**Fiche Mémo Python**

## Fonctions, Scope, Listes, \*args/\*\*kwargs, Lambda & Dictionnaires

### 1. Utilisation des fonctions

#### Définition d'une fonction :

Utilisez le mot-clé def pour déclarer une fonction.

def ma\_fonction(param1, param2):

# instructions

return résultat

#### Appel de fonction (Call) :

Appelez une fonction en utilisant son nom suivi de parenthèses et des arguments nécessaires.

résultat = ma\_fonction(10, 20)

#### Arguments nommés (Keyword arguments) :

Permettent de préciser quels arguments correspondent à quels paramètres.

ma\_fonction(param2=20, param1=10)

#### Valeurs par défaut (Default values) :

Attribuez une valeur par défaut à un paramètre dans la définition de la fonction.

def ma\_fonction(param1, param2=5):

return param1 + param2

**2. Scope dans les fonctions**

#### Scope local :

Les variables définies à l’intérieur d’une fonction ne sont accessibles qu’à l’intérieur de celle-ci.

#### Scope global :

Les variables définies en dehors des fonctions sont accessibles globalement.

#### Déclaration explicite :

Utilisez le mot-clé global pour modifier une variable globale depuis une fonction.

compteur = 0

def incremente():

global compteur

compteur += 1

**3. Le mot-clé return**

#### Retourner une valeur :

return met fin à l’exécution de la fonction et renvoie une valeur.

def somme(a, b):

return a + b

#### Retour multiple :

Vous pouvez renvoyer plusieurs valeurs sous forme de tuple.

def operations(a, b):

return a + b, a \* b

**4. Passing list as function arguments & Modifying a list in a function**

#### Passage d'une liste :

Les listes sont passées par référence, ce qui signifie que la fonction peut modifier la liste originale.

def ajoute\_element(liste, element):

liste.append(element)

ma\_liste = [1, 2, 3]

ajoute\_element(ma\_liste, 4)

# ma\_liste devient [1, 2, 3, 4]

#### Reprendre une nouvelle liste :

Si vous souhaitez éviter la modification de l’originale, faites-en une copie.

def ajoute\_element\_copie(liste, element):

nouvelle\_liste = liste.copy()

nouvelle\_liste.append(element)

return nouvelle\_liste

**5. \*args, \*\*kwargs et paramètres variadiques**

#### \*args :

Permet de passer un nombre variable d’arguments positionnels à une fonction.

def addition(\*args):

return sum(args)

print(addition(1, 2, 3)) # renvoie 6

#### \*\*kwargs :

Permet de passer un nombre variable d’arguments nommés.

def afficher\_info(\*\*kwargs):

for cle, valeur in kwargs.items():

print(f"{cle} : {valeur}")

afficher\_info(nom="Alice", age=30)

#### Combinaison des deux :

Vous pouvez combiner \*args et \*\*kwargs dans la définition d’une fonction.

def fonction\_mixte(\*args, \*\*kwargs):

print(args)

print(kwargs)

**6. Fonctions Lambda, Map, Reduce & Filter**

#### Fonctions lambda :

Fonctions anonymes en une seule ligne.

carre = lambda x: x \* x

print(carre(5)) # affiche 25

#### Fonction map() :

Applique une fonction à chaque élément d’un itérable.

nombres = [1, 2, 3, 4]

carres = list(map(lambda x: x\*\*2, nombres))

#### Fonction filter() :

Filtre un itérable selon une condition.

pairs = list(filter(lambda x: x % 2 == 0, nombres))

#### Fonction reduce() (à importer depuis functools) :

Applique de façon cumulative une fonction aux éléments d’un itérable.

from functools import reduce

produit = reduce(lambda x, y: x \* y, nombres)

**7. Les dictionnaires**

#### Création d’un dictionnaire :

Un dictionnaire stocke des paires clé/valeur.

mon\_dict = {"nom": "Alice", "age": 30, "ville": "Paris"}

#### Accès et modification :

Accédez aux valeurs par leur clé et modifiez-les.

print(mon\_dict["nom"]) # affiche Alice

mon\_dict["age"] = 31

#### Méthodes utiles :

* + keys() pour obtenir les clés.
  + values() pour obtenir les valeurs.
  + items() pour obtenir les paires clé/valeur.

for cle, valeur in mon\_dict.items():

print(cle, ":", valeur)