Structures itératives

Doc. 2,4

Objectif

En manipulant les boucles `for` et `while` dans la console et dans Thonny, vous allez apprendre à répéter des instructions un nombre de fois connu ou inconnu à l'avance, et à choisir la structure itérative adaptée à chaque situation.

§1. Combien de fois une instruction doit-elle être répétée ?

Un programme est souvent amené à répéter plusieurs fois les mêmes instructions : afficher les nombres de 1 à 1000, demander un mot de passe jusqu'à ce qu'il soit correct, ou additionner les notes d'une classe entière. Écrire autant de fois l'instruction à répéter serait absurde ; les langages de programmation proposent pour cela des **structures itératives**, plus couramment appelées **boucles**. Chaque passage dans une boucle est appelé une **itération**.

Q1. Pour chacune des situations suivantes, indiquer si le nombre de répétitions est connu **avant** de commencer à répéter, ou s'il n'est découvert qu'**au fur et à mesure** de l'exécution :

- afficher les nombres de 1 à 100 ;
- demander un mot de passe jusqu'à ce qu'il soit correct ;
- additionner les 12 notes d'un bulletin ;
- lancer un dé jusqu'à l'obtention d'un 6.

À retenir

- Une **boucle déterministe** (ou boucle **Pour**) répète un nombre d'itérations **connu à l'avance** ; elle utilise une variable, le **compteur de boucle**, pour compter les itérations effectuées.
- Une **boucle non déterministe** (ou boucle **conditionnelle**) répète des itérations **tant qu'une condition d'arrêt n'est pas vérifiée** ; ce nombre n'est pas connu à l'avance.

Python ne fournit qu'une seule boucle non déterministe, la boucle `while` (**Tant que**). Une boucle non déterministe peut toujours remplacer une boucle déterministe ; l'inverse n'est pas vrai.

§2. La boucle « Tant que » : l'instruction `while`

§2.1. Syntaxe et fonctionnement

La syntaxe d'une boucle `while` est :

```
while <condition>:
    instructions à répéter
```
Python

La **condition** est une expression booléenne, évaluée **au début de chaque itération**. Tant qu'elle vaut `True`, le bloc indenté est exécuté, puis la condition est évaluée à nouveau. Dès qu'elle vaut `False`, la boucle s'arrête et l'exécution reprend après le bloc. Il est même possible que le bloc ne soit exécuté **aucune fois**, si la condition est déjà fausse dès la première évaluation.

Q2. Recopier la fonction suivante dans la partie principale de l'éditeur Thonny :

```python
def affiche_jusqua(n: int) -> None:
    """
    Affiche les entiers de 1 à n, un par ligne, à l'aide d'une boucle Tant que.
    """
    i = 1            # (1)
    while i <= n:    # (2)
        print(i)     # (3)
        i += 1       # (4)
```

Remplacer chacun des numéros ci-dessus par une phrase qui décrit ce que l'instruction réalise.

Q3. Appeler ensuite, dans la console, la fonction : `affiche_jusqua(5)` puis `affiche_jusqua(0)`.
Combien de fois le bloc indenté est-il exécuté dans le premier appel ? Dans le second appel ?

Q4. Quelle instruction fait progresser `i` vers la condition d'arrêt ?

⚠ Boucle infinie

Si aucune instruction du bloc ne fait progresser la condition vers sa fin (si, par exemple, `i += 1` avait été oublié), la condition `i <= n` resterait indéfiniment vraie : la boucle s'exécuterait sans jamais s'arrêter. Ce défaut porte un nom : la **boucle infinie**.

✓ À retenir

Écrire une boucle `while` suppose de réunir trois éléments :

1. une **initialisation** de la ou des variables testées par la condition ;
2. une **condition d'arrêt**, expression booléenne évaluée avant chaque itération ;
3. une instruction qui **fait progresser** cette condition à chaque itération, jusqu'à ce qu'elle devienne fausse.

Oublier ce troisième élément est la cause la plus fréquente de **boucle infinie**.

§2.2. Faire tourner une boucle à la main

Vérifier qu'une boucle fait ce qu'on attend d'elle exige de savoir la « faire tourner à la main » : appliquer méthodiquement les instructions, itération après itération, en notant dans un tableau l'évolution du contenu de chaque variable.

Q5. On donne la fonction suivante, dont il faut déterminer l'action :

```python
def inconnue(a: int, b: int) -> int:
    """ Fonction dont il faut déterminer l'action. """
    r = a % b
    while r != 0:
        a = b
        b = r
        r = a % b
    return b
```

1. Repérer les instructions répétées par la boucle et l'instruction qui teste la fin de la boucle.
2. Quelle instruction modifie le résultat de l'évaluation de ce test ?
3. Sans utiliser Python, compléter le tableau d'évolution ci-dessous pour l'appel `inconnue(30, 42)`, puis vérifier en exécutant le programme.

Étape	a	b	r
avant la boucle	30	42	
itération 1			
itération 2			
itération 3			

4. Visualiser le fonctionnement de cette fonction à l'aide du débogueur intégré à Thonny.
5. Reprendre la question précédente pour l'appel `inconnue(35, 6)`, sans reconstruire de tableau : donner uniquement la valeur retournée.
6. Quel calcul réalise cette fonction ?

Q6. Voici une autre fonction, dont il faut également déterminer l'action :

```python
def inconnue_deux(a: int) -> int:
    """ Fonction dont il faut déterminer l'action. """
    b = 1
    i = 2
    while i <= a:
        b *= i
        i += 1
    return b
```

1. Quelle est la valeur initiale de `i` ? Quelle est sa valeur lorsque la boucle s'arrête ?
2. Sans utiliser Python, compléter le tableau d'évolution ci-dessous pour l'appel `inconnue_deux(6)`, puis vérifier en exécutant le programme.

Étape	i	b
avant la boucle	2	1
itération 1		
itération 2		
itération 3		
itération 4		
itération 5		

3. Quel calcul réalise cette fonction ?

§3. La boucle « Pour chaque » : l'instruction for et l'itérateur range

§3.1. Syntaxe de la boucle « Pour chaque »

En Python, l'instruction `for` implémente en réalité une boucle **Pour chaque** : elle parcourt, un par un, les éléments d'un objet **itérable**, et non un simple compteur comme dans d'autres langages.

```python
for element in itérable:
    instructions à répéter
```

À chaque itération, la variable `element` prend la valeur de l'élément courant de l'itérable, pour être utilisée dans le bloc indenté. **Cette variable ne doit jamais être modifiée dans le corps de la boucle** : dans le cas contraire, le comportement de la boucle devient imprévisible.

Q7. Recopier la fonction suivante dans la partie principale de l'éditeur :

```python
def affiche_intervalle(n: int) -> None:
    """Affiche, séparés par un espace, les entiers produits par
    range(1, n + 1).
    """
    for nbre in range(1, n + 1):
        print(nbre, end=' ')
```

Appeler `affiche_intervalle(10)` dans la console. Quels entiers sont produits par `range(1, 11)` ? Combien y en a-t-il ?

Q8. Recopier la fonction suivante dans la partie principale de l'éditeur :

```python
def table_de(n: int) -> str:
    """ Retourne la table de multiplication de n, de 1 à 10. """
    reponse = ""
    for nbre in range(1, 11):
        calcul = n * nbre
        reponse += str(calcul) + " "
    return reponse
```

Cette fonction parcourt le même itérable que la précédente. Sans exécuter le programme, prévoir le résultat de `table_de(4)`, puis vérifier dans la console.

Réécrire ensuite l'instruction `reponse += str(calcul) + " "` à l'aide d'une f-string (voir Doc. 2,2), et vérifier que `table_de(4)` retourne toujours le même résultat.

§3.2. L'itérateur range

L'itérateur range

L'itérateur `range` génère, à la volée, des entiers compris entre une valeur de départ `début` et une valeur finale `fin` (exclue), par pas de `pas`. Trois écritures sont possibles :

`range(fin)`	entiers de 0 à `fin` (exclu), par pas de 1
`range(début, fin)`	entiers de `début` à `fin` (exclu), par pas de 1
`range(début, fin, pas)`	entiers de `début` à `fin` (exclu), par pas de `pas`

Comme pour les tranches, la valeur `fin` n'est **jamais** atteinte : Python considère systématiquement les intervalles sous la forme [début, fin[.

Q9. Recopier la fonction suivante dans la partie principale de l'éditeur :

```python
def demo_range(debut: int, fin: int, pas: int) -> None:
    """Affiche, séparés par un espace, les entiers produits par
    range(debut, fin, pas).
    """
```

```
    for i in range(debut, fin, pas):
        print(i, end=' ')
```

Prévoir puis vérifier dans la console le résultat de chacun des appels suivants :

```
>>> demo_range(1, 10, 1)
>>> demo_range(1, 10, 3)
>>> demo_range(10, 1, -1)
>>> demo_range(10, -15, -3)
```
Python

Le pas `pas` peut-il être négatif ? Que se passe-t-il alors ?

Q10. Écrire, dans la partie principale de l'éditeur, deux fonctions `demo_range_1arg` et `demo_range_2arg`, qui affichent respectivement, séparés par un espace, les entiers produits par `range(fin)` et par `range(debut, fin)`.

Tester `demo_range_1arg(10)` et `demo_range_2arg(1, 10)` dans la console.

À retenir

La boucle `for element in itérable:` parcourt un par un les éléments d'un itérable ; la variable `element` ne doit jamais être modifiée dans le corps de la boucle.

L'itérateur `range(début, fin, pas)` génère les entiers de `début` (inclus) à `fin` (exclu), par pas de `pas`. Avec un seul argument, `range(fin)` part de 0 avec un pas de 1 ; avec deux arguments, `range(début, fin)` utilise un pas de 1.

Une boucle `for` peut toujours être réécrite avec une boucle `while`, en gérant soi-même le compteur ; l'inverse n'est vrai que si le nombre d'itérations est connu à l'avance.

§4. Interrompre une boucle ou passer à l'itération suivante : break et continue

§4.1. L'instruction break

`break` **met fin immédiatement à la boucle en cours** et poursuit l'exécution juste après elle. On l'utilise généralement lorsqu'une condition testée avec `if` doit provoquer une sortie immédiate de la boucle. `break` s'utilise aussi bien dans une boucle `for` que dans une boucle `while` ; elle est en particulier indispensable pour sortir d'une boucle délibérément infinie, écrite avec `while True:`.

Q11. Les deux fonctions suivantes déterminent, par une méthode peu efficace, le plus grand diviseur d'un entier `nbre` (différent de `nbre` lui-même) : on teste, en partant de `nbre - 1`, tous les diviseurs possibles, jusqu'à en trouver un qui convient.

```
def plus_grand_diviseur(nbre: int) -> str:
    """Retourne un message donnant le plus grand diviseur de nbre,
    différent de nbre lui-même.
    """
    diviseur = nbre - 1
    while diviseur > 0:
        if nbre % diviseur == 0:
            break
        diviseur -= 1
    return f"Le plus grand diviseur de {nbre} est {diviseur}."
```
Python

```python
def plus_grand_diviseur_2(nbre: int) -> str:
    """Retourne un message donnant le plus grand diviseur de nbre,
    différent de nbre lui-même.
    """
    for diviseur in range(nbre - 1, 0, -1):
        if nbre % diviseur == 0:
            break
    return f"Le plus grand diviseur de {nbre} est {diviseur}."
```

Tester les deux fonctions avec `nbre = 18`. Pourquoi est-il inutile de continuer la boucle une fois un diviseur trouvé ? Dans la seconde fonction, à quel endroit du programme la variable `diviseur` reste-t-elle accessible, alors qu'elle est définie par la boucle `for` ?

§4.2. L'instruction continue

`continue` **interrompt seulement l'itération courante** et reprend au début de l'itération suivante : nouvelle évaluation de la condition pour une boucle `while`, élément suivant pour une boucle `for`.

Q12. Les deux fonctions suivantes retournent une chaîne de caractères formée des entiers, inférieurs à `nbre`, qui **ne le divisent pas** :

```python
def non_diviseur(nbre: int) -> str:
    """Retourne les entiers de 1 à nbre - 1 qui ne divisent pas nbre."""
    reponse = ""
    diviseur = nbre - 1
    while diviseur > 0:
        if nbre % diviseur == 0:
            diviseur -= 1
            continue
        reponse += str(diviseur) + " "
        diviseur -= 1
    return reponse

def non_diviseur_2(nbre: int) -> str:
    """Retourne les entiers de 1 à nbre - 1 qui ne divisent pas nbre."""
    reponse = ""
    for diviseur in range(nbre - 1, 0, -1):
        if nbre % diviseur == 0:
            continue
        reponse += str(diviseur) + " "
    return reponse
```

Tester les deux fonctions avec `nbre = 18`. Dans `non_diviseur`, pourquoi l'instruction `diviseur -= 1` apparaît-elle à deux endroits différents ?

Q13. Réécrire la fonction `non_diviseur_2` sous la forme `non_diviseur_3`, **sans utiliser** `continue`, à l'aide d'un simple `if`, et en construisant `reponse` à l'aide d'une f-string plutôt que de `str(diviseur) + " "`. Cette nouvelle version est-elle plus lisible ?

À retenir

— `break` **arrête définitivement** la boucle en cours ; l'exécution reprend juste après elle.
— `continue` **arrête seulement l'itération en cours** ; l'exécution reprend au début de l'itération suivante.

Ces deux instructions s'utilisent aussi bien dans une boucle `for` que dans une boucle `while`.

§5. Exercices de synthèse

Consigne

Écrire chaque fonction dans la partie principale de l'éditeur, avec sa spécification complète (signature + documentation), en utilisant des f-strings pour construire les chaînes de caractères (voir Doc. 2,2). Tester ensuite son appel dans la console.

§5.1. Construire un résultat au fil des itérations

Q14. Écrire la fonction `triple_du_precedent` dont la spécification est :

```python
def triple_du_precedent(a: int) -> str:
    """
    Retourne une chaîne de caractères formée d'une suite de 12 entiers dont chaque terme est
    le triple du précédent, à partir du premier entier a (positif) passé en argument.

    >>> triple_du_precedent(1)
    '1 3 9 27 81 243 729 2187 6561 19683 59049 177147 '
    >>> triple_du_precedent(5)
    '5 15 45 135 405 1215 3645 10935 32805 98415 295245 885735 '
    """
```

Utiliser une boucle `while`.

Q15. Écrire la fonction `multiplication_multiple_trois` dont la spécification est :

```python
def multiplication_multiple_trois(a: int) -> str:
    """Retourne les 10 premiers termes de la table de multiplication
    de a, en signalant par un astérisque ceux qui sont multiples de 3.

    >>> multiplication_multiple_trois(2)
    '2 4 6* 8 10 12* 14 16 18* 20 '
    >>> multiplication_multiple_trois(7)
    '7 14 21* 28 35 42* 49 56 63* 70 '
    """
```

Q16. Écrire la fonction `multiplication_multiple_sept` dont la spécification est :

```python
def multiplication_multiple_sept(a: int) -> str:
    """Calcule les 50 premiers termes de la table de multiplication
    de a et retourne une chaîne de caractères contenant uniquement
```

```
    les multiples de 7.

    >>> multiplication_multiple_sept(2)
    '14 28 42 56 70 84 98 '
    >>> multiplication_multiple_sept(9)
    '63 126 189 252 315 378 441 '
    """
```

En quoi cette fonction diffère-t-elle de `multiplication_multiple_trois` : marque-t-elle simplement des valeurs, ou en sélectionne-t-elle certaines ?

Q17. Écrire la fonction `diviseurs` dont la spécification est :

```python
def diviseurs(a: int) -> str:
    """Retourne une chaîne de caractères formée de la liste des
    diviseurs de a, de a lui-même jusqu'à 1.

    >>> diviseurs(18)
    '18 9 6 3 2 1 '
    """
```

Q18. Écrire et exécuter la fonction `loto_naif` dont la spécification est :

```python
def loto_naif() -> str:
    """Retourne 6 entiers sélectionnés aléatoirement dans
    l'intervalle [1, 49].
    """
```

Remarque. Normalement, un numéro déjà tiré ne peut pas réapparaître. On acceptera ici qu'un même numéro apparaisse plusieurs fois, faute de disposer encore d'une structure de données adaptée pour mémoriser les numéros déjà sortis.

§5.2. Boucles imbriquées

Q19. Écrire la fonction `table_multiplication` dont la spécification est :

```python
def table_multiplication() -> str:
    """Retourne les tables de multiplication de 1 à 10, chaque table
    étant écrite sur une ligne distincte.
    """
```

Indication. Utiliser deux boucles `while` imbriquées, et le caractère `"\n"` pour passer à la ligne. Tester avec `print(table_multiplication())` dans la console, pour visualiser le résultat sous forme de tableau.

§5.3. Demander une information à l'utilisateur

La fonction `input` retourne **toujours** une chaîne de caractères ; une conversion est donc nécessaire pour récupérer un nombre :

```python
valeur = float(input("Entrez votre valeur : "))
```

Q20. Écrire la fonction `plus_grand_plus_petit` dont la spécification est :

```python
def plus_grand_plus_petit(nbre_valeurs: int) -> str:
    """Demande à l'utilisateur nbre_valeurs valeurs et retourne une
    chaîne de caractères donnant le maximum et le minimum obtenus.
    """
```

Q21. Écrire la fonction `moyenne` dont la spécification est :

```python
def moyenne(nbre_notes: int) -> float:
    """Demande à l'utilisateur nbre_notes notes et retourne leur
    moyenne.
    """
```

§5.4. Boucles pilotées par un événement imprévisible

Dans les deux fonctions précédentes, le nombre de saisies est un paramètre fixé à l'avance : une boucle `while` n'était donc pas indispensable. Il existe des situations où ce nombre ne peut au contraire **pas** être connu avant l'exécution.

Q22. On souhaite ne plus imposer le nombre de notes à saisir : l'utilisateur arrête la saisie en entrant une note négative. Écrire la fonction `moyenne_ter` dont la spécification est :

```python
def moyenne_ter() -> str:
    """Demande des notes à l'utilisateur et retourne un message
    donnant leur nombre et leur moyenne. La saisie s'arrête dès
    qu'une note négative est entrée.
    """
```

Indication. Écrire `while True:` et utiliser `break` pour sortir de la boucle. Pourquoi une boucle `for` ne convient-elle pas ici ?

Q23. Écrire et exécuter la fonction `devine` dont la spécification est :

```python
def devine() -> None:
    """Détermine aléatoirement un entier compris entre 1 et 50 et
    demande à l'utilisateur de le deviner. À chaque proposition,
    affiche si elle est trop grande ou trop petite. S'arrête et
    annonce le nombre de tentatives lorsque le nombre est deviné.
    """
```

Indication. Le module `random` et sa fonction `randint` ont été présentés dans le Doc. 2,1.

§5.5. Manipuler le code ASCII

Q24. Écrire et exécuter la fonction `alphabet` dont la spécification est :

```python
def alphabet(sens: str) -> str:
    """Retourne les 26 lettres de l'alphabet dans l'ordre croissant
    si sens vaut 'croissant', dans l'ordre décroissant si sens vaut
    'decroissant'.
    """
```

Indication. `ord(car)` retourne le code ASCII du caractère `car` ; `chr(code)` retourne le caractère correspondant à `code`. Le code de `'a'` est `97`, celui de `'z'` est `122`.

§5.6. Suites définies par récurrence

Q25. On rappelle la suite de Fibonacci, définie par $u_1 = 1$, $u_2 = 1$ et, pour $n > 2$, $u_n = u_{n-1} + u_{n-2}$.

Écrire la fonction `fibo` dont la spécification est :

```python
def fibo(n: int) -> str:
    """Retourne les n premiers termes de la suite de Fibonacci,
    sous forme d'une chaîne de caractères.
    """
```

Indication. Traiter séparément les cas `n == 1` et `n == 2`, puis utiliser deux variables `u` et `v` qui mémorisent les deux derniers termes calculés.

§5.7. Choisir entre for et while

Q26. Reprendre les fonctions `triple_du_precedent`, `multiplication_multiple_trois`, `multiplication_multiple_sept`, `diviseurs`, `table_multiplication`, `plus_grand_plus_petit`, `moyenne`, `loto_naif` et `alphabet`, et les réécrire à l'aide de l'instruction `for`.

Pourquoi `moyenne_ter` et `devine` ne peuvent-elles pas, elles, être réécrites avec `for` ?

À retenir

Le choix entre `for` et `while` dépend de la nature du problème, pas des préférences du programmeur :

- si le nombre d'itérations est **connu avant l'exécution** (un nombre de termes, une taille fixée), `for` est le choix naturel et le plus lisible ;
- si ce nombre **dépend du déroulement de l'exécution** (une saisie jusqu'à une valeur correcte, une recherche jusqu'à un résultat), seule une boucle `while` convient.

§5.8. Pour aller plus loin (difficile)

Q27. **Code ISBN.** Le code ISBN (« International Standard Book Number ») identifie un livre à l'aide de dix chiffres. Le dernier chiffre est une **somme de contrôle**, calculée à partir des neuf premiers : si $d_1, \ldots, d_9$ désignent ces neuf chiffres et d_{10} le dixième, la somme $d_1 + 2d_2 + 3d_3 + \ldots + 10d_{10}$ doit être un multiple de 11.

Écrire une fonction qui demande à l'utilisateur les neuf premiers chiffres d'un code ISBN et retourne le code complet sous forme d'une chaîne de caractères.

Q28. **Calendrier.** Écrire une fonction qui affiche un calendrier allant du 1er janvier 2001 au 31 décembre 3000, en indiquant pour chaque jour le nombre de jours écoulés depuis le 1er janvier 2001, le jour de la semaine, le quantième, le mois et l'année.

Indication. Une année est bissextile si son millésime est multiple de 4, sauf s'il est multiple de 100 sans être multiple de 400 (1900 n'était pas bissextile, 2000 l'a été).

Q29. **Le taxi de Ramanujan.** Trouver tous les entiers inférieurs ou égaux à un nombre N choisi par l'utilisateur qui peuvent s'écrire comme somme de deux cubes, de deux façons différentes : trouver quatre entiers a, b, c, d, inférieurs à N, tels que $a^3 + b^3 = c^3 + d^3$.

Indication. Quatre boucles imbriquées suffisent, même si le résultat n'est pas optimisé.

§6. Bilan : où en suis-je ?

☑ Bilan : où en suis-je ?

Pour chaque ligne, cochez la colonne qui correspond. Toute case **Pas encore** indique une partie à reprendre.

Je sais…	**Oui**	**Pas encore**	**À revoir**
distinguer une **boucle déterministe** (`for`) d'une **boucle non déterministe** (`while`)			Partie 1
écrire une boucle `while` (initialisation, condition d'arrêt, mise à jour) et prévoir si elle peut ne jamais s'exécuter			Partie 2
« faire tourner une boucle à la main » à l'aide d'un **tableau d'évolution**			Partie 2
écrire une boucle `for` en utilisant `range` avec un, deux ou trois arguments			Partie 3
utiliser `break` pour interrompre une boucle et `continue` pour passer à l'itération suivante			Partie 4
écrire une fonction qui accumule un résultat (chaîne, somme, maximum) au fil d'une boucle			Partie 5
choisir entre `for` et `while` selon que le nombre d'itérations est connu ou non à l'avance			Partie 5