Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG
DUT- IDIA 1 - S2
M124 – Programmation en Python
Année universitaire: 2024-2025

Support de cours

Programmation en Python

Chapitre II: Structures de contrôle et les fonctions

Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- LES STRUCTURES CONDITIONNELLES
- LES BOUCLES ET LES ITÉRATIONS
- LES FONCTIONS
- LES FONCTIONS RÉCURSIVES
- LA PORTÉE DES VARIABLES
- EXERCICES

Les structures conditionnelles

Les structures conditionnelles permettent d'exécuter différentes instructions en fonction d'une condition. Python utilise les mots-clés if, elif et else pour gérer les décisions.

❑ Syntaxe de `if, elif, else`

```
if condition:
    # Bloc exécuté si la condition est vraie
elif autre_condition:
    # Bloc exécuté si l'autre condition est vraie
else:
    # Bloc exécuté si aucune condition n'est vraie
```

Exemple

```
age = 24

if age < 18:
    print("Vous êtes mineur.")
elif age == 18:
    print("Vous venez d'atteindre la majorité !")
else:
    print("Vous êtes majeur.")
```

Sortie (Résultat)

```
Vous êtes majeur.
```

Les structures conditionnelles

→ Opérateurs de comparaison et logiques

❑ Opérateurs de comparaison

Opérateur	Signification	Exemple (a = 5, b = 10)	Résultat
==	Égalité	a == b	False
!=	Différent	a != b	True
<	Inférieur	a < b	True
>	Supérieur	a > b	False
<=	Inférieur ou égal	a <= 5	True
>=	Supérieur ou égal	b >= 10	True

❑ Opérateurs logiques

Opérateur	Signification	Exemple
and	Toutes les conditions doivent être vraies	age >= 18 and age < 65
or	Au moins une condition doit être vraie	age < 18 or age > 65
not	Inverse une condition	not (age >= 18)

→ Expressions conditionnelles (ternary operator)

Python permet d'écrire des **conditions en une seule ligne** grâce aux expressions conditionnelles.

Syntaxe

valeur_si_vrai **`if`** *condition* **`else`** *valeur_si_faux*

Exemple

```
age = 24
statut = "Majeur" if age >= 18 else "Mineur"
print(f"Vous êtes {statut}.")
```

```
Vous êtes Majeur.
```

→ Match - case

En Python, avant la version 3.10, il n'existait pas d'équivalent direct du ***switch-case*** que l'on trouve dans d'autres langages comme C, Java ou JavaScript. On devait utiliser if-elif-else ou un dictionnaire pour imiter ce comportement.

Avec Python 3.10, la structure match-case a été introduite, offrant une syntaxe plus lisible et plus puissante pour comparer des valeurs.

Syntaxe :

```
match variable:
    case valeur1:
        # Code à exécuter si variable == valeur1
    case valeur2:
        # Code à exécuter si variable == valeur2
    case _:
        # Code exécuté si aucune des valeurs ne correspond
```

Le `_` sert de cas *par défaut*, comme le *default* en switch-case.

→ Match - case

Exemple

```
def verifier_jour(jour):
    match jour:
        case "Lundi":
            print("Début de la semaine !")
        case "Vendredi":
            print("Le week-end approche !")
        case "Dimanche":
            print("Jour de repos.")
        case _:
            print("Juste un autre jour !")

verifier_jour("Vendredi")
```

```
Le week-end approche !
```

→ **Match case**

Exemple

On peut regrouper plusieurs valeurs pour un même case :

```
def verifier_jour(jour):
    match jour:
        case "Samedi" | "Dimanche":
            print("C'est le week-end !")
        case "Lundi":
            print("Début de la semaine.")
        case _:
            print("Jour ordinaire.")

# Exemple d'utilisation
verifier_jour("Samedi")
verifier_jour("Lundi")
verifier_jour("Mercredi")
```

```
C'est le week-end !
Début de la semaine.
Jour ordinaire.
```

Le | sert d'opérateur "**ou**" pour vérifier plusieurs valeurs.

Les boucles et les itérations

→ La boucle `for`

La boucle **`for`** permet de répéter une action un nombre défini de fois.

Syntaxe

`for` *variable* **`in range(`***début*, *fin*, *pas***`):**

 # Instructions répétées

***inclut** début mais **exclut** fin*

Exemples

```
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

```
for i in range(1, 6,2):
    print(i)
```

```
1
3
5
```

```
# Compter de 0 à 9 (sans spécifier de début)
# Par défaut, commence à 0
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

```
# Compter de 10 à 1 en décrémentant
for i in range(10, 0, -1):
    print(i)
```

```
10
9
8
7
6
5
4
3
2
1
```

Les boucles et les itérations

→ La boucle `While`

La boucle **`while`** exécute du code tant qu'une condition est vraie.

Syntaxe

```
while condition:
    # Instructions répétées
```

Exemples

```
#Compter jusqu'à 5 avec while :
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
1
2
3
4
5
```

```
#Compter à rebours de 5 à 1 :
x = 5
while x > 0:
    print(x)
    x -= 1
```

```
5
4
3
2
1
```

Les boucles et les itérations

→ Instructions de contrôle (`break, continue, pass`)

- **`break`** : Sortir immédiatement de la boucle
- **`continue`** : Passer à l'itération suivante sans exécuter le reste du bloc
- **`pass`** : Ne rien faire (utile pour les blocs en attente d'implémentation)

Exemple

```
for i in range(10):
    if i == 5:
        break  # Arrête la boucle dès que i == 5
    print(i)
```

```
0
1
2
3
4
```

```
for i in range(5):
    if i == 2:
        continue  # Ignore 2 et passe au suivant
    print(i)
```

```
0
1
3
4
```

```
for i in range(3):
    pass  # Placeholder pour une future implémentation
```

→ Les boucles Imbriquées

Une boucle imbriquée est une boucle dans une autre boucle.

Exemple

→ Table de multiplication

```
for i in range(1, 4):  # 1 à 3
    for j in range(1, 4):  # 1 à 3
        print(f"{i} x {j} = {i * j}")
```

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

→ Définition et appel de fonctions

❑ Définition d'une fonction :

Une fonction est définie avec le mot-clé **`def`**, suivie d'un nom et de parenthèses contenant les **paramètres**.

Exemple

```
def dire_bonjour():
    print("Bonjour !")

# Appel de la fonction
dire_bonjour()
```

```
Bonjour !
```

```
#Fonction avec paramètres
def saluer(nom):
    print(f"Bonjour, {nom} !")

saluer("Ilham")
```

```
Bonjour, Ilham !
```

→ Paramètres et arguments

❑ Paramètres positionnels (positional arguments)

Les arguments sont passés dans l'ordre de la définition de la fonction.

Exemple

```
def additionner(a, b):
    print(a + b)

additionner(3, 5)  # a = 3, b = 5
```

```
8
```

❑ Paramètres nommés (keyword arguments)

On peut préciser le nom des paramètres lors de l'appel.

Exemple

```
def afficher_infos(nom, age):
    print(f"Nom : {nom}, Âge : {age}")

afficher_infos(age=20, nom="Sara")  # L'ordre n'a plus d'importance
```

```
Nom : Sara, Âge : 20
```

→ Paramètres et arguments

❑ Paramètres avec valeurs par défaut (default arguments)

Si aucun argument n'est fourni, la valeur par défaut est utilisée.

Exemple

```
def presenter(nom="Invité"):
    print(f"Bienvenue, {nom} !")

presenter()          # Utilise la valeur par défaut
presenter("Imane")    # Remplace la valeur par défaut
```

```
Bienvenue, Invité !
Bienvenue, Imane !
```

→ Paramètres et arguments

❑ Paramètres variables (*args, **kwargs)

- `*args` permet de passer un nombre variable d'arguments positionnels.
- `**kwargs` permet de passer un nombre variable d'arguments nommés.

Exemple

```
def additionner_tout(*nombres):
    total = sum(nombres)
    print(f"Somme : {total}")

additionner_tout(1, 2, 3, 4)  # Passe plusieurs nombres
```

```
Somme : 10
```

```
def afficher_details(**infos):
    for cle, valeur in infos.items():
        print(f"{cle} : {valeur}")

afficher_details(nom="Ilham", age=20, ville="Béni Mellal")
```

```
nom : Ilham
age : 20
ville : Béni Mellal
```

→ Retour de valeurs avec `return`

Une fonction peut renvoyer un résultat au lieu de simplement afficher quelque chose..

Exemple

```
def multiplier(a, b):
    return a * b  # Retourne le résultat

resultat = multiplier(4, 5)
print(resultat)  # Utilisation du résultat
```

```
20
```

→ Si une fonction ne contient pas return, elle renvoie None par défaut.

Les fonctions récursives

- Une fonction récursive est une fonction qui s'appelle elle-même pour résoudre un problème en le décomposant en sous-problèmes plus petits.

- Chaque appel récursif est stocké dans une **pile d'appels**, une structure de données qui garde en mémoire les fonctions en attente d'exécution.

→ Une fonction récursive doit respecter deux règles essentielles :

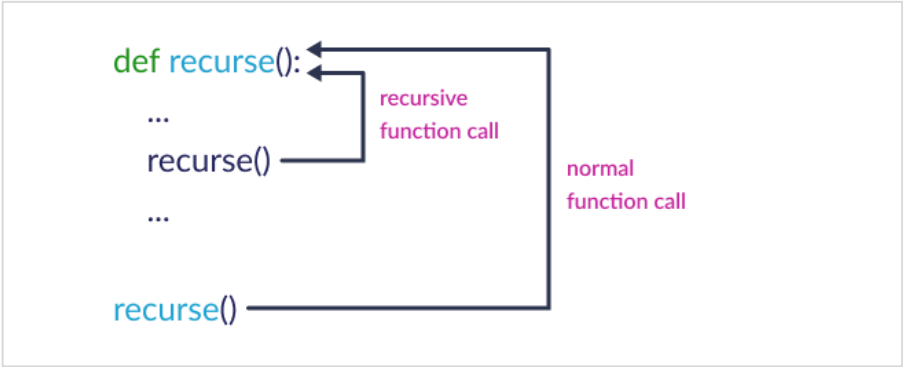
  - Un cas de base qui met fin à la récursion.

  - Un appel récursif qui rapproche la solution du cas de base.

💡 ***Sans un cas de base, la fonction s'exécutera indéfiniment et provoquera une erreur (RecursionError).***

- Lorsqu'une fonction récursive atteint son **cas de base**, Python commence à dépiler les appels dans l'ordre inverse pour résoudre le problème.

Les fonctions récursives



Exemple : Calcul de la Factorielle

La factorielle d'un nombre n est définie comme : $n!=n\times(n-1)!$

Avec le cas de base : $0!=1$

```
def factorielle(n):
    if n == 0:  # Cas de base
        return 1
    return n * factorielle(n - 1)  # Appel récursif

print(factorielle(5))
```

```
120
```

→ Récursivité vs Itération

La récursivité peut être remplacée par une boucle pour économiser de la mémoire et éviter l'erreur ***RecursionError***.

Exemple : **Version itérative de la factorielle**

```
def factorielle_iterative(n):
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat

print(factorielle_iterative(5))
```

```
120
```

→ L'itération est souvent plus efficace car elle n'utilise pas la pile d'appels récursifs.

Les fonctions récursives

→ Quand utiliser la récursivité ?

- **Cas où la récursivité est utile :**
  - Problèmes naturellement récursifs (ex : factorielle, Fibonacci…).
  - Structures récursives comme les arbres et les graphes.
  - Exploration de dossiers (ex : parcourir tous les fichiers d'un répertoire).
- **Cas où éviter la récursivité :**
  - Si une solution itérative est plus simple et efficace.
  - Si le langage a une limite de récursion (ex : Python a une limite d'environ 1000 appels récursifs).

La portée des variables

- **Portée locale** : Une variable déclarée dans une fonction n'est accessible que dans cette fonction.
- **Portée globale:** Une variable déclarée hors d'une fonction est accessible partout.

Exemple

→ **Modifier une variable globale avec `global`**

```
z = 5
def modifier_global():
    global z  # Modifier la variable globale
    z = 10

modifier_global()
print(z)
```

```
10
```

La portée des variables

- **Portée nonlocale** : Permet de modifier une variable d'une fonction englobante.

Exemple

```
def fonction_externe():
    message = "Bonjour"  # Variable définie dans la fonction externe
    print("Avant modification dans fonction_externe :", message)

    def fonction_interne():
        nonlocal message  # On veut modifier la variable de la fonction externe
        message = "Salut"  # Modification de la variable
        print("Après modification dans fonction_interne :", message)

    fonction_interne()  # Appel de la fonction interne

fonction_externe()
```

```
Avant modification dans fonction_externe : Bonjour
Après modification dans fonction_interne : Salut
```

En Python, une fonction interne (ou fonction imbriquée) est définie à l'intérieur d'une autre fonction. Cela signifie que :

- La fonction interne est incluse dans la fonction externe → Elle ne peut pas être appelée directement de l'extérieur.
- La fonction interne peut accéder aux variables de la fonction externe → Mais pour les modifier, on doit utiliser ***nonlocal***.

Exercices

Exercice 1

Crée une fonction est ***pair(nombre)*** qui prend un entier en paramètre et retourne "Pair" s'il est pair et "Impair" sinon.

Exercice 2

Crée un programme qui :

- Permet à l'utilisateur d'entrer plusieurs produits (nom, prix unitaire, quantité).
- Calcule le prix total de chaque produit.
- Affiche un menu récapitulant tous les produits avec leur prix.
- Permet à l'utilisateur de choisir s'il veut ajouter un autre produit ou terminer l'entrée.