Boucles

Bases de Python

1 Pourquoi les boucles ?

En programmation, on a souvent besoin de **répéter** une action : afficher chaque caractère d'un texte, calculer une somme, recommencer une saisie tant qu'elle est incorrecte, etc. Écrire manuellement chaque instruction serait fastidieux et peu flexible. Pour afficher cinq fois « Bonjour », on pourrait recopier cinq fois la même ligne :

```
print("Bonjour")
print("Bonjour")
print("Bonjour")
print("Bonjour")
print("Bonjour")
```

Mais pour l'afficher mille fois, ce serait ingérable. Une **boucle** répète la ligne à notre place, en deux lignes seulement ; pour mille affichages, il suffit de remplacer `5` par `1000`.

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

```
Bonjour
Bonjour
Bonjour
Bonjour
Bonjour
```

Python propose deux types de boucles : la boucle `for` (quand on connaît le nombre de répétitions) et la boucle `while` (quand on répète tant qu'une condition est vérifiée).

2 La boucle `for`

Parcourir une chaîne de caractères

La boucle `for` parcourt un à un les éléments d'une **séquence**. La première séquence rencontrée en Python est la chaîne de caractères : la boucle en parcourt les caractères, dans l'ordre. À chaque tour de boucle, la variable `lettre` prend la valeur du caractère suivant.

```
mot = "NSI"
for lettre in mot:
    print(lettre, end=" ")
```

Question.
Qu'affiche ce programme ?

Réponse.

```
N S I
```

Au premier tour, `lettre` vaut `"N"`, puis `"S"`, puis `"I"`. L'argument `end=" "` remplace le retour à la ligne de `print()` par une espace : les caractères s'affichent sur une même ligne.

D'autres séquences, comme les listes, se parcourent de la même façon ; elles sont étudiées plus loin.

La fonction `range()`

La fonction `range()` génère une séquence de nombres entiers. Elle est très utilisée avec la boucle `for` :

```
for i in range(5):
    print(i, end=" ")
```

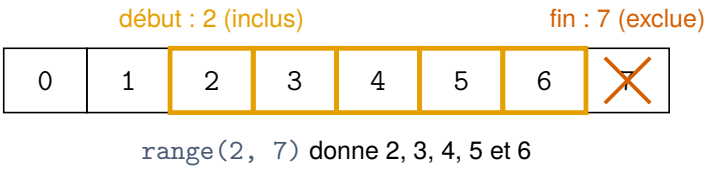
Question.
Quelles valeurs la variable `i` prend-elle ?

Réponse.

```
0 1 2 3 4
```

`range(5)` donne cinq nombres : le comptage commence à 0, et la fin, 5, est **exclue**.

On peut aussi choisir le premier nombre : `range(début, fin)` va de `début` inclus à `fin` exclue.



`range(2, 7)` donne 2, 3, 4, 5 et 6

```
# range(début, fin) : de début à fin-1
for i in range(2, 7):
    print(i, end=" ")
```

```
2 3 4 5 6
```

Un troisième nombre fixe le **pas**, c'est-à-dire l'écart entre deux valeurs :

```python
# range(début, fin, pas) : de début à fin-1, par pas
for i in range(0, 20, 5):
    print(i, end=" ")
```

```
0 5 10 15
```

⚠ **Attention.** la borne de fin est **exclue** : `range(2, 7)` s'arrête à 6. Pour compter de 1 à 5, il faut donc écrire `range(1, 6)`.

Accumuler un résultat

Un schéma très courant consiste à initialiser une variable (un **accumulateur**) puis à la modifier à chaque tour de boucle :

```python
# Calculer la somme des nombres de 1 à 5
total = 0
for i in range(1, 6):
    total += i
print(f"La somme vaut {total}")
```

```
La somme vaut 15
```

Trace d'exécution. Pour bien comprendre, suivons les valeurs des variables à chaque tour de boucle :

Tour	i	total avant	total += i	total après
1	1	0	0 + 1	1
2	2	1	1 + 2	3
3	3	3	3 + 3	6
4	4	6	6 + 4	10
5	5	10	10 + 5	15

Ce tableau s'appelle une **trace d'exécution** (ou **table de trace**). C'est un outil essentiel pour comprendre et vérifier le comportement d'un programme.

⚠ **Attention.** l'accumulateur reçoit sa valeur de départ **avant** la boucle, et jamais à l'intérieur. Voici ce qui arrive sinon :

```python
for i in range(1, 6):
    total = 0
    total += i
print(total)
```

Question.
Qu'affiche ce programme ?

Réponse.

```
5
```

À chaque tour, la ligne `total = 0` remet l'accumulateur à zéro, puis `total += i` lui ajoute seulement `i`. Après le dernier tour, `total` vaut donc `0 + 5`, c'est-à-dire `5`, et non la somme `15`.

Le même schéma permet de construire une chaîne de caractères, en partant de la chaîne vide :

```
# Construire une chaîne lettre par lettre
mot = "Python"
resultat = ""
for lettre in mot:
    resultat += lettre + "-"
print(resultat)
```

```
P-y-t-h-o-n-
```

Exercice 1 *Boucle `for` et accumulateur* (4 points)

1. Que produit le programme suivant ?

```
for i in range(1, 6):
    print(i * i, end=" ")
```

2. Écrire un programme qui calcule le **produit** des entiers de 1 à n (factorielle de n), où n est demandé à l'utilisateur. Par exemple, pour $n = 5$: $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.

3. Écrire un programme qui demande un entier n et affiche les n premiers termes de la table de multiplication de 7.

   Exemple pour $n = 5$: `7 14 21 28 35`.

③ La boucle `while`

Principe

La boucle `while` (« tant que ») répète un bloc d'instructions **tant qu'une condition est vraie** :

```
compteur = 1
while compteur <= 5:
    print(compteur)
    compteur += 1
```

```
1
2
3
4
5
```

Trace d'exécution :

Tour	compteur	compteur <= 5 ?	Action	compteur après
1	1	Vrai	affiche 1	2
2	2	Vrai	affiche 2	3
3	3	Vrai	affiche 3	4
4	4	Vrai	affiche 4	5
5	5	Vrai	affiche 5	6
—	6	Faux	sortie de boucle	—

Question.
Combien de fois la condition `compteur <= 5` est-elle testée ?

Réponse.
Six fois. La condition est testée **avant** chaque tour : cinq fois elle est vraie, et la boucle exécute son bloc ; la sixième fois, `compteur` vaut 6, la condition est fausse, et la boucle s'arrête. C'est l'instruction `compteur += 1` qui permet à la boucle de s'arrêter.

Attention. si la condition ne devient jamais fausse, la boucle tourne indéfiniment (boucle infinie). Il faut toujours s'assurer que quelque chose dans le corps de la boucle finira par rendre la condition fausse.

Quand utiliser `while` plutôt que `for` ?

On utilise `for` quand on connaît à l'avance le nombre de tours : cinq tours, ou un tour par lettre d'un mot. On utilise `while` lorsqu'on **ne sait pas à l'avance** combien de fois la boucle va s'exécuter :

```
# Demander un mot de passe jusqu'à obtenir le bon
mot_de_passe = ""
while mot_de_passe != "nsi2026":
```

```
    mot_de_passe = input("Mot de passe : ")
print("Accès autorisé !")
```

Personne ne sait combien d'essais il faudra. De même, on ne connaît pas à l'avance le nombre de tours de la boucle suivante :

```
# Trouver la plus petite puissance de 2 supérieure à 1000
puissance = 1
while puissance <= 1000:
    puissance *= 2
print(puissance)  # 1024
```

Exercice 2 *Boucle* `while` **(4 points)**

1. Écrire un programme qui affiche tous les multiples de 3 inférieurs ou égaux à 50 en utilisant une boucle `while`.

2. Écrire un programme qui demande des nombres à l'utilisateur jusqu'à ce qu'il entre 0. Le programme affiche alors la somme de tous les nombres entrés (sans compter le 0 final).

3. Écrire un programme qui calcule le plus petit entier n tel que $1 + 2 + 3 + \cdots + n \geq 1000$.

④ Instructions de contrôle

`break` : sortir de la boucle

L'instruction `break` **interrompt immédiatement** la boucle, même si la condition est encore vraie ou que la séquence n'est pas terminée :

```
# Chercher la première voyelle dans un mot
mot = "python"
for lettre in mot:
    if lettre in "aeiouy":
```

```
        print(f"Première voyelle : {lettre}")
        break
```

```
Première voyelle : y
```

Simuler un `do...while`

Python n'a pas de boucle `do...while`. Pour exécuter le corps au moins une fois, on peut utiliser `while True` avec `break` :

```
while True:
    reponse = input("Entrez un nombre positif : ")
    nombre = int(reponse)
    if nombre > 0:
        break
    print("Erreur : le nombre doit être positif !")
print(f"Vous avez entré {nombre}")
```

`continue` : passer au tour suivant

L'instruction `continue` **saute le reste du bloc** et passe directement au tour suivant :

```
# Afficher uniquement les nombres impairs de 1 à 10
for i in range(1, 11):
    if i % 2 == 0:
        continue
    print(i, end=" ")
```

```
1 3 5 7 9
```

⑤ Boucles imbriquées

On peut placer une boucle à l'intérieur d'une autre boucle. La boucle intérieure s'exécute entièrement à chaque tour de la boucle extérieure :

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} × {j} = {i*j}", end="   ")
    print()  # retour à la ligne
```

```
1 × 1 = 1   1 × 2 = 2   1 × 3 = 3
2 × 1 = 2   2 × 2 = 4   2 × 3 = 6
3 × 1 = 3   3 × 2 = 6   3 × 3 = 9
```

Question.
Combien de produits ce programme affiche-t-il, et pourquoi ?

Réponse.
Neuf. La boucle extérieure fait 3 tours, et à chacun d'eux la boucle intérieure fait ses 3 tours en entier : $3 \times 3 = 9$ produits, trois par ligne.

Exercice 3 *Boucles imbriquées et* `break` **(5 points)**

1. Écrire un programme qui affiche un triangle rectangle d'étoiles de hauteur n (demandé à l'utilisateur).

   Exemple pour $n = 4$:

   ```
   *
   **
   ***
   ****
   ```

2. Écrire un programme qui affiche tous les nombres premiers inférieurs ou égaux à n.

   Rappel : un nombre premier est un entier supérieur ou égal à 2 qui n'est divisible que par 1 et par lui-même.

   Indication : pour chaque nombre k de 2 à n, tester s'il existe un diviseur entre 2 et $k-1$. Si aucun diviseur n'est trouvé, k est premier.

6 Le piège de la boucle infinie

Une **boucle infinie** est une boucle dont la condition ne devient jamais fausse. Le programme tourne alors indéfiniment et il faut l'interrompre manuellement (Ctrl+C dans le terminal, ou bouton Stop dans l'éditeur).

```
# NE PAS EXÉCUTER : boucle infinie !
n = 1
while n <= 5:
```

```
    print(n)
    # Oubli : n n'est jamais incrémenté, donc n reste à 1
    # La condition n <= 5 est toujours vraie → boucle infinie
```

La trace montre le problème :

Tour	n	n <= 5 ?	Action
1	1	Vrai	affiche 1, n ne change pas
2	1	Vrai	affiche 1, n ne change pas
3	1	Vrai	affiche 1, n ne change pas
...	1	Vrai	indéfiniment

La correction est simple : il faut ajouter `n += 1` dans le corps de la boucle, pour faire évoluer la variable testée dans la condition.

⑦ Erreurs fréquentes

Modifier la variable de boucle dans un `for` : la modification n'a aucun effet sur le parcours.

```
for i in range(5):
    i = 100       # sans effet : i reprendra la valeur suivante au prochain tour
    print(i)      # affiche 100 à chaque tour, mais ne change rien au parcours
```

Erreur de borne dans `range()` : `range(1, 5)` va de 1 à 4, pas de 1 à 5. Pour aller de 1 à 5, il faut écrire `range(1, 6)`.

Initialiser l'accumulateur dans la boucle : la variable est remise à sa valeur de départ à chaque tour (voir la section « Accumuler un résultat »).

Oublier de faire évoluer la condition d'un `while` : la boucle ne s'arrête jamais.

⑧ À retenir

1. Une boucle répète un bloc d'instructions indenté, sans le recopier.
2. `for` parcourt une séquence : les caractères d'une chaîne, ou les nombres donnés par `range()`. On l'utilise quand on connaît le nombre de tours.
3. `range(n)` donne `0, 1, …, n - 1` ; dans `range(début, fin, pas)`, la fin est toujours exclue.
4. Un accumulateur reçoit sa valeur de départ avant la boucle, jamais à l'intérieur.
5. `while` répète son bloc tant que sa condition est vraie ; la condition est testée avant chaque tour. On l'utilise quand on ne connaît pas le nombre de tours.
6. Dans une boucle `while`, il faut faire évoluer la condition, sinon la boucle ne s'arrête jamais.
7. `break` sort de la boucle, `continue` passe au tour suivant.
8. Une trace d'exécution suit la valeur des variables, tour après tour.

⑨ Résumé

Concept	Syntaxe	Usage
`for`	`for x in sequence:`	Nombre de tours connu
`while`	`while condition:`	Nombre de tours inconnu
`range()`	`range(début, fin, pas)`	Générer des entiers
`break`	`break`	Sortir de la boucle
`continue`	`continue`	Passer au tour suivant