



Correction du TD Python 1 Premiers pas : variables, conditions et boucles

Tous les scripts s'écrivent et s'exécutent dans Basthon (<https://basthon.fr>) ou Capytale. Pour chaque exercice, on écrit d'abord ce que l'on attend, puis on exécute, puis on compare.

Exercice 1 (★☆☆☆) obligatoire Prédire la sortie

Pour chacun des scripts suivants, écrire sur le papier ce qu'il affiche, en suivant l'évolution de chaque variable ligne par ligne. Vérifier ensuite en l'exécutant.

```
1. a = 7
2  b = 3
3  a = a + b
4  b = a - b
5  a = a - b
6  print(a, b)
```

Après la ligne 3, $a = 10$; après la ligne 4, $b = 10 - 3 = 7$; après la ligne 5, $a = 10 - 7 = 3$.
Les deux valeurs ont été échangées sans variable auxiliaire.

3 7

```
2. n = 2025
2  q = n // 100
3  r = n % 100
4  print(q, r, q * 100 + r)
5  print(n // 10 % 10)
```

$2025 = 100 \times 20 + 25$, donc $q = 20$ et $r = 25$. À la ligne 5, $//$ et $%$ ont la même priorité et se calculent de gauche à droite : $2025 // 10 = 202$ puis $202 \% 10 = 2$, le chiffre des dizaines.

20 25 2025
2

```
3. x = 2
2  y = x ** 3 - 3 * x ** 2 + 1
3  x = y
4  print(x, y)
5  print(-7 // 2, -7 % 2, 7 // -2)
```

$y = 8 - 12 + 1 = -3$, puis x reçoit cette valeur. Pour la dernière ligne, Python choisit le quotient de sorte que le reste soit compris entre 0 et le diviseur (exclu) lorsque le diviseur est positif : $-7 = 2 \times (-4) + 1$. Avec un diviseur négatif, le quotient est arrondi vers $-\infty$: $7 // (-2) = -4$.

-3 -3
-4 1 -4

Exercice 2 (★☆☆☆) Types, conversions et erreurs

1. Pour chacune des expressions suivantes, donner sa valeur et son type, ou indiquer le type de l'erreur qu'elle provoque. Vérifier en console.

- | | |
|------------------------------------|--------------------------------|
| a) <code>7 / 2</code> | f) <code>5 // 0</code> |
| b) <code>7 // 2 * 2 + 7 % 2</code> | g) <code>1e3 == 1000</code> |
| c) <code>"3" + "4"</code> | h) <code>0.1 * 3 == 0.3</code> |
| d) <code>int("3") + 4</code> | i) <code>type(2 ** 100)</code> |
| e) <code>"3" + 4</code> | j) <code>int(-3.7)</code> |

```
>>> 7 / 2
3.5
>>> 7 // 2 * 2 + 7 % 2
7
>>> "3" + "4"
'34'
>>> int("3") + 4
7
>>> "3" + 4
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> 5 // 0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 1e3 == 1000
True
>>> 0.1 * 3 == 0.3
False
>>> type(2 ** 100)
<class 'int'>
>>> int(-3.7)
-3
```

- ▷ a) 3.5, flottant : la division / renvoie toujours un flottant.
- ▷ b) 7, entier : c'est $bq + r$ avec $b = 2$, on retrouve le dividende.
- ▷ c) '34', chaîne : + met bout à bout deux chaînes.
- ▷ d) 7, entier, grâce à la conversion.
- ▷ e) **TypeError** : on ne peut pas additionner une chaîne et un entier.
- ▷ f) **ZeroDivisionError**, pour // comme pour / et %.
- ▷ g) **True** : 1000,0 et 1000 sont égaux, un entier et un flottant se comparent librement.
- ▷ h) **False** : 0,1 n'est pas représentable exactement en machine, $0,1 \times 3$ vaut 0.30000000000000004.
- ▷ i) **int** : les entiers de Python n'ont pas de taille limite.
- ▷ j) -3 : **int** tronque vers zéro, il n'arrondit pas et ne prend pas la partie entière au sens mathématique ($\lfloor -3,7 \rfloor = -4$).

2. Chacun des trois scripts suivants provoque une erreur. Sans l'exécuter, donner le type de l'erreur (**SyntaxError**, **NameError**, **TypeError**, **ZeroDivisionError**), puis le corriger.

- a) `prix = input("Prix ? ")`
`2 print(prix * 1.2)`
- b) `for k in range(5)`
`2 print(k)`
- c) `total = 0`
`2 for k in range(1, 5) :`

```

3     total = total + k
4     print(totale)

```

- ▷ a) `input` renvoie une chaîne; multiplier une chaîne par un flottant est impossible. Dernière ligne du message :

`TypeError: can't multiply sequence by non-int of type 'float'`

Correction : `prix = float(input("Prix ? "))`.

- ▷ b) Le deux-points manque après `range(5)` : le script n'est même pas lancé.

`SyntaxError: expected ':'`

Correction : `for k in range(5) :`

- ▷ c) Faute de frappe à la ligne 4 : le nom `totale` n'existe pas. Python suggère lui-même la correction.

`NameError: name 'totale' is not defined. Did you mean: 'total'?`

Correction : `print(total)`, qui affiche 10.

Exercice 3 (★☆☆☆☆) obligatoire Multiples et diviseurs

1. Écrire une boucle `for` qui affiche, sur une seule ligne, tous les multiples de 7 compris entre 1 et 100. On utilisera `range` avec un pas, et `print(k, end=" ")` pour rester sur la même ligne.
2. Écrire une boucle qui affiche tous les diviseurs positifs de 84.
3. Combien y a-t-il d'entiers de $\llbracket 1, 1000 \rrbracket$ divisibles à la fois par 4 et par 6 ? Écrire un script qui les compte, puis retrouver ce nombre par un raisonnement.

```

1 for k in range(7, 101, 7) :
2     print(k, end=" ")
3 print()
4 for d in range(1, 85) :
5     if 84 % d == 0 :
6         print(d, end=" ")
7 print()
8 compteur = 0
9 for k in range(1, 1001) :
10     if k % 4 == 0 and k % 6 == 0 :
11         compteur = compteur + 1
12 print(compteur)

```

```

7 14 21 28 35 42 49 56 63 70 77 84 91 98
1 2 3 4 6 7 12 14 21 28 42 84
83

```

- ▷ Ligne 1 : `range(7, 101, 7)` produit 7, 14, ..., 98; la borne 101 garantit que 98 est inclus. L'instruction `print()` seule (ligne 3) passe à la ligne.
- ▷ Ligne 4 : la borne 85 inclut le diviseur 84.
- ▷ Un entier est divisible par 4 et par 6 si et seulement s'il est divisible par leur plus petit commun multiple 12 (ce que le chapitre PGCD justifiera; on peut ici le vérifier à la main sur $\llbracket 1, 24 \rrbracket$). Les multiples de 12 dans $\llbracket 1, 1000 \rrbracket$ sont $12k$ avec $1 \leq k \leq \lfloor 1000/12 \rfloor = 83$.

Il y a 83 entiers de $\llbracket 1, 1000 \rrbracket$ divisibles par 4 et par 6.

Exercice 4 (★★☆☆☆) obligatoire Tables de vérité

On rappelle qu'une boucle sur (**True**, **False**) imbriquée dans une autre parcourt les quatre lignes d'une table de vérité à deux propositions.

1. Le *ou exclusif* de deux propositions P et Q est vrai lorsqu'exactlyement l'une des deux est vraie. Écrire un script qui affiche, sur chacune des quatre lignes, les valeurs de P , Q et des deux expressions

$$P \neq Q \quad \text{et} \quad (P \text{ or } Q) \text{ and not } (P \text{ and } Q).$$

Que constate-t-on ?

```

1 for P in (True, False) :
2     for Q in (True, False) :
3         print(P, Q, P != Q, (P or Q) and not (P and Q))

True True False False
True False True True
False True True True
False False False False

```

Les deux dernières colonnes coïncident : le ou exclusif s'écrit aussi bien $P \neq Q$ (les deux booléens sont différents) que « P ou Q , mais pas les deux ». Il est vrai sur exactement deux des quatre lignes, contrairement au **or** de Python, qui est le ou inclusif du cours.

2. Vérifier par un script que l'implication $P \implies Q$, programmée par (**not** P) **or** Q , a la même table de vérité que sa contraposée $\overline{Q} \implies \overline{P}$.

La contraposée $\overline{Q} \implies \overline{P}$ s'écrit (**not** (**not** Q)) **or** (**not** P).

```

1 for P in (True, False) :
2     for Q in (True, False) :
3         print(P, Q, ((not P) or Q) == ((not (not Q)) or (not P)))

True True True
True False True
False True True
False False True

```

Les deux implications ont la même valeur dans les quatre cas : une implication et sa contraposée sont équivalentes, ce qui fonde le raisonnement par contraposition du chapitre Divers modes de raisonnement.

3. Avec trois boucles imbriquées, vérifier sur les huit cas la distributivité

$$P \wedge (Q \vee R) \iff (P \wedge Q) \vee (P \wedge R).$$

On utilisera un accumulateur booléen **tout**, initialisé à **True**, qui reste vrai tant que tous les cas testés sont vrais, et l'on affichera seulement sa valeur finale.

```

1 tout = True
2 for P in (True, False) :
3     for Q in (True, False) :
4         for R in (True, False) :
5             gauche = P and (Q or R)
6             droite = (P and Q) or (P and R)
7             tout = tout and (gauche == droite)
8 print(tout)

True

```

L'accumulateur **tout** est le « et » de tous les tests : il ne peut devenir **False** qu'en rencontrant un cas où les deux membres diffèrent, et ne redevient jamais **True** ensuite. Sa valeur finale **True** signifie que les huit cas sont vérifiés.

Exercice 5 (★★☆☆) Sommes et conjectures

1. Écrire un script qui calcule la somme $1 + 3 + 5 + \dots + (2n - 1)$ des n premiers entiers impairs pour $n = 10$.

```
1 n = 10
2 somme = 0
3 for k in range(1, n + 1) :
4     somme = somme + (2 * k - 1)
5 print(somme)

100
```

2. Modifier le script pour afficher, pour chaque n de 1 à 10, la valeur de n , la somme, et n^2 . Conjecturer une formule.

La boucle de calcul est placée à l'intérieur d'une boucle sur n ; l'accumulateur est remis à zéro à chaque nouvelle valeur de n .

```
1 for n in range(1, 11) :
2     somme = 0
3     for k in range(1, n + 1) :
4         somme = somme + (2 * k - 1)
5     print(n, somme, n ** 2, somme == n ** 2)

1 1 1 True
2 4 4 True
3 9 9 True
4 16 16 True
5 25 25 True
6 36 36 True
7 49 49 True
8 64 64 True
9 81 81 True
10 100 100 True
```

Conjecture : $\sum_{k=1}^n (2k - 1) = n^2$. Elle se démontre par récurrence, ou directement :

$$\sum_{k=1}^n (2k - 1) = 2 \frac{n(n+1)}{2} - n = n^2.$$

3. Vérifier par un script que, pour tout $n \in \llbracket 1, 100 \rrbracket$, on a $\sum_{k=1}^n k^3 = \left(\frac{n(n+1)}{2} \right)^2$. On n'affichera qu'un seul booléen. Cette vérification constitue-t-elle une démonstration ?

```
1 tout = True
2 for n in range(1, 101) :
3     cubes = 0
4     for k in range(1, n + 1) :
5         cubes = cubes + k ** 3
6     tout = tout and (cubes == (n * (n + 1) // 2) ** 2)
7 print(tout)

True
```

Ce n'est pas une démonstration : cent cas vérifiés ne disent rien de $n = 101$ ni des suivants. La table de vérité de l'exercice précédent était, elle, une démonstration, parce que les cas

y sont en nombre fini et tous examinés. Ici, la démonstration est la récurrence du chapitre Récurrence.

Exercice 6 (★★☆☆) Une suite définie par récurrence

Soit (u_n) la suite définie par $u_0 = 3$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = 3u_n - 4$.

1. Écrire un script qui calcule et affiche u_{10} .

```
1 u = 3
2 for n in range(1, 11) :
3     u = 3 * u - 4
4     print(u)
```

59051

Dix tours de boucle font passer de u_0 à u_{10} . $u_{10} = 59\,051$.

2. Écrire un script qui détermine le plus petit entier n tel que $u_n > 10^6$, et affiche n et u_n .

```
1 u = 3
2 n = 0
3 while u <= 10 ** 6 :
4     u = 3 * u - 4
5     n = n + 1
6     print(n, u)
```

13 1594325

La condition de poursuite est la négation de la condition d'arrêt : on continue tant que $u_n \leq 10^6$. $n = 13$, $u_{13} = 1\,594\,325$.

3. Vérifier par un script que $u_n = 2 + 3^n$ pour $n \in \llbracket 1, 10 \rrbracket$.

```
1 u = 3
2 for n in range(1, 11) :
3     u = 3 * u - 4
4     print(n, u, 2 + 3 ** n)
```

1 5 5
2 11 11
3 29 29
4 83 83
5 245 245
6 731 731
7 2189 2189
8 6563 6563
9 19685 19685
10 59051 59051

Les deux colonnes coïncident. On retrouve $u_{13} = 2 + 3^{13} = 2 + 1\,594\,323$. La formule se démontre par récurrence : si $u_n = 2 + 3^n$, alors $u_{n+1} = 3(2 + 3^n) - 4 = 2 + 3^{n+1}$.

4. On remplace $u_0 = 3$ par $u_0 = 2$ dans le script de la question 2. Que se passe-t-il ? Expliquer, puis ajouter un garde-fou.

Le script ne s'arrête plus : $3 \times 2 - 4 = 2$, donc la suite est constante égale à 2 et ne dépasse jamais 10^6 . La condition `u <= 10 ** 6` reste vraie à chaque tour : c'est une boucle infinie, qu'il faut interrompre avec le bouton *Stop*. Avec un garde-fou sur le nombre de tours :

```

1 u = 2
2 n = 0
3 while u <= 10 ** 6 and n < 50 :
4     u = 3 * u - 4
5     n = n + 1
6 print(n, u)

50 2

```

La sortie « 50 tours et u toujours égal à 2 » signale que le seuil n'a pas été atteint et met sur la piste du point fixe. En général, un algorithme de seuil n'est légitime que si l'on a prouvé auparavant que la suite dépasse le seuil, par exemple parce qu'elle tend vers $+\infty$.

Exercice 7 (★★☆☆) obligatoire La suite de Syracuse

Partant d'un entier $n \geq 1$, on applique la règle suivante : si n est pair, on le remplace par $n/2$; sinon, on le remplace par $3n + 1$. On s'arrête lorsque n vaut 1. On conjecture (sans qu'aucune démonstration ne soit connue) que l'on atteint toujours 1.

1. Écrire un script qui, partant de $n = 27$, compte le nombre d'étapes nécessaires pour atteindre 1.

```

1 n = 27
2 etapes = 0
3 while n != 1 :
4     if n % 2 == 0 :
5         n = n // 2
6     else :
7         n = 3 * n + 1
8     etapes = etapes + 1
9 print(etapes)

111

```

On utilise `n // 2` et non `n / 2`, pour que `n` reste un entier : avec `/`, `n` deviendrait un flottant et le test `n != 1` resterait correct, mais les valeurs s'afficheraient avec `.0`.

Depuis 27, il faut 111 étapes.

2. Le modifier pour afficher aussi la plus grande valeur atteinte au cours du vol.

```

1 n = 27
2 etapes = 0
3 maximum = n
4 while n != 1 :
5     if n % 2 == 0 :
6         n = n // 2
7     else :
8         n = 3 * n + 1
9     etapes = etapes + 1
10    if n > maximum :
11        maximum = n
12 print(etapes, maximum)

111 9232

```

L'accumulateur `maximum` est initialisé avec la valeur de départ et mis à jour après chaque

étape. Le vol de 27 culmine à 9 232.

3. Parmi les entiers de départ de 1 à 100, lequel demande le plus d'étapes ? Combien ?

La boucle **while** est imbriquée dans une boucle **for** sur la valeur de départ **a** ; on recopie **a** dans **n** pour ne pas perdre la valeur de départ, et deux accumulateurs retiennent le record et l'entier qui l'a produit.

```
1 record = 0
2 depart = 0
3 for a in range(1, 101) :
4     n = a
5     etapes = 0
6     while n != 1 :
7         if n % 2 == 0 :
8             n = n // 2
9         else :
10            n = 3 * n + 1
11            etapes = etapes + 1
12        if etapes > record :
13            record = etapes
14            depart = a
15 print(depart, record)
```

97 118

L'entier 97 demande 118 étapes, le maximum pour un départ inférieur ou égal à 100.

Exercice 8 (★★☆☆) Nombres parfaits

Un entier $n \geq 2$ est *parfait* s'il est égal à la somme de ses diviseurs positifs autres que lui-même. Par exemple $6 = 1 + 2 + 3$.

1. Écrire un script qui calcule la somme des diviseurs de 28 autres que 28, et affiche si 28 est parfait.

```
1 n = 28
2 somme = 0
3 for d in range(1, n) :
4     if n % d == 0 :
5         somme = somme + d
6 print(somme, somme == n)
```

28 True

La borne **range(1, n)** exclut n lui-même. $28 = 1 + 2 + 4 + 7 + 14$ est parfait.

2. Écrire un script qui affiche tous les nombres parfaits inférieurs à 10 000.

```
1 for n in range(2, 10001) :
2     somme = 0
3     for d in range(1, n) :
4         if n % d == 0 :
5             somme = somme + d
6     if somme == n :
7         print(n, end=" ")
8 print()
```

6 28 496 8128

Les nombres parfaits inférieurs à 10 000 sont 6, 28, 496 et 8128. Ce script effectue environ 50 millions de tours de boucle intérieure (la somme des n pour $n \leq 10\,000$) ; il prend une à deux secondes.

3. Un diviseur de n autre que n est au plus égal à $n/2$. En déduire une amélioration du script.

Si d divise n et $d \neq n$, alors $n = dq$ avec $q \geq 2$, donc $d \leq n/2$. Il suffit de parcourir $\llbracket 1, \lfloor n/2 \rfloor \rrbracket$, ce qui divise le temps par deux.

```
1 for n in range(2, 10001) :
2     somme = 0
3     for d in range(1, n // 2 + 1) :
4         if n % d == 0 :
5             somme = somme + d
6     if somme == n :
7         print(n, end=" ")
8 print()
```

6 28 496 8128

Le chapitre Nombres premiers donnera une méthode bien plus rapide, en ne parcourant que les diviseurs $d \leq \sqrt{n}$ et en récupérant n/d en même temps.

Exercice 9 (★★★★☆) Nombres premiers

1. Soit $n \geq 2$. Écrire un script qui cherche le plus petit diviseur $d \geq 2$ de n à l'aide d'une boucle **while**, en s'arrêtant dès que $d^2 > n$ (on justifiera pourquoi c'est suffisant). Le script affiche si n est premier et, sinon, le diviseur trouvé. Tester avec $n = 221$.

Si n n'est pas premier, il s'écrit $n = ab$ avec $2 \leq a \leq b$, et alors $a^2 \leq ab = n$: le plus petit diviseur $d \geq 2$ de n vérifie $d^2 \leq n$. Par contraposition, si aucun entier d tel que $d^2 \leq n$ ne divise n , alors n est premier.

```
1 n = 221
2 d = 2
3 while d * d <= n and n % d != 0 :
4     d = d + 1
5 if d * d > n :
6     print(n, "est premier")
7 else :
8     print(n, "n'est pas premier, divisible par", d)
```

221 n'est pas premier, divisible par 13

La boucle s'arrête pour l'une des deux raisons : $d * d > n$ (aucun diviseur trouvé, n est premier) ou $n \% d == 0$ (diviseur trouvé). Le test qui suit distingue les deux cas.

221 = 13 × 17.

2. En imbriquant ce test dans une boucle **for**, afficher tous les nombres premiers inférieurs à 100 et les compter.

```
1 compteur = 0
2 for n in range(2, 101) :
3     d = 2
4     while d * d <= n and n % d != 0 :
5         d = d + 1
```

```

6         if d * d > n :
7             compteur = compteur + 1
8             print(n, end=" ")
9     print()
10    print(compteur)

```

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
25

Il y a 25 nombres premiers inférieurs à 100.

3. Combien y a-t-il de nombres premiers inférieurs à 10 000 ?

On retire l'affichage et l'on change la borne :

```

1    compteur = 0
2    for n in range(2, 10001) :
3        d = 2
4        while d * d <= n and n % d != 0 :
5            d = d + 1
6        if d * d > n :
7            compteur = compteur + 1
8    print(compteur)

```

1229

Il y a 1229 nombres premiers inférieurs à 10 000. Grâce à l'arrêt à $\sqrt{n}$, ce script est presque instantané, alors qu'une recherche jusqu'à n prendrait plusieurs secondes.

Exercice 10 (★★★★☆) Additionner des dixièmes

1. Écrire un script qui part de $x = 0$ et ajoute 0,1 dix fois de suite, puis affiche x et le booléen `x == 1`. Commenter.

```

1    x = 0
2    for k in range(10) :
3        x = x + 0.1
4    print(x, x == 1)

```

0.9999999999999999 False

Le nombre 0,1 n'a pas d'écriture finie en base 2 ; il est stocké avec une erreur d'arrondi, et dix additions accumulent ces erreurs. Le résultat diffère de 1 d'environ 10^{-16} , mais il en diffère : le test d'égalité échoue.

2. Écrire une boucle `while` qui ajoute 0,1 à x tant que $x < 1$ et compte les tours. Combien de tours obtient-on ? Pourquoi ?

```

1    x = 0
2    n = 0
3    while x < 1 :
4        x = x + 0.1
5        n = n + 1
6    print(n, x)

```

11 1.0999999999999999

Mathématiquement, dix tours suffisent puisque $10 \times 0,1 = 1$ n'est pas strictement infé-

rieur à 1. Mais après dix tours, x vaut $0,999\dots 9 < 1$: la boucle fait un onzième tour.

Onze tours, à cause de l'arrondi.

3. Que se passe-t-il si la condition est $x \neq 1$? Écrire cette boucle avec un garde-fou de 100 tours et observer.

```
1 x = 0
2 n = 0
3 while x != 1 and n < 100 :
4     x = x + 0.1
5     n = n + 1
6 print(n, x)
```

100 9.999999999999998

La valeur 1 n'est jamais atteinte exactement, donc sans garde-fou la boucle serait infinie ; avec lui, elle s'arrête après 100 tours, avec un x proche de 10 mais lui aussi entaché d'erreur. On ne teste jamais l'égalité de deux flottants.

4. Proposer deux corrections : l'une avec une tolérance, l'autre en ne manipulant que des entiers.

Avec une tolérance, on s'arrête dès que x est à moins de 10^{-9} de 1 :

```
1 x = 0
2 n = 0
3 while abs(x - 1) > 1e-9 :
4     x = x + 0.1
5     n = n + 1
6 print(n, x)
```

10 0.9999999999999999

En ne manipulant que des entiers, on compte des dixièmes : x est un entier, les calculs sont exacts, et l'on ne divise par 10 qu'à l'affichage.

```
1 x = 0
2 n = 0
3 while x < 10 :
4     x = x + 1
5     n = n + 1
6 print(n, x / 10)
```

10 1.0

Dix tours dans les deux cas. La seconde méthode est la meilleure lorsque c'est possible : rester chez les entiers, où tout est exact.

Exercice 11 (★★★☆☆) Les chiffres d'un entier

Pour un entier $n \geq 1$ écrit en base dix, $n \% 10$ est son chiffre des unités et $n // 10$ est l'entier obtenu en effaçant ce chiffre.

1. Écrire un script qui calcule la somme des chiffres de $n = 2025$ à l'aide d'une boucle **while**.

```
1 n = 2025
2 somme = 0
3 while n > 0 :
4     somme = somme + n % 10
```

```

5     n = n // 10
6     print(somme)

9

```

À chaque tour, on ajoute le chiffre des unités et on l'efface ; n prend les valeurs 2025,202,20,2,0 et la boucle s'arrête. $2 + 0 + 2 + 5 = 9$.

2. Écrire un script qui compte le nombre de chiffres de n .

```

1  n = 2025
2  nombre = 0
3  while n > 0 :
4      n = n // 10
5      nombre = nombre + 1
6  print(nombre)

4

```

Le nombre de chiffres est le nombre d'effacements nécessaires pour arriver à 0.

3. Vérifier par un script que, pour tout $n \in \llbracket 1, 10\,000 \rrbracket$, n est divisible par 9 si et seulement si la somme de ses chiffres l'est. On n'affichera qu'un booléen.

Attention à ne pas détruire n : la boucle **while** travaille sur une copie m . L'équivalence de deux propositions se teste avec `==` entre deux booléens.

```

1  tout = True
2  for n in range(1, 10001) :
3      m = n
4      somme = 0
5      while m > 0 :
6          somme = somme + m % 10
7          m = m // 10
8      tout = tout and ((n % 9 == 0) == (somme % 9 == 0))
9  print(tout)

True

```

Le critère de divisibilité par 9 est vérifié jusqu'à 10 000 ; sa démonstration, pour tout n , viendra avec le chapitre Congruences dans $\mathbb{Z}$: $10 \equiv 1 \pmod{9}$, donc n est congru à la somme de ses chiffres.

4. Écrire un script qui construit le *miroir* de n , c'est-à-dire l'entier obtenu en lisant ses chiffres de droite à gauche : le miroir de 12345 est 54321.

```

1  n = 12345
2  miroir = 0
3  while n > 0 :
4      miroir = 10 * miroir + n % 10
5      n = n // 10
6  print(miroir)

54321

```

L'accumulateur `miroir` est décalé d'un rang vers la gauche (multiplication par 10) avant d'accueillir le chiffre prélevé à droite de n : il prend les valeurs 5,54,543,5432,54321.

Exercice 12 (★★★★) Le problème des pièces

On dispose de pièces de 7 et de 5 unités, en quantité illimitée. Un montant $n \in \mathbb{N}$ est *atteignable* s'il existe des entiers naturels x et y tels que $n = 7x + 5y$.

1. Écrire un script qui, pour $n = 23$, parcourt tous les couples (x, y) possibles avec deux boucles imbriquées et affiche une décomposition si elle existe, ou indique que n n'est pas atteignable. On précisera les bornes des boucles.

Si $7x + 5y = n$ avec $x, y \geq 0$, alors $7x \leq n$ donc $x \leq \lfloor n/7 \rfloor$, et de même $y \leq \lfloor n/5 \rfloor$: les boucles sont finies. Un booléen `trouve` mémorise si une décomposition a été rencontrée.

```
1 n = 23
2 trouve = False
3 for x in range(0, n // 7 + 1) :
4     for y in range(0, n // 5 + 1) :
5         if 7 * x + 5 * y == n :
6             trouve = True
7             print(n, "=", 7, "x", x, "+", 5, "x", y)
8 if not trouve :
9     print(n, "n'est pas atteignable")

23 n'est pas atteignable
```

2. Afficher tous les montants non atteignables entre 0 et 40. Quel est le plus grand ?

```
1 for n in range(0, 41) :
2     trouve = False
3     for x in range(0, n // 7 + 1) :
4         for y in range(0, n // 5 + 1) :
5             if 7 * x + 5 * y == n :
6                 trouve = True
7     if not trouve :
8         print(n, end=" ")
9 print()

1 2 3 4 6 8 9 11 13 16 18 23
```

Le plus grand montant non atteignable est 23. Au-delà, les montants 24, 25, 26, 27, 28 sont atteignables ($24 = 2 \times 7 + 2 \times 5$, $25 = 5 \times 5$, $26 = 3 \times 7 + 5$, $27 = 7 + 4 \times 5$, $28 = 4 \times 7$), et tout montant supérieur s'en déduit en ajoutant des pièces de 5 : cinq montants consécutifs atteignables suffisent.

3. Reprendre la question précédente avec des pièces de a et de $b = a + 1$ unités pour a allant de 2 à 7, en cherchant le plus grand montant non atteignable parmi $\llbracket 0, ab - 1 \rrbracket$. Conjecturer une formule en fonction de a et b .

Quatre boucles imbriquées : sur a , sur le montant n , puis sur x et y . L'accumulateur `plus_grand` retient le dernier montant non atteignable rencontré, c'est-à-dire le plus grand puisque n croît. La dernière colonne affiche $ab - a - b$ pour comparaison.

```
1 for a in range(2, 8) :
2     b = a + 1
3     plus_grand = -1
4     for n in range(0, a * b) :
5         trouve = False
6         for x in range(0, n // a + 1) :
7             for y in range(0, n // b + 1) :
8                 if a * x + b * y == n :
9                     trouve = True
10    if not trouve :
```

```

11         plus_grand = n
12     print(a, b, plus_grand, a * b - a - b)

```

```

2 3 1 1
3 4 5 5
4 5 11 11
5 6 19 19
6 7 29 29
7 8 41 41

```

Conjecture : le plus grand montant non atteignable est $ab - a - b$. On retrouve $7 \times 5 - 7 - 5 = 23$ pour la question 2. Ce résultat, dû à Sylvester, est vrai dès que a et b sont premiers entre eux (ce qui est le cas de deux entiers consécutifs) ; lorsqu'ils ne le sont pas, les montants non multiples de leur PGCD ne sont jamais atteignables et il n'y a pas de plus grand. Le chapitre PGCD de deux entiers fournira les outils pour le démontrer.