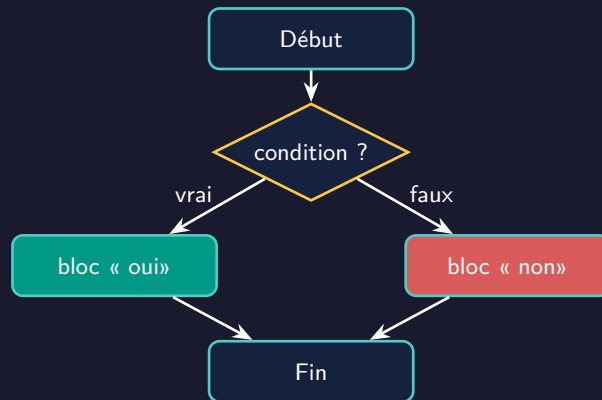


Algorithmique & Python

Conditions & boucles ▪ Fonctions ▪ Listes ▪ Logique



Première ▪ Spécialité Mathématiques ▪ Programme officiel



Table des matières

1	Pourquoi programmer en maths ?	3
1.1	La machine fait le calcul, toi tu fais le raisonnement	3
1.2	Algorithme ou programme ?	3
1.3	L'idée directrice	3
2	L'idée avant le code	4
2.1	Une variable, c'est une boîte avec une étiquette	4
2.2	Choisir, répéter : les deux gestes de base	4
2.3	Découper pour ne pas se noyer : les fonctions	4
3	Le cours complet	5
3.1	Variables, types et affectation	5
3.2	Instructions conditionnelles : choisir	6
3.3	Boucles bornées : répéter un nombre connu de fois	6
3.4	Boucles non bornées : répéter jusqu'à une condition	7
3.5	Fonctions et programmation modulaire	7
3.6	Les listes : la grande nouveauté	8
3.7	Un peu de logique et d'ensembles	10
4	Boîte à outils : réflexes pour le bac	12
5	Exercices	14
6	Problème type prépa	16
7	✓ Corrigés détaillés	17

1 Pourquoi programmer en maths ?

1.1 La machine fait le calcul, toi tu fais le raisonnement

Beaucoup de problèmes mathématiques demandent de **répéter** un calcul des centaines de fois, ou de **tester** une condition sur une longue liste de nombres. À la main, c'est interminable et source d'erreurs. La **programmation** confie cette exécution mécanique à la machine : tu décris une fois pour toutes la **méthode** (l'**algorithme**), et l'ordinateur l'applique sans fatigue ni erreur. Ton travail reste le plus important : **penser** la méthode.

Calculer
sommes, moyennes,
tables de valeurs

Chercher
un seuil, un
maximum, un zéro

Simuler
le hasard, des
échantillons

Organiser
des données
dans des listes

1.2 Algorithme ou programme ?

Un **algorithme** est une suite finie d'instructions non ambiguës qui résout un problème. On peut l'écrire en **langage naturel** (en français, avec le symbole \leftarrow pour « prend la valeur ») ou dans un **langage de programmation** comme **Python**. Le programme, c'est la traduction de l'algorithme dans un langage que la machine comprend.

1.3 L'idée directrice

L'idée directrice :

Tout programme se construit avec **quatre briques** seulement : des **variables** (pour stocker), des **conditions** (pour choisir), des **boucles** (pour répéter), et des **fonctions** (pour réutiliser). La grande nouveauté de Première, ce sont les **listes** : une « boîte » qui range plusieurs valeurs d'un coup. Maîtriser ces briques, c'est pouvoir tout programmer.

Intuition | Pourquoi c'est un chapitre transversal

L'algorithmique n'est pas un chapitre « à part » : elle traverse **toute** l'année. On programme des **suites** (fiche 1), on cherche des **seuils**, on calcule des **espérances** (fiche 10), on simule des **probabilités** (fiche 9). Les **listes** servent partout : tableaux de valeurs, échantillons, séries statistiques. Bien tenir ces outils, c'est se simplifier tous les autres chapitres.

2 L'idée avant le code

2.1 Une variable, c'est une boîte avec une étiquette

Intuition | Stocker pour réutiliser

Une **variable** est une « boîte » dans la mémoire, avec un **nom** (l'étiquette) et une **valeur** (le contenu). L'**affectation** $x = 5$ (ou $x \leftarrow 5$ en langage naturel) signifie : « mets la valeur 5 dans la boîte nommée x ». On peut ensuite **lire** la boîte, ou **remplacer** son contenu : $x = x + 1$ veut dire « prends le contenu actuel, ajoute 1, et remets le résultat dans la même boîte ». Le $=$ de la programmation n'est donc **pas** l'égalité des maths : c'est un **ordre** (« range ça ici »), pas une affirmation.

2.2 Choisir, répéter : les deux gestes de base

Intuition | Si... alors..., et « recommence »

Deux gestes suffisent à donner de l'intelligence à un programme :

- **Choisir** avec une **condition** : « **si** le nombre est pair, **alors** fais ceci, **sinon** fais cela ».
- **Répéter** avec une **boucle** : « **pour** chaque nombre de 1 à 100... » (on sait combien de fois) ou « **tant que** le résultat est trop petit... » (on répète jusqu'à une condition).

Avec « stocker », « choisir » et « répéter », on programme déjà l'immense majorité des algorithmes du lycée.

2.3 Découper pour ne pas se noyer : les fonctions

Intuition | Une tâche complexe = plein de petites tâches

Plutôt qu'un seul long programme, on **découpe** en **fonctions** : chaque fonction fait **une** chose, porte un **nom** clair, reçoit des **entrées** (les paramètres) et **renvoie** un résultat. C'est la **programmation modulaire** : on assemble ensuite ces briques comme des LEGO. Avantages : c'est plus lisible, on peut **tester** chaque brique séparément, et on **réutilise** une fonction autant de fois qu'on veut.

3 Le cours complet

3.1 Variables, types et affectation

Définition | Variable et affectation

Une **variable** associe un **nom** à une **valeur** stockée en mémoire. L'**affectation** donne (ou change) cette valeur :

- en **langage naturel** : $x \leftarrow 5$ (« x prend la valeur 5 »);
- en **Python** : `x = 5`.

✓ Propriété | Les principaux types

Type	Description	Exemple Python
int	nombre entier	<code>n = 7</code>
float	nombre décimal	<code>x = 3.14</code>
str	chaîne de caractères	<code>nom = "Ada"</code>
bool	booléen (vrai/faux)	<code>test = True</code>
list	liste de valeurs	<code>L = [2, 4, 6]</code>

Attention | Le = n'est pas l'égalité des maths

En maths, « $x = x + 1$ » est **faux** (aucun nombre n'égale son successeur). En programmation, `x = x + 1` est un **ordre** parfaitement valide : « calcule $x+1$ et range le résultat dans x ». Pour **tester** une égalité (et obtenir vrai/faux), on utilise un **double** égal : `x == 1`. Confondre = (affecter) et == (tester) est l'erreur n°1 du débutant.

Exemple | Échanger deux variables

Pour échanger les contenus de a et b, on passe par une **variable temporaire** (sinon on écrase une valeur) :

```
1 a = 3
2 b = 8
3 temp = a      # on met de cote l'ancien a
4 a = b         # a recoit b
5 b = temp      # b recoit l'ancien a
6 print(a, b)   # affiche 8 3
```

3.2 Instructions conditionnelles : choisir

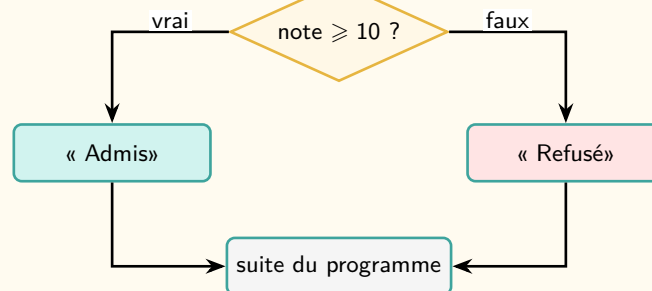
Définition | Conditionnelle if / elif / else

Une **instruction conditionnelle** exécute un bloc **si** une condition est vraie, un autre **sinon**. En Python, l'indentation (le décalage) délimite les blocs.

```
1 if note >= 10:
2     print("Admis")
3 elif note >= 8:
4     print("Rattrapage")
5 else:
6     print("Refuse")
```

Intuition | L'organigramme d'un if

Une condition crée un **embranchement** : selon que le test est vrai ou faux, le programme suit un chemin ou l'autre, puis les chemins se rejoignent.



3.3 Boucles bornées : répéter un nombre connu de fois

Définition | Boucle for

La boucle **for** répète un bloc pour chaque valeur d'une plage. `range(a, b)` parcourt les entiers de `a` à `b-1` (la borne de droite est **exclue**).

```
1 somme = 0
2 for k in range(1, 101):    # k va de 1 à 100
3     somme = somme + k
4 print(somme)               # 5050
```

Attention | `range(a, b)` s'arrête à `b-1`

`range(1, 5)` produit 1, 2, 3, 4 : **quatre** valeurs, pas cinq. Pour parcourir 1 à n **inclus**, on écrit `range(1, n+1)`. Oublier le +1 est une erreur très fréquente. `range(n)` seul donne 0, 1, ..., $n-1$ (n valeurs en partant de 0).

3.4 Boucles non bornées : répéter jusqu'à une condition

Définition | Boucle while

La boucle while répète un bloc **tant que** sa condition reste vraie. On l'utilise quand on **ne sait pas d'avance** combien de répétitions seront nécessaires (recherche de seuil, par exemple).

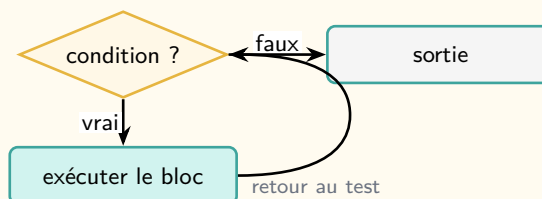
```
1 capital = 1000
2 annees = 0
3 while capital < 2000:      # tant qu'on n'a pas doublé
4     capital = capital * 1.05 # +5% par an
5     annees = annees + 1
6 print(annees)              # nombre d'annees pour doubler
```

Attention | La boucle infinie

Si la condition d'un while ne devient **jamais** fausse, le programme tourne indéfiniment. Il faut **toujours** s'assurer que quelque chose **progressé** à chaque tour vers l'arrêt (ici, capital augmente). Une boucle while True sans break ne s'arrête pas : à éviter tant qu'on débute.

Intuition | Organigramme du while

On teste **avant** chaque tour : si la condition est vraie, on exécute le bloc puis on **revient** au test ; sinon on sort.



3.5 Fonctions et programmation modulaire

Définition | Fonction

Une **fonction** regroupe un bloc d'instructions sous un **nom**. Elle reçoit des **paramètres** (entrées) et renvoie un résultat avec return.

```
1 def aire_disque(rayon):
2     return 3.14159 * rayon**2
3
4 a = aire_disque(2)      # appel : rayon vaut 2
5 print(a)                # 12.56636
```

Attention | return n'est pas print

`print` **affiche** une valeur à l'écran (pour l'humain) mais ne la **renvoie pas** au programme. `return` **renvoie** la valeur pour qu'on puisse la **réutiliser** (la stocker, la recombinaison). Une fonction qui calcule un résultat doit le `return`, pas seulement l'afficher : sinon on ne peut rien en faire ensuite. Après un `return`, la fonction s'arrête immédiatement.

Méthode | Programmation modulaire : découper en sous-tâches

Pour un problème complexe :

1. **Identifier** les sous-tâches indépendantes (« calculer la moyenne », « chercher le maximum »...).
2. **Écrire une fonction** pour chacune, avec un nom clair et un `return`.
3. **Tester** chaque fonction séparément sur des cas simples.
4. **Assembler** les fonctions dans un programme principal.

On obtient un code lisible, testable et réutilisable.

3.6 Les listes : la grande nouveauté**Définition | Liste**

Une **liste** range **plusieurs valeurs** dans un ordre précis, sous un seul nom. Chaque élément a un **indice** (sa position), qui **commence à 0**.

```

1 L = [5, 12, 8, 20]      # liste en extension
2 print(L[0])            # 5   (premier element, indice 0)
3 print(L[3])            # 20  (quatrième element, indice 3)
4 print(len(L))          # 4   (nombre d'elements)
```

Attention | Les indices commencent à 0

Dans une liste de longueur n , les indices valides vont de 0 à $n-1$. Le **premier** élément est `L[0]`, le **dernier** est `L[n-1]` (ou `L[-1]` en Python). Accéder à `L[n]` provoque une **erreur** (indice hors limites). C'est l'autre grand piège du débutant.

✓ Propriété | Trois façons de créer une liste

```

1 # 1. En extension : on écrit les éléments
2 carres = [0, 1, 4, 9, 16]
3
4 # 2. Par ajouts successifs : on part du vide et on remplit
5 carres = []
6 for k in range(5):
7     carres.append(k**2)      # ajoute k^2 à la fin
8
```



```

9 # 3. En comprehension : une formule + une plage (+ une condition)
10 carres = [k**2 for k in range(5)]
11 pairs  = [k for k in range(20) if k % 2 == 0] # avec condition

```

Intuition | La liste en compréhension se lit comme un ensemble

`[k**2 for k in range(5)]` se lit « l'ensemble des k^2 **pour** k allant de 0 à 4 », exactement comme la notation mathématique $\{k^2 \mid 0 \leq k \leq 4\}$. La condition `if` joue le rôle du « tel que » : `[k for k in range(20) if k%2==0]` se lit $\{k \mid 0 \leq k < 20 \text{ et } k \text{ pair}\}$. C'est le pont direct entre **listes** et **ensembles**.

Méthode | Opérations courantes sur les listes

Opération	Code Python
Longueur	<code>len(L)</code>
Accéder à l'élément d'indice i	<code>L[i]</code>
Ajouter à la fin	<code>L.append(x)</code>
Supprimer l'élément d'indice i	<code>del L[i]</code>
Tester l'appartenance	<code>x in L</code>
Somme / maximum / minimum	<code>sum(L), max(L), min(L)</code>

✓ Propriété | Deux façons de parcourir une liste

```

1 L = [5, 12, 8, 20]
2
3 # Parcourir par INDICES (utile si on a besoin de la position)
4 for i in range(len(L)):
5     print(i, L[i])
6
7 # ITERER sur les ELEMENTS (plus simple si la position est inutile)
8 for valeur in L:
9     print(valeur)

```

Exemple | Moyenne d'une liste de notes

```

1 def moyenne(notes):
2     if len(notes) == 0:
3         return None # liste vide : pas de moyenne
4     return sum(notes) / len(notes)
5

```

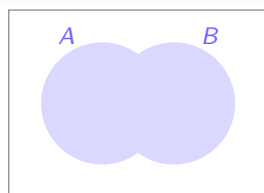
```
6 print(moyenne([12, 8, 15, 10])) # 11.25
```

3.7 Un peu de logique et d'ensembles

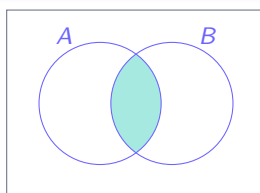
Définition | Ensembles : appartenance, inclusion, opérations

Un **ensemble** est une collection d'éléments. On note :

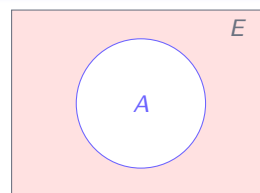
- $x \in A$: « x **appartient** à A » ; $A \subset B$: « A est **inclus** dans B » (tout élément de A est dans B) ;
- $A \cap B$: l'**intersection** (éléments dans A **et** dans B) ; $A \cup B$: la **réunion** (dans A **ou** dans B) ;
- \bar{A} (ou $E \setminus A$) : le **complémentaire** de A dans E (les éléments de E qui ne sont **pas** dans A).



$A \cup B$



$A \cap B$



\bar{A} (zone colorée)

Définition | Logique : connecteurs, implication, équivalence

- « P **et** Q » est vraie quand les **deux** le sont ; « P **ou** Q » est vraie quand **au moins une** l'est.
- **Implication** $P \Rightarrow Q$: « si P alors Q ». P est une **condition suffisante** pour Q ; Q est une **condition nécessaire** pour P .
- **Réciproque** de $P \Rightarrow Q$: c'est $Q \Rightarrow P$ (attention, elle peut être fausse même si l'implication est vraie).
- **Équivalence** $P \Leftrightarrow Q$: $P \Rightarrow Q$ **et** $Q \Rightarrow P$ (« P si et seulement si Q »).

Attention | Une implication n'est pas sa réciproque

« S'il pleut, alors le sol est mouillé » est vraie. Sa réciproque « si le sol est mouillé, alors il pleut » est **fausse** (quelqu'un a pu arroser). De même en maths : « $x = 2 \Rightarrow x^2 = 4$ » est vraie, mais la réciproque « $x^2 = 4 \Rightarrow x = 2$ » est fausse (car $x = -2$ marche aussi). **Un seul contre-exemple** suffit à montrer qu'une proposition est fausse.

✓ Propriété | Négation des propositions quantifiées

Proposition	Sa négation
« Tous les x vérifient P »	« Il existe un x qui ne vérifie pas P »
« Il existe un x tel que P »	« Aucun x ne vérifie P »
« P et Q »	« (non P) ou (non Q) »
« P ou Q »	« (non P) et (non Q) »

4 Boîte à outils : réflexes pour le bac

Méthode | Mémo syntaxe Python

```

1 x = 5                                # affectation
2 if cond:                             # condition (: puis indentation)
3     ...
4 elif autre_cond:
5     ...
6 else:
7     ...
8 for k in range(a, b):                # boucle bornée (b exclu)
9     ...
10 while cond:                         # boucle non bornée
11     ...
12 def f(parametres):                  # fonction
13     return resultat
14 L = [1, 2, 3]                        # liste ; L[0] premier ; len(L) longueur
15 L.append(x)                          # ajoute à la fin
16 [expr for k in range(n) if cond]    # liste en compréhension

```

Attention | Top des erreurs à éviter

- **Confondre = et ==** : = affecte, == teste l'égalité.
- **Oublier l'indentation** ou le : après if, for, while, def.
- **range(1, n) au lieu de range(1, n+1)** : la borne droite est exclue.
- **Indices** : le premier élément est L[0], le dernier L[len(L)-1]. L[len(L)] = erreur.
- **print au lieu de return** : une fonction qui calcule doit **renvoyer** son résultat.
- **Boucle infinie** : dans un while, vérifie que la condition finit par devenir fausse.
- **Confondre implication et réciproque**, ou oublier qu'un contre-exemple suffit à réfuter.

✓ Propriété | Quel outil pour quel besoin ?

Besoin	Outil
Choisir entre plusieurs cas	if / elif / else
Répéter un nombre connu de fois	for ... in range(...)
Répéter jusqu'à une condition	while ...
Réutiliser un calcul	def ...: return ...
Stocker plusieurs valeurs	une list
Construire une liste « en sélectionnant »	liste en compréhension avec if

Méthode | Trois algorithmes à savoir refaire par cœur

```
1 # 1. Somme / moyenne d'une liste
2 def moyenne(L):
3     return sum(L) / len(L)
4
5 # 2. Recherche du maximum (sans max())
6 def maximum(L):
7     m = L[0]
8     for x in L:
9         if x > m:
10             m = x
11     return m
12
13 # 3. Recherche de seuil (premier n tel que u_n dépasse S)
14 def seuil(S):
15     u = 1
16     n = 0
17     while u <= S:
18         u = u * 2      # exemple : u_n = 2^n
19         n = n + 1
20     return n
```

5 Exercices

Exercice 1 ★★ : Lire une affectation

On exécute : $a = 4$; $b = 7$; $a = a + b$; $b = a - b$. Quelles sont les valeurs finales de a et b ?

Exercice 2 ★★ : Trace d'une boucle

Que vaut s après ce code ? Détailler les valeurs successives.

```
1 s = 0
2 for k in range(1, 5):
3     s = s + k
```

Exercice 3 ★★ : Lire une liste

Soit $L = [3, 9, 1, 7, 4]$. Donner $L[0]$, $L[2]$, $\text{len}(L)$, $L[\text{len}(L)-1]$, et la valeur de $\text{sum}(L)$.

Exercice 4 ★★ : Pair ou impair

Écrire une fonction $\text{parite}(n)$ qui renvoie la chaîne "pair" si n est pair, "impair" sinon. (Indice : $n \% 2$ vaut 0 si n est pair.)

Exercice 5 ★★ : Compter

Écrire une fonction $\text{compte_pairs}(L)$ qui renvoie le nombre d'éléments pairs d'une liste L .

Exercice 6 ★★ : Liste en compréhension

Écrire, en une seule ligne (compréhension), la liste des cubes k^3 pour k de 1 à 10. Puis la liste des multiples de 3 entre 0 et 30 inclus.

Exercice 7 ★★ : Somme conditionnelle

Écrire une fonction $\text{somme_positifs}(L)$ qui renvoie la somme des éléments **strictement positifs** d'une liste L .

Exercice 8 ★★ : Table de multiplication

Écrire une fonction $\text{table}(n)$ qui renvoie la **liste** $[n, 2n, 3n, \dots, 10n]$.

Exercice 9 ★★ : Maximum et sa position

Écrire une fonction $\text{indice_max}(L)$ qui renvoie l'**indice** du plus grand élément de L (le premier en cas d'égalité).

Exercice 10 ★★ : Suite et seuil

On définit $u_0 = 2$ et $u_{n+1} = 1,5 u_n$. Écrire une fonction $\text{seuil}(S)$ qui renvoie le plus petit n tel que $u_n > S$.

Exercice 11 ★★★ : Recherche dans une liste

Écrire une fonction `est_present(x, L)` qui renvoie `True` si `x` apparaît dans `L`, `False` sinon, **sans** utiliser le mot-clé `in` (avec une boucle).

Exercice 12 ★★★ : Logique

Pour chaque implication, dire si elle est vraie, puis donner sa réciproque et dire si **elle** est vraie. **(a)** « n multiple de 4 $\Rightarrow n$ pair ». **(b)** « $x > 3 \Rightarrow x > 1$ ». **(c)** « $x^2 = 9 \Rightarrow x = 3$ ».

Exercice 13 ★★★ : Crible des nombres premiers

Écrire une fonction `est_premier(n)` qui renvoie `True` si `n` est premier ($n \geq 2$), puis une fonction `premiers_jusqua(N)` qui renvoie la liste des nombres premiers inférieurs ou égaux à `N`.

Exercice 14 ★★★ : Simulation et espérance

On lance deux dés et on s'intéresse à la **somme** `S`. Écrire une fonction `moyenne_somme(n)` qui simule `n` lancers de deux dés et renvoie la moyenne des sommes obtenues. Vers quelle valeur doit-elle tendre ? (Lien fiche 10.)

Exercice 15 ★★★ : Inverser une liste

Écrire une fonction `renverse(L)` qui renvoie une **nouvelle** liste contenant les éléments de `L` dans l'ordre inverse, sans utiliser `L.reverse()` ni `L[::-1]`.

Exercice 16 ★★★ : Ensembles via les listes

Écrire une fonction `intersection(A, B)` qui renvoie la liste des éléments présents à la fois dans `A` et dans `B` (sans doublon). Relier au symbole mathématique correspondant.

6 Problème type prépa

Problème style prépa : *un mini-tableur de statistiques*

Ce problème mobilise **toutes** les briques de la fiche : variables, conditions, boucles, fonctions, listes et compréhensions. On construit, brique par brique, un petit programme modulaire d'analyse de notes. Chaque fonction sera testée avant d'être réutilisée.

Partie A : les briques de base

On dispose d'une liste de notes, par exemple `notes = [12, 8, 15, 10, 6, 18, 9]`.

1. Écrire `moyenne(L)` qui renvoie la moyenne d'une liste non vide.
2. Écrire `maximum(L)` et `minimum(L)` **sans** utiliser `max` ni `min`.
3. Écrire `nb_admis(L)` qui renvoie le nombre de notes supérieures ou égales à 10.

Partie B : compréhensions et sélection

4. En **une ligne** (compréhension), construire la liste des notes ≥ 10 .
5. Écrire `ajuste(L)` qui renvoie une **nouvelle** liste où chaque note est augmentée de 1 point, **sans dépasser 20**.

Partie C : assemblage modulaire

6. Écrire `bilan(L)` qui **réutilise** les fonctions précédentes et renvoie un texte du type : « Moyenne : 11.1 ; Min : 6 ; Max : 18 ; Admis : 4/7 ».
7. Expliquer en deux phrases pourquoi découper en petites fonctions (plutôt qu'un seul long bloc) rend le programme plus sûr et plus réutilisable.

7 ✓ Corrigés détaillés

Corrigé 1

Démonstration

On suit les affectations **ligne par ligne**, en gardant en tête les valeurs des boîtes :

- $a = 4 : a = 4.$ $b = 7 : b = 7.$
- $a = a + b : a = 4 + 7 = 11$ (et b reste 7).
- $b = a - b : b = 11 - 7 = 4.$

Valeurs finales : $a = 11, b = 4$. (Ce code échange presque les valeurs, mais a garde la somme.)

Corrigé 2

Démonstration

`range(1, 5)` parcourt $k = 1, 2, 3, 4$. On accumule dans s :

$$s : 0 \xrightarrow{+1} 1 \xrightarrow{+2} 3 \xrightarrow{+3} 6 \xrightarrow{+4} 10.$$

Donc s vaut 10 .

Corrigé 3

Démonstration

$L = [3, 9, 1, 7, 4]$. Les indices vont de 0 à 4 :

$$L[0] = 3, \quad L[2] = 1, \quad \text{len}(L) = 5, \quad L[\text{len}(L)-1] = L[4] = 4.$$

$$\text{sum}(L) = 3 + 9 + 1 + 7 + 4 = 24.$$

Corrigé 4

Démonstration

```
1 def parite(n):  
2     if n % 2 == 0:  
3         return "pair"  
4     else:  
5         return "impair"
```

Le test `n % 2 == 0` vérifie que le reste de la division par 2 est nul. Noter le **double** égal `==` (test) et le `:` suivi de l'indentation.

Corrigé 5

Démonstration

On parcourt la liste et on compte :

```
1 def compte_paires(L):
2     c = 0
3     for x in L:
4         if x % 2 == 0:
5             c = c + 1
6     return c
```

On itère directement sur les éléments (`for x in L`) car la position ne sert pas. `c` compte les nombres pairs rencontrés.

Corrigé 6

Démonstration

```
1 cubes = [k**3 for k in range(1, 11)]          # 1 à 10 inclus
2 mult3 = [k for k in range(0, 31) if k % 3 == 0]
```

Pour aller jusqu'à 10 **inclus**, on écrit `range(1, 11)`. La condition `if k%3==0` ne garde que les multiples de 3. Résultats : `[1, 8, 27, ..., 1000]` et `[0, 3, 6, ..., 30]`.

Corrigé 7

Démonstration

```
1 def somme_positifs(L):
2     s = 0
3     for x in L:
4         if x > 0:
5             s = s + x
6     return s
```

On n'ajoute `x` à la somme que s'il est strictement positif. Par exemple sur `[3, -2, 5, -7]` on obtient $3 + 5 = 8$.

Corrigé 8

Démonstration

```
1 def table(n):
2     return [n * k for k in range(1, 11)]
```

La compréhension construit `[n*1, n*2, ..., n*10]`. On pouvait aussi partir d'une liste vide et faire `append` dans une boucle : les deux méthodes sont équivalentes.

Corrigé 9

Démonstration

On retient à la fois la **valeur** du maximum et son **indice** :

```
1 def indice_max(L):
2     imax = 0
3     for i in range(len(L)):
4         if L[i] > L[imax]:
5             imax = i
6     return imax
```

On compare chaque élément $L[i]$ au meilleur trouvé $L[imax]$. L'inégalité **stricte** $>$ garantit qu'en cas d'égalité on garde le **premier** indice. Ici on parcourt par **indices** car c'est la position qu'on veut renvoyer.

Corrigé 10

Démonstration

```
1 def seuil(S):
2     u = 2          #  $u_0$ 
3     n = 0
4     while u <= S:
5         u = 1.5 * u    #  $u_{n+1} = 1.5 u_n$ 
6         n = n + 1
7     return n
```

La suite est géométrique de raison $1,5 > 1$, donc elle **croît sans limite** : le while finit toujours par s'arrêter. On compte dans n le nombre d'étapes jusqu'à dépasser S . Par exemple `seuil(10)` renvoie 4 (car $u_0 = 2$, $u_1 = 3$, $u_2 = 4,5$, $u_3 = 6,75$, $u_4 = 10,125 > 10$).

Corrigé 11

Démonstration

```
1 def est_present(x, L):
2     for valeur in L:
3         if valeur == x:
4             return True    # trouve : on sort tout de suite
5     return False          # parcours fini sans trouver
```

Dès qu'on trouve x , le `return True` **arrête** la fonction (inutile de continuer). Si la boucle se termine sans rien trouver, on renvoie `False`.

Corrigé 12

Démonstration

(a) « n multiple de 4 $\Rightarrow n$ pair » : **vraie** (tout multiple de 4 est multiple de 2). Réciproque : « n pair $\Rightarrow n$ multiple de 4 » : **fausse** ($n = 2$ est pair mais pas multiple de 4).

(b) « $x > 3 \Rightarrow x > 1$ » : **vraie**. Réciproque : « $x > 1 \Rightarrow x > 3$ » : **fausse** ($x = 2$).

(c) « $x^2 = 9 \Rightarrow x = 3$ » : **fausse** (contre-exemple $x = -3$). Réciproque : « $x = 3 \Rightarrow x^2 = 9$ » : **vraie**.

Leçon : une implication et sa réciproque sont **indépendantes**, et un seul contre-exemple suffit à réfuter.

Corrigé 13

Démonstration

```

1 def est_premier(n):
2     if n < 2:
3         return False
4     for d in range(2, n):           # diviseurs possibles
5         if n % d == 0:
6             return False          # un diviseur trouve : pas premier
7     return True
8
9 def premiers_jusqua(N):
10    return [n for n in range(2, N + 1) if est_premier(n)]

```

`est_premier` cherche un diviseur entre 2 et $n - 1$: s'il en trouve un, n n'est pas premier. `premiers_jusqua` **réutilise** cette fonction dans une compréhension (programmation modulaire). Par exemple `premiers_jusqua(20)` renvoie `[2, 3, 5, 7, 11, 13, 17, 19]`.

Corrigé 14

Démonstration

```

1 from random import randint
2
3 def moyenne_somme(n):
4     total = 0
5     for _ in range(n):
6         total += randint(1, 6) + randint(1, 6)    # somme de 2 des
7     return total / n

```

D'après la fiche 10, l'espérance de la somme de deux dés vaut 7. Par la loi des grands nombres, `moyenne_somme(n)` se rapproche de 7 quand n devient grand.

Corrigé 15

Démonstration

On parcourt L de la **fin** vers le **début** en ajoutant dans une nouvelle liste :

```
1 def renverse(L):
2     R = []
3     for i in range(len(L) - 1, -1, -1):    # de len-1 jusqu'à 0
4         R.append(L[i])
5     return R
```

range(len(L)-1, -1, -1) compte à rebours : indice de départ len(L)-1, on s'arrête **avant** -1 (donc à 0), pas de -1. Par exemple renverse([1,2,3]) renvoie [3, 2, 1].

Corrigé 16

Démonstration

```
1 def intersection(A, B):
2     R = []
3     for x in A:
4         if x in B and x not in R:    # dans B, et pas déjà pris
5             R.append(x)
6     return R
```

On garde un élément de A s'il est **aussi** dans B (condition `x in B`) et pas déjà ajouté (évite les doublons). Cela réalise l'**intersection** d'ensembles, notée $A \cap B$ en mathématiques. Par exemple `intersection([1,2,3,2], [2,3,5])` renvoie [2, 3].

Corrigé du problème type prépa

Démonstration / Partie A : les briques de base

```
1 def moyenne(L):
2     return sum(L) / len(L)
3
4 def maximum(L):
5     m = L[0]
6     for x in L:
7         if x > m:
8             m = x
9     return m
10
11 def minimum(L):
12     m = L[0]
13     for x in L:
14         if x < m:
15             m = x
16     return m
17
18 def nb_admis(L):
19     c = 0
20     for note in L:
21         if note >= 10:
22             c = c + 1
23     return c
```

Chaque fonction fait **une seule** chose et la renvoie. On peut les tester isolément, par exemple `maximum([12, 8, 15])` doit donner 15.

Démonstration / Partie B : compréhensions et sélection

```
1 # 4. notes >= 10 en une ligne
2 def admis(L):
3     return [note for note in L if note >= 10]
4
5 # 5. +1 point, plafonne a 20
6 def ajuste(L):
7     return [min(note + 1, 20) for note in L]
```

La compréhension de la question 4 se lit « les notes **telles que** $\text{note} \geq 10$ ». Pour la question 5, `min(note + 1, 20)` garantit qu'on **ne dépasse jamais** 20 : un 20 reste 20, un 19 devient 20, un 12 devient 13.

Démonstration / Partie C : assemblage modulaire

```
1 def bilan(L):
2     moy = round(moyenne(L), 1)
3     mini = minimum(L)
```

```
4     maxi = maximum(L)
5     adm = nb_admis(L)
6     total = len(L)
7     return ("Moyenne : " + str(moy) +
8            " ; Min : " + str(mini) +
9            " ; Max : " + str(maxi) +
10           " ; Admis : " + str(adm) + "/" + str(total))
11
12 notes = [12, 8, 15, 10, 6, 18, 9]
13 print(bilan(notes))
14 # Moyenne : 11.1 ; Min : 6 ; Max : 18 ; Admis : 4/7
```

7. Pourquoi découper ? Chaque petite fonction peut être **testée et corrigée séparément**, ce qui rend les erreurs faciles à localiser ; et une fois sûre, chaque fonction se **réutilise** dans d'autres programmes sans tout réécrire. Un seul long bloc serait illisible et chaque modification risquerait de tout casser.

Félicitations ! Tu as bouclé les **11 fiches** de Première Spé. Tu maîtrises désormais les **quatre briques** (variables, conditions, boucles, fonctions), la **programmation modulaire**, les **listes** (extension, ajouts, compréhension) et le **vocabulaire logique**. Ces outils te suivront toute l'année et en Terminale. Beau travail !