

Exercices de synthèse

Variables, types, conditions et boucles

NSI – Première – Séquence 1

Cette feuille mélange **toutes** les notions de la séquence 1 : séquences d'instructions, variables, types et conversions, structures conditionnelles, boucles bornées et non bornées. Les exercices sont nettement plus exigeants que les précédents : la plupart demandent de combiner au moins trois notions à la fois.

La partie A se traite **sans ordinateur**, la partie B **sur machine**, la partie C regroupe des **défis** à traiter en autonomie.

Remarque : Deux opérateurs à connaître

Plusieurs exercices utilisent les deux opérateurs suivants sur les entiers :

- `a // b` : le **quotient entier** de la division euclidienne de `a` par `b` (`17 // 5` vaut `3`);
- `a % b` : le **reste** de cette division (`17 % 5` vaut `2`).

En particulier, `n % 2 == 0` teste si `n` est pair, et, pour un entier positif `n`, `n % 10` donne son chiffre des unités tandis que `n // 10` donne le nombre privé de son chiffre des unités.

Partie A – Analyser un programme (sans ordinateur)

1 Types en cascade

Exercice 1: Types en cascade

On exécute le programme suivant :

```

1  a = "7"
2  b = 3
3  c = int(a) + b
4  d = a * b
5  e = (c > 10) + (c == 10)
6  f = str(c) + a
7  g = float(b) / 2

```

1. Recopier et compléter le tableau suivant, en donnant pour chaque variable sa **valeur** (écrite exactement comme Python l'afficherait) et son **type**.

Variable	a	b	c	d	e	f	g
Valeur							
Type							

2. L'instruction `d = a * b` ne provoque aucune erreur, alors que `a + b` en provoquerait une. Expliquer précisément pourquoi.

3. La variable `e` contient un nombre alors qu'elle est construite uniquement à partir de comparaisons. Expliquer.
4. Les variables `c` et `f` ont-elles la même valeur ? Le même type ? Justifier.
5. L'instruction `g = float(b) / 2` donne `1.5`. Qu'obtiendrait-on avec `b / 2` ? Avec `b // 2` ? Que peut-on en conclure sur le type du résultat de l'opérateur `/` ?

Correction de l'exercice 1

Variable	a	b	c	d	e	f	g
1. Valeur	'7'	3	10	'777'	1	'107'	1.5
Type	str	int	int	str	int	str	float

Détail des calculs : `c = int("7") + 3 = 7 + 3 = 10` ; `d = "7" * 3 = "777"` ; `e = True + False`, car `10 > 10` est faux et `10 == 10` est vrai, d'où `e = 0 + 1 = 1` ; `f = "10" + "7" = "107"` ; `g = 3.0 / 2 = 1.5`.

2. L'opérateur `*` entre une chaîne et un entier a un sens : il **répète** la chaîne (ici 3 fois `"7"`). En revanche, l'opérateur `+` attend soit deux nombres (addition), soit deux chaînes (concaténation) : entre une `str` et un `int`, Python ne peut pas choisir, et lève une `TypeError`.
3. Une comparaison produit un **booléen**, et en Python `True` et `False` se comportent comme les entiers `1` et `0` dès qu'on les utilise dans un calcul. Ici `False + True` vaut donc `0 + 1 = 1`, un `int`.
4. Non : `c` vaut le nombre `10`, de type `int`, alors que `f` vaut la chaîne `'107'`, de type `str`. Ni la valeur, ni le type ne coïncident. `str(c)` a produit la chaîne `"10"`, qui a ensuite été concaténée avec `a`, c'est-à-dire `"7"`.
5. `b / 2` vaut également `1.5` : l'opérateur `/` renvoie **toujours** un `float`, même lorsque la division tombe juste (`4 / 2` vaut `2.0`, pas `2`). La conversion `float(b)` était donc inutile. En revanche `b // 2` vaut `1`, de type `int` : c'est le quotient entier.

2 Deux boucles imbriquées

Exercice 2: Deux boucles imbriquées

On exécute le programme suivant :

```

1 total = 0
2 for i in range(1, 4):
3     for j in range(i, 4):
4         if (i + j) % 2 == 0:
5             total = total + i * j
6         else:
7             total = total - 1
8     print(i, total)
9 print("Fin", total)

```

1. Combien de fois, au total, le test `(i + j) % 2 == 0` est-il évalué ? Justifier sans exécuter le programme.
2. Recopier et compléter le tableau ci-dessous, avec une ligne par tour de la boucle intérieure.

i	j	(i+j) pair ?	total après ce tour
1	1		
...			

3. En déduire l’affichage complet du programme, ligne par ligne.
4. L’instruction `print(i, total)` est-elle dans la boucle intérieure, dans la boucle extérieure, ou en dehors des deux ? Qu’afficherait le programme si on l’indentait d’un niveau supplémentaire ?

Correction de l’exercice 2

1. La boucle extérieure prend les valeurs `i = 1, 2, 3`. Pour chaque `i`, la boucle intérieure parcourt `range(i, 4)`, soit $4 - i$ valeurs : 3 tours pour `i = 1`, 2 tours pour `i = 2`, 1 tour pour `i = 3`. Le test est donc évalué $3 + 2 + 1 = 6$ fois.

i	j	(i+j) pair ?	total après ce tour
1	1	oui (2)	$0 + 1 = 1$
1	2	non (3)	$1 - 1 = 0$
2.	1	oui (4)	$0 + 3 = 3$
	2	oui (4)	$3 + 4 = 7$
	2	non (5)	$7 - 1 = 6$
	3	oui (6)	$6 + 9 = 15$

3. Le programme affiche :

```

1  1 3
2  2 6
3  3 15
4  Fin 15

```

4. Elle est indentée au même niveau que la boucle intérieure : elle appartient donc au corps de la boucle **extérieure**, et s’exécute une fois par valeur de `i`, après que la boucle intérieure a terminé tous ses tours. Si on l’indentait d’un niveau de plus, elle entrerait dans la boucle intérieure et serait exécutée à chaque tour, soit 6 lignes affichées au lieu de 3 :

```

1  1 1
2  1 0
3  1 3
4  2 7
5  2 6
6  3 15
7  Fin 15

```

3 Une boucle while à deux variables

Exercice 3: Une boucle while à deux variables

On exécute le programme suivant :

```

1  a = 42
2  b = 30
3  while a != b:
4      if a > b:
5          a = a - b
6      else:
7          b = b - a
8  print(a)

```

1. Tracer l'exécution : recopier et compléter un tableau donnant les valeurs de **a** et **b** après chaque tour de boucle. Combien de tours la boucle effectue-t-elle ? Qu'affiche le programme ?
2. Reprendre la question précédente avec **a = 18** et **b = 12**, puis avec **a = 20** et **b = 7**.
3. À partir de ces trois exemples, formuler une conjecture : que calcule ce programme ?
4. Que se passe-t-il si l'on exécute ce programme avec **a = 5** et **b = 5** ? Et avec **a = 5** et **b = 0** ? Le programme se termine-t-il toujours ? Justifier.
5. Proposer une modification du programme pour qu'il affiche aussi le **nombre de tours** effectués.

Indication

Pour la question 4, on pourra remarquer que la quantité **a + b** diminue strictement à chaque tour **tant que** les deux variables restent strictement positives.

Correction de l'exercice 3

	Tour	Test $a \neq b$	a	b
	(initial)		42	30
	1	vrai, $a > b$	12	30
1.	2	vrai, $a < b$	12	18
	3	vrai, $a < b$	12	6
	4	vrai, $a > b$	6	6
	–	faux : sortie	6	6

La boucle effectue 4 tours et le programme affiche **6**.

2. Avec $a = 18, b = 12$: (6, 12), puis (6, 6) – 2 tours, affichage **6**.

Avec $a = 20, b = 7$: (13, 7), (6, 7), (6, 1), (5, 1), (4, 1), (3, 1), (2, 1), (1, 1) – 8 tours, affichage **1**.

3. On obtient $\text{pgcd}(42, 30) = 6$, $\text{pgcd}(18, 12) = 6$ et $\text{pgcd}(20, 7) = 1$: le programme calcule le **PGCD** de **a** et **b**. C'est l'algorithme d'Euclide par soustractions successives.

4. Avec $a = b = 5$, la condition $a \neq b$ est fausse immédiatement : la boucle n'est jamais exécutée (0 tour) et le programme affiche **5**, ce qui est correct puisque $\text{pgcd}(5, 5) = 5$.
Avec $a = 5$ et $b = 0$: la condition est vraie, et comme $a > b$ on exécute $a = a - 0$, qui laisse **a** inchangé. Aucune des deux variables ne change plus jamais : c'est une **boucle infinie**. Le programme ne se termine donc pas toujours : il faut que **a** et **b** soient tous deux strictement positifs. Dans ce cas, à chaque tour, on retranche à la plus grande des deux variables une quantité strictement positive : la somme $a + b$ est un entier positif qui décroît strictement, elle ne peut donc pas décroître indéfiniment et la boucle finit par s'arrêter.

5. Il suffit d'ajouter un compteur, initialisé avant la boucle et incrémenté à chaque tour :

```

1  a = 42
2  b = 30
3  tours = 0
4  while a != b:
5      if a > b:
6          a = a - b
7      else:
8          b = b - a
9      tours = tours + 1
10 print(a, "obtenu en", tours, "tours")

```

Attention à placer $\text{tours} = \text{tours} + 1$ dans le corps de la boucle, mais **en dehors** du **if / else**, pour qu'il compte tous les tours.

4 Cinq erreurs à débuser

Exercice 4: Cinq erreurs à débuser

Un élève souhaite écrire un programme qui demande un entier `n` à l'utilisateur, puis affiche combien il y a de multiples de 3 parmi les entiers de 1 à `n` (bornes comprises). Il propose ceci :

```
1 n = input("Jusqu'à quel nombre ? ")
2 compteur = 0
3 i = 1
4 while i < n:
5     if i % 3 = 0:
6         compteur = compteur + 1
7 i = i + 1
8 print("Il y a " + compteur + " multiples de 3.")
```

Ce programme contient **cinq** erreurs.

1. Les repérer toutes, en indiquant pour chacune le numéro de ligne concerné.
2. Classer chaque erreur dans l'une des trois catégories suivantes :
 - **erreur de syntaxe** : le programme refuse de démarrer ;
 - **erreur d'exécution** : le programme démarre puis s'interrompt sur un message d'erreur ;
 - **erreur de logique** : le programme s'exécute sans message d'erreur, mais ne fait pas ce qui était attendu.
3. Réécrire le programme corrigé.
4. Vérifier mentalement le programme corrigé avec `n = 10` : quelle valeur doit-il afficher ?

Correction de l'exercice 4

1. Les cinq erreurs :
 - (a) ligne 1 : `input()` renvoie toujours une `str`. Il manque la conversion `int(...)` ;
 - (b) ligne 4 : la condition `i < n` exclut la valeur `n`, alors que l'énoncé demande les multiples de 3 **jusqu'à n compris** : il faut `i <= n` ;
 - (c) ligne 5 : `=` est l'opérateur d'affectation ; pour tester une égalité il faut `==` ;
 - (d) ligne 7 : `i = i + 1` n'est pas indentée, elle se trouve donc **après** la boucle et non dedans. La variable `i` ne change jamais pendant la boucle : boucle infinie ;
 - (e) ligne 8 : on tente de concaténer une chaîne et un entier (`compteur`). Il faut soit une conversion `str(compteur)`, soit – plus simple – des virgules dans le `print`.
2. Classement :
 - (c) est une **erreur de syntaxe** : Python refuse de lancer le fichier (`SyntaxError`) ;
 - (a) et (e) sont des **erreurs d'exécution** : comparaison `int < str` pour (a), concaténation `str + int` pour (e), qui lèvent toutes deux une `TypeError` ;
 - (b) et (d) sont des **erreurs de logique** : le programme est valide pour Python, mais son comportement n'est pas celui attendu (résultat faux d'une unité dans un cas, boucle infinie dans l'autre).

Remarque : tant que (c) n'est pas corrigée, le programme ne démarre même pas – les autres erreurs ne se manifestent que les unes après les autres, ce qui est une situation de débogage très courante.

3. Programme corrigé :

```
1  n = int(input("Jusqu'à quel nombre ? "))
2  compteur = 0
3  i = 1
4  while i <= n:
5      if i % 3 == 0:
6          compteur = compteur + 1
7      i = i + 1
8  print("Il y a", compteur, "multiples de 3.")
```

4. Avec `n = 10`, les multiples de 3 concernés sont 3, 6 et 9 : le programme doit afficher `Il y a 3 multiples de 3.`

Partie B – Écrire un programme (sur machine)

5 Racine numérique

Exercice 5: Racine numérique

La **racine numérique** d'un entier positif s'obtient en additionnant ses chiffres, puis en recommençant sur le résultat, jusqu'à n'obtenir qu'un seul chiffre. Par exemple, pour 9875 :

$$9875 \longrightarrow 9 + 8 + 7 + 5 = 29 \longrightarrow 2 + 9 = 11 \longrightarrow 1 + 1 = 2.$$

La racine numérique de 9875 est donc 2, obtenue en 3 étapes.

Dans un fichier `racine_numerique.py` :

1. Écrire un programme qui demande un entier positif `n` à l'utilisateur et affiche la **somme de ses chiffres**. On utilisera une boucle `while` ainsi que les opérateurs `%` et `//`, sans convertir `n` en chaîne de caractères.
2. Compléter ce programme pour qu'il calcule la **racine numérique** de `n`, en répétant l'opération précédente tant que le nombre obtenu comporte plus d'un chiffre. Le programme affichera la racine numérique **et** le nombre d'étapes effectuées.
3. Faire afficher au passage chaque étape intermédiaire, sous la forme `9875 -> 29`.
4. Tester le programme sur `9875`, `1`, `99999999` et `10`. Que remarque-t-on pour `1` et pour `10` ?
5. **Pour aller plus loin.** Écrire une seconde version de la question 1 qui, cette fois, convertit `n` en chaîne de caractères et parcourt ses caractères avec une boucle `for`. Laquelle des deux versions vous paraît la plus lisible ?

Indication

Question 1 : à chaque tour, `n % 10` donne le chiffre des unités, et `n // 10` donne le nombre débarrassé de ce chiffre. On s'arrête quand il ne reste plus rien, c'est-à-dire quand `n` vaut `0`. Attention : si l'on modifie `n` dans cette boucle, sa valeur initiale est perdue – travailler sur une copie si l'on en a encore besoin ensuite.

Correction de l'exercice 5

Questions 1 à 3 (programme complet) :

```

1  n = int(input("Entrez un entier positif : "))
2  etapes = 0
3
4  while n >= 10:                # tant qu'il reste plus d'un chiffre
5      reste = n                 # copie de travail
6      somme = 0
7      while reste > 0:          # somme des chiffres de reste
8          somme = somme + reste % 10
9          reste = reste // 10
10     print(n, "->", somme)
11     n = somme
12     etapes = etapes + 1
13
14     print("Racine numérique :", n)
15     print("Obtenue en", etapes, "étape(s)")

```

Points clés : la boucle **intérieure** calcule la somme des chiffres d'un nombre, la boucle **extérieure** recommence tant que le résultat comporte au moins deux chiffres (`n >= 10`). La variable `reste` est une copie de `n` : sans elle, `n` serait détruite par la boucle intérieure avant d'avoir pu être affichée.

4. Pour `9875`, l'affichage est :

```

1  9875 -> 29
2  29 -> 11
3  11 -> 2
4  Racine numérique : 2
5  Obtenue en 3 étape(s)

```

Pour `1`, la condition `n >= 10` est fausse dès le départ : la boucle extérieure n'est jamais exécutée, la racine vaut `1` en `0` étape. C'est le comportement correct : un nombre à un seul chiffre est déjà sa propre racine numérique. Pour `10`, on obtient `1` en une étape. `99999999` donne `72`, puis `9` : racine `9` en 2 étapes.

5. Version avec conversion en chaîne :

```

1  n = int(input("Entrez un entier positif : "))
2  somme = 0
3  for chiffre in str(n):
4      somme = somme + int(chiffre)
5  print("Somme des chiffres :", somme)

```

Cette version est plus courte et plus lisible, mais elle repose sur une conversion vers l'écriture décimale du nombre, alors que la première n'utilise que des opérations arithmétiques. Les deux approches sont correctes : la seconde est souvent préférée en pratique, la première est celle qui se généralise à d'autres bases que 10.

6 La conjecture de Syracuse

Exercice 6: La conjecture de Syracuse

On part d'un entier $u_0 \geq 1$ et on construit une suite de nombres selon la règle suivante :

- si le terme courant est **pair**, le suivant est sa moitié ;
- s'il est **impair**, le suivant vaut son triple augmenté de 1.

Par exemple, en partant de 6 : 6, 3, 10, 5, 16, 8, 4, 2, 1. La **conjecture de Syracuse** affirme que, quel que soit le point de départ, la suite finit toujours par atteindre 1 – personne n'a su le démontrer à ce jour.

Dans un fichier `syracuse.py` :

1. Écrire un programme qui demande u_0 et affiche tous les termes de la suite jusqu'à atteindre 1 (compris).
2. Faire afficher le **temps de vol** : le nombre d'étapes nécessaires pour atteindre 1.
3. Faire afficher l'**altitude maximale** : la plus grande valeur atteinte par la suite.
4. Faire afficher le **temps de vol en altitude** : le nombre d'étapes effectuées avant que la suite ne devienne, pour la première fois, strictement inférieure à u_0 .
5. Tester avec $u_0 = 15$, puis $u_0 = 27$. Commenter.
6. Pourquoi une boucle `while` est-elle ici indispensable ? Une boucle `for` conviendrait-elle ?

Indication

Question 3 : tenir à jour une variable `altitude_max`, initialisée à u_0 , et la mettre à jour à chaque nouveau terme s'il est plus grand.

Question 4 : le mot important est « pour la première fois ». Il faut donc un moyen de ne retenir que le premier passage sous u_0 – par exemple à l'aide d'un booléen indiquant si ce passage a déjà eu lieu.

Correction de l'exercice 6

```

1  u0 = int(input("Valeur de départ : "))
2
3  u = u0
4  temps_de_vol = 0
5  altitude_max = u0
6  temps_en_altitude = 0
7  deja_descendu = False
8
9  print(u)
10 while u != 1:
11     if u % 2 == 0:
12         u = u // 2
13     else:
14         u = 3 * u + 1
15     print(u)
16
17     temps_de_vol = temps_de_vol + 1
18     if u > altitude_max:
19         altitude_max = u
20     if u < u0 and not deja_descendu:
21         temps_en_altitude = temps_de_vol
22         deja_descendu = True
23
24 print("Temps de vol :", temps_de_vol)
25 print("Altitude maximale :", altitude_max)
26 print("Temps de vol en altitude :", temps_en_altitude)

```

5. Pour $u_0 = 15$: temps de vol 17, altitude maximale 160, temps de vol en altitude 11.
 Pour $u_0 = 27$: temps de vol 111, altitude maximale 9232, temps de vol en altitude 96.
 Deux valeurs de départ très proches donnent des comportements radicalement différents : la suite issue de 27 grimpe jusqu'à 9232, soit plus de 340 fois son point de départ, avant de retomber. Rien, dans l'écriture du programme, ne laissait prévoir un tel écart.
6. Le nombre d'étapes n'est pas connu à l'avance : il dépend de u_0 d'une manière qu'on ne sait pas prévoir (c'est précisément l'objet de la conjecture). Une boucle `for` exigerait de fixer à l'avance un nombre de tours, ce qui est impossible ici. Une boucle `while`, dont la condition d'arrêt est `u != 1`, est donc indispensable. On notera d'ailleurs que **rien ne garantit** que ce programme se termine pour toute valeur de départ : sa terminaison est exactement le contenu de la conjecture, non démontrée.

7 Anatomie d'un mot

Exercice 7: Anatomie d'un mot

Dans un fichier `mot.py`, écrire un programme qui demande un mot à l'utilisateur (en minuscules, sans accent) et affiche successivement :

1. le nombre de **voyelles** et le nombre de **consonnes** qu'il contient ;
2. le mot **à l'envers**, sans utiliser aucune fonction toute faite ;
3. si le mot est un **palindrome** (il se lit de la même façon dans les deux sens) ;
4. si le mot contient **deux lettres identiques consécutives** (comme le double `n` de

"bonne"), et, le cas échéant, laquelle est la première ;

5. la plus longue **suite de voyelles consécutives** qu'il contient (par exemple 3 pour "oiseau", qui contient eau).

Tester sur radar , bonne , oiseau , rythme et a .

Indication

Question 2 : construire une chaîne **envers** initialement vide, puis, pour chaque lettre du mot, placer cette lettre **devant** ce qui a déjà été construit.

Questions 4 et 5 : il faut se souvenir de la lettre précédente. On utilisera une variable **precedente** , initialisée à la chaîne vide et mise à jour à la fin de chaque tour de boucle.

Correction de l'exercice 7

```

1  mot = input("Entrez un mot : ")
2
3  voyelles = 0
4  consonnes = 0
5  envers = ""
6  double = ""
7  suite = 0
8  suite_max = 0
9  precedente = ""
10
11 for lettre in mot:
12     # 1. voyelles et consonnes
13     if lettre in "aeiouy":
14         voyelles = voyelles + 1
15         suite = suite + 1
16     else:
17         consonnes = consonnes + 1
18         suite = 0
19     if suite > suite_max:
20         suite_max = suite
21
22     # 2. mot à l'envers : la nouvelle lettre passe devant
23     envers = lettre + envers
24
25     # 4. première répétition de deux lettres identiques
26     if lettre == precedente and double == "":
27         double = lettre
28
29     precedente = lettre
30
31 print("Voyelles :", voyelles, "- Consonnes :", consonnes)
32 print("À l'envers :", envers)
33
34 if mot == envers:
35     print("C'est un palindrome.")
36 else:
37     print("Ce n'est pas un palindrome.")
38
39 if double == "":
40     print("Aucune lettre doublée.")
41 else:
42     print("Première lettre doublée :", double)
43
44 print("Plus longue suite de voyelles :", suite_max)

```

Commentaires :

- question 2 : écrire `envers = envers + lettre` reconstruirait le mot à l'endroit. C'est bien l'ordre de la concaténation qui fait tout le travail ;
- question 3 : une fois `envers` construite, le test du palindrome tient en une comparaison de chaînes ;
- question 5 : le compteur `suite` est remis à 0 dès qu'une consonne est rencontrée, et `suite_max` conserve la plus grande valeur atteinte. C'est le même schéma « compteur courant / record » que l'altitude maximale de l'exercice précédent.

Résultats attendus : `radar` est un palindrome (2 voyelles, 3 consonnes, suite maximale 1) ;

bonne a un **n** doublé; **oiseau** a une suite de 3 voyelles; **rythme** compte 2 voyelles et 4 consonnes, le **y** étant ici rangé parmi les voyelles; **a** est un palindrome d'une seule lettre.

Partie C – Défis

8 Défi 1 : la pyramide

Exercice 8: La pyramide

Écrire un programme qui demande un entier **n** compris entre 1 et 9, et affiche une pyramide de chiffres de hauteur **n**. Pour **n = 5**, on doit obtenir exactement :

```

1      1
2     121
3    12321
4   1234321
5  123454321

```

1. Avant d'écrire la moindre ligne de code, décrire précisément la ligne numéro **i** : combien d'espaces au début ? quels chiffres ensuite, et dans quel ordre ? combien de caractères en tout ?
2. Écrire le programme. On construira chaque ligne dans une variable de type **str** avant de l'afficher.
3. Pourquoi l'énoncé impose-t-il $n \leq 9$? Que se passerait-il pour **n = 12** ?

Indication

La partie descendante de la ligne peut s'obtenir avec `range(i - 1, 0, -1)`, qui engendre les entiers de **i-1** jusqu'à **1** en décroissant.

Correction de l'exercice 8

1. La ligne **i** (pour **i** allant de 1 à **n**) comporte : $n - i$ espaces, puis les chiffres $1, 2, \dots, i$ en montant, puis les chiffres $i - 1, i - 2, \dots, 1$ en descendant. Elle contient donc $2i - 1$ chiffres, et $n - i + 2i - 1 = n + i - 1$ caractères en tout.

```

12. n = int(input("Hauteur (entre 1 et 9) : "))
13.
14. for i in range(1, n + 1):
15.     ligne = " " * (n - i)           # les espaces du début
16.     for j in range(1, i + 1):       # partie montante
17.         ligne = ligne + str(j)
18.     for j in range(i - 1, 0, -1):   # partie descendante
19.         ligne = ligne + str(j)
20.     print(ligne)

```

On remarquera l'emploi de `" " * (n - i)` : la répétition d'une chaîne évite une troisième boucle.

- Au-delà de 9, les nombres s'écrivent avec deux chiffres : chacun occupe alors deux caractères au lieu d'un, et la pyramide perd sa symétrie visuelle (la ligne 12 commencerait par `123456789101112` : elle occuperait 15 colonnes rien que pour sa partie montante, au lieu des 12 attendues). Le calcul de largeur de la question 1 n'est valable que tant qu'un nombre occupe exactement une colonne.

9 Défi 2 : l'ordinateur devine votre nombre

Exercice 9: L'ordinateur devine votre nombre

On inverse les rôles du jeu du « plus petit / plus grand » : c'est **vous** qui choisissez mentalement un nombre entier entre 1 et 100, et c'est le **programme** qui doit le deviner. À chaque tour, le programme propose un nombre, et vous répondez en tapant :

- `+` si le nombre cherché est plus grand que la proposition ;
- `-` s'il est plus petit ;
- `=` si c'est le bon.

- Écrire un programme qui procède ainsi, en proposant à chaque tour le **milieu** de l'intervalle des valeurs encore possibles. Le programme s'arrête dès que vous répondez `=`, et affiche le nombre de propositions qu'il lui a fallu.
- Tester le programme en pensant à 1, puis à 100, puis à 37. Quel est le nombre **maximal** de propositions nécessaires ? Comparer à un programme qui essaierait `1`, puis `2`, puis `3`, etc.
- Que se passe-t-il si vous trichez, c'est-à-dire si vos réponses sont incohérentes ? Modifier le programme pour qu'il détecte cette situation et affiche « Vous avez triché ! ».

Indication

Le programme doit garder en mémoire deux bornes, `mini` et `maxi`, entre lesquelles le nombre cherché se trouve encore. Une réponse `+` signifie que la proposition est trop petite : toutes les valeurs jusqu'à la proposition comprise sont éliminées.

Correction de l'exercice 9

```
1 mini = 1
2 maxi = 100
3 reponse = ""
4 propositions = 0
5 triche = False
6
7 while reponse != "=" and not triche:
8     if mini > maxi:
9         triche = True
10    else:
11        proposition = (mini + maxi) // 2
12        propositions = propositions + 1
13        print("Est-ce", proposition, "?")
14        reponse = input("Répondez +, - ou = : ")
15
16        if reponse == "+":
17            mini = proposition + 1
18        elif reponse == "-":
19            maxi = proposition - 1
20
21 if triche:
22     print("Vous avez triché !")
23 else:
24     print("Trouvé en", propositions, "proposition(s).")
```

2. Chaque proposition élimine environ la moitié des valeurs encore possibles : de 100 candidats on passe à 50, puis 25, 12, 6, 3, 1. Il faut donc au plus **7** propositions, quel que soit le nombre choisi. Un programme qui essaierait 1, 2, 3, ... en demanderait jusqu'à 100. C'est le principe de la **recherche dichotomique**, que vous retrouverez plus tard dans l'année.
3. Si les réponses sont incohérentes, l'intervalle des valeurs possibles finit par se vider : on obtient `mini > maxi`, situation impossible si le joueur est honnête. Le test placé en début de boucle détecte exactement cela. Sans ce test, `(mini + maxi) // 2` continuerait de produire des propositions absurdes, situées hors de l'intervalle de départ.

10 Défi 3 : le programme mystère

Exercice 10: Le programme mystère

On considère le programme suivant :

```

1  a = 13
2  b = 11
3  resultat = 0
4  while a > 0:
5      if a % 2 == 1:
6          resultat = resultat + b
7          a = a // 2
8          b = b * 2
9  print(resultat)

```

1. Tracer l'exécution complète dans un tableau à quatre colonnes : **a**, **b**, **resultat**, et le test **a % 2 == 1**. Qu'affiche le programme ?
2. Refaire la trace avec **a = 6** et **b = 5**, puis avec **a = 4** et **b = 7**. Que constate-t-on ?
3. Formuler une conjecture : que calcule ce programme ?
4. Ce programme n'utilise jamais l'opérateur ***** entre **a** et **b**. Expliquer, en observant la colonne **b** de votre trace, comment il parvient malgré tout à ce résultat.
5. Combien de tours de boucle sont effectués pour **a = 13** ? Pour **a = 1000** ? Comparer avec un programme qui calculerait le même résultat en ajoutant **b** à lui-même **a** fois.

Correction de l'exercice 10

	Tour	a (début de tour)	b (début de tour)	a impair ?	resultat
1.	1	13	11	oui	$0 + 11 = 11$
	2	6	22	non	11
	3	3	44	oui	$11 + 44 = 55$
	4	1	88	oui	$55 + 88 = 143$
	–	0	176	sortie de boucle	

Le programme affiche **143**.

2. Avec $a = 6$ et $b = 5$, on obtient **30**. Avec $a = 4$ et $b = 7$, on obtient **28**. Or $13 \times 11 = 143$, $6 \times 5 = 30$ et $4 \times 7 = 28$.
3. Le programme calcule le **produit** $a \times b$. C'est la « multiplication du paysan russe », ou multiplication égyptienne.
4. La colonne **b** contient les valeurs 11, 22, 44, 88, c'est-à-dire 11×1 , 11×2 , 11×4 , 11×8 : les doublements successifs fabriquent les produits de b par les puissances de 2. Parallèlement, les divisions entières de **a** par 2 examinent un à un les chiffres de l'écriture **binaire** de a : le test **a % 2 == 1** demande si le chiffre courant vaut 1. Ici $13 = 8 + 4 + 1$, et le programme additionne précisément 11×8 , 11×4 et 11×1 , soit 11×13 . Seuls des doublements et des divisions par 2 sont nécessaires – des opérations très simples à réaliser, y compris pour une machine.
5. La variable **a** est divisée par 2 à chaque tour : le nombre de tours est le nombre de chiffres de l'écriture binaire de **a**, soit 4 tours pour $a = 13$ et 10 tours pour $a = 1000$. Un programme additionnant **b** à lui-même **a** fois effectuerait 13 tours dans le premier cas et 1000 dans le second. L'écart se creuse spectaculairement à

mesure que **a** grandit : c'est une première rencontre avec la question du **coût** d'un algorithme, qui sera étudiée en terminale.