Comparaison `while` / `for` . . . 23
3.4 Algorithmes classiques . . . 23

4 Fonctions **25**
4.1 Définir et appeler une fonction . . . 25
4.2 Valeur de retour . . . 26
4.3 Portée des variables . . . 26
4.4 Paramètres par défaut . . . 27
4.5 Spécification d'une fonction . . . 27
4.6 Exemples . . . 27

5 Listes **29**
5.1 Création d'une liste . . . 29
5.2 Accès aux éléments . . . 30
5.3 Modification d'une liste . . . 30
5.4 Opérations sur les listes . . . 31
5.5 Parcours d'une liste . . . 31
5.6 Compréhension de liste . . . 31
5.7 Copie d'une liste . . . 32
5.8 P-uplets (tuples) . . . 33
5.9 Ensembles (`set`) . . . 34

6 Algorithmes sur les listes **35**
6.1 Calculs sur une liste . . . 35
6.2 Recherche dans une liste . . . 36
6.3 Tests sur les listes . . . 37
6.4 Construction de listes . . . 38
6.5 Listes de listes (tableaux 2D) . . . 38
6.6 Application : nombres parfaits . . . 40

7 Fichiers texte **41**
7.1 Ouvrir et fermer un fichier . . . 41

7.2 Lire un fichier . . . 41
7.3 Écrire dans un fichier . . . 42
7.4 Fichiers CSV . . . 43
7.5 Exemple complet : analyse d'une série de mesures . . . 43

8 Modules 45
8.1 Importer un module . . . 45
8.2 Le module `math` . . . 45
8.3 Le module `random` . . . 46
8.4 Le module `numpy` . . . 47
8.5 Le module `matplotlib.pyplot` . . . 48
8.6 Exemple synthétique : intégration graphique . . . 50

9 Fonctions récursives 53
9.1 Principe de la récursion . . . 53
9.2 Exemples classiques . . . 54
9.3 Récursion et itération . . . 55
9.4 Condition de terminaison . . . 55
9.5 Application : processus de branchement . . . 56

10 Recherche dichotomique 59
10.1 Recherche dans une liste triée . . . 59
10.2 Résolution approchée d'une équation . . . 60

11 Algorithmes de tri 63
11.1 Tri par sélection . . . 63
11.2 Tri par insertion . . . 64
11.3 Tri par comptage . . . 65
11.4 Comparaison des algorithmes . . . 65

12 Tris avancés 67
12.1 Limites des tris élémentaires . . . 67
12.2 Tri fusion (*mergesort*) . . . 67
12.3 Comparaison des algorithmes de tri . . . 69

13 Mutabilité et effets de bord 71
13.1 Types mutables et immuables . . . 71
13.2 Aliasing . . . 72
13.3 Effets de bord dans les fonctions . . . 73
13.4 Conséquences pratiques . . . 73

14 Dictionnaires 75
14.1 Créer et accéder à un dictionnaire . . . 75
14.2 Modifier un dictionnaire . . . 76
14.3 Tester l'appartenance . . . 76
14.4 Parcourir un dictionnaire . . . 76
14.5 Applications . . . 76
14.6 Résumé des opérations . . . 78

15 Bases de données et SQL 79
15.1 Comment tester les exemples . . . 79
15.2 Modèle relationnel . . . 80
15.3 Requêtes de base : SELECT . . . 80
15.4 Fonctions d'agrégation . . . 81
15.5 Jointures . . . 82
15.6 Utilisation avec Python . . . 82

16 SQL avancé 85
16.1 Alias . . . 85
16.2 Sous-requêtes . . . 85
16.3 LEFT JOIN . . . 86
16.4 Modifier les données . . . 86
16.5 Définir le schéma . . . 87

17 Graphes — bases **89**
17.1 Définitions . . . 89
17.2 Représentations en Python . . . 90
17.3 Parcours d'un graphe . . . 91
17.4 Application : distance entre sommets . . . 92
17.5 Énumération : exploration d'un arbre de recherche . . . 92

18 Graphes avancés **97**
18.1 Graphes pondérés . . . 97
18.2 Algorithme de Dijkstra . . . 97
18.3 Détection de cycles . . . 99

19 Méthodes numériques **101**
19.1 Intégration numérique . . . 101
19.2 Intégration à partir de données discrètes . . . 104
19.3 Méthode de Newton . . . 104
19.4 Méthode d'Euler . . . 106

20 Statistiques et simulations **111**
20.1 Simuler des lois de probabilité . . . 111
20.2 Modèles d'urnes . . . 112
20.3 Estimer par simulation . . . 114
20.4 Statistiques descriptives . . . 116
20.5 Régression linéaire . . . 117

21 Intervalles de fluctuation et tests statistiques **121**
21.1 Intervalle de fluctuation d'une fréquence . . . 121
21.2 Intervalle de confiance pour une proportion . . . 122
21.3 Test de conformité d'une proportion . . . 123
21.4 Applications . . . 124

22 Annexe : `numpy` pour l'algèbre linéaire **127**
22.1 Rappel : tableaux à deux dimensions . . . 127
22.2 Produit terme à terme et produit matriciel . . . 128
22.3 Extraire des lignes et des colonnes . . . 129
22.4 Déterminant, inverse, valeurs propres . . . 129
22.5 Le sous-module `numpy.random` : incertitudes-type en physique . . . 130

Index **133**

Introduction

Objectif de ce cours

Ce cours est une introduction à l'informatique et à la programmation en Python, destinée aux étudiants de classes préparatoires BCPST. Il suit le programme officiel de 2021 et couvre l'ensemble des deux années de préparatoire. L'informatique n'est pas une discipline annexe : elle est aujourd'hui au cœur des sciences du vivant, de la biologie, de la physique et des mathématiques appliquées. Savoir programmer permet de simuler des phénomènes, d'analyser des données, de résoudre des problèmes numériques inaccessibles à la main.

Programme et concours

Ce cours couvre l'intégralité du programme officiel d'informatique BCPST : bases du langage Python, algorithmes, méthodes numériques et statistiques, traitement de fichiers et d'images, ainsi que des approfondissements sur les structures de données avancées, les bases de données et les graphes.

> **Remarque sur les concours :**
>
> Les épreuves informatiques des concours BCPST portent sur l'algorithmique et la simulation. La **syntaxe des fonctions de modules** (`numpy`, `matplotlib`, etc.) **sera rappelée dans l'énoncé** si elle est nécessaire : vous n'avez pas à la mémoriser.
>
> En revanche, certaines notions de ce cours — mutabilité et effets de bord, dictionnaires, bases de données, graphes et les approfondissements — **ne sont pas au programme des concours** et pourront ne pas être traitées dans l'année selon le temps disponible.

Ces notions apriori non évaluées au concours sont signalées dans le cours par l'encadré suivant :

> ★ **Complément —**
>
> Ce contenu ne fait pas partie du programme des concours. Il peut ne pas être traité dans l'année selon le temps disponible.

Remarque sur les exemples arithmétiques : plusieurs exemples de ce cours font appel à des notions d'arithmétique élémentaire — divisibilité, PGCD, nombres premiers — qui ne constituent pas un chapitre à part entière du programme BCPST. Ces notions sont cependant **fondamentales et apparaissent couramment dans les sujets de concours** comme support algorithmique. Elles sont supposées connues des mathématiques du lycée ou du collège.

Comment utiliser ce cours

Ce cours ne se lit pas comme un roman. Pour en tirer le meilleur parti :

• **Tapez les exemples vous-même.** Ne vous contentez pas de lire le code : ouvrez Python et exécutez-le. Modifiez-le, cassez-le, observez ce qui se passe.

• **Faites les exercices.** La programmation s'apprend en pratiquant, pas en lisant.

• **Expérimentez.** Chaque fois qu'une notion vous semble abstraite, testez-la dans l'interpréteur Python. Une ligne de code vaut souvent mieux qu'un long paragraphe.

• **Lisez les messages d'erreur.** Python indique précisément ce qui ne va pas. Apprendre à lire ces messages fait partie de l'apprentissage.

Remarque : Dans un premier temps vous pouvez laisser de coté les chapitres qui apparaissent en orange dans la table des matière.

Installer et utiliser Python

Python est un logiciel libre et gratuit. Plusieurs environnements permettent de l'utiliser ; en voici trois recommandés selon vos besoins.

Basthon — sans installation

Basthon (`basthon.fr`) est un environnement Python qui fonctionne entièrement dans un navigateur web, sans rien installer. C'est la solution idéale pour commencer immédiatement ou travailler sur un ordinateur que vous ne maîtrisez pas. En bas des pages de cahier-de-prépa vous avez une icône liée à l'environnement `basthon`.

Spyder (via Anaconda) — recommandé pour le travail scientifique

Anaconda est une distribution Python qui installe en une seule fois Python et toutes les bibliothèques scientifiques utiles : `numpy`, `matplotlib`, `scipy`, etc. L'éditeur **Spyder** inclus est particulièrement adapté au travail scientifique : il affiche les variables en mémoire, permet d'exécuter le code bloc par bloc, et intègre une console interactive.
Téléchargement : `anaconda.com/download`

Pyzo — léger et simple

Pyzo est un environnement Python léger et simple d'utilisation, avec une console interactive et un éditeur côte à côte. Il est une bonne alternative à Spyder si vous préférez un outil moins volumineux.
Téléchargement : `pyzo.org`

Conseil : quel que soit l'environnement choisi, prenez l'habitude d'utiliser la **console interactive** pour tester rapidement une expression ou vérifier le comportement d'une fonction. C'est votre meilleur outil d'exploration.

Un peu d'histoire sur Python

Python est créé par le Néerlandais Guido van Rossum, qui commence son développement fin 1989 aux Pays-Bas et publie la première version en 1991. Son nom ne vient pas du serpent mais de la troupe comique britannique *Monty Python*, dont van Rossum était fan — un clin d'œil qui se retrouve dans la documentation officielle, truffée d'exemples utilisant *spam* et *eggs* (sketch des Monty Python), et dans des choix de noms parfois espiègles (le module `this` qui affiche le « Zen de Python » en est un exemple).
Dès l'origine, l'objectif est un langage à la syntaxe claire et lisible, accessible aux débutants tout en restant puissant, privilégiant une philosophie où « il devrait y avoir une — et de préférence une seule — manière évidente de faire les choses ».
Cette accessibilité, combinée à une licence libre, a permis l'émergence d'une immense communauté qui a développé des bibliothèques spécialisées (NumPy et SciPy pour le calcul scientifique, Pandas pour les données, Django pour le web, TensorFlow pour l'apprentissage automatique), faisant de Python un langage généraliste utilisé aussi bien en enseignement qu'en recherche ou en production. Il se distingue ainsi de langages plus spécialisés comme R (statistiques), MATLAB (calcul matriciel et ingénierie) ou C (performance et bas niveau), Python misant davantage sur la polyvalence et la simplicité d'écriture que sur l'optimisation d'un domaine précis.

Règles de style Python

Le code Python obéit à quelques conventions de lisibilité, formalisées dans la PEP 8, le guide de style officiel du langage. Ces règles ne sont pas vérifiées par l'interpréteur (un code qui ne les respecte pas s'exécute quand même) mais elles sont attendues dans tout code « propre » et cohérentes entre elles.

Ces règles ne feront pas l'objet d'une évaluation notée : elles sont données à titre indicatif, pour habituer progressivement à des habitudes de rédaction propres et lisibles.

Nommage. Les variables et fonctions s'écrivent en minuscules, mots séparés par des underscores : `nombre_lignes`, `calculer_moyenne`. Les constantes s'écrivent en majuscules avec underscores : `TAILLE_MAX`. On évite les noms ambigus comme `l`, `O` ou `I`, trop proches visuellement de `1` et `0`.

Indentation. Chaque niveau de bloc s'indente de 4 espaces, jamais de tabulation.

Longueur de ligne. Une ligne ne dépasse pas 79 caractères (tolérance courante jusqu'à une centaine en pratique).

Espacement. On met un espace autour des opérateurs (`x = a + b`, pas `x=a+b`), un espace après une virgule (`f(a, b)`, pas `f(a,b)`), mais pas d'espace juste après une parenthèse ou un crochet ouvrant, ni juste avant celui qui ferme (`f(x)`, pas `f( x )`). De même, on ne met pas d'espace entre le nom d'une fonction et la parenthèse ouvrante de l'appel : `f(a, b)`, pas `f (a, b)`.

Une instruction par ligne. On écrit une seule instruction par ligne, et l'on évite le point-virgule (`;`) pour en regrouper plusieurs — pratique héritée d'autres langages, mais qui nuit à la lisibilité en Python.

Parenthèses autour de `return`. `return` est un mot-clé, pas une fonction : les parenthèses autour de la valeur renvoyée sont inutiles et à éviter, sauf si elles servent réellement à regrouper une expression (comme pour n'importe quelle expression Python) : `return x`, pas `return (x)`.

Organisation du fichier. Les imports se placent en haut du fichier, un par ligne. On sépare deux fonctions par deux lignes vides.

Commentaires. PEP 8 distingue plusieurs types de commentaires.

— *Commentaires en bloc* : placés sur leur propre ligne, au même niveau d'indentation que le code qu'ils décrivent, en commençant par `#` suivi d'un espace.

```
# Calcule la moyenne ponderee des notes
# en tenant compte des coefficients de chaque epreuve
moyenne = calculer_moyenne_ponderee(notes, coefficients)
```

— *Commentaires en ligne* : placés sur la même ligne que l'instruction, séparés du code par au moins deux espaces, puis `#` . À utiliser avec parcimonie.

```
x = x + 1  # incremente le compteur
```

— *Contenu* : un commentaire doit expliquer le *pourquoi*, pas le *quoi* — le code montre déjà ce qu'il fait. Un commentaire qui contredit le code est pire que l'absence de commentaire : il faut le maintenir à jour.

— *Docstrings* : les chaînes de documentation, placées en première ligne d'une fonction, d'un module ou d'une classe, s'écrivent entre triples guillemets doubles (`"""..."""`).

1

Types de base, variables et opérations

Plan du chapitre

1.1 Types de base **9**
1.2 Variables et affectation **10**
1.3 Opérations sur les entiers et les flottants **11**
1.3.1 Opérations arithmétiques 11
1.3.2 Fonctions numériques natives 12
1.4 Opérations booléennes et comparaisons **12**
1.4.1 Comparaisons 12
1.4.2 Opérations logiques 12
1.5 Opérations sur les chaînes de caractères **13**
1.6 Entrées et sorties **13**
1.6.1 `print` 13
1.6.2 `input` 13
1.6.3 Formatage des chaînes (f-strings) 14
1.7 Fonctions natives **14**

1.1 Types de base

En Python, toute valeur possède un **type**. Le type détermine les opérations autorisées sur la valeur. La fonction `type(x)` renvoie le type de `x`.

Définition : Les quatre types de base

- `int` : entier relatif (taille illimitée en Python).
  Exemples : `0`, `42`, `-7`, `1000000000000`
- `float` : nombre à virgule flottante (approximation des réels).
  Exemples : `3.14`, `-1.0`, `2.3e10`, `1e-5`
- `bool` : booléen, deux valeurs seulement.
  Exemples : `True`, `False`
- `str` : chaîne de caractères (séquence de caractères entre guillemets).
  Exemples : `"Python"`, `'Bonjour'`, `""`

Remarques :

- Python est à **typage dynamique** : le type d'une variable est déterminé automatiquement à l'affectation.
- Les guillemets simples `'...'` et doubles `"..."` sont équivalents pour les chaînes.
- Un `float` n'est pas un réel exact : `0.1 + 0.2` vaut `0.30000000000000004` en Python.

Pour stocker le nombre 0.4 : on approche au mieux ce nombre en binaire, c'est-à-dire en développant chaque chiffre comme une puissance de 2 :

$$^{2}0,0110011001100110011001100110011001100110011001100110011010$$

$$0\times 2^{-1}+1\times 2^{-2}+1\times 2^{-3}+0\times 2^{-4}+0\times 2^{-5}+1\times 2^{-6}+1\times 2^{-7}+\cdots$$

qui est égale à :

$$\frac{7205759403792794}{18014398509481984}$$

ce nombre vaut environ :

$$0,4000000000000000022204460492503$$

Piège : ne jamais tester l'égalité entre deux flottants avec ==.

En raison des erreurs d'arrondi inhérentes à la représentation binaire, deux calculs mathématiquement égaux peuvent donner des flottants légèrement différents :

```
print(0.1 + 0.2 == 0.3)        # False  (!!)
print(0.1 + 0.2)               # 0.30000000000000004
```

Règle : pour comparer deux flottants `a` et `b`, tester `abs(a - b) < eps` où `eps` est la précision souhaitée :

```
eps = 1e-9
print(abs(0.1 + 0.2 - 0.3) < eps)    # True
```

Propriété : Transtypage

On peut convertir une valeur d'un type à un autre avec les fonctions de conversion :

`int(x)`	convertit `x` en entier	`int(3.9)` → `3` (troncature)
`float(x)`	convertit `x` en flottant	`float(3)` → `3.0`
`bool(x)`	convertit `x` en booléen	`bool(0)` → `False`, `bool(5)` → `True`
`str(x)`	convertit `x` en chaîne	`str(42)` → `"42"`

Remarque : `bool(x)` vaut `False` si `x` est `0`, `0.0`, `""` ou `[]`, et `True` sinon.
En particulier, `True` se comporte comme `1` et `False` comme `0` dans les calculs arithmétiques.

```
print(int(3.9))        # affiche 3  (troncature vers zero)
print(float(5))        # affiche 5.0
print(bool(0))         # affiche False
print(True + True)     # affiche 2  (True vaut 1)
```

1.2 Variables et affectation

Définition : Variable, affectation

Une **variable** est un nom associé à une valeur stockée en mémoire.

L'**affectation** s'écrit `nom = expression` : Python évalue l'expression à droite du `=`, puis associe le résultat au nom à gauche.

```
x = 5           # x recoit la valeur 5
y = x + 1       # y recoit la valeur 6
x = x * 2       # x recoit la valeur 10  (l'ancienne valeur de x est perdue)
```

Propriété : Règles de nommage

Un nom de variable :
- est composé de lettres, chiffres et underscores `_` ;
- ne commence pas par un chiffre ;
- est sensible à la casse : `x` et `X` sont deux variables distinctes ;
- ne peut pas être un mot réservé de Python : `if`, `for`, `while`, `def`, `return`, `True`, `False`, `not`, `and`, `or`, `in`, `None`, etc.

Remarque : Python autorise techniquement d'utiliser le nom d'une fonction native comme nom de variable, mais c'est une très mauvaise pratique car cela **masque** la fonction originale pour la suite du programme. On évitera donc des noms comme `abs`, `min`, `max`, `sum`, `len`, `type`, `input`, `print`, `round`, `ord`, `chr`, etc.

```
max = 10        # max n'est plus la fonction max() !
print(max(3, 5))  # erreur : TypeError
```

Affectation multiple : Python autorise l'affectation simultanée de plusieurs variables :

```
a, b = 2, 3       # a vaut 2, b vaut 3
a, b = b, a       # echange de a et b : a vaut 3, b vaut 2
```

Remarque : L'approche classique pour inverser deux valeurs utilise une variable auxiliaire :

```
a = 2
b = 3
aux = a
a = b
b = aux
```

1.3 Opérations sur les entiers et les flottants

1.3.1 Opérations arithmétiques

Définition : Opérateurs arithmétiques

`a + b`	addition	
`a - b`	soustraction	
`a * b`	multiplication	
`a ** b`	puissance (a^b)	`2**10` → `1024`
`a / b`	division réelle	renvoie toujours un `float` : `5/2` → `2.5`
`a // b`	quotient de la division euclidienne	`5//2` → `2`
`a % b`	reste de la division euclidienne (modulo)	`5%2` → `1`

Remarques :

- Pour deux `int`, `+`, `-`, `*`, `//`, `%` et `**` renvoient un `int`. Dès qu'un opérande est `float`, le résultat est `float`.
- La relation de la division euclidienne s'écrit : `a == (a//b)*b + (a%b)`.
- `abs(x)` renvoie la valeur absolue de `x`.

Propriété : Priorités des opérations

Les opérations sont évaluées dans l'ordre de priorité suivant (du plus prioritaire au moins prioritaire) :

Priorité	Opérateur(s)	Ordre d'évaluation
1	`()`	parenthèses (de l'intérieur vers l'extérieur)
2	`**`	puissance (droite à gauche)
3	`+x`, `-x`	signe unaire
4	`*`, `/`, `//`, `%`	multiplication, divisions (gauche à droite)
5	`+`, `-`	addition, soustraction (gauche à droite)

Exemples :

```
print(2 + 3 * 4)       # affiche 14  (pas 20)
print(2 ** 3 ** 2)     # affiche 512 (** est associatif a droite : 2**(3**2) = 2**9)
print(10 // 3 * 3)     # affiche 9   (gauche a droite : (10//3)*3 = 3*3)
```

1.3.2 Fonctions numériques natives

`abs(x)`	valeur absolue de `x`
`round(x, n)`	arrondi de `x` à `n` décimales
`divmod(a, b)`	renvoie le couple `(a//b, a%b)`
`pow(a, b)`	équivalent à `a**b`
`max(a, b, ...)`	maximum des arguments
`min(a, b, ...)`	minimum des arguments

1.4 Opérations booléennes et comparaisons

1.4.1 Comparaisons

Définition : Opérateurs de comparaison

Les comparaisons renvoient un **booléen** (`True` ou `False`) :

`a == b`	égalité
`a != b`	différence
`a < b`	strictement inférieur
`a <= b`	inférieur ou égal
`a > b`	strictement supérieur
`a >= b`	supérieur ou égal

Remarque : Python autorise les comparaisons chaînées : `2 <= a < 8` est équivalent à `(2 <= a) and (a < 8)`.

1.4.2 Opérations logiques

Définition : Opérateurs logiques

`a and b`	ET logique	`True` si `a` et `b` sont tous les deux `True`
`a or b`	OU logique	`True` si au moins l'un des deux est `True`
`not a`	NON logique	`True` si `a` est `False`, et réciproquement

Propriété : Priorités des opérateurs logiques et de comparaison

En complétant le tableau des priorités arithmétiques :

Priorité	Opérateur(s)
6	==, !=, <, <=, >, >= (comparaisons)
7	not
8	and
9	or

Ainsi `a or b and not c` s'évalue comme `a or (b and (not c))`.

Exemples :

```
x, y, z = 3, 7, 10
print((x < y) or (z < y) and not (x == 3))  # True or (False and False) = True
print(not (x > 0 or y < 0))                 # not True = False
```

1.5 Opérations sur les chaînes de caractères

Définition : Opérations sur les chaînes

Soit `s` et `t` deux chaînes de caractères et `n` un entier :

`s + t`	concaténation	`"bon" + "jour"` → `"bonjour"`
`s * n`	répétition	`"ha" * 3` → `"hahaha"`
`len(s)`	longueur	`len("Python")` → `6`
`s[i]`	accès au i-ème caractère (à partir de 0)	`"abc"[1]` → `"b"`
`s[i:j]`	tranche (*slice*) : caractères d'indice i à $j-1$	`"abcdef"[1:4]` → `"bcd"`

Propriété : Indexation

Pour une chaîne `s` de longueur n :
- Les indices **positifs** vont de `0` (premier caractère) à `n-1` (dernier).
- Les indices **négatifs** : `s[-1]` est le dernier caractère, `s[-2]` l'avant-dernier, etc.
- Un indice hors de `[-n, n-1]` provoque une **erreur** à l'exécution (`IndexError`).

```
s = "Python"
print(s[0])       # affiche P
print(s[-1])      # affiche n
print(s[1:4])     # affiche yth
print(len(s))     # affiche 6
```

Méthodes utiles sur les chaînes :

`s.upper()`	chaîne en majuscules
`s.lower()`	chaîne en minuscules
`s.split(sep)`	liste des sous-chaînes séparées par `sep`
`s.replace(a, b)`	chaîne où chaque occurrence de `a` est remplacée par `b`
`c in s`	`True` si le caractère `c` apparaît dans `s`

Remarque : ces méthodes renvoient toutes une **nouvelle** chaîne ; `s` n'est jamais modifiée (les chaînes sont **immuables** en Python).

```
mot = "bonjour"
print(mot.replace("o", "0"))     # affiche b0nj0ur
print(mot)                       # affiche bonjour (mot n'a pas change)
```

1.6 Entrées et sorties

1.6.1 Sorties

Définition : `print`

`print(expr1, expr2, ...)` affiche les valeurs des expressions séparées par des espaces, puis passe à la ligne.

1.6.2 Entrées

Définition : `input`

★ **Hors programme** — la fonction `input(message)` affiche `message`, attend une saisie au clavier et renvoie une **chaîne de caractères**. Elle est utile pour écrire des scripts interactifs, mais ne figure pas au programme officiel : elle ne sera vraisemblablement pas exigée au concours.

`input` renvoie toujours une `str`. Pour obtenir un nombre, il faut convertir :

```python
x = float(input("Entrer un nombre : "))
n = int(input("Entrer un entier : "))
```

Exemple (complément) : calcul du discriminant d'un trinôme $ax^2 + bx + c$, saisi au clavier.

```python
a = float(input("Entrer a : "))
b = float(input("Entrer b : "))
c = float(input("Entrer c : "))
print("Le discriminant vaut :", b**2 - 4*a*c)
```

1.6.3 Formatage des chaînes (f-strings)

Définition : f-string

★ **Hors programme** — les f-strings ne figurent pas au programme officiel : cette section ne sera vraisemblablement pas exigée au concours, mais elle est très utile en pratique pour afficher des résultats.

Une **f-string** (chaîne formatée) est une chaîne précédée de `f` dans laquelle on peut insérer des expressions Python entre accolades `{...}` :

```python
x = 3.14159
print(f"x vaut {x}")              # x vaut 3.14159
print(f"x arrondi : {x:.2f}")     # x arrondi : 3.14
print(f"2 + 2 = {2 + 2}")         # 2 + 2 = 4
```

Le **spécificateur de format** (après `:`) contrôle l'affichage :

`f"{x:.3f}"`	flottant avec 3 décimales
`f"{x:.2e}"`	notation scientifique

```python
n = 42
pi = 3.14159265
print(f"n = {n}, pi = {pi:.4f}")    # n = 42, pi = 3.1416

# Sans f-string, equivalent avec print :
print("n =", n, ", pi =", round(pi, 4))
```

1.7 Fonctions natives

Python fournit un ensemble de **fonctions natives** (*built-in functions*) disponibles sans importation. En plus de `print` et `input` (section 5) et des fonctions numériques `abs`, `round`, `max`, `min` (section 3), voici les principales :

Conversion de type :

`int(x)`	convertit `x` en entier	(`int(3.7)` → `3`, `int("42")` → `42`)
`float(x)`	convertit `x` en flottant	(`float("3.14")` → `3.14`)
`str(x)`	convertit `x` en chaîne	(`str(42)` → `"42"`)
`bool(x)`	convertit `x` en booléen	(`bool(0)` → `False`)
`type(x)`	renvoie le type de `x`	(`type(3)` → `<class 'int'>`)

Fonctions sur les séquences :

`len(s)`	longueur de la séquence `s`
`sum(s)`	somme des éléments de `s`
`sorted(s)`	renvoie une nouvelle liste triée
`range(n)`	entiers de `0` à `n-1` (voir chapitre 3)

compléments

`reversed(s)`	itérateur parcourant `s` en sens inverse
`enumerate(s)`	produit des paires `(indice, valeur)`

```python
s = "banane"
print(len(s))        # 6
print(sorted(s))     # ['a', 'a', 'b', 'e', 'n', 'n']

for i, c in enumerate(s):
    print(i, c)      # 0 b / 1 a / 2 n / 3 a / 4 n / 5 e
```

Ces fonctions natives couvrent les opérations générales les plus courantes. Pour définir ses propres fonctions adaptées à un problème, voir le chapitre 4.

2

Instructions conditionnelles

Plan du chapitre

2.1 L'instruction `if` **17**
2.2 Conditions composées **18**
2.3 Imbrication de conditions **18**
2.4 Exemples algorithmiques **19**
2.4.1 Parité d'un entier 19
2.4.2 Signe d'un nombre 19
2.4.3 Maximum de deux valeurs 19
2.4.4 Maximum de trois valeurs 19
2.4.5 Discriminant d'un trinôme 20

2.1 L'instruction `if`

En Python, une **instruction conditionnelle** permet d'exécuter un bloc de code uniquement si une condition est vérifiée.

Définition : Instruction `if` / `else`

```
if condition:
    instruction_1    # executee si condition est True
    instruction_2
    ...
else:
    instruction_3    # executee si condition est False
    instruction_4
    ...
```

La clause `else` est **facultative** : on peut écrire un `if` sans `else`.

Remarque fondamentale — l'indentation : En Python, le bloc d'instructions associé à un `if` (ou `else`) est délimité par l'**indentation** (décalage vers la droite), et non par des accolades comme dans d'autres langages. L'indentation standard est de **4 espaces**.

```
x = 5
if x > 0:
    print("x est positif")    # dans le bloc if
    print("et non nul")       # dans le bloc if
print("fin du programme")     # hors du bloc if, toujours execute
```

Définition : Instruction `if` / `elif` / `else`

Lorsqu'il y a plusieurs cas à distinguer, on utilise `elif` (contraction de *else if*) :

```
if condition_1:
    ...          # si condition_1 est True
elif condition_2:
    ...          # si condition_1 est False et condition_2 est True
elif condition_3:
    ...          # si les deux precedentes sont False et condition_3 est True
else:
    ...          # si toutes les conditions precedentes sont False
```

Les conditions sont testées **dans l'ordre**. Dès qu'une condition est vraie, le bloc correspondant est exécuté et les suivants sont ignorés.

Remarque : on peut enchaîner autant de `elif` que nécessaire. La clause `else` finale est facultative, mais recommandée pour traiter tous les cas.

2.2 Conditions composées

Les conditions utilisées dans un `if` sont des expressions booléennes. On peut les combiner avec les opérateurs `and`, `or` et `not` vus au chapitre 1.

```
x = 3
if x > 0 and x < 10:
    print("x est compris entre 0 et 10 (exclus)")

if x < 0 or x > 100:
    print("x est hors de [0, 100]")

if not (x == 0):
    print("x est non nul")
```

Remarque : Python autorise les comparaisons enchaînées. La condition `0 < x < 10` est équivalente à `x > 0 and x < 10` et s'écrit de façon plus lisible.

2.3 Imbrication de conditions

On peut placer un `if` à l'intérieur d'un autre `if` : on parle de conditions **imbriquées**. Chaque niveau d'imbrication ajoute un niveau d'indentation.

```
x = 5
if x >= 0:
    if x == 0:
        print("x est nul")
    else:
        print("x est strictement positif")
else:
    print("x est strictement negatif")
```

Propriété : `elif` vs imbrication

Lorsque les cas sont **mutuellement exclusifs** et forment une liste de situations distinctes, `elif` est préférable à l'imbrication : le code est plus lisible et plus facile à maintenir.

L'imbrication est réservée aux situations où une condition dépend du résultat d'une autre.

2.4 Exemples algorithmiques

2.4.1 Parité d'un entier

```
n = int(input("Entrer un entier : "))
if n % 2 == 0:
    print(n, "est pair")
else:
    print(n, "est impair")
```

2.4.2 Signe d'un nombre

```
x = float(input("Entrer un nombre : "))
if x > 0:
    print("x est positif")
elif x < 0:
    print("x est negatif")
else:
    print("x est nul")
```

2.4.3 Maximum de deux valeurs

```
a = float(input("Entrer a : "))
b = float(input("Entrer b : "))
if a >= b:
    m = a
else:
    m = b
print("Le maximum est", m)
```

Remarque : Python dispose de la fonction native `max(a, b)` qui réalise la même chose.

2.4.4 Maximum de trois valeurs

On illustre deux approches pour calculer le maximum de trois valeurs a, b, c.

```
a = float(input("Entrer a : "))
b = float(input("Entrer b : "))
c = float(input("Entrer c : "))

# Approche 1 : comparaisons successives
m = a
if b > m:
    m = b
if c > m:
    m = c
print("Le maximum est", m)
```

```
# Approche 2 : conditions combinees avec elif
if a >= b and a >= c:
    m = a
elif b >= c:
    m = b
else:
    m = c
print("Le maximum est", m)
```

2.4.5 Discriminant d'un trinôme

Étant donné un trinôme ax^2+bx+c avec $a \neq 0$, on détermine le nombre de racines réelles à partir du discriminant $\Delta = b^2 - 4ac$.

```
a = float(input("Entrer a : "))
b = float(input("Entrer b : "))
c = float(input("Entrer c : "))
delta = b**2 - 4*a*c
if delta > 0:
    print("Deux racines :", (-b - delta**0.5)/(2*a),
                            (-b + delta**0.5)/(2*a))
elif delta == 0:
    print("Une racine double :", -b/(2*a))
else:
    print("Pas de racine reelle")
```

3

Boucles

Plan du chapitre

3.1 La boucle `while` 21
3.2 La boucle `for` 22
3.3 Comparaison `while` / `for` 23
3.4 Algorithmes classiques 23
3.4.1 Somme et produit 23
3.4.2 Suites récurrentes 23
3.4.3 Recherche du premier entier vérifiant une propriété 23
3.4.4 Algorithme d'Euclide 24

Une **boucle** permet de répéter un bloc d'instructions plusieurs fois. Python dispose de deux structures de boucle : `while` et `for`.

3.1 La boucle `while`

Définition : Boucle `while`

```
while condition:
    instruction_1
    instruction_2
    ...
```

Tant que `condition` est `True`, le bloc d'instructions est exécuté. La condition est testée **avant chaque itération** ; si elle est fausse dès le départ, le bloc n'est jamais exécuté.

Exemple : afficher les entiers de 1 à 5.

```
i = 1
while i <= 5:
    print(i)
    i = i + 1    # sans cette ligne, boucle infinie !
```

Propriété : Condition d'arrêt

Pour qu'une boucle `while` se termine, il faut que la condition devienne `False` à un moment. Cela nécessite que **le bloc modifie au moins une variable intervenant dans la condition**.

Si la condition reste toujours vraie, la boucle ne s'arrête jamais : on parle de **boucle infinie**. Il faut alors interrompre le programme manuellement (Ctrl+C).

Propriété : Terminaison d'une boucle `while`

Pour prouver qu'une boucle `while` se termine, on identifie un **variant de boucle** : une grandeur qui évolue à chaque itération et qui progresse vers la condition de sortie, la boucle s'arrête nécessairement.

• Si le variant est un **entier** positif (ex. $n - i$), la preuve est immédiate : un entier positif qui diminue strictement atteint 0 en un nombre fini d'étapes.

• Si le variant est **réel** (ex. $d - g$ en dichotomie), on peut majorer le nombre d'itérations par une formule.

Exemples de variants courants :

Situation	Variant
`while i < n: i += 1`	$n - i$, entier qui diminue de 1 à chaque tour
`while d - g > eps: ...` (dichotomie)	$d - g$ réel qui est divisé par 2 à chaque tour
`while abs(f(u)) >= eps: ...` (Newton)	nombre d'itérations restantes (majoré par `n_max`)

3.2 La boucle `for`

Définition : Boucle `for` et `range`

```
for variable in range(...):
    instruction_1
    instruction_2
    ...
```

La variable prend successivement chaque valeur produite par `range`.
Les trois formes de `range` sont :

`range(n)`	entiers de 0 à $n-1$	`range(5)` → `0, 1, 2, 3, 4`
`range(a, b)`	entiers de a à $b-1$	`range(2, 6)` → `2, 3, 4, 5`
`range(a, b, p)`	entiers de a à $b-1$ par pas p	`range(0, 10, 3)` → `0, 3, 6, 9`

Le pas p peut être négatif : `range(5, 0, -1)` → `5, 4, 3, 2, 1`.

Exemples :

```
for i in range(4):
    print(i)           # affiche 0, 1, 2, 3

for i in range(1, 6):
    print(i)           # affiche 1, 2, 3, 4, 5

for i in range(0, 11, 2):
    print(i)           # affiche 0, 2, 4, 6, 8, 10
```

Propriété : Itération sur une chaîne de caractères

Une boucle `for` peut parcourir directement une chaîne de caractères : la variable prend successivement la valeur de chaque caractère.

```
for c in "Python":
    print(c)     # affiche P, y, t, h, o, n (un par ligne)
```

Remarque : On peut aussi avec une boucle `for` parcourir les listes par valeurs (voir la chapitre 5).

3.3 Comparaison `while` / `for`

	`for`	`while`
Nombre d'itérations	connu à l'avance	inconnu à l'avance
Usage typique	parcourir une séquence	attendre qu'une condition soit atteinte
Risque de boucle infinie	non	oui

Règle pratique : si on sait combien de fois répéter, on utilise `for`. Sinon, on utilise `while`.

3.4 Algorithmes classiques

3.4.1 Somme et produit

Somme des entiers de 1 à n :

```
n = int(input("Entrer n : "))
s = 0
for i in range(1, n + 1):
    s = s + i
print("La somme vaut", s)
```

Produit des entiers de 1 à n (factorielle) :

```
n = int(input("Entrer n : "))
p = 1
for i in range(1, n + 1):
    p = p * i
print("n! =", p)
```

Remarque : le calcul de somme et de produit suit toujours le même schéma : initialiser un accumulateur (`s = 0` pour une somme, `p = 1` pour un produit), puis le mettre à jour à chaque itération.

3.4.2 Suites récurrentes

Une suite définie par $u_0 = a$ et $u_{n+1} = f(u_n)$ se calcule facilement avec une boucle `for`.

Exemple : calculer les 10 premiers termes de la suite $u_0 = 1$, $u_{n+1} = \frac{u_n}{2} + 1$.

```
u = 1.0
for i in range(10):
    print("u_", i, "=", u)
    u = u / 2 + 1
```

3.4.3 Recherche du premier entier vérifiant une propriété

Lorsqu'on cherche le premier entier n vérifiant une condition, on utilise `while` car on ne sait pas à l'avance combien d'itérations seront nécessaires.

Exemple : trouver le plus petit entier n tel que $2^n > 1000$.

```
n = 0
while 2**n <= 1000:
    n = n + 1
print("Le plus petit n tel que 2**n > 1000 est", n)
```

3.4.4 Algorithme d'Euclide

Le PGCD de deux entiers a et b ($a \geqslant b > 0$) peut être calculé par l'algorithme d'Euclide : on remplace (a, b) par $(b, a \mod b)$ jusqu'à ce que le reste soit nul.

```
a = int(input("Entrer a : "))
b = int(input("Entrer b : "))
while b != 0:
    a, b = b, a % b
print("PGCD =", a)
```

★ **Complément** —

Instructions `break` et `continue`

`break` interrompt immédiatement la boucle en cours, quelle que soit la condition.
`continue` passe directement à l'itération suivante sans exécuter la fin du bloc.

```
# Trouver le premier multiple de 7 superieur a 50
i = 51
while True:         # boucle a priori infinie
    if i % 7 == 0:
        print(i)
        break       # on sort des qu'on a trouve
    i = i + 1
```

Exemple avec `continue` :

```
# Afficher les entiers de 1 a 10 en sautant les multiples de 3
for i in range(1, 11):
    if i % 3 == 0:
        continue        # on passe directement a i suivant
    print(i)            # affiche 1, 2, 4, 5, 7, 8, 10
```

Ces instructions rendent parfois le code plus lisible, mais peuvent aussi le rendre difficile à analyser. On les utilise avec parcimonie.

4

Fonctions

Plan du chapitre

4.1 Définir et appeler une fonction **25**
4.2 Valeur de retour **26**
4.3 Portée des variables **26**
4.4 Paramètres par défaut **27**
4.5 Spécification d'une fonction **27**
4.6 Exemples **27**
4.6.1 Parité 27
4.6.2 Factorielle 28
4.6.3 PGCD (algorithme d'Euclide) 28
4.6.4 Test de primalité 28

Une **fonction** est un bloc de code auquel on donne un nom, que l'on peut **appeler** autant de fois que nécessaire avec des données différentes. Les fonctions permettent de structurer un programme, d'éviter les répétitions et de rendre le code lisible. Python propose également des **fonctions natives** (`len`, `sum`, `sorted`, `int`, `float`...) présentées au chapitre 1, section 6 ; ce chapitre explique comment définir ses propres fonctions.

4.1 Définir et appeler une fonction

Définition : Syntaxe d'une fonction

```
def nom_fonction(parametre_1, parametre_2, ...):
    instruction_1
    instruction_2
    ...
    return valeur
```

• `def` introduit la définition de la fonction. • Les **paramètres** sont les données reçues par la fonction (entre parenthèses). • Le corps de la fonction est indenté (comme pour `if` et `while`). • `return` renvoie une valeur et **termine immédiatement** la fonction.

Exemple :

```
def maximum(a, b):
    if a >= b:
        return a
    return b

m = maximum(3, 7)    # appel : a vaut 3, b vaut 7
print(m)             # affiche 7
print(maximum(10, 2))  # affiche 10
```

Remarques :

• Définir une fonction **n'exécute pas** son corps : il faut l'appeler pour cela.

• À l'appel, les **arguments** (valeurs fournies) sont associés aux paramètres dans l'ordre.

• Une fonction peut être appelée depuis n'importe quel endroit du programme, y compris dans une autre fonction.

4.2 Valeur de retour

Propriété : Instruction `return`

• `return expr` renvoie la valeur de `expr` et **termine immédiatement** la fonction : aucune instruction suivante n'est exécutée.

• Une fonction peut contenir **plusieurs** `return` (dans des branches différentes).

• Si l'exécution atteint la fin du corps sans rencontrer de `return`, la fonction renvoie `None`.

• `return` sans expression renvoie également `None`.

Conséquence : lorsque la branche `if` se termine par `return`, le `else` est inutile — si la condition était vraie, la fonction s'est déjà arrêtée ; le code qui suit ne s'exécute que si la condition était fausse. On préfère donc écrire :

```
def valeur_absolue(x):
    if x < 0:
        return -x
    return x       # atteint seulement si x >= 0
```

```
def signe(x):
    if x > 0:
        return "positif"
    if x < 0:
        return "negatif"
    return "nul"

print(signe(-3))    # affiche negatif
print(signe(0))     # affiche nul
```

Remarque : une fonction qui se contente d'afficher un résultat (avec `print`) sans `return` ne renvoie rien d'utilisable. On préfère en général écrire des fonctions qui **renvoient** une valeur, et laisser l'affichage au code appelant.

4.3 Portée des variables

Définition : Variables locales et globales

• Une variable définie **à l'intérieur** d'une fonction est **locale** : elle n'existe que pendant l'exécution de la fonction et disparaît ensuite.

• Une variable définie **en dehors** de toute fonction est **globale** : elle est accessible en lecture depuis l'intérieur d'une fonction.

```
x = 10          # variable globale

def f():
    x = 5       # variable locale : ne modifie pas le x global
    print(x)    # affiche 5

f()
print(x)        # affiche 10 (x global inchange)
```

Remarque : il existe l'instruction `global x` qui permet de modifier une variable globale depuis une fonction, mais c'est une **mauvaise pratique** à éviter : elle rend le programme difficile à lire et à déboguer. On préfère toujours passer les données en paramètres et récupérer le résultat avec `return`.

4.4 Paramètres par défaut

Propriété : Valeurs par défaut

On peut donner une **valeur par défaut** à un paramètre. Si l'argument correspondant est omis lors de l'appel, la valeur par défaut est utilisée.

```
def puissance(x, n=2):
    p = 1
    for i in range(n):
        p = p * x
    return p

print(puissance(3))      # n vaut 2 par defaut : affiche 9
print(puissance(3, 4))   # n vaut 4 : affiche 81
```

Les paramètres avec valeur par défaut doivent être placés **après** les paramètres sans valeur par défaut.

4.5 Spécification d'une fonction

Définition : Spécification (docstring)

La **spécification** d'une fonction décrit ce qu'elle fait, ses paramètres et ce qu'elle renvoie, sans entrer dans les détails de l'implémentation. En Python, elle s'écrit sous forme d'une **docstring** : une chaîne de caractères placée immédiatement après la ligne `def`.

```
def pgcd(a, b):
    """Renvoie le PGCD de deux entiers a et b (a, b > 0)."""
    while b != 0:
        a, b = b, a % b
    return a
```

Remarque : écrire une spécification est une bonne habitude. Cela force à réfléchir à ce que doit faire la fonction avant de l'écrire, et aide les autres (et soi-même) à la comprendre et à l'utiliser correctement.

4.6 Exemples

4.6.1 Parité

```
def est_pair(n):
    """Renvoie True si l'entier n est pair, False sinon."""
    return n % 2 == 0

print(est_pair(4))    # True
print(est_pair(7))    # False
```

4.6.2 Factorielle

```python
def factorielle(n):
    """Renvoie n! pour un entier n >= 0."""
    p = 1
    for i in range(1, n + 1):
        p = p * i
    return p

print(factorielle(5))    # 120
```

4.6.3 PGCD (algorithme d'Euclide)

```python
def pgcd(a, b):
    """Renvoie le PGCD de deux entiers strictement positifs a et b."""
    while b != 0:
        a, b = b, a % b
    return a

print(pgcd(48, 18))    # 6
```

4.6.4 Test de primalité

Un entier $n \geqslant 2$ est premier si et seulement s'il n'est divisible par aucun entier compris entre 2 et $\lfloor \sqrt{n} \rfloor$.

```python
def est_premier(n):
    """Renvoie True si n est premier, False sinon (n entier >= 2)."""
    if n < 2:
        return False
    for d in range(2, int(n**0.5) + 1):
        if n % d == 0:
            return False
    return True

print(est_premier(17))    # True
print(est_premier(15))    # False
```

★ **Complément —**

Effets de bord

On dit qu'une fonction a un **effet de bord** lorsqu'elle modifie quelque chose en dehors d'elle-même : afficher à l'écran, modifier une variable globale, ou modifier un objet reçu en paramètre.

Une fonction **pure** n'a aucun effet de bord : elle se contente de calculer et de renvoyer une valeur. C'est le modèle à privilégier.

```python
# Fonction pure : pas d'effet de bord
def double(x):
    return 2 * x

# Fonction avec effet de bord (affichage)
def double_affiche(x):
    print(2 * x)    # effet de bord : affichage
```

Les effets de bord liés à la modification d'objets reçus en paramètre (listes, etc.) seront abordés dans le chapitre sur la mutabilité.

5

Listes

Plan du chapitre

5.1 Création d'une liste 29
5.2 Accès aux éléments 30
5.3 Modification d'une liste 30
5.4 Opérations sur les listes 31
5.5 Parcours d'une liste 31
5.6 Compréhension de liste 31
5.7 Copie d'une liste 32
5.8 P-uplets (tuples) 33
5.8.1 Déballage de tuple 33
5.8.2 Renvoi de plusieurs valeurs par une fonction 33
5.8.3 Tuples dans les boucles `for` 33
5.9 Ensembles (`set`) 34

Une **liste** est une suite ordonnée d'éléments, éventuellement de types différents. C'est la structure de données la plus utilisée en Python pour regrouper plusieurs valeurs.

5.1 Création d'une liste

Définition : Liste

Une liste se crée en plaçant ses éléments entre crochets, séparés par des virgules :

```
L = [3, 1, 4, 1, 5]         # liste de 5 entiers
M = ["chat", "chien", "oiseau"]   # liste de chaines
N = [1, 3.14, True, "abc"]   # types melanges
vide = []                    # liste vide
```

La fonction `list(s)` convertit une séquence `s` (chaîne, `range`, etc.) en liste :

```
list("abc")         # ['a', 'b', 'c']
list(range(5))      # [0, 1, 2, 3, 4]
```

5.2 Accès aux éléments

Propriété : Indexation

Pour une liste `L` de longueur n :

- `L[i]` désigne l'élément d'indice i, les indices commençant à `0`.
- Les indices **négatifs** permettent de compter depuis la fin : `L[-1]` est le dernier élément, `L[-2]` l'avant-dernier, etc.
- Un indice hors de $[\![0, n-1]\!]$ provoque une **erreur** (`IndexError`).

```
L = [10, 20, 30, 40, 50]
print(L[0])     # 10
print(L[2])     # 30
print(L[-1])    # 50
print(L[-2])    # 40
```

Propriété : Tranches (*slices*)

`L[i:j]` renvoie une **nouvelle liste** contenant les éléments d'indice i à $j-1$:

```
L = [10, 20, 30, 40, 50]
print(L[1:4])     # [20, 30, 40]
print(L[:3])      # [10, 20, 30]  (depuis le debut)
print(L[2:])      # [30, 40, 50]  (jusqu'a la fin)
print(L[:])       # [10, 20, 30, 40, 50]  (copie)
print(L[::2])     # [10, 30, 50]  (un element sur deux)
print(L[::-1])    # [50, 40, 30, 20, 10]  (liste inversee)
```

5.3 Modification d'une liste

Les listes sont **modifiables** : on peut changer, ajouter ou supprimer des éléments après la création.

Définition : Opérations de modification

`L[i] = v`	modifie l'élément d'indice i
`L.append(x)`	ajoute `x` en fin de liste
`L.pop()`	supprime et renvoie le dernier élément
`L.pop(i)`	supprime et renvoie l'élément d'indice i
`L.insert(i, x)`	insère `x` à l'indice i
`L.remove(x)`	supprime la première occurrence de `x`
`del L[i]`	supprime l'élément d'indice i

```
L = [1, 2, 3]
L[0] = 10           # L vaut [10, 2, 3]
L.append(4)         # L vaut [10, 2, 3, 4]
L.pop()             # renvoie 4, L vaut [10, 2, 3]
L.insert(1, 99)     # L vaut [10, 99, 2, 3]
del L[2]            # L vaut [10, 99, 3]
```

5.4 Opérations sur les listes

Propriété : Opérations courantes

`len(L)`	longueur de la liste	`len([1,2,3])` → `3`
`L + M`	concaténation	`[1,2]+[3,4]` → `[1,2,3,4]`
`L * n`	répétition	`[0]*3` → `[0,0,0]`
`x in L`	appartenance (`True`/`False`)	`3 in [1,2,3]` → `True`
`x not in L`	non-appartenance	`5 not in [1,2,3]` → `True`
`min(L)`	plus petit élément	
`max(L)`	plus grand élément	
`sum(L)`	somme des éléments	

5.5 Parcours d'une liste

Il existe deux façons de parcourir une liste avec une boucle `for`.

Propriété : Parcours par valeur et par indice

Parcours par valeur — on accède directement à chaque élément :

```
L = [10, 20, 30]
for x in L:
    print(x)        # affiche 10, puis 20, puis 30
```

Parcours par indice — utile quand on a besoin de la position :

```
L = [10, 20, 30]
for i in range(len(L)):
    print(i, L[i])    # affiche 0 10, puis 1 20, puis 2 30
```

Le parcours par indice est indispensable lorsqu'on veut **modifier** les éléments de la liste ou accéder à plusieurs indices simultanément.

5.6 Compréhension de liste

Définition : Liste en compréhension

La **compréhension de liste** permet de construire une liste en une seule ligne à partir d'une séquence :

```
[expression for variable in sequence]
```

On peut ajouter une **condition** pour filtrer les éléments :

```
[expression for variable in sequence if condition]
```

Exemples :

```
carres = [x**2 for x in range(1, 6)]
# [1, 4, 9, 16, 25]

pairs = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]

longueurs = [len(mot) for mot in ["chat", "chien", "oiseau"]]
# [4, 5, 6]
```

Remarque : une compréhension de liste est équivalente à une boucle `for` avec `append`, mais plus concise et souvent plus lisible :

```
# Ces deux codes produisent le meme resultat :
carres = [x**2 for x in range(1, 6)]

carres = []
for x in range(1, 6):
    carres.append(x**2)
```

Double boucle : une compréhension peut contenir plusieurs clauses `for`, ce qui revient à imbriquer des boucles. La clause la plus à gauche est la boucle externe.

```
# Toutes les paires (x, y) avec x dans L1 et y dans L2
L1 = [1, 2, 3]
L2 = [10, 20]
paires = [(x, y) for x in L1 for y in L2]
# [(1,10),(1,20),(2,10),(2,20),(3,10),(3,20)]

# Equivalent avec des boucles for imbriquees :
paires = []
for x in L1:
    for y in L2:
        paires.append((x, y))
```

```
# Aplatir une liste de listes
M = [[1, 2, 3], [4, 5], [6, 7, 8]]
plat = [x for ligne in M for x in ligne]
# [1, 2, 3, 4, 5, 6, 7, 8]
```

Expressions génératrices : en remplaçant les crochets par des parenthèses dans `sum`, `min`, `max`, on obtient une **expression génératrice** qui calcule le résultat sans construire la liste intermédiaire :

```
# Compréhension de liste (construit toute la liste, puis la somme)
s = sum([x**2 for x in range(1000)])

# Expression generatrice (calcule element par element, plus economique)
s = sum(x**2 for x in range(1000))

# Avec condition : compter les elements verifiant un critere
nb_pairs = sum(1 for x in range(100) if x % 2 == 0)    # 50
```

5.7 Copie d'une liste

Attention : l'affectation `M = L` ne crée **pas** une copie de la liste `L` — les deux noms désignent alors la **même** liste. Toute modification de `M` affecte aussi `L`.

```
L = [1, 2, 3]
M = L            # M et L designent la meme liste
M[0] = 99
print(L)         # [99, 2, 3]  L est modifiee !
```

Pour obtenir une **copie indépendante**, on utilise une tranche complète ou `list()` :

```
L = [1, 2, 3]
M = L[:]         # copie independante
M[0] = 99
print(L)         # [1, 2, 3]  L est inchangee
```

> ★ **Complément** —
> Le comportement de `M = L` s'explique par la notion de **mutabilité** : les listes sont des objets modifiables, et une affectation ne copie pas l'objet mais crée un second nom pointant vers le même objet en mémoire. Ce mécanisme, ainsi que ses conséquences dans les fonctions (effets de bord), est étudié dans le chapitre 13.

5.8 P-uplets (tuples)

Définition : Tuple

Un **tuple** (ou **p-uplet**) est une suite ordonnée d'éléments, comme une liste, mais **immuable** : on ne peut pas modifier ses éléments après sa création. On le crée avec des parenthèses (ou simplement des virgules) :

```
t = (1, 2, 3)           # tuple de 3 entiers
t = 1, 2, 3             # les parentheses sont optionnelles
t = ("chat", 3.14)      # types melanges
t = (42,)               # tuple a un seul element (la virgule est obligatoire)
```

L'accès aux éléments se fait comme pour une liste : `t[0]`, `t[-1]`, `t[1:3]`.

5.8.1 Déballage de tuple

Une opération très courante est le **déballage** (*unpacking*) : affecter les éléments d'un tuple à plusieurs variables en une seule ligne.

```
t = (3, 7)
a, b = t          # a vaut 3, b vaut 7

# Echange de deux variables (sans variable temporaire)
a, b = b, a
```

5.8.2 Renvoi de plusieurs valeurs par une fonction

Lorsqu'une fonction **renvoie plusieurs valeurs** avec `return a, b`, elle renvoie en réalité un tuple. On récupère les valeurs par déballage :

```
def min_max(L):
    """Renvoie le minimum et le maximum de la liste L."""
    return min(L), max(L)

m, M = min_max([3, 1, 4, 1, 5, 9])
print(m, M)    # 1  9
```

5.8.3 Tuples dans les boucles `for`

On rencontre souvent des listes de tuples (coordonnées, paires clé/valeur, mesures). La boucle `for` permet de déballer chaque tuple directement :

```
points = [(0, 0), (1, 2), (3, -1)]
for x, y in points:
    print(f"x={x}, y={y}")

# Autre exemple : liste de paires (temps, valeur)
mesures = [(0.0, 20.1), (0.5, 21.3), (1.0, 22.8)]
for t, temp in mesures:
    print(t, temp)
```

Listes vs tuples :

	Liste	Tuple
Syntaxe	`[1, 2, 3]`	`(1, 2, 3)`
Modifiable	oui	non
Utilisation typique	collection de données	valeur composite fixe

En pratique, on utilise les tuples pour regrouper des valeurs qui vont naturellement ensemble (coordonnées, résultat d'une fonction à deux sorties, ligne d'une base de données) et les listes pour des collections qu'on modifie.

5.9 Ensembles (`set`)

Définition : Type `set`

Un **set** (ensemble) est une collection **non ordonnée** d'éléments **uniques**. Il se crée avec des accolades ou `set()` :

```
s = {1, 2, 3, 2, 1}    # {1, 2, 3}  (doublons supprimes automatiquement)
t = set([3, 4, 5])      # depuis une liste
vide = set()            # set vide  -- attention : {} creerait un dict vide
```

Opérations courantes :

`x in s`	test d'appartenance
`s.add(x)`	ajout d'un élément
`s.remove(x)`	suppression (erreur si `x` absent)
`s | t`	union
`s & t`	intersection
`s - t`	différence
`len(s)`	nombre d'éléments

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
print(a | b)     # {1, 2, 3, 4, 5, 6}
print(a & b)     # {3, 4}
print(a - b)     # {1, 2}
print(3 in a)    # True
```

Remarque : un `set` est un type **mutable** : `add` et `remove` modifient l'objet en place (comme pour les listes). Cette notion de mutabilité, ainsi que ses conséquences (aliasing, effets de bord), est étudiée dans le chapitre 13. En revanche, ses éléments doivent être **immuables** (entiers, flottants, chaînes, tuples) — on ne peut pas mettre de listes dans un set.

Cas d'usage typiques :

- **Éliminer les doublons** d'une liste : `valeurs_uniques = list(set(L))`
- **Tester l'appartenance** à une grande collection : `x in s` est très rapide, indépendamment du nombre d'éléments.

6

Algorithmes sur les listes

Plan du chapitre

6.1 Calculs sur une liste **35**
6.1.1 Somme et produit 35
6.1.2 Recherche du maximum 36
6.1.3 Comptage d'occurrences 36
6.2 Recherche dans une liste **36**
6.2.1 Présence d'un élément vérifiant une propriété 36
6.2.2 Appartenance à une liste 37
6.3 Tests sur les listes **37**
6.3.1 Palindrome 37
6.3.2 Listes distinctes et doublons 37
6.3.3 Anagrammes 38
6.4 Construction de listes **38**
6.4.1 Filtrage 38
6.4.2 Transformation 38
6.5 Listes de listes (tableaux 2D) **38**
6.5.1 Parcours d'un tableau 2D 39
6.5.2 Transposée d'une matrice 39
6.5.3 Produit de matrices 40
6.6 Application : nombres parfaits **40**

Ce chapitre présente les algorithmes fondamentaux sur les listes. On y apprend à calculer des valeurs à partir d'une liste (somme, maximum, comptage), à tester des propriétés (palindrome, présence d'un élément), et à construire de nouvelles listes. On aborde aussi les **listes de listes**, qui permettent de représenter des tableaux à deux dimensions.

6.1 Calculs sur une liste

Le schéma de base est celui de l'**accumulateur** : on initialise une variable avant la boucle, puis on la met à jour à chaque étape.

6.1.1 Somme et produit

```
def somme(L):
    """Renvoie la somme des elements de L (L liste de nombres)."""
    s = 0
    for x in L:
        s = s + x
    return s

def produit(L):
    """Renvoie le produit des elements de L (L liste non vide de nombres)."""
    p = 1
```

```
    for x in L:
        p = p * x
    return p
```

Remarque : les fonctions natives `sum(L)`, `min(L)`, `max(L)` existent, mais savoir les programmer reste essentiel pour les adapter.

6.1.2 Recherche du maximum

Sans utiliser `max`, on peut chercher le maximum en maintenant le meilleur candidat rencontré :

```
def maximum(L):
    """Renvoie le plus grand element de L (L liste non vide)."""
    m = L[0]          # meilleur candidat courant
    for x in L:
        if x > m:
            m = x
    return m
```

Variante : trouver en même temps le maximum et son indice.

```
def indice_max(L):
    """Renvoie l'indice du plus grand element de L (L liste non vide)."""
    i_max = 0
    for i in range(len(L)):
        if L[i] > L[i_max]:
            i_max = i
    return i_max
```

6.1.3 Comptage d'occurrences

```
def compte(L, v):
    """Renvoie le nombre de fois que v apparait dans la liste L."""
    c = 0
    for x in L:
        if x == v:
            c = c + 1
    return c
```

Remarque : la méthode `L.count(v)` fait la même chose.

6.2 Recherche dans une liste

6.2.1 Présence d'un élément vérifiant une propriété

Le schéma consiste à parcourir la liste et à renvoyer `True` dès qu'on trouve un élément satisfaisant, ou `False` après avoir tout parcouru sans succès.

Exemple : tester si une liste contient un entier pair.

```
def contient_pair(L):
    """Renvoie True si L contient au moins un entier pair."""
    for x in L:
        if x % 2 == 0:
            return True
    return False
```

Propriété : Schéma de recherche

Pour tester si **au moins un** élément vérifie une propriété P :

```
for x in L:
    if P(x):
        return True
return False
```

Pour tester si **tous** les éléments vérifient une propriété P :

```
for x in L:
    if not P(x):
        return False
return True
```

6.2.2 Appartenance à une liste

```
def appartient(v, L):
    """Renvoie True si v est dans la liste L."""
    for x in L:
        if x == v:
            return True
    return False
```

Remarque : l'opérateur `in` fait exactement cela (`v in L`), mais savoir l'écrire est important pour les adaptations.

6.3 Tests sur les listes

6.3.1 Palindrome

Une liste (ou une chaîne) est un **palindrome** si elle se lit de la même façon dans les deux sens.

```
def est_palindrome(L):
    """Renvoie True si la liste L est un palindrome."""
    n = len(L)
    for i in range(n // 2):
        if L[i] != L[n - 1 - i]:
            return False
    return True

print(est_palindrome([1, 2, 3, 2, 1]))    # True
print(est_palindrome([1, 2, 3]))          # False
```

Remarque : pour une chaîne de caractères, on utilise exactement le même algorithme (ou `s == s[::-1]`).

6.3.2 Listes distinctes et doublons

Tester si tous les éléments sont distincts :

```
def tous_distincts(L):
    """Renvoie True si tous les elements de L sont distincts."""
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] == L[j]:
                return False
    return True
```

Construire la liste sans doublons :

```python
def sans_doublon(L):
    """Renvoie une liste contenant les elements de L sans repetition."""
    R = []
    for x in L:
        if x not in R:
            R.append(x)
    return R

print(sans_doublon([1, 3, 2, 1, 4, 3]))    # [1, 3, 2, 4]
```

6.3.3 Anagrammes

Deux mots sont **anagrammes** s'ils contiennent les mêmes lettres, avec les mêmes multiplicités. On peut les comparer en triant leurs lettres.

```python
def sont_anagrammes(mot1, mot2):
    """Renvoie True si mot1 et mot2 sont des anagrammes."""
    return sorted(mot1) == sorted(mot2)

print(sont_anagrammes("chien", "niche"))    # True
print(sont_anagrammes("chat", "tache"))     # False
```

6.4 Construction de listes

6.4.1 Filtrage

Construire une liste en ne gardant que les éléments vérifiant une condition.

```python
def multiples(L, k):
    """Renvoie la liste des elements de L divisibles par k."""
    R = []
    for x in L:
        if x % k == 0:
            R.append(x)
    return R

# equivalent en comprehension :
def multiples_comp(L, k):
    return [x for x in L if x % k == 0]
```

6.4.2 Transformation

Appliquer une opération à chaque élément.

```python
def carres(L):
    """Renvoie la liste des carres des elements de L."""
    return [x**2 for x in L]
```

6.5 Listes de listes (tableaux 2D)

Une **liste de listes** permet de représenter un tableau à deux dimensions, comme une matrice.

Définition : Tableau 2D

On représente un tableau de n lignes et p colonnes par une liste de n listes, chacune de longueur p :

```
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
```

- `M[i]` désigne la ligne d'indice i (une liste).
- `M[i][j]` désigne l'élément à la ligne i, colonne j.
- `len(M)` donne le nombre de lignes, `len(M[0])` le nombre de colonnes.

Création d'un tableau de zéros (n lignes, p colonnes) :

```
def zeros(n, p):
    """Renvoie un tableau n lignes x p colonnes rempli de zeros."""
    return [[0] * p for i in range(n)]
```

Attention à la copie : ne pas écrire `[[0]*p]*n` — toutes les lignes seraient le même objet en mémoire.

```
# CORRECT : chaque ligne est un objet distinct
M = [[0] * 3 for i in range(2)]
M[0][1] = 99
print(M)     # [[0, 99, 0], [0, 0, 0]]  correct

# INCORRECT : toutes les lignes partagent la meme liste
M = [[0] * 3] * 2
M[0][1] = 99
print(M)     # [[0, 99, 0], [0, 99, 0]]  FAUX !
```

6.5.1 Parcours d'un tableau 2D

```
def somme_tableau(M):
    """Renvoie la somme de tous les elements du tableau M."""
    s = 0
    for i in range(len(M)):
        for j in range(len(M[i])):
            s = s + M[i][j]
    return s
```

6.5.2 Transposée d'une matrice

La **transposée** d'une matrice M à n lignes et p colonnes est la matrice M^T à p lignes et n colonnes telle que $M^T[j][i] = M[i][j]$.

```
def transposee(M):
    """Renvoie la transposee de la matrice M."""
    n = len(M)
    p = len(M[0])
    T = [[0] * n for j in range(p)]
    for i in range(n):
        for j in range(p):
            T[j][i] = M[i][j]
    return T
```

6.5.3 Produit de matrices

Le produit de deux matrices A $(n \times p)$ et B $(p \times q)$ est la matrice C $(n \times q)$ définie par :

$$C[i][k] = \sum_{j=0}^{p-1} A[i][j] \times B[j][k]$$

```python
def produit_matrices(A, B):
    """Renvoie le produit matriciel A * B.
    A : n lignes x p colonnes, B : p lignes x q colonnes."""
    n = len(A)
    p = len(B)
    q = len(B[0])
    C = [[0] * q for i in range(n)]
    for i in range(n):
        for k in range(q):
            for j in range(p):
                C[i][k] = C[i][k] + A[i][j] * B[j][k]
    return C
```

Remarque : le module `numpy` propose des fonctions optimisées pour le calcul matriciel (`np.dot`, `@`) ; il sera présenté au chapitre 8.

6.6 Application : nombres parfaits

Un entier $n \geqslant 1$ est **parfait** s'il est égal à la somme de ses diviseurs stricts (c'est-à-dire tous ses diviseurs sauf lui-même). Par exemple, $6 = 1 + 2 + 3$ est parfait.

```python
def diviseurs_stricts(n):
    """Renvoie la liste des diviseurs stricts de n."""
    return [d for d in range(1, n) if n % d == 0]

def est_parfait(n):
    """Renvoie True si n est un nombre parfait."""
    return sum(diviseurs_stricts(n)) == n

# Liste des nombres parfaits inferieurs a 1000
parfaits = [n for n in range(1, 1001) if est_parfait(n)]
print(parfaits)   # [6, 28, 496]
```

7

Fichiers texte

Plan du chapitre

7.1 Ouvrir et fermer un fichier 41
7.2 Lire un fichier 41
7.3 Écrire dans un fichier 42
7.4 Fichiers CSV 43
7.4.1 Lire un fichier CSV 43
7.4.2 Écrire un fichier CSV 43
7.5 Exemple complet : analyse d'une série de mesures 43

En sciences, les données sont souvent stockées dans des **fichiers texte** : résultats de mesures, relevés biologiques, séquences génétiques, etc. Python permet de lire et d'écrire ces fichiers facilement. Ce chapitre présente les outils essentiels pour travailler avec des fichiers.

7.1 Ouvrir et fermer un fichier

Définition : La fonction `open`

`open(nom, mode)` ouvre le fichier `nom` et renvoie un **objet fichier**. Le paramètre `mode` précise l'opération souhaitée :

`'r'`	lecture (*read*)	le fichier doit exister
`'w'`	écriture (*write*)	crée ou écrase le fichier
`'a'`	ajout (*append*)	crée ou ajoute à la fin

Il faut **toujours fermer** le fichier après utilisation avec `f.close()`.

Propriété : Instruction `with`

L'instruction `with` ouvre le fichier et le ferme **automatiquement** à la fin du bloc, même en cas d'erreur. C'est la façon recommandée de travailler avec des fichiers.

```
with open("donnees.txt", "r") as f:
    # travailler avec f ici
    contenu = f.read()
# f est automatiquement ferme ici
```

7.2 Lire un fichier

Il existe plusieurs façons de lire le contenu d'un fichier ouvert en lecture.

Définition : Méthodes de lecture

`f.read()`	renvoie tout le contenu sous forme d'une chaîne
`f.readline()`	renvoie la prochaine ligne (avec le `'\n'` final)
`f.readlines()`	renvoie la liste de toutes les lignes

On peut aussi itérer directement sur le fichier, ligne par ligne :

```
with open("donnees.txt", "r") as f:
    for ligne in f:
        print(ligne)
```

Remarque : chaque ligne lue contient le caractère de fin de ligne `'\n'`. On l'enlève avec `ligne.strip()`.

Exemple : supposons que le fichier `temperatures.txt` contienne :

```
15.2
17.8
14.5
19.1
16.3
```

Pour lire toutes les valeurs et les stocker dans une liste :

```
temperatures = []
with open("temperatures.txt", "r") as f:
    for ligne in f:
        t = float(ligne.strip())    # convertir la chaine en flottant
        temperatures.append(t)

print(temperatures)    # [15.2, 17.8, 14.5, 19.1, 16.3]
print("Moyenne :", sum(temperatures) / len(temperatures))
```

7.3 Écrire dans un fichier

Définition : Méthode `write`

`f.write(chaine)` écrit la chaîne `chaine` dans le fichier. Contrairement à `print`, `write` **n'ajoute pas** de retour à la ligne : il faut l'inclure explicitement avec `'\n'`.

Exemple : écrire les carrés de 1 à 5 dans un fichier.

```
with open("carres.txt", "w") as f:
    for i in range(1, 6):
        f.write(str(i) + "^2 = " + str(i**2) + "\n")
```

Le fichier `carres.txt` contiendra :

```
1^2 = 1
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25
```

Remarque : `write` n'accepte que des chaînes de caractères. Pour écrire un nombre `x`, il faut écrire `str(x)`.

7.4 Fichiers CSV

Le format **CSV** (*Comma-Separated Values*) est un format texte très courant pour stocker des données tabulaires : chaque ligne représente un enregistrement, et les valeurs sont séparées par un délimiteur (virgule, point-virgule ou espace).

Exemple de fichier `mesures.csv` :

```
temps;temperature;pression
0;15.2;1013.2
10;16.1;1012.8
20;17.3;1011.9
30;18.0;1012.5
```

7.4.1 Lire un fichier CSV

On lit le fichier ligne par ligne, puis on découpe chaque ligne avec `split(';')` :

```
temps = []
temperature = []

with open("mesures.csv", "r") as f:
    f.readline()   # ignorer la ligne d'en-tete
    for ligne in f:
        valeurs = ligne.strip().split(";")
        temps.append(int(valeurs[0]))
        temperature.append(float(valeurs[1]))

print(temps)          # [0, 10, 20, 30]
print(temperature)    # [15.2, 16.1, 17.3, 18.0]
```

7.4.2 Écrire un fichier CSV

```
resultats = [(0, 15.2), (10, 16.1), (20, 17.3), (30, 18.0)]

with open("sortie.csv", "w") as f:
    f.write("temps;temperature\n")   # en-tete
    for t, temp in resultats:
        f.write(str(t) + ";" + str(temp) + "\n")
```

7.5 Exemple complet : analyse d'une série de mesures

On dispose d'un fichier `donnees.csv` contenant des mesures d'absorbance :

```
longueur_onde;absorbance
400;0.12
450;0.34
500;0.87
550;1.23
600;0.56
```

On veut trouver la longueur d'onde correspondant à l'absorbance maximale.

```
longueurs = []
absorbances = []

with open("donnees.csv", "r") as f:
    f.readline()   # en-tete
    for ligne in f:
        vals = ligne.strip().split(";")
        longueurs.append(int(vals[0]))
```

```
        absorbances.append(float(vals[1]))

# Trouver l'indice du maximum
i_max = 0
for i in range(len(absorbances)):
    if absorbances[i] > absorbances[i_max]:
        i_max = i

print("Absorbance maximale :", absorbances[i_max])
print("Longueur d'onde :", longueurs[i_max], "nm")
```

★ **Complément —**

Lecture avec `numpy`

Le module `numpy` propose `np.loadtxt` et `np.genfromtxt` pour lire des fichiers numériques directement sous forme de tableau :

```
import numpy as np

data = np.loadtxt("mesures.csv", delimiter=";", skiprows=1)
longueurs   = data[:, 0]    # premiere colonne
absorbances = data[:, 1]    # deuxieme colonne

print("Maximum :", np.max(absorbances))
```

Ces fonctions sont très pratiques pour les fichiers purement numériques, mais nécessitent le module `numpy`. La lecture ligne par ligne reste indispensable pour les fichiers mixtes (texte et nombres) ou de format non standard.

8

Modules

Plan du chapitre

8.1 Importer un module **45**
8.2 Le module `math` **45**
8.3 Le module `random` **46**
8.3.1 Fonctions principales 46
8.3.2 Exemple : simulation d'une expérience aléatoire 47
8.4 Le module `numpy` **47**
8.5 Le module `matplotlib.pyplot` **48**
8.5.1 Tracer sans `numpy` : avec `math` et des listes 49
8.6 Exemple synthétique : intégration graphique **50**

Python est livré avec de nombreuses **bibliothèques** (aussi appelées **modules**) qui étendent ses capacités. En sciences, quatre modules sont particulièrement importants : `math` pour les fonctions mathématiques, `random` pour les simulations aléatoires, `numpy` pour le calcul numérique sur des tableaux, et `matplotlib` pour les représentations graphiques.

8.1 Importer un module

Définition : Importation

Il existe plusieurs façons d'importer un module :

`import math`	importe le module ; accès via `math.sqrt(2)`
`from math import sqrt`	importe une seule fonction ; accès via `sqrt(2)`
`from math import *`	importe tout ; accès direct `sqrt(2)`, `pi`, ...
`import math as m`	importe avec alias ; accès via `m.sqrt(2)`

8.2 Le module `math`

Le module `math` fournit les fonctions et constantes mathématiques de base.

Propriété : Fonctions et constantes du module `math`

Constantes :

`math.pi`	$\pi \approx 3{,}14159\ldots$
`math.e`	$e \approx 2{,}71828\ldots$

Fonctions :

`math.sqrt(x)`	$\sqrt{x}$	racine carrée
`math.exp(x)`	e^x	exponentielle
`math.log(x)`	$\ln(x)$	logarithme naturel
`math.log(x, b)`	$\log_b(x)$	logarithme en base b
`math.sin(x)`	$\sin(x)$	(x en radians)
`math.cos(x)`	$\cos(x)$	
`math.tan(x)`	$\tan(x)$	
`math.floor(x)`	$\lfloor x \rfloor$	partie entière inférieure
`math.ceil(x)`	$\lceil x \rceil$	partie entière supérieure
`math.factorial(n)`	$n!$	factorielle
`math.gcd(a, b)`	$\text{PGCD}(a, b)$	

```python
from math import *

print(sqrt(2))           # 1.4142135623730951
print(pi)                # 3.141592653589793
print(exp(1))            # 2.718281828459045
print(log(exp(3)))       # 3.0
print(sin(pi / 6))       # 0.4999999999999999  (~ 0.5)
print(floor(3.7))        # 3
print(factorial(5))      # 120
```

8.3 Le module `random`

Le module `random` permet de générer des nombres aléatoires et de simuler des expériences probabilistes.

8.3.1 Fonctions principales

`random.random()`	flottant aléatoire dans $[0, 1[$
`random.uniform(a, b)`	flottant aléatoire dans $[a, b]$
`random.randint(a, b)`	entier aléatoire dans $[\![a, b]\!]$ (bornes incluses)
`random.choice(L)`	élément choisi au hasard dans la liste `L`
`random.shuffle(L)`	mélange la liste `L` **en place**
`random.sample(L, k)`	liste de `k` éléments distincts tirés de `L`

```python
import random as rd

print(rd.random())            # ex : 0.7342...  (dans [0,1[)
print(rd.randint(1, 6))       # ex : 4  (de de six faces)
print(rd.choice([10, 20, 30, 40]))   # ex : 20

L = [1, 2, 3, 4, 5]
rd.shuffle(L)
print(L)                      # ex : [3, 1, 5, 2, 4]
```

Remarque : reproductibilité avec `rd.seed`. Par défaut, chaque exécution donne des résultats différents. Pour obtenir la même séquence aléatoire (utile pour déboguer ou comparer des résultats), on fixe la **graine** (*seed*) avant les tirages :

```python
import random as rd

rd.seed(42)           # graine fixee : resultats identiques a chaque execution
print(rd.randint(1, 6))    # toujours le meme resultat
print(rd.random())         # idem
```

Remarque : Est-ce vraiment du hasard ? Un ordinateur ne sait pas générer de « vrai » hasard, seulement des **nombres pseudo-aléatoires**, calculés par un algorithme déterministe qui produit une suite *statistiquement indiscernable* du hasard. Python et `numpy` utilisent par défaut l'algorithme *Mersenne Twister*.

Concevoir et évaluer la qualité de ces générateurs constitue un véritable domaine de recherche en informatique. Un générateur médiocre peut fausser une simulation en produisant des résultats plausibles mais faux : ses valeurs, prises isolément, paraissent parfaitement bien réparties, et le défaut n'apparaît que lorsqu'on les utilise par couples ou par triplets. Question distincte, un générateur peut aussi être suffisamment prévisible pour qu'un adversaire devine les tirages suivants ; c'est pourquoi il existe des générateurs séparés, dits *cryptographiquement sûrs*, réservés aux usages sensibles (jeux d'argent en ligne, clés de chiffrement).

Dans tout ce cours, nous ferons *comme si* `rd.random()` renvoyait un réel tiré uniformément dans $[0, 1[$, et surtout *comme si* les appels successifs étaient indépendants : connaître les valeurs déjà obtenues ne donne aucune information sur la suivante. C'est cette seconde hypothèse qui autorise tous nos raisonnements probabilistes, puisqu'elle permet de traiter une suite d'appels à `rd.random()` comme une suite de variables aléatoires indépendantes.

8.3.2 Exemple : simulation d'une expérience aléatoire

Estimer la probabilité d'obtenir au moins un 6 en lançant deux dés :

```python
import random as rd

def simul_deux_des():
    """Renvoie True si au moins un des deux des vaut 6."""
    d1 = rd.randint(1, 6)
    d2 = rd.randint(1, 6)
    return d1 == 6 or d2 == 6

N = 100000
favorables = 0
for k in range(N):
    if simul_deux_des():
        favorables = favorables + 1

print("Probabilite estimee :", favorables / N)
# Valeur theorique : 1 - (5/6)^2 = 11/36 ~ 0.306
```

8.4 Le module numpy

`numpy` (alias `np`) est la bibliothèque centrale du calcul numérique en Python. Elle introduit le type **tableau** (`array`), qui permet d'effectuer des opérations mathématiques sur des listes entières de nombres, de façon efficace.

Définition : Tableaux `numpy`

Un tableau `numpy` est une séquence de nombres du même type, sur laquelle les opérations arithmétiques s'appliquent **élément par élément**.

```
import numpy as np

a = np.array([1, 2, 3, 4, 5])
print(a * 2)       # [2  4  6  8  10]
print(a ** 2)      # [ 1  4  9 16 25]
print(a + 10)      # [11 12 13 14 15]
```

Ce comportement est différent des listes Python, où `[1,2,3] * 2` donne `[1,2,3,1,2,3]`.

Propriété : Création de tableaux

`np.array([a, b, c])`	tableau à partir d'une liste
`np.linspace(a, b, n)`	n valeurs régulièrement espacées entre a et b
`np.zeros(n)`	tableau de n zéros
`np.ones(n)`	tableau de n uns
`np.arange(a, b, pas)`	valeurs de a à b (exclu) par pas

```
x = np.linspace(0, 1, 5)     # [0.   0.25  0.5   0.75  1.  ]
z = np.zeros(4)               # [0. 0. 0. 0.]
```

Propriété : Fonctions mathématiques sur un tableau

`numpy` fournit des versions vectorisées des fonctions mathématiques, qui s'appliquent à chaque élément d'un tableau :

`np.sqrt(a)`, `np.exp(a)`, `np.log(a)`	fonctions usuelles
`np.sin(a)`, `np.cos(a)`, `np.tan(a)`	fonctions trigonométriques
`np.sum(a)`, `np.min(a)`, `np.max(a)`	réductions
`np.mean(a)`	moyenne

```
x = np.linspace(0, np.pi, 5)
print(np.sin(x))    # [ 0.   0.707  1.   0.707  0. ]  (approx)
```

Remarque : les fonctions de `math` (`math.sin`, etc.) ne s'appliquent qu'à un seul nombre. Pour traiter un tableau, il faut utiliser les versions `numpy`.

8.5 Le module `matplotlib.pyplot`

`matplotlib.pyplot` (alias `plt`) permet de tracer des courbes et des graphiques. C'est l'outil standard de visualisation en Python scientifique.

Définition : Tracer une courbe

Le principe est toujours le même :

① Créer des tableaux de valeurs `x` et `y`.
② Appeler `plt.plot(x, y)` pour tracer la courbe.
③ Appeler `plt.show()` pour afficher la fenêtre graphique.

Exemple minimal : tracer $y = \sin(x)$ sur $[0, 2\pi]$.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 200)
y = np.sin(x)

plt.plot(x, y)
plt.show()
```

Propriété : Personnalisation du graphique

`plt.xlabel("texte")`	légende de l'axe des abscisses
`plt.ylabel("texte")`	légende de l'axe des ordonnées
`plt.title("texte")`	titre du graphique
`plt.grid()`	ajoute une grille
`plt.legend()`	affiche la légende (nécessite `label=` dans `plot`)
`plt.axis([a,b,c,d])`	fixe les limites des axes

Le troisième argument de `plot` fixe couleur et style :

`"b"`	bleu	`"r"` rouge, `"g"` vert, `"k"` noir	
`"-"`	trait	`"--"` tirets, `"o"` points, `"o-"` points reliés	

Exemple complet : tracer $\sin(x)$ et $\cos(x)$ sur $[0, 2\pi]$.

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 300)

plt.plot(x, np.sin(x), "b-", label="sin(x)")
plt.plot(x, np.cos(x), "r--", label="cos(x)")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Fonctions trigonometriques")
plt.legend()
plt.grid()
plt.show()
```

8.5.1 Tracer sans `numpy` : avec `math` et des listes

Lorsque `numpy` n'est pas disponible, on peut construire les valeurs à la main avec une boucle :

```python
import matplotlib.pyplot as plt

def f(x):
    """ Renvoie la valeur de f(x)"""
    ...
    return ...

# Tracer de la représe'ntation de f sur [a,b]

n = 200
a, b = ..., ...
x = [ a + k*(b-a)/n for k in range(n + 1)]
y = [f(xi) for xi in x]

plt.plot(x, y)
plt.show()
```

Exemple

```
from math import sin, pi
import matplotlib.pyplot as plt

def f(x):
    if x < pi:
        return sin(x)
    return - sin(x)


n = 200
x = [2 * pi * k / n for k in range(n + 1)]
y = [f(xi) for xi in x]

plt.plot(x, y)
plt.show()
```

Remarque : cette approche produit le même résultat qu'avec `numpy` mais elle permet de s'adapter à des situations plus délicates où on ne représente pas des fonctions usuelles.

8.6 Exemple synthétique : intégration graphique

On trace une fonction et son approximation par la méthode des rectangles (qui sera présentée au chapitre 19) :

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2

# Courbe exacte
x = np.linspace(0, 2, 200)
plt.plot(x, f(x), "b-", label="f(x) = x^2")

# Rectangles (n = 6 sous-intervalles)
n = 6
a, b = 0, 2
h = (b - a) / n
for k in range(n):
    xk = a + k * h
    plt.bar(xk, f(xk), width=h, align="edge",
            edgecolor="k", facecolor="lightblue", alpha=0.5)

plt.xlabel("x")
plt.title("Methode des rectangles a gauche, n = 6")
plt.legend()
plt.grid()
plt.show()
```

★ **Complément —**

Tableaux 2D avec `numpy`

`numpy` permet aussi de manipuler des matrices (tableaux à deux dimensions) :

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(A + B)           # addition terme a terme
print(A @ B)           # produit matriciel
print(np.transpose(A)) # transposee

print(np.linalg.det(A))     # determinant
print(np.linalg.inv(A))     # inverse
print(np.linalg.eig(A))     # valeurs et vecteurs propres
```

Ces fonctions sont utilisées dans les exercices de l'oral du concours Agro. L'annexe (chapitre 22) approfondit ce sujet : distinction entre `A * B` et `A @ B`, extraction de lignes/colonnes par *slicing*, et usage du sous-module `numpy.random` pour les calculs d'incertitude-type en physique.

9

Fonctions récursives

Plan du chapitre

9.1 Principe de la récursion **53**
9.1.1 Déroulement des appels 54
9.2 Exemples classiques **54**
9.2.1 Suite de Fibonacci 54
9.2.2 PGCD (algorithme d'Euclide) 54
9.2.3 Exponentiation rapide 55
9.2.4 Somme récursive d'une liste 55
9.3 Récursion et itération **55**
9.4 Condition de terminaison **55**
9.5 Application : processus de branchement **56**
9.5.1 Modèle 56
9.5.2 Terminaison probabiliste 56

Une fonction est dite **récursive** lorsqu'elle s'appelle elle-même dans son propre corps. C'est une technique puissante qui traduit directement les définitions mathématiques par récurrence, et qui s'avère naturelle pour de nombreux algorithmes.

9.1 Principe de la récursion

Définition : Fonction récursive

Une fonction récursive comporte obligatoirement deux parties :

- Un **cas de base** (ou condition d'arrêt) : une situation simple qui se traite directement, sans appel récursif.
- Un **appel récursif** : la fonction se rappelle elle-même sur un problème **strictement plus petit**, qui se rapproche du cas de base.

Sans cas de base, la récursion ne s'arrête jamais (analogue à la boucle infinie).

Exemple : la définition mathématique de la factorielle est $0! = 1$ et $n! = n \times (n-1)!$ pour $n \geqslant 1$. On la traduit directement :

```python
def factorielle(n):
    """Renvoie n! pour un entier n >= 0."""
    if n == 0:            # cas de base
        return 1
    return n * factorielle(n - 1)

print(factorielle(5))    # 120
```

9.1.1 Déroulement des appels

Pour comprendre comment Python exécute un appel récursif, on peut tracer l'arbre des appels. Par exemple, `factorielle(4)` se déroule ainsi :

```
factorielle(4)
  = 4 * factorielle(3)
  = 4 * (3 * factorielle(2))
  = 4 * (3 * (2 * factorielle(1)))
  = 4 * (3 * (2 * (1 * factorielle(0))))
  = 4 * (3 * (2 * (1 * 1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Remarque : Python empile les appels en mémoire jusqu'à atteindre le cas de base, puis remonte en calculant. Cela consomme de la mémoire proportionnellement à la profondeur des appels. Python impose une limite (généralement 1000 appels imbriqués) au-delà de laquelle une erreur `RecursionError` est déclenchée.

9.2 Exemples classiques

9.2.1 Suite de Fibonacci

La suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$.

```python
def fibonacci(n):
    """Renvoie le n-ieme terme de la suite de Fibonacci (n >= 0)."""
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))    # 55
```

Attention : cette version est **très inefficace** car elle recalcule de nombreuses fois les mêmes valeurs. Par exemple, `fibonacci(5)` provoque deux appels à `fibonacci(3)`, qui provoquent chacun deux appels à `fibonacci(2)`, etc. La version itérative est ici nettement préférable :

```python
def fibonacci_iter(n):
    """Version iterative : calcule F_n sans recalculs inutiles."""
    if n == 0:
        return 0
    a, b = 0, 1
    for i in range(n - 1):
        a, b = b, a + b
    return b
```

9.2.2 PGCD (algorithme d'Euclide)

L'algorithme d'Euclide se prête naturellement à une formulation récursive :

$$\mathrm{PGCD}(a, 0) = a \qquad \text{et} \qquad \mathrm{PGCD}(a, b) = \mathrm{PGCD}(b,\ a \bmod b) \quad \text{pour } b > 0$$

```python
def pgcd(a, b):
    """Renvoie le PGCD de a et b (a, b entiers positifs)."""
    if b == 0:                # cas de base
        return a
    return pgcd(b, a % b)     # appel recursif

print(pgcd(48, 18))    # 6
```

9.2.3 Exponentiation rapide

Calculer x^n naïvement nécessite $n-1$ multiplications. L'algorithme suivant utilise la remarque :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ (x^{n/2})^2 & \text{si } n \text{ est pair} \\ x \times x^{n-1} & \text{si } n \text{ est impair} \end{cases}$$

```
def puissance(x, n):
    """Renvoie x**n pour un entier n >= 0."""
    if n == 0:
        return 1
    if n % 2 == 0:
        moitie = puissance(x, n // 2)
        return moitie * moitie
    return x * puissance(x, n - 1)

print(puissance(2, 10))    # 1024
```

Intérêt : au lieu de n multiplications, on n'en effectue que de l'ordre de $\log_2 n$. Pour $n = 10^6$, cela représente environ 20 multiplications au lieu d'un million.

9.2.4 Somme récursive d'une liste

```
def somme(L):
    """Renvoie la somme des elements de la liste L."""
    if len(L) == 0:          # cas de base : liste vide
        return 0
    return L[0] + somme(L[1:])    # premier + somme du reste
```

9.3 Récursion et itération

Tout algorithme récursif peut être réécrit de façon itérative, et vice versa. Le choix dépend de la lisibilité et de l'efficacité.

	Récursion	**Itération**
Lisibilité	proche de la définition math.	parfois plus explicite
Mémoire	pile d'appels (limitée)	pas de surcoût
Risque	`RecursionError`	boucle infinie
Usage naturel	arbres, diviser pour régner	boucles simples

Règle pratique : préférer l'itération pour les suites simples (factorielle, Fibonacci) ; la récursion est naturelle quand le problème se découpe en sous-problèmes de même nature (recherche dichotomique, tris avancés).

9.4 Condition de terminaison

Propriété : Terminaison d'une fonction récursive

Pour garantir qu'une fonction récursive termine, il faut montrer qu'il existe une **quantité entière positive** qui **décroît strictement** à chaque appel récursif et qui atteint le cas de base.

Pour `factorielle(n)` : la quantité est n, qui décroît de 1 à chaque appel, et le cas de base est $n = 0$.
Pour `pgcd(a, b)` : la quantité est b, qui décroît strictement (le reste a mod b est strictement inférieur à b), et le cas de base est $b = 0$.

9.5 Application : processus de branchement

9.5.1 Modèle

On modélise une **lignée génétique** : à chaque génération, une lignée s'éteint avec probabilité $\frac{1}{2}$ (aucun descendant), ou se divise en deux lignées indépendantes avec probabilité $\frac{1}{2}$ (deux descendants). On cherche à simuler le **temps d'extinction**, c'est-à-dire la génération à laquelle la totalité des descendants disparaît.

```python
import random as rd

def temps_extinction():
    """Renvoie le temps d'extinction d'une lignee.
    A chaque generation : extinction (proba 1/2) ou deux descendants (proba 1/2)
        ."""
    if rd.random() < 0.5:
        return 1                          # extinction des la generation suivante
    t1 = temps_extinction()               # temps d'extinction de la premiere lignee
    t2 = temps_extinction()               # temps d'extinction de la seconde lignee
    return 1 + max(t1, t2)
```

Lecture de la récursion :

• Si `rd.random() < 0.5` : la lignée ne se propage pas — on renvoie `1` (la génération courante est la dernière).

• Sinon : deux lignées indépendantes se développent. Le temps d'extinction global est $1 + \max(t_1, t_2)$: on attend que la plus longue des deux sous-lignées s'éteigne.

L'arbre des appels récursifs est exactement **l'arbre généalogique** de la population : chaque appel correspond à un individu, ses deux appels récursifs à ses deux descendants.

9.5.2 Terminaison probabiliste

Cette fonction ne vérifie pas le critère habituel de terminaison (pas de variant entier décroissant) : à chaque appel, la fonction peut encore se ramifier. Pourtant, elle se termine **presque sûrement** : la probabilité que la lignée survive indéfiniment est nulle (résultat classique des processus de branchement).

Estimation par simulation :

```python
N = 10000
resultats = [temps_extinction() for _ in range(N)]
print(f"Temps moyen d'extinction : {sum(resultats) / N:.2f}")
print(f"Temps maximal observe  : {max(resultats)}")
```

★ **Complément** —

Récursion croisée et mémoïsation

Deux fonctions peuvent s'appeler mutuellement (**récursion croisée**). Par exemple, pour tester la parité :

```
def est_pair(n):
    if n == 0:
        return True
    return est_impair(n - 1)

def est_impair(n):
    if n == 0:
        return False
    return est_pair(n - 1)
```

Pour remédier à l'inefficacité de `fibonacci` récursif, on peut mémoriser les résultats déjà calculés (**mémoïsation**) :

```
memo = {}

def fib_memo(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n - 1) + fib_memo(n - 2)
    return memo[n]
```

Cette technique (aussi appelée *programmation dynamique*) rend l'algorithme linéaire en temps.

10

Recherche dichotomique

Plan du chapitre

10.1 Recherche dans une liste triée **59**
10.1.1 Recherche linéaire : rappel 59
10.1.2 Principe de la dichotomie 59
10.1.3 Version itérative 60
10.1.4 Version récursive 60
10.1.5 Comparaison des performances 60
10.2 Résolution approchée d'une équation **60**
10.2.1 Exemple : recherche d'un zéro sur plusieurs intervalles 62

La **recherche dichotomique** (ou recherche par bissection) est un algorithme qui permet de trouver efficacement un élément dans une liste **triée**, ou une valeur particulière d'une fonction monotone. Son principe est de diviser par deux l'espace de recherche à chaque étape.

10.1 Recherche dans une liste triée

10.1.1 Recherche linéaire : rappel

Pour trouver si une valeur `v` est dans une liste quelconque, on la parcourt entièrement :

```
def recherche_lineaire(L, v):
    """Renvoie True si v est dans la liste L."""
    for x in L:
        if x == v:
            return True
    return False
```

Dans le pire cas, on examine tous les n éléments. Si la liste est **triée**, on peut faire beaucoup mieux.
Dans la suite de ce paragraphe la liste `L` est triée.

10.1.2 Principe de la dichotomie

Propriété : Recherche dichotomique

On maintient deux indices `g` (gauche) et `d` (droite) encadrant la zone de recherche. À chaque étape :
① On calcule l'indice du milieu : `m = (g + d) // 2`.
② On compare `L[m]` à la valeur cherchée `v` :
- si `L[m] == v` : trouvé, on s'arrête ;
- si `L[m] < v` : `v` est dans la moitié droite, on pose `g = m + 1` ;
- si `L[m] > v` : `v` est dans la moitié gauche, on pose `d = m - 1`.

On s'arrête dès que `v` est trouvé ou que `g > d` (zone vide : `v` absent).
À chaque étape, la taille de la zone de recherche est **divisée par 2** : on effectue au plus $\lfloor \log_2 n \rfloor + 1$ comparaisons pour une liste de n éléments.

10.1.3 Version itérative

```python
def recherche_dicho(L, v):
    """Renvoie True si v est dans la liste triee L, False sinon."""
    g = 0
    d = len(L) - 1
    while g <= d:
        m = (g + d) // 2
        if L[m] == v:
            return True
        if L[m] < v:
            g = m + 1
        else:
            d = m - 1
    return False

L = [1, 3, 5, 7, 9, 11, 13, 15]
print(recherche_dicho(L, 7))     # True
print(recherche_dicho(L, 6))     # False
```

Remarque : la dichotomie exige que la liste soit triée. Sur une liste non triée, le résultat est incorrect.

10.1.4 Version récursive

```python
def recherche_dicho_rec(L, v, g, d):
    """Renvoie True si v est dans L[g:d+1] (liste triee)."""
    if g > d:                # zone vide : v absent
        return False
    m = (g + d) // 2
    if L[m] == v:
        return True
    if L[m] < v:
        return recherche_dicho_rec(L, v, m + 1, d)
    return recherche_dicho_rec(L, v, g, m - 1)

L = [1, 3, 5, 7, 9, 11, 13, 15]
print(recherche_dicho_rec(L, 7, 0, len(L) - 1))    # True
```

10.1.5 Comparaison des performances

Pour une liste de n éléments :

	Recherche linéaire	**Recherche dichotomique**
Nombre de comparaisons (pire cas)	au plus n	au plus $\lceil \log_2 n \rceil$
Condition	liste quelconque	liste **triée**

Pour $n = 10^6$: la recherche linéaire examine jusqu'à 1 000 000 éléments, la dichotomie au plus 20.

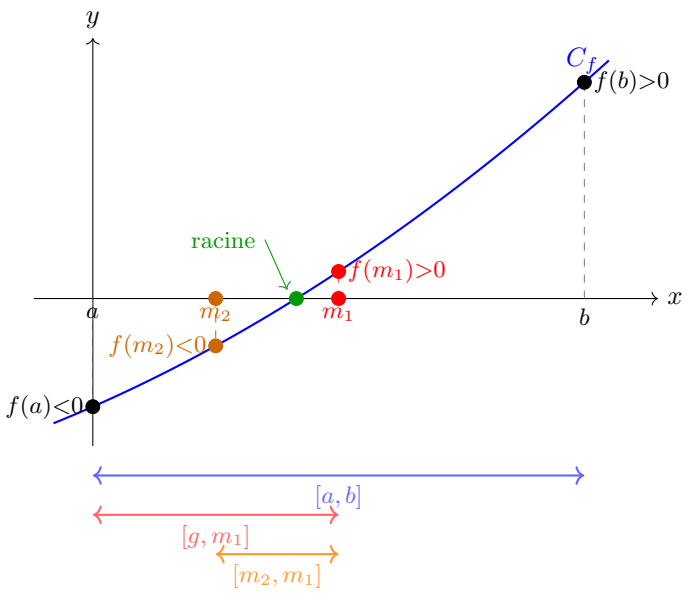
Remarque (hors programme) : en informatique, on résume ce constat par la notation $O(\cdot)$: on dit que la recherche linéaire est en $O(n)$ et la dichotomie en $O(\log n)$, ce qui signifie que le nombre d'opérations croît respectivement proportionnellement à n et à $\log n$. Cette notation n'est pas exigible aux concours, mais elle apparaît dans la littérature et dans les tableaux comparatifs — on la lit comme un ordre de grandeur du coût.

10.2 Résolution approchée d'une équation

La dichotomie s'applique aussi pour trouver une solution approchée de $f(x) = 0$ sur un intervalle $[a, b]$, lorsque f est continue et que $f(a)$ et $f(b)$ sont de signes opposés (théorème des valeurs intermédiaires).

Propriété : Méthode de dichotomie pour $f(x) = 0$

On dispose d'une fonction f continue sur $[a, b]$ avec $f(a) \cdot f(b) < 0$.
On maintient un encadrement $[g, d]$ de la racine. À chaque étape :
① On calcule le milieu $m = \dfrac{g+d}{2}$.
② Si $f(g) \cdot f(m) < 0$: la racine est dans $[g, m]$, on pose $d = m$.
③ Sinon : la racine est dans $[m, d]$, on pose $g = m$.
On s'arrête lorsque $d - g < \varepsilon$ (précision souhaitée). La racine approchée est alors $\dfrac{g+d}{2}$.
À chaque étape, la longueur de l'intervalle est **divisée par 2** : après n étapes, l'erreur est inférieure à $\dfrac{b-a}{2^n}$.



Exemple : trouver une valeur approchée de $\sqrt{2}$, racine de $f(x) = x^2 - 2$ sur $[1, 2]$.

```python
def f(x):
    return x**2 - 2

def dichotomie(f, a, b, eps):
    """Renvoie une valeur approchee a eps/2 pres de la racine de f sur [a,b].
    Condition : f(a) et f(b) de signes opposes."""
    g, d = a, b
    while d - g > eps:
        m = (g + d) / 2
        if f(g) * f(m) < 0:
            d = m
        else:
            g = m
    return (g + d) / 2

racine = dichotomie(f, 1, 2, 1e-6)
print(racine)          # 1.4142136...  (valeur approchee de sqrt(2))
```

Remarque : la boucle s'arrête dès que $d - g \leqslant \varepsilon$, donc la valeur renvoyée $\frac{g+d}{2}$ est à au plus $\frac{\varepsilon}{2}$ de la racine. Le nombre d'itérations nécessaires est $\left\lceil \log_2\left(\dfrac{b-a}{\varepsilon}\right) \right\rceil$. Pour $b - a = 1$ et $\varepsilon = 10^{-6}$, cela donne environ 20 itérations.

10.2.1 Exemple : recherche d'un zéro sur plusieurs intervalles

Pour trouver toutes les racines de $f(x) = x^3 - 3x + 1$ sur $[-2, 2]$, on commence par localiser les changements de signe, puis on affine par dichotomie :

```
def f(x):
    return x**3 - 3*x + 1

def dichotomie(f, a, b, eps):
    g, d = a, b
    while d - g > eps:
        m = (g + d) / 2
        if f(g) * f(m) < 0:
            d = m
        else:
            g = m
    return (g + d) / 2

# Localisation des changements de signe par pas de 0.1
n = 40
pas = 4 / n
racines = []
for k in range(n):
    a = -2 + k * pas
    b = a + pas
    if f(a) * f(b) < 0:
        racines.append(dichotomie(f, a, b, 1e-9))

print(racines)   # trois racines approchees
```

11

Algorithmes de tri

Plan du chapitre

11.1 Tri par sélection **63**
11.1.1 Principe 63
11.1.2 Mise en œuvre 63
11.2 Tri par insertion **64**
11.2.1 Principe 64
11.2.2 Mise en œuvre 64
11.3 Tri par comptage **65**
11.3.1 Principe 65
11.3.2 Mise en œuvre 65
11.4 Comparaison des algorithmes **65**

Trier une liste, c'est réorganiser ses éléments dans un ordre croissant (ou décroissant). Le tri est une opération fondamentale en informatique : il conditionne l'efficacité de nombreux algorithmes, dont la recherche dichotomique. Python dispose de la fonction native `sorted(L)` (qui renvoie une nouvelle liste triée) et de la méthode `L.sort()` (qui trie `L` en place). Ce chapitre présente comment construire soi-même des algorithmes de tri.

Définition : Tri en place

Un tri est dit **en place** lorsqu'il réorganise les éléments de la liste `L` sans créer de liste auxiliaire de taille comparable. La liste est modifiée directement.

L'opération d'échange de deux éléments s'écrit en Python :

```
L[i], L[j] = L[j], L[i]
```

11.1 Tri par sélection

11.1.1 Principe

Pour trier `L = [L[0], ..., L[n-1]]` :

— Pour k allant de 0 à $n-2$: trouver l'indice du minimum de `L[k:]`, puis l'échanger avec `L[k]`.

Après l'étape k, les $k+1$ premiers éléments sont à leur place définitive.

11.1.2 Mise en œuvre

```
def indice_min(L, k):
    """Renvoie l'indice du minimum de L[k:]."""
    i_min = k
    for i in range(k + 1, len(L)):
```

```
        if L[i] < L[i_min]:
            i_min = i
    return i_min

def tri_selection(L):
    """Trie la liste L en place par selection."""
    n = len(L)
    for k in range(n - 1):
        i_min = indice_min(L, k)
        L[k], L[i_min] = L[i_min], L[k]

L = [5, 3, 8, 1, 9, 2]
tri_selection(L)
print(L)    # [1, 2, 3, 5, 8, 9]
```

Déroulement sur `[5, 3, 8, 1, 9, 2]` :

$k=0$	[**1**, 3, 8, 5, 9, 2]	(échange 5 et 1)
$k=1$	[1, **2**, 8, 5, 9, 3]	(échange 3 et 2)
$k=2$	[1, 2, **3**, 5, 9, 8]	(échange 8 et 3)
$k=3$	[1, 2, 3, **5**, 9, 8]	(déjà en place)
$k=4$	[1, 2, 3, 5, **8**, 9]	(échange 9 et 8)

11.2 Tri par insertion

11.2.1 Principe

Pour trier `L = [L[0], ..., L[n-1]]` :

— Pour j allant de 1 à $n-1$: insérer `L[j]` à sa place dans la sous-liste `L[0:j]` déjà triée.

C'est le tri qu'on effectue naturellement lorsqu'on range des cartes dans sa main.

11.2.2 Mise en œuvre

```
def insere(L, j):
    """Insere L[j] dans L[0:j] supposee triee, de sorte que L[0:j+1] soit triee.
    """
    val = L[j]
    i = j
    while i > 0 and L[i - 1] > val:
        L[i] = L[i - 1]
        i = i - 1
    L[i] = val

def tri_insertion(L):
    """Trie la liste L en place par insertion."""
    for j in range(1, len(L)):
        insere(L, j)

L = [5, 3, 8, 1, 9, 2]
tri_insertion(L)
print(L)    # [1, 2, 3, 5, 8, 9]
```

Déroulement sur `[5, 3, 8, 1, 9, 2]` :

$j=1$	[**3, 5**, 8, 1, 9, 2]	(insertion de 3)
$j=2$	[**3, 5, 8**, 1, 9, 2]	(8 déjà en place)
$j=3$	[**1, 3, 5, 8**, 9, 2]	(insertion de 1)
$j=4$	[**1, 3, 5, 8, 9**, 2]	(9 déjà en place)
$j=5$	[**1, 2, 3, 5, 8, 9**]	(insertion de 2)

11.3 Tri par comptage

11.3.1 Principe

Le tri par comptage s'applique lorsque les valeurs de `L` sont des entiers dans un intervalle connu $[\![0, k-1]\!]$. Il ne procède pas par comparaisons mais par dénombrement :

1. Compter combien de fois chaque valeur de $[\![0, k-1]\!]$ apparaît dans `L`.
2. Reconstruire la liste triée en plaçant chaque valeur autant de fois que nécessaire.

11.3.2 Mise en œuvre

```
def occurrences(L, k):
    """Renvoie la liste des occurrences des entiers 0, 1, ..., k-1 dans L."""
    occ = [0] * k
    for x in L:
        occ[x] = occ[x] + 1
    return occ

def tri_comptage(L, k):
    """Trie en place la liste L dont les valeurs sont dans [[0, k-1]]."""
    occ = occurrences(L, k)
    i = 0
    for v in range(k):
        for c in range(occ[v]):
            L[i] = v
            i = i + 1

L = [3, 1, 4, 1, 5, 2, 3, 2]
tri_comptage(L, 6)
print(L)    # [1, 1, 2, 2, 3, 3, 4, 5]
```

Exemple de déroulement pour `L = [3, 1, 4, 1, 5, 2, 3, 2]`, `k = 6` :

Occurrences : `occ = [0, 2, 2, 2, 1, 1]`
(0 apparaît 0 fois, 1 apparaît 2 fois, 2 apparaît 2 fois, etc.)
Reconstruction : `1, 1, 2, 2, 3, 3, 4, 5`

11.4 Comparaison des algorithmes

	Sélection	Insertion	Comptage
Comparaisons (pire cas)	$\approx n^2/2$	$\approx n^2/2$	aucune
En place	oui	oui	oui (mém. aux. k cases)
Condition	aucune	aucune	valeurs entières bornées
Cas favorable	aucun	liste presque triée	valeurs dans $[\![0, k-1]\!]$, k petit

Remarque : le tri par insertion est particulièrement efficace lorsque la liste est déjà presque triée. En pratique, `sorted()` et `L.sort()` utilisent un algorithme hybride (Timsort) dont le nombre de comparaisons croît comme $n \log n$ — bien plus rapide que $n^2/2$ pour les grandes listes (notation hors programme : $O(n \log n)$).

★ **Complément** —

Tri rapide (*quicksort*)

Le tri rapide est un algorithme récursif en $O(n \log n)$ en moyenne. Son principe est de choisir un **pivot**, de réorganiser la liste pour placer le pivot à sa position définitive (tous les éléments plus petits à gauche, tous les plus grands à droite), puis de trier récursivement les deux parties.

```
def segmente(L, i, j):
    """Place L[j-1] (pivot) a sa position definitive dans L[i:j].
    Renvoie l'indice final du pivot."""
    pivot = L[j - 1]
    place = i
    for k in range(i, j - 1):
        if L[k] < pivot:
            L[place], L[k] = L[k], L[place]
            place = place + 1
    L[place], L[j - 1] = L[j - 1], L[place]
    return place

def tri_rapide(L, i, j):
    """Trie L[i:j] en place par tri rapide."""
    if j - i <= 1:
        return
    place = segmente(L, i, j)
    tri_rapide(L, i, place)
    tri_rapide(L, place + 1, j)

L = [5, 3, 8, 1, 9, 2]
tri_rapide(L, 0, len(L))
print(L)    # [1, 2, 3, 5, 8, 9]
```

Dans le pire cas (liste déjà triée avec ce choix de pivot), la complexité est $O(n^2)$. En pratique, le tri rapide est l'un des algorithmes les plus utilisés.

12

Tris avancés

Plan du chapitre

12.1 Limites des tris élémentaires **67**
12.2 Tri fusion (*mergesort*) **67**
12.2.1 Principe 67
12.2.2 Fusion de deux listes triées 68
12.2.3 Tri fusion récursif 68
12.2.4 Déroulement sur un exemple 68
12.2.5 Complexité 68
12.3 Comparaison des algorithmes de tri **69**

★ **Ce chapitre est entièrement hors programme des concours.**
Il présente des algorithmes de tri plus efficaces que les tris élémentaires du chapitre précédent, et permet de comprendre pourquoi Python utilise un algorithme sophistiqué pour `sorted()` et `L.sort()`.

12.1 Limites des tris élémentaires

Les tris par sélection et par insertion effectuent de l'ordre de $\frac{n(n-1)}{2}$ comparaisons pour une liste de n éléments, soit une complexité $O(n^2)$. Pour $n = 10^6$ éléments, cela représente environ 5×10^{11} opérations — bien trop lent en pratique.

On peut montrer qu'un algorithme de tri par comparaisons ne peut pas faire mieux que $O(n \log n)$ comparaisons dans le pire cas. Le tri fusion atteint cette borne.

12.2 Tri fusion (*mergesort*)

12.2.1 Principe

Le tri fusion est un algorithme récursif fondé sur le paradigme **diviser pour régner** :

1. **Diviser :** couper la liste en deux moitiés de taille égale.
2. **Régner :** trier récursivement chaque moitié.
3. **Combiner :** fusionner les deux moitiés triées en une seule liste triée.

L'opération clé est la **fusion** de deux listes déjà triées : on compare les premières valeurs de chaque liste et on prend la plus petite, en avançant dans la liste correspondante.

12.2.2 Fusion de deux listes triées

```python
def fusion(L1, L2):
    """Fusionne deux listes triees, renvoie une nouvelle liste triee."""
    res = []
    i, j = 0, 0
    while i < len(L1) and j < len(L2):
        if L1[i] <= L2[j]:
            res.append(L1[i])
            i = i + 1
        else:
            res.append(L2[j])
            j = j + 1
    return res + L1[i:] + L2[j:]

print(fusion([1, 3, 5], [2, 4, 6]))    # [1, 2, 3, 4, 5, 6]
```

La fusion de deux listes de longueurs p et q s'effectue en $O(p+q)$ comparaisons.

12.2.3 Tri fusion récursif

```python
def tri_fusion(L):
    """Renvoie une nouvelle liste triee contenant les elements de L."""
    n = len(L)
    if n <= 1:
        return L[:]
    m = n // 2
    gauche = tri_fusion(L[:m])
    droite = tri_fusion(L[m:])
    return fusion(gauche, droite)

L = [5, 3, 8, 1, 9, 2]
print(tri_fusion(L))    # [1, 2, 3, 5, 8, 9]
print(L)                # [5, 3, 8, 1, 9, 2]  (L non modifiee)
```

Remarque : contrairement aux tris par sélection et insertion, `tri_fusion` **ne trie pas en place** : il renvoie une nouvelle liste et ne modifie pas `L`.

12.2.4 Déroulement sur un exemple

Entrée : `[5, 3, 8, 1]`

Division	`[5, 3]` et `[8, 1]`
Division	`[5]` et `[3]`, puis `[8]` et `[1]`
Fusion	`[5]` + `[3]` → `[3, 5]`
Fusion	`[8]` + `[1]` → `[1, 8]`
Fusion finale	`[3, 5]` + `[1, 8]` → `[1, 3, 5, 8]`

12.2.5 Complexité

À chaque niveau de récursion, on traite n éléments au total (pour les fusions). Il y a $\lceil \log_2 n \rceil$ niveaux de récursion. La complexité totale est donc $O(n \log n)$.

12.3 Comparaison des algorithmes de tri

	Sélection	**Insertion**	**Fusion**	**Rapide**
Complexité (moyen)	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Complexité (pire)	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
En place	oui	oui	non	oui
Stable	non	oui	oui	non

Un tri est dit **stable** s'il préserve l'ordre relatif des éléments égaux.

En pratique : Python utilise **Timsort**, un algorithme hybride combinant le tri par insertion (sur les petites séquences) et le tri fusion (pour les grandes), avec une complexité garantie en $O(n \log n)$.

13

Mutabilité et effets de bord

Plan du chapitre

13.1 Types mutables et immuables 71
13.2 Aliasing 72
13.2.1 Deux variables, un seul objet 72
13.2.2 Copier une liste 72
13.3 Effets de bord dans les fonctions 73
13.3.1 Passage par objet 73
13.3.2 Exemple : erreur classique 73
13.4 Conséquences pratiques 73

★ Ce chapitre est aux bords du programme des concours. La notion de mutabilité peut apparaître en particulier à l'oral.

En Python, toutes les valeurs sont des **objets**. Une variable n'est pas une case mémoire contenant directement une valeur, mais une **étiquette** qui pointe vers un objet. Cette distinction, invisible pour les types simples, devient cruciale avec les listes.

13.1 Types mutables et immuables

Définition : Mutabilité

Un objet est dit **immuable** si son contenu ne peut pas être modifié après sa création. Il est dit **mutable** s'il peut être modifié en place.

Types immuables `int`, `float`, `bool`, `str`, `tuple`
Types mutables `list`, `dict`, `set`

Pour les types immuables, toute opération qui semble « modifier » la valeur crée en réalité un **nouvel objet** :

```
s = "bonjour"
s = s + " monde"    # cree un nouvel objet str, s pointe dessus
n = 3
n = n + 1           # cree un nouvel objet int (4), n pointe dessus
```

Pour les listes (type mutable), les opérations comme `append`, `sort`, ou l'affectation par indice `L[i] = ...` modifient l'objet existant sans en créer un nouveau.

13.2 Aliasing

13.2.1 Deux variables, un seul objet

Lorsqu'on écrit `L2 = L1`, on ne copie pas la liste : les deux variables `L1` et `L2` pointent vers le **même objet**. Modifier l'un modifie l'autre.

```
L1 = [1, 2, 3]
L2 = L1           # L2 et L1 designent le meme objet

L2[0] = 99
print(L1)         # [99, 2, 3]  -- L1 est modifiee !
print(L2)         # [99, 2, 3]
```

Propriété : Aliasing

Lorsque deux variables désignent le même objet mutable, toute modification via l'une est visible depuis l'autre. On parle d'**alias** (ou **aliasing**).

Pour tester si deux variables désignent le même objet, on utilise `is` (et non `==`) :

```
L1 = [1, 2, 3]
L2 = L1
L3 = [1, 2, 3]    # meme contenu, objet different

print(L1 is L2)    # True  -- meme objet
print(L1 is L3)    # False -- objets distincts
print(L1 == L3)    # True  -- meme contenu
```

13.2.2 Copier une liste

Pour obtenir une copie indépendante d'une liste, on utilise le **slicing** ou la fonction `list` :

```
L1 = [1, 2, 3]
L2 = L1[:]         # copie superficielle
L3 = list(L1)      # copie superficielle

L2[0] = 99
print(L1)    # [1, 2, 3]  -- L1 n'est pas modifiee
print(L2)    # [99, 2, 3]
```

Attention : listes de listes

`L1[:]` et `list(L1)` effectuent une **copie superficielle** : les éléments eux-mêmes ne sont pas copiés. Pour une liste de listes, les sous-listes restent partagées :

```
M1 = [[1, 2], [3, 4]]
M2 = M1[:]           # copie superficielle

M2[0][0] = 99
print(M1)    # [[99, 2], [3, 4]]  -- M1 modifiee !
```

Pour une copie complète (indépendante à tous les niveaux), utiliser `copy.deepcopy` :

```
import copy
M2 = copy.deepcopy(M1)
```

13.3 Effets de bord dans les fonctions

13.3.1 Passage par objet

En Python, lorsqu'on passe une liste à une fonction, la fonction reçoit une **référence vers le même objet**. La fonction peut donc modifier la liste originale.

```
def ajoute_zero(L):
    """Ajoute un zero en fin de L (modifie L en place)."""
    L.append(0)

L = [1, 2, 3]
ajoute_zero(L)
print(L)    # [1, 2, 3, 0]  -- L a ete modifiee
```

Propriété : Fonctions avec effet de bord

Une fonction qui **modifie une liste passée en argument** produit un **effet de bord**. Elle agit sur l'objet original, ce qui peut être voulu (tri en place) ou inattendu (bug).

Règle : indiquer clairement dans la docstring si la fonction modifie ses arguments.

Fonction en place	modifie `L`, ne renvoie rien (ou `None`)
Fonction pure	ne modifie pas `L`, renvoie un nouveau résultat

Exemples :

```
# Fonction en place (effet de bord)
tri_selection(L)     # modifie L, renvoie None
L.sort()             # idem

# Fonction pure (pas d'effet de bord)
L_triee = tri_fusion(L)    # L inchangee, renvoie une nouvelle liste
L_triee = sorted(L)        # idem
```

13.3.2 Exemple : erreur classique

La confusion entre effet de bord et valeur de retour est une source fréquente d'erreurs :

```
L = [3, 1, 2]
L2 = L.sort()     # sort() modifie L et renvoie None
print(L2)         # None  -- erreur classique !
print(L)          # [1, 2, 3]
```

`L.sort()` trie `L` en place et renvoie `None`. Si l'on veut une nouvelle liste triée sans toucher à `L`, il faut utiliser `sorted(L)`.

13.4 Conséquences pratiques

Règles à retenir :

- `L2 = L1` crée un alias, pas une copie. Toute modification de `L2` affecte `L1`.
- Pour copier une liste simple : `L2 = L1[:]` ou `L2 = list(L1)`.
- Pour copier une liste de listes : `import copy; L2 = copy.deepcopy(L1)`.
- Une fonction peut modifier une liste passée en argument (effet de bord). Le documenter dans la docstring.
- Ne pas affecter le résultat de `L.sort()` ou `L.append()` : ces méthodes renvoient `None`.

★ **Complément** —

Le modèle objet de Python : `id` et `is`

Chaque objet en Python possède un **identifiant unique** renvoyé par `id()`. L'opérateur `is` teste si deux variables pointent vers le même objet (même `id`), tandis que `==` teste l'égalité de contenu.

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(id(a), id(b), id(c))    # id(a) == id(b) != id(c)
print(a is b)    # True
print(a is c)    # False
print(a == c)    # True
```

Pour les petits entiers (de -5 à 256), Python réutilise les mêmes objets pour des raisons d'optimisation, ce qui peut donner des résultats surprenants avec `is` :

```
a = 5
b = 5
print(a is b)      # True  (optimisation interne)

a = 1000
b = 1000
print(a is b)      # False (en dehors de la plage optimisee)
```

En pratique : utiliser `==` pour comparer des valeurs, et réserver `is` au test `x is None`.

14

Dictionnaires

Plan du chapitre

14.1 Créer et accéder à un dictionnaire **75**
14.2 Modifier un dictionnaire **76**
14.3 Tester l'appartenance **76**
14.4 Parcourir un dictionnaire **76**
14.5 Applications **76**
14.5.1 Comptage de fréquences 76
14.5.2 Représentation de données structurées 77
14.5.3 Mémoïsation 77
14.6 Résumé des opérations **78**

★ Ce chapitre est aux limites du programme des concours, il apparait souvent dans les sujets d'oraux, presque systématiquement au concours G2E 2026.

Un **dictionnaire** est une structure de données qui associe des **clés** à des **valeurs**. Contrairement aux listes (indexées par des entiers), les clés peuvent être de n'importe quel type immuable : entier, chaîne de caractères, tuple, etc.

14.1 Créer et accéder à un dictionnaire

Définition : Dictionnaire

Un dictionnaire se note avec des accolades `{}` et des paires `clé: valeur` séparées par des virgules. L'accès à une valeur se fait par sa clé entre crochets.

```
d = {"nom": "Alice", "age": 20, "note": 17.5}

print(d["nom"])     # Alice
print(d["age"])     # 20
```

- Les clés doivent être de type **immuable** (`str`, `int`, `tuple`, ...).
- Les valeurs peuvent être de n'importe quel type.
- Un dictionnaire est **mutable** : on peut modifier, ajouter ou supprimer des entrées.

Créer un dictionnaire vide :

```
d = {}            # dictionnaire vide
d = dict()        # equivalent
```

14.2 Modifier un dictionnaire

```python
d = {"a": 1, "b": 2}

d["c"] = 3          # ajout d'une nouvelle entree
d["a"] = 10         # modification d'une entree existante
del d["b"]          # suppression d'une entree

print(d)    # {"a": 10, "c": 3}
```

Si on accède à une clé absente avec `d[clé]`, Python lève une erreur `KeyError`. Pour éviter cela, on utilise la méthode `get` :

```python
d = {"a": 1}
print(d.get("a", 0))    # 1  (cle presente : renvoie la valeur)
print(d.get("b", 0))    # 0  (cle absente : renvoie la valeur par defaut)
```

14.3 Tester l'appartenance

Propriété : Opérateur `in`

L'opérateur `in` teste si une **clé** est présente dans le dictionnaire :

```python
d = {"x": 10, "y": 20}

print("x" in d)     # True
print("z" in d)     # False
```

Remarque : `in` porte sur les clés, pas sur les valeurs.

14.4 Parcourir un dictionnaire

```python
d = {"a": 1, "b": 2, "c": 3}

# Parcours des cles
for cle in d:
    print(cle, d[cle])

# Parcours des cles et valeurs simultanement
for cle, val in d.items():
    print(cle, "->", val)

# Listes des cles, valeurs, paires
print(list(d.keys()))      # ["a", "b", "c"]
print(list(d.values()))    # [1, 2, 3]
print(list(d.items()))     # [("a", 1), ("b", 2), ("c", 3)]
```

Remarque : depuis Python 3.7, les dictionnaires préservent l'ordre d'insertion.

14.5 Applications

14.5.1 Comptage de fréquences

Un dictionnaire est idéal pour compter les occurrences d'éléments. La logique est la suivante : si la valeur est déjà une clé du dictionnaire, on incrémente son compteur ; sinon, on l'initialise à 1.

```python
liste = ["chat", "chien", "chat", "oiseau", "chien", "chat"]

occurrences = {}
for valeur in liste:
    if valeur in occurrences:
        occurrences[valeur] = occurrences[valeur] + 1
    else:
        occurrences[valeur] = 1

print(occurrences)    # {'chat': 3, 'chien': 2, 'oiseau': 1}
```

La méthode `.get(cle, defaut)` permet de condenser ce `if/else` en une seule ligne : `occurrences.get(valeur, 0)` renvoie le compteur actuel s'il existe, ou `0` sinon — ce qui correspond exactement au test explicite ci-dessus. Il n'y a pas de logique nouvelle, seulement une écriture plus compacte.

```python
liste = ["chat", "chien", "chat", "oiseau", "chien", "chat"]

occurrences = {}
for valeur in liste:
    occurrences[valeur] = occurrences.get(valeur, 0) + 1

print(occurrences)    # {'chat': 3, 'chien': 2, 'oiseau': 1}
```

14.5.2 Représentation de données structurées

Un dictionnaire permet de regrouper des informations hétérogènes sous un même objet :

```python
etudiant = {
    "nom": "Dupont",
    "prenom": "Marie",
    "notes": [14, 16, 12, 18],
    "absent": False
}

moyenne = sum(etudiant["notes"]) / len(etudiant["notes"])
print(etudiant["nom"], ":", moyenne)    # Dupont : 15.0
```

14.5.3 Mémoïsation

La **mémoïsation** consiste à stocker les résultats déjà calculés pour éviter de les recalculer. C'est particulièrement utile pour les fonctions récursives coûteuses :

```python
memo = {}

def fibonacci(n):
    """Renvoie le n-ieme terme de Fibonacci (avec memorisation)."""
    if n in memo:
        return memo[n]
    if n == 0:
        memo[0] = 0
        return 0
    if n == 1:
        memo[1] = 1
        return 1
    resultat = fibonacci(n - 1) + fibonacci(n - 2)
    memo[n] = resultat
    return resultat

print(fibonacci(50))    # calcul instantane
```

Sans mémoïsation, `fibonacci(50)` effectue plus de 10^{10} appels récursifs. Avec mémoïsation, chaque valeur est calculée une seule fois : la complexité passe de $O(2^n)$ à $O(n)$.

14.6 Résumé des opérations

Opération	Syntaxe
Accès	`d[cle]`
Accès sécurisé	`d.get(cle, defaut)`
Ajout / modification	`d[cle] = valeur`
Suppression	`del d[cle]`
Test d'appartenance	`cle in d`
Parcours des clés	`for cle in d`
Parcours clés + valeurs	`for cle, val in d.items()`
Liste des clés	`list(d.keys())`
Liste des valeurs	`list(d.values())`

★ **Complément —**

Complexité des opérations sur un dictionnaire

En Python, les dictionnaires sont implémentés par une **table de hachage**. La clé est transformée par une fonction de hachage en un indice, permettant un accès direct.

Opération	Dictionnaire	Liste
Accès par clé/indice	$O(1)$ moyen	$O(1)$
Test d'appartenance	$O(1)$ moyen	$O(n)$
Insertion	$O(1)$ moyen	$O(1)$ en fin, $O(n)$ en milieu

Le test `x in d` pour un dictionnaire est en $O(1)$, contre $O(n)$ pour une liste. Pour un grand nombre d'éléments, cette différence est décisive.

15

Bases de données et SQL

Plan du chapitre

15.1 Comment tester les exemples 79
15.2 Modèle relationnel 80
15.3 Requêtes de base : SELECT 80
15.3.1 Sélectionner des colonnes 80
15.3.2 Filtrer avec WHERE 80
15.3.3 Trier et limiter les résultats 81
15.4 Fonctions d'agrégation 81
15.4.1 Grouper avec GROUP BY 81
15.5 Jointures 82
15.6 Utilisation avec Python 82

★ **Ce chapitre est entièrement hors programme des concours.**

Une **base de données** permet de stocker et d'interroger de grandes quantités de données structurées. Le langage **SQL** (*Structured Query Language*) est le langage standard pour interagir avec une base de données relationnelle.

15.1 Comment tester les exemples

Plusieurs outils permettent d'exécuter les requêtes SQL de ce chapitre et du suivant.

DB Browser for SQLite

DB Browser for SQLite est un logiciel graphique gratuit (Windows, Mac, Linux) qui permet de créer une base de données, d'écrire des requêtes et de visualiser les résultats sans écrire de Python.

① Télécharger et installer **DB Browser for SQLite** (rechercher « DB Browser SQLite » sur le Web).
② Ouvrir le logiciel, créer une nouvelle base (*New Database*).
③ Dans l'onglet *Execute SQL*, saisir les requêtes `CREATE TABLE` et `INSERT` pour construire les tables, puis tester les `SELECT`.

En ligne de commande

SQLite est souvent disponible directement dans le terminal. Sous Windows (avec Python installé), on peut lancer l'interpréteur SQLite via Python :

```
# Depuis Python : creer une base en memoire et tester des requetes
import sqlite3
conn = sqlite3.connect(":memory:")    # base temporaire en memoire
curseur = conn.cursor()
# ... CREATE TABLE, INSERT, SELECT ...
```

L'option `":memory:"` crée une base temporaire qui disparaît à la fin du script — pratique pour tester sans créer de fichier.

Services en ligne

Des services en ligne permettent d'exécuter du SQL directement dans le navigateur sans installation (rechercher « SQLite online » ou « SQL sandbox »). Ils sont utiles pour tester rapidement une requête isolée.

15.2 Modèle relationnel

Définition : Table, attribut, enregistrement

Une **table** est un tableau où :
• chaque **colonne** (ou **attribut**) porte un nom et a un type fixé ;
• chaque **ligne** (ou **enregistrement**) représente un individu ou une observation.

La **clé primaire** est un attribut (ou un ensemble d'attributs) qui identifie de façon unique chaque enregistrement.

Exemple : table `especes`

id	nom	famille	masse_kg
1	Lion	Felidae	190
2	Éléphant	Elephantidae	5000
3	Guépard	Felidae	55
4	Girafe	Giraffidae	900
5	Tigre	Felidae	220

15.3 Requêtes de base : SELECT

15.3.1 Sélectionner des colonnes

```sql
-- Toutes les colonnes
SELECT * FROM especes;

-- Colonnes choisies
SELECT nom, masse_kg FROM especes;
```

15.3.2 Filtrer avec WHERE

```sql
-- Especes de masse superieure a 500 kg
SELECT nom, masse_kg FROM especes
WHERE masse_kg > 500;

-- Especes de la famille Felidae
SELECT nom FROM especes
WHERE famille = 'Felidae';

-- Conditions combinées
SELECT nom FROM especes
WHERE famille = 'Felidae' AND masse_kg > 100;
```

Propriété : Opérateurs de filtrage

Opérateur	Signification
`=`, `<>`	égal, différent
`<`, `<=`, `>`, `>=`	comparaisons numériques
`AND`, `OR`, `NOT`	opérateurs logiques
`BETWEEN a AND b`	valeur dans $[a, b]$
`IN (v1, v2, ...)`	valeur dans une liste
`LIKE 'motif'`	correspondance de chaîne (`%` = n'importe quels caractères)
`IS NULL` / `IS NOT NULL`	valeur absente / présente

```sql
-- Masse entre 100 et 500 kg
SELECT nom FROM especes WHERE masse_kg BETWEEN 100 AND 500;

-- Nom commençant par 'G'
SELECT nom FROM especes WHERE nom LIKE 'G%';

-- Famille Felidae ou Giraffidae
SELECT nom FROM especes WHERE famille IN ('Felidae', 'Giraffidae');
```

15.3.3 Trier et limiter les résultats

```sql
-- Trier par masse decroissante
SELECT nom, masse_kg FROM especes
ORDER BY masse_kg DESC;

-- Les 3 plus legeres
SELECT nom, masse_kg FROM especes
ORDER BY masse_kg ASC
LIMIT 3;
```

15.4 Fonctions d'agrégation

```sql
-- Nombre total d'especes
SELECT COUNT(*) FROM especes;

-- Masse moyenne
SELECT AVG(masse_kg) FROM especes;

-- Masse minimale et maximale
SELECT MIN(masse_kg), MAX(masse_kg) FROM especes;
```

15.4.1 Grouper avec GROUP BY

`GROUP BY` regroupe les enregistrements par valeur d'un attribut et applique une fonction d'agrégation à chaque groupe :

```sql
-- Nombre d'especes par famille
SELECT famille, COUNT(*) FROM especes
GROUP BY famille;

-- Masse moyenne par famille, uniquement si le groupe a au moins 2 especes
SELECT famille, AVG(masse_kg) FROM especes
GROUP BY famille
HAVING COUNT(*) >= 2;
```

Résultat pour la première requête :

famille	COUNT(*)
Elephantidae	1
Felidae	3
Giraffidae	1

15.5 Jointures

Souvent, les données sont réparties sur plusieurs tables reliées entre elles par des clés. Une **jointure** combine les lignes de deux tables selon une condition.

Exemple : table `observations(id, espece_id, lieu, annee, nombre)`

id	espece_id	lieu	annee	nombre
1	1	Serengeti	2020	12
2	3	Serengeti	2020	5
3	2	Amboseli	2021	3
4	1	Kruger	2021	8
5	3	Amboseli	2022	2

```sql
-- Noms des especes observees au Serengeti
SELECT especes.nom, observations.annee, observations.nombre
FROM especes
JOIN observations ON especes.id = observations.espece_id
WHERE observations.lieu = 'Serengeti';

-- Nombre total d'individus observes par espece
SELECT especes.nom, SUM(observations.nombre) AS total
FROM especes
JOIN observations ON especes.id = observations.espece_id
GROUP BY especes.nom;
```

Propriété : JOIN

`JOIN ... ON condition` combine chaque ligne de la première table avec chaque ligne de la seconde table pour lesquelles la condition est vraie.

Type de jointure	Lignes conservées
`JOIN` (ou `INNER JOIN`)	uniquement les paires vérifiant la condition
`LEFT JOIN`	toutes les lignes de la table de gauche

15.6 Utilisation avec Python

Python intègre le module `sqlite3` qui permet de créer et d'interroger une base de données SQLite directement depuis un script.

```python
import sqlite3

# Connexion (crée le fichier si inexistant)
conn = sqlite3.connect("animaux.db")
curseur = conn.cursor()

# Creation de la table
curseur.execute("""
    CREATE TABLE IF NOT EXISTS especes (
        id INTEGER PRIMARY KEY,
        nom TEXT,
```

```
        famille TEXT,
        masse_kg REAL
    )
""")

# Insertion de donnees
curseur.execute("INSERT INTO especes VALUES (1, 'Lion', 'Felidae', 190)")
curseur.execute("INSERT INTO especes VALUES (2, 'Elephant', 'Elephantidae',
    5000)")
conn.commit()

# Requete
curseur.execute("SELECT nom, masse_kg FROM especes WHERE masse_kg > 500")
resultats = curseur.fetchall()
for ligne in resultats:
    print(ligne)    # ('Elephant', 5000.0)

conn.close()
```

```
# Requete avec parametre (eviter les injections SQL)
espece_cherchee = "Felidae"
curseur.execute(
    "SELECT nom FROM especes WHERE famille = ?",
    (espece_cherchee,)
)
resultats = curseur.fetchall()
```

Méthodes du curseur :

- `execute(requete)` : exécute une requête SQL.
- `fetchall()` : renvoie tous les résultats sous forme de liste de tuples.
- `fetchone()` : renvoie le premier résultat.
- `conn.commit()` : valide les modifications (INSERT, UPDATE, DELETE).

16

SQL avancé

Plan du chapitre

16.1 Alias **85**
16.2 Sous-requêtes **85**
16.2.1 Sous-requête dans WHERE 86
16.2.2 Sous-requête dans FROM 86
16.3 LEFT JOIN **86**
16.4 Modifier les données **86**
16.4.1 Insérer : INSERT 86
16.4.2 Modifier : UPDATE 87
16.4.3 Supprimer : DELETE 87
16.5 Définir le schéma **87**
16.5.1 Créer une table 87
16.5.2 Supprimer une table 88

> ★ **Ce chapitre est entièrement hors programme des concours.**

Ce chapitre approfondit le SQL introduit au chapitre précédent. On utilise les mêmes tables `especes` et `observations` comme fil conducteur.

16.1 Alias

Les **alias** permettent de renommer temporairement une table ou une colonne dans une requête, pour la lisibilité ou pour éviter les ambiguïtés.

```sql
-- Alias de colonne (AS)
SELECT nom AS espece, masse_kg AS masse
FROM especes
ORDER BY masse DESC;

-- Alias de table (utile pour les jointures)
SELECT e.nom, o.lieu, o.annee
FROM especes AS e
JOIN observations AS o ON e.id = o.espece_id
WHERE o.annee = 2021;
```

16.2 Sous-requêtes

Une **sous-requête** est une requête imbriquée dans une autre. Elle peut apparaître dans `WHERE`, `FROM` ou `SELECT`.

16.2.1 Sous-requête dans WHERE

```sql
-- Especes dont la masse est superieure a la moyenne
SELECT nom, masse_kg FROM especes
WHERE masse_kg > (SELECT AVG(masse_kg) FROM especes);

-- Especes qui ont ete observees au moins une fois
SELECT nom FROM especes
WHERE id IN (SELECT espece_id FROM observations);

-- Especes qui n'ont jamais ete observees
SELECT nom FROM especes
WHERE id NOT IN (SELECT espece_id FROM observations);
```

16.2.2 Sous-requête dans FROM

On peut utiliser une sous-requête comme table temporaire (il faut alors lui donner un alias) :

```sql
-- Nombre moyen d'individus par observation, par espece
SELECT e.nom, stats.moy
FROM especes AS e
JOIN (
    SELECT espece_id, AVG(nombre) AS moy
    FROM observations
    GROUP BY espece_id
) AS stats ON e.id = stats.espece_id;
```

16.3 LEFT JOIN

Un `JOIN` ordinaire (`INNER JOIN`) n'affiche que les lignes ayant une correspondance dans les deux tables. Un `LEFT JOIN` conserve **toutes** les lignes de la table de gauche, même sans correspondance dans la table de droite (les colonnes manquantes valent alors `NULL`).

```sql
-- Toutes les especes, avec leur nombre total d'observations
-- (0 si aucune observation)
SELECT e.nom, COALESCE(SUM(o.nombre), 0) AS total
FROM especes AS e
LEFT JOIN observations AS o ON e.id = o.espece_id
GROUP BY e.nom;
```

`COALESCE(val, defaut)` renvoie `val` si elle est non nulle, `defaut` sinon — ici, 0 à la place de `NULL`.

16.4 Modifier les données

16.4.1 Insérer : INSERT

```sql
INSERT INTO especes (id, nom, famille, masse_kg)
VALUES (6, 'Leopard', 'Felidae', 60);

-- Insertion multiple
INSERT INTO especes (id, nom, famille, masse_kg) VALUES
    (7, 'Zebre', 'Equidae', 350),
    (8, 'Hyene', 'Hyaenidae', 70);
```

16.4.2 Modifier : UPDATE

```sql
-- Corriger la masse du lion
UPDATE especes
SET masse_kg = 200
WHERE nom = 'Lion';

-- Augmenter de 10 % la masse de tous les Felidae
UPDATE especes
SET masse_kg = masse_kg * 1.1
WHERE famille = 'Felidae';
```

Attention : un `UPDATE` sans `WHERE` modifie **toutes** les lignes.

16.4.3 Supprimer : DELETE

```sql
-- Supprimer une espece
DELETE FROM especes WHERE nom = 'Hyene';

-- Supprimer toutes les observations avant 2020
DELETE FROM observations WHERE annee < 2020;
```

16.5 Définir le schéma

16.5.1 Créer une table

```sql
CREATE TABLE especes (
    id        INTEGER PRIMARY KEY,
    nom       TEXT    NOT NULL,
    famille   TEXT    NOT NULL,
    masse_kg  REAL
);

CREATE TABLE observations (
    id         INTEGER PRIMARY KEY,
    espece_id  INTEGER NOT NULL REFERENCES especes(id),
    lieu       TEXT,
    annee      INTEGER,
    nombre     INTEGER DEFAULT 1
);
```

Propriété : Contraintes

Contrainte	Signification
`PRIMARY KEY`	identifiant unique, non nul, par table
`NOT NULL`	la colonne ne peut pas être vide
`UNIQUE`	toutes les valeurs de la colonne sont distinctes
`DEFAULT val`	valeur utilisée si aucune n'est fournie
`REFERENCES t(col)`	clé étrangère : référence une clé primaire d'une autre table

La **clé étrangère** (*foreign key*) garantit l'intégrité référentielle : on ne peut pas insérer une observation pour une espèce inexistante.

16.5.2 Supprimer une table

```sql
DROP TABLE observations;    -- supprime la table et ses donnees
```

★ Complément —

Vues et index

Vue : une vue est une requête nommée et stockée, utilisable comme une table virtuelle.

```sql
CREATE VIEW felins AS
SELECT nom, masse_kg FROM especes WHERE famille = 'Felidae';

SELECT * FROM felins;    -- utilisation de la vue
```

Index : un index accélère les recherches sur une colonne fréquemment utilisée dans `WHERE` ou `JOIN`, au prix d'un espace de stockage supplémentaire.

```sql
CREATE INDEX idx_famille ON especes(famille);
```

Sans index, une recherche par `famille` nécessite de parcourir toute la table ($O(n)$). Avec un index B-arbre, la complexité devient $O(\log n)$.

17

Graphes — bases

Plan du chapitre

17.1 Définitions **89**
17.2 Représentations en Python **90**
17.2.1 Dictionnaire de listes de voisins 90
17.2.2 Matrice d'adjacence 90
17.3 Parcours d'un graphe **91**
17.3.1 Parcours en largeur (BFS) 91
17.3.2 Parcours en profondeur (DFS) 91
17.3.3 Connexité 92
17.4 Application : distance entre sommets **92**
17.5 Énumération : exploration d'un arbre de recherche **92**
17.5.1 Arbre de recherche 92
17.5.2 Suites avec répétition 93
17.5.3 Élagage de l'arbre 93

★ **Ce chapitre est entièrement hors programme des concours.**

Les **graphes** sont une structure mathématique fondamentale pour modéliser des relations : réseaux routiers, réseaux alimentaires, molécules, réseaux sociaux, etc. Ce chapitre présente les bases et leur implémentation en Python.

17.1 Définitions

Définition : Graphe

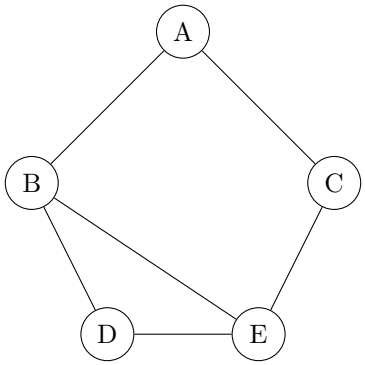
Un **graphe** $G = (S, A)$ est composé de :
- un ensemble de **sommets** S (ou nœuds, *vertices*) ;
- un ensemble d'**arêtes** A (ou *edges*), chaque arête reliant deux sommets.

Graphe non orienté	les arêtes n'ont pas de sens ($\{u, v\} = \{v, u\}$)
Graphe orienté	les arcs ont un sens ($u \to v \neq v \to u$)
Graphe pondéré	chaque arête porte un poids (distance, coût...)

Vocabulaire courant :

Voisins de u	sommets reliés à u par une arête
Degré de u	nombre de voisins de u
Chemin	suite de sommets consécutivement reliés
Cycle	chemin qui revient à son point de départ
Graphe connexe	tout sommet est accessible depuis tout autre sommet
Arbre	graphe connexe sans cycle

Exemple de graphe :



17.2 Représentations en Python

17.2.1 Dictionnaire de listes de voisins

C'est la représentation la plus naturelle et la plus utilisée en pratique. Chaque sommet est une clé du dictionnaire ; sa valeur est la liste de ses voisins.

```
# Graphe de l'exemple ci-dessus (non oriente)
graphe = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'E'],
    'D': ['B', 'E'],
    'E': ['C', 'D', 'B']
}

print(graphe['A'])          # ['B', 'C']  -- voisins de A
print(len(graphe['B']))     # 3           -- degre de B
```

Pour un **graphe orienté**, on n'ajoute la relation que dans un sens :

```
graphe_oriente = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
```

17.2.2 Matrice d'adjacence

On numérote les sommets de 0 à $n-1$. La matrice d'adjacence M est telle que $M[i][j] = 1$ s'il existe une arête entre i et j, 0 sinon.

```
# Graphe a 4 sommets : 0-1, 0-2, 1-3, 2-3
n = 4
M = [[0]*n for _ in range(n)]

def ajoute_arete(M, i, j):
    """Ajoute l'arete {i, j} dans la matrice M (graphe non oriente)."""
    M[i][j] = 1
    M[j][i] = 1

ajoute_arete(M, 0, 1)
ajoute_arete(M, 0, 2)
ajoute_arete(M, 1, 3)
ajoute_arete(M, 2, 3)
```

```python

# Voisins du sommet 0
voisins_0 = [j for j in range(n) if M[0][j] == 1]
print(voisins_0)   # [1, 2]
```

Comparaison des deux représentations :

	Dict. de voisins	Matrice d'adjacence
Espace mémoire	$O(\|S\| + \|A\|)$	$O(\|S\|^2)$
Test d'adjacence	$O(\text{degré})$	$O(1)$
Liste des voisins	$O(\text{degré})$	$O(\|S\|)$
Adapté pour	graphes creux	graphes denses

17.3 Parcours d'un graphe

Un **parcours** visite tous les sommets accessibles depuis un sommet de départ, sans en visiter deux fois. On maintient une liste des sommets déjà visités.

17.3.1 Parcours en largeur (BFS)

Le **parcours en largeur** (*Breadth-First Search*) visite d'abord les voisins immédiats, puis les voisins des voisins, etc. Il utilise une **file** (premier entré, premier sorti).

```python
def bfs(graphe, depart):
    """Renvoie la liste des sommets visites depuis depart en largeur d'abord."""
    visites = [depart]
    file = [depart]
    while file:
        sommet = file.pop(0)
        for voisin in graphe[sommet]:
            if voisin not in visites:
                visites.append(voisin)
                file.append(voisin)
    return visites

print(bfs(graphe, 'A'))    # ['A', 'B', 'C', 'D', 'E']
```

Propriété clé : le BFS visite les sommets par ordre croissant de distance (en nombre d'arêtes) depuis le sommet de départ.

17.3.2 Parcours en profondeur (DFS)

Le **parcours en profondeur** (*Depth-First Search*) explore le plus loin possible avant de revenir en arrière. La version récursive est naturelle.

```python
def dfs(graphe, depart, visites=None):
    """Renvoie la liste des sommets visites depuis depart en profondeur d'abord.
       """
    if visites is None:
        visites = []
    visites.append(depart)
    for voisin in graphe[depart]:
        if voisin not in visites:
            dfs(graphe, voisin, visites)
    return visites

print(dfs(graphe, 'A'))    # ['A', 'B', 'D', 'E', 'C']  (ordre possible)
```

Remarque : l'argument `visites=None` avec le test `if visites is None: visites = []` est le schéma correct pour un argument mutable par défaut dans une fonction récursive (cf. chapitre sur la mutabilité).

17.3.3 Connexité

On peut tester si un graphe est connexe : il suffit de lancer un parcours depuis n'importe quel sommet et de vérifier que tous les sommets ont été visités.

```
def est_connexe(graphe):
    """Renvoie True si le graphe est connexe."""
    if len(graphe) == 0:
        return True
    depart = next(iter(graphe))   # premier sommet du dictionnaire
    visites = bfs(graphe, depart)
    return len(visites) == len(graphe)

print(est_connexe(graphe))   # True
```

17.4 Application : distance entre sommets

Le BFS permet de calculer la distance minimale (en nombre d'arêtes) entre deux sommets :

```
def distance(graphe, depart, arrivee):
    """Renvoie la distance (nb d'aretes) entre depart et arrivee, ou -1 si
       inaccessible."""
    dist = {depart: 0}
    file = [depart]
    while file:
        sommet = file.pop(0)
        if sommet == arrivee:
            return dist[sommet]
        for voisin in graphe[sommet]:
            if voisin not in dist:
                dist[voisin] = dist[sommet] + 1
                file.append(voisin)
    return -1

print(distance(graphe, 'A', 'E'))   # 2
print(distance(graphe, 'A', 'D'))   # 2
```

17.5 Énumération : exploration d'un arbre de recherche

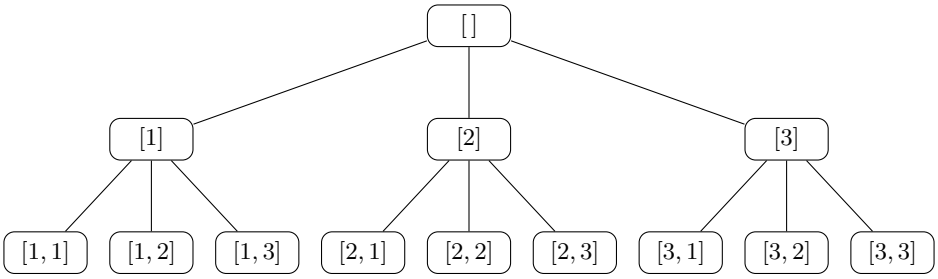
17.5.1 Arbre de recherche

Nombreux problèmes consistent à **générer toutes les séquences** vérifiant une propriété. Ces algorithmes ont tous la même structure : à chaque étape, on choisit un élément à ajouter à la séquence en cours. L'ensemble des choix possibles forme un **arbre de recherche**.

Définition : Arbre de recherche

Un **arbre de recherche** est un arbre dont :
- la **racine** est la séquence vide $[\,]$;
- chaque **nœud** à profondeur k représente une séquence partielle de longueur k ;
- les **fils** d'un nœud sont obtenus en ajoutant un élément valide ;
- les **feuilles** (profondeur p) sont les séquences complètes cherchées.

Exemple : arbre de recherche pour toutes les listes de 2 éléments dans $\{1, 2, 3\}$.



17.5.2 Suites avec répétition

La fonction suivante génère toutes les suites de p entiers dans $\{1, \dots, n\}$ (chaque entier pouvant apparaître plusieurs fois). La version itérative construit l'arbre **niveau par niveau** (parcours en largeur) ; la version récursive l'explore **en profondeur**.

```
# Version iterative (parcours en largeur de l'arbre)
def suites(p, n):
    """Renvoie la liste de toutes les suites de p entiers dans {1,...,n}."""
    L = [[]]
    for i in range(p):
        S = []
        for x in L:
            for k in range(1, n + 1):
                S.append(x + [k])
        L = S
    return L

print(suites(2, 3))   # [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
print(len(suites(3, 5)))   # 125  (= 5^3)
```

ou encore avec des listes en compréhension

```
# Version iterative (parcours en largeur de l'arbre)
def suites(p, n):
    """Renvoie la liste de toutes les suites de p entiers dans {1,...,n}."""
    L = [[]]
    for i in range(p):
        L = [x + [k] for x in L  for k in range(1, n + 1)]
    return L
```

```
# Version recursive (parcours en profondeur de l'arbre)
def suites(p, n):
    """Renvoie la liste de toutes les suites de p entiers dans {1,...,n}."""
    if p == 0:
        return [[]]
    return [x + [k] for x in suites(p - 1, n) for k in range(1, n + 1)]
```

Correspondance arbre / code :

Racine []	`L = [[]]` (initialisation)
Un niveau de l'arbre	une itération de la boucle `for i in range(p)`
Fils d'un nœud	boucle `for k in range(1, n+1)`
Cas de base $p = 0$	feuille atteinte, on renvoie `[[]]`

17.5.3 Élagage de l'arbre

On peut **élaguer** l'arbre de recherche en ajoutant une condition sur k. Seules les branches vérifiant la contrainte sont développées.

Listes (arrangements sans répétition)

```python
def listes(p, n):
    """Renvoie la liste de toutes les listes de p entiers distincts dans {1,..., n}."""
    L = [[]]
    for i in range(p):
        S = []
        for x in L:
            for k in range(1, n + 1):
                if k not in x:          # elagage : k deja present -> branche coupee
                    S.append(x + [k])
        L = S
    return L

print(len(listes(2, 4)))    # 12  (= 4 * 3)
```

Suites strictement croissantes (combinaisons)

```python
def strict_croissantes(p, n):
    """Renvoie la liste de toutes les suites strictement croissantes
    de p entiers dans {1,...,n}."""
    L = [[k] for k in range(1, n + 1)]
    for i in range(p - 1):
        L = [x + [k] for x in L for k in range(x[-1] + 1, n + 1)]
    return L

print(len(strict_croissantes(4, 30)))    # 27405  (= C(30,4))
```

Remarque : les feuilles de cet arbre sont exactement les combinaisons de p éléments parmi n, et leur nombre est $\binom{n}{p}$. On peut vérifier :

```python
def parmi(p, n):
    """Renvoie le nombre de combinaisons de p elements parmi n (C(n,p))."""
    if not (0 <= p <= n):
        return 0
    num, den = 1, 1
    for i in range(p):
        num, den = num * (n - i), den * (i + 1)
    return num // den

print(parmi(4, 30))    # 27405
```

Anagrammes

La génération de tous les anagrammes d'un mot suit la même structure : à chaque niveau, on choisit une lettre non encore « épuisée » dans le mot.

```python
def occurrences(c, mot):
    """Renvoie le nombre d'occurrences de c dans mot."""
    return sum(1 for t in mot if t == c)

def lettres_distinctes(mot):
    """Renvoie la liste des lettres distinctes de mot (sans doublon)."""
    S = []
    for t in mot:
        if t not in S:
            S.append(t)
    return S

def anagrammes(mot):
    """Renvoie la liste de tous les anagrammes de mot."""
```

```
    L = ['']
    for _ in range(len(mot)):
        S = []
        for x in L:
            for k in lettres_distinctes(mot):
                if occurrences(k, x) < occurrences(k, mot):    # lettre encore
                    disponible
                    S.append(x + k)
        L = S
    return L

print(len(anagrammes('sarah')))      # 60
print(len(anagrammes('thimothe')))   # 10080
```

Synthèse — pattern général de l'énumération par arbre :

```
L = [[]]                           # racine : sequence vide
for i in range(profondeur):
    S = []
    for x in L:                    # pour chaque noeud courant
        for k in candidats:        # pour chaque fils possible
            if valide(x, k):       # elagage : on ne suit que les branches valides
                S.append(x + [k])
    L = S
return L                           # feuilles = solutions
```

18

Graphes avancés

Plan du chapitre

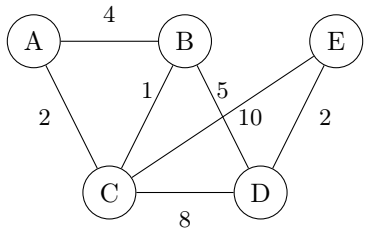
18.1 Graphes pondérés **97**
18.2 Algorithme de Dijkstra **97**
18.2.1 Problème du plus court chemin 97
18.2.2 Implémentation 98
18.2.3 Déroulement sur l'exemple 98
18.2.4 Reconstruction du chemin 98
18.3 Détection de cycles **99**

★ **Ce chapitre est entièrement hors programme des concours.**

18.1 Graphes pondérés

Un **graphe pondéré** associe un **poids** (distance, coût, durée) à chaque arête. On représente un tel graphe par un dictionnaire dont les valeurs sont des listes de paires `(voisin, poids)` :

```
# Graphe pondere non oriente
# Chaque entree : sommet -> [(voisin, poids), ...]
graphe = {
    'A': [('B', 4), ('C', 2)],
    'B': [('A', 4), ('C', 1), ('D', 5)],
    'C': [('A', 2), ('B', 1), ('D', 8), ('E', 10)],
    'D': [('B', 5), ('C', 8), ('E', 2)],
    'E': [('C', 10), ('D', 2)]
}
```



18.2 Algorithme de Dijkstra

18.2.1 Problème du plus court chemin

Le problème du **plus court chemin** consiste à trouver, dans un graphe pondéré à poids positifs, le chemin de coût minimal entre un sommet source et tous les autres sommets.

Propriété : Algorithme de Dijkstra

On maintient pour chaque sommet une distance provisoire `dist[s]`, initialement $+\infty$ sauf pour le sommet de départ (distance 0). À chaque étape :

① Choisir le sommet non encore traité de distance minimale.
② Le marquer comme traité.
③ Mettre à jour la distance de ses voisins : si passer par lui est plus court, on améliore.

L'algorithme se termine quand tous les sommets sont traités.

18.2.2 Implémentation

```python
def dijkstra(graphe, depart):
    """Renvoie un dictionnaire des distances minimales depuis depart.
    graphe : dict {sommet: [(voisin, poids), ...]}
    Les poids doivent etre positifs ou nuls."""
    dist = {s: float('inf') for s in graphe}
    dist[depart] = 0
    non_traites = list(graphe.keys())

    while non_traites:
        # Sommet non traite de distance minimale
        u = min(non_traites, key=lambda s: dist[s])
        if dist[u] == float('inf'):
            break
        non_traites.remove(u)
        for voisin, poids in graphe[u]:
            nouvelle_dist = dist[u] + poids
            if nouvelle_dist < dist[voisin]:
                dist[voisin] = nouvelle_dist

    return dist

print(dijkstra(graphe, 'A'))
# {'A': 0, 'B': 3, 'C': 2, 'D': 8, 'E': 10}
```

`float('inf')` représente $+\infty$ en Python : toute valeur finie lui est inférieure.

18.2.3 Déroulement sur l'exemple

Départ : A. Distances initiales : $A = 0$, autres $= +\infty$.

Étape	Sommet traité	dist[A]	dist[B]	dist[C]	dist[D]	dist[E]
0	—	0	∞	∞	∞	∞
1	A	0	4	2	∞	∞
2	C	0	3	2	10	12
3	B	0	3	2	8	12
4	D	0	3	2	8	10
5	E	0	3	2	8	10

Le chemin le plus court de A à E est $A \to C \to B \to D \to E$, de longueur 10.

Remarque : cette implémentation est en $O(|S|^2)$. En utilisant une file de priorité (`heapq`), on peut atteindre $O((|S| + |A|) \log |S|)$.

18.2.4 Reconstruction du chemin

Pour retrouver le chemin optimal (et pas seulement sa longueur), on mémorise le prédécesseur de chaque sommet :

```python
def dijkstra_chemin(graphe, depart, arrivee):
    """Renvoie (distance, chemin) du plus court chemin de depart a arrivee."""
    dist = {s: float('inf') for s in graphe}
    pred = {s: None for s in graphe}
    dist[depart] = 0
    non_traites = list(graphe.keys())

    while non_traites:
        u = min(non_traites, key=lambda s: dist[s])
        if dist[u] == float('inf'):
            break
        non_traites.remove(u)
        for voisin, poids in graphe[u]:
            if dist[u] + poids < dist[voisin]:
                dist[voisin] = dist[u] + poids
                pred[voisin] = u

    # Reconstitution du chemin
    chemin = []
    s = arrivee
    while s is not None:
        chemin.append(s)
        s = pred[s]
    chemin.reverse()
    return dist[arrivee], chemin

d, c = dijkstra_chemin(graphe, 'A', 'E')
print(d, c)   # 10  ['A', 'C', 'B', 'D', 'E']
```

18.3 Détection de cycles

Pour un **graphe orienté**, on détecte un cycle avec un DFS en distinguant trois états pour chaque sommet :

- **non visité** : pas encore atteint ;
- **en cours** : dans la pile d'appels récursifs en cours ;
- **terminé** : entièrement exploré.

Un cycle existe si et seulement si on rencontre un sommet **en cours** lors de l'exploration.

```python
def a_cycle(graphe):
    """Renvoie True si le graphe oriente contient un cycle."""
    visites = []
    en_cours = []

    def explore(sommet):
        en_cours.append(sommet)
        for voisin in graphe[sommet]:
            if voisin in en_cours:
                return True
            if voisin not in visites:
                if explore(voisin):
                    return True
        en_cours.remove(sommet)
        visites.append(sommet)
        return False

    for sommet in graphe:
        if sommet not in visites:
            if explore(sommet):
                return True
    return False

g_sans_cycle = {'A': ['B'], 'B': ['C'], 'C': []}
g_avec_cycle = {'A': ['B'], 'B': ['C'], 'C': ['A']}
```

```python
print(a_cycle(g_sans_cycle))    # False
print(a_cycle(g_avec_cycle))    # True
```

★ **Complément —**

Arbre couvrant minimal — algorithme de Kruskal

Un **arbre couvrant minimal** d'un graphe connexe pondéré est un arbre qui relie tous les sommets avec le coût total minimal. L'algorithme de Kruskal trie les arêtes par poids croissant et les ajoute une à une, en rejetant celles qui créeraient un cycle.

```python
def kruskal(sommets, aretes):
    """Renvoie les aretes de l'arbre couvrant minimal.
    aretes : liste de (poids, u, v)"""
    aretes_triees = sorted(aretes)
    parent = {s: s for s in sommets}

    def trouve(s):
        if parent[s] != s:
            parent[s] = trouve(parent[s])
        return parent[s]

    def union(u, v):
        parent[trouve(u)] = trouve(v)

    acm = []
    for poids, u, v in aretes_triees:
        if trouve(u) != trouve(v):
            union(u, v)
            acm.append((poids, u, v))
    return acm

sommets = ['A', 'B', 'C', 'D', 'E']
aretes = [(4,'A','B'),(2,'A','C'),(1,'B','C'),(5,'B','D'),
          (8,'C','D'),(10,'C','E'),(2,'D','E')]
print(kruskal(sommets, aretes))
# [(1,'B','C'), (2,'A','C'), (2,'D','E'), (5,'B','D')]  -- cout total 10
```

La structure **Union-Find** (ou *Disjoint Set Union*) utilisée ici permet de tester efficacement si deux sommets sont dans la même composante connexe.

19

Méthodes numériques

Plan du chapitre

19.1 Intégration numérique **101**
19.1.1 Principe 101
19.1.2 Méthode des rectangles 102
19.1.3 Méthode des trapèzes 103
19.1.4 Erreur et convergence 103
19.2 Intégration à partir de données discrètes **104**
19.3 Méthode de Newton **104**
19.3.1 Principe 104
19.3.2 Implémentation 105
19.3.3 Convergence 105
19.4 Méthode d'Euler **106**
19.4.1 Équations différentielles du premier ordre 106
19.4.2 Implémentation 106
19.4.3 Exemples 106
19.4.4 Précision et pas de temps 108

Les **méthodes numériques** permettent d'obtenir des valeurs approchées de quantités qui n'ont pas d'expression analytique simple. Ce chapitre traite de l'intégration numérique ; la résolution approchée d'équations par dichotomie a été vue au chapitre 10.

19.1 Intégration numérique

19.1.1 Principe

On cherche à approcher $I = \displaystyle\int_a^b f(x)\,\mathrm{d}x$ lorsque f n'a pas de primitive explicite ou que l'on ne dispose que de valeurs discrètes de f. L'idée est de découper $[a, b]$ en n sous-intervalles de largeur $h = \dfrac{b-a}{n}$ et d'approcher f sur chaque sous-intervalle par une forme simple.

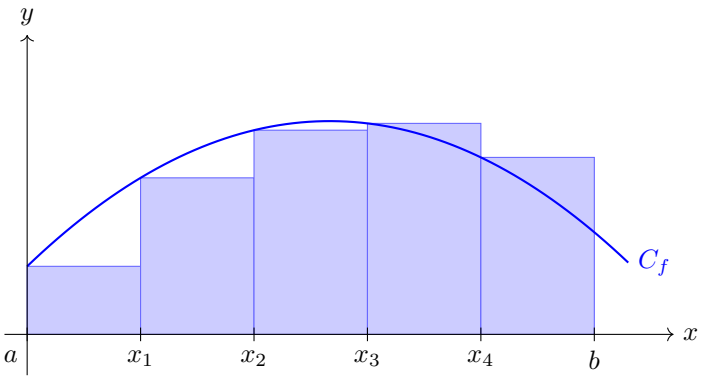
Les points de subdivision sont $x_k = a + k\,h$ pour $k = 0, 1, \ldots, n$.

19.1.2 Méthode des rectangles

Propriété : Méthode des rectangles

On approche f sur $[x_k, x_{k+1}]$ par sa valeur en un point choisi :

Point gauche	$f(x_k)$	$R_g = h\sum_{k=0}^{n-1} f(x_k)$
Point droit	$f(x_{k+1})$	$R_d = h\sum_{k=1}^{n} f(x_k)$
Point milieu	$f\left(\frac{x_k + x_{k+1}}{2}\right)$	$R_m = h\sum_{k=0}^{n-1} f\left(x_k + \frac{h}{2}\right)$



Rectangles (point gauche), $n = 5$

```python
def integrale_rect_gauche(f, a, b, n):
    """Approximation de l'integrale de f sur [a,b] par n rectangles (point gauche)."""
    h = (b - a) / n
    somme = 0
    for k in range(n):
        somme = somme + f(a + k * h)
    return h * somme

def integrale_rect_milieu(f, a, b, n):
    """Approximation de l'integrale de f sur [a,b] par n rectangles (point milieu)."""
    h = (b - a) / n
    somme = 0
    for k in range(n):
        somme = somme + f(a + (k + 0.5) * h)
    return h * somme

def integrale_rect_droit(f, a, b, n):
    """Approximation de l'integrale de f sur [a,b] par n rectangles (point droit)."""
    h = (b - a) / n
    somme = 0
    for k in range(1, n + 1):
        somme = somme + f(a + k * h)
    return h * somme
```

Exemple : approximation de $\int_0^1 e^{-x^2}\,\mathrm{d}x$ (pas de primitive élémentaire).

```python
from math import exp

def f(x):
```

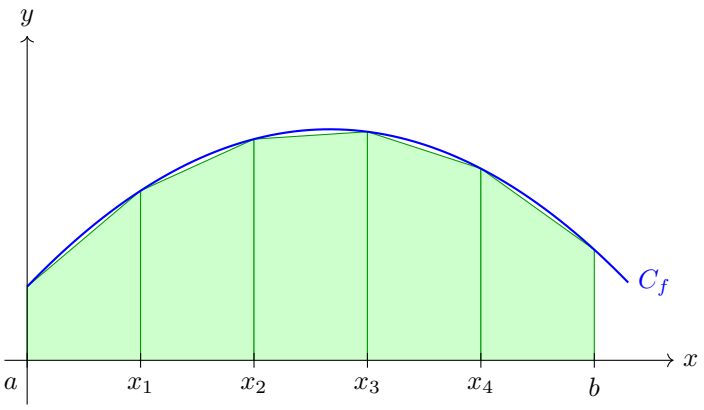
```
    return exp(-x**2)

print(integrale_rect_milieu(f, 0, 1, 1000))    # 0.7468241...
```

19.1.3 Méthode des trapèzes

Propriété : Méthode des trapèzes

On approche f sur $[x_k, x_{k+1}]$ par le segment reliant $(x_k, f(x_k))$ à $(x_{k+1}, f(x_{k+1}))$. L'aire du trapèze vaut $h \cdot \dfrac{f(x_k) + f(x_{k+1})}{2}$.
En sommant sur tous les sous-intervalles, on obtient :

$$T = h\left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f(x_k)\right)$$



Méthode des trapèzes, $n = 5$

```
def integrale_trapezes(f, a, b, n):
    """Approximation de l'integrale de f sur [a,b] par la methode des trapezes.
       """
    h = (b - a) / n
    somme = (f(a) + f(b)) / 2
    for k in range(1, n):
        somme = somme + f(a + k * h)
    return h * somme

print(integrale_trapezes(f, 0, 1, 1000))    # 0.7468241...
```

19.1.4 Erreur et convergence

Méthode	**Erreur** (n sous-intervalles)	**Ordre**
Rectangles gauche/droit	$O\left(\frac{1}{n}\right)$	1
Rectangles milieu	$O\left(\frac{1}{n^2}\right)$	2
Trapèzes	$O\left(\frac{1}{n^2}\right)$	2

Doubler n divise l'erreur par 2 (ordre 1) ou par 4 (ordre 2). En pratique, la méthode des trapèzes est préférable aux rectangles gauche/droit.

Vérification sur $\displaystyle\int_0^1 x^2\,\mathrm{d}x = \frac{1}{3}$ **:**

```python
def g(x):
    return x**2

for n in [10, 100, 1000]:
    rg = integrale_rect_gauche(g, 0, 1, n)
    tr = integrale_trapezes(g, 0, 1, n)
    print(f"n={n:5d}  rect_gauche={rg:.6f}  trapezes={tr:.6f}")

# n=   10  rect_gauche=0.285000  trapezes=0.335000
# n=  100  rect_gauche=0.328500  trapezes=0.333350
# n= 1000  rect_gauche=0.332850  trapezes=0.333334
```

19.2 Intégration à partir de données discrètes

En pratique (expériences, mesures biologiques), on ne dispose pas de la fonction f mais d'un tableau de valeurs (x_i, y_i). La méthode des trapèzes s'adapte directement :

```python
def integrale_trapezes_discret(X, Y):
    """Approximation de l'integrale a partir de points (X[i], Y[i]).
    X doit etre trie par ordre croissant."""
    somme = 0
    for i in range(len(X) - 1):
        somme = somme + (X[i+1] - X[i]) * (Y[i] + Y[i+1]) / 2
    return somme

# Exemple : absorbance mesuree a differentes longueurs d'onde
longueurs = [400, 420, 450, 500, 550, 600]
absorbances = [0.1, 0.3, 0.8, 1.2, 0.6, 0.2]

print(integrale_trapezes_discret(longueurs, absorbances))   # 92.5
```

Remarque : cette version fonctionne même si les x_i ne sont pas régulièrement espacés.

19.3 Méthode de Newton

19.3.1 Principe

La **méthode de Newton** (ou méthode de Newton-Raphson) est une autre méthode pour résoudre $f(x) = 0$ lorsque f est dérivable. Elle est en général beaucoup plus rapide que la dichotomie.

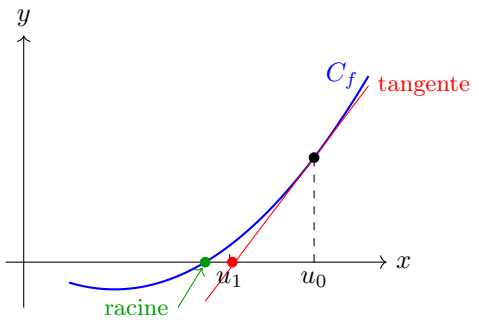
Propriété : Méthode de Newton

On part d'une valeur initiale u_0 proche de la racine. À chaque étape, on remplace f par sa **tangente** en u_n et on prend son zéro comme nouvelle approximation :

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}$$

On s'arrête lorsque $|f(u_n)| < \varepsilon$.

Interprétation géométrique : la tangente à la courbe de f en $(u_n, f(u_n))$ coupe l'axe des abscisses en u_{n+1}.



19.3.2 Implémentation

```python
def newton(f, fprime, u0, eps, n_max=100):
    """Renvoie une valeur approchee d'un zero de f par la methode de Newton.
    f       : fonction dont on cherche un zero
    fprime  : derivee de f
    u0      : valeur initiale (doit etre proche de la racine)
    eps     : precision souhaitee (arret quand |f(u)| < eps)
    n_max   : nombre maximal d'iterations (securite contre la divergence)"""
    u = u0
    for _ in range(n_max):
        if abs(f(u)) < eps:
            return u
        u = u - f(u) / fprime(u)
    return u    # renvoie la meilleure approximation apres n_max iterations
```

Exemple : $\sqrt{2}$, racine de $f(x) = x^2 - 2$, $f'(x) = 2x$.

```python
def f(x):
    return x**2 - 2

def fprime(x):
    return 2 * x

print(newton(f, fprime, 2.0, 1e-10))    # 1.4142135623730951
```

19.3.3 Convergence

	Dichotomie	Newton
Conditions	$f(a) \cdot f(b) < 0$	$f'(u_0) \neq 0$, u_0 proche de la racine
Convergence	linéaire : $\sim \log_2\left(\dfrac{b-a}{\varepsilon}\right)$ itérations	quadratique : très rapide
Robustesse	toujours converge	peut diverger si u_0 mal choisi
Dérivée requise	non	oui

La convergence **quadratique** signifie que le nombre de décimales correctes double à chaque itération. En pratique, 10 itérations de Newton donnent souvent une précision de 10^{-10} contre 60 itérations pour la dichotomie.

Comparaison sur $f(x) = x^2 - 2$, $u_0 = 2$, précision 10^{-10} :

```python
# Newton : affichage des iterees
u = 2.0
for k in range(5):
    u = u - f(u) / fprime(u)
    print(f"k={k}  u={u:.15f}  |f(u)|={abs(f(u)):.2e}")

# k=0  u=1.500000000000000  |f(u)|=2.50e-01
# k=1  u=1.416666666666667  |f(u)|=6.94e-03
```

```
# k=2   u=1.414215686274510   |f(u)|=6.01e-06
# k=3   u=1.414213562374690   |f(u)|=4.51e-12
# k=4   u=1.414213562373095   |f(u)|=0.00e+00
```

19.4 Méthode d'Euler

19.4.1 Équations différentielles du premier ordre

On cherche à résoudre numériquement une **équation différentielle** de la forme :

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

où f est une fonction donnée et y_0 la condition initiale. Sauf cas particuliers, y n'a pas d'expression analytique simple.

Propriété : Méthode d'Euler explicite

On subdivise $[t_0, T]$ en n pas de taille $h = \dfrac{T - t_0}{n}$, et on pose $t_k = t_0 + k\,h$.

En approchant $y'(t_k)$ par $\dfrac{y(t_{k+1}) - y(t_k)}{h}$, on obtient le schéma d'Euler :

$$y_{k+1} = y_k + h\, f(t_k, y_k)$$

Géométriquement : à chaque pas, on suit la **tangente** à la courbe solution pendant une durée h.

19.4.2 Implémentation

```
def euler(f, t0, y0, T, n):
    """Resolution approchee de y' = f(t, y), y(t0) = y0, par la methode d'Euler.
    Renvoie deux listes (temps, valeurs) sur [t0, T] avec n pas."""
    h = (T - t0) / n
    t = t0
    y = y0
    temps = [t]
    valeurs = [y]
    for k in range(n):
        y = y + h * f(t, y)
        t = t + h
        temps.append(t)
        valeurs.append(y)
    return temps, valeurs
```

19.4.3 Exemples

Décroissance exponentielle : $y' = -y$, $y(0) = 1$

La solution exacte est $y(t) = e^{-t}$.

```
from math import exp

def f(t, y):
    return -y

temps, valeurs = euler(f, 0, 1, 3, 100)

# Comparaison avec la solution exacte en t = 3
print(valeurs[-1])          # 0.0476...  (Euler, n=100)
print(exp(-3))              # 0.0498...  -- valeur exacte
```

```
11
12 # Avec plus de pas :
13 temps, valeurs = euler(f, 0, 1, 3, 1000)
14 print(valeurs[-1])          # 0.0496...  (encore plus proche)
```

Croissance logistique : $y' = r\,y\left(1 - \dfrac{y}{K}\right)$

Ce modèle (équation de Verhulst) décrit une population limitée par la capacité d'accueil K.

```
1  import matplotlib.pyplot as plt
2
3  r = 0.5     # taux de croissance
4  K = 1000    # capacite d'accueil
5
6  def f_logistique(t, y):
7      return r * y * (1 - y / K)
8
9  temps, valeurs = euler(f_logistique, 0, 10, 30, 500)
10
11 plt.plot(temps, valeurs)
12 plt.xlabel("Temps")
13 plt.ylabel("Population")
14 plt.title("Croissance logistique")
15 plt.show()
```

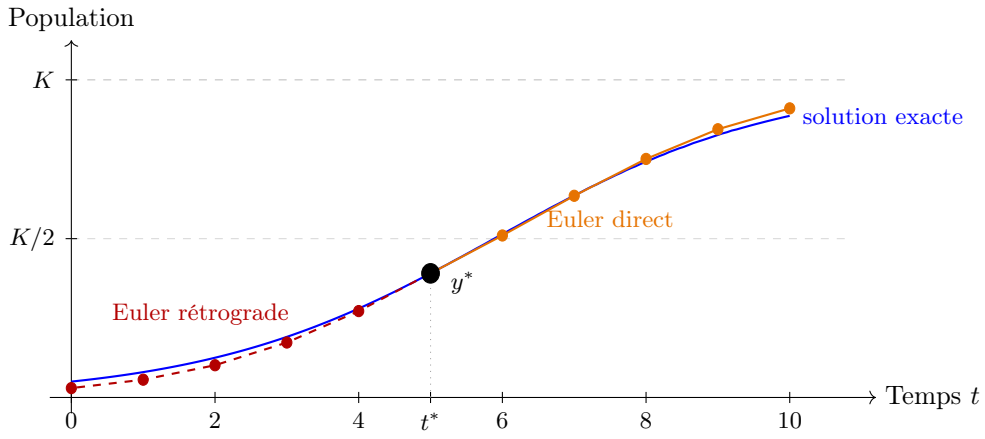
Euler rétrograde (pas négatif)

La même fonction `euler` s'applique avec $T < t_0$: le pas $h = (T - t_0)/n$ est alors **négatif** et l'intégration **remonte le temps**. On peut ainsi, partant d'une condition connue en un instant intermédiaire t^*, reconstruire la courbe solution dans les **deux directions**.

Application : on connaît la population logistique en $t^* = 5$: $y^* \approx 391$ individus. On reconstitue l'évolution sur $[0, 10]$ en lançant Euler à droite *et* à gauche de ce point.

```
1  r, K = 0.5, 1000
2  y_star = 391.0    # population connue en t* = 5
3
4  def f_logistique(t, y):
5      return r * y * (1 - y / K)
6
7  # Euler vers l'avenir : t* -> 10  (h > 0)
8  temps_av,  val_av  = euler(f_logistique, 5, y_star, 10, 200)
9
10 # Euler vers le passe : t* -> 0   (h < 0  car T < t0)
11 temps_ret, val_ret = euler(f_logistique, 5, y_star,  0, 200)
12
13 import matplotlib.pyplot as plt
14 plt.plot(temps_av,  val_av,         label="Euler vers l'avenir")
15 plt.plot(temps_ret, val_ret, '--', label="Euler vers le passe")
16 plt.scatter([5], [y_star], color='black', zorder=5, label="condition en $t^*$")
17 plt.xlabel("Temps")
18 plt.ylabel("Population")
19 plt.legend()
20 plt.show()
```

La figure ci-dessous illustre les deux polygones avec $n = 5$ pas (grossiers) depuis le point central ($t^* = 5,\ y^* = 391$) :



Avec $h < 0$, la formule $y_{k+1} = y_k + h\, f(t_k, y_k)$ reste inchangée. Les deux polygones, issus du même point y^*, reconstituent la courbe logistique dans les deux sens et convergent vers la solution exacte quand $n \to \infty$.

19.4.4 Précision et pas de temps

La méthode d'Euler est d'**ordre 1** : l'erreur globale est en $O(h)$. Diviser h par 2 (doubler n) divise l'erreur par 2.

Paramètre	**Effet sur la précision**
h petit (grand n)	meilleure approximation, mais plus de calculs
h grand (petit n)	rapide, mais peut diverger ou être très imprécis

Remarque : pour certaines équations dites *raides*, un pas h trop grand rend la méthode d'Euler instable. Des méthodes d'ordre supérieur (Runge-Kutta) sont alors préférables.

★ **Complément** —

Utiliser `scipy.integrate.odeint`

Le module `scipy.integrate` fournit la fonction `odeint`, bien plus précise que la méthode d'Euler (elle utilise un algorithme d'ordre élevé à pas adaptatif). On ne cherche pas à comprendre la méthode utilisée, mais à savoir l'appeler.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as si
```

Convention importante : `odeint` attend une fonction `F(y, t)` (y **avant** t, contrairement à notre fonction `euler`). **Équation scalaire :** $y' = f(y,t)$, $y(t_0) = y_0$

```
# Exemple : y' = 2*y + t,  y(0) = 1,  sur [0, 2]
def F(y, t):
    return 2*y + t

t = np.linspace(0, 2, 1000)     # tableau des instants
y = si.odeint(F, [1], t)        # y[k] contient la solution en t[k]

plt.plot(t, y)
plt.xlabel("t")
plt.ylabel("y")
plt.show()
```

`odeint` renvoie un tableau numpy de forme `(n, 1)` pour une équation scalaire. Pour tracer, on peut utiliser `y` directement. **Système d'équations :** $X'(t) = F(X(t), t)$ avec $X = (x, y)$

```
# Modele de Lotka-Volterra (proies x, predateurs y)
# x' = (a - b*y)*x
# y' = -(c - d*x)*y
def F(Y, t):
    a, b, c, d = 0.8, 0.4, 0.9, 0.1
    x = Y[0]
    y = Y[1]
    return [(a - b*y)*x, -(c - d*x)*y]

t = np.linspace(0, 30, 2000)
Y = si.odeint(F, [5, 3], t)    # Y[k] = [x_k, y_k]

x = Y[:, 0]    # composante x pour tous les instants
y = Y[:, 1]    # composante y pour tous les instants

plt.plot(t, x, label="proies")
plt.plot(t, y, label="predateurs")
plt.legend()
plt.show()
```

Équation d'ordre 2 : on réduit à un système en posant $v = y'$.

```
# Pendule : theta'' + w0^2 * sin(theta) = 0
# Systeme : Y = [theta, v]  avec  theta' = v,  v' = -w0^2 * sin(theta)
def F(Y, t):
    theta = Y[0]
    v     = Y[1]
    w0 = 2 * np.pi
    return [v, -w0**2 * np.sin(theta)]

t = np.linspace(0, 5, 1000)
Y = si.odeint(F, [0.3, 0], t)    # theta(0) = 0.3 rad, theta'(0) = 0
plt.plot(t, Y[:, 0])
plt.xlabel("t")
plt.ylabel("theta")
plt.show()
```

★ **Complément** —

Méthode de Simpson

La **méthode de Simpson** approche f sur chaque paire de sous-intervalles $[x_{2k}, x_{2k+2}]$ par une parabole passant par les trois points $(x_{2k}, f(x_{2k}))$, $(x_{2k+1}, f(x_{2k+1}))$, $(x_{2k+2}, f(x_{2k+2}))$. Elle nécessite un nombre pair de sous-intervalles.

$$S = \frac{h}{3}\left(f(a) + f(b) + 4\sum_{k \text{ impair}} f(x_k) + 2\sum_{k \text{ pair, } 0<k<n} f(x_k)\right)$$

```python
def integrale_simpson(f, a, b, n):
    """Approximation de l'integrale de f sur [a,b] par la methode de Simpson.
    n doit etre pair."""
    h = (b - a) / n
    somme = f(a) + f(b)
    for k in range(1, n):
        if k % 2 == 1:
            somme = somme + 4 * f(a + k * h)
        else:
            somme = somme + 2 * f(a + k * h)
    return (h / 3) * somme
```

La méthode de Simpson est d'ordre 4 : son erreur est en $O\left(\frac{1}{n^4}\right)$. Elle est nettement plus précise que les méthodes des rectangles et des trapèzes pour un même nombre de points.

20

Statistiques et simulations

Plan du chapitre

20.1 Simuler des lois de probabilité **111**
20.1.1 Rappel : fonctions du module `random` 111
20.1.2 Simuler une loi discrète quelconque 112
20.1.3 Simuler une loi à densité 112
20.2 Modèles d'urnes **112**
20.2.1 Tirages avec et sans remise 112
20.2.2 Tirage jusqu'à l'obtention d'une valeur 113
20.2.3 Urne avec règle de remplacement 113
20.2.4 Fabrication d'une permutation aléatoire 114
20.3 Estimer par simulation **114**
20.3.1 Loi des grands nombres 114
20.3.2 Estimer une probabilité 115
20.3.3 Estimer une espérance et la loi empirique 115
20.4 Statistiques descriptives **116**
20.4.1 Indicateurs numériques 116
20.4.2 Médiane et quartiles 116
20.4.3 Histogramme 116
20.4.4 Courbe des fréquences cumulées 117
20.4.5 Intervalle empirique à 95 % 117
20.5 Régression linéaire **117**
20.5.1 Droite des moindres carrés 117
20.5.2 Avec NumPy 118

Ce chapitre regroupe les outils Python liés aux probabilités et aux statistiques : simuler des lois, estimer des grandeurs par simulation, décrire une série statistique, et ajuster un modèle par régression.

20.1 Simuler des lois de probabilité

20.1.1 Rappel : fonctions du module `random`

Les fonctions de base du module `random` (vues au chapitre 8) permettent de simuler les lois usuelles :

`random.random()`	loi uniforme sur $[0, 1[$
`random.uniform(a, b)`	loi uniforme sur $[a, b]$
`random.randint(a, b)`	entier aléatoire dans $[\![a, b]\!]$
`random.choice(L)`	élément aléatoire de `L`
`random.shuffle(L)`	mélange `L` en place
`random.sample(L, n)`	tirage sans remise de `n` éléments

20.1.2 Simuler une loi discrète quelconque

Pour simuler une loi donnée par un tableau de valeurs et de probabilités, on utilise la **méthode des probabilités cumulées** : on tire $U \hookrightarrow \mathscr{U}([0,1[)$ et on renvoie la valeur x_i telle que $F(x_{i-1}) \leqslant U < F(x_i)$.

```
import random

def simulation(valeurs, probabilites):
    """Simule la loi qui a valeurs[i] associe probabilites[i].
    La somme des probabilites doit etre egale a 1."""
    r = random.random()
    cumul = 0
    for i in range(len(valeurs)):
        cumul = cumul + probabilites[i]
        if r < cumul:
            return valeurs[i]
    return valeurs[-1]    # cas limite numerique

# Exemple : X vaut 1 avec p=0.3, 2 avec p=0.5, 3 avec p=0.2
valeurs = [1, 2, 3]
proba   = [0.3, 0.5, 0.2]
print(simulation(valeurs, proba))
```

20.1.3 Simuler une loi à densité

Propriété : Méthode de la fonction de répartition inverse

Soit X une variable aléatoire à densité, de fonction de répartition F strictement croissante. Si $U \hookrightarrow \mathscr{U}([0,1[)$, alors $F^{-1}(U)$ suit la même loi que X.

Application : loi exponentielle $\mathscr{E}(\lambda)$ dont la CDF est $F(x) = 1 - e^{-\lambda x}$.

On résout $F(x) = U$: $x = -\dfrac{1}{\lambda}\ln(1-U)$.

```
from math import log
import random

def simul_expo(lam):
    """Simule la realisation d'une loi exponentielle E(lam)."""
    u = random.random()
    return -log(1 - u) / lam

# Estimation de l'esperance (doit etre proche de 1/lam)
N = 100000
moyenne = sum(simul_expo(2) for _ in range(N)) / N
print(moyenne)    # ~0.5  (esperance de E(2) est 1/2)
```

20.2 Modèles d'urnes

Les **modèles d'urnes** sont des expériences aléatoires fondamentales en probabilités : on tire successivement des boules dans une urne, selon des règles variées.

20.2.1 Tirages avec et sans remise

On tire un indice aléatoire avec `int(rd.random() * len(urne))` pour obtenir une boule. Pour le tirage sans remise, on travaille sur une copie de l'urne dont on retire chaque boule tirée.

```
import random as rd

def tirage_avec_remise(urne, n):
    """Renvoie une liste de n tirages successifs avec remise dans urne."""
```

```
    resultats = []
    for _ in range(n):
        i = int(rd.random() * len(urne))
        resultats.append(urne[i])
    return resultats

def tirage_sans_remise(urne, n):
    """Renvoie une liste de n tirages successifs sans remise dans urne."""
    copie = list(urne)
    resultats = []
    for _ in range(n):
        i = int(rd.random() * len(copie))
        resultats.append(copie.pop(i))
    return resultats

urne = ["rouge"]*5 + ["verte"]*10
print(tirage_avec_remise(urne, 3))     # ex. ['verte', 'rouge', 'verte']
print(tirage_sans_remise(urne, 3))     # 3 boules distinctes
```

20.2.2 Tirage jusqu'à l'obtention d'une valeur

```
def tirage_jusque(urne, a):
    """Repete des tirages avec remise jusqu'a obtenir a.
    Renvoie la liste de tous les tirages effectues."""
    resultats = []
    tirage = None
    while tirage != a:
        i = int(rd.random() * len(urne))
        tirage = urne[i]
        resultats.append(tirage)
    return resultats

urne = list(range(1, 7))    # les 6 faces d'un de
tirages = tirage_jusque(urne, 6)
print(f"Nombre de lancers avant le premier 6 : {len(tirages)}")
```

On peut estimer l'espérance du nombre de tirages par simulation :

```
N = 100000
moyenne_attente = sum(len(tirage_jusque(urne, 6)) for _ in range(N)) / N
print(moyenne_attente)    # ~6.0  (esperance theorique : 1/p = 6)
```

20.2.3 Urne avec règle de remplacement

Certains modèles d'urnes ont une composition qui évolue au fil des tirages. On les simule en modifiant la liste `urne` à chaque tirage.

Exemple : une urne contient 7 boules vertes et 5 boules rouges. À chaque tirage :

- si on obtient une boule verte, on la remet dans l'urne ;
- si on obtient une boule rouge, on remet **deux** boules rouges dans l'urne.

```
def experience_urne_polya(nb_tirages):
    """Simule nb_tirages dans l'urne avec regle de remplacement.
    Renvoie la liste des boules tirees."""
    urne = ["verte"]*7 + ["rouge"]*5
    resultats = []
    for _ in range(nb_tirages):
        i = int(rd.random() * len(urne))
        boule = urne[i]
        resultats.append(boule)
        if boule == "rouge":
```

```
            urne.append("rouge")    # on ajoute une rouge supplementaire
    return resultats

# Estimation des probabilites demandees
N = 100000
tirages = [experience_urne_polya(4) for _ in range(N)]

p_toutes_rouges = sum(1 for t in tirages if all(b == "rouge" for b in t)) / N
p_toutes_vertes = sum(1 for t in tirages if all(b == "verte" for b in t)) / N
p_plus_rouges   = sum(1 for t in tirages
                      if t.count("rouge") > t.count("verte")) / N

print(f"P(toutes rouges) ~ {p_toutes_rouges:.4f}")
print(f"P(toutes vertes) ~ {p_toutes_vertes:.4f}")
print(f"P(plus rouges)   ~ {p_plus_rouges:.4f}")
```

20.2.4 Fabrication d'une permutation aléatoire

Une **permutation aléatoire** est un mélange uniforme de la liste initiale : chaque arrangement est équiprobable. L'algorithme de **Fisher-Yates** le réalise en un seul parcours de la liste : à l'étape i (de la fin vers le début), on échange l'élément d'indice i avec un élément d'indice j choisi aléatoirement dans $\{0, 1, \ldots, i\}$.

```
def permutation_aleatoire(L):
    """Renvoie une copie de L melangee aleatoirement (Fisher-Yates)."""
    p = list(L)
    n = len(p)
    for i in range(n - 1, 0, -1):
        j = int(rd.random() * (i + 1))    # j dans {0, 1, ..., i}
        p[i], p[j] = p[j], p[i]
    return p

print(permutation_aleatoire([1, 2, 3, 4, 5]))
# ex. [3, 1, 5, 2, 4]
```

Application : simuler un tirage sans remise revient à prendre les `n` premiers éléments d'une permutation aléatoire de l'urne.

```
def tirage_sans_remise_v2(urne, n):
    """Autre implementation du tirage sans remise via permutation."""
    p = permutation_aleatoire(urne)
    return p[:n]
```

20.3 Estimer par simulation

20.3.1 Loi des grands nombres

Propriété : Loi des grands nombres

Lorsqu'on répète un grand nombre de fois une expérience aléatoire :
- les **fréquences** de réalisation des événements convergent vers leurs **probabilités** ;
- la **moyenne** des réalisations d'une variable aléatoire converge vers son **espérance**.

C'est la base de toute estimation par simulation.

Remarque : cette approche — répéter un grand nombre de fois une expérience aléatoire pour estimer une probabilité ou une espérance — est connue sous le nom de **méthode de Monte-Carlo**. En physique, elle est utilisée notamment pour évaluer les incertitudes de type A.

20.3.2 Estimer une probabilité

```python
import random

def estime_proba(experience, N):
    """Estime par simulation la probabilite que experience() renvoie True."""
    succes = 0
    for _ in range(N):
        if experience():
            succes = succes + 1
    return succes / N

# Exemple : probabilite d'obtenir deux rouges en tirant 2 boules
# dans une urne de 5 rouges et 10 vertes (avec remise)
def experience():
    urne = ["rouge"]*5 + ["verte"]*10
    return random.choice(urne) == "rouge" and random.choice(urne) == "rouge"

print(estime_proba(experience, 100000))   # ~0.111  (theorique : (5/15)^2)
```

20.3.3 Estimer une espérance et la loi empirique

```python
def estime_esperance(simul_X, N):
    """Estime l'esperance de X par la moyenne de N simulations."""
    return sum(simul_X() for _ in range(N)) / N

def affiche_loi_empirique(simul_X, N):
    """Affiche la frequence empirique de chaque valeur sur N simulations."""
    resultats = [simul_X() for _ in range(N)]
    # Valeurs distinctes (sans dictionnaire)
    valeurs = []
    for v in resultats:
        if v not in valeurs:
            valeurs.append(v)
    valeurs.sort()
    for v in valeurs:
        print(f"P(X={v}) ~ {resultats.count(v) / N:.4f}")

# Exemple : X = nombre de 6 en 10 lancers de de
def simul_X():
    return sum(1 for _ in range(10) if random.randint(1, 6) == 6)

print(estime_esperance(simul_X, 100000))   # ~1.667  (theorique : 10/6)
affiche_loi_empirique(simul_X, 10000)
```

> ★ **Complément —**
> Avec un dictionnaire (chapitre 14), on peut renvoyer directement la loi empirique sous forme de tableau de fréquences :

```python
def loi_empirique(simul_X, N):
    """Renvoie le dictionnaire {valeur: frequence} sur N simulations."""
    freq = {}
    for _ in range(N):
        v = simul_X()
        freq[v] = freq.get(v, 0) + 1
    return {v: freq[v] / N for v in freq}
```

20.4 Statistiques descriptives

20.4.1 Indicateurs numériques

```python
def moyenne(L):
    """Renvoie la moyenne de la liste L."""
    return sum(L) / len(L)

def variance(L):
    """Renvoie la variance de la liste L."""
    m = moyenne(L)
    return sum((x - m)**2 for x in L) / len(L)

def ecart_type(L):
    """Renvoie l'ecart-type de la liste L."""
    return variance(L)**0.5

# Exemple : serie de 10000 simulations de la loi N(0,1) via numpy
# np.random.normal(mu, sigma, n) renvoie un tableau de n valeurs N(mu, sigma)
import numpy as np
L = list(np.random.normal(0, 1, 10000))
print(f"Moyenne   : {moyenne(L):.4f}")      # ~0
print(f"Ecart-type: {ecart_type(L):.4f}")   # ~1
```

20.4.2 Médiane et quartiles

La **médiane** est la valeur qui partage la série en deux moitiés égales. Les **quartiles** Q_1, Q_2 (médiane) et Q_3 partagent la série en quatre quarts.

```python
def mediane(L):
    """Renvoie la mediane de la liste L."""
    L_tri = sorted(L)
    n = len(L_tri)
    if n % 2 == 1:
        return L_tri[n // 2]
    return (L_tri[n // 2 - 1] + L_tri[n // 2]) / 2

def quartiles(L):
    """Renvoie (Q1, Q2, Q3) de la liste L."""
    L_tri = sorted(L)
    n = len(L_tri)
    Q2 = mediane(L_tri)
    Q1 = mediane(L_tri[:n // 2])
    Q3 = mediane(L_tri[(n + 1) // 2:])
    return Q1, Q2, Q3

import numpy as np
L = list(np.random.normal(0, 1, 1000))
print(f"Mediane : {mediane(L):.3f}")
Q1, Q2, Q3 = quartiles(L)
print(f"Q1={Q1:.3f}  Q2={Q2:.3f}  Q3={Q3:.3f}")
```

20.4.3 Histogramme

Un **histogramme** représente la distribution des données en regroupant les valeurs par intervalles (**classes**). On utilise `plt.hist` :

```python
import matplotlib.pyplot as plt

# Histogramme d'une serie de donnees
plt.hist(L, bins=30, edgecolor="black")
plt.xlabel("Valeurs")
plt.ylabel("Effectifs")
```

```
plt.title("Histogramme")
plt.show()

# Avec density=True, l'aire totale vaut 1 (histogramme de densite)
plt.hist(L, bins=30, density=True, edgecolor="black")
plt.show()
```

20.4.4 Courbe des fréquences cumulées

La **courbe des fréquences cumulées** (ou fonction de répartition empirique) trace, pour chaque valeur x, la proportion de données inférieures ou égales à x.

```
import matplotlib.pyplot as plt

def trace_freq_cumulees(L, titre=""):
    """Trace la courbe des frequences cumulees de la liste L."""
    L_tri = sorted(L)
    n = len(L_tri)
    freq_cum = [(i + 1) / n for i in range(n)]
    plt.plot(L_tri, freq_cum)
    plt.xlabel("Valeurs")
    plt.ylabel("Frequences cumulees")
    plt.title(titre)
    plt.grid()
    plt.show()

trace_freq_cumulees(L, "Loi N(0,1)")
```

20.4.5 Intervalle empirique à 95 %

Propriété : Intervalle $[\bar{x} \pm 2s]$

Pour une série statistique de moyenne $\bar{x}$ et d'écart-type s, l'intervalle $[\bar{x} - 2s,\ \bar{x} + 2s]$ contient en général autour de 95 % des données (exact pour la loi normale, approximatif pour d'autres lois).

```
def proportion_dans_intervalle(L, a, b):
    """Renvoie la proportion d'elements de L dans [a, b]."""
    return sum(1 for x in L if a <= x <= b) / len(L)

m = moyenne(L)
s = ecart_type(L)
p = proportion_dans_intervalle(L, m - 2*s, m + 2*s)
print(f"Proportion dans [m-2s, m+2s] : {p:.3f}")    # ~0.954
```

20.5 Régression linéaire

20.5.1 Droite des moindres carrés

On dispose de n points (x_i, y_i) et on cherche la droite $y = ax + b$ qui minimise $\sum_{i=1}^{n} (y_i - ax_i - b)^2$.

Propriété : Coefficients de la droite de régression

Les coefficients optimaux sont :

$$a = \frac{\overline{xy} - \bar{x}\,\bar{y}}{\overline{x^2} - \bar{x}^2} = \frac{\mathrm{cov}(X,Y)}{\mathrm{var}(X)} \qquad \text{et} \qquad b = \bar{y} - a\,\bar{x}$$

où $\bar{x} = \frac{1}{n}\sum x_i$, $\bar{y} = \frac{1}{n}\sum y_i$, $\overline{xy} = \frac{1}{n}\sum x_i y_i$, $\overline{x^2} = \frac{1}{n}\sum x_i^2$.

Remarque : Vous trouverez plusieurs démonstrations dans le cours de mathématiques.

```python
def regression_lineaire(X, Y):
    """Renvoie (a, b) de la droite de regression y = a*x + b."""
    n = len(X)
    moy_x  = sum(X) / n
    moy_y  = sum(Y) / n
    moy_xy = sum(X[i]*Y[i] for i in range(n)) / n
    moy_x2 = sum(X[i]**2   for i in range(n)) / n
    a = (moy_xy - moy_x * moy_y) / (moy_x2 - moy_x**2)
    b = moy_y - a * moy_x
    return a, b

X = [2,  4,  6,  8,  10, 12, 14, 16, 18, 20]
Y = [8, 13, 20, 27,  32, 38, 44, 50, 56, 62]
a, b = regression_lineaire(X, Y)
print(f"y = {a:.3f} x + {b:.3f}")    # y = 3.012 x + 1.867
```

20.5.2 Avec NumPy

```python
import numpy as np
import matplotlib.pyplot as plt

X = np.array([2,  4,  6,  8, 10, 12, 14, 16, 18, 20], dtype=float)
Y = np.array([8, 13, 20, 27, 32, 38, 44, 50, 56, 62], dtype=float)

# np.polyfit(X, Y, 1) renvoie [a, b] pour y = a*x + b
coeffs = np.polyfit(X, Y, 1)
a, b = coeffs[0], coeffs[1]
print(f"y = {a:.3f} x + {b:.3f}")

# Trace
plt.plot(X, Y, "o", label="donnees")
plt.plot(X, a*X + b, label=f"y = {a:.2f}x + {b:.2f}")
plt.legend()
plt.show()
```

★ **Complément** —

Illustration du théorème central limite

Soit $(X_k)_{k\geqslant 1}$ une suite de variables aléatoires indépendantes de même loi, d'espérance μ et de variance σ^2. La moyenne empirique $M_n = \frac{1}{n}\sum_{k=1}^{n} X_k$ vérifie :

$$\frac{M_n - \mu}{\sigma/\sqrt{n}} \xrightarrow[n\to+\infty]{\mathscr{L}} \mathscr{N}(0,1)$$

On illustre ce résultat par simulation : pour différentes valeurs de n, on calcule un grand nombre de réalisations de M_n et on trace l'histogramme normalisé.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm

def simul_X():
    return np.random.randint(0, 3)    # loi U({0,1,2})

mu    = 1.0      # esperance de U({0,1,2})
sigma = (2/3)**0.5   # ecart-type

for n in [1, 5, 30]:
    Mn = [sum(simul_X() for _ in range(n)) / n for _ in range(5000)]
    Zn = [(m - mu) / (sigma / n**0.5) for m in Mn]    # normalisation
    plt.figure()
    plt.hist(Zn, bins=40, density=True, label=f"n={n}")
    t = np.linspace(-4, 4, 200)
    plt.plot(t, norm.pdf(t), "r-", label="N(0,1)")
    plt.legend()
    plt.title(f"Distribution de la moyenne normalisee (n={n})")
    plt.show()
```

Pour $n = 1$ la distribution est uniforme, pour $n = 30$ elle est déjà très proche de la gaussienne — quelle que soit la loi de départ.

21

Intervalles de fluctuation et tests statistiques

Plan du chapitre

21.1 Intervalle de fluctuation d'une fréquence **121**
21.1.1 Fréquence observée et fluctuation 121
21.1.2 Vérification par simulation 122
21.2 Intervalle de confiance pour une proportion **122**
21.2.1 Problème inverse 122
21.2.2 Implémentation 123
21.2.3 Vérification par simulation 123
21.3 Test de conformité d'une proportion **123**
21.3.1 Cadre du test 123
21.3.2 Implémentation 124
21.3.3 Simulation du risque de première espèce 124
21.4 Applications **124**
21.4.1 Génétique mendélienne 124
21.4.2 Estimation d'une fréquence allélique 125
21.4.3 Synthèse : démarche d'un test 125

Ce chapitre présente les outils permettant de prendre des décisions statistiques à partir d'observations : intervalles de fluctuation, intervalles de confiance, et tests de conformité d'une proportion. Ces notions sont illustrées par simulation Python, en lien direct avec le chapitre précédent.

21.1 Intervalle de fluctuation d'une fréquence

21.1.1 Fréquence observée et fluctuation

Lorsqu'on répète n fois une épreuve de Bernoulli de paramètre p (probabilité de succès), le nombre de succès X suit une loi binomiale $\mathscr{B}(n, p)$. La **fréquence observée** $f_n = X/n$ est une variable aléatoire qui fluctue autour de p.

Propriété : Intervalle de fluctuation à 95 %

Soit f_n la fréquence de succès obtenue en répétant n fois une épreuve de Bernoulli de paramètre p. Pour $n \geqslant 30$ et $0{,}05 \leqslant p \leqslant 0{,}95$:

$$P\left(|f_n - p| \leqslant \frac{1}{\sqrt{n}}\right) \approx 0{,}95$$

L'intervalle $\left[p - \frac{1}{\sqrt{n}},\ p + \frac{1}{\sqrt{n}}\right]$ est l'**intervalle de fluctuation** de f_n au niveau 95 %.

21.1.2 Vérification par simulation

On simule N expériences de n épreuves de Bernoulli et on vérifie que f_n tombe dans l'intervalle de fluctuation dans $\approx 95\,\%$ des cas :

```python
import random
from math import sqrt

def simule_fn(n, p):
    """Renvoie la frequence de succes sur n tirages de Bernoulli(p)."""
    succes = sum(1 for _ in range(n) if random.random() < p)
    return succes / n

def verifie_fluctuation(n, p, N):
    """Estime la probabilite que fn soit dans l'intervalle de fluctuation."""
    borne = 1 / sqrt(n)
    dans_intervalle = 0
    for _ in range(N):
        fn = simule_fn(n, p)
        if abs(fn - p) <= borne:
            dans_intervalle += 1
    return dans_intervalle / N

print(verifie_fluctuation(100, 0.5, 100000))    # ~0.95
print(verifie_fluctuation(400, 0.3, 100000))    # ~0.95
```

Largeur de l'intervalle : $\dfrac{2}{\sqrt{n}}$. Elle diminue quand n augmente : plus on observe, plus la fréquence est concentrée autour de p.

n	25	100	400	1600
Largeur $2/\sqrt{n}$	0,40	0,20	0,10	0,05

Pour diviser la largeur par 2, il faut **multiplier n par 4**.

21.2 Intervalle de confiance pour une proportion

21.2.1 Problème inverse

En pratique, p est **inconnu** : on observe f_n et on cherche à encadrer p. On résout le problème à l'envers.

Propriété : Intervalle de confiance à 95 %

À partir d'une fréquence observée f_n sur n observations, on construit l'**intervalle de confiance** au niveau 95 % pour p :

$$I_c = \left[f_n - \frac{1}{\sqrt{n}},\ f_n + \frac{1}{\sqrt{n}} \right]$$

Cet intervalle contient la vraie valeur p dans environ 95 % des expériences possibles.

Interprétation correcte : ce n'est pas p qui est aléatoire, c'est l'intervalle I_c (qui dépend de f_n). Affirmer que I_c contient p à 95 % signifie que, si on répétait l'expérience un grand nombre de fois, 95 % des intervalles ainsi construits contiendraient p.

21.2.2 Implémentation

```python
def intervalle_confiance(fn, n):
    """Renvoie (borne_inf, borne_sup) de l'IC a 95% pour une proportion p.
    fn : frequence observee  /  n : nombre d'observations"""
    borne = 1 / sqrt(n)
    return fn - borne, fn + borne

# Exemple : sur 200 individus, 94 presentent un caractere
n  = 200
fn = 94 / 200
ic = intervalle_confiance(fn, n)
print(f"Frequence observee : {fn:.3f}")
print(f"IC a 95% : [{ic[0]:.3f}, {ic[1]:.3f}]")
# Frequence observee : 0.470
# IC a 95% : [0.400, 0.541]
```

21.2.3 Vérification par simulation

```python
def verifie_confiance(n, p, N):
    """Estime la probabilite que l'IC construite contienne p."""
    contient_p = 0
    borne = 1 / sqrt(n)
    for _ in range(N):
        fn = simule_fn(n, p)
        if fn - borne <= p <= fn + borne:
            contient_p += 1
    return contient_p / N

print(verifie_confiance(100, 0.4, 100000))    # ~0.95
print(verifie_confiance(400, 0.7, 100000))    # ~0.95
```

21.3 Test de conformité d'une proportion

21.3.1 Cadre du test

Un **test de conformité** permet de décider, à partir de données observées, si une valeur théorique p_0 est compatible avec les observations.

Définition : Hypothèses et règle de décision

Hypothèse nulle H_0 : $p = p_0$ (valeur théorique à tester)
Hypothèse alternative H_1 : $p \neq p_0$ (test bilatéral)

Règle de décision au niveau 5 % : on rejette H_0 si f_n est **hors** de l'intervalle de fluctuation de p_0 :

$$f_n \notin \left[p_0 - \frac{1}{\sqrt{n}},\ p_0 + \frac{1}{\sqrt{n}} \right]$$

Propriété : Risque de première espèce

Le **risque de première espèce** α est la probabilité de rejeter H_0 alors qu'elle est vraie :

$$\alpha = P\left(f_n \notin \left[p_0 - \frac{1}{\sqrt{n}},\ p_0 + \frac{1}{\sqrt{n}} \right] \;\middle|\; p = p_0 \right) \approx 5\,\%$$

On dit que le test est au **niveau** 5 %. C'est une erreur inévitable : on accepte de se tromper une fois sur vingt.

21.3.2 Implémentation

```python
def test_conformite(fn, n, p0):
    """Teste H0 : p = p0 au niveau 5%.
    Renvoie True si on rejette H0, False sinon."""
    borne = 1 / sqrt(n)
    return abs(fn - p0) > borne

# Exemple : 63 succes sur 100 epreuves -- la proportion vaut-elle 0.5 ?
n  = 100
fn = 63 / 100
p0 = 0.5
if test_conformite(fn, n, p0):
    print("On rejette H0 : la proportion semble differente de 0.5")
else:
    print("On ne rejette pas H0")
# -> On rejette H0  (0.63 est hors de [0.40, 0.60])
```

21.3.3 Simulation du risque de première espèce

```python
def risque_alpha(n, p0, N):
    """Estime le risque de rejeter H0 a tort (quand p = p0 est vraie)."""
    rejets = 0
    for _ in range(N):
        fn = simule_fn(n, p0)
        if test_conformite(fn, n, p0):
            rejets += 1
    return rejets / N

print(risque_alpha(100, 0.5, 100000))   # ~0.05
print(risque_alpha(400, 0.3, 100000))   # ~0.05
```

21.4 Applications

21.4.1 Génétique mendélienne

Dans un croisement entre deux hétérozygotes pour un caractère à dominance complète, la loi de Mendel prédit une proportion $p_0 = 3/4$ d'individus présentant le phénotype dominant.

```python
# Observation : 740 individus dominants sur 1000
n  = 1000
fn = 740 / 1000
p0 = 3 / 4

ic = intervalle_confiance(fn, n)
print(f"IC a 95% : [{ic[0]:.4f}, {ic[1]:.4f}]")
# IC a 95% : [0.7083, 0.7717]

if test_conformite(fn, n, p0):
    print(f"Au seuil de 95 %, l'échantillon est incompatible avec la loi de
        Mendel (p0 = {p0})")
    print("(on rejette H0 : l'écart observé est trop important pour être dû au
        seul hasard d'échantillonnage)")
else:
    print(f"Au seuil de 95 %, l'échantillon est compatible avec la loi de Mendel
         (p0 = {p0})")
    print("(on ne rejette pas H0 : l'écart observé est compatible avec un simple
         effet du hasard d'échantillonnage)")
```

21.4.2 Estimation d'une fréquence allélique

Dans une population, on génotype n individus pour un locus biallélique (allèles A et a). On estime la fréquence de l'allèle A et on construit un intervalle de confiance :

```python
# Sur 500 individus genotypes, 320 portent l'allele A
n  = 500
fn = 320 / 500

ic = intervalle_confiance(fn, n)
print(f"Frequence estimee de A : {fn:.3f}")
print(f"IC a 95% : [{ic[0]:.3f}, {ic[1]:.3f}]")
# Frequence estimee de A : 0.640
# IC a 95% : [0.595, 0.685]
```

21.4.3 Synthèse : démarche d'un test

① Formuler les hypothèses : $H_0 : p = p_0$ contre $H_1 : p \neq p_0$.

② Calculer la fréquence observée f_n et l'intervalle de fluctuation $\left[p_0 - \dfrac{1}{\sqrt{n}},\ p_0 + \dfrac{1}{\sqrt{n}}\right]$.

③ Conclure :

- si f_n est **dans** l'intervalle : on ne rejette pas H_0 (résultat compatible avec p_0 au niveau 5 %) ;
- si f_n est **hors** de l'intervalle : on rejette H_0 (résultat significativement différent de p_0).

Attention : ne pas rejeter H_0 ne signifie pas que H_0 est vraie, mais seulement que les données ne permettent pas de la contredire au niveau 5 %.

★ **Complément —**

p-valeur

La **p-valeur** est la probabilité, sous H_0, d'obtenir un écart $|f_n - p_0|$ au moins aussi grand que celui observé. Plus elle est petite, plus le résultat est improbable sous H_0.

```python
def p_valeur(fn, n, p0, N=100000):
    """Estime par simulation la p-valeur du test bilateral H0 : p = p0."""
    ecart_obs = abs(fn - p0)
    aussi_extreme = sum(
        1 for _ in range(N) if abs(simule_fn(n, p0) - p0) >= ecart_obs
    )
    return aussi_extreme / N

print(p_valeur(0.63, 100, 0.5))   # ~0.009  -> tres significatif
print(p_valeur(0.55, 100, 0.5))   # ~0.32   -> non significatif
```

Convention : on rejette H_0 au niveau 5 % si la p-valeur est inférieure à 0,05.

22

Annexe : numpy pour l'algèbre linéaire

Plan du chapitre

22.1 Rappel : tableaux à deux dimensions 127
22.2 Produit terme à terme et produit matriciel 128
22.3 Extraire des lignes et des colonnes 129
22.4 Déterminant, inverse, valeurs propres 129
22.5 Le sous-module numpy.random : incertitudes-type en physique 130

★ **Ce chapitre est entièrement hors programme des concours.**

Il approfondit l'usage de numpy sur les tableaux à deux dimensions, déjà introduit au chapitre 8. Ces outils sont utiles pour les exercices d'oral (Agro, G2E) et pour les TIPE ; le sous-module numpy.random est en outre très utilisé en physique-chimie pour l'évaluation des incertitudes-type par simulation.

22.1 Rappel : tableaux à deux dimensions

Un tableau numpy à deux dimensions représente une matrice. On le crée à partir d'une liste de listes, chaque sous-liste étant une ligne :

```
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])

print(A.shape)    # (2, 3)  -> 2 lignes, 3 colonnes
```

Propriété : `shape` contre `size`

`A.shape` (ou `np.shape(A)`)	**tuple** `(lignes, colonnes)` : la forme du tableau.
`A.size` (ou `np.size(A)`)	**entier** : le nombre **total** d'éléments (lignes × colonnes).
`np.size(A, axis)`	**entier** : nombre d'éléments **selon un axe** (`axis=0` : lignes, `axis=1` : colonnes) — équivalent à `A.shape[axis]`.

```
B = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8]])

print(B.shape)          # (2, 4)    -> 2 lignes, 4 colonnes
print(np.shape(B))      # (2, 4)    -> syntaxe fonction, meme resultat

print(B.size)            # 8         -> nombre total d'elements
print(np.size(B))        # 8         -> syntaxe fonction, meme resultat

print(np.size(B, 0))     # 2         -> nombre de lignes, comme B.shape[0]
print(np.size(B, 1))     # 4         -> nombre de colonnes, comme B.shape[1]
```

Attention : l'**attribut** `A.size` donne toujours le nombre total d'éléments. C'est la **fonction** `np.size(A, axis)`, appelée avec un axe, qui redonne le nombre de lignes ou de colonnes séparément — sans argument `axis`, elle revient au même que `A.size`. `np.size(B)` (sans axe) est notamment pratique pour calculer une moyenne « à la main » : `np.sum(B) / np.size(B)`.

22.2 Produit terme à terme et produit matriciel

Définition : `A * B` contre `A @ B`

`A * B`	**produit terme à terme** (produit de Hadamard) : `A` et `B` doivent avoir la même forme.
`A @ B`	**produit matriciel** usuel : le nombre de colonnes de `A` doit égaler le nombre de lignes de `B`.
`np.dot(A, B)`	équivalent de `A @ B` pour des tableaux 2D — syntaxe plus ancienne, antérieure à l'opérateur `@` (introduit en Python 3.5).

```
import numpy as np

A = np.array([[1, 2],
              [3, 4]])
B = np.array([[5, 6],
              [7, 8]])

print(A * B)
# [[ 5 12]
#  [21 32]]      -> terme a terme : (1*5, 2*6), (3*7, 4*8)

print(A @ B)
# [[19 22]
#  [43 50]]      -> produit matriciel usuel

print(np.dot(A, B))
# [[19 22]
#  [43 50]]      -> meme resultat, syntaxe equivalente a @
```

★ **Piège classique**

Pour deux matrices carrées de même taille, `A * B` et `A @ B` sont **toutes les deux valides syntaxiquement**, mais ne calculent pas la même chose : Python ne signale aucune erreur si on se trompe d'opérateur, seul le résultat numérique est faux. Avant d'écrire un produit, il faut donc se demander explicitement : « terme à terme » ou « produit matriciel » ?

22.3 Extraire des lignes et des colonnes

Propriété : Syntaxe de slicing sur un tableau 2D

`A[i, j]`	élément ligne i, colonne j
`A[i, :]`	ligne i entière (tableau 1D)
`A[:, j]`	colonne j entière (tableau 1D)
`A[i1:i2, :]`	bloc des lignes i_1 à $i_2 - 1$
`A[:, j1:j2]`	bloc des colonnes j_1 à $j_2 - 1$
`A[i1:i2, j1:j2]`	sous-matrice
`A[::-1, :]`	lignes dans l'ordre inverse

```
import numpy as np

A = np.array([[ 1,  2,  3,  4],
              [ 5,  6,  7,  8],
              [ 9, 10, 11, 12]])

print(A[1, :])         # [5 6 7 8]          -> ligne d'indice 1
print(A[:, 2])          # [ 3  7 11]         -> colonne d'indice 2
print(A[0:2, 1:3])
# [[2 3]
#  [6 7]]               -> sous-matrice (lignes 0-1, colonnes 1-2)
print(A[-1, :])          # [ 9 10 11 12]      -> derniere ligne
```

Remarque : comme pour les listes (chapitre 13), une slice `numpy` est une **vue** sur le tableau d'origine, et non une copie : modifier une slice modifie aussi le tableau de départ.

```
ligne = A[0, :]
ligne[0] = 99

print(A)
# [[99  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]   -> A a ete modifie, alors qu'on n'a touche qu'a "ligne" !
```

Pour obtenir une copie indépendante, il faut appeler explicitement `A[0, :].copy()`.

22.4 Déterminant, inverse, valeurs propres

Le sous-module `np.linalg` regroupe les opérations classiques d'algèbre linéaire :

`np.transpose(A)`	transposée de `A`
`np.linalg.det(A)`	déterminant
`np.linalg.inv(A)`	inverse (si `A` est inversible)
`np.linalg.eigvals(A)`	valeurs propres seules (plus léger que `eig` si l'on n'a pas besoin des vecteurs propres)
`np.linalg.eig(A)`	valeurs propres **et** vecteurs propres

```python
A = np.array([[1, 2], [3, 4]])

print(np.linalg.det(A))     # -2.0
print(np.linalg.inv(A))     # [[-2.    1. ]
                            #  [ 1.5  -0.5]]

print(np.linalg.eigvals(A))
# [-0.37228132  5.37228132]

valeurs_propres, vecteurs_propres = np.linalg.eig(A)
print(valeurs_propres)
# [-0.37228132  5.37228132]      -> memes valeurs que eigvals
print(vecteurs_propres)
# [[-0.82456484 -0.41597356]
#  [ 0.56576746 -0.90937671]]    -> vecteurs propres normes, en colonnes
```

22.5 Le sous-module `numpy.random` : incertitudes-type en physique

En physique-chimie, une grandeur mesurée x est en général connue par une valeur et une **incertitude-type** $u(x)$. Quand une grandeur S se calcule à partir de plusieurs mesures (par exemple $S = L \times \ell$), on peut estimer son incertitude-type par **simulation de Monte-Carlo** plutôt que par le calcul analytique des dérivées partielles : on tire un grand nombre N de valeurs simulées de chaque grandeur mesurée, on calcule S pour chaque tirage, puis on prend la moyenne et l'écart-type de l'échantillon obtenu.

Propriété : Lois de tirage usuelles

`np.random.seed(n)`	fixe la graine (résultats reproductibles)
`np.random.normal(mu, sigma, N)`	N tirages de loi normale $\mathcal{N}(\mu, \sigma)$
`np.random.uniform(a, b, N)`	N tirages de loi uniforme sur $[a, b]$

La loi normale modélise en général une incertitude de type A (dispersion de mesures répétées), la loi uniforme une incertitude de type B (résolution d'un instrument numérique, donnée constructeur).

Exemple : incertitude-type sur une aire

```python
import numpy as np

np.random.seed(0)

# Mesures et incertitudes-type (donnees d'un TP)
L, u_L = 5.00, 0.02    # longueur (cm), mesures repetees -> loi normale
l, u_l = 3.00, 0.01    # largeur (cm), mesures repetees -> loi normale

N = 100_000
L_sim = np.random.normal(L, u_L, N)
l_sim = np.random.normal(l, u_l, N)

S_sim = L_sim * l_sim    # une aire par tirage : produit terme a terme

print(f"Aire moyenne      : {np.mean(S_sim):.3f} cm^2")
print(f"Incertitude-type : {np.std(S_sim):.3f} cm^2")
```

Remarque : `L_sim` et `l_sim` sont deux tableaux 1D de N tirages indépendants. `L_sim * l_sim` calcule bien, tirage par tirage, le produit terme à terme (une aire simulée par tirage) — c'est l'opérateur `*` qu'il faut utiliser ici, pas `@`, qui n'aurait d'ailleurs pas de sens sur des tableaux 1D pour ce calcul.

Pour une incertitude de type B, par exemple la résolution r d'un afficheur numérique, on ajoute un bruit de loi uniforme sur $[-r/2, r/2]$:

```
r = 0.01   # resolution de l'instrument
bruit = np.random.uniform(-r / 2, r / 2, N)
mesures_simulees = valeur_lue + bruit
```

Cette méthode par simulation évite le calcul analytique des dérivées partielles de la formule de propagation des incertitudes : elle est explicitement admise, et de plus en plus utilisée, pour l'évaluation des incertitudes-type en physique-chimie.

Index

affectation, **10**
aliasing, **72**
anagramme, 94
arbre de recherche, **92**
arrangement, 94

base de données, **80**
Bernoulli
    épreuve de, 121
bool (type), **9**
boucle
    for, **22**
    infinie, 21
    while, **21**

combinaison, 94
comparaison, **12**
compréhension de liste, **31**

def (mot-clé), **25**
dichotomie, **59**
    équation, **61**
dictionnaire, **75**
Dijkstra (algorithme), **98**
diviser pour régner, 67
docstring, **27**

écart-type, **116**
effet de bord, **73**
égalité
    entre flottants, **10**
élagage (arbre de recherche), **93**
elif (instruction), **18**
else (instruction), **17**
ensemble (type), 34
enumerate (fonction), 14
énumération combinatoire, **92**
équation
    résolution numérique, **104**
équation différentielle
    résolution numérique, **106**
Euler (méthode d'), **106**
    rétrograde, 107
extinction (temps d'), 56

f-string, **14**
fichier, **41**
Fisher-Yates (algorithme), **114**
float (type), **9**
flottant
    représentation, **10**
fonction
    définition, **25**
    récursive, **53**
fonction de répartition inverse, **112**
fonction native, **14**
fréquence observée, **121**

graphe, **89**
    arête, 89
    parcours, **91**
    plus court chemin, **98**
    sommet, 89
graphique, **48**

hypothèse nulle, **123**

if (instruction), **17**
import (instruction), **45**
incertitude-type, **130**
indentation, **17**
input (fonction), **13**
instruction conditionnelle, **17**
int (type), **9**
intégration numérique, **102**
intervalle
    empirique à 95%, **117**
intervalle de confiance, **122**
intervalle de fluctuation, **121**

len (fonction), 14
liste, **29**
    de listes, **39**
liste (combinatoire), **94**
loi binomiale, 121
loi des grands nombres, **114**

matplotlib (module), **48**
méthode
    des rectangles, **102**
    des trapèzes, **103**
module, **45**
Monte-Carlo (méthode de), 114, 130
moyenne, **116**
mutabilité, **71**

Newton (méthode de), **104**
nombre pseudo-aléatoire, **47**
numpy (module), **48**
    dot, 128
    eigvals, 129

linalg, **129**
produit matriciel, **128**
produit terme à terme, **128**
random (sous-module), **130**
shape, 128
size, 128
slicing, **129**
tableau 2D, 127

objet
immuable, **71**
mutable, **71**
open (fonction), **41**
opérateur
arithmétique, **11**
comparaison, **12**
logique, **12**

p-uplet, *voir* tuple
paramètre, **25**
permutation aléatoire, **114**
portée (variable), **26**
print (fonction), **13**
processus de branchement, **56**

random (module), **46**
range (fonction), **22**
recherche
dans une liste, **37**
dichotomique, **59**
récursivité, **53**
aléatoire, 56
terminaison, **55**
terminaison probabiliste, 56
régression linéaire, **118**
return (instruction), **26**
risque de première espèce, **123**

set (type), **34**
simulation
loi à densité, **112**
loi de probabilité, **111**
Monte-Carlo, 114
probabiliste, 46
slice, **30**
sorted (fonction), 14
spécification, **27**
SQL, **80**
jointure, **82**
SELECT, **80**
sous-requête, **85**
statistiques descriptives, **116**
str (type), **9**
suite
strictement croissante, 94
sum (fonction), 14

table (SQL), **80**
tableau
2D, **39**
numpy, **48**
test de conformité, **123**
tirage avec remise, **112**
tirage sans remise, **112**
transtypage, **10**
tri, **63**
en place, **63**
fusion, **67**
par comptage, **65**
par insertion, **64**
par sélection, **63**
tuple, **33**
type, **9**

urne
modèle, **112**

valeur de retour, **26**
variable, **10**
globale, **26**
locale, **26**
variance, **116**
variant de boucle, **22**

with (instruction), **41**

zip (fonction), 14