Les fonctions en Python

Introduction

Dans tout langage de programmation, **les fonctions** constituent une brique essentielle pour structurer, organiser et réutiliser le code. En Python, une fonction est un bloc d'instructions nommé qui permet d'exécuter une tâche précise lorsque celle-ci est appelée. On la définit à l'aide du mot-clé `def`, suivi du nom de la fonction, d'une liste éventuelle de paramètres entre parenthèses et d'un bloc d'instructions indenté. Lorsqu'une fonction accomplit son travail, elle peut renvoyer un résultat à l'aide du mot-clé `return`.

L'intérêt majeur des fonctions réside dans la décomposition d'un programme en sous-problèmes cohérents et indépendants. On peut ainsi isoler des calculs, éviter la répétition du code, et faciliter les tests ou la maintenance. Une bonne pratique consiste à donner aux fonctions des noms explicites et à documenter leur comportement à l'aide d'une **docstring**.

Python introduit également la notion de portée des variables, c'est-à-dire la zone du programme où une variable est accessible. Les variables définies à l'intérieur d'une fonction sont locales et disparaissent à la fin de son exécution, tandis que celles définies à l'extérieur ont une portée globale. Cette distinction garantit l'isolation des fonctions et prévient les effets de bord indésirables.

Les fonctions peuvent recevoir un nombre variable de paramètres, utiliser des valeurs par défaut ou des arguments nommés, ce qui leur confère une grande flexibilité. Il est aussi possible de créer des fonctions anonymes, dites **lambda**, pour des opérations simples et ponctuelles. Grâce à cette richesse syntaxique et conceptuelle, Python permet d'adopter un style de programmation clair, modulaire et expressif, où chaque fonction joue le rôle d'un outil bien défini au service de la lisibilité et de la robustesse du programme.

Exemples de fonctions

Il y a deux types de fonctions de manière générale, les **procédures** et les **fonctions**. Même si les deux se construisent de la même façon, ils ont des différences importantes.

Une procédure est une fonction qui ne renvoie pas de résultat.

```python
def dire_bonjour(nom):
    print("Bonjour " + nom)

dire_bonjour("Jean") # Bonjour Jean
```

La fonction `dire_bonjour` est une procédure qui renvoie `None` et ne fait rien d'autre que afficher le message « Bonjour » suivi du nom donné en argument.

On le sait, en survolant la fonction, on peut voir `-> None` qui indique que la fonction ne renvoie pas de résultat.

Une fonction, au contraire, renvoie (return) un résultat.

```
def somme(a, b):
    return a + b

resultat = somme(2, 3)
print(resultat) # 5
```

Donc si elle retourne un résultat, il faut donc stocker le retour dans une variable.

La fonction `somme` est une fonction qui renvoie le résultat de l'addition de ses deux arguments. Litéralement, elle retourne le résultat de `a + b`.

Les développeurs n'aiment pas trop les procédures, car elles ne retournent rien. De ce fait, on ne sait pas trop si elles ont bien fonctionné.

Lorsqu'on écrit une fonction, on cherche avant tout à produire un résultat, c'est-à-dire une valeur exploitable par le reste du programme. Une fonction qui retourne une valeur, même simple comme un booléen `True` ou `False`, est donc toujours plus utile qu'une procédure qui ne renvoie rien.

```
def creates_folder(folder_name):
    os.mkdir(folder_name)
    return True

if creates_folder("my_folder"):
    print("Folder created")
```

Cette fonction nous permet ainsi de créer un dossier et de s'assurer que la création a réussi.

En Python, une fonction qui ne comporte pas d'instruction `return` renvoie implicitement None. Ce type de comportement limite souvent les possibilités de test, de réutilisation et de composition du code. À l'inverse, une fonction qui renvoie une valeur permet de chaîner les appels, de vérifier facilement son comportement dans des tests unitaires, et de prendre des décisions à partir de son résultat.

Paramètres et arguments

Remettons les termes à leur place. Dans la fonction `somme`, on a deux `paramètres`, `a` et `b`. Lorsque on appelle cette fonction, on lui fournit deux **arguments**, `2` et `3`.

Ces arguments peuvent être transmis de deux manières différentes : par **position** ou par **nom**. Comprendre cette distinction est fondamental, car elle influence la façon dont les valeurs sont associées aux paramètres de la fonction.

Un **positional argument** (argument positionnel) est un argument dont la valeur est associée à un paramètre selon l'ordre dans lequel il apparaît dans l'appel de la fonction. Cela signifie que la première valeur transmise correspond au premier paramètre, la deuxième valeur au deuxième paramètre, et ainsi de suite. Ce mode est le plus direct, mais il exige de respecter strictement l'ordre défini dans la signature de la fonction. Par exemple :

```
def afficher_message(nom, message):
    print(nom + " dit : " + message)

afficher_message("Alice", "Bonjour !")
```

Ici, `"Alice"` est associé à `nom`, et `"Bonjour !"` à `message` simplement par leur position respective.

Un **naming argument** (ou argument nommé) est un argument dont la valeur est associée explicitement à un paramètre par son nom. Ce mode d'appel rend le code plus lisible et plus souple, car l'ordre des arguments n'a plus d'importance. On précise directement à quel paramètre on souhaite affecter une valeur :

```
afficher_message(message="Salut !", nom="Bob")
```

Même si les arguments ne sont pas dans le même ordre que dans la définition de la fonction, Python sait que `message` reçoit `"Salut !"` et que `nom` reçoit `"Bob"`.

> L'équipe de Python préconise d'ailleurs d'utiliser le Naming Argument

Les arguments nommés sont donc particulièrement utiles lorsque la fonction comporte de nombreux paramètres ou lorsque certains ont des valeurs par défaut. Ils rendent les appels plus explicites et évitent les erreurs d'ordre, tout en améliorant la compréhension du code.

> En Python, les **f-strings** (pour formatted strings) sont une manière moderne et efficace d'insérer des valeurs de variables directement à l'intérieur d'une chaîne de caractères. Au lieu de :
>
> ```
> print(nom + " dit : " + message)
> ```
>
> on peut écrire :
>
> ```
> print(f"{nom} dit : {message}")
> ```

> Une f-string se reconnaît grâce à la lettre `f` placée juste avant les guillemets ouvrants de la chaîne. Les f-strings permettent également d'évaluer directement des expressions, ce qui les rend très puissantes. Par exemple, on peut écrire :
>
> ```
> print(f"Le nombre de lettres du message est : {len(message)}")
> ```
>
> Cela permet de calculer la longueur d'une chaîne de caractères à la volée.

Les opérateurs * et / dans les fonctions

Lorsqu'on définit une fonction en Python, on peut rencontrer une syntaxe qui surprend au début :

```
def ma_fonction(a, b, *, c, d):
    ...
```

Le symbole `*` placé ici sert à **imposer une règle de clarté** : tous les paramètres écrits **après** cette étoile doivent **obligatoirement** être passés par **nom** (Naming Argument) lors de l'appel de la fonction.

Autrement dit, `a` et `b` peuvent être donnés dans l'ordre (Positional Argument), mais `c` et `d` doivent être indiqués avec leur nom pour que Python sache clairement à quoi chaque valeur correspond.

Regardons un exemple :

```
def ma_fonction(a, b, *, c, d):
    print(a, b, c, d)
```

Voici un appel correct :

```
ma_fonction(1, 2, c=3, d=4)
```

Et voici un appel qui provoque une erreur :

```
ma_fonction(1, 2, 3, 4)
# TypeError: ma_fonction() takes 2 positional arguments but 4 were given
```

Pourquoi cette contrainte ? Parce qu'elle **force la lisibilité**. Lorsqu'une fonction commence à avoir beaucoup de paramètres, il devient facile de se tromper dans l'ordre. Grâce à cette étoile, Python oblige à nommer explicitement certains arguments, ce qui rend le code beaucoup plus clair :

```
def creer_utilisateur(nom, prenom, *, age, ville):
    print(f"{prenom} {nom}, {age} ans, habite à {ville}")
```

```
creer_utilisateur("Durand", "Alice", age=25, ville="Paris")
```

Ici, personne ne peut confondre ce que représente chaque valeur : le code est **auto-explicite**.

En résumé, cette étoile `*` agit comme une **barrière logique** : tout ce qui est avant peut être passé par position, et tout ce qui est après doit être passé par nom. C'est une manière élégante pour Python d'encourager les bonnes pratiques de lisibilité et d'éviter les erreurs d'ordre dans les appels de fonction.

En Python, le symbole `/` dans la définition des paramètres d'une fonction joue un rôle opposé à celui de l'astérisque `*`. Là où `*` impose que les paramètres qui le suivent soient passés **par nom**, le `/` impose au contraire que les paramètres qui le précèdent soient passés **par position uniquement**.

On appelle ce symbole **l'opérateur de séparation des arguments positionnels** (*positional-only argument separator*).

Prenons un exemple simple :

```
def afficher_coordonnees(x, y, /):
    print(f"Position : x = {x}, y = {y}")
```

Ici, le `/` signifie que `x` et `y` sont des **arguments positionnels seulement**. Cela veut dire que lors de l'appel de la fonction, on ne peut pas écrire leurs noms :

```
afficher_coordonnees(10, 20)     # ✅ Correct
afficher_coordonnees(x=10, y=20) # ❌ Erreur : arguments positionnels
seulement
```

Cette restriction est utile dans plusieurs situations. D'abord, elle empêche qu'on s'appuie sur des noms de paramètres qui pourraient changer à l'avenir. Par exemple, les fonctions intégrées comme `len()` ou `pow()` utilisent souvent ce mécanisme pour garantir la compatibilité du code même si les développeurs internes de Python modifient un jour leurs noms de paramètres. Ensuite, elle permet d'écrire des fonctions dont les arguments sont purement conceptuels ou anonymes, où l'ordre seul a un sens logique (comme des coordonnées, des valeurs mathématiques, ou des comparaisons).

On peut aussi combiner `/` et `*` dans la même signature pour définir précisément quels paramètres doivent être donnés par position, lesquels peuvent être donnés librement, et lesquels doivent être nommés. Par exemple :

```
def exemple(a, b, /, *, c, d):
    print(a, b, c, d)

exemple(1, 2, c=3, d=4)  # ✅
```

Dans cet exemple :

- `a` et `b` doivent être donnés par position,
- `c` et `d` doivent être donnés par nom.

Ainsi, l'opérateur `/` est une manière de **contrôler la façon dont les arguments doivent être passés** à une fonction, renforçant la cohérence et la lisibilité du code.

Si on parle maintenant de ces deux symboles , `/` et `*`, c'est parce que vous allez commencer à les croiser en survolant les signatures de fonctions et de méthodes dans votre éditeur. Vous savez déjà comment lire ces infobulles et décrypter les paramètres qu'elles affichent, donc l'objectif ici est simplement de vous permettre de **comprendre ce que signifient ces signes particuliers** lorsqu'ils apparaissent.

Par exemple, si vous voyez une fonction affichée comme :

```
len(obj, /)
```

cela veut dire que `obj` doit être passé **par position uniquement**.

Vous ne pourrez pas écrire :

```
len(obj="Jean") # ❌ Erreur
len("Jean")     # ✅ Correct
```

car `obj` doit être passé **par position**.

En connaissant cette règle, vous saurez immédiatement comment appeler correctement la fonction sans provoquer d'erreur, et surtout, vous comprendrez la logique de conception qui se cache derrière. L'idée n'est pas de les mémoriser maintenant, mais de **reconnaître leur rôle** quand vous les verrez. Vous saurez qu'à gauche d'un `/` se trouve un argument positionnel seulement, et à droite d'un `*` se trouve un argument nommé.

Les mots clés `pass` et Elipsis `...`

Le mot-clé `pass` en Python est une instruction particulière qui ne fait **absolument rien** lorsqu'elle est exécutée. Cela peut sembler étrange au premier abord, mais cette instruction a une utilité bien précise : elle permet de **laisser un bloc de code vide** sans provoquer d'erreur de syntaxe.

Dans le contexte d'une fonction, `pass` est souvent utilisé comme **bouchon temporaire**. Lorsque l'on définit la structure d'une fonction, mais qu'on ne veut pas encore écrire son contenu, Python exige qu'il y ait au moins une instruction à l'intérieur du bloc. Sans cela, l'interpréteur lèverait une erreur. C'est dans cette situation que `pass` entre en jeu.

Exemple :

```
def calculer_total():
    pass
```

Ici, la fonction `calculer_total()` ne fait rien, mais elle est **syntaxiquement valide**. On peut donc déjà l'appeler, la documenter, l'intégrer à une classe ou à un module, et y revenir plus tard pour en écrire le contenu.

C'est une pratique très utilisée lorsqu'on construit l'architecture d'un programme avant d'en implémenter les détails. Cela permet de **préparer la structure du code**, d'écrire les signatures des fonctions et de tester l'enchaînement global, sans bloquer la progression du projet.

On rencontre aussi `pass` dans d'autres contextes : pour laisser vide une classe, une condition `if`, ou une boucle, lorsqu'on veut réserver la place du code à venir.

Ainsi, dans une fonction, `pass` sert de **placeholder**, un mot-clé d'attente qui permet de garder un code propre et fonctionnel, tout en indiquant clairement qu'il reste une partie à compléter plus tard.

L'instruction `pass` et l'objet spécial `...` (appelé **ellipsis**) peuvent sembler similaires, car tous deux permettent de laisser un bloc de code vide sans provoquer d'erreur. Pourtant, ils n'ont **ni la même nature**, **ni le même usage**.

L'ellipse `...` n'est pas une instruction : c'est un **objet Python** de type `EllipsisType`. On peut l'utiliser **partout où une expression est attendue**, comme une valeur symbolique ou un marqueur de code non implémenté. Elle est donc davantage **sémantique** que syntaxique : elle exprime une intention, un "à venir" dans le code.

Exemple :

```
def calculer_total():
    ...
```

Ici, Python ne fait pas appel à une instruction, mais évalue simplement l'objet `Ellipsis`, ce qui est autorisé car c'est une expression valide. Dans la pratique, cela revient à avoir une fonction vide, mais l'idée est différente : `...` indique que **le corps est volontairement incomplet** , une sorte de promesse implicite que la fonction sera complétée plus tard.

On rencontre souvent `...` dans des classes abstraites, des squelettes de modules ou des *stubs* destinés à l'auto-complétion ou à la documentation. Par exemple :

```
class Repository:
    def save(self, entity): ...
    def delete(self, id): ...
```

Nous y reviendrons plus tard.

En résumé :

- `pass` est **une instruction** qui "ne fait rien" mais rend le bloc exécutable.

- `...` est **un objet** qui exprime “non implémenté” et peut être utilisé comme un marqueur plus conceptuel, notamment dans les architectures ou les interfaces.

Dans un code pédagogique ou en développement progressif, `pass` est plus courant. `...` devient intéressant lorsqu’on veut signaler une intention, une structure en attente ou un comportement abstrait.

La portée d'une variable et le principe LEGB

La **portée d’une variable** (ou *variable scope*) désigne l’endroit du programme où cette variable est **connue, visible et accessible**. En Python, toutes les variables ne vivent pas dans le même espace : certaines existent uniquement à l’intérieur d’une fonction, d’autres sont accessibles partout dans le fichier. Comprendre cette différence est essentiel pour éviter des erreurs ou des comportements inattendus.

Commençons par un exemple simple :

```
x = 10  # variable définie à l’extérieur d’une fonction

def afficher():
    y = 5 # variable définie à l’intérieur d’une fonction
    print("x =", x)
    print("y =", y)

afficher()
print("x à l’extérieur =", x)
print("y à l’extérieur =", y)  # Erreur !
```

Ici, `x` est définie **à l’extérieur** de la fonction : on dit qu’elle a une **portée globale**. Elle est accessible **partout** dans le programme, y compris à l’intérieur de la fonction.

En revanche, `y` est définie **à l’intérieur** de la fonction `afficher()`. Elle n’existe que **le temps de l’exécution de cette fonction**, et on ne peut pas l’utiliser ailleurs. Lorsqu’on essaie d’afficher `y` à la fin, Python déclenche une erreur :

```
NameError: name 'y' is not defined
```

Cela illustre une règle simple :

- une variable définie **dans une fonction** est **locale** à cette fonction,
- une variable définie **en dehors de toute fonction** est **globale** dans le module.

Python applique cette logique à travers ce qu’on appelle le modèle **LEGB** (Local, Enclosing, Global, Built-in), qui correspond à l’ordre dans lequel l’interpréteur cherche une variable :

1. **Local** : dans la fonction en cours.
2. **Enclosing** : dans une fonction englobante (si on est dans une fonction à l’intérieur d’une autre).
3. **Global** : dans le fichier courant (en dehors des fonctions).
4. **Built-in** : dans les noms prédéfinis de Python (comme `len`, `print`, etc.).

Regardons un exemple plus visuel :

```
x = "globale"

def exterieure():
    x = "enclosing"  # variable de la fonction englobante

    def interieure():
        x = "locale"
        print(x)

    interieure() # On execute la fonction 'interieur'
    print(x)

exterieure() # On execute la fonction 'exterieure'
print(x)
```

Résultat :

```
locale
enclosing
globale
```

Python commence toujours par chercher la variable dans la portée la plus proche, puis remonte progressivement.

Enfin, on peut **modifier une variable globale** à l'intérieur d'une fonction, mais uniquement si on la déclare explicitement avec le mot-clé `global` :

```
compteur = 0

def incrementer():
    global compteur
    compteur += 1
```

Sans ce mot-clé, Python considérerait `compteur` comme une nouvelle variable locale et lèverait une erreur. Avec le mot clé "global", on lui dit en somme "Cherche une variable à l'extérieur de ton scope qui porte le nom de 'compteur'".

En résumé :

- Les variables **locales** existent à l'intérieur d'une fonction.
- Les variables **globales** existent dans tout le programme.
- Python choisit toujours la version la plus proche selon la règle **LEGB**.

Comprendre cette portée, c'est apprendre à **contrôler la durée de vie et la visibilité** des données qu'on manipule, et donc à écrire un code plus propre, plus prévisible et plus sûr.

Les Docstrings

Les **docstrings** (ou *documentation strings*) sont un élément fondamental du style de programmation en Python. Ce sont des chaînes de caractères placées **juste après la définition d'une fonction, d'une classe ou d'un module**, et elles servent à **documenter le code** directement à l'intérieur du programme.

Une docstring permet d'expliquer ce que fait une fonction, à quoi servent ses paramètres, et ce qu'elle renvoie. Elle est accessible depuis le code lui-même, grâce à l'attribut spécial `__doc__`, ou via la fonction `help()`.

Voici un exemple simple :

```
def saluer(nom):
    """
    Affiche un message de salutation personnalisé.

    Paramètres :
        nom : le prénom de la personne à saluer.

    Retour :
        None
    """
    print(f"Bonjour {nom} !")
```

Ici, la chaîne placée entre triple guillemets `""" ... """` constitue la docstring de la fonction. Python la reconnaît automatiquement, sans qu'il soit nécessaire de l'affecter à une variable.

On peut ensuite y accéder directement :

```
print(saluer.__doc__)
```

ou bien afficher une aide plus lisible :

```
help(saluer)
```

L'intérêt principal des docstrings est de **rendre le code compréhensible sans quitter le fichier source**. Elles servent de documentation intégrée, utile aussi bien pour les autres développeurs que pour soi-même quelques semaines plus tard.

Il est d'usage d'y préciser :

- le rôle général de la fonction ou de la classe,
- les paramètres d'entrée (avec leur type attendu),

- la valeur de retour,
- et parfois les exceptions possibles.

Dans les projets professionnels ou pédagogiques, les docstrings suivent souvent un format reconnu comme **PEP 257**, ou bien des styles plus structurés comme *Google style* ou *NumPy style*. Exemple au format Google :

```
def addition(a, b):
    """
    Calcule la somme de deux entiers.

    Args:
        a (float): premier nombre à additionner.
        b (float): second nombre à additionner.

    Returns:
        somme (float): la somme de a et b.
    """
    return a + b
```

Souvenez-vous que, plus tôt, on a survolé des fonctions et des méthodes Python dans VS Code pour observer les petites boîtes d'aide qui s'affichaient. Ces boîtes indiquaient la signature de la fonction, la liste des paramètres et une courte description de son rôle. Ce texte que vous aviez lu ne vient pas d'un fichier caché ou d'un site externe : il provient directement des docstrings écrites dans le code source de Python lui-même.

Autrement dit, lorsqu'on voit par exemple :

```
help(len)
```

ou qu'on survole `len()` dans l'éditeur, l'explication qui apparaît est issue de la docstring de cette fonction.

Cela signifie que si vous écrivez vos propres fonctions avec une docstring claire et structurée, vous obtiendrez exactement le même comportement : en survolant votre fonction, VS Code affichera son nom, ses paramètres, et la description que vous aurez fournie.

C'est donc un moyen de rendre votre code professionnel et auto-documenté : non seulement il fonctionne, mais il "s'explique lui-même" lorsque quelqu'un d'autre (ou vous-même plus tard) le lit ou le survole.

Les générateurs

Avant de voir les générateurs, rappel de la fonction `range`.

La fonction `range()` en Python est très utilisée pour **générer une suite de nombres**, souvent dans le cadre d'une boucle `for`. Elle ne crée pas une liste, mais un **objet spécial** qui produit les nombres un à un. Cela la rend très efficace, même pour de grandes séquences.

On peut l'appeler de trois manières différentes selon le nombre d'arguments qu'on lui donne :

- un seul argument → la borne de fin,
- deux arguments → la borne de début et la borne de fin,
- trois arguments → la borne de début, la borne de fin et le pas d'incrémentation.

Voyons ces trois cas :

```
range(10)
```

Ici, `range(10)` commence à `0` et s'arrête juste **avant** `10`. Python suit toujours cette règle : la borne de fin n'est **jamais incluse**. Cela équivaut donc à la suite :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Deuxième cas :

```
range(0, 10)
```

Cette fois, on indique explicitement le début (`0`) et la fin (`10`). Le résultat est identique au précédent, car `range(10)` et `range(0, 10)` reviennent au même :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Troisième cas :

```
range(0, 10, 2)
```

Ici, on ajoute un troisième argument : le **pas**. Cela veut dire que Python va compter de `2` en `2`, en partant de `0` et en s'arrêtant avant `10` :

```
[0, 2, 4, 6, 8]
```

Autrement dit :

- le premier argument est la **valeur de départ** (incluse),
- le deuxième est la **valeur de fin** (exclue),
- le troisième est le **pas** (positif ou négatif).

On peut aussi l'utiliser à rebours :

```
range(10, 0, -1)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Et pour visualiser ce qu'un `range` produit, on peut toujours le convertir en liste :

```
list(range(0, 10, 2))
```

Cela affiche concrètement les valeurs générées.

Revenons aux générateurs.

Commençons par un exemple très simple : une fonction qui affiche chaque élément d'une liste grâce à une boucle `for`.

```
def afficher_elements(n):
    for element in range(n):
        print(element)

afficher_elements(5)
# 0
# 1
# 2
# 3
# 4
```

Ici, la fonction parcourt la liste et affiche chaque élément un par un. Tout se passe bien, mais une fois le `print` exécuté, les valeurs sont perdues : on ne peut pas les récupérer, ni les réutiliser ailleurs dans le programme. Cette fonction ne **retourne rien**, elle se contente d'afficher.

Essayons maintenant avec un `return` :

```
def retourner_elements(n):
    for element in range(n):
        return element

print(retourner_elements(10))
# 0
```

La fonction ne retourne que le premier élément de la liste. Pourquoi ? Parce que **`return`** met fin à la fonction. Gardez bien cela à l'esprit ! Tout le code après la ligne `return` n'est pas exécuté.

C'est là qu'interviennent les **générateurs**.

Un générateur permet de produire les valeurs **une par une**, au moment où on en a besoin, sans tout stocker en mémoire. On crée un générateur en remplaçant `return` par le mot-clé `yield`, on va créer un générateur qui renvoie les carrés des entiers de `0` à `n-1` :

```
def generer_carres(n):
    for i in range(n):
        yield i ** 2
```

Cette fonction ne renvoie pas une liste, mais un **objet générateur**, c'est-à-dire une sorte de "machine à produire des valeurs au fur et à mesure".

Si vous faîtes :

```
carres = generer_carres(10)
print(carres)
# <generator object generer_carres at 0x1006e9970>
```

ici, `carres` n'est pas une liste, mais un objet générateur. Il ne contient pas encore les valeurs, **mais sait comment les produire**.

On peut l'utiliser ainsi :

```
for valeur in carres:
    print(valeur)
```

Le comportement est le même qu'avant : les valeurs s'affichent une par une. Mais la différence est profonde : la fonction ne renvoie pas tout d'un coup. Elle **suspend son exécution** à chaque `yield`, en gardant en mémoire son état, puis reprend là où elle s'était arrêtée au tour suivant.

Et pour bien comprendre ce qu'il se passe, on peut écrire :

```


print(next(carres))
print(next(carres))
print(next(carres))

print("Début de la boucle")
for valeur in carres:
    print(valeur)
```

```
0
1
4
Début de la boucle
9
16
25
```

```
36
49
64
81
```

Lorsqu'on utilise un générateur, on peut le parcourir avec une boucle `for`, mais il est aussi possible de récupérer les valeurs une à une manuellement grâce à la fonction intégrée `next()`.

Chaque appel à `next(carres)` demande au générateur de fournir la valeur suivante.

À ce stade, la fonction s'est arrêtée temporairement au troisième `next()`, mais elle n'est pas terminée : Python a simplement mis son exécution en pause, prête à reprendre au prochain `next()`.

La boucle `for` utilise elle aussi le mécanisme de `next()` en interne. Elle reprend le générateur là où il s'était arrêté.

Quand on remet Python dans le contexte de sa création, au début des années 1990, il faut se souvenir que les ordinateurs n'avaient rien à voir avec ceux d'aujourd'hui. La mémoire vive se comptait souvent en mégaoctets, parfois même en kilooctets, et il fallait faire extrêmement attention à la façon dont on stockait et manipulait les données.

Les **générateurs** ont été inventés dans cet esprit : ils permettent de **produire des valeurs une par une**, sans jamais tout charger en mémoire d'un coup. À l'époque, c'était un moyen très concret d'économiser les ressources limitées d'un ordinateur. Par exemple, au lieu de créer une énorme liste de nombres, Python pouvait simplement "donner le suivant quand on en a besoin", ce qui évitait de saturer la mémoire.

Aujourd'hui, les machines sont infiniment plus puissantes, et la plupart du temps, on ne pense plus à ces questions d'économie mémoire. On écrit des boucles sur des listes entières sans se poser de problème. Pourtant, le mécanisme des générateurs n'a pas perdu son intérêt , bien au contraire.

Vous pouvez litéralement écrire :

```
carres = generer_carres(10000000000000000000)
print(carres)

print(next(carres))
print(next(carres))
print(next(carres))

# ⚠ retirer les lignes contenant la boucle for
```

Et lancer le programme, vous n'avez aucune crainte de faire planter votre ordinateur, car contrairement à une liste il n'a pas produit les valeurs

Alors que la ligne suivante, elle, risque de planter votre ordinateur :

```
print(list(range(10000000000000000000))) # ⚠ A ne pas exécuter
```

Dans des programmes modernes qui traitent des fichiers volumineux, des flux de données, ou des appels réseau répétés, **les générateurs restent un outil essentiel** pour améliorer les performances. Ils permettent de réduire l'usage de la mémoire et d'accélérer les traitements, surtout lorsqu'on travaille avec des données qu'on n'a pas besoin de charger entièrement.

En réalité, **on a un peu "oublié"** qu'ils existent, simplement parce que nos ordinateurs masquent leurs avantages. Mais du point de vue d'un développeur conscient des ressources, utiliser un générateur au bon endroit peut faire une **grande différence** : le programme devient plus fluide, plus rapide, et plus économe.

En résumé, les générateurs sont un **héritage intelligent des débuts de Python**, conçus pour pallier la faiblesse du matériel d'alors, mais toujours extrêmement pertinents aujourd'hui pour écrire du code performant et élégant.

Les fonctions lambda

Imaginons qu'on ait un dictionnaire dont les clés sont des noms et les valeurs des âges :

```
personnes = {"Alice": 25, "Bob": 19, "Charlie": 32}
```

On aimerait trier ces personnes **en fonction de leur âge**, c'est-à-dire selon la valeur du dictionnaire et non selon la clé.

La fonction intégrée `sorted()` de Python permet de le faire.

Si je fais :

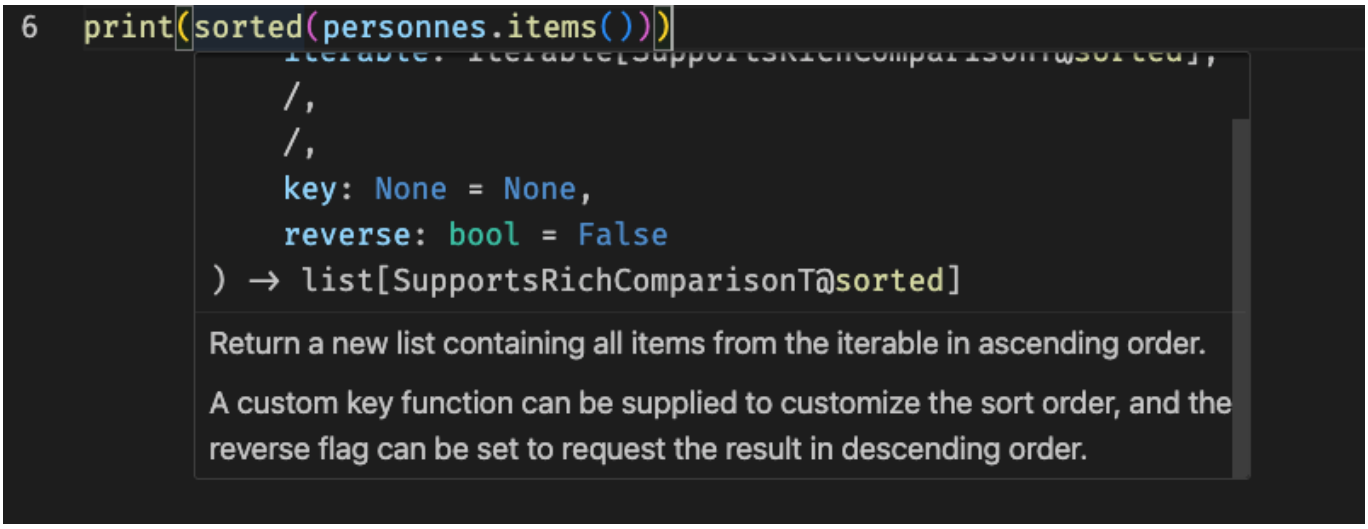
```
print(sorted(personnes))            # ['Alice', 'Bob', 'Charlie']
print(sorted(personnes.keys()))     # ['Alice', 'Bob', 'Charlie']
print(sorted(personnes.values()))   # [19, 25, 32]
print(sorted(personnes.items()))    # [('Alice', 25), ('Bob', 19),
('Charlie', 32)]
```

On constate que le premier et le deuxième `print` donnent le même résultat. On en déduit donc que **par défaut**, Python itère sur les clés.

Le dernier `print` affiche les tuples triés par ordre alphabétique de leur clé.

Revenons sur notre problématique, on souhaite obtenir les informations `[('Alice', 25), ('Bob', 19), ('Charlie', 32)]` mais triées par ordre croissant de **leur valeur**.

Si on survole la fonction `sorted()` on voit qu'elle accepte un paramètre optionnel `key=` qui indique **quelle fonction utiliser pour comparer les éléments**.



Commençons avec une fonction classique, je vais créer une fonction qui lorsque je lui donne un tuple de type `("Jean", 19)`, elle me renvoie `19`

```
def extraire_age(tup):
    return tup[1]

resultat = sorted(personnes.items(), key=extraire_age)
print(resultat)
```

> On n'invoque pas la fonction `extraire_age()` directement, mais on l'utilise comme argument. C'est `sorted` qui va invoquer la fonction pour chaque élément.

Ici, `personnes.items()` renvoie une liste de tuples comme :

```
[('Alice', 25), ('Bob', 19), ('Charlie', 32)]
```

La fonction `extraire_age()` reçoit un de ces tuples et renvoie la valeur à comparer, l'âge, donc `tup[1]`. Le paramètre `key=extraire_age` indique à `sorted()` d'utiliser cette fonction pour trier les éléments.

Cela fonctionne et on obtient le résultat attendu.

```
[('Bob', 19), ('Alice', 25), ('Charlie', 32)]
```

Mais on remarque une chose : cette fonction `extraire_age()` n'aura **aucune autre utilité** dans le reste du programme. Elle ne sert qu'une seule fois, pour une tâche très ponctuelle.

C'est exactement dans ce genre de situation qu'interviennent les **fonctions lambda**. Une *lambda* permet de définir une fonction **anonyme** (c'est-à-dire sans nom) directement dans l'appel d'une autre fonction.

On peut donc réécrire le code ainsi :

```
resultat = sorted(personnes.items(), key=lambda item: item[1])
print(resultat)
```

Ici, `lambda item: item[1]` définit une petite fonction “sur place” :

- `item` est le paramètre,
- `item[1]` est la valeur renvoyée,
- et on n’a pas besoin de donner de nom à cette fonction, car elle ne servira qu’ici.

En somme, la fonction `lambda` est une **fonction anonyme, légère et temporaire**, créée uniquement pour un besoin ponctuel. Elle se comporte exactement comme une fonction normale, mais elle s’écrit en une seule ligne, sans `def`, ni `return`.

C’est une manière élégante de dire à Python :

“Voici une petite fonction que je n’ai pas besoin de nommer, utilise-la juste pour ce tri.”

On retrouve ce même principe avec d’autres fonctions comme `map()`, `filter()` ou `sorted()`, dès qu’on a besoin de passer une petite fonction simple en paramètre.

Ainsi, la fonction lambda n’est pas un nouveau concept magique : c’est juste une **façon raccourcie d’écrire une fonction éphémère**.

Les fonctions de haut niveau (Higher-order functions)

Une fonction d’ordre supérieur est une fonction qui peut **recevoir une autre fonction en argument** ou **retourner une fonction**. En Python, ce concept provient de la programmation **fonctionnelle** et permet d’écrire un code plus expressif et concis.

> La programmation fonctionnelle est une manière de penser le code où l’on privilégie **les fonctions plutôt que les objets ou les instructions séquentielles**.
>
> Dans ce paradigme, on cherche à décomposer un problème en **petites fonctions pures**, c’est-à-dire des fonctions qui **renvoient toujours le même résultat** pour les mêmes données d’entrée et **n’ont pas d’effet secondaire** (elles ne modifient pas de variables extérieures, ne lisent pas de fichiers, etc.).
>
> L’idée est de **transformer les données** en les passant d’une fonction à l’autre, un peu comme sur une chaîne de montage.
>
> Cette approche rend le code plus **prévisible, concis et facile à tester**, car chaque fonction se comporte comme une petite boîte isolée qui ne dépend que de ses paramètres.

Les fonctions comme `map()`, `filter()`, `reduce()` ou encore `partial()` du module `functools` sont des exemples classiques de fonctions d’ordre supérieur.

La fonction `map()`

`map()` applique une fonction à chaque élément d’un itérable (liste, tuple, etc.) et retourne un objet `map`, que l’on peut convertir en liste.

Exemple :

```
nombres = [1, 2, 3, 4]

def carre(x):
    return x ** 2

resultat = map(carre, nombres)
print(list(resultat))  # [1, 4, 9, 16]
```

On peut aussi l'utiliser avec une fonction `lambda` :

```
nombres = [1, 2, 3, 4]
resultat = map(lambda x: x ** 2, nombres)
print(list(resultat))
```

La fonction `filter()`

`filter()` permet de **sélectionner** uniquement les éléments d'un itérable pour lesquels une fonction retourne `True`.

Exemple :

```
nombres = [1, 2, 3, 4, 5, 6]

def pair(x):
    return x % 2 == 0

resultat = filter(pair, nombres)
print(list(resultat))  # [2, 4, 6]
```

Avec une `lambda` :

```
resultat = filter(lambda x: x % 2 == 0, nombres)
print(list(resultat))
```

La fonction `reduce()`

`reduce()` n'est pas incluse directement dans le langage, mais se trouve dans le module `functools`. Elle **réduit** une séquence en une seule valeur en appliquant une fonction cumulativement.

Exemple : calculer la somme des éléments d'une liste.

```

```

```
from functools import reduce

nombres = [1, 2, 3, 4]

def addition(x, y):
    return x + y

resultat = reduce(addition, nombres)
print(resultat)  # 10
```

Ici, `reduce` exécute :

- `addition(1, 2)` → 3
- `addition(3, 3)` → 6
- `addition(6, 4)` → 10

Ou avec une fonction `lambda` :

```
from functools import reduce

nombres = [1, 2, 3, 4]

resultat = reduce(lambda x, y: x + y, nombres)
print(resultat)  # 10
```

Ce type de raisonnement est courant en programmation fonctionnelle.

La fonction `partial()`

`partial()` permet de **fixer certains arguments** d'une fonction et de créer une **nouvelle fonction spécialisée**.

Exemple :

```
from functools import partial

def puissance(base, exposant):
    return base ** exposant

carre = partial(puissance, exposant=2)
cube = partial(puissance, exposant=3)

print(carre(5))  # 25
print(cube(2))   # 8
```

On a ici créé deux fonctions dérivées de `puissance`, sans dupliquer le code. `partial` est très utile pour adapter une fonction générale à un cas d'usage précis.

En résumé, les fonctions d'ordre supérieur sont un moyen d'écrire un code plus **déclaratif** : on décrit ce que l'on veut faire (appliquer, filtrer, réduire, spécialiser) plutôt que de détailler chaque boucle. Elles s'accordent parfaitement avec les fonctions anonymes (`lambda`) et les compréhensions, qui sont des piliers du style fonctionnel en Python.

Conclusion

L'étude des fonctions constitue une étape essentielle dans la maîtrise du langage Python. À travers l'ensemble des notions vues, on comprend qu'une fonction n'est pas seulement un moyen de regrouper des instructions, mais un véritable outil d'organisation, de clarté et de réutilisation du code.

On a d'abord vu qu'une fonction se définit avec `def`, qu'elle peut accepter des **paramètres** et retourner une **valeur** grâce à `return`. Cette distinction entre **procédure** (aucune valeur renvoyée) et **fonction** (retour d'un résultat) traduit deux manières de structurer la logique d'un programme.

Les **paramètres** permettent d'adapter le comportement d'une fonction à différents contextes, tandis que les **arguments nommés** et les opérateurs `*` et `/` offrent un contrôle précis sur la manière de les transmettre, garantissant à la fois lisibilité et robustesse.

Les mots-clés `pass` et `...` jouent quant à eux un rôle structurel : ils autorisent la création de fonctions "vides" lors de la phase de conception, marquant l'intention d'un développement ultérieur.

La compréhension de la **portée des variables** (modèle LEGB) permet de mieux raisonner sur la visibilité et la durée de vie des données, en distinguant clairement les variables locales, globales ou internes à une fonction imbriquée.

Les **docstrings**, intégrées directement dans le code, assurent une documentation lisible et accessible. Elles font partie intégrante de la philosophie Python : écrire un code explicite, compréhensible et auto-documenté.

Les **générateurs** et les **fonctions lambda** enrichissent cette approche fonctionnelle. Les premiers illustrent la puissance du modèle itératif de Python, en permettant la production de données à la demande sans surcharge mémoire. Les secondes introduisent une forme concise et expressive de définition de fonctions, parfaitement adaptée aux traitements ponctuels.

Enfin, les **fonctions d'ordre supérieur** (`map`, `filter`, `reduce`, `partial`) ouvrent la voie vers une pensée plus abstraite : on ne décrit plus seulement *comment* exécuter une tâche, mais *quelle transformation* appliquer à une donnée. Elles illustrent la convergence de Python entre programmation impérative et programmation fonctionnelle.

En somme, les fonctions constituent le socle de la programmation structurée en Python. Elles incarnent l'esprit du langage : un équilibre entre simplicité syntaxique, flexibilité d'usage et rigueur conceptuelle. Maîtriser ces notions, c'est acquérir la capacité d'écrire un code clair, modulaire et performant — une étape incontournable avant d'aborder la programmation orientée objet.