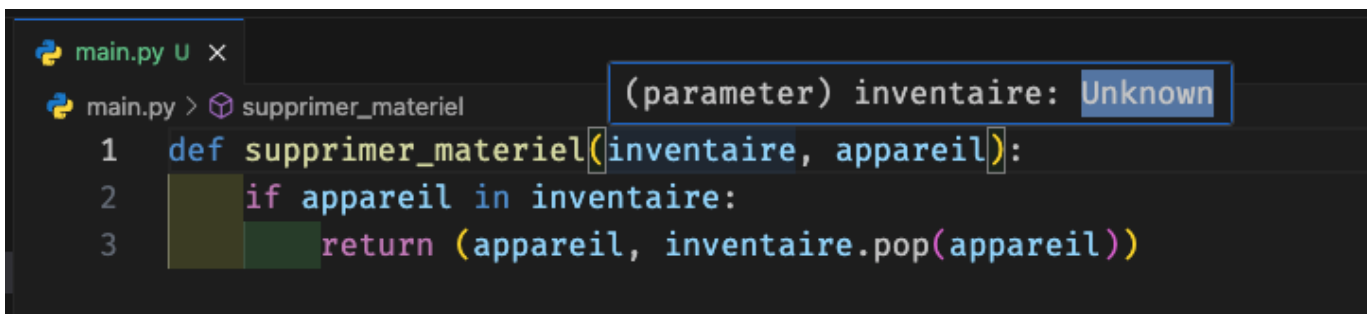


Le typage dynamique

Dans l'exercice sur les dictionnaires, quand on a écrit la fonction suivante :

```
def supprimer_materiel(inventaire, appareil):  
    if appareil in inventaire:  
        return (appareil, inventaire.pop(appareil))
```

Visual Studio Code n'a pas proposé la méthode `pop` avec la **dot notation** (`inventaire.`) parce qu'il n'a pas su **déduire le type** de la variable `inventaire`.



Pour l'éditeur, `inventaire` est **unknown**, c'est-à-dire qu'il ne sait pas s'il s'agit d'un dictionnaire, d'une liste ou d'un autre objet. Cette incertitude empêche l'autocomplétion.

C'est ici qu'intervient la notion de **typage** et plus précisément de **typage implicite** et **typage explicite**.

Python a fait le pari d'un **typage dynamique et implicite** : on n'a pas besoin d'indiquer le type d'une variable, il est déduit à l'exécution. Ainsi :

```
inventaire = {"pc": 2, "imprimante": 1}
```

Python sait au moment de l'exécution que `inventaire` est un dictionnaire, et l'instruction `inventaire.pop("pc")` fonctionnera.

Mais l'éditeur, lui, ne sait pas que `inventaire` est un dictionnaire si on ne l'indique pas dans la signature de la fonction.

C'est pour cela qu'on peut aider l'éditeur grâce à **l'inférence de type** :

```
def supprimer_materiel(inventaire: dict, appareil: str) -> tuple | None:  
    if appareil in inventaire:  
        return (appareil, inventaire.pop(appareil))
```

Dès qu'on ajoute ces **annotations de type**, VS Code et son moteur d'analyse (Pylance) comprennent ce qu'est `inventaire` et proposent `pop`, `items`, `keys`, etc.

On peut être plus précis dans l'annotation de type :

```
def supprimer_materiel(inventaire: dict[str, int], appareil: str) ->
tuple[str, int] | None:
    if appareil in inventaire:
        return (appareil, inventaire.pop(appareil))
```

En effet, nous avons un dictionnaire dont les clés sont des chaînes de caractères et les valeurs des entiers.

S'il entre dans la condition il renverra un **tuple** de deux éléments, l'un est une chaîne de caractères et l'autre est un entier. S'il n'entre pas dans la condition, la fonction renverra **None**.

C'est toute la force du typage progressif de Python : on garde la souplesse du langage dynamique, mais on gagne les avantages d'un langage typé statiquement, comme l'autocomplétion, la détection d'erreurs avant l'exécution et la lisibilité du code.

L'inférence de type, c'est la capacité de Python à deviner certains types à partir des valeurs. Par exemple :

```
a = 5          # int
b = "texte"    # str
c = [1, 2]     # list[int]
```

Mais cette inférence reste limitée dès qu'on entre dans le cadre des fonctions ou des structures complexes, où Python n'exécute pas le code pour deviner les types : d'où l'importance des annotations.

Le typage aide à **aller plus vite** car il donne à l'éditeur des informations précises sur les objets manipulés, ce qui améliore la complétion, la documentation contextuelle et la détection d'erreurs logiques avant même d'exécuter le programme.

Cependant, Python reste tolérant : même si on se trompe dans les types, le code s'exécute. Par exemple :

```
def addition(a: int, b: int) -> int:
    return a + b

print(addition("4", "5")) # s'exécute, renvoie '45'
```

Python n'empêche pas l'exécution car les annotations ne sont **pas contraignantes** à l'exécution. Elles servent uniquement à **informer l'humain et les outils** (VS Code, mypy, pyright).

C'est à la fois une **force** (souplesse) et une **faiblesse** (on peut faire des erreurs silencieuses). En pratique, un bon code Python moderne combine donc :

- un **typage explicite** dans les fonctions et classes,
- une **vérification statique** optionnelle avec un outil comme **mypy**,
- et des **tests unitaires** pour s'assurer du comportement réel à l'exécution.

Ainsi, en ajoutant simplement des types, on aide Python et surtout son environnement à mieux comprendre notre code, à prévenir les erreurs et à accélérer notre travail.

Avant d'aborder la POO, on peut aller plus loin sur le **typage** en expliquant plusieurs aspects essentiels pour comprendre comment Python gère les valeurs et la mémoire.

D'abord, il faut rappeler que **tout a un type en Python**. Même si on ne l'écrit pas, chaque valeur a un type interne :

```
type(42)           # int
type(3.14)         # float
type("bonjour")   # str
type(True)        # bool
type([1, 2, 3])   # list
type({"a": 1})    # dict
```

Cela signifie que chaque donnée sait comment elle doit se comporter. Par exemple, une chaîne sait se concaténer ("`a`" + "`b`"), une liste sait s'étendre (`append()`), un dictionnaire sait retirer un élément (`pop()`).

Ensuite, il est important de distinguer **le typage dynamique** et **le typage statique**. Dans un langage à typage statique (comme C, Java ou Rust), le type doit être défini avant l'exécution, et il ne peut pas changer. Dans un langage à typage dynamique comme Python, c'est à l'exécution que le type est connu.

Ainsi :

```
x = 10
x = "dix"
```

est parfaitement valide en Python, car la variable `x` n'a pas de type fixe : elle **référence un objet** (d'abord un entier, puis une chaîne). Autrement dit, en Python, **les variables ne sont pas typées**, mais **les valeurs le sont**.

C'est une différence essentielle : quand on écrit `x = 10`, `x` ne contient pas 10 directement, il pointe vers un objet `int` en mémoire qui contient la valeur 10.

Le typage de Python est donc :

- **dynamique**, car on peut changer le type d'une variable à tout moment,
- **fort**, car Python ne convertit pas automatiquement les types incompatibles.

Par exemple :

```
print("5" + 5)
```

provoque une erreur, car Python refuse de mélanger une chaîne et un entier sans conversion explicite (`int("5")` ou `str(5)`).

Le typage en Python repose donc sur la **cohérence des opérations**. Chaque type sait ce qu'il peut faire et avec qui il peut interagir. Cela permet d'écrire du code flexible, mais aussi de se tromper facilement si on ne reste pas attentif.

C'est pourquoi les **annotations de type** ont été introduites : elles servent à documenter notre intention. Elles ne changent rien à l'exécution, mais elles rendent le code plus lisible et plus sûr :

```
def aire_rectangle(longueur: float, largeur: float) -> float:  
    return longueur * largeur
```

Grâce à cela :

- les outils comme VS Code peuvent signaler les erreurs,
- on sait immédiatement ce que la fonction attend,
- et le code devient plus clair pour celui qui le relira.

Enfin, on peut évoquer deux idées importantes :

1. **Les types composés** : les listes, tuples et dictionnaires peuvent eux-mêmes être typés :

```
notes: list[int] = [12, 15, 18]  
eleve: dict[str, int] = {"math": 14, "anglais": 16}
```

Cela permet de préciser le type des éléments qu'ils contiennent.

2. **La compatibilité des types** : Python accepte souvent plusieurs types compatibles dans une fonction. Par exemple :

```
def afficher(x: str | int) -> None:  
    print(f"Valeur : {x}")
```

Ici, `x` peut être une chaîne **ou** un entier. Cette écriture utilise `|` pour dire « ou ».

En somme, le typage en Python, c'est un équilibre entre liberté et rigueur. Le langage nous laisse faire ce qu'on veut, mais il nous encourage à **exprimer nos intentions** à travers les annotations, pour que le code reste fiable, clair et rapide à comprendre — surtout quand on travaille à plusieurs.