Structures de contrôle

I. Introduction

Définition 1 : *Structure de contrôle*

Une structure de contrôle est un type d'instruction permettant de modifier le cours normal d'un programme en modifiant l'ordre dans lequel les instructions sont exécutées ou en sautant certaines instructions.
Nous allons dans ce chapitre étudier deux types de structures de contrôle : les instructions conditionnelles (`if`, `elif`, `else`) et les instructions itératives ou boucles (`for`, `while`).
Une troisième structure de contrôle, les fonctions (`def`), fera l'objet du chapitre suivant.

Indentation

Dans toutes les syntaxes des structures de contrôle, on retrouve le même schéma :

```
[mot clé] [partie variable]:
    [bloc d'instructions]
```

- ⋆ Le *mot clé* est celui qui définit le type de structure : `if`, `elif`, `else`, `for`, `while` ou `def`.
- ⋆ Ce qu'on doit mettre dans la *partie variable* dépend du type de structure. Cette ligne doit **toujours** se terminer par le symbole `:`.
- ⋆ Enfin le contenu de la structure de contrôle se trouve dans le *bloc d'instructions*. Celui-ci doit être **obligatoirement** indenté par rapport à la première ligne c'est-à-dire décalé sur la droite de 4 espaces. Ce décalage s'appelle l'indentation : il permet à Python de savoir quelles sont les instructions qui sont dans la structure de contrôle et quelles sont celles qui n'y sont pas. Pour sortir du bloc, on revient au même niveau d'indentation que le mot clé.

II. L'instruction conditionnelle

Définition 2 : *Instruction conditionnelle*

Une instruction conditionnelle permet à un programme d'exécuter une ou plusieurs instructions uniquement si une certaine condition est vraie. Cela permet donc d'adapter le comportement du programme selon les cas.

1) Différentes syntaxes possibles

Syntaxes de l'instruction conditionnelle

⋆ **Structure de base** : une instruction conditionnelle commence par le mot-clé `if`, suivi d'une **expression booléenne**, se terminant par un deux-points `:`, puis d'un bloc indenté :

```
if condition:
    bloc_d'instructions_si_condition_vraie
```

⋆ **Si on souhaite exécuter un autre bloc si la condition est fausse**, on ajoute un `else` :

```
if condition:
    bloc_d'instructions_si_condition_vraie
else:
    bloc_d'instructions_si_condition_fausse
```

⋆ **Si on a plusieurs cas à distinguer**, on peut insérer des blocs intermédiaires avec `elif` :

```
if condition1:
    bloc_d'instructions1
elif condition2:
    bloc_d'instructions2
elif condition3:
    bloc_d'instructions3
else:
    bloc_d'instructions_par_defaut
```

Exemple :

⋆
```
if a > 0:
    print("a est positif")
```

⋆
```
if a > 5:
    print("a est supérieur à 5")
    print("Cette ligne fait partie du bloc conditionnel")
print("Ceci est hors du bloc conditionnel")
```

⋆
```
if a > 5:
    print("a est supérieur à 5")
else:
    print("a est inférieur ou égal à 5")
```

⋆
```
if a > 15:
    print("a est supérieur à 15")
elif a > 5:
    print("a est supérieur à 5 mais inférieur ou égal à 15")
else:
    print("a est inférieur ou égal à 5")
```

Remarque :

⋆ `else` se traduit par « sinon ». `else` est **toujours** facultatif : un principe général à garder en tête est que si on n'a rien à demander à Python, alors il ne faut rien lui dire ! Exemple de chose à ne pas faire lorsqu'on veut augmenter `a` de `1` uniquement lorsque `a` est négatif :

```
if a < 0:
    a += 1
else:
    a = a       # Ligne complètement inutile !
```

⋆ `elif` est l'abbréviation de `else if` (« sinon si »). On peut écrire autant de blocs `elif` que nécessaire. Dès qu'une condition est vraie, toutes les conditions qui suivent sont ignorées : ainsi un seul bloc d'instructions au maximum est exécuté (un bloc est forcément exécuté s'il y a un `else`).

Ne jamais mettre de condition après un `else` !

2) Plusieurs `if` à la suite ou un bloc `if`–`elif` ?

Considérons les deux programmes suivants :

Programme 1 :

```
if x < 0:
    x += 1
elif x == 0:
    x += 2
else:
    x += 3
```

Programme 2 :

```
if x < 0:
    x += 1
if x == 0:
    x += 2
if x > 0:
    x += 3
```

Que se passe-t-il si `x = -1` dans chacun des deux programmes ?

⋆ Pour le programme 1 c'est facile : puisqu'il n'y a qu'un seul bloc `if`, on est sûr qu'une seule instruction sera exécutée. Ici `x < 0` donc c'est la première qui le sera et à la fin du programme, `x` sera égal à `0`.

⋆ Pour le programme 2 c'est beaucoup plus compliqué et il faut suivre le programme ligne par ligne pour être sûr du résultat.

- Dans le premier bloc `if`, la condition est vraie donc `x = 0` après ce bloc.
- Dans le deuxième bloc `if`, la condition est fausse avec la valeur initiale de `x` mais à ce moment du programme, `x` a déjà été modifiée et vaut maintenant `0` donc la condition est vraie et `x = 2` après ce bloc.
- De même, dans le dernier bloc `if`, la condition est vraie à ce moment du programme donc `x = 5` après ce bloc.

Finalement, `x = 5` à la fin du programme 2.

Bonne pratique

Enchaîner des blocs `if` est une **mauvaise pratique** car cela rend le programme plus compliqué à comprendre car plusieurs instructions peuvent être exécutées et il peut y avoir des modifications de variables « en cascade ».

3) Imbrication de blocs conditionnels

Il est possible d'imbriquer des instructions conditionnelles pour tester plusieurs critères :

```
if x > 0:
    if y > 0:
        print("x et y sont positifs")
    else:
        print("x est positif et y est négatif ou nul")
else:
    if y > 0:
        print("x est négatif ou nul et y est positif")
    else:
        print("x et y sont négatifs ou nuls")
```

mais en général, on préfèrera tout regrouper en un seul bloc dans la mesure du possible :

```
if x > 0 and y > 0:
    print("x et y sont positifs")
elif x > 0 and y <= 0:
        print("x est positif et y est négatif ou nul")
elif y > 0:     # Il n'est pas nécessaire d'ajouter x <= 0, pourquoi ?
    print("x est négatif ou nul et y est positif")
else:
    print("x et y sont négatifs ou nuls")
```

> Remarque : Chaque nouveau niveau de bloc doit être **indenté** d'un niveau supplémentaire. L'indentation indique clairement à Python la structure logique du programme.

4) Tests de divisibilité

a) Pourquoi la division ne permet pas de tester la divisibilité ?

En Python, la division avec `/` renvoie **toujours** un `float`, même si les nombres sont divisibles.

```
a = 10
b = 5
print(a / b)  # Renvoie 2.0 donc un float
```

Ainsi le test suivant ne fonctionnera pas :

```
if type(a / b) == int:
    print(f"{a} est divisible par {b}")
```

Inversement, la division entière avec `//` renvoie **toujours** un `int`, même si les nombres ne sont pas divisibles.

```
a = 10
b = 6
print(a // b)  # Renvoie 1 donc un int
```

Ainsi le test suivant ne fonctionnera pas non plus :

```
if type(a // b) == int:
    print(f"{a} est divisible par {b}")
```

b) Division euclidienne

Pour tester la divisibilité, il faut utiliser l'opérateur `%` qui calcule le reste de la division euclidienne :

`a` est divisible par `b` si et seulement si `a % b == 0`

Exemple :

⋆
```python
if a % b == 0:
    print(f"{a} est divisible par {b}")
else:
    print(f"{a} n'est pas divisible par {b}")
```

⋆
```python
if n % 3 == 0 and n % 5 == 0:
    print(f"{n} est un multiple de 3 et de 5")
elif n % 3 == 0:
    print(f"{n} est un multiple de 3")
elif n % 5 == 0:
    print(f"{n} est un est un multiple de 5")
else:
    print(f"{n} n'est ni un multiple de 3, ni de 5")
```

III. Instructions itératives ou boucles

Définition 3 : *Instruction itérative*

Une instruction itérative, ou boucle, permet de répéter automatiquement un bloc d'instructions un certain nombre de fois.
Python propose deux types de boucles : la boucle `for` et la boucle `while`.

1) La boucle `for`

a) Syntaxes de la boucle `for`

La boucle `for` permet d'exécuter un bloc d'instructions un nombre connu de fois, en parcourant un objet itérable (comme une chaîne de caractères, une liste, ou une séquence de nombres générée par `range`).

Syntaxes de la boucle `for`

```
for variable in objet_iterable:
    bloc_d'instructions
```

La plupart du temps, on utilise la fonction `range` pour créer un objet itérable :

```
for compteur in range(n):
    bloc_d'instructions
```

- ⋆ Il y a exactement `n` itérations : le bloc d'instructions est exécuté `n` fois.
- ⋆ `compteur` est une variable créée par la boucle qui est incrémentée automatiquement à chaque itération.
- ⋆ `compteur` commence à la valeur `0` et termine à la valeur `n-1`.

```
for compteur in range(a, b):
    bloc_d'instructions
```

- ⋆ Il y a exactement `b-a` itérations : le bloc d'instructions est exécuté `b-a` fois.
- ⋆ `compteur` est une variable créée par la boucle qui est incrémentée automatiquement à chaque itération.
- ⋆ `compteur` commence à la valeur `a` et termine à la valeur `b-1`.

Exemple :

⋆ **Code :**

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

Résultat :

```
0
1
2
3
4
```

⋆ **Code :**

```
for i in range(3, 6):
    print(i)
```

Résultat :

```
3
4
5
```

⋆ **Code :**

```
x = 3
for i in range(4):
    x += 2
    print(x)
```

Résultat :

```
5
7
9
11
```

⋆ **Code :**

```
x = 3
for i in range(4):
    x += i
print(x)
```

Résultat :

```
9
```

⋆ **Code :**

```
for i in [2, 3, 5, 7, 11]:
    print(f"{i} est un nombre
   premier")
```

Résultat :

```
2 est un nombre premier
3 est un nombre premier
5 est un nombre premier
7 est un nombre premier
11 est un nombre premier
```

Remarque :

- ⋆ `range(n)` génère la séquence d'entiers `0, 1, 2, ..., n-1` : le nombre `n` est exclu mais il y a bien exactement `n` nombres puisqu'on commence à `0`.
- ⋆ `range(a, b)` génère la séquence d'entiers `a, a + 1, a + 2, ..., b-1` : le nombre `b` est exclu mais il y a bien exactement `b-a`.
- ⋆ Le compteur peut être utilisé ou non dans le bloc d'instructions.

b) Dérouler une boucle `for`

Dérouler une boucle `for`

Pour mieux comprendre ce qu'il se passe lors du déroulement d'une boucle `for`, on peut construire un tableau permettant d'étudier l'état des variables à chaque itération de la boucle :

- ⋆ L'étape 0 est l'état des variables avant la boucle.
- ⋆ L'étape 1 est l'état des variables après la première exécution des instructions indentées.
- ⋆ L'étape 2 est l'état des variables après la deuxième exécution des instructions indentées.
- ⋆ Etc.

Ainsi, nous pouvons voir très rapidement si tout se passe comme prévu dès les premières itérations.

Cela permet également d'identifier un invariant de boucle c'est-à-dire une propriété qui est vraie avant et après chaque itération. Il permet donc de deviner l'état des variables à la dernière étape de la boucle et donc de déterminer le nombre d'itérations nécessaires pour atteindre un objectif.

Exemple :

- ⋆ Considérons le programme suivant :

```
u = 2
for i in range(20):
    u = u * u
```

Quelle est la valeur de `u` à la fin du programme ?

Étape	`i`	`u`
0		2
1	0	$2 \times 2 = 2^2 = 2^{2^1}$
2	1	$2^2 \times 2^2 = 2^4 = 2^{2^2}$
3	2	$2^4 \times 2^4 = 2^8 = 2^{2^3}$
⋮	⋮	⋮
20	19	$2^{2^{20}}$

Les nombres en rouge montrent l'identification d'un invariant de boucle : « Après l'étape k la valeur de `u` est 2^{2^k} ». On devine alors qu'à la fin du programme, la valeur de `u` est $2^{2^{20}}$ (et non 2^{20} comme on pourrait le croire).

⋆ Pour calculer 2^{20}, il faudrait initialiser `u = 2` puis exécuter `u = u * 2` jusqu'à ce que `u` atteigne la valeur 2^{20}.

```
u = 2
for i in range(?):
    u = u * 2
```

Que doit-on mettre dans le range ?

Étape	`i`	`u`
0		2
1	0	$2 \times 2 = 2^2 = 2^{1+1}$
2	1	$2^2 \times 2 = 2^3 = 2^{1+2}$
3	2	$2^3 \times 2 = 2^4 = 2^{1+3}$
⋮	⋮	⋮
19	18	$2^{20} = 2^{1+19}$

Grâce à l'invariant de boucle, on devine que la dernière étape doit porter le numéro 19 donc le programme doit être

```
u = 2
for i in range(19):
    u = u * 2
```

c) Problèmes classiques

Voici une liste de problèmes classiques qui nécessitent d'utiliser une boucle `for` :

⋆ **Calculer une somme :** $\displaystyle\sum_{k=a}^{n} u_k$

⋆ **Calculer un produit :** $\displaystyle\prod_{k=a}^{n} u_k$

⋆ **Calculer le n^{e} terme d'une suite récurrente simple**

⋆ **Calculer le n^{e} terme d'une suite récurrente double**

Calculer une somme

Pour calculer la somme $\displaystyle\sum_{k=a}^{b} u_k = u_a + u_{a+1} + \cdots + u_b$:

⋆ on initialise `S` à `0` (**on initialise toujours une somme à** `0`)

⋆ on fait une boucle `for k in range(a, b+1)` afin que `k` varie de `a` à `b+1-1 = b`

Exemple : Pour calculer la somme $\displaystyle\sum_{k=10}^{100} \frac{k+1}{k^2}$:

```
S = 0
for k in range(10, 101):
    S = S + (k+1) / (k**2)  # ou bien S += (k+1) / (k**2)
print(S)
```

Calculer un produit

Pour calculer le produit $\prod_{k=a}^{b} u_k = u_a \times u_{a+1} \times \cdots \times u_b$:

- ⋆ on initialise `P` à `1` (**on initialise toujours un produit à** `1`)
- ⋆ on fait une boucle `for k in range(a, b+1)` afin que `k` varie de `a` à `b+1-1 = b`

Exemple : Pour calculer la somme $\prod_{k=10}^{100} \frac{k+1}{k^2}$:

```
P = 1
for k in range(10, 101):
    P = P * (k+1) / (k**2)  # ou bien P *= (k+1) / (k**2)
print(P)
```

2) La boucle `while`

La boucle `while` permet d'exécuter un bloc tant qu'une condition est vraie. On ne sait donc pas forcément à l'avance combien de fois le bloc sera exécuté.

Syntaxe de la boucle `while`

```
while condition:
    bloc_d'instructions
```

Remarque :

- ⋆ C'est la même syntaxe que celle de l'instruction conditionnelle `if` (qui n'est pas une boucle) mais le comportement est différent : si la `condition` est réalisée, le bloc d'instructions est exécuté puis on teste de nouveau la `condition`. Si elle est encore réalisée, on exécute de nouveau le bloc d'instructions et ainsi de suite jusqu'à ce que la `condition` ne soit plus réalisée.
- ⋆ `while condition` se traduit par « Tant que la condition est vérifiée, faire ».

Exemple :

- ⋆ **Code :**

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

Résultat :

```
1
2
3
4
5
```

⋆ **Code :**

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

Résultat :

```
2
3
4
5
6
```

⋆ **Code :**

```
u = 2
while u < 1000:
    u *= 2
print(u)
```

Résultat :

```
1024
```

Avec une boucle `while`, il y a un risque de créer une boucle infinie lorsque la condition est éternellement vraie. Un cas fréquent d'erreur est d'oublier de modifier la variable concernée par la condition :

```
i = 1
while i < 1000:
    print(i)
```

Une boucle infinie ne donne pas d'erreur : l'ordinateur exécute les instructions sans s'arrêter et on ne peut plus rien demander à Python. Il faut interrompre manuellement Python en appuyant sur l'icône « Interrompre » de Pyzo (en jaune) ou en tapant `CTRL + I`.

Exemple : La boucle `while` est très utile dans la programmation de « petits jeux ». Par exemple pour faire deviner un mot à l'utilisateur :

```
mot_secret = "python"
mot_utilisateur = ""

while mot_utilisateur != mot_secret:
    mot_utilisateur = input("Devine le mot : ")

print("Bravo !")
```

3) Comparaison entre `for` et `while`

Quel type de boucle utiliser ?

Il est **toujours** possible de remplacer une boucle `for` par une boucle `while` : il suffit de gérer nous même le `compteur` en l'initialisant (avant la boucle) et en l'incrémentant (dans la boucle). Ainsi les deux boucles suivantes sont équivalentes :

```
for i in range(a, b):
    bloc_d'instructions

i = a
while i < b:
    bloc_d'instructions
    i += 1          # Si cette ligne est oubliée, la boucle est infinie
```

Inversement, certaines boucles `while` ne sont pas remplaçables par des boucles `for`. On pourrait croire alors qu'il suffit d'apprendre la boucle `while` mais en pratique on préfèrera utiliser :

- ⋆ la boucle `for` **lorsque l'on sait a priori combien de fois les instructions doivent être exécutées** ;
- ⋆ la boucle `while` **lorsque l'on ne sait pas a priori combien de fois les instructions doivent être exécutées**, typiquement lorsque les instructions doivent être exécutées jusqu'à ce que quelque chose se passe.

Exemple : Voici un exemple de situation où la boucle `while` est indispensable. Déterminons le plus petit entier N tel que $1 \times 2 \times 3 \times \cdots \times n \geqslant 10^9$:

```
u = 1
N = 1
while u < 10**9:
    N += 1
    u = u * N
print(f"Le plus petit entier N tel que 1 x 2 x ... x N >= 10^9 est {N}.")
```

4) Complément : deux mots-clés qui peuvent être utiles

Ces deux instructions peuvent être utilisées aussi bien dans une boucle `for` que dans une boucle `while` :

- ⋆ `break` : interrompt immédiatement la boucle.
- ⋆ `continue` : passe immédiatement à l'itération suivante.

Exemple :

⋆ **Code :**

```
for i in range(1, 11):
    if i % 5 == 0:
        break
    print(i)
```

Résultat :

```
1
2
3
4
```

⋆ **Code :**

```
for i in range(1, 11):
    if i % 5 == 0:
        continue
    print(i)
```

Résultat :

```
1
2
3
4
6
7
8
9
```