Introducción a los Tipos de Python

Python 3.6+ tiene soporte para "type hints" opcionales.

Estos **type hints** son una nueva sintáxis, desde Python 3.6+, que permite declarar el tipo de una variable.

Usando las declaraciones de tipos para tus variables, los editores y otras herramientas pueden proveerte un soporte mejor.

Este es solo un **tutorial corto** sobre los Python type hints. Solo cubre lo mínimo necesario para usarlos con **FastAPI**... realmente es muy poco lo que necesitas.

Todo **FastAPI** está basado en estos type hints, lo que le da muchas ventajas y beneficios.

Pero, así nunca uses **FastAPI** te beneficiarás de aprender un poco sobre los type hints.

!!! note "Nota" Si eres un experto en Python y ya lo sabes todo sobre los type hints, salta al siguiente capítulo.

Motivación

Comencemos con un ejemplo simple:

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

Llamar este programa nos muestra el siguiente output:

```
John Doe
```

La función hace lo siguiente:

- Toma un `first_name` y un `last_name`.
- Convierte la primera letra de cada uno en una letra mayúscula con `title()`.
- Las concatena con un espacio en la mitad.

`Python hl_lines="2" {!../../../docs_src/python_types/tutorial001.py!}`

Edítalo

Es un programa muy simple.

Ahora, imagina que lo estás escribiendo desde ceros.

En algún punto habrías comenzado con la definición de la función, tenías los parámetros listos...

Pero, luego tienes que llamar "ese método que convierte la primera letra en una mayúscula".

Era `upper`? O era `uppercase`? `first_uppercase`? `capitalize`?

Luego lo intentas con el viejo amigo de los programadores, el autocompletado del editor.

Escribes el primer parámetro de la función `first_name`, luego un punto (`.`) y luego presionas `Ctrl+Space` para iniciar el autocompletado.

Tristemente, no obtienes nada útil:

Añade tipos

Vamos a modificar una única línea de la versión previa.

Vamos a cambiar exactamente este fragmento, los parámetros de la función, de:

```
first_name, last_name
```

a:

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

Eso es todo.

Esos son los "type hints":

```
Python hl_lines="1" {!../../../docs_src/python_types/tutorial002.py!}
```

No es lo mismo a declarar valores por defecto, como sería con:

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

Es algo diferente.

Estamos usando los dos puntos (`:`), no un símbolo de igual (`=`).

Añadir los type hints normalmente no cambia lo que sucedería si ellos no estuviesen presentes.

Pero ahora imagina que nuevamente estás creando la función, pero con los type hints.

En el mismo punto intentas iniciar el autocompletado con `Ctrl+Space` y ves:

Con esto puedes moverte hacia abajo viendo las opciones hasta que encuentras una que te suene:

Más motivación

Mira esta función que ya tiene type hints:

```
Python hl_lines="1" {!../../../docs_src/python_types/tutorial003.py!}
```

Como el editor conoce el tipo de las variables no solo obtienes autocompletado, si no que también obtienes chequeo de errores:

Ahora que sabes que tienes que arreglarlo convierte `age` a un string con `str(age)`:

```
Python hl_lines="2" {!../../../docs_src/python_types/tutorial004.py!}
```

Declarando tipos

Acabas de ver el lugar principal para declarar los type hints. Como parámetros de las funciones.

Este es también el lugar principal en que los usarías con **FastAPI**.

Tipos simples

Puedes declarar todos los tipos estándar de Python, no solamente `str`.

Por ejemplo, puedes usar:

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

```
Python hl_lines="1" {!../../../docs_src/python_types/tutorial005.py!}
```

Tipos con sub-tipos

Existen algunas estructuras de datos que pueden contener otros valores, como `dict`, `list`, `set` y `tuple`. Los valores internos pueden tener su propio tipo también.

Para declarar esos tipos y sub-tipos puedes usar el módulo estándar de Python `typing`.

Él existe específicamente para dar soporte a este tipo de type hints.

Listas Por ejemplo, vamos a definir una variable para que sea una `list` compuesta de `str`.

De `typing`, importa `List` (con una `L` mayúscula):

```
Python hl_lines="1" {!../../../docs_src/python_types/tutorial006.py!}
```

Declara la variable con la misma sintáxis de los dos puntos (`:`).

Pon `List` como el tipo.

Como la lista es un tipo que permite tener un "sub-tipo" pones el sub-tipo en corchetes `[]`:

```
Python hl_lines="4" {!../../../docs_src/python_types/tutorial006.py!}
```

Esto significa: la variable `items` es una `list` y cada uno de los ítems en esta lista es un `str`.

Con esta declaración tu editor puede proveerte soporte inclusive mientras está procesando ítems de la lista.

Sin tipos el autocompletado en este tipo de estructura es casi imposible de lograr:

Observa que la variable `item` es unos de los elementos en la lista `items`.

El editor aún sabe que es un `str` y provee soporte para ello.

Tuples y Sets Harías lo mismo para declarar `tuple`s y `set`s:

```
Python hl_lines="1  4" {!../../../docs_src/python_types/tutorial007.py!}
```

Esto significa:

- La variable `items_t` es un `tuple` con 3 ítems, un `int`, otro `int`, y un `str`.
- La variable `items_s` es un `set` y cada uno de sus ítems es de tipo `bytes`.

Diccionarios (Dicts) Para definir un `dict` le pasas 2 sub-tipos separados por comas.

El primer sub-tipo es para los keys del `dict`.

El segundo sub-tipo es para los valores del `dict`:

```
Python hl_lines="1  4" {!../../../docs_src/python_types/tutorial008.py!}
```

Esto significa:

- La variable `prices` es un `dict`:
  - Los keys de este `dict` son de tipo `str` (Digamos que son el nombre de cada ítem).
  - Los valores de este `dict` son de tipo `float` (Digamos que son el precio de cada ítem).

Clases como tipos

También puedes declarar una clase como el tipo de una variable.

Digamos que tienes una clase `Person`con un nombre:

```
Python hl_lines="1-3" {!../../../docs_src/python_types/tutorial009.py!}
```

Entonces puedes declarar una variable que sea de tipo `Person`:

```
Python hl_lines="6" {!../../../docs_src/python_types/tutorial009.py!}
```

Una vez más tendrás todo el soporte del editor:

Modelos de Pydantic

Pydantic es una library de Python para llevar a cabo validación de datos.

Tú declaras la "forma" de los datos mediante clases con atributos.

Cada atributo tiene un tipo.

Luego creas un instance de esa clase con algunos valores y Pydantic validará los valores, los convertirá al tipo apropiado (si ese es el caso) y te dará un objeto con todos los datos.

Y obtienes todo el soporte del editor con el objeto resultante.

Tomado de la documentación oficial de Pydantic:

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

!!! info "Información" Para aprender más sobre Pydantic mira su documentación.

FastAPI está todo basado en Pydantic.

Vas a ver mucho más de esto en práctica en el Tutorial - User Guide.

Type hints en FastAPI

FastAPI aprovecha estos type hints para hacer varias cosas.

Con **FastAPI** declaras los parámetros con type hints y obtienes:

- **Soporte en el editor**.
- **Type checks**.

. . . y **FastAPI** usa las mismas declaraciones para:

- **Definir requerimientos**: desde request path parameters, query parameters, headers, bodies, dependencies, etc.
- **Convertir datos**: desde el request al tipo requerido.
- **Validar datos**: viniendo de cada request:
  - Generando **errores automáticos** devueltos al cliente cuando los datos son inválidos.
- **Documentar** la API usando OpenAPI:
  - que en su caso es usada por las interfaces de usuario de la documentación automática e interactiva.

Puede que todo esto suene abstracto. Pero no te preocupes que todo lo verás en acción en el Tutorial - User Guide.

Lo importante es que usando los tipos de Python estándar en un único lugar (en vez de añadir más clases, decorator, etc.) **FastAPI** hará mucho del trabajo por ti.

!!! info "Información" Si ya pasaste por todo el tutorial y volviste a la sección de los tipos, una buena referencia es la "cheat sheet" de `mypy`.