Introduction aux Types Python

Python supporte des annotations de type (ou *type hints*) optionnelles.

Ces annotations de type constituent une syntaxe spéciale qui permet de déclarer le type d'une variable.

En déclarant les types de vos variables, cela permet aux différents outils comme les éditeurs de texte d'offrir un meilleur support.

Ce chapitre n'est qu'un **tutoriel rapide / rappel** sur les annotations de type Python. Seulement le minimum nécessaire pour les utiliser avec **FastAPI** sera couvert... ce qui est en réalité très peu.

FastAPI est totalement basé sur ces annotations de type, qui lui donnent de nombreux avantages.

Mais même si vous n'utilisez pas ou n'utiliserez jamais **FastAPI**, vous pourriez bénéficier d'apprendre quelques choses sur ces dernières.

!!! note Si vous êtes un expert Python, et que vous savez déjà **tout** sur les annotations de type, passez au chapitre suivant.

Motivations

Prenons un exemple simple :

```
{!../../../docs_src/python_types/tutorial001.py!}
```

Exécuter ce programe affiche :

```
John Doe
```

La fonction :

- Prend un `first_name` et un `last_name`.
- Convertit la première lettre de chaque paramètre en majuscules grâce à `title()`.
- Concatène les résultats avec un espace entre les deux.

```
{!../../../docs_src/python_types/tutorial001.py!}
```

Limitations

C'est un programme très simple.

Mais maintenant imaginez que vous l'écriviez de zéro.

À un certain point vous auriez commencé la définition de la fonction, vous aviez les paramètres prêts.

Mais vous aviez besoin de "cette méthode qui convertit la première lettre en majuscule".

Était-ce `upper` ? `uppercase` ? `first_uppercase` ? `capitalize` ?

Vous essayez donc d'utiliser le vieil ami du programmeur, l'auto-complétion de l'éditeur.

Vous écrivez le premier paramètre, `first_name`, puis un point (`.`) et appuyez sur `Ctrl+Espace` pour déclencher l'auto-complétion.

Mais malheureusement, rien d'utile n'en résulte :

Ajouter des types

Modifions une seule ligne de la version précédente.

Nous allons changer seulement cet extrait, les paramètres de la fonction, de :

```
    first_name, last_name
```

à :

```
    first_name: str, last_name: str
```

C'est tout.

Ce sont des annotations de types :

```
{!../../../docs_src/python_types/tutorial002.py!}
```

À ne pas confondre avec la déclaration de valeurs par défaut comme ici :

```
    first_name="john", last_name="doe"
```

C'est une chose différente.

On utilise un deux-points (`:`), et pas un égal (`=`).

Et ajouter des annotations de types ne crée normalement pas de différence avec le comportement qui aurait eu lieu si elles n'étaient pas là.

Maintenant, imaginez que vous êtes en train de créer cette fonction, mais avec des annotations de type cette fois.

Au même moment que durant l'exemple précédent, vous essayez de déclencher l'auto-complétion et vous voyez :

Vous pouvez donc dérouler les options jusqu'à trouver la méthode à laquelle vous pensiez.

Plus de motivations

Cette fonction possède déjà des annotations de type :

```
{!../../../docs_src/python_types/tutorial003.py!}
```

Comme l'éditeur connaît le type des variables, vous n'avez pas seulement l'auto-complétion, mais aussi de la détection d'erreurs :

Maintenant que vous avez connaissance du problème, convertissez `age` en chaine de caractères grâce à `str(age)` :

```
{!../../../docs_src/python_types/tutorial004.py!}
```

Déclarer des types

Vous venez de voir là où les types sont généralement déclarés : dans les paramètres de fonctions.

C'est aussi ici que vous les utiliseriez avec **FastAPI**.

Types simples

Vous pouvez déclarer tous les types de Python, pas seulement `str` .

Comme par exemple :

- `int`
- `float`
- `bool`
- `bytes`

```
{!../../../docs_src/python_types/tutorial005.py!}
```

Types génériques avec des paramètres de types

Il existe certaines structures de données qui contiennent d'autres valeurs, comme `dict` , `list` , `set` et `tuple` . Et les valeurs internes peuvent elles aussi avoir leurs propres types.

Pour déclarer ces types et les types internes, on utilise le module standard de Python `typing` .

Il existe spécialement pour supporter ces annotations de types.

`List`

Par exemple, définissons une variable comme `list` de `str` .

Importez `List` (avec un `L` majuscule) depuis `typing` .

```
{!../../../docs_src/python_types/tutorial006.py!}
```

Déclarez la variable, en utilisant la syntaxe des deux-points (`:`).

Et comme type, mettez `List` .

Les listes étant un type contenant des types internes, mettez ces derniers entre crochets (`[` , `]`) :

```
{!../../../docs_src/python_types/tutorial006.py!}
```

!!! tip "Astuce" Ces types internes entre crochets sont appelés des "paramètres de type".

```
Ici, `str` est un paramètre de type passé à `List`.
```

Ce qui signifie : "la variable `items` est une `list`, et chacun de ses éléments a pour type `str`.

En faisant cela, votre éditeur pourra vous aider, même pendant que vous traitez des éléments de la liste.

Sans types, c'est presque impossible à réaliser.

Vous remarquerez que la variable `item` n'est qu'un des éléments de la list `items`.

Et pourtant, l'éditeur sait qu'elle est de type `str` et pourra donc vous aider à l'utiliser.

`Tuple` et `Set`

C'est le même fonctionnement pour déclarer un `tuple` ou un `set` :

```
{!../../../docs_src/python_types/tutorial007.py!}
```

Dans cet exemple :

- La variable `items_t` est un `tuple` avec 3 éléments, un `int`, un deuxième `int`, et un `str`.
- La variable `items_s` est un `set`, et chacun de ses éléments est de type `bytes`.

`Dict`

Pour définir un `dict`, il faut lui passer 2 paramètres, séparés par une virgule (`,`).

Le premier paramètre de type est pour les clés et le second pour les valeurs du dictionnaire (`dict`).

```
{!../../../docs_src/python_types/tutorial008.py!}
```

Dans cet exemple :

- La variable `prices` est de type `dict` :
    - Les clés de ce dictionnaire sont de type `str`.
    - Les valeurs de ce dictionnaire sont de type `float`.

`Optional`

Vous pouvez aussi utiliser `Optional` pour déclarer qu'une variable a un type, comme `str` mais qu'il est "optionnel" signifiant qu'il pourrait aussi être `None`.

```
{!../../../docs_src/python_types/tutorial009.py!}
```

Utiliser `Optional[str]` plutôt que `str` permettra à l'éditeur de vous aider à détecter les erreurs où vous supposeriez qu'une valeur est toujours de type `str`, alors qu'elle pourrait aussi être `None`.

Types génériques

Les types qui peuvent contenir des paramètres de types entre crochets, comme :

- `List`
- `Tuple`
- `Set`
- `Dict`
- `Optional`
- ...et d'autres.

sont appelés des **types génériques** ou **Generics**.

Classes en tant que types

Vous pouvez aussi déclarer une classe comme type d'une variable.

Disons que vous avez une classe `Person`, avec une variable `name` :

```
{!../../../docs_src/python_types/tutorial010.py!}
```

Vous pouvez ensuite déclarer une variable de type `Person` :

```
{!../../../docs_src/python_types/tutorial010.py!}
```

Et vous aurez accès, encore une fois, au support complet offert par l'éditeur :

Les modèles Pydantic

Pydantic est une bibliothèque Python pour effectuer de la validation de données.

Vous déclarez la forme de la donnée avec des classes et des attributs.

Chaque attribut possède un type.

Puis vous créez une instance de cette classe avec certaines valeurs et **Pydantic** validera les valeurs, les convertira dans le type adéquat (si c'est nécessaire et possible) et vous donnera un objet avec toute la donnée.

Ainsi, votre éditeur vous offrira un support adapté pour l'objet résultant.

Extrait de la documentation officielle de **Pydantic** :

```
{!../../../docs_src/python_types/tutorial011.py!}
```

!!! info Pour en savoir plus à propos de Pydantic, allez jeter un coup d'oeil à sa documentation.

FastAPI est basé entièrement sur **Pydantic**.

Vous verrez bien plus d'exemples de son utilisation dans Tutoriel - Guide utilisateur{.internal-link target=_blank}.

Les annotations de type dans FastAPI

FastAPI utilise ces annotations pour faire différentes choses.

Avec **FastAPI**, vous déclarez des paramètres grâce aux annotations de types et vous obtenez :

- **du support de l'éditeur**
- **de la vérification de types**

...et **FastAPI** utilise ces mêmes déclarations pour :

- **Définir les prérequis** : depuis les paramètres de chemins des requêtes, les entêtes, les corps, les dépendances, etc.
- **Convertir des données** : depuis la requête vers les types requis.
- **Valider des données** : venant de chaque requête :
    - Générant automatiquement des **erreurs** renvoyées au client quand la donnée est invalide.
- **Documenter** l'API avec OpenAPI :
    - ce qui ensuite utilisé par les interfaces utilisateur automatiques de documentation interactive.

Tout cela peut paraître bien abstrait, mais ne vous inquiétez pas, vous verrez tout ça en pratique dans Tutoriel - Guide utilisateur{.internal-link target=_blank}.

Ce qu'il faut retenir c'est qu'en utilisant les types standard de Python, à un seul endroit (plutôt que d'ajouter plus de classes, de décorateurs, etc.), **FastAPI** fera une grande partie du travail pour vous.

!!! info Si vous avez déjà lu le tutoriel et êtes revenus ici pour voir plus sur les types, une bonne ressource est la "cheat sheet" de `mypy` .