Python Types Intro

Python has support for optional "type hints".

These **"type hints"** are a special syntax that allow declaring the type of a variable.

By declaring types for your variables, editors and tools can give you better support.

This is just a **quick tutorial / refresher** about Python type hints. It covers only the minimum necessary to use them with **FastAPI**... which is actually very little.

FastAPI is all based on these type hints, they give it many advantages and benefits.

But even if you never use **FastAPI**, you would benefit from learning a bit about them.

!!! note If you are a Python expert, and you already know everything about type hints, skip to the next chapter.

Motivation

Let's start with a simple example:

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

Calling this program outputs:

```
John Doe
```

The function does the following:

- Takes a `first_name` and `last_name`.
- Converts the first letter of each one to upper case with `title()`.
- Concatenates them with a space in the middle.

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

Edit it

It's a very simple program.

But now imagine that you were writing it from scratch.

At some point you would have started the definition of the function, you had the parameters ready...

But then you have to call "that method that converts the first letter to upper case".

Was it `upper`? Was it `uppercase`? `first_uppercase`? `capitalize`?

Then, you try with the old programmer's friend, editor autocompletion.

You type the first parameter of the function, `first_name`, then a dot (`.`) and then hit `Ctrl+Space` to trigger the completion.

But, sadly, you get nothing useful:

Add types

Let's modify a single line from the previous version.

We will change exactly this fragment, the parameters of the function, from:

```
    first_name, last_name
```

to:

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

That's it.

Those are the "type hints":

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

That is not the same as declaring default values like would be with:

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

It's a different thing.

We are using colons (`:`), not equals (`=`).

And adding type hints normally doesn't change what happens from what would happen without them.

But now, imagine you are again in the middle of creating that function, but with type hints.

At the same point, you try to trigger the autocomplete with `Ctrl+Space` and you see:

With that, you can scroll, seeing the options, until you find the one that "rings a bell":

More motivation

Check this function, it already has type hints:

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

Because the editor knows the types of the variables, you don't only get completion, you also get error checks:

Now you know that you have to fix it, convert `age` to a string with `str(age)`:

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

Declaring types

You just saw the main place to declare type hints. As function parameters.

This is also the main place you would use them with **FastAPI**.

Simple types

You can declare all the standard Python types, not only `str`.

You can use, for example:

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

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

Generic types with type parameters

There are some data structures that can contain other values, like `dict`, `list`, `set` and `tuple`. And the internal values can have their own type too.

These types that have internal types are called "**generic**" types. And it's possible to declare them, even with their internal types.

To declare those types and the internal types, you can use the standard Python module `typing`. It exists specifically to support these type hints.

Newer versions of Python

The syntax using `typing` is **compatible** with all versions, from Python 3.6 to the latest ones, including Python 3.9, Python 3.10, etc.

As Python advances, **newer versions** come with improved support for these type annotations and in many cases you won't even need to import and use the `typing` module to declare the type annotations.

If you can chose a more recent version of Python for your project, you will be able to take advantage of that extra simplicity. See some examples below.

List

For example, let's define a variable to be a `list` of `str`.

=== "Python 3.6 and above"

```
From `typing`, import `List` (with a capital `L`):

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

Declare the variable, with the same colon (`:`) syntax.

As the type, put the `List` that you imported from `typing`.

As the list is a type that contains some internal types, you put them in square
brackets:

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

=== "Python 3.9 and above"

```
Declare the variable, with the same colon (`:`) syntax.

As the type, put `list`.

As the list is a type that contains some internal types, you put them in square
brackets:

```Python hl_lines="1"
{!> ../../../docs_src/python_types/tutorial006_py39.py!}
```
```

!!! info Those internal types in the square brackets are called "type parameters".

```
In this case, `str` is the type parameter passed to `List` (or `list` in Python 3.9
and above).
```

That means: "the variable `items` is a `list`, and each of the items in this list is a `str`".

!!! tip If you use Python 3.9 or above, you don't have to import `List` from `typing`, you can use the same regular `list` type instead.

By doing that, your editor can provide support even while processing items from the list:

Without types, that's almost impossible to achieve.

Notice that the variable `item` is one of the elements in the list `items`.

And still, the editor knows it is a `str`, and provides support for that.

Tuple and Set

You would do the same to declare `tuple`s and `set`s:

=== "Python 3.6 and above"

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

=== "Python 3.9 and above"

```Python hl_lines="1"
{!> ../../../docs_src/python_types/tutorial007_py39.py!}
```

This means:

- The variable `items_t` is a `tuple` with 3 items, an `int`, another `int`, and a `str`.
- The variable `items_s` is a `set`, and each of its items is of type `bytes`.

Dict

To define a `dict`, you pass 2 type parameters, separated by commas.

The first type parameter is for the keys of the `dict`.

The second type parameter is for the values of the `dict`:

=== "Python 3.6 and above"

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

=== "Python 3.9 and above"

```Python hl_lines="1"
{!> ../../../docs_src/python_types/tutorial008_py39.py!}
```

This means:

- The variable `prices` is a `dict`:
    - The keys of this `dict` are of type `str` (let's say, the name of each item).
    - The values of this `dict` are of type `float` (let's say, the price of each item).

Union

You can declare that a variable can be any of **several types**, for example, an `int` or a `str`.

In Python 3.6 and above (including Python 3.10) you can use the `Union` type from `typing` and put inside the square brackets the possible types to accept.

In Python 3.10 there's also an **alternative syntax** were you can put the possible types separated by a vertical bar (`|`).

=== "Python 3.6 and above"

```Python hl_lines="1  4"
{!> ../../../docs_src/python_types/tutorial008b.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="1"
{!> ../../../docs_src/python_types/tutorial008b_py310.py!}
```

In both cases this means that `item` could be an `int` or a `str`.

Possibly `None`

You can declare that a value could have a type, like `str`, but that it could also be `None`.

In Python 3.6 and above (including Python 3.10) you can declare it by importing and using `Optional` from the `typing` module.

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

Using `Optional[str]` instead of just `str` will let the editor help you detecting errors where you could be assuming that a value is always a `str`, when it could actually be `None` too.

`Optional[Something]` is actually a shortcut for `Union[Something, None]`, they are equivalent.

This also means that in Python 3.10, you can use `Something | None`:

=== "Python 3.6 and above"

```Python hl_lines="1  4"
{!> ../../../docs_src/python_types/tutorial009.py!}
```

=== "Python 3.6 and above - alternative"

```Python hl_lines="1  4"
{!> ../../../docs_src/python_types/tutorial009b.py!}
```

=== "Python 3.10 and above"

```Python hl_lines="1"
{!> ../../../docs_src/python_types/tutorial009_py310.py!}
```

Generic types

These types that take type parameters in square brackets are called **Generic types** or **Generics**, for example:

=== "Python 3.6 and above"

```
* `List`
* `Tuple`
* `Set`
* `Dict`
* `Union`
* `Optional`
* ...and others.
```

=== "Python 3.9 and above"

```
You can use the same builtin types as generics (with square brakets and types inside):

* `list`
* `tuple`
* `set`
* `dict`

And the same as with Python 3.6, from the `typing` module:

* `Union`
* `Optional`
* ...and others.
```

=== "Python 3.10 and above"

```
You can use the same builtin types as generics (with square brakets and types inside):

* `list`
* `tuple`
* `set`
* `dict`

And the same as with Python 3.6, from the `typing` module:

* `Union`
* `Optional` (the same as with Python 3.6)
* ...and others.

In Python 3.10, as an alternative to using the generics `Union` and `Optional`, you
can use the <abbr title='also called "bitwise or operator", but that meaning is not
relevant here'>vertical bar (`|`)</abbr> to declare unions of types.
```

Classes as types

You can also declare a class as the type of a variable.

Let's say you have a class `Person`, with a name:

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

Then you can declare a variable to be of type `Person`:

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

And then, again, you get all the editor support:

Pydantic models

Pydantic is a Python library to perform data validation.

You declare the "shape" of the data as classes with attributes.

And each attribute has a type.

Then you create an instance of that class with some values and it will validate the values, convert them to the appropriate type (if that's the case) and give you an object with all the data.

And you get all the editor support with that resulting object.

An example from the official Pydantic docs:

=== "Python 3.6 and above"

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

=== "Python 3.9 and above"

```
```Python
{!> ../../../docs_src/python_types/tutorial011_py39.py!}
```
```

=== "Python 3.10 and above"

```
```Python
{!> ../../../docs_src/python_types/tutorial011_py310.py!}
```
```

!!! info To learn more about Pydantic, check its docs.

FastAPI is all based on Pydantic.

You will see a lot more of all this in practice in the Tutorial - User Guide{.internal-link target=_blank}.

Type hints in FastAPI

FastAPI takes advantage of these type hints to do several things.

With **FastAPI** you declare parameters with type hints and you get:

- **Editor support**.
- **Type checks**.

...and **FastAPI** uses the same declarations to:

- **Define requirements**: from request path parameters, query parameters, headers, bodies, dependencies, etc.
- **Convert data**: from the request to the required type.
- **Validate data**: coming from each request:
    - Generating **automatic errors** returned to the client when the data is invalid.
- **Document** the API using OpenAPI:
    - which is then used by the automatic interactive documentation user interfaces.

This might all sound abstract. Don't worry. You'll see all this in action in the Tutorial - User Guide{.internal-link target=_blank}.

The important thing is that by using standard Python types, in a single place (instead of adding more classes, decorators, etc), **FastAPI** will do a lot of the work for you.

!!! info If you already went through all the tutorial and came back to see more about types, a good resource is the "cheat sheet" from `mypy` .