Programmation en Python - Séance 3 : Structures de Contrôle

Hervé TALE KALACHI

ENSPY – Université de Yaoundé 1

Structures de contrôle — définition & vue d'ensemble

Définition. Une structure de contrôle est une *construction du langage* qui **organise le flot d'exécution** d'un programme.

Trois familles en Python :

- **Séquence** : instructions exécutées dans l'ordre d'apparition.
- **Sélection** : exécuter un bloc selon une condition (`if`, `elif`, `else`).
- **Répétition** : répéter un bloc (`while`, `for`), avec `break`/`continue`.

Points clés en Python :

- **Blocs & indentation** : l'indentation (4 espaces) *définit* la portée des blocs.
- **Truthiness** : certaines valeurs sont `False` (`0`, `""`, `[]`, `{}`, `None`), les autres sont `True`.

Rappels syntaxiques

- Un en-tête se termine par `:` puis un **bloc indenté** (4 espaces).
- Comparaisons : `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Logique : `and`, `or`, `not`.
- **Attention** : `=` (affectation) $\neq$ `==` (comparaison).
- Python accepte `if ...:`, `if ... else ...`, `if ... elif ... else`.

Séquence d'instructions

Définition. Une **séquence d'instructions** est une suite d'ordres exécutés dans l'ordre d'apparition, de haut en bas, sans saut ni condition particulière.

Syntaxe générale :

```
instruction_1
instruction_2
instruction_3
# ...
instruction_n
```

Exemple concret :

```
print("1) initialisation")
x = 10
x = x + 5
print("2) x =", x)
print("3) fin")
```

Observation : les instructions s'exécutent dans l'ordre, comme un récit

Structure de sélection — syntaxe générale

Définition. Une **structure de sélection** (ou *conditionnelle*) permet d'exécuter une ou plusieurs instructions **seulement si une condition est vraie**.

```
if condition:
    bloc_si_vrai
# suite du programme
```

Avec alternative :

```
if condition:
    bloc_si_vrai
else:
    bloc_si_faux
# suite du programme
```

Rappel :

- Chaque en-tête se termine par `:`.
- Le bloc est défini par **4 espaces d'indentation**.
- Une seule des branches est exécutée.

Sélection : exemple concret

Objectif : afficher si un nombre est pair ou impair.

```
n = 7
if n % 2 == 0:
    print("pair")
else:
    print("impair")
```

Sélection : exemple concret

Objectif : afficher si un nombre est pair ou impair.

```
n = 7
if n % 2 == 0:
    print("pair")
else:
    print("impair")
```

Résultat :

```
impair
```

Explication : - Le test `n % 2 == 0` vérifie si le reste de la division par 2 est nul. - Si vrai $\rightarrow$ affiche "pair" ; sinon $\rightarrow$ affiche "impair".

À vous de jouer : modifiez le programme pour afficher un message supplémentaire : « Le nombre est divisible par 3 » si c'est le cas.

Structure de sélection multiple — syntaxe générale

Définition. Quand plusieurs conditions sont possibles, on utilise la structure `if ... elif ... else` pour **choisir un seul bloc** à exécuter.

Syntaxe générale :

```
if condition_1:
    bloc_1
elif condition_2:
    bloc_2
elif condition_3:
    bloc_3
else:
    bloc_par_defaut
# suite du programme
```

Règles importantes :

- Les conditions sont testées **dans l'ordre**.
- Dès qu'une condition est vraie, les suivantes sont ignorées.
- Le bloc `else` est optionnel.

Sélection multiple : exemple concret

```
note = 13.5
if note < 10:
    mention = "Recalé"
elif note < 12:
    mention = "Passable"
elif note < 14:
    mention = "Assez bien"
elif note < 16:
    mention = "Bien"
else:
    mention = "Très bien"

print("Mention :", mention)
```

À vous de jouer :

- Modifiez les seuils selon votre propre barème.
- Ajoutez un message d'encouragement pour les mentions « Bien » et « Très bien ».

Comparaison, logique, truthiness

```
x, y = 5, 12
print(x < y, x == 5, x != 0) # True True True
print((x < 10) and (y > 10)) # True
print((x < 0) or (y % 2 == 0))# True
print(not (x == 5)) # False

# Truthiness : "", 0, 0.0, [], {}, set(), None -> False; sinon
    True
if "Python": # non vide -> True
    print("Vrai")
```

Blocs et indentation (4 espaces)

```
# Correct
if True:
    print("ligne 1")
    print("ligne 2")

# Erreurs fréquentes :
# - Oublier le ':' en fin d'en-tête
# - Mélanger tabulations et espaces
# - Mauvaise indentation (bloc incohérent)
```

Conseil : configurez l'éditeur pour *insérer des espaces* à la place des tabulations.

Expression conditionnelle (ternaire)

```
age = 17
statut = "majeur" if age >= 18 else "mineur"
print(statut) # "mineur"
```

Usage : écriture compacte d'un `if/else` simple.

Boucle `while` — définition et syntaxe

Définition. La boucle `while` permet de **répéter un bloc d'instructions tant qu'une condition est vraie**. C'est une boucle dite **conditionnelle**.

Syntaxe générale :

```
while condition:
    bloc_d_instructions
# suite du programme
```

Fonctionnement :

- La condition est testée avant chaque itération.
- Si elle est `True` $\rightarrow$ le bloc s'exécute, puis on reteste.
- Si elle devient `False` $\rightarrow$ la boucle s'arrête.

Attention : la condition doit **évoluer à chaque tour**, sinon $\rightarrow$ *boucle infinie*.

Boucle `while` — exemple

Objectif : afficher les nombres de 1 à 5.

```
i = 1
while i <= 5:
    print(i, end=" ")
    i += 1
# 1 2 3 4 5
```

Explication :

- Initialisation : `i = 1`.
- Tant que `i <= 5` → on affiche puis on incrémente.
- Quand `i` devient 6 → la condition est fausse → arrêt.

À vous de jouer :

- Modifiez le programme pour afficher les nombres pairs de 2 à 10.
- Ajoutez un message « boucle terminée » après la fin du programme.

Point sur la creation de scripts

Définition. Un **script Python** est un fichier texte contenant des instructions Python (en général sauvegardé avec l'extension `.py`) que l'on peut exécuter d'un seul coup.

Différence avec le REPL :

- Dans le **REPL** (interpréteur interactif) : on tape et on voit le résultat ligne par ligne.
- Dans un **script** : on écrit tout le programme, on le sauvegarde, puis on l'exécute.

Avantages :

- On garde une trace de son code.
- On peut corriger, tester et réutiliser facilement.
- On peut créer de vrais programmes avec entrées, sorties et boucles.

Créer et exécuter un script Python

Étapes :

1. Ouvrir un éditeur de texte ou un IDE (`VS Code`, `Thonny`, `PyCharm`, ou `IDLE`).
2. Taper le code du programme.
3. Enregistrer le fichier avec l'extension `.py`, ex. `mon_programme.py`.
4. Exécuter dans un terminal :

```
python mon_programme.py
# ou
python3 mon_programme.py
```

Exemple minimal :

```
# fichier: bonjour.py
print("Bonjour, Python !")
```

Résultat à l'exécution :

```
Bonjour, Python !
```

Exemple de script complet

Objectif : calculer l'aire d'un cercle à partir du rayon saisi.

```
# fichier: aire_cercle.py
import math

r = float(input("Rayon du cercle : "))
aire = math.pi * r**2
print(f"L'aire du cercle de rayon {r} est {aire:.2f}")
```

Exécution dans le terminal :

```
$ python aire_cercle.py
Rayon du cercle : 5
L'aire du cercle de rayon 5.0 est 78.54
```

À vous de jouer : créez un script `perimetre_cercle.py` pour afficher le périmètre.

Boucle `for` — définition et syntaxe

Définition. La boucle `for` permet de **parcourir une séquence** (liste, chaîne, intervalle, etc.) et d'exécuter un bloc pour chaque élément.

Syntaxe générale :

```
for variable in sequence:
    bloc_d_instructions
# suite du programme
```

Fonctionnement :

- À chaque itération, la variable prend la valeur suivante de la séquence.
- Quand tous les éléments ont été parcourus, la boucle se termine.

Exemples de séquences courantes :

- `range(10)` → nombres de 0 à 9
- `[2, 4, 6, 8]` → liste de valeurs
- `"Python"` → caractères de la chaîne

Boucle `for` avec `range()`

Objectif : afficher les nombres de 1 à 5.

```
for i in range(1, 6):
    print(i, end=" ")
# Résultat : 1 2 3 4 5
```

Rappels utiles :

- `range(début, fin)` → s'arrête avant `fin`.
- `range(5)` équivaut à `range(0, 5)`.
- `range(début, fin, pas)` permet d'incrémenter autrement.

Exemples :

```
range(0, 10, 2) # 0, 2, 4, 6, 8
range(5, 0, -1) # 5, 4, 3, 2, 1
```

Boucle `for` sur des séquences

Exemples de parcours :

```
# Parcourir une liste
fruits = ["pomme", "banane", "kiwi"]
for fruit in fruits:
    print(fruit)

# Parcourir une chaîne
for lettre in "Python":
    print(lettre, end="-")
# P-y-t-h-o-n-
```

Observation : le `for` "lit" directement les éléments sans gestion d'indice explicite.

enumerate() et zip()

Exemples pratiques :

```
# Avec enumerate : obtenir les indices
fruits = ["pomme", "banane", "kiwi"]
for i, fruit in enumerate(fruits, start=1):
    print(i, fruit)
# 1 pomme
# 2 banane
# 3 kiwi

# Avec zip : parcourir deux listes en parallèle
noms = ["Ada", "Alan", "Grace"]
notes = [18, 15, 17]
for nom, note in zip(noms, notes):
    print(f"{nom} a eu {note}/20")
```

Intérêt : simplifie le code et évite les erreurs d'indice.

À vous de jouer — exercices sur la boucle `for`

```
# A) Afficher les carrés des entiers de 1 à10.

# B) Parcourir la chaîne "PYTHON" et afficher chaque
# lettre en minuscule sur la même ligne.

# C) Pour la liste nombres = [3, 8, 12, 5, 7],
# afficher seulement les nombres pairs.

# D) Créer deux listes : noms et ages.
# Parcourir avec zip et afficher :
# "Alice a 20 ans", "Bob a 25 ans", etc.

# E) Calculer la somme des multiples de 3 entre 1 et 100.
```

Astuce : utilisez `range()`, `if` et `zip()` si besoin.

break et continue

```
# break : sortir de la boucle
for k in range(10):
    if k == 5:
        break
    print(k, end=" ") # 0 1 2 3 4

print()

# continue : passer àl'itération suivante
for k in range(6):
    if k % 2 == 0:
        continue
    print(k, end=" ") # 1 3 5
```

Clause else sur les boucles

```
# L'else s'exécute si la boucle n'a pas été interrompue par break
for n in range(2, 10):
    if n % 7 == 0:
        print("trouvé:", n)
        break
else:
    print("aucun multiple de 7 dans 2..9")
```

Idée : else signale une *fin "normale"* de boucle (pas de break).

À vous de jouer — mini-exercices

```
# A) Conditionnels
# Demander un entier n, afficher "pair"/"impair" et "positif/nul/
    négatif".

# B) While : table géométrique
# Afficher 10 termes: partir de 1, multiplier par 2 à~chaque ité
    ration.

# C) For + accumulateur
# Somme des multiples de 3 dans 1..100 (inclus).
# D) Enumerate + formatage
# Pour la liste ["pomme","banane","kiwi"], afficher:
# 1) pomme
# 2) banane
# 3) kiwi

# E) Recherche + break/else
# Vérifier si un nombre premier p est dans une liste L.
# Si trouvé -> "trouvé àl'indice i", sinon -> "absent".
```