FastAPI Learn

Python Types Intro

Python has support for optional "type hints" (also called "type annotations").

These **"type hints"** or annotations 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:

Python 3.8+

```python
def get_full_name(first_name, last_name):
    full_name = first_name.title() + " " + last_name.title()
    return full_name


print(get_full_name("john", "doe"))
```

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.

Python 3.8+

```
def get_full_name(first_name, last_name):
    full_name = first_name.title() + " " + last_name.title()
    return full_name


print(get_full_name("john", "doe"))
```

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...

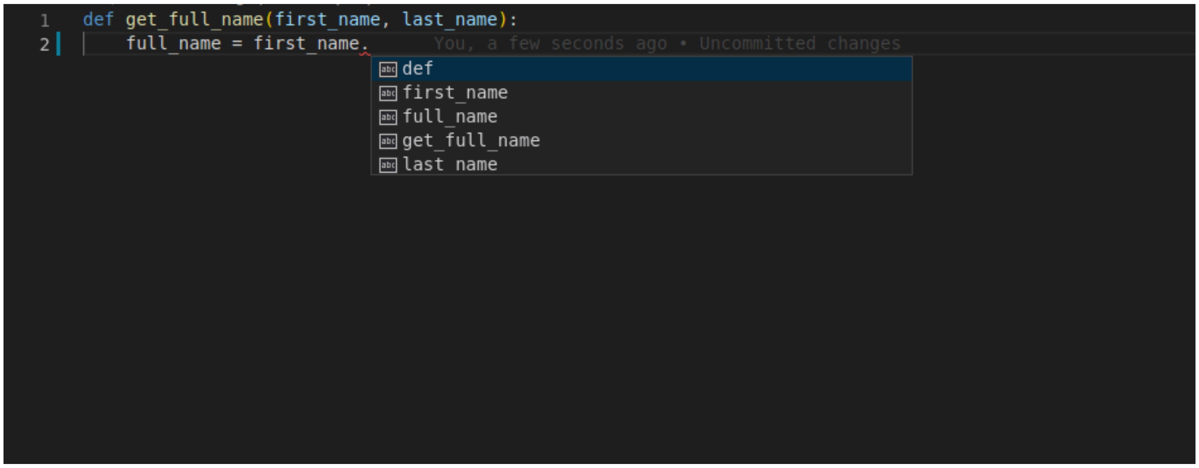
But then you have to call "that method that converts the first letter to upper case".

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:



```
def get_full_name(first_name, last_name):
    full_name = first_name.
```

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":

Python 3.8+

```python
def get_full_name(first_name: str, last_name: str):
    full_name = first_name.title() + " " + last_name.title()
    return full_name


print(get_full_name("john", "doe"))
```

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

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

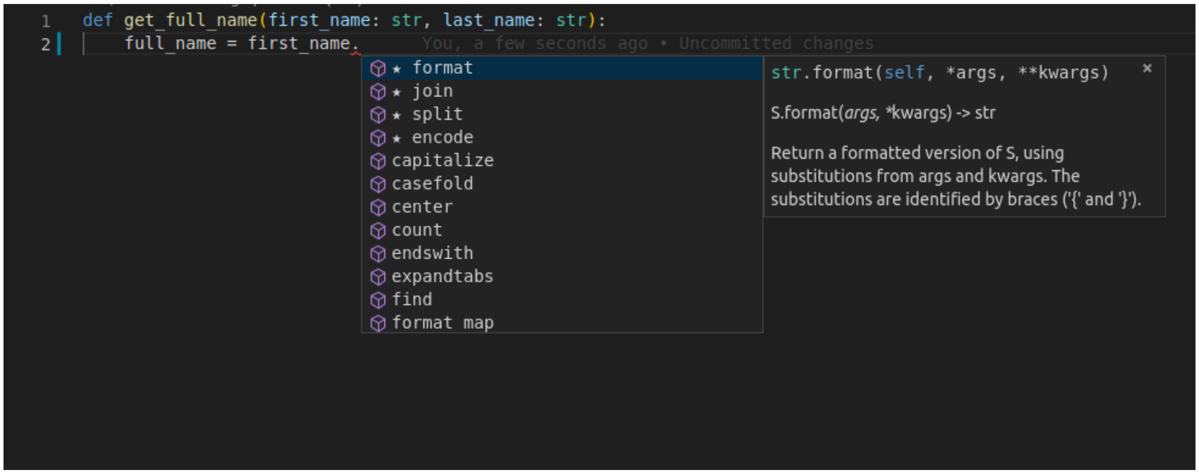
It's a different thing.

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:



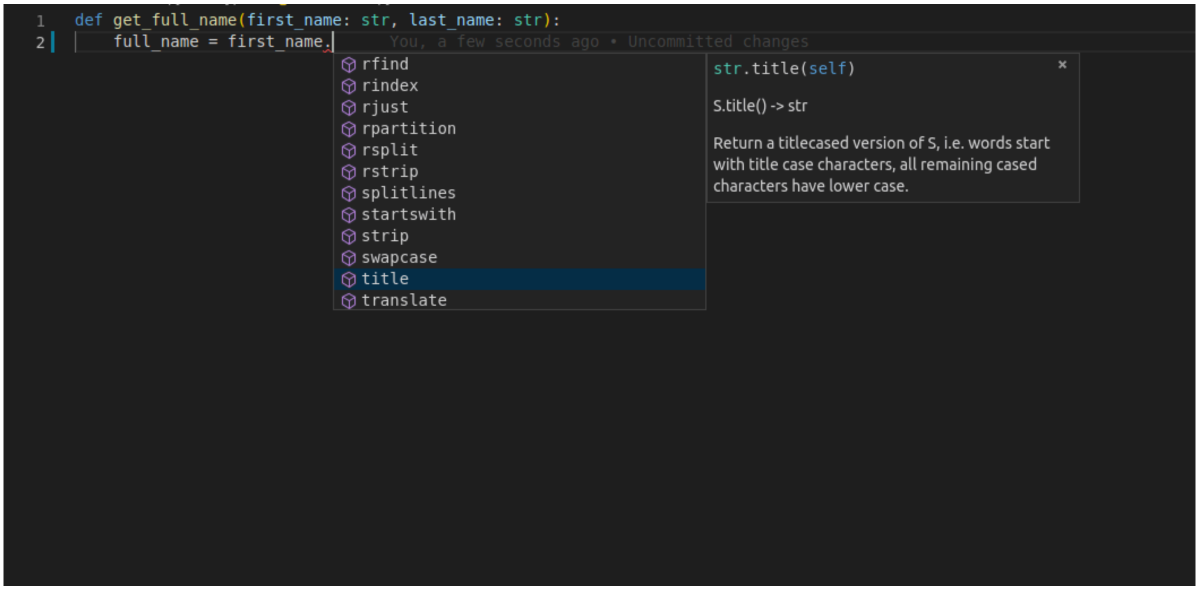
```
def get_full_name(first_name: str, last_name: str):
    full_name = first_name.
```

str.format(self, *args, **kwargs)

S.format(args, *kwargs) -> str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

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



```
def get_full_name(first_name: str, last_name: str):
    full_name = first_name.
```

str.title(self)

S.title() -> str

Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

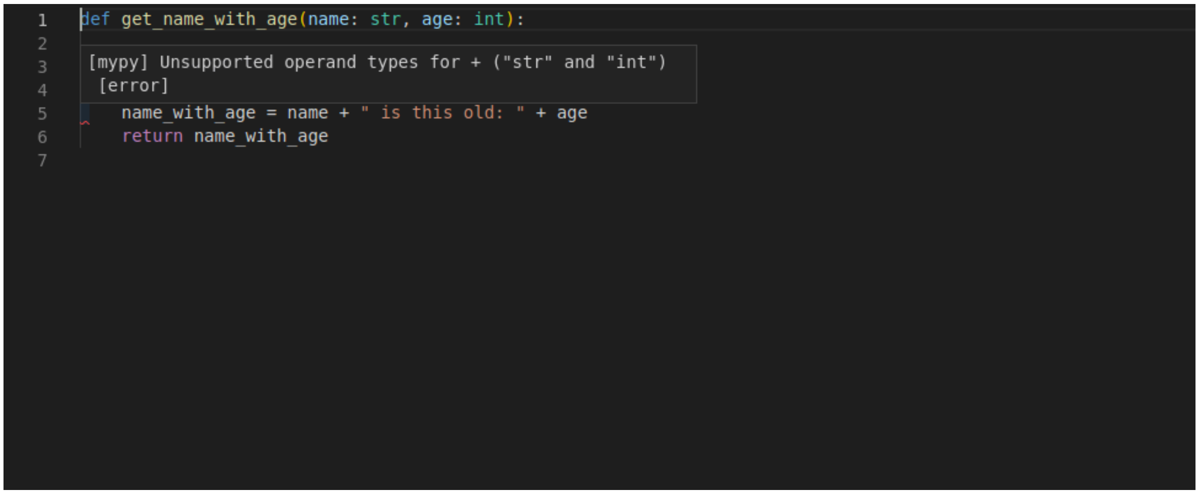
More motivation

Check this function, it already has type hints:

Python 3.8+

```python
def get_name_with_age(name: str, age: int):
    name_with_age = name + " is this old: " + age
    return name_with_age
```

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



```
def get_name_with_age(name: str, age: int):
[mypy] Unsupported operand types for + ("str" and "int")
 [error]
    name_with_age = name + " is this old: " + age
    return name_with_age
```

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

Python 3.8+

```python
def get_name_with_age(name: str, age: int):
    name_with_age = name + " is this old: " + str(age)
    return name_with_age
```

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`

Python 3.8+

```python
def get_items(item_a: str, item_b: int, item_c: float, item_d: bool, item_e: bytes):
    return item_a, item_b, item_c, item_d, item_d, item_e
```

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 choose a more recent version of Python for your project, you will be able to take advantage of that extra simplicity.

In all the docs there are examples compatible with each version of Python (when there's a difference).

For example "**Python 3.6+**" means it's compatible with Python 3.6 or above (including 3.7, 3.8, 3.9, 3.10, etc). And "**Python 3.9+**" means it's compatible with Python 3.9 or above (including 3.10, etc).

If you can use the **latest versions of Python**, use the examples for the latest version, those will have the **best and simplest syntax**, for example, "**Python 3.10+**".

List

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

Python 3.9+

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
def process_items(items: list[str]):
    for item in items:
        print(item)
```

Python 3.8+

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

```python
from typing import List


def process_items(items: List[str]):
    for item in items:
        print(item)
```

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
from typing import List


def process_items(items: List[str]):
    for item in items:
        print(item)
```

Info

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

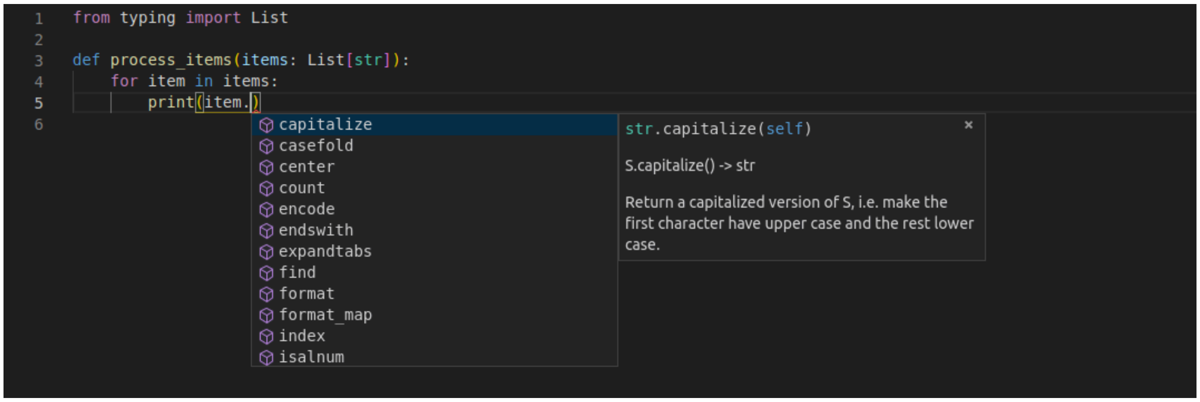
In this case, `str` is the type parameter passed to `List` (or `list` in Python 3.9 and above).

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:



```
from typing import List

def process_items(items: List[str]):
    for item in items:
        print(item.)

```

str.capitalize(self)

S.capitalize() -> str

Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

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 3.9+

```python
def process_items(items_t: tuple[int, int, str], items_s: set[bytes]):
    return items_t, items_s
```

Python 3.8+

```python
from typing import Set, Tuple


def process_items(items_t: Tuple[int, int, str], items_s: Set[bytes]):
    return items_t, items_s
```

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 3.9+

```python
def process_items(prices: dict[str, float]):
    for item_name, item_price in prices.items():
        print(item_name)
        print(item_price)
```

Python 3.8+

```python
from typing import Dict


def process_items(prices: Dict[str, float]):
    for item_name, item_price in prices.items():
        print(item_name)
        print(item_price)
```

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 a **new syntax** where you can put the possible types separated by a vertical bar (`|`).

Python 3.10+

```python
def process_item(item: int | str):
    print(item)
```

Python 3.8+

```python
from typing import Union


def process_item(item: Union[int, str]):
    print(item)
```

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.

```python
from typing import Optional


def say_hi(name: Optional[str] = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        print("Hello World")
```

Using `Optional[str]` instead of just `str` will let the editor help you detect 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 3.10+

```python
def say_hi(name: str | None = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        print("Hello World")
```

Python 3.8+

```python
from typing import Optional


def say_hi(name: Optional[str] = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        print("Hello World")
```

Python 3.8+ alternative

```python
from typing import Union


def say_hi(name: Union[str, None] = None):
    if name is not None:
        print(f"Hey {name}!")
    else:
        print("Hello World")
```

Using `Union` or `Optional`

If you are using a Python version below 3.10, here's a tip from my very **subjective** point of view:

- 🚨 Avoid using `Optional[SomeType]`
- Instead ✨ **use `Union[SomeType, None]`** ✨.

Both are equivalent and underneath they are the same, but I would recommend `Union` instead of `Optional` because the word "**optional**" would seem to imply that the value is optional, and it actually means "it can be `None`", even if it's not optional and is still required.

I think `Union[SomeType, None]` is more explicit about what it means.

It's just about the words and names. But those words can affect how you and your teammates think about the code.

As an example, let's take this function:

Python 3.8+

```python
from typing import Optional


def say_hi(name: Optional[str]):
    print(f"Hey {name}!")
```

🤓 **Other versions and variants**

Python 3.10+

```python
def say_hi(name: str | None):
    print(f"Hey {name}!")
```

The parameter `name` is defined as `Optional[str]`, but it is **not optional**, you cannot call the function without the parameter:

```python
say_hi()  # Oh, no, this throws an error! 😱
```

The `name` parameter is **still required** (not *optional*) because it doesn't have a default value. Still, `name` accepts `None` as the value:

```python
say_hi(name=None)  # This works, None is valid 🎉
```

The good news is, once you are on Python 3.10 you won't have to worry about that, as you will be able to simply use `|` to define unions of types:

Python 3.10+

```python
def say_hi(name: str | None):
    print(f"Hey {name}!")
```

🤓 **Other versions and variants**

Python 3.8+

```python
from typing import Optional


def say_hi(name: Optional[str]):
    print(f"Hey {name}!")
```

And then you won't have to worry about names like `Optional` and `Union`. 😎

Generic types

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

Python 3.10+

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

- `list`
- `tuple`
- `set`
- `dict`

And the same as with Python 3.8, from the `typing` module:

- `Union`
- `Optional` (the same as with Python 3.8)
- ...and others.

In Python 3.10, as an alternative to using the generics `Union` and `Optional`, you can use the vertical bar (`|`) to declare unions of types, that's a lot better and simpler.

Python 3.9+

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

- `list`
- `tuple`
- `set`
- `dict`

And the same as with Python 3.8, from the `typing` module:

- `Union`
- `Optional`
- ...and others.

Python 3.8+

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

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:

Python 3.8+

```python
class Person:
    def __init__(self, name: str):
        self.name = name


def get_person_name(one_person: Person):
    return one_person.name
```

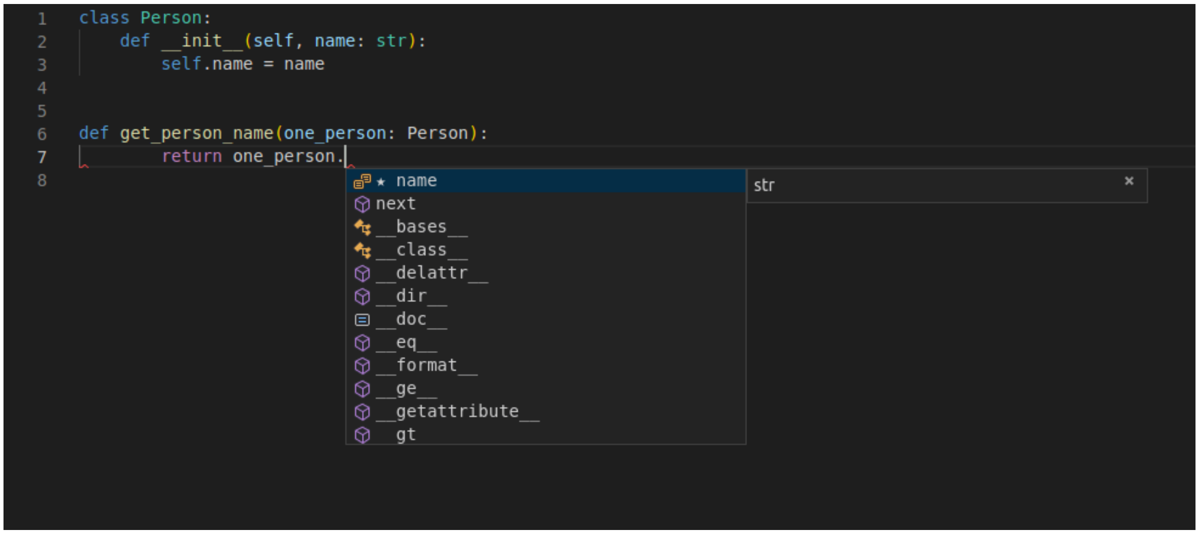
Then you can declare a variable to be of type `Person`:

Python 3.8+

```python
class Person:
    def __init__(self, name: str):
        self.name = name


def get_person_name(one_person: Person):
    return one_person.name
```

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



```
class Person:
    def __init__(self, name: str):
        self.name = name


def get_person_name(one_person: Person):
        return one_person.
```

Notice that this means "`one_person` is an **instance** of the class `Person`".

It doesn't mean "`one_person` is the **class** called `Person`".

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 3.10+

```
from datetime import datetime

from pydantic import BaseModel


class User(BaseModel):
    id: int
    name: str = "John Doe"
    signup_ts: datetime | None = None
    friends: list[int] = []
```

```python
external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22)
friends=[1, 2, 3]
print(user.id)
# > 123
```

Python 3.9+

```python
from datetime import datetime
from typing import Union

from pydantic import BaseModel


class User(BaseModel):
    id: int
    name: str = "John Doe"
    signup_ts: Union[datetime, None] = None
    friends: list[int] = []


external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22)
friends=[1, 2, 3]
print(user.id)
# > 123
```

Python 3.8+

```python
from datetime import datetime
from typing import List, Union

from pydantic import BaseModel


class User(BaseModel):
    id: int
    name: str = "John Doe"
    signup_ts: Union[datetime, None] = None
    friends: List[int] = []


external_data = {
    "id": "123",
    "signup_ts": "2017-06-01 12:22",
    "friends": [1, "2", b"3"],
}
user = User(**external_data)
print(user)
# > User id=123 name='John Doe' signup_ts=datetime.datetime(2017, 6, 1, 12, 22)
friends=[1, 2, 3]
print(user.id)
# > 123
```

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 ↪.

Tip

Pydantic has a special behavior when you use `Optional` or `Union[Something, None]` without a default value, you can read more about it in the Pydantic docs about Required Optional fields [↪].

Type Hints with Metadata Annotations

Python also has a feature that allows putting **additional metadata** in these type hints using `Annotated`.

Python 3.9+

In Python 3.9, `Annotated` is part of the standard library, so you can import it from `typing`.

```
from typing import Annotated


def say_hello(name: Annotated[str, "this is just metadata"]) -> str:
    return f"Hello {name}"
```

Python 3.8+

In versions below Python 3.9, you import `Annotated` from `typing_extensions`.

It will already be installed with **FastAPI**.

```
from typing_extensions import Annotated


def say_hello(name: Annotated[str, "this is just metadata"]) -> str:
    return f"Hello {name}"
```

Python itself doesn't do anything with this `Annotated`. And for editors and other tools, the type is still `str`.

But you can use this space in `Annotated` to provide **FastAPI** with additional metadata about how you want your application to behave.

The important thing to remember is that **the first *type parameter*** you pass to `Annotated` is the **actual type**. The rest, is just metadata for other tools.

For now, you just need to know that `Annotated` exists, and that it's standard Python. 😎

Later you will see how **powerful** it can be.

> **Tip**
>
> The fact that this is **standard Python** means that you will still get the **best possible developer experience** in your editor, with the tools you use to analyze and refactor your code, etc. ✨
>
> And also that your code will be very compatible with many other Python tools and libraries. 🚀

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 ↪.

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` [↪].

Was this page helpful?

☺ ☹