NOTE DE SYNTHÈSE

Annotations de types en Python

I Introduction : pourquoi annoter les types ?

Python est un langage *dynamiquement typé* : le type d'une variable n'est pas fixé à l'avance, mais déterminé à l'exécution. Cela rend le langage très souple, mais peut nuire à la lisibilité et à la rigueur dans des programmes à visée algorithmique ou mathématique.

Les **annotations de types** (ou *type hints*) permettent d'indiquer, *sans modifier le comportement du programme*, le type attendu :

- des paramètres d'une fonction ;
- de la valeur qu'elle renvoie ;
- plus généralement, de la structure des données manipulées.

Objectif recherché :

- rendre le code plus **lisible** et plus **proche des notations mathématiques** ;
- expliciter les hypothèses sur les données (nature, dimension, structure) ;
- faciliter le raisonnement algorithmique et la relecture du code ;
- préparer à des outils de vérification statique utilisés dans l'enseignement supérieur.

Point fondamental : les annotations de types **ne sont pas des contraintes d'exécution**. Python n'effectue aucune vérification automatique des types à l'exécution.

II Principe général et syntaxe

Une annotation de type s'écrit directement dans la déclaration d'une fonction.

Principe :

- le type d'un paramètre est indiqué après son nom, séparé par `:` ;
- le type de retour est indiqué après `->` ;
- le corps de la fonction n'est pas modifié.

On peut lire une fonction annotée comme :

> « Cette fonction *attend* des objets de tels types et *renvoie* un objet de tel type. »

Ainsi on écrira :

```
def ma_fonction(var1:type1,var2:type2) -> type de sortie:
```

Ainsi par exemple, pour l'algorithme de recherche dichotomique dans un tableau trié, nous écririons :

```
def bin_search(L:list,x:int) -> bool:
    g = 0
    d = len(L)-1
    while g <= d:
        m=(g+d)//2 # milieu ou partie entière du milieu
        if L[m] == x:
            return True
        elif L[m]<x:
            g=m+1
        else:
            d=m-1
    return False
```

A noter, que, si la fonction `bin_search` est par exemple définie de la sorte, alors :
`bin_search.__annotations__` permet d'afficher la signature de la fonction `bin_search`.

On retiendra que cette annotation n'est pas évaluée par Python lors d'une exécution de la fonction, mais est considérée comme un commentaire. Notamment, lors d'un appel de la fonction aucun test ne sera effectué pour vérifié l'appartenance des types des arguments aux types spécifiés — il s'agit vraiment juste d'une « *annotation* ».

III Les types à connaître

1. Types numériques, booléens, chaînes de caractères

Types primitifs à connaître :

`int`, `float`, `bool`, `str`.

Exemples :

```
def carre(x: float) -> float:
    return x * x
```

```
def est_pair(n: int) -> bool:
    return n % 2 == 0
```

2. Listes et tuples

Une liste de réels s'annotera : `list[float]`.

Exemple :

```
def moyenne(L: list[float]) -> float:
    return sum(L) / len(L)
```

Un tuple permet de regrouper un nombre fixé d'éléments, éventuellement de types différents.

Exemple :

```
def point() -> tuple[float, str]:
    return (1.0, ''bonjour'')
```

3. Dictionnaires

Un dictionnaire associe des *clés* à des *valeurs*. On peut l'interpréter comme une **application finie** :

$$\text{clé} \longmapsto \text{valeur}.$$

Le type général est :

`dict[TypeCle, TypeValeur]`.

Exemple de l'histogramme :

```
  def effectifs(L: list[int]) -> dict[int, int]:
    d = {}
    for x in L:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
    return d
```

4. Tableaux NumPy

NumPy ne fournit pas encore de typage mathématique fin. On utilise le type générique `np.ndarray`.

Exemple :

```
import numpy as np

def is_connexe(G: np.ndarray) -> bool:
    ...
```

5. Fonctions comme objets

Les fonctions peuvent être passées en argument. On utilise alors le type `Callable`.

Exemple :

```
from typing import Callable

def f(x : float) -> float
  return x**2

def appliquer(f: Callable[[float], float], x: float) -> float:
    return f(x)
```

6. Types optionnels

Lorsqu'une fonction peut ne rien renvoyer (`None`) :

Exemple :

```
from typing import Optional

def inverse(x: float) -> Optional[float]:
    if x == 0:
        return None
    return 1 / x
```

Ce sera en particulier le cas pour une fonction qui sert à afficher un graphe ou à afficher à l'aide de `print`.

« Reasonable-looking algorithms can easily be incorrect. Al- gorithm correctness is a property that must be carefully demonstrated. »

Steven Skiena dans *The Algorithm Design Manual*, Springer (2008).