Introduction

Python étant un langage **interprété**, il est typé dynamiquement. Ce qui signifie que les types sont associés aux valeurs au moment de l'exécution, mais l'annotation de type permet d'introduire une forme de typage statique facultatif pour améliorer la qualité et la lisibilité du code. Cette pratique augmente considérablement la qualité et la lisibilité de votre code, et est donc une pratique répandue en milieu professionnel.

Les type sont associés aux variables, aux paramètres de fonction et aux valeurs de retour.

```
# On précise ici qu'on attend deux integers, et que la fonction retourne un
integer
def add(a: int, b: int) -> int:
    return a + b

x: int = 4
y: int = 8
z: str = "coucou"

print(add(x, y))
print(add(x, z)) # soulèvera une erreur explicite
```

Elle n'affecte pas l'exécution du code, mais elle peut être utilisée par des outils (comme SonarLint sur VSC) et des IDE pour détecter des erreurs potentielles et améliorer la documentation du code. Elle permet aussi de profiter de l'auto-complétions dans la plupart des IDE modernes facilitant grandement le développement et empêchant un bon nombre de bugs.

Nouveautés de la 3.10

Depuis la toute récente 3.10, les annotations de types ont été améliorées.

```
# Synthaxe avant 3.10
from typing import Union
ma_liste: list[Union[int, str]] = [2, "coucou", 1, 76, "test"]

# 3.10: choisir plusieurs types possibles avec l'opérateur | (pipe)
ma_liste: list[int | str] = [2, "coucou", 1, 76, "test"]

# Aliasing de type
Nombre = int | float

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

Entiers (int)

```
age: int = 25
```

Flottants (float)

```
prix: float = 10.99
```

Chaînes de caractères (str)

```
nom: str = "Alice"
```

Booléens (bool)

```
est_actif: bool = True
```

Listes

```
from typing import List

nombres: List[int] = [1, 2, 3]
```

Tuples

```
from typing import Tuple

coordonnees: Tuple[float, float] = (2.5, 3.0)
```

Typage de Dictionnaires

```
from typing import Dict

informations: Dict[str, str] = {"nom": "Bob", "ville": "Paris"}
```

Typage de Paramètres et de Retours de Fonctions

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

Fonctions avec des Paramètres Optionnels

```
from typing import Optional

def afficher_message(age: int, nom: Optional[str] = None) -> None:
    if nom:
        print(f"{nom} a {age} ans.")
    else:
        print(f"Cet individu a {age} ans.")
```

Typage d'Objets Personnalisés

Si vous avez défini une classe, vous pouvez indiquer le type de l'objet qui sera créé à partir de cette classe.

```
class Personne:
    def __init__(self, nom: str, age: int):
        self.nom = nom
        self.age = age
```

Vous pouvez annoter le type de l'objet créé avec cette classe comme suit :

```
def creer_personne(nom: str, age: int) -> Personne:
    return Personne(nom, age)
```