Structures de contrôle et manipulation de données en Python

Les structures conditionnelles : if, elif et else

La structure conditionnelle constitue l'un des piliers fondamentaux de la programmation en Python. Elle permet d'exécuter différents blocs de code en fonction de l'évaluation d'expressions booléennes. La forme la plus simple est l'instruction `if`, qui évalue une condition et exécute le bloc indenté suivant uniquement si cette condition est vraie.

Considérons un premier exemple illustrant cette structure élémentaire :

```
temperature = 25

if temperature > 30:
    print("Il fait très chaud")
```

Dans cet exemple, le message ne s'affichera que si la température dépasse **strictement** 30 degrés. Lorsque plusieurs alternatives doivent être envisagées, la clause `elif` (contraction de "else if") permet de tester successivement différentes conditions. Cette structure s'avère particulièrement utile lorsque plusieurs scénarios mutuellement exclusifs doivent être gérés.

Examinons un exemple plus élaboré intégrant `elif` :

```
note = 14

if note >= 16:
    mention = "Très bien"
elif note >= 14:
    mention = "Bien"
elif note >= 12:
    mention = "Assez bien"
elif note >= 10:
    mention = "Passable"
else:
    mention = "Insuffisant"

print(f"Mention obtenue : {mention}")
```

Il faut noter que Python évalue les conditions dans l'ordre séquentiel d'apparition et s'arrête dès qu'une condition est satisfaite. Ainsi, dans l'exemple précédent, une note de 14 satisfait la condition `note >= 14` et le programme n'évaluera pas les conditions suivantes. La clause `else` finale capture tous les cas non traités par les conditions précédentes, jouant le rôle d'une option par défaut.

Les conditions peuvent également être combinées à l'aide des opérateurs logiques `and`, `or` et `not`, permettant d'exprimer des logiques complexes. Par exemple :

```
age = 20
possede_permis = True

if age >= 18 and possede_permis:
    print("Vous pouvez conduire")
elif age >= 18 and not possede_permis:
    print("Vous devez obtenir votre permis")
else:
    print("Vous êtes trop jeune pour conduire")
```

Les boucles `for` et la fonction `range`

La boucle `for` en Python représente une structure itérative qui permet de parcourir séquentiellement les éléments d'un objet itérable. Contrairement à d'autres langages où la boucle `for` repose sur un compteur explicite, Python adopte une approche plus élégante basée sur l'itération directe. La fonction `range` se révèle particulièrement utile lorsqu'il est nécessaire de générer une séquence numérique pour contrôler le nombre d'itérations.

La fonction `range` peut être invoquée selon trois signatures différentes. La forme la plus simple, `range(n)`, génère une séquence d'entiers de 0 à n-1. Voici une illustration basique :

```
for i in range(5): # 0 1 2 3 4
    print(f"Itération numéro {i}")
```

Ce code produira les itérations numérotées de 0 à 4, démontrant que la borne supérieure est exclusive. La deuxième forme, `range(debut, fin)`, permet de spécifier à la fois la valeur de départ et la valeur finale. Par exemple :

```
somme = 0
for nombre in range(1, 11): # 1 2 3 4 5 6 7 8 9 10
    somme += nombre
print(f"La somme des nombres de 1 à 10 est : {somme}")
```

La troisième forme, `range(debut, fin, pas)`, introduit un paramètre supplémentaire définissant l'incrément entre chaque valeur. Cette variante s'avère précieuse pour parcourir des séquences avec un pas différent de 1, ou même pour itérer en ordre décroissant avec un pas négatif :

```
print("Compte à rebours :")
for seconde in range(10, 0, -1): # 10 9 8 7 6 5 4 3 2 1
    print(seconde)
print("Décollage !")
```

Les boucles `for` peuvent également être imbriquées pour traiter des structures multidimensionnelles. Considérons la génération d'une table de multiplication :

```
for i in range(1, 6):
    for j in range(1, 6):
        produit = i * j
        print(f"{i} × {j} = {produit}")
    print()  # Ligne vide après chaque table
```

Il est également possible d'utiliser `range` en conjonction avec des listes existantes lorsqu'on souhaite accéder simultanément aux indices et aux valeurs :

```
fruits = ["pomme", "banane", "orange", "kiwi"]

for index in range(len(fruits)):
    print(f"Le fruit à l'index {index} est : {fruits[index]}")
```

Les boucles `while` en Python

La boucle `while` permet d'exécuter un bloc d'instructions tant qu'une condition est vraie. Sa syntaxe repose sur un principe simple : **"tant que la condition est vraie, continue"**.

```
compteur = 0

while compteur < 5:
    print(f"Compteur : {compteur}")
    compteur += 1
```

Ici, la boucle commence avec `compteur = 0`. À chaque itération, Python vérifie la condition `compteur < 5`. Tant qu'elle est vraie, le bloc s'exécute ; dès qu'elle devient fausse (`compteur` vaut 5), la boucle s'arrête.

Il faut toujours s'assurer que la condition finira par devenir fausse, sinon la boucle devient infinie :

```
# Exemple de boucle infinie (à éviter)
while True:
    print("Ceci s'affichera sans fin")
```

Les boucles `while` sont donc particulièrement utiles lorsque **on ne connaît pas à l'avance le nombre d'itérations nécessaires**. Elles s'appuient sur une **condition logique** plutôt que sur une séquence.

Exemple concret : attendre une saisie correcte d'un utilisateur.

```
mot_de_passe = ""
while mot_de_passe != "python":
    mot_de_passe = input("Entrez le mot de passe : ")

print("Mot de passe correct !")
```

Ici, la boucle se répète jusqu'à ce que l'utilisateur entre la bonne valeur. Le nombre de tours n'est pas défini : il dépend du comportement de l'utilisateur.

Quand utiliser `for` ou `while`

La boucle `for` s'utilise lorsqu'on **connaît à l'avance le nombre d'itérations** ou lorsqu'on souhaite **parcourir une séquence** (liste, chaîne, dictionnaire, etc.).

Exemple :

```
for i in range(5):
    print(i)
```

Ici, on sait qu'il y aura exactement cinq tours.

La boucle `while`, au contraire, s'utilise quand **on ne connaît pas à l'avance le nombre de répétitions**, et qu'on dépend d'une **condition de sortie**. Elle convient pour :

- attendre un événement ou une saisie utilisateur ;
- vérifier une condition jusqu'à ce qu'elle soit remplie ;
- répéter une opération tant qu'un état n'est pas atteint.

Situation	Boucle recommandée
On connaît le nombre d'itérations	`for`
On ne connaît pas le nombre d'itérations	`while`
On parcourt une collection (liste, tuple, dict)	`for`
On attend une condition (entrée, changement d'état)	`while`

Exemple comparatif

```
# Boucle for : on connaît la fin
for i in range(3):
    print("Essai numéro", i + 1)

# Boucle while : on attend la bonne réponse
reponse = ""
while reponse.lower() != "oui":
    reponse = input("Voulez-vous continuer ? (oui/non) : ")
```

La première s'arrête après trois passages ; la seconde continue tant que la condition n'est pas satisfaite.

La gestion des interruptions de boucle : `break`, `continue` et `else` dans les boucles

Lorsqu'on écrit une boucle, il est fréquent d'avoir besoin de **modifier son déroulement normal**. Par exemple, on peut vouloir quitter la boucle prématurément, passer une itération sans exécuter le reste du code, ou exécuter un bloc final seulement si la boucle s'est terminée naturellement. Python met à disposition trois mots-clés dédiés à ce contrôle fin du flux d'exécution : `break`, `continue` et `else`.

1. L'instruction `break`

L'instruction `break` permet de **sortir immédiatement d'une boucle**, quelle que soit la condition initiale. Dès que Python rencontre un `break`, il interrompt la boucle et passe directement à la suite du programme.

Exemple :

```
while True:
    reponse = input("Entrez un mot (ou 'stop' pour quitter) : ")
    if reponse == "stop":
        print("Arrêt demandé, on quitte la boucle.")
        break
    print(f"Vous avez écrit : {reponse}")
```

Ici, la boucle `while` est théoriquement infinie (`while True:`), mais le `break` permet d'en sortir lorsque l'utilisateur saisit le mot *stop*. C'est une structure très courante lorsqu'on attend une saisie valide, un signal d'arrêt ou une condition particulière.

2. L'instruction `continue`

L'instruction `continue` permet, au contraire, de **sauter le reste du bloc courant** et de **passer directement à l'itération suivante**. Elle est utile lorsqu'on souhaite ignorer certains cas sans interrompre la boucle entière.

Exemple :

```
for nombre in range(1, 6):
    if nombre % 2 == 0:
        continue  # ignore les nombres pairs
    print(f"Nombre impair : {nombre}")
```

Dans cet exemple, lorsque le nombre est pair, la ligne `continue` fait passer immédiatement Python à la prochaine valeur du `for`, sans exécuter `print()`. Résultat : seuls les nombres impairs s'affichent.

On peut s'en servir également dans des traitements où certaines données doivent être écartées :

```
nombres = [12, -4, 0, 15, -9, 7]
for n in nombres:
    if n < 0:
        continue  # on ignore les nombres négatifs
    print(f"Nombre positif : {n}")
```

3. La clause `else` sur une boucle

Moins connue mais très élégante, Python permet d'associer un bloc `else` à une boucle `for` ou `while`. Le `else` s'exécute **uniquement si la boucle s'est terminée sans qu'un `break` n'ait été rencontré**.

Cela permet de distinguer une sortie "naturelle" d'une sortie "interrompue".

Exemple avec une boucle `for` :

```
for tentative in range(3):
    mot = input("Mot de passe : ")
    if mot == "python":
        print("Accès autorisé.")
        break
else:
    print("Trop de tentatives, accès refusé.")
```

Explication :

- Si l'utilisateur entre "python" avant la troisième tentative, le `break` interrompt la boucle et le bloc `else` n'est **pas exécuté**.
- Si aucune des trois tentatives n'a réussi, la boucle se termine normalement et le bloc `else` est exécuté.

On peut également utiliser `else` avec une boucle `while`, par exemple pour vérifier une condition jusqu'à un certain seuil :

```
n = 0
while n < 5:
    if n == 3:
        print("Valeur interdite rencontrée, arrêt du programme.")
        break
    n += 1
else:
    print("La boucle s'est terminée normalement sans interruption.")
```

Mot-clé	Effet sur la boucle	Quand l'utiliser
`break`	Stoppe immédiatement la boucle	Lorsqu'une condition de sortie est atteinte (ex. mot de passe correct, élément trouvé)

Mot-clé	Effet sur la boucle	Quand l'utiliser
`continue`	Ignore le reste du bloc et passe à l'itération suivante	Lorsqu'on veut sauter certains cas sans arrêter la boucle
`else`	S'exécute uniquement si la boucle s'est terminée sans `break`	Pour distinguer une sortie normale d'une sortie interrompue

Exemple complet

```
nombres = [3, 5, 7, 9, 10, 13]

for n in nombres:
    if n % 2 == 0:
        print(f"{n} est pair, on arrête la recherche.")
        break
    if n < 0:
        continue
    print(f"{n} est un nombre impair positif.")
else:
    print("Aucun nombre pair trouvé dans la liste.")
```

Ici :

- `continue` aurait permis d'ignorer les nombres négatifs (s'il y en avait),
- `break` arrête la boucle dès qu'un nombre pair est trouvé,
- le `else` ne s'exécute que si aucun `break` n'a eu lieu.

Ce mécanisme rend les boucles Python **plus expressives et plus précises** : on ne se contente plus de répéter un bloc d'instructions, on contrôle réellement le comportement du programme selon les événements rencontrés.

Les listes de compréhension

Les listes de compréhension constituent une fonctionnalité distinctive et élégante de Python, permettant de créer des listes de manière concise et expressive. Cette construction syntaxique offre une alternative compacte aux boucles traditionnelles pour générer ou transformer des séquences de données. La syntaxe générale d'une liste de compréhension s'articule autour de l'expression `[expression for element in iterable]`.

Comparons d'abord une approche classique avec une liste de compréhension. Pour créer une liste des carrés des dix premiers entiers, on pourrait traditionnellement écrire :

```
carres = []
for i in range(10):
    carres.append(i ** 2)
```

Cette même opération peut être exprimée de manière beaucoup plus concise avec une liste de compréhension :

```
carres = [i ** 2 for i in range(10)]
```

Les listes de compréhension peuvent incorporer des conditions pour filtrer les éléments. La syntaxe devient alors `[expression for element in iterable if condition]`. Prenons l'exemple suivant qui extrait uniquement les nombres pairs d'une séquence :

```
nombres = range(20)
pairs = [n for n in nombres if n % 2 == 0]
print(pairs)  # Affiche [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Il est également possible d'appliquer des transformations conditionnelles sur les éléments. Supposons que nous souhaitions créer une liste où les nombres pairs sont multipliés par deux et les nombres impairs par trois :

```
transformes = [n * 2 if n % 2 == 0 else n * 3 for n in range(10)]
```

Notons que dans cette construction, l'expression conditionnelle précède le `for`, ce qui diffère de la syntaxe de filtrage présentée précédemment. Cette distinction syntaxique est fondamentale : lorsque le `if` apparaît après le `for`, il s'agit d'un filtre ; lorsqu'il fait partie d'une expression ternaire avant le `for`, il s'agit d'une transformation conditionnelle.

Retenez:

- Si **`if`** tout seul, il se retrouve après le **`for`** et ne fait pas partie d'une expression ternaire.
- Si **`if/else`**, alors il se retrouve avant le **`for`** et fait partie d'une expression ternaire.

Les listes de compréhension peuvent également opérer sur des chaînes de caractères et d'autres structures itérables. Par exemple, pour extraire les voyelles d'une phrase :

```
phrase = "Python est un langage extraordinaire"
voyelles = [lettre for lettre in phrase if lettre.lower() in "aeiouy"]
print(''.join(voyelles))
```

Les listes de compréhension imbriquées permettent de traiter des structures bidimensionnelles. Pour créer une matrice 3×3 contenant les coordonnées de chaque position :

```
matrice = [[i, j] for i in range(3) for j in range(3)]
```

Cette expression génère une liste plate de paires de coordonnées. Pour obtenir une véritable structure bidimensionnelle, on peut imbriquer une liste de compréhension dans une autre :

```
matrice = [[i * j for j in range(5)] for i in range(5)]
```

Cette construction crée une table de multiplication sous forme de liste de listes, où chaque sous-liste représente une ligne de la table.

Les listes de compréhension offrent non seulement une syntaxe plus concise, mais elles sont également généralement plus performantes que les boucles `for` traditionnelles avec `append`, car Python optimise leur exécution en interne. Toutefois, il convient de préserver la lisibilité du code : lorsqu'une liste de compréhension devient trop complexe, il peut être préférable de revenir à une boucle explicite pour maintenir la clarté du programme.