# Les annotations de fonctions Python

---

C'est quelque chose de bien connu : sous Python, pas besoin de spécifier les **types des variables**. Cette flexibilité permet de convertir très facilement des variables, ce qui est souvent le cas lorsque l'on manipule des données.
En revanche, lorsque l'on défini des fonctions, on aimerait insister sur le fait que certains paramètres soient d'un certain type, pour éviter des incohérences, voir des erreurs dans la suite du programme. Pour faciliter la lecture et la documentation des fonctions, la PEP 3107 autorise l'annotation des types des paramètres d'une fonction. Voyons en détail comment cela fonctionne.

## Les annotations de fonctions

Prenons la fonction `longueur` qui va calculer la longueur d'une chaîne de caractère.

```
1def longueur(x):
2  return len(x)
3
4longueur("Hello !")
```

Cette fonction requiert le paramètre `x` qui est une chaîne de caractères, et retourne un entier qui correspond à la longueur de cette même chaîne.
Comment spécifier ces deux types ? C'est là que les **annotations de fonctions** interviennent. Chaque paramètre va être suivi de son type, et la nous allons pouvoir spécifier la type retourné par la fonction avec `->`.

```
1def longueur(x: str) -> int:
2  return len(x)
3
4longueur("Hello !")
```

Cette fonction se lit : on s'attends à avoir un `x` de type chaîne de caractères, et retourne un entier.
L'avantage de cette représentation, c'est que cela **est beaucoup plus clair** sur le type attendu par la fonction, puisque en un coup d'oeil, on sait exactement quels types doivent être passés en argument.

Attention néamoins, et comme défini dans la PEP, l'annotation de fonctions **est optionnelle** : même si lors de l'appel de la fonction, le paramètre n'est pas du même type que celui inscrit dans la définition, Python ne générera pas d'erreurs.

```
1longueur([-1, 2, 3, 4])
```

Ce que cela signifie, c'est que c'est **à la charge du développeur** de vérifier que les types des paramètres sont bien conformes.

```
1def longueur(x: str) -> int:
2  if not isinstance(x, str):  # isistance retourne Faux si le
paramètre x n'est pas une chaîne de caractère
3    return "Erreur ! Pas un str"
4  return len(x)
5
6print(longueur("Hello !"))
7print(longueur([-1, 2, 3, 4]))
```

Ainsi, avec des vérifications de type comme la fonction `isinstance`, il faudra toujours vérifier le type des paramètres dans le cadre d'exécution d'une fonction critique. Les annotations de fonctions sont purement à titre d'amélioration de la lisibilité du code. Notons, au passage, qu'il y a beaucoup de types prêt-à-l'emploi dans les annotations de fonctions :

- `str`, `int`, `float` pour les types classiques.
- `List`, `Tuple`, `Set`, `Map` pour les listes, tuples, ensembles and maps.
- `Iterable` pour les itérateurs.
- `Union` pour spécifier plusieurs types possibles.
- `Optional` équivalent à `Union[T, None]`.
- `Callable` pour les fonctions en paramètres.
- `Any` pour n'importe quel type.

Par exemple, la fonction suivante va appliquer une autre fonction `f` à chaque élément d'un itérable (comme une liste).

```
1from typing import Callable, Any, Iterable, List
2
3def vectorize(f: Callable[[Any], Any], l: Iterable[Any]) ->
List[Any]:
4  output = []
5  for elem in l:
6    output.append(f(elem))
7  return output
```

Comment s'interprète ces annotations ?

- Tout d'abord, on s'attend à ce que `f` soit un *callable* (une fonction). Le type `Callable` nécessite deux informations : la liste des types des paramètres, et le type retourné. Ici, il n'y a qu'un seul paramètre donc le type est quelconque, matérialisé par `[Any]`. Le type retourné par la fonction `f` est lui aussi `Any`.
- Ensuite, la second argument `l` doit être itérable (liste, générateur, ...) dont les éléments sont de n'improte quel type.
- Enfin, la résultat retourné est de type `List[Any]`, c'est-à-dire une liste Python dont les éléments sont de type quelconque.

Testons maintenant cette fonction.

```
def carre(x):
  return x * x

def lettre(x):
  return ['a', 'b', 'c', 'd', 'e'][x - 1]

liste = [1, 2, 3, 4, 5]

print(vectorize(carre, liste))
print(vectorize(lettre, liste))
```

Les deux appels à la fonction `vectorize` produisent le résultat attendu.

❓ Quel est donc l'intérêt d'annoter les fonctions ?

Le principal intérêt, comme évoqué plus haut, est la **lisibilité du code**. Lorsque l'on collabore à plusieurs sur un projet, il est important de bien comprendre le comportement de chaque fonction, qu'est-ce qui y est attendu et qu'est-ce qui est retourné comme résultat.
C'est encore plus le cas lorsque l'on travaille sur des projets Data où l'on manipule des données de natures très différentes (DataFrames, tenseurs, textes).