2nde / Python : algorithmique et programmation — corrigé

benprof particulier.fr

1. *Variables, types et affectation*

Ex. 1 ★☆☆ **Automatismes : types et affectations**

(a) `-4` est un int, `4.0` un float, `"4"` un str (il est entre guillemets) et `False` un bool.

(b) `15 / 5` vaut 3.0 : c'est un float, car `/` donne toujours un flottant.

(c) Python calcule d'abord $7 + 1 = 8$, puis range 8 dans n : n vaut 8. Ce n'est pas une équation.

(d) `p == 4` compare 3 et 4 : il vaut False. Comparer ne modifie pas p, qui vaut toujours 3.

(e) Python écrit les nombres décimaux avec un point : `prix = 3.75`.

(f) Faux. Une variable ne contient qu'une seule valeur : la deuxième affectation remplace 10, qui est perdu. a contient 4.

Ex. 2 ★☆☆ **Automatismes : les opérations**

(a) $23 = 4 \times 5 + 3$: `23 // 4` affiche le quotient 5 et `23 % 4` affiche le reste 3.

(b) Non. `/` fait la division décimale et donne un flottant : `print(10 / 4)` affiche 2.5. C'est `10 // 4` qui donne le quotient 2.

(c) La multiplication passe d'abord : $2 + 15 = 17$, donc 17. Avec les parenthèses : $7 \times 3 = 21$, donc 21.

(d) Non : en Python, `^` ne calcule pas la puissance. On écrit `3 ** 4`, qui vaut 81.

(e) `45 // 7` vaut 6 et `45 % 7` vaut 3, car $45 = 7 \times 6 + 3$: 6 semaines et 3 jours.

Ex. 3 ★★☆ **Places de cinéma**

(a)

Ligne	prix	places	total
1	9	–	–
2	9	4	–
3	9	4	36
4	7	4	36
5	7	4	40

(b) Le script affiche :

```
40
7
```

Ex. 3 (suite)

(c) Non. La 3e ligne a calculé $9 \times 4 = 36$ une fois pour toutes et a rangé 36 dans total. Changer prix ensuite ne change pas total : total vaut encore 36 après la 4e ligne.

(d) prix devient 7 avant le calcul : total $= 7 \times 4 = 28$, puis $28 + 4 = 32$. Le script affiche 32 puis 7 : l'ordre des instructions change le résultat.

Ex. 4 ★★☆ **Températures inversées**

(a)

Étape	matin	soir
Ligne 2	12	19
Ligne 3	19	19
Ligne 4	19	19

À la ligne 3, matin reçoit la valeur de soir : 12 est effacé. À la ligne 4, soir reçoit cette nouvelle valeur, 19.

(b) Le script affiche 19 puis 19. Les valeurs ne sont pas échangées : la 3e ligne a remplacé 12 par 19 dans matin, donc 12 est perdu. La 4e ligne recopie 19 dans soir.

(c) Après la ligne 4, matin et soir valent tous les deux 19 : ecart vaut $19 - 19 = 0$. Ben attendait $19 - 12 = 7$ °C. La valeur 12, perdue à la ligne 3, fausse aussi les calculs suivants.

(d) On met d'abord l'ancienne valeur de côté dans une variable intermédiaire :

```
matin = 12
soir = 19
temp = matin
matin = soir
soir = temp
print(matin)
print(soir)
```

Le script affiche 19 puis 12 : matin vaut bien 19 et soir 12, l'échange fonctionne.

Ex. 5 ★★☆ **Les points d'un jeu**

(a) Le script affiche 46, et non 10 : il n'a pas fait la somme des points.

(b) `input` renvoie toujours une chaîne : a contient `"4"`, de type str. Avec deux chaînes, `+` les met bout à bout : `"4" + "6"` donne `"46"`.

Ex. 5 **(suite)**

(c) On convertit chaque chaîne en entier :

```
a = int(input("Points manche 1 : "))
b = int(input("Points manche 2 : "))
print(a + b)
```

Si elle tape 4 puis 6, le script affiche 10 : il fait bien la somme.

(d) `taille = float(input("Taille en mètres : "))` : une taille peut avoir une virgule, donc on convertit en float et non en int. Si elle tape 1,62, `float` provoque une erreur, car Python attend un point : il faut taper 1.62.

Ex. 6 ★★★ **Le chrono du trail**

(a) $8\,000 = 3\,600 \times 2 + 800$: cela fait 2 h, et il reste 800 s. Ensuite, $800 = 60 \times 13 + 20$: cela fait 13 min, et il reste 20 s. Conclusion : 8 000 s font 2 h 13 min 20 s.

(b)

```
duree = int(input("Durée en secondes : "))
heures = duree // 3600
reste = duree % 3600
minutes = reste // 60
secondes = reste % 60
print(heures, minutes, secondes)
```

Pour 5000 : $5\,000 = 3\,600 \times 1 + 1\,400$, puis $1\,400 = 60 \times 23 + 20$. Le programme affiche `1 23 20` : la course a duré 1 h 23 min 20 s.

(c) `/` donne toujours un flottant et non le quotient entier : `8000 / 3600` vaut environ 2,22 et `800 / 60` environ 13,33. La variable reste, calculée avec `%`, vaut toujours 800. Le programme affiche `2.2222222222222223 13.333333333333334 20` : les heures et les minutes ne sont pas entières. Pour obtenir un quotient entier, il faut `//`.

(d) Sans `int`, `input` renvoie la chaîne `"8000"`, de type str, et non l'entier 8 000. Dans le programme de Léo, la ligne 2 est `heures = duree / 3600`. On ne peut pas diviser une chaîne par un entier, donc Python s'arrête sur une erreur à la ligne 2. Il faut écrire `int(input(…))`.

(e) Le signe = n'est pas une égalité mathématique, c'est une affectation : Python calcule $8\,000 + 30 = 8\,030$ puis range ce résultat dans `duree`, ce qui efface l'ancienne valeur. Ensuite, $8\,030 = 3\,600 \times 2 + 830$ et $830 = 60 \times 13 + 50$. Le programme affiche `2 13 50` : avec la pénalité, cela fait 2 h 13 min 50 s.

Ex. 7 ★★★ **Soldes : l'ordre compte**

(a)

Ligne	prix
2	80
3	60.0
4	50.0

À la ligne 3, Python calcule $80 \times 0{,}75 = 60$. Comme `0.75` est un flottant, le résultat est le flottant `60.0`. Le programme A affiche `50.0` : la veste coûte 50 €.

(b)

Ligne	prix
8	80
9	70
10	52.5

Le programme B affiche `52.5` : la veste coûte 52,5 €. L'ordre des instructions change le résultat.

(c) Faux. A calcule $0{,}75p - 10$. B calcule $0{,}75(p - 10) = 0{,}75p - 7{,}5$. Les deux résultats diffèrent toujours de 2,5 € : B est toujours plus cher. Les instructions sont les mêmes, mais chacune utilise la valeur laissée par la précédente : l'ordre compte.

(d)

```
prix = float(input("Prix initial : "))
prix_a = prix * 0.75 - 10
prix_b = (prix - 10) * 0.75
print(prix_a, prix_b)
print(prix_a == prix_b)
```

Si l'on tape 80, le programme affiche `50.0 52.5` puis `False`. Les parenthèses de `prix_b` sont indispensables, et on écrit `0.75` avec un point.

(e) En Python, `^` n'est pas la puissance : avec un flottant, il provoque une erreur. La puissance s'écrit `**` : `80 * 0.75 ** 2`. La puissance se calcule avant la multiplication : $0{,}75^2 = 0{,}5625$, puis $80 \times 0{,}5625 = 45$. Python affiche `45.0` : après deux baisses, la veste coûte 45 €.

Ex. 8 ★★★ **La rotation au volley**

(a)

Ligne	poste1	poste2	poste3
3	Lina	Tom	Sami
4	Tom	Tom	Sami
5	Tom	Sami	Sami
6	Tom	Sami	Tom

Le script affiche `Tom Sami Tom`. À la ligne 4, l'ancienne valeur `"Lina"` est effacée. Donc, à la ligne 6, `poste1` contient déjà `"Tom"` : Lina a disparu et Tom apparaît deux fois.

Ex. 8 **(suite)**

(b) On garde Lina en mémoire dans `aux` avant de l'effacer :

```
poste1 = "Lina"
poste2 = "Tom"
poste3 = "Sami"
aux = poste1
poste1 = poste2
poste2 = poste3
poste3 = aux
print(poste1, poste2, poste3)
```

Le script affiche `Tom Sami Lina` : la rotation est correcte.

(c) Départ : Lina, Tom, Sami. Après la 1re rotation : Tom, Sami, Lina. Après la 2e : Sami, Lina, Tom. Après la 3e : Lina, Tom, Sami. Il faut donc au minimum 3 rotations. Ensuite, $10 = 3 \times 3 + 1$, donc `10 % 3` vaut 1 : après 10 rotations, la situation est la même qu'après une seule. C'est Tom qui occupe le poste 1.

(d) `capitaine` contient `"Lina"`. L'affectation range la valeur de `poste1` au moment où elle est exécutée. Quand `poste1` change ensuite, `capitaine` ne change pas : les deux variables ne sont pas liées comme dans une égalité mathématique.

(e) Après la rotation, `poste3` contient `"Lina"`, donc le script affiche `True`. C'est une valeur de type booléen (bool). == compare deux valeurs, alors que = range une valeur : `print(poste3 = "Lina")` ne fait pas de comparaison et provoque une erreur.

2. *Instruction conditionnelle*

Ex. 9 ★☆☆ **Valeur d'une condition**

(a) $9 > 9$ est faux : `False`.

(b) $9 \geqslant 9$ est vrai : `True`
(la frontière est comprise avec >=).

(c) `b > 0` est faux, donc avec `and` : `False`.

(d) `a > 5` est vrai, il suffit avec `or` : `True`.

(e) `b < 0` vaut `True`, son contraire vaut `False`.

(f) $9 = 2 \times 4 + 1$, donc `a % 2` vaut 1 : `False`
(9 est impair).

(g) $-5 \leqslant -2$ et $-2 \leqslant 0$ sont vrais : `True`.

Ex. 10 ★☆☆ **Écrire un test sans erreur**

(a) Non : = est une affectation. Le test d'égalité s'écrit `if x == 5:`.

(b) Non : il manque les deux-points. On écrit `if vitesse >= 90:`.

Ex. 10 **(suite)**

(c) Python signale une erreur d'indentation : le bloc doit être décalé de 4 espaces, c'est ce décalage qui le délimite.

(d) Faux : le `else` est facultatif. Sans lui, si le test est faux, il ne se passe rien.

(e) `if n % 2 == 0:` (le reste de la division par 2 est nul). Par exemple, `26 % 2` vaut 0 : 26 est pair ; `33 % 2` vaut 1 : 33 est impair.

(f) `if 15 <= n <= 40:` ou `if n >= 15 and n <= 40:` (et non `or`, qui serait vrai pour tout `n`).

Ex. 11 ★★☆ **Tarif de cinéma**

(a) $9 < 14$: le premier test est vrai, la fonction renvoie 5. Pour 19, le premier test est faux et $19 < 24$: 7. Pour 40, les deux tests sont faux : `10`. Total :
$5 + 7 + 2 \times 10 = 32$. Le groupe paie 32 €.

(b) Pour 14 : $14 < 14$ est faux et $14 < 24$ est vrai : 7. Pour 24 : les deux tests sont faux : `10`. Les frontières sont correctes.

(c) Pour 24 : $24 \leqslant 24$ est vrai, la fonction renvoie 7 au lieu de `10`. Pour les autres âges, rien ne change (23 donne toujours 7 et 25 donne 10). Or « à partir de 24 ans » inclut 24 : il faut garder <.

(d) Le test le plus restrictif se place en premier. Sinon, `age < 14`, vrai pour 2 ans, donnerait 5 €.

```
def tarif(age):
    if age < 3:
        return 0
    elif age < 14:
        return 5
    elif age < 24:
        return 7
    else:
        return 10
```

Par exemple, `tarif(2)` renvoie `0` et `tarif(3)` renvoie `5`.

Ex. 12 ★★☆ **Badge de marche : trouver l'erreur**

(a) $12\,500 \geqslant 4\,000$: le premier test est vrai, donc la fonction renvoie `"bronze"`. Or $12\,500 \geqslant 10\,000$: on devrait obtenir `"or"`. Le résultat de la fonction de Lina est faux.

(b) Avec `elif`, seul le premier test vrai est exécuté. Tout nombre de pas supérieur ou égal à 4 000 s'arrête au premier test : « argent » et « or » ne sont jamais atteints. Il faut placer le cas le plus restrictif d'abord.

Ex. 12 **(suite)**

(c)
```
def badge(pas):
    if pas >= 10000:
        return "or"
    elif pas >= 7000:
        return "argent"
    elif pas >= 4000:
        return "bronze"
    else:
        return "aucun"
```
Ainsi, `badge(12500)` renvoie bien `"or"`.

(d) `"aucun"`, `"bronze"`, `"argent"`, `"or"`. Les valeurs frontières sont bien dans le bon badge grâce à `>=`.

Ex. 13 ★★☆ **Jeu de deux dés**

(a)
```
de1 = 3
de2 = 5
if de1 == de2:
    print("rejoue")
elif (de1 + de2) % 2 == 0:
    print("gagne 2 points")
else:
    print("perd")
```
Le test d'égalité s'écrit `==`, et la somme est paire quand son reste dans la division par 2 vaut 0.

(b) $3 \neq 5$: le premier test est faux. $3 + 5 = 8$ et `8 % 2` vaut 0 : le deuxième test est vrai. Le programme affiche `gagne 2 points`.

(c) `4 == 4` est vrai : le programme affiche `rejoue`. Seul le premier test vrai est exécuté, le `elif` n'est même pas examiné.

(d) Par exemple 1 et 6 : pas de double et $1 + 6 = 7$, `7 % 2` vaut 1. Les deux tests sont faux, le programme affiche `perd`.

Ex. 14 ★★★ **Années bissextiles : corriger et réordonner**

(a) $2032 = 4 \times 508$, donc le premier test est vrai : la fonction renvoie `True`. $2025 = 4 \times 506 + 1$: aucun test n'est vrai, elle renvoie `False`. $1800 = 4 \times 450$: le premier test est vrai, elle renvoie `True`. Or 1800 est divisible par 100 mais pas par 400 ($1800 = 4 \times 400 + 200$) : 1800 n'est pas bissextile. Le résultat `True` pour 1800 est faux.

(b) On arrive au premier `elif` seulement si `annee % 4 == 0` est faux. Comme $100 = 4 \times 25$ et $400 = 4 \times 100$, une année qui n'est pas divisible par 4 n'est divisible ni par 100 ni par 400. Les deux tests sont donc toujours faux quand Python les examine. La fonction de Tom revient à tester seulement la divisibilité par 4.

Ex. 14 **(suite)**

(c)
```
def bissextile(annee):
    if annee % 400 == 0:
        return True
    elif annee % 100 == 0:
        return False
    elif annee % 4 == 0:
        return True
    else:
        return False
```
Un multiple de 400 est aussi un multiple de 100 et de 4 : c'est le cas le plus restrictif, on le teste donc en premier. On teste ensuite les multiples de 100, puis ceux de 4.

(d)
```
def bissextile(annee):
    return (annee % 4 == 0 and not (annee % 100 == 0))
```
Le `and` traduit « divisible par 4 mais pas par 100 ». Le `or` traduit Le `not` exprime « pas divisible par 100 ».

(e) 2000 donne `True` (multiple de 400). 2100 donne `False` (multiple de 100, mais $2100 = 400 \times 5 + 100$). 2024 donne `True` (multiple de 4 mais pas de 100). 2026 donne `False` ($2026 = 4 \times 506 + 2$). Il y a une année par cas : multiple de 400, multiple de 100 seulement, multiple de 4 seulement, autre. Les deux versions renvoient ces quatre résultats corrects.

Ex. 15 ★★★ **Serre : and ou or ?**

(a) La condition est fausse à chaque fois, donc Python exécute le bloc du `else` : les trois appels renvoient `normal`. C'est correct pour 21, mais 12 et 31 sont hors de l'intervalle : il faudrait obtenir `alerte`. La fonction de Lou ne déclenche jamais l'alerte.

(b) Pour aucune valeur. Avec `and`, les deux conditions doivent être vraies en même temps. Or aucun nombre n'est à la fois inférieur à 18 et supérieur à 25 : la condition vaut toujours `False`. Il suffit qu'une des deux conditions soit vraie, donc on utilise `or` :
```
def etat(t):
    if t < 18 or t > 25:
        return "alerte"
    else:
        return "normal"
```
Après correction, on obtient `alerte` pour 12, `normal` pour 18 et pour 25, et `alerte` pour 31.

(c) `18 <= t <= 25` traduit « température correcte ». L'alerte en est le contraire : `if not 18 <= t <= 25:` (le `not` porte sur toute la comparaison enchaînée). Pour $t = 12$: `18 <= 12 <= 25` vaut `False`, donc `not` donne `True` : `alerte`. Pour $t = 25$: la comparaison vaut `True`, le test est faux : `normal`. Ce test équivaut à celui avec `or`.

Ex. 15 **(suite)**

(d)

```
t = 25
if t < 18:
    print("chauffage")
elif t > 25:
    print("ventilation")
```

$25 < 18$ est faux et $25 > 25$ est faux aussi. Aucun bloc n'est exécuté : le script n'affiche rien. Le `else` est facultatif.

(e) Script de Mia :

```
t = 25
if t <= 18:
    print("chauffage")
elif t >= 25:
    print("ventilation")
```

Les tests ne diffèrent que pour $t = 18$ et $t = 25$. Pour $t = 18$, $18 \leqslant 18$ est vrai : Mia affiche `chauffage`, alors que ton script n'affiche rien. Pour $t = 25$, Mia affiche `ventilation`, alors que ton script n'affiche rien. Pour toute autre valeur, les deux scripts affichent la même chose. Le script de Mia se trompe en 18 et en 25, qui sont des températures correctes.

Ex. 16 ★★★ **Jeu des cartes numérotées**

(a)

```
def points(n):
    if n % 6 == 0:
        return 10
    elif n % 2 == 0:
        return 3
    elif n % 3 == 0:
        return 2
    else:
        return 0
```

Un multiple de 6 est à la fois pair et multiple de 3 : c'est le cas le plus restrictif, on le teste en premier.

(b) $42 = 6 \times 7$, donc on obtient 10. 34 n'est pas un multiple de 6 mais il est pair, donc on obtient 3. 27 est impair et $27 = 3 \times 9$, donc on obtient 2. 35 est impair et $35 = 3 \times 11 + 2$, donc on obtient 0. La fonction renvoie donc 10, 3, 2 et 0.

(c) Pour 42, le test `42 % 6 == 0` est vrai : `return 10` renvoie 10 et termine aussitôt la fonction. Un `return` arrête la fonction : chaque `if` n'est donc examiné que si tous les tests précédents sont faux, comme avec `elif`. La fonction de Léa renvoie toujours la même valeur que la tienne : elle ne diffère pour aucun numéro de 1 à 50.

(d) C'est faux. Contre-exemple : 15 est impair et $15 = 3 \times 5$, donc `points(15)` vaut 2. Un nombre impair peut être multiple de 3. L'affirmation est fausse.

Ex. 16 **(suite)**

(e)

```
def manche(a, b):
    pa = points(a)
    pb = points(b)
    if pa > pb:
        return "A gagne"
    elif pa < pb:
        return "B gagne"
    else:
        return "égalité"
```

`points(30)` vaut 10, car $30 = 6 \times 5$. `points(44)` vaut 3, car 44 est pair et $44 = 6 \times 7 + 2$. Comme $10 > 3$, `manche(30, 44)` renvoie `A gagne`.

3. *Boucle bornée for*

Ex. 17 ★☆☆ **Lire un range**

(a) `range(5)` commence à 0 et s'arrête avant 5 : k vaut 0, 1, 2, 3, 4
(5 tours).

(b) k vaut 3, 4, 5, 6, 7 : la boucle fait $8 - 3 = 5$ tours.

(c) On part de 2 et on avance de 4 en 4 sans atteindre 15 : k vaut 2, 6, 10, 14.

(d) Faux. k vaut 10, 7, 4, 1. La valeur suivante serait -2, qui dépasse la borne 0, et cette borne est de toute façon exclue.

(e) Non. La borne de droite est exclue : k va de 1 à 99, soit 99 tours. Il faut écrire `for k in range(1, 101):`.

(f) k vaut 0, 5, 10, 15 : la dernière valeur est 15, mais la boucle fait 4 tours. Le nombre de tours n'est pas la dernière valeur de k.

Ex. 18 ★☆☆ **Choisir le bon range**

(a) `for k in range(12):` : on part de 0 et on s'arrête avant 12.

(b) `for k in range(5, 41):` : la borne de droite est exclue, donc on écrit 41. La boucle fait $41 - 5 = 36$ tours.

(c) `for k in range(2, 21, 2):` : pas de 2, et la borne 21 laisse passer 20.

(d) `for k in range(30, 4, -5):` : pas de -5, et la borne 4 laisse passer 5.

(e) k vaut 3, 4, 5, 6, 7, 8, 9 : c'est `range(3, 10)` et la dernière valeur est $3 + 7 - 1 = 9$.

Ex. 19 ★★☆ **Trace d'une boucle qui utilise k**

(a) `range(1, 5)` donne k = 1, 2, 3, 4 : la boucle fait $5 - 1 = 4$ tours.

(b) À chaque tour, a reçoit $2 \times a - k$: $2 \times 2 - 1 = 3$, puis $2 \times 3 - 2 = 4$, puis $2 \times 4 - 3 = 5$, puis $2 \times 5 - 4 = 6$.

Tour	k	a
Avant	—	2
1	1	3
2	2	4
3	3	5
4	4	6

(c) Le `print` est aligné sur le `for` : il ne s'exécute qu'une fois, après la boucle. Le programme affiche 6.

(d) Le `print` est maintenant dans la boucle : il s'exécute à chaque tour. Le programme affiche 3, 4, 5 et 6, un nombre par ligne.

Ex. 20 ★★☆ **Le programme de pompes de Noé**

(a) À chaque tour, total est remis à 0 puis reçoit $3 \times k$. Après la boucle, il ne reste que le dernier tour, avec k = 19 : le programme affiche $3 \times 19 = 57$.

(b) Erreur 1 : `total = 0` est dans la boucle, donc la somme repart de zéro à chaque tour. Il faut l'écrire avant le `for`. Erreur 2 : `range(1, 20)` s'arrête à 19, donc le jour 20 est oublié. Il faut écrire `range(1, 21)`.

(c)
```
total = 0
for k in range(1, 21):
    total = total + 3 * k
print(total)
```

(d) total vaut $3 + 6 + 9 + \cdots + 60 = 3 \times (1 + 2 + \cdots + 20) = 3 \times 210 = 630$. Le programme affiche 630 : Noé fait 630 pompes en tout.

Ex. 21 ★★☆ **Population d'une ville sur 10 ans**

(a) Augmenter de 4 %, c'est multiplier par $1 + \frac{4}{100} = 1{,}04$.

(b) Le compteur part de 0 et augmente de 1 à chaque année où la population dépasse 10 000. Pour aller de 1 à 10, on écrit `range(1, 11)`, ce qui fait 10 tours.

```
pop = 8000
c = 0
for annee in range(1, 11):
    pop = pop * 1.04
    if pop > 10000:
        c = c + 1
print(round(pop))
print(c)
```

Ex. 21 (suite)

(c) Après 10 ans, la population vaut $8000 \times 1{,}04^{10} \approx 11\,842$. Elle vaut environ 9 733 après 5 ans et environ 10 123 après 6 ans : elle dépasse 10 000 de l'année 6 à l'année 10, donc c vaut 5. Le programme affiche 11842, puis 5.

(d) Il remplace `if pop > 10000:` par `if pop > 9000:`. La population vaut environ 8 999 après 3 ans et environ 9 359 après 4 ans : elle dépasse 9 000 de l'année 4 à l'année 10, donc c vaut 7.

Ex. 22 ★★★ **Les tickets gagnants de la tombola**

(a) k est un multiple de 3 quand le reste de sa division par 3 est nul. La condition s'écrit : `k % 3 == 0 and k % 5 != 0`.

(b) Le compteur part de 0, avant la boucle. Pour aller de 1 à 200 inclus, on écrit `range(1, 201)` :

```
c = 0
for k in range(1, 201):
    if k % 3 == 0 and k % 5 != 0:
        c = c + 1
print(c)
```

Le programme affiche 53 : il y a 53 tickets gagnants.

(c) Non, Lina n'a pas raison. `range(1, 200)` s'arrête à 199 : le ticket 200 n'est jamais testé. Ici, l'affichage reste 53 par chance, car 200 n'est pas un multiple de 3 et ne gagne donc pas. Avec 201 tickets, son programme oublierait le ticket 201, qui est gagnant ($201 = 3 \times 67$, et 201 n'est pas un multiple de 5). Pour aller jusqu'à 200, il faut écrire `range(1, 201)`.

(d) Entre 1 et 200, il y a `200 // 3` = 66 multiples de 3 ($3 \times 66 = 198$). Un multiple de 3 qui est aussi multiple de 5 est un multiple de 15 : il y en a `200 // 15` = 13 ($15 \times 13 = 195$). On les retire : $66 - 13 = 53$. On retrouve bien 53 tickets gagnants.

(e) Avec un pas de 3, k prend les valeurs 3, 6, 9, …, 198 : ce sont tous des multiples de 3. Il ne reste donc à tester que `k % 5 != 0`. k prend $\frac{198-3}{3} + 1 = 66$ valeurs, donc la boucle fait 66 tours, contre 200 dans la première version :

```
c = 0
for k in range(3, 201, 3):
    if k % 5 != 0:
        c = c + 1
print(c)
```

Ce programme affiche aussi 53.

Ex. 23 ★★★ **La carte de cinéma à tarif dégressif**

(a) Les 5 premières places coûtent $5 \times 8 = 40$ €, les 5 suivantes $5 \times 6 = 30$ €, puis les places n° 11 et 12 coûtent $2 \times 4 = 8$ €. Total : $40 + 30 + 8 = 78$ €.

(b) La somme part de 0, avant la boucle. k doit aller de 1 à n inclus, donc on écrit `range(1, n + 1)`. Les places n° 6 à 10 vérifient `k <= 10` (le cas `k <= 5` est déjà traité par le `if`) :

```
def prix_total(n):
    S = 0
    for k in range(1, n + 1):
        if k <= 5:
            S = S + 8
        elif k <= 10:
            S = S + 6
        else:
            S = S + 4
    return S
```

`prix_total(12)` renvoie bien 78.

(c) Avec `k < 5`, la place n° 5 ne passe plus dans le `if` : elle tombe dans le `elif` et coûte 6 €. Places 1 à 4 : $4 \times 8 = 32$; places 5 à 10 : $6 \times 6 = 36$; places 11 et 12 : $2 \times 4 = 8$. La fonction renvoie $32 + 36 + 8 = 76$, soit $78 - 76 = 2$ € de moins : la place n° 5 est comptée à 6 € au lieu de 8 €.

(d) `range(12)` donne k = 0, 1, …, 11 : la boucle fait bien 12 tours, mais les numéros sont décalés de 1. Les valeurs 0 à 5 vérifient `k <= 5` : $6 \times 8 = 48$; k = 6 à 10 : $5 \times 6 = 30$; k = 11 : 4. La fonction renvoie $48 + 30 + 4 = 82$ au lieu de 78. Une place de trop est comptée à 8 € et une de moins à 4 €, d'où l'écart de $82 - 78 = 4$ €.

(e) 10 places coûtent $40 + 30 = 70$ €. Il reste 30 € et chaque place suivante coûte 4 € : $7 \times 4 = 28 \leqslant 30$, mais $8 \times 4 = 32 > 30$.
Donc `prix_total(17)` renvoie $70 + 28 = 98 \leqslant 100$ et `prix_total(18)` renvoie $102 > 100$. Avec 100 €, on peut acheter au maximum 17 places.

Ex. 24 ★★★ **L'élevage de lapins**

(a) `range(6)` donne k = 0, 1, 2, 3, 4, 5 : la boucle fait 6 tours et la dernière valeur de k est 5. Le nombre de tours n'est pas égal à la dernière valeur de k, car k commence à 0.

Ex. 24 **(suite)**

(b) À chaque tour, c reçoit a + b, puis a prend l'ancienne valeur de b, puis b prend la valeur de c :

Tour	k	c	a	b
Avant	—	—	1	1
1	0	2	1	2
2	1	3	2	3
3	2	5	3	5
4	3	8	5	8
5	4	13	8	13
6	5	21	13	21

Le print est placé après la boucle. Le programme affiche donc 21, une seule fois.

(c) On crée un accumulateur total, initialisé à 0 AVANT la boucle. On lui ajoute b à la fin de chaque tour, après `b = c` :

```
a = 1
b = 1
total = 0
for k in range(6):
    c = a + b
    a = b
    b = c
    total = total + b
print(b)
print(total)
```

D'après la trace, total vaut $2 + 3 + 5 + 8 + 13 + 21 = 52$. Le programme affiche 21, puis 52.

(d) Après `a = b`, a et b sont égaux.
Donc `b = a + b` double b à chaque tour : b vaut 2, 4, 8, 16, 32 puis 64. Le programme affiche 64. L'ancienne valeur de a est perdue avant de servir : c'est pourquoi Ben la sauvegarde dans la variable c.

(e) D'après la trace, après chaque tour, b vaut successivement 2, 3, 5, 8, 13, 21. En continuant, on obtient 34, 55, 89 et 144 ($13 + 21 = 34$, $21 + 34 = 55$, $34 + 55 = 89$, $55 + 89 = 144$). Il faut 10 tours pour obtenir 144, donc on écrit `range(10)`.

4. *Boucle non bornée while*

Ex. 25 ★☆☆ **Automatismes : écrire la bonne condition**

(a) La condition d'arrêt est $s > 500$. Son contraire est $s \leqslant 500$: `while s <= 500:`

(b) La condition d'arrêt est $t < 20$. Son contraire est $t \geqslant 20$: `while t >= 20:`

Ex. 25 **(suite)**

(c) Non. « Atteint 100 » veut dire *score* $\geqslant 100$. Avec `<=`, la boucle continue quand le score vaut exactement 100. Il faut écrire `while score < 100:`.

(d) Une boucle `while` : on ne connaît pas à l'avance le nombre d'années (le nombre de tours). On répète tant que l'épargne est inférieure au double.

(e) 0 fois : $0 > 0$ est faux dès le départ, donc le bloc n'est jamais exécuté.

(f) Non. En mathématiques, $10 \times 0{,}1 = 1$. Mais les flottants sont codés de façon approchée : après 10 ajouts, `x` vaut 0.9999999999999999 et non 1, puis environ 1.1. La condition `x != 1` reste vraie pour toujours : c'est une boucle infinie. Il faut écrire `while x < 1:`.

Ex. 26 ★☆☆ **Automatismes : nombre de tours et valeur à la sortie**

(a) `a` vaut successivement 6, 12, 24, 48 puis 96. La condition $96 < 50$ est fausse : à la sortie, `a` vaut 96 (et non 48), la première valeur qui dépasse le seuil.

(b) `n` vaut 75, 50 puis 25. Comme $25 < 30$, la boucle s'arrête : 3 tours.

(c) 13, 14 et 15 ne sont pas multiples de 4, mais 16 l'est : `m` vaut 16 à la sortie.

(d) `c` vaut 2, 4, 6, 8… : il « saute » par-dessus 7. La condition reste vraie : c'est une boucle infinie.

(e) Le programme affiche 3 sans jamais s'arrêter : le bloc ne modifie pas `t`, donc $t < 10$ reste vrai (boucle infinie).

(f) `p` vaut 10, 100 puis 1000. Comme $1000 \leqslant 1000$ est vrai, la boucle fait un tour de plus (`p` devient 10000) : 4 tours. Avec `<`, la condition $1000 < 1000$ est fausse : 3 tours seulement.

Ex. 27 ★★☆ **Épargne : trace d'un seuil**

(a) Une hausse de 4 % revient à multiplier par $1 + \frac{4}{100} = 1{,}04$. La ligne 4 calcule donc le capital de l'année suivante.

Ex. 27 **(suite)**

(b)

Étape	capital	annees	Test
Début	2500	0	Vrai
Tour 1	2600	1	Vrai
Tour 2	2704	2	Vrai
Tour 3	2812.16	3	Vrai
Tour 4	2924.65	4	Vrai
Tour 5	3041.63	5	Faux

Après le tour 5, la condition est fausse : la boucle s'arrête.

(c) Le programme affiche 5. À la sortie, `capital` vaut environ 3 041,63 €. C'est la première valeur qui atteint 3 000 €, pas la dernière en dessous.

(d) Le programme affiche 2. Vérification : $2\,800 \times 1{,}04 = 2\,912 < 3\,000$ et $2\,800 \times 1{,}04^2 = 3\,028{,}48 \geqslant 3\,000$. Le seuil est franchi pendant la 2^{e} année : avec 2 800 €, Ben atteint 3 000 € au bout de 2 ans.

(e) On s'arrête dès que *capital* $> 3\,000$, donc on continue tant que *capital* $\leqslant 3\,000$: `while capital <= 3000:`. L'affichage reste 5, car aucun capital ne vaut exactement 3 000 €. La différence entre « atteint » et « dépasse » ne change le résultat que si le seuil est atteint pile.

Ex. 28 ★★☆ **Voiture d'occasion : programme à compléter**

(a) Perdre 15 % revient à multiplier par $1 - \frac{15}{100} = 0{,}85$.

(b) On s'arrête dès que la valeur est inférieure à 6 000 € (*valeur* $< 6\,000$). On continue donc tant que *valeur* $\geqslant 6\,000$: `while valeur >= 6000:`

(c)
```
valeur = 18000
annees = 0
while valeur >= 6000:
    valeur = valeur * 0.85
    annees = annees + 1
print(annees)
```

Le compteur part de 0 et augmente de 1 à chaque tour.

(d) Le programme affiche 7. Vérification : $18\,000 \times 0{,}85^6 \approx 6\,788{,}69 \geqslant 6\,000$ et $18\,000 \times 0{,}85^7 \approx 5\,770{,}39 < 6\,000$. La voiture vaut moins de 6 000 € au bout de 7 ans.

(e) Son programme affiche 8. La boucle fait toujours 7 tours, mais le compteur part de 1 au lieu de 0 : le résultat est décalé d'une unité. Avant la boucle, aucune année n'est passée, donc il faut `annees = 0`.

Ex. 29 ★★★ **Jeu de dés : attendre le 6**

(a) La condition $4 \neq 6$ est vraie. Le bloc augmente `lancers`, mais ne modifie jamais `de` : la condition reste vraie pour toujours. C'est une boucle infinie, et la fonction ne renvoie rien.

(b) Il faut relancer le dé dans le bloc :

```
from random import randint

def attente_six():
    de = randint(1, 6)
    lancers = 1
    while de != 6:
        de = randint(1, 6)
        lancers = lancers + 1
    return lancers
```

Un lancer est déjà fait avant la boucle, d'où `lancers = 1`. La fonction est maintenant correcte.

(c) `de` est un entier tiré à chaque tour parmi 1, 2, 3, 4, 5, 6 : il peut valoir exactement 6, sans arrondi ni saut par-dessus 6. Avec `x = x + 0.1`, le flottant est approché et peut ne jamais valoir exactement 1.

(d)

```
from random import randint

def attente_six():
    de = randint(1, 6)
    lancers = 1
    while de != 6:
        de = randint(1, 6)
        lancers = lancers + 1
    return lancers

total = 0
for k in range(10000):
    total = total + attente_six()
print(total / 10000)
```

La moyenne change à chaque exécution, mais elle reste toujours proche de 6 : en moyenne, il faut environ 6 lancers pour obtenir un 6.

Ex. 30 ★★★ **Tasse de thé : corriger un programme de seuil**

(a) Au départ, `temp` vaut 85. La condition `temp < 40` est fausse dès le premier test, donc le bloc n'est jamais exécuté. Le programme affiche directement la valeur de départ de `minutes` : **1**. La condition est inversée : Lina a écrit la condition d'arrêt au lieu de la condition pour continuer.

(b) Elle écrit `while temp >= 40:`. Cette condition est vraie au départ ($85 \geq 40$). Mais le bloc ne modifie que `minutes` : `temp` reste à 85 pour toujours. La condition reste donc vraie et le programme ne s'arrête jamais : c'est une **boucle infinie**. Le bloc doit modifier la variable testée.

Ex. 30 **(suite)**

(c) Baisser de 6 %, c'est multiplier par $1 - \frac{6}{100} = 0{,}94$. On continue tant que $temp \geqslant 40$ (contraire de « passer sous 40 ») :

```
temp = 85
minutes = 0
while temp >= 40:
    temp = temp * 0.94
    minutes = minutes + 1
print(minutes)
```

Le bloc fait baisser `temp` et ajoute 1 au compteur à chaque minute.

(d) La ligne 2 de Lina est `minutes = 1`. La boucle fait toujours le même nombre de tours, mais chaque tour ajoute 1 à partir de 1 : le programme afficherait **14** au lieu de 13. C'est un décalage d'une unité. Avant la boucle, aucune minute ne s'est écoulée : le compteur doit partir de 0.

(e) Le programme affiche **13**. Vérification : $85 \times 0{,}94^{12} \approx 40{,}45$, encore au-dessus de 40, donc la boucle continue ; $85 \times 0{,}94^{13} \approx 38{,}03$, sous 40, donc la condition devient fausse. Le seuil est franchi entre 12 et 13 minutes : le thé passe sous 40 °C au bout de 13 minutes.

Ex. 31 ★★★ **Course à pied : un objectif jamais atteint**

(a)

Tour	distance	semaines	Condition
Début	4	0	True
1	7	1	True
2	10	2	True
3	13	3	True
4	16	4	True
5	19	5	True
6	22	6	True

La distance augmente de 3 en 3 à partir de 4 : 4, 7, 10, 13, 16, 19, 22, 25… Elle saute de 19 à 22 et ne vaut jamais 20 ($20 - 4 = 16$ n'est pas un multiple de 3). La condition `distance != 20` reste toujours vraie : c'est une **boucle infinie**.

(b) On s'arrête dès que $distance \geqslant 20$, donc on continue tant que $distance < 20$: `while distance < 20:`. D'après la trace, la condition devient fausse au tour 6. Le programme affiche **6 22** : la valeur affichée est la première qui atteint le seuil, et ce n'est pas 20.

Ex. 31 **(suite)**

(c) **Faux.** « Atteindre 19 » : on s'arrête dès que *distance* $\geqslant 19$, donc `while distance < 19:`, qui s'arrête à 19 et affiche **5 19**. « Dépasser 19 » : on s'arrête dès que *distance* > 19, donc `while distance <= 19:`, qui continue à 19 et affiche **6 22**. Les deux réponses diffèrent d'une semaine.

(d) On continue tant que la distance est strictement inférieure à l'objectif :

```
def nb_semaines(objectif):
    distance = 4
    semaines = 0
    while distance < objectif:
        distance = distance + 3
        semaines = semaines + 1
    return semaines
```

`nb_semaines(42)` renvoie **13**. Vérification : $4 + 12 \times 3 = 40 < 42$ et $4 + 13 \times 3 = 43 \geqslant 42$. Ben court au moins 42 km après 13 semaines d'ajouts.

(e) Le nombre de tours est connu à l'avance (8) : on utilise une boucle **for**.

```
distance = 4
for k in range(8):
    distance = distance + 3
print(distance)
```

Il affiche **28**, car $4 + 8 \times 3 = 28$. La boucle while sert quand on ne connaît pas le nombre de tours.

Ex. 32 ★★★ **Deux villes : quand A dépasse-t-elle B ?**

(a) $12\,000 \times 1{,}03 = 12\,360$ et $18\,000 \times 0{,}98 = 17\,640$. Gagner 3 %, c'est multiplier par $1 + 0{,}03 = 1{,}03$; perdre 2 %, c'est multiplier par $1 - 0{,}02 = 0{,}98$. Après n années, on a multiplié n fois, d'où la puissance `** n`.

(b) On s'arrête dès que $A > B$, donc on continue tant que $A \leqslant B$. Le compteur n part de 0 :

```
def depassement():
    n = 0
    while ville_a(n) <= ville_b(n):
        n = n + 1
    return n
```

La fonction en appelle deux autres. À chaque tour, n augmente : la population de A grandit, celle de B diminue, donc la boucle finit par s'arrêter.

(c) Elle renvoie **9**. Pour n = 8 : $12\,000 \times 1{,}03^8 \approx 15\,201$ et $18\,000 \times 0{,}98^8 \approx 15\,314$: A est encore moins peuplée que B, donc la boucle continue. Pour n = 9 : $12\,000 \times 1{,}03^9 \approx 15\,657$ et $18\,000 \times 0{,}98^9 \approx 15\,007$: A dépasse B, donc la condition devient fausse. La ville A dépasse la ville B au bout de 9 ans.

Ex. 32 **(suite)**

(d) n avance par sauts de 1 : pour n = 8, A est plus petite que B ; pour n = 9, A est déjà plus grande. Les deux populations ne sont jamais exactement égales, d'autant que ce sont des flottants. Ensuite, l'écart grandit. La condition `!=` reste donc toujours vraie : la boucle est **infinie**. Il faut tester une inégalité.

(e) Le programme affiche **9** puis **None**. L'appel exécute `print(n)`, qui affiche 9. Mais la fonction ne renvoie rien : `x` reçoit la valeur `None`, et `print(x)` affiche None. Afficher n'est pas renvoyer : pour récupérer la valeur, il faut `return n`.

5. *Fonctions*

Ex. 33 ★☆☆ **Lire un appel de fonction**

(a) Le paramètre x prend la valeur $4 : 5 - 2 \times 4 = -3$. `g(4)` renvoie -3.

(b) x prend la valeur -1 : $5 - 2 \times (-1) = 5 + 2 = 7$. `g(-1)` renvoie 7.

(c) a prend la valeur 3 et b la valeur $7 : 3 \times 10 + 7 = 37$. L'appel renvoie 37.

(d) Ici a prend la valeur 7 et b la valeur $3 : 7 \times 10 + 3 = 73$. Le résultat n'est pas 37 : l'ordre des arguments compte.

(e) On calcule d'abord `h(0, 2)` $= 0 \times 10 + 2 = 2$, puis `g(2)` $= 5 - 4 = 1$. L'appel renvoie 1.

(f) `g(1)` $= 5 - 2 = 3$ et `h(1, 1)` $= 10 + 1 = 11$, donc y vaut $3 + 11 = 14$.

(g) h a deux paramètres, a et b, mais on ne lui donne qu'un seul argument : b n'a pas de valeur. Le nombre d'arguments doit être égal au nombre de paramètres.

Ex. 34 ★☆☆ **Fonction mathématique et fonction Python**

(a) `return 4 * x ** 2 - 5` : le signe * est obligatoire et le carré s'écrit ** 2.

(b) `return (x + 1) / 2` : il faut des parenthèses autour du numérateur.

(c) Python ne comprend pas « 3x » : il faut écrire le signe de multiplication, `return 3 * x - 1`.

(d) La fonction $x \mapsto (x - 2)^2$.

(e) $f(3) = 4 \times 9 - 5 = 31$, donc `f(3)` doit renvoyer 31.

(f) La division passe avant l'addition : son calcul donne $\frac{6}{2} + 1 = 4$, qui s'affiche 4.0. Il faut écrire `return 6 / (x + 1)` : le bon `k(2)` renvoie $\frac{6}{3} = 2$, qui s'affiche 2.0.

Ex. 35 ★★☆ Tarif de cinéma : print ou return

(a) Rien ne s'affiche : les deux fonctions sont seulement définies. Une fonction ne fait rien tant qu'on ne l'appelle pas.

(b) n prend la valeur 3 : $8 \times 3 + 5 = 29$. L'écran affiche 29. La ligne `print("Bon film")` n'est jamais exécutée, car return renvoie la valeur et termine tout de suite la fonction.

(c) `tarif_b(3)` affiche 29. `print(tarif_b(3))` affiche d'abord 29 (le print dans la fonction), puis None : sans return, la fonction ne renvoie rien d'autre que None.

```
29
29
None
```

(d) `tarif_b(3)` vaut None, et on ne peut pas calculer None + 2. Il faut remplacer print par return :

```
def tarif_b(n):
    return 8 * n + 5
```

Alors `total = tarif_b(3) + 2` vaut 31. Conclusion : pour réutiliser un résultat dans un calcul, on le renvoie avec return.

Ex. 36 ★★☆ Admis ou non : compléter un programme

(a) `return note >= 10` : la comparaison vaut déjà True ou False, pas besoin d'un if.

(b) Ligne 5 : `if est_admis(note):` ; ligne 7 : `else:`.

```
def est_admis(note):
    return note >= 10

def resultat(note):
    if est_admis(note):
        return "admis"
    else:
        return "non admis"
```

La fonction resultat appelle est_admis, et chaque branche se termine par son propre return.

(c) 9,5 est inférieur à 10 : `est_admis(9.5)` renvoie False. L'argument `7.5 + 2.5` est calculé avant l'appel : il vaut 10.0. Comme $10 \geqslant 10$, `est_admis(10.0)` renvoie True, et `resultat(7.5 + 2.5)` renvoie "admis".

(d) Oui, il a raison. Si la note est supérieure ou égale à 10, return True renvoie la valeur et termine la fonction : la dernière ligne n'est pas exécutée. Sinon, on saute le if et on arrive à return False. Par exemple, sa fonction renvoie True pour 12 et False pour 7.

Ex. 37 ★★☆ Vitesse moyenne d'un coureur

(a) La distance et la durée changent d'une course à l'autre : ce sont les paramètres.

```
def vitesse(distance, duree):
    return distance / duree
```

Test : 10 km en 2 h donnent 5 km/h, et `vitesse(10, 2)` renvoie bien 5.0.

(b) $15 \div 1{,}25 = 12$: l'appel renvoie 12.0, soit une vitesse moyenne de 12 km/h.

(c) distance prend la valeur 1.25 et duree la valeur 15 : $1{,}25 \div 15 \approx 0{,}08$. Une vitesse de 0,08 km/h est absurde pour un coureur : les arguments sont donnés dans le mauvais ordre.

(d) Une minute vaut $\frac{1}{60}$ h.

```
def temps_en_heures(h, m):
    return h + m / 60
```

On a `temps_en_heures(2, 30)` $= 2 + \frac{30}{60} = 2{,}5$, puis $25 \div 2{,}5 = 10$. L'appel renvoie 10.0 : la vitesse moyenne est de 10 km/h.

Ex. 38 ★★★ Location de kayaks

(a) Le nombre d'heures change d'une location à l'autre : c'est le paramètre. On a deux cas, donc deux return.

```
def prix_kayak(heures):
    if heures <= 2:
        return 12
    else:
        return 12 + 5 * (heures - 2)
```

Test à la main : $2 \leqslant 2$, donc `prix_kayak(2)` renvoie 12. Pour 5 heures : $12 + 5 \times (5 - 2) = 12 + 15 = 27$, donc `prix_kayak(5)` renvoie 27. La fonction donne bien 12 € et 27 €.

(b) La fonction `prix_groupe` réutilise `prix_kayak` au lieu de refaire le calcul.

```
def prix_groupe(heures, personnes):
    total = prix_kayak(heures) * personnes
    if personnes >= 4:
        return total - 10
    return total
```

Pour `prix_groupe(3, 4)` : `prix_kayak(3)` renvoie $12 + 5 \times 1 = 17$, donc total vaut $17 \times 4 = 68$. Comme $4 \geqslant 4$, la fonction renvoie $68 - 10 = 58$. Le groupe paie 58 €.

Ex. 38 **(suite)**

(c) Les arguments se placent dans l'ordre des paramètres : heures $= 4$ et personnes $= 3$. `prix_kayak(4)` renvoie $12 + 5 \times 2 = 22$, donc total vaut $22 \times 3 = 66$. Comme $3 < 4$, il n'y a pas de réduction. Léo obtient 66 au lieu de 58 : il a inversé les deux arguments. Il fallait taper `prix_groupe(3, 4)`.

(d) La fonction a deux paramètres, mais l'appel ne donne qu'un argument. Python ne peut pas donner de valeur à personnes : il s'arrête sur une erreur (TypeError, il manque l'argument personnes). Rien n'est calculé : à l'appel, il faut autant d'arguments que de paramètres.

Ex. 39 ★★★ **Températures : corriger le programme de Noé**

(a) Avec les parenthèses, Python ajoute 32 avant de multiplier : $1{,}8 \times (0 + 32) = 57{,}6$. Or la formule $F = 1{,}8 \times C + 32$ multiplie d'abord par 1,8, puis ajoute 32. On corrige en écrivant `f = 1.8 * c + 32`. On retrouve alors $1{,}8 \times 0 + 32 = 32$.

(b) L'appel calcule $f = 1{,}8 \times 25 + 32 = 45 + 32 = 77$. Le print placé dans la fonction affiche donc 77.0. La fonction n'a pas de return : elle renvoie None, et c'est ce que le print extérieur affiche. L'écran montre :

```
77.0
None
```

(c) `fahrenheit(30)` et `fahrenheit(10)` renvoient None. Python ne sait pas calculer None - None : il s'arrête sur une erreur (TypeError). Afficher une valeur ne revient pas à la renvoyer. Il faut remplacer print par return :

```
def fahrenheit(c):
    return 1.8 * c + 32
```

(d) `fahrenheit(30)` renvoie $1{,}8 \times 30 + 32 = 86$ et `fahrenheit(10)` renvoie $1{,}8 \times 10 + 32 = 50$.
Donc `ecart(10, 30)` renvoie $86 - 50 = 36$, ce qui s'affiche 36.0. Vérification : un écart de 20 °C correspond à $1{,}8 \times 20 = 36$ °F. L'écart est de 36 °F.

(e) Faux. return renvoie la valeur et termine aussitôt la fonction : la ligne qui le suit n'est jamais exécutée. `y = fahrenheit(-40)` range $1{,}8 \times (-40) + 32 = -72 + 32 = -40$ (soit -40.0) dans y, et l'écran reste vide. La ligne ajoutée ne sert à rien.

Ex. 40 ★★★ **Années bissextiles**

(a) d divise n quand le reste de la division de n par d est nul. On écrit donc `return n % d == 0` : la comparaison vaut True ou False, et la fonction renvoie ce booléen. Comme $2026 = 4 \times 506 + 2$, on a `2026 % 4` qui vaut 2.
Donc `divisible(2026, 4)` renvoie False.

(b) Ligne 8 : `return False` (divisible par 100 mais pas par 400). Ligne 10 : `return divisible(annee, 4)`.

```
def est_bissextile(annee):
    if divisible(annee, 400):
        return True
    elif divisible(annee, 100):
        return False
    else:
        return divisible(annee, 4)
```

1900 : le reste par 400 vaut 300, mais 1900 est divisible par 100, donc False. 2000 : divisible par 400, donc True. 2028 : ni par 400 ni par 100, mais $2028 = 4 \times 507$, donc True. Les résultats sont False, True et True.

(c) La boucle va jusqu'à fin, donc on écrit `range(debut, fin + 1)`, et on ajoute 1 au compteur à chaque année bissextile.

```
def nb_bissextiles(debut, fin):
    compteur = 0
    for annee in range(debut, fin + 1):
        if est_bissextile(annee):
            compteur = compteur + 1
    return compteur
```

De 1901 à 2000, les années bissextiles sont 1904, 1908, …, 2000. 1900 n'en fait pas partie, et 2000 compte car elle est divisible par 400. Il y en a $\frac{2000-1904}{4} + 1 = 25$. La fonction renvoie 25.

(d) `range(1901, 2000)` s'arrête à 1999 : l'année 2000 n'est jamais testée. Les années bissextiles comptées vont de 1904 à 1996, soit $\frac{1996-1904}{4} + 1 = 24$. La fonction renvoie 24 au lieu de 25 : il manque l'année 2000.

(e) Non. Une définition par def ne fait que créer la fonction : tant qu'on ne l'appelle pas, elle ne fait rien. Il n'y a aucun appel ni aucun print dans le fichier, donc l'écran reste vide. Pour tester, il faut ajouter par exemple `print(nb_bissextiles(1901, 2000))`, qui affiche 25.

6. *Simulation avec random*

Ex. 41 ★☆☆ **Lire les instructions du module random**

(a) On écrit `from random import randint, random`.

(b) Les deux bornes sont comprises : 3, 4, 5, 6 ou 7, soit $7 - 3 + 1 = 5$ valeurs possibles.

(c) La borne de fin est exclue : `k` prend les valeurs 3, 4, 5 et 6
(pas 7).

(d) Faux : `random()` renvoie un flottant de l'intervalle $[0\,;1[$, où 1 est exclu.

(e) `random()` renvoie un flottant (`float`) ; `randint(3, 7)` renvoie un entier
(`int`).

(f) `random() < 0.35` est vrai avec la probabilité 0,35, donc `random() >= 0.35` est vrai avec la probabilité $1 - 0{,}35 = 0{,}65$.

Ex. 42 ★☆☆ **Fréquence, moyenne et fluctuation**

(a) La fréquence vaut $\frac{c}{n} = \frac{46}{300} \approx 0{,}153$
(le compteur 46 seul n'est pas une fréquence).

(b) La probabilité vaut $\frac{1}{6} \approx 0{,}167$; l'écart vaut $\frac{1}{6} - \frac{46}{300} = \frac{4}{300} \approx 0{,}013$.

(c) Faux : la fréquence fluctue d'une simulation à l'autre. Un écart de 0,013 sur 300 lancers est normal et ne prouve pas que le dé est truqué.

(d) Faux : la simulation est aléatoire, le nombre de six change à chaque exécution ; on ne peut pas annoncer sa valeur exacte.

(e) La moyenne vaut $\frac{s}{n} = \frac{1090}{400} = 2{,}725$.

Ex. 43 ★★☆ **Lancers francs au basket**

(a) `random() < 0.72` est vrai avec la probabilité 0,72 : on écrit `if random() < 0.72:`.

Ex. 43 **(suite)**

(b) Le compteur part de 0, la boucle fait `n` tours et on divise par `n` :

```
from random import random

def lancer_franc():
    if random() < 0.72:
        return 1
    return 0

def frequence(n):
    c = 0
    for i in range(n):
        c = c + lancer_franc()
    return c / n
```

La fonction `frequence` appelle `n` fois la fonction qui simule un lancer.

(c) Elle renvoie `c`, le nombre de lancers réussis : un entier entre 0 et 500, souvent proche de $0{,}72 \times 500 = 360$. Ce n'est pas une fréquence : il faut le diviser par `n`.

(d) La fréquence fluctue d'une exécution à l'autre, et d'autant plus que n est petit. Sur 20 lancers, un écart de $0{,}72 - 0{,}55 = 0{,}17$ est possible. Pour $n = 10\,000$, la fréquence est proche de la probabilité (écart $0{,}72 - 0{,}7183 = 0{,}0017$). C'est donc 0.7183 qui estime le mieux 0,72, même si aucune des deux valeurs n'est exacte.

Ex. 44 ★★☆ **La somme de deux dés**

(a) `randint(2, 12)` renvoie l'un des 11 entiers de 2 à 12, chacun avec la même probabilité : $\frac{1}{11} \approx 0{,}091$.

(b) Les couples (1 ; 6), (2 ; 5), (3 ; 4), (4 ; 3), (5 ; 2) et (6 ; 1) donnent 7 : 6 couples sur 36. La probabilité vaut $\frac{6}{36} = \frac{1}{6} \approx 0{,}167$.

(c) Les sommes de deux dés ne sont pas équiprobables (7 sort plus souvent que 2), alors que `randint(2, 12)` les rend équiprobables. On simule chaque dé :

```
def somme_des():
    return randint(1, 6) + randint(1, 6)
```

Ici, les deux appels à `randint` simulent bien deux dés différents.

(d) Avant correction, la fréquence est proche de $\frac{1}{11} \approx 0{,}091$; après correction, elle est proche de la probabilité $\frac{1}{6} \approx 0{,}167$. Seul le programme corrigé estime correctement la probabilité du bonus.

Ex. 45 ★★☆ **Le stand de la kermesse**

(a) Trois faces sur six sont paires (2, 4 et 6) : la probabilité vaut $\frac{3}{6} = \frac{1}{2}$.

(b) $5 = 2 \times 2 + 1$, donc `5 % 2` vaut 1. Le reste de la division de `d` par 2 vaut 0 exactement quand `d` est pair : le test est vrai pour 2, 4 et 6, et faux pour 1, 3 et 5.

(c) La fonction affiche 2 ou -3 avec `print`, mais elle ne renvoie rien : sa valeur renvoyée est `None`. Python ne sait pas calculer `s + None`, d'où l'erreur. On remplace `print` par `return` :

```
def partie():
    d = randint(1, 6)
    if d % 2 == 0:
        return 2
    return -3
```

Maintenant, la fonction renvoie le résultat de la partie et on peut l'additionner.

(d) On additionne les résultats dans un accumulateur, puis on divise par `n` :

```
def moyenne(n):
    s = 0
    for i in range(n):
        s = s + partie()
    return s / n
```

(e) Total : $2 + 2 + 2 - 3 - 3 - 3 = -3$, soit $\frac{-3}{6} = -0{,}5$ point par partie. Pour n grand, `moyenne(n)` renvoie une valeur proche de $-0{,}5$, différente à chaque exécution. Le jeu est défavorable au joueur : il perd en moyenne un demi-point par partie.

Ex. 46 ★★★ **Sortir de prison au jeu de plateau**

(a) Il y a 6 issues équiprobables, dont 2 favorables (5 et 6). La probabilité vaut $\frac{2}{6} = \frac{1}{3} \approx 0{,}333$.

(b) `range(1, n)` va de 1 à $n - 1$. Pour $n = 500$, la boucle fait donc $500 - 1 = 499$ tours au lieu de 500. De plus, `return c` renvoie le compteur, c'est-à-dire le nombre de succès (un entier entre 0 et 499), et pas une fréquence. On corrige en écrivant `for i in range(n):` pour faire exactement n tours, puis `return c / n` : fréquence = compteur divisé par n.

Ex. 46 (suite)

(c) Chaque appel `lancer()` fait un nouveau lancer. Si le premier appel ne donne pas 5, Python relance le dé pour tester le 6 : on compare donc deux lancers différents. Il faut lancer une seule fois par tour et stocker le résultat dans une variable :

```
from random import randint

def lancer():
    return randint(1, 6)

def frequence_sortie(n):
    c = 0
    for i in range(n):
        d = lancer()
        if d == 5 or d == 6:
            c = c + 1
    return c / n
```

Maintenant, chaque tour simule bien un seul lancer.

(d) Si $a = 5$, la condition est vraie quel que soit b : 6 couples. Si $a \neq 5$, il faut $b = 6$: 5 couples, de $(1\,;6)$ à $(6\,;6)$ sauf $(5\,;6)$. Au total, $6 + 5 = 11$ couples sur 36. La probabilité vaut $\frac{11}{36} \approx 0{,}306$. Le programme de Malik donnerait donc des fréquences proches de 0,306 au lieu de 0,333 : son erreur fausse l'estimation.

(e) Non. La fréquence observée fluctue d'une exécution à l'autre. L'écart $0{,}342 - \frac{1}{3} \approx 0{,}009$ est très petit pour 500 lancers : c'est une fluctuation normale. Un tel écart ne prouve pas que le dé est truqué. On peut augmenter n : la fréquence sera alors, sauf exception, encore plus proche de $\frac{1}{3}$.

Ex. 47 ★★★ **Les sachets de graines de tomate**

(a) Le test est vrai quand le nombre tiré tombe dans $[0\,;0{,}8[$. Cet intervalle a pour longueur 0,8 : il représente une part 0,8 de l'intervalle $[0\,;1[$, qui a pour longueur 1. Comme `random()` répartit ses valeurs uniformément sur $[0\,;1[$, le test est vrai avec la probabilité 0,8.

(b)

```
from random import random

def germe():
    if random() < 0.8:
        return 1
    return 0
```

La fonction simule une seule graine : elle renvoie 1 avec la probabilité 0,8.

Ex. 47 **(suite)**

(c)

```
def semis():
    nb = 0
    for i in range(5):
        nb = nb + germe()
    return nb
```

Le compteur `nb` est initialisé à 0 avant la boucle. Il ajoute 1 à chaque graine qui germe. La fonction renvoie un entier compris entre 0 et 5.

(d)

```
def moyenne(n):
    s = 0
    for i in range(n):
        s = s + semis()
    return s / n
```

L'accumulateur `s` additionne les résultats des n sachets. `s / n` est la moyenne de la série obtenue.

(e) Chaque exécution fait de nouveaux tirages au hasard, donc la moyenne obtenue fluctue : on ne peut pas annoncer à l'avance la valeur exacte. Chaque graine germe avec la probabilité 0,8, donc en proportion 80 % des graines germent. Sur 5 graines, on attend environ $5 \times 0{,}8 = 4$ graines germées. Pour n grand, les moyennes obtenues sont donc proches de 4.

Ex. 48 ★★★ **En attendant le six aux petits chevaux**

(a)

Étape	tirage	tirage != 6	nb
Début	aucun	aucun	1
Test 1	2	True	2
Test 2	5	True	3
Test 3	6	False	3

Au 3e test, la condition est fausse, donc la boucle s'arrête. La fonction renvoie 3 : il a fallu 3 lancers pour obtenir le premier 6.

(b) Première erreur : avec `import random`, on ne peut pas écrire `randint` seul. Python ne le reconnaît pas et le programme s'arrête sur une erreur. Il faut écrire `from random import randint`. Seconde erreur : `randint(1, 7)` inclut ses deux bornes. Il renvoie donc un entier de 1 à 7 : Léo simule un dé à 7 faces. Le 6 sort alors avec la probabilité $\frac{1}{7} \approx 0{,}143$ au lieu de $\frac{1}{6} \approx 0{,}167$. Il faut écrire `randint(1, 6)`.

Ex. 48 **(suite)**

(c) À chaque tour, `s = attente()` remplace l'ancienne valeur de `s`. À la fin, `s` contient seulement la dernière attente, et `s / n` est cette attente divisée par n (proche de 0 pour n grand) : ce n'est pas une moyenne. Il faut un accumulateur :

```
def moyenne_attente(n):
    s = 0
    for i in range(n):
        s = s + attente()
    return s / n
```

On additionne les n résultats, puis on divise par n.

(d) Les tirages sont aléatoires : la moyenne fluctue d'une exécution à l'autre, et on ne peut pas annoncer sa valeur exacte. Les deux résultats sont proches de 6. Avec une probabilité de $\frac{1}{6}$, un 6 sort en proportion une fois sur six lancers. Il faut donc environ 6 lancers en moyenne pour obtenir un 6.

(e) Faux. À chaque tour, le tirage peut donner 6 (probabilité $\frac{1}{6}$). La condition finit donc par devenir fausse, et la boucle s'arrête. Elle serait réellement infinie si la condition ne pouvait jamais devenir fausse. Par exemple, avec `while randint(1, 5) != 6:`, 6 n'est jamais tiré.