

Programmation Python

Séance 3 — Contrôle du flux

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et
Techniques, Université Cheikh Anta Diop de Dakar



<https://ebtoure.sn/cours/programmation-python/chapitres/03-controle-du-flux.html>

Version du 11 octobre 2026



Version en ligne

Programmation Python

— Séance 3

Contrôle du flux

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

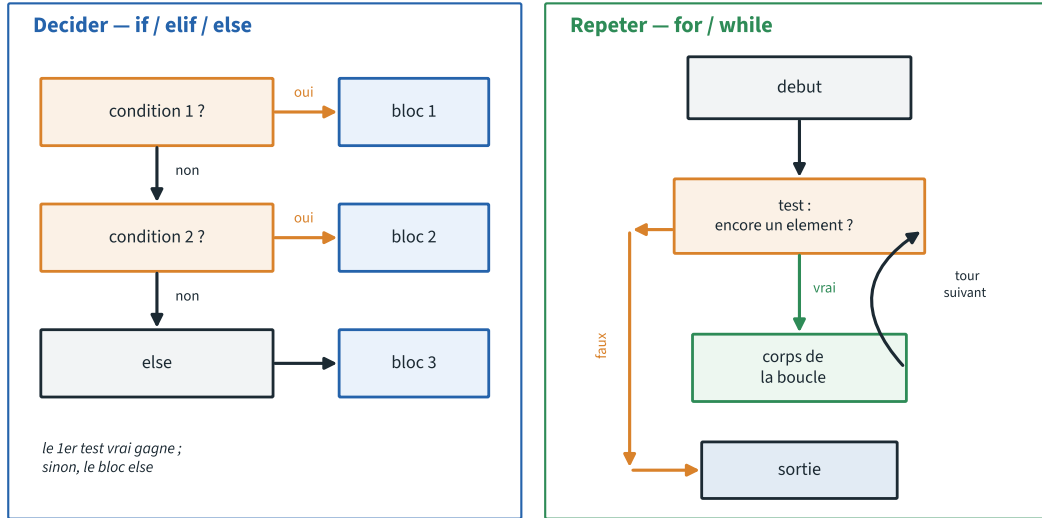
Contenu

Jusqu'ici, le code s'exécutait ligne après ligne, de haut en bas. On apprend à **casser cette ligne droite** : exécuter un bloc *sous condition*, et *répéter* un bloc autant de fois que nécessaire. C'est ce qui rend un programme réellement utile.

- **Conditions** : `if` / `elif` / `else` pour décider.
- **Boucle for** : répéter pour chaque élément ; `range`, accumulation.
- **enumerate** et **zip** : parcourir avec l'indice, ou deux listes à la fois.
- **Boucle while** : répéter tant qu'une condition tient.
- **break** et **continue** : contrôler le déroulement d'une boucle.

L'entraînement se fait dans le **notebook**, avec une cellule « À vous » après chaque notion.

La carte du contrôle du flux



Deux mécanismes : *décider* (suivre une branche selon un test) et *répéter* (reboucler tant qu'un test reste vrai).

Les conditions

L'idée : exécuter un bloc sous condition

L'idée en une phrase

Une instruction `if` exécute un bloc **seulement si** un test est vrai. Le test est un booléen (Séance 1). Deux éléments de syntaxe sont obligatoires : les deux points `:` en fin de ligne, et l'**indentation** du bloc (son décalage vers la droite).

L'indentation n'est pas décorative en Python : c'est elle qui délimite le bloc. Tout ce qui est décalé sous le `if` lui appartient ; dès qu'on revient à gauche, on sort du bloc.

Deux issues : `if` et `else`

```
note = 12
if note >= 10:
    print("Admis")
else:
    print("Recalé")
```

RÉSULTAT →

Admis

Le bloc du `if` s'exécute si le test est vrai ; sinon, c'est le bloc du `else`. Ici `note` vaut 12, donc « Admis ».

Plusieurs cas : `elif`

```
note = 14
if note >= 16:
    mention = "Tres bien"
elif note >= 14:
    mention = "Bien"
elif note >= 10:
    mention = "Passable"
else:
    mention = "Insuffisant"
print(mention)
```

RÉSULTAT →

Bien

Python teste dans l'ordre et s'arrête au **premier** test vrai : 14 n'est pas ≥ 16 , mais est ≥ 14 .

Combiner des conditions

```
note = 15
assidue = True
if note >= 10 and assidue:
    print("Valide")
else:
    print("A revoir")
```

RÉSULTAT →

Valide

On réutilise **and** / **or** / **not** (Séance 1) pour former des tests plus riches.

Piège — les deux points et l'indentation

La syntaxe des blocs

Trois erreurs reviennent sans cesse : **1.** oublier les deux points `:` à la fin de la ligne `if` ; **2.** ne pas indenter le bloc (ou l'indenter de travers) — Python lève une `IndentationError` ; **3.** écrire `=` (affecter) au lieu de `==` (comparer) dans le test.

Règle simple : toute ligne qui se termine par `:` ouvre un bloc, et ce bloc est décalé de quatre espaces. On garde une indentation cohérente dans tout le fichier.

La boucle for

L'idée : répéter pour chaque élément

L'idée en une phrase

Une boucle `for` exécute un bloc **une fois par élément** d'une collection. À chaque tour, la variable de boucle prend la valeur de l'élément courant.

C'est le moyen naturel de traiter une liste, une chaîne, ou toute suite de valeurs, sans écrire le même code plusieurs fois. Comme pour le `if`, le corps de la boucle est un **bloc indenté** après les deux points.

Parcourir une collection

```
villes = ["Dakar", "Louga", "Mbour"]  
for ville in villes:  
    print(ville)
```

RÉSULTAT →

```
Dakar  
Louga  
Mbour
```

La variable `ville` prend successivement chaque valeur de la liste. Le bloc s'exécute donc trois fois.

Répéter un nombre fixé de fois : range

La fonction range

`range(n)` produit les entiers de 0 à `n-1` ; `range(a, b)` va de `a` à `b` exclu.

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

```
0  
1  
2  
3  
4
```

```
for i in range(1, 4):  
    print("tour", i)
```

```
tour 1  
tour 2  
tour 3
```

Le motif clé : l'accumulation

Le schéma de l'accumulation

On prépare une variable **avant** la boucle, avec une valeur de départ, puis on la met à jour à chaque tour.

```
notes = [12, 15, 9, 18]
total = 0
for note in notes:
    total = total + note
print("total :", total)
print("moyenne :", total / len(notes))
```

RÉSULTAT →

```
total : 54
moyenne : 13.5
```

Accumuler pour compter

```
villes = ["Dakar", "Saint-Louis", "Mbour", "Ziguinchor"]
grandes = 0
for ville in villes:
    if len(ville) > 5:
        grandes = grandes + 1
print("villes de plus de 5 lettres :", grandes)
```

RÉSULTAT →

```
villes de plus de 5 lettres : 2
```

On combine boucle et condition : le compteur n'avance que pour les villes dont le nom dépasse 5 lettres (Saint-Louis, Ziguinchor).

Piège — préparer l'accumulateur

Initialiser avant, mettre à jour dedans

La variable qui accumule (`total`, `grandes`) doit être créée **avant** la boucle, avec une valeur de départ (`0` pour une somme). L'initialiser à l'intérieur la remettrait à zéro à chaque tour. Et l'oublier complètement provoque une `NameError` au premier `+=`.

Le schéma est toujours le même : *préparer* → *parcourir en mettant à jour* → *utiliser le résultat après la boucle*.

enumerate et zip

enumerate : l'indice et la valeur

```
villes = ["Dakar", "Louga", "Mbour"]  
for i, ville in enumerate(villes):  
    print(i, ville)
```

RÉSULTAT →

```
0 Dakar  
1 Louga  
2 Mbour
```

`enumerate` fournit à chaque tour un couple (`indice`, `valeur`), qu'on déballe dans deux variables — pratique pour numéroter un affichage.

zip : deux listes en parallèle

```
villes = ["Dakar", "Thies", "Kaolack"]  
population = [1200000, 320000, 240000]  
for ville, hab in zip(villes, population):  
    print(ville, ":", hab)
```

RÉSULTAT →

```
Dakar : 1200000  
Thies : 320000  
Kaolack : 240000
```

`zip` avance dans les deux listes *en même temps* et associe les éléments qui se correspondent, position par position.

La boucle while

L'idée : répéter tant qu'une condition tient

L'idée en une phrase

Une boucle `while` répète un bloc **tant qu'**un test reste vrai. On l'emploie quand on ne connaît pas d'avance le nombre de tours, mais qu'on sait quand *s'arrêter*.

Différence avec `for` : `for` parcourt une collection connue ; `while` tourne jusqu'à ce qu'une condition devienne fausse. Il faut donc faire **évoluer** cette condition à chaque tour, pour finir par sortir.

Compter avec `while`

```
compteur = 0
while compteur < 3:
    print("compteur =", compteur)
    compteur = compteur + 1
print("fin")
```

RÉSULTAT →

```
compteur = 0
compteur = 1
compteur = 2
fin
```

À chaque tour, `compteur` augmente ; quand il atteint 3, le test devient faux et la boucle s'arrête.

Piège — la boucle infinie

Toujours avancer vers la sortie

Si la condition d'un `while` ne devient **jamais** fausse — par exemple si l'on oublie `compteur = compteur + 1` — la boucle tourne sans fin et le programme se bloque. Avant d'écrire un `while`, il faut identifier la variable qui progresse vers la sortie, et vérifier qu'elle change bien à chaque tour.

En pratique : une boucle emballée s'interrompt avec le bouton *Interrupt* du noyau Jupyter.

break et continue

break : arrêter la boucle

```
for n in [1, 2, 3, 4, 5]:  
    if n == 3:  
        break  
    print(n)
```

RÉSULTAT →

1
2

Dès que `n` vaut 3, `break` interrompt complètement la boucle : les valeurs 3, 4 et 5 ne sont jamais affichées.

continue : sauter un tour

```
for n in [1, 2, 3, 4, 5]:  
    if n % 2 == 0:  
        continue  
    print(n)
```

RÉSULTAT →

```
1  
3  
5
```

`continue` saute le reste du bloc et passe au tour suivant. Ici, les nombres pairs sont ignorés, seuls les impairs sont affichés.

Retour sur les compréhensions

Une compréhension est un **for** compact

```
# Version longue : boucle for + append
carres = []
for n in [1, 2, 3, 4]:
    carres.append(n * n)
print(carres)

# Version compacte : comprehension
carres2 = [n * n for n in [1, 2, 3, 4]]
print(carres2)
```

RÉSULTAT →

```
[1, 4, 9, 16]
[1, 4, 9, 16]
```

La compréhension (Séance 2) n'est rien d'autre que cette boucle, écrite en une ligne.

Synthèse

À retenir — Séance 3

L'essentiel

Conditions : `if` / `elif` / `else` ; deux points : puis **bloc indenté** ; le premier test vrai gagne.

Boucle for : `for element in collection` : répète pour chaque élément ; `range(n)` répète n fois.

Accumulation : préparer une variable **avant**, la mettre à jour dans la boucle (somme, compteur).

enumerate : indice + valeur. **zip** : deux listes en parallèle.

Boucle while : `while condition` : tant que c'est vrai ; faire évoluer la condition.

break arrête la boucle ; **continue** passe au tour suivant.

Pièges fréquents

- Oublier les deux points `:` ou mal indenter un bloc : `IndentationError`.
- Confondre `=` (affecter) et `==` (comparer) dans un test.
- Initialiser l'accumulateur *dans* la boucle (remis à zéro chaque tour) au lieu d'*avant*.
- `while` sans progression vers la sortie : **boucle infinie**.
- Croire que `continue` arrête la boucle : il ne fait que **sauter** au tour suivant.

Prochaine séance

Séance 4 — Fonctions, modules et robustesse

On sait décider et répéter. La Séance 4 apprend à **organiser** le code en **fonctions** réutilisables (`def`, paramètres, `return`), à importer des **modules**, à lire et écrire des **fichiers**, et à gérer les erreurs proprement avec `try/except`.

À faire d'ici là : refaire les cellules « À vous » du notebook de la séance.

Références

- Severance, C. — *Python for Everybody*, 2016.
- Downey, A. — *Think Python*, 2e éd., O'Reilly, 2015.
- Matthes, E. — *Python Crash Course*, 3e éd., No Starch Press, 2023.
- Python Software Foundation — *The Python Tutorial* (Control Flow), 2024.

Ressources de la séance

- **Notebook** · [TP 3 — Contrôle du flux](#) (14 Ko)