Plaquette 2 – Variables, boucles bornées et fonctions

Algorithmique et Python — Première spé maths

Corrections détaillées

Boîte à outils

Affectation. $x = 5$ place la valeur 5 dans la variable x. L'affectation $x = x + 1$ **lit** l'ancienne valeur puis **écrit** la nouvelle.

Boucle bornée.

```
for k in range(n):
    instruction
```

range(n) parcourt 0, 1, …, $n-1$ — soit n valeurs, et **pas** $n+1$.

Écriture	Valeurs parcourues	Nombre
range(5)	0, 1, 2, 3, 4	5
range(1, 5)	1, 2, 3, 4	4
range(1, 6)	1, 2, 3, 4, 5	5
range(0, 10, 2)	0, 2, 4, 6, 8	5

Fonction.

```
def aire(L, l):
    return L * l
```

Le mot-clé *return* **renvoie** une valeur et **quitte** la fonction. Une fonction sans *return* renvoie *None*.

Opérateurs utiles. *//* division entière · *%* reste · * *puissance* · abs()* valeur absolue.

Tracer une exécution. On dresse un tableau avec une colonne par variable et une ligne par tour de boucle — c'est la méthode la plus sûre pour prévoir un affichage.

Trois erreurs à traquer.

1. Confondre *range(n)* (n valeurs, de 0 à $n-1$) et une boucle allant de 1 à n. 2. Placer le *print* **dans** la boucle au lieu d'après (ou l'inverse). 3. Oublier d'**initialiser** un accumulateur avant la boucle.

Correction de l'exercice 1 – Trace d'exécution

— **Valeurs de** k : 0, 1, 2, 3, 4 — soit **cinq** tours ✓

— **Tableau d'exécution** :

! tour	k	s après
début	·	0
1	0	0
2	1	1
3	2	3
4	3	6
5	4	10

— **Affichage** : 10 ✓
— **Contrôle par la formule** : $0 + 1 + 2 + 3 + 4 = \frac{4 \times 5}{2} = 10$ ✓
— **Piège** : k ne monte **pas** jusqu'à 5 — *range(5)* s'arrête à 4 ✓
— **Variante** : avec *range(1, 6)*, on aurait $1 + 2 + 3 + 4 + 5 = 15$.
— **Généralisation** : *range(n)* donne la somme $\frac{(n-1)n}{2}$.
— **Contrôle** : pour $n = 5$, $\frac{4 \times 5}{2} = 10$ ✓

Réponse. $s = 10$.

Correction de l'exercice 2 – Compter les tours

— **a)** k va de 0 à 9 : **dix** tours ✓
— **b)** k va de 3 à 9 : $9 - 3 + 1$ = **sept** tours ✓
— **c)** k prend 0, 5, 10, 15 : **quatre** tours ✓
— **d)** k devrait aller de 10 à 2 avec un pas de +1 — c'est **impossible**, la boucle ne s'exécute **jamais** : **zéro** tour ✓
— **Règle générale** : *range(a, b, p)* parcourt a, $a + p$, … tant que la valeur reste **strictement inférieure** à b (pour $p > 0$) ✓
— **Nombre de tours** : $\left|\frac{b-a}{p}\right|$ si positif, 0 sinon.
— **Contrôle pour c)** : $\left|\frac{20-0}{5}\right| = 4$ ✓
— **Contrôle pour b)** : $\left|\frac{10-3}{1}\right| = 7$ ✓
— **Pour descendre** : il faut un pas négatif, comme *range(10, 3, -1)*, qui donne 10, 9, …, 4 — soit **sept** tours ✓
— **Erreur silencieuse** : le cas d) ne provoque **aucune** erreur Python, juste une boucle vide — d'où sa dangerosité.

Réponse. a) 10 · b) 7 · c) 4 · d) 0 (boucle vide).

Correction de l'exercice 3 – Écrire une fonction

Fonction.

```
def somme(n):
    s = 0
    for k in range(1, n + 1):
        s = s + k
    return s
```

— **Test** : *somme(100)* renvoie 5050 ✓
— **Contrôle par la formule** : $\frac{100 \times 101}{2} = 5050$ ✓
— **Attention à la borne** : *range(1, n + 1)* est nécessaire pour inclure n ✓
— **Contrôle sur de petites valeurs** : *somme(1)* = 1 ; *somme(3)* = 6 ; *somme(4)* = 10 ✓
— **Cas limite** : *somme(0)* renvoie 0, car la boucle ne tourne pas — comportement correct ✓
— **Version en une ligne** : *return n* (n + 1) // 2* — immédiate et exacte, sans boucle ✓
— **Comparaison de coût** : la boucle fait n additions, la formule une multiplication et une division — pour $n = 10^9$, la différence est décisive.
— **Pourquoi la division entière** *//* : elle garantit un résultat **entier** ; avec /, Python renverrait 5050,0 (un flottant) ✓

Réponse. *somme(100)* = 5050 (ou directement $n(n+1)//2$).

Correction de l'exercice 4 – Boucle imbriquée

— **Boucle extérieure** : 4 tours.
— **Boucle intérieure** : 3 tours **à chaque** tour extérieur.
— **Total** :

$$4 \times 3 = 12 \text{ exécutions}$$

— **Couples affichés** : $(0,0)$, $(0,1)$, $(0,2)$, $(1,0)$, …, $(3,2)$ ✓
— **Ordre** : j varie **le plus vite**, i le plus lentement ✓
— **Contrôle** : le dernier couple est $(3,2)$ ✓
— **Généralisation** : deux boucles de n et m tours donnent nm exécutions ✓
— **Enjeu de complexité** : deux boucles imbriquées sur n donnent n^2 opérations — pour $n = 10\,000$, cela fait 10^8, à la limite du raisonnable ✓
— **Trois boucles imbriquées** : n^3, soit 10^{12} pour $n = 10\,000$ — hors de portée.
— **Usage typique** : parcourir un tableau à deux dimensions, ou énumérer tous les couples possibles (par exemple les 36 lancers de deux dés avec *range(1, 7)* deux fois).

Réponse. 12 fois (4×3).

Correction de l'exercice 5 – Accumuler un produit

Fonction.

```
def factorielle(n):
    p = 1
    for k in range(1, n + 1):
        p = p * k
    return p
```

— **Initialisation à** 1 : essentielle ! Avec $p = 0$, tout produit resterait nul ✓
— **Calcul de** 6! : $1 \times 2 \times 3 \times 4 \times 5 \times 6 = 720$ ✓
— **Tableau d'exécution** : p vaut successivement 1, 2, 6, 24, 120, 720 ✓
— **Cas limite** : *factorielle(0)* renvoie 1, car la boucle ne tourne pas — et c'est mathématiquement **correct** ($0! = 1$) ✓
— **Croissance** : $10! = 3\,628\,800$ et $20! \approx 2{,}43 \times 10^{18}$ — la factorielle explose ✓
— **Contrôle** : $7! = 5040$ et $8! = 40\,320$ ✓
— **Usage** : les coefficients binomiaux, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ — mais attention, calculer $\binom{100}{50}$ ainsi ferait manipuler des nombres à 158 chiffres, alors que le résultat n'en a que 30.
— **Meilleure méthode pour un coefficient** : multiplier et diviser alternativement, $\prod_{i=0}^{k-1} \frac{n-i}{i+1}$ ✓
— **Différence avec la somme** : l'accumulateur d'un produit s'initialise à 1, celui d'une somme à 0 — c'est l'**élément neutre** de l'opération ✓

Réponse. $6! = 720$ (accumulateur initialisé à 1).

Correction de l'exercice 6 – Compter selon une condition

Script.

```
c = 0
for k in range(1, 101):
```

```
    if k % 3 == 0:
        c = c + 1
print(c)
```

— **Test de divisibilité** : *k % 3 == 0* signifie « le reste de *k* par 3 est nul » ✓
— **Résultat** : 33 multiples ✓
— **Contrôle** : les multiples sont 3, 6, …, 99, soit $\frac{99}{3} = 33$ ✓
— **Méthode directe** : $\left\lfloor \frac{100}{3} \right\rfloor = 33$, calculable par *100 // 3* ✓
— **Attention à la borne** : *range(1, 101)* est nécessaire pour inclure 100 ✓
— **Variante sans boucle** : *print(100 // 3)* — instantané ✓
— **Autre comptage** : les multiples de 3 **ou** de 5 sont au nombre de $33 + 20 - 6 = 47$ (principe d'inclusion-exclusion, les multiples de 15 étant comptés deux fois) ✓
— **Contrôle** : $\left\lfloor \frac{100}{5} \right\rfloor = 20$ et $\left\lfloor \frac{100}{15} \right\rfloor = 6$ ✓
— **Piège du double égal** : *k % 3 = 0* (un seul =) provoquerait une **erreur de syntaxe** — l'affectation n'est pas un test ✓

Réponse. 33 multiples (ou directement *100 // 3*).

Correction de l'exercice 7 – Maximum d'une liste

Fonction.

```
def maximum(L):
    m = L[0]
    for x in L:
        if x > m:
            m = x
    return m
```

— **Initialisation** : avec le **premier élément**, jamais avec 0 ✓
— **Pourquoi** : une liste de nombres tous négatifs, comme $[-5, -2, -9]$, renverrait 0 à tort ✓
— **Contrôle** : *maximum([-5, -2, -9])* renvoie bien -2 ✓
— **Trace sur** $[3, 7, 2, 9, 5]$: m vaut 3, puis 7, puis 9 — résultat 9 ✓
— **Coût** : une comparaison par élément, soit n comparaisons — complexité **linéaire** ✓
— **Optimalité** : impossible de faire mieux, puisque tout élément non lu pourrait être le maximum ✓
— **Cas de la liste vide** : *L[0]* provoque une erreur *IndexError* — il faudrait tester *if len(L) == 0* selon l'usage voulu ✓
— **Variante pour le minimum** : inverser le test en $x < m$ ✓
— **Les deux en un seul parcours** : maintenir deux variables m et M, ce qui coûte $2n$ comparaisons au lieu de $2n$ pour deux parcours séparés — même coût, mais une seule lecture des données ✓

Réponse. Parcours avec mémorisation, initialisée à *L[0]* ; coût linéaire.

Correction de l'exercice 8 – Somme des carrés

Script.

```
s = 0
for k in range(1, 21):
    s = s + k**2
print(s)
```

— **Résultat** : 2870 ✓
— **Contrôle par la formule** :

$$\frac{20 \times 21 \times 41}{6} = \frac{17\,220}{6} = 2870$$

— Les deux méthodes concordent ✓
— **Contrôle sur un petit cas** : pour $n = 3$, le script donne $1 + 4 + 9 = 14$, et la formule $\frac{3 \times 4 \times 7}{6} = 14$ ✓
— **Ordre de grandeur** : la somme est voisine de $\frac{n^3}{3} = \frac{8000}{3} \approx 2667$ ✓
— **Opérateur de puissance** : *k2* ou *k*k** — les deux fonctionnent, le second étant légèrement plus rapide ✓
— **Piège** : *k^2* en Python n'est **pas** une puissance mais un « ou exclusif » binaire — *3^2* vaut 1 et non 9 ✓
— **Somme des cubes** : $\left(\frac{n(n+1)}{2}\right)^2 = 210^2 = 44\,100$ pour $n = 20$ ✓
— **Vérification croisée** : la somme des cubes est le **carré** de la somme des entiers — une identité remarquable qu'un script confirme immédiatement.

Réponse. 2870, conforme à $\frac{20 \times 21 \times 41}{6}$.

Correction de l'exercice 9 – Fonction à deux paramètres

Fonction.

```
def terme(u0, r, n):
    return u0 + n * r
```

— **Calcul** : $7 + 20 \times (-3) = 7 - 60 = -53$ ✓
— **Version par boucle** (utile pour une suite définie par récurrence) :

```
def terme_boucle(u0, r, n):
    u = u0
    for k in range(n):
        u = u + r
    return u
```

— **Contrôle** : les deux versions donnent -53 ✓
— **Nombre de tours** : *range(n)* donne exactement n additions, ce qui mène de u_0 à u_n ✓
— **Contrôles sur de petites valeurs** : *terme(7, -3, 0)* = 7 ✓ ; *terme(7, -3, 1)* = 4 ✓ ; *terme(7, -3, 2)* = 1 ✓
— **Avantage de la formule directe** : coût **constant**, alors que la boucle coûte n opérations ✓
— **Quand la boucle est indispensable** : pour une suite comme $u_{n+1} = 1{,}05u_n + 3$, qui n'a pas de forme explicite évidente ✓
— **Version géométrique** : *return u0* q**n* ✓
— **Piège de l'ordre des paramètres** : *terme(-3, 7, 20)* donnerait 137 — nommer les arguments (*terme(u0=7, r=-3, n=20)*) évite l'erreur ✓

Réponse. $u_{20} = -53$.

Correction de l'exercice 10 – Moyenne d'une liste

Fonction.

```
def moyenne(L):
    s = 0
    for x in L:
        s = s + x
    return s / len(L)
```

— **Somme** : $12 + 15 + 9 + 20 + 14 = 70$ ✓
— **Effectif** : 5 éléments.
— **Moyenne** :

$$\frac{70}{5} = 14$$

— **Contrôle** : la moyenne est bien entre le minimum (9) et le maximum (20) ✓
— **Choix de la division** : / donne un flottant (14,0), ce qui est correct pour une moyenne ; // donnerait 14 mais tronquerait les cas non entiers ✓
— **Contrôle sur un cas non entier** : $[1, 2]$ donne 1,5 avec / mais 1 avec // — la seconde serait **fausse** ✓
— **Cas de la liste vide** : division par zéro, donc erreur *ZeroDivisionError* — il faut la traiter si nécessaire ✓
— **Version en une ligne** : *return sum(L) / len(L)* ✓
— **Écart type** : il faudrait un second parcours pour $\sum (x_i - \overline{x})^2$, ou un seul parcours avec la formule de König-Huygens $\frac{\sum x_i^2}{n} - \overline{x}^2$ ✓
— **Contrôle** : $\frac{144 + 225 + 81 + 400 + 196}{5} - 196 = \frac{1046}{5} - 196 = 209{,}2 - 196 = 13{,}2$, donc $\sigma \approx 3{,}633$ ✓

Réponse. Moyenne = 14 (somme 70 divisée par 5).

Correction de l’exercice 11 – Tableau de valeurs

Script.

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

for k in range(6):
    print(k, f(k))
```

— **Valeurs affichées** :

! x	0	1	2	3	4	5
$f(x)$	1	-1	-1	1	5	11

— **Contrôles** : $f(0) = 1$ ✓ ; $f(2) = 4 - 6 + 1 = -1$ ✓ ; $f(5) = 25 - 15 + 1 = 11$ ✓
— **Symétrie** : $f(1) = f(2) = -1$, cohérent avec un sommet en $x = 1{,}5$ ✓
— **Minimum** : $f(1{,}5) = 2{,}25 - 4{,}5 + 1 = -1{,}25$ ✓
— **Pas de** 0,5 : il faut alors *for k in range(11): x = k* 0.5 — *car* range *n’accepte que des entiers** ✓
— **Racines** : $\Delta = 9 - 4 = 5$, donc $x = \frac{3 \pm \sqrt{5}}{2}$, soit environ 0,382 et 2,618 — le tableau montre bien un changement de signe entre 0 et 1, puis entre 2 et 3 ✓
— **Piège du pas décimal** : *range(0, 5, 0.5)* provoque une **erreur** — *range* exige des entiers ✓

— **Usage** : ce genre de tableau permet de **localiser** les racines et les extremums avant une résolution exacte.

Réponse. 1 ; −1 ; −1 ; 1 ; 5 ; 11 pour $x = 0$ à 5.

Correction de l'exercice 12 – Corriger un script

— **Première erreur** : *s = 0* est **dans** la boucle, donc l'accumulateur est réinitialisé à chaque tour ✓
— **Conséquence** : le script affiche 9 (la dernière valeur de *k*) au lieu de la somme ✓
— **Seconde erreur** : *range(10)* parcourt 0 à 9, donc 10 est **exclu** ✓
— **Script corrigé** :

```
s = 0
for k in range(1, 11):
    s = s + k
print(s)
```

— **Résultat correct** : 55 ✓
— **Contrôle** : $\frac{10 \times 11}{2} = 55$ ✓
— **Ce qu'affichait le script erroné** : 9 — très loin de 55, ce qui aurait dû alerter ✓
— **Règle d'or** : un accumulateur s'initialise **avant** la boucle, jamais dedans ✓
— **Deuxième règle** : pour parcourir 1 à *n*, écrire *range(1, n + 1)* ✓
— **Réflexe de vérification** : tester le script sur un cas où l'on connaît la réponse (1 à 3 doit donner 6) révèle instantanément ce genre de bug.
— **Variante subtile** : avec *range(11)*, le script correct donnerait aussi 55, puisque le 0 ajouté ne change rien — mais *range(1, 11)* exprime mieux l'intention ✓

Réponse. *s = 0* doit être **avant** la boucle, et *range(10)* doit devenir *range(1, 11)* : le résultat est 55.

Correction de l'exercice 13 – Fonction avec condition

Deux nombres.

```
def max2(a, b):
    if a > b:
        return a
    return b
```

— **Pas besoin de* else** *: le* return a* quitte la fonction, donc la suite n'est atteinte que si $a \leq b$ ✓

Trois nombres.

```
def max3(a, b, c):
    return max2(max2(a, b), c)
```

— **Principe** : on réutilise la fonction précédente — c'est la **composition** ✓
— **Contrôles** : *max3(3, 7, 5)* = 7 ✓ ; *max3(9, 2, 4)* = 9 ✓ ; *max3(1, 1, 1)* = 1 ✓
— **Cas d'égalité** : *max2(5, 5)* renvoie 5 (la seconde branche) — correct ✓
— **Version par comparaisons directes** : trois tests imbriqués, plus long et plus sujet aux erreurs ✓
— **Nombre de comparaisons** : 2 pour trois nombres, ce qui est **optimal** ✓
— **Généralisation** : pour *n* nombres, il faut $n - 1$ comparaisons — c'est le principe du tournoi à élimination directe ✓

— **Avantage de la réutilisation** : si *max2* est correcte, *max3* l'est automatiquement. C'est le grand intérêt de découper un programme en fonctions ✓

Réponse. *max2* par un test, puis *max3(a, b, c) = max2(max2(a, b), c)*.

Correction de l'exercice 14 – Simulation simple

Script.

```
from random import randint

c = 0
for i in range(1000):
    if randint(1, 6) == 6:
        c = c + 1
print(c / 1000)
```

— **Valeur attendue** : environ $\frac{1}{6} \approx 0{,}1667$ ✓

— **Écart type de la fréquence** :

$$\sqrt{\frac{\frac{1}{6} \times \frac{5}{6}}{1000}} \approx 0{,}011785$$

— **Intervalle typique** ($\pm 2\sigma$) : $[0{,}1431\,;0{,}1902]$ ✓

— **Conclusion** : le script affichera presque toujours une valeur entre 0,143 et 0,190.

— **Nombre de** 6 : environ 167, avec $\sigma(X) \approx 11{,}785$ ✓

— **Attention aux bornes** : *randint(1, 6)* inclut les **deux** bornes — c'est bien un dé à six faces ✓

— **Erreur classique** : *randint(0, 6)* simulerait sept faces, ramenant la fréquence à $\frac{1}{7} \approx 0{,}1429$ ✓

— **Détectabilité de ce bug** : $\frac{0{,}1429 - 0{,}1667}{0{,}011785} \approx -2{,}019\,\sigma$ — juste détectable sur 1000 lancers, franchement visible sur 10 000 ✓

— **Exécutions successives** : deux lancements du script donneront des résultats **différents** — c'est le propre d'une simulation ✓

— **Pour fixer le hasard** (utile pour déboguer) : *from random import seed* puis *seed(42)* rend la simulation reproductible.

Réponse. Environ 0,167, typiquement entre 0,143 et 0,190.

Correction de l'exercice 15 – Table de multiplication

Script.

```
s = 0
for k in range(1, 11):
    p = 7 * k
    print(7, "x", k, "=", p)
    s = s + p
print("somme :", s)
```

— **Produits** : 7, 14, 21, 28, 35, 42, 49, 56, 63, 70 ✓

— **Somme** :

$$7 \times (1 + 2 + \cdots + 10) = 7 \times 55 = 385$$

— **Contrôle** : $7 + 14 + 21 + 28 + 35 + 42 + 49 + 56 + 63 + 70 = 385$ ✓

— **Factorisation** : la somme est 7 fois la somme des dix premiers entiers — inutile de tout additionner ✓
— **Position du* print*** : dans la boucle pour chaque ligne, après pour le total ✓
— **Affichage** : la virgule dans *print* insère automatiquement une espace entre les éléments ✓
— **Variante avec f-string** : *print(f"7 x {k} = {p}")* est plus lisible.
— **Généralisation** : la somme de la table de n vaut $55n$ ✓
— **Contrôle** : pour $n = 9$, $55 \times 9 = 495$, et $9 + 18 + \cdots + 90 = 495$ ✓
— **Toutes les tables de** 1 **à** 10 : $55 \times 55 = 3025$, avec deux boucles imbriquées ✓

Réponse. Les dix produits, de somme $7 \times 55 = 385$.

Correction de l’exercice 16 – Synthèse : étude d’une suite

a) **Fonction.**

```
def u(n):
    x = 2
    for k in range(n):
        x = 3*x - 4
    return x
```

b) **Valeurs.**

— $u_1 = 6 - 4 = 2$;
— $u_2 = 6 - 4 = 2$;
— $u_3 = 2$; $u_4 = 2$; $u_5 = 2$.
— **Observation** : la suite est **constante** égale à 2 ✓
— **Explication** : 2 est le **point fixe** de la relation, car $3 \times 2 - 4 = 2$ ✓
— **Vérification du point fixe** : $3x - 4 = x$ donne $2x = 4$, soit $x = 2$ ✓
— **Sensibilité** : avec $u_0 = 3$, on aurait 5, 11, 29, 83, 245 — la suite **explose** ✓
— **Forme explicite générale** : $u_n = 2 + (u_0 - 2) \times 3^n$ ✓
— **Contrôle pour** $u_0 = 3$: $u_5 = 2 + 1 \times 243 = 245$ ✓
— **Contrôle pour** $u_0 = 2$: $u_n = 2 + 0 = 2$ ✓ — le point fixe est **répulsif** : toute autre valeur initiale s’en éloigne.

c) **Somme.**

```
s = 0
x = 2
for k in range(11):
    s = s + x
    x = 3*x - 4
print(s)
```

— **Résultat** : $11 \times 2 = 22$ ✓
— **Attention à l’ordre** : on ajoute x **avant** de le faire évoluer, sinon on manquerait u_0 et compterait u_{11} ✓
— **Contrôle** : 11 termes tous égaux à 2 ✓
— **Avec** $u_0 = 3$: la somme vaudrait
$11 \times 2 + \dfrac{3^{11} - 1}{3 - 1} = 22 + \dfrac{177\,146}{2} = 22 + 88\,573 = 88\,595$ ✓
— **Vérification** :
$3 + 5 + 11 + 29 + 83 + 245 + 731 + 2189 + 6563 + 19\,685 + 59\,051 = 88\,595$ ✓

Réponse. b) la suite est **constante** égale à 2 (point fixe) · c) somme = 22.