

FastAPI Aprender

Introdução aos tipos Python

O Python possui suporte para "dicas de tipo" ou "type hints" (também chamado de "anotações de tipo" ou "type annotations")

Esses "**type hints**" são uma sintaxe especial que permite declarar o tipo de uma variável.

Ao declarar tipos para suas variáveis, editores e ferramentas podem oferecer um melhor suporte.

Este é apenas um **tutorial rápido / atualização** sobre type hints do Python. Ele cobre apenas o mínimo necessário para usá-los com o **FastAPI**... que é realmente muito pouco.

O **FastAPI** é baseado nesses type hints, eles oferecem muitas vantagens e benefícios.

Mas mesmo que você nunca use o **FastAPI**, você se beneficiaria de aprender um pouco sobre eles.



Nota

Se você é um especialista em Python e já sabe tudo sobre type hints, pule para o próximo capítulo.

Motivação

Vamos começar com um exemplo simples:

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"))
```

A chamada deste programa gera:

John Doe

A função faz o seguinte:

- Pega um `first_name` e `last_name`.
- Converte a primeira letra de cada uma em maiúsculas com `title()`.
- Concatena com um espaço no meio.

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"))
```

Edite-o

É um programa muito simples.

Mas agora imagine que você estava escrevendo do zero.

Em algum momento você teria iniciado a definição da função, já tinha os parâmetros prontos...

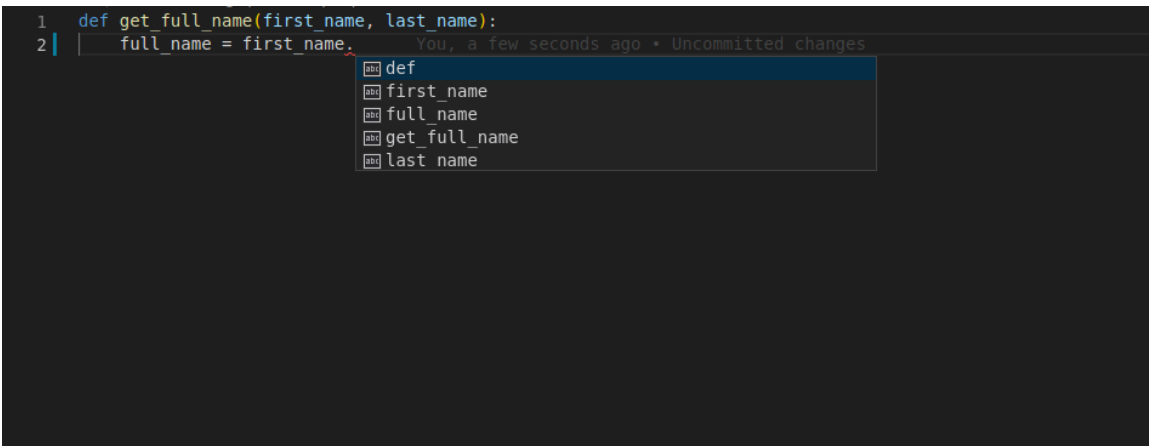
Mas então você deve chamar "esse método que converte a primeira letra em maiúscula".

Era `upper` ? Era `uppercase` ? `first_uppercase` ? `capitalize` ?

Em seguida, tente com o velho amigo do programador, o preenchimento automático do editor.

Você digita o primeiro parâmetro da função, `first_name`, depois um ponto (`.`) e, em seguida, pressiona `Ctrl + Space` para acionar a conclusão.

Mas, infelizmente, você não obtém nada útil:



```
1 def get_full_name(first_name, last_name):  
2 | full_name = first_name.  
   You, a few seconds ago • Uncommitted changes  
   def  
   first_name  
   full_name  
   get_full_name  
   last_name
```

Adicionar tipos

Vamos modificar uma única linha da versão anterior.

Vamos mudar exatamente esse fragmento, os parâmetros da função, de:

```
first_name, last_name
```

para:

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

É isso aí.

Esses são os "type hints":

Python 3.8+

```
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"))
```

Isso não é o mesmo que declarar valores padrão como seria com:

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

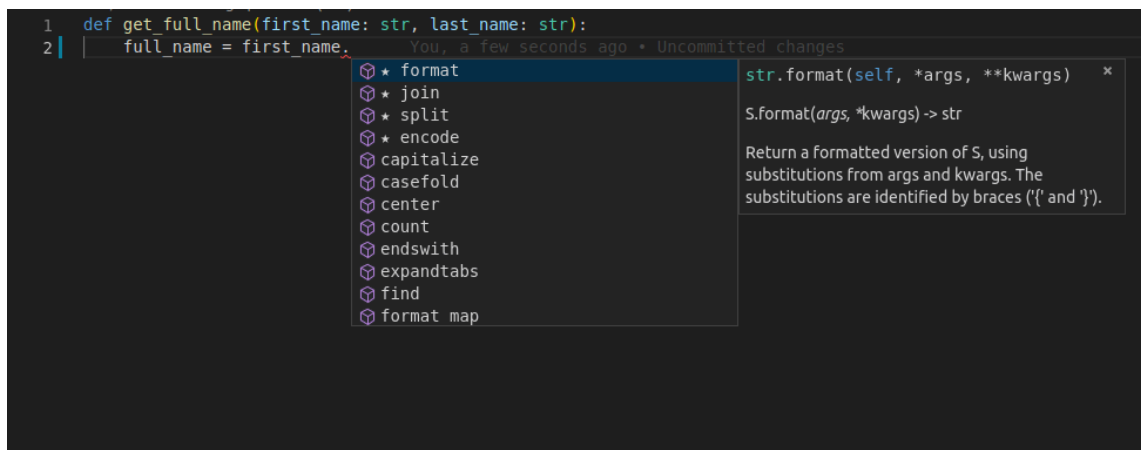
É uma coisa diferente.

Estamos usando dois pontos (:), não é igual a (=).

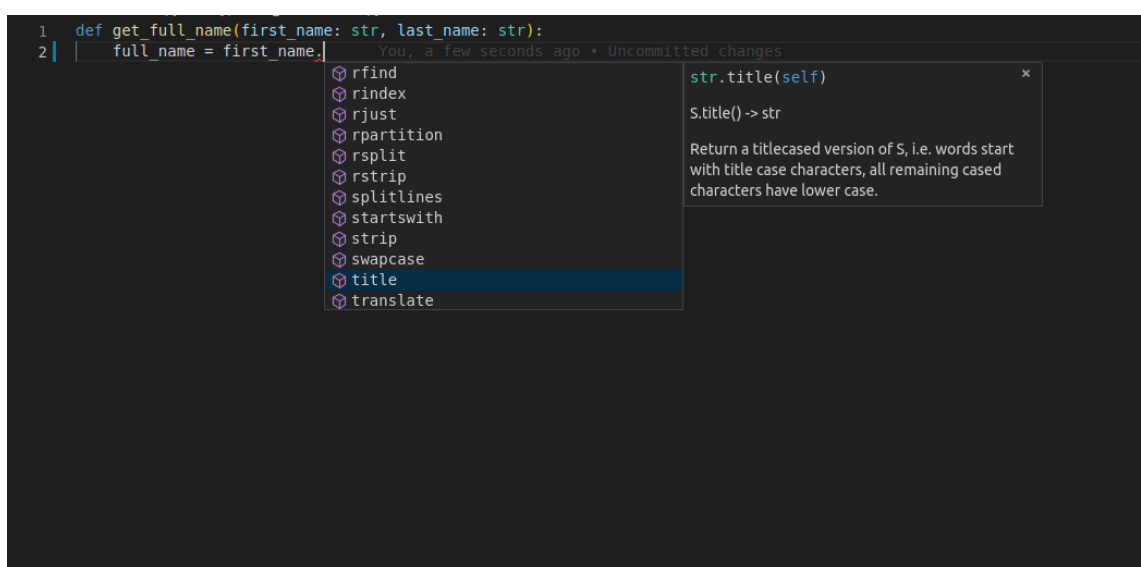
E adicionar type hints normalmente não muda o que acontece do que aconteceria sem eles.

Mas agora, imagine que você está novamente no meio da criação dessa função, mas com type hints.

No mesmo ponto, você tenta acionar o preenchimento automático com o `Ctrl+Space` e vê:



Com isso, você pode rolar, vendo as opções, até encontrar o que "soa familiar":



Mais motivação

Verifique esta função, ela já possui type hints:

Python 3.8+

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

Como o editor conhece os tipos de variáveis, você não obtém apenas o preenchimento automático, mas também as verificações de erro:

```
1 def get_name_with_age(name: str, age: int):
2
3     [mypy] Unsupported operand types for + ("str" and "int")
4     [error]
5     name_with_age = name + " is this old: " + age
6     return name_with_age
7
```

Agora você sabe que precisa corrigí-lo, converta `age` em uma string com `str(age)`:

Python 3.8+

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

Declarando Tipos

Você acabou de ver o local principal para declarar type hints. Como parâmetros de função.

Este também é o principal local em que você os usaria com o **FastAPI**.

Tipos simples

Você pode declarar todos os tipos padrão de Python, não apenas `str`.

Você pode usar, por exemplo:

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

Python 3.8+

```
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
```

Tipos genéricos com parâmetros de tipo

Existem algumas estruturas de dados que podem conter outros valores, como `dict`, `list`, `set` e `tuple`. E os valores internos também podem ter seu próprio tipo.

Estes tipos que possuem tipos internos são chamados de tipos "**genéricos**". E é possível declará-los mesmo com os seus tipos internos.

Para declarar esses tipos e os tipos internos, você pode usar o módulo Python padrão `typing`. Ele existe especificamente para suportar esses type hints.

Versões mais recentes do Python

A sintaxe utilizando `typing` é **compatível** com todas as versões, desde o Python 3.6 até as últimas, incluindo o Python 3.9, 3.10, etc.

Conforme o Python evolui, **novas versões** chegam com suporte melhorado para esses type annotations, e em muitos casos, você não precisará nem importar e utilizar o módulo `typing` para declarar os type annotations.

Se você pode escolher uma versão mais recente do Python para o seu projeto, você poderá aproveitar isso ao seu favor.

Em todos os documentos existem exemplos compatíveis com cada versão do Python (quando existem diferenças).

Por exemplo, "**Python 3.6+**" significa que é compatível com o Python 3.6 ou superior (incluindo o 3.7, 3.8, 3.9, 3.10, etc). E "**Python 3.9+**" significa que é compatível com o Python 3.9 ou mais recente (incluindo o 3.10, etc).

Se você pode utilizar a **versão mais recente do Python**, utilize os exemplos para as últimas versões. Eles terão as **melhores e mais simples sintaxes**, como por exemplo, "**Python 3.10+**".

List

Por exemplo, vamos definir uma variável para ser uma `list` de `str`.

Python 3.9+

Declare uma variável com a mesma sintaxe com dois pontos (`:`)

Como tipo, coloque `list`.

Como a lista é o tipo que contém algum tipo interno, você coloca o tipo dentro de colchetes:

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

Python 3.8+

De `typing`, importe `List` (com o `L` maiúsculo):

```
from typing import List

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

Declare uma variável com a mesma sintaxe com dois pontos (`:`)

Como tipo, coloque o `List` que você importou de `typing`.

Como a lista é o tipo que contém algum tipo interno, você coloca o tipo dentro de colchetes:

```
from typing import List

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

Informação

Estes tipos internos dentro dos colchetes são chamados "parâmetros de tipo" (type parameters).

Neste caso, `str` é o parâmetro de tipo passado para `List` (ou `list` no Python 3.9 ou superior).

Isso significa: "a variável `items` é uma `list`, e cada um dos itens desta lista é uma `str`".

Dica

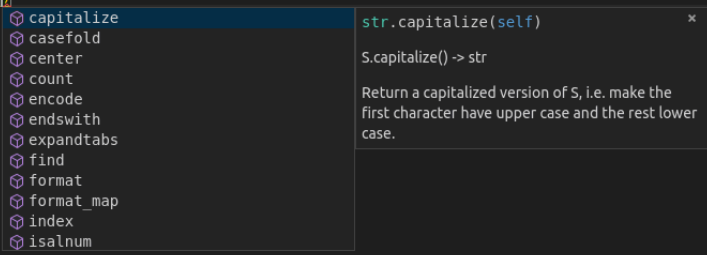
Se você usa o Python 3.9 ou superior, você não precisa importar `List` de `typing`. Você pode utilizar o mesmo tipo `list` no lugar.

Ao fazer isso, seu editor pode fornecer suporte mesmo durante o processamento de itens da lista:

```

1  from typing import List
2
3  def process_items(items: List[str]):
4      for item in items:
5          print(item.capitalize())
6

```



Sem tipos, isso é quase impossível de alcançar.

Observe que a variável `item` é um dos elementos da lista `items`.

E, ainda assim, o editor sabe que é um `str` e fornece suporte para isso.

Tuple e Set

Você faria o mesmo para declarar `tuple` s e `set` s:

Python 3.9+

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

Python 3.8+

```
from typing import Set, Tuple
```

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

Isso significa que:

- A variável `items_t` é uma `tuple` com 3 itens, um `int`, outro `int` e uma `str`.
- A variável `items_s` é um `set`, e cada um de seus itens é do tipo `bytes`.

Dict

Para definir um `dict`, você passa 2 parâmetros de tipo, separados por vírgulas.

O primeiro parâmetro de tipo é para as chaves do `dict`.

O segundo parâmetro de tipo é para os valores do `dict`:

Python 3.9+


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

Python 3.8+

```
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)
```

Isso significa que:

- A variável `prices` é um `dict`:
 - As chaves deste `dict` são do tipo `str` (digamos, o nome de cada item).
 - Os valores deste `dict` são do tipo `float` (digamos, o preço de cada item).

Union

Você pode declarar que uma variável pode ser de qualquer um dentre **diversos tipos**. Por exemplo, um `int` ou um `str`.

No Python 3.6 e superior (incluindo o Python 3.10), você pode utilizar o tipo `Union` de `typing`, e colocar dentro dos colchetes os possíveis tipos aceitáveis.

No Python 3.10 também existe uma **nova sintaxe** onde você pode colocar os possíveis tipos separados por uma barra vertical (`|`).

Python 3.10+

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

Python 3.8+

```
from typing import Union
```

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

Em ambos os casos, isso significa que `item` poderia ser um `int` ou um `str`.

Possivelmente `None`

Você pode declarar que um valor pode ter um tipo, como `str`, mas que ele também pode ser `None`.

No Python 3.6 e superior (incluindo o Python 3.10) você pode declará-lo importando e utilizando `Optional` do módulo `typing`.

```
from typing import Optional

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

O uso de `Optional[str]` em vez de apenas `str` permitirá que o editor o ajude a detectar erros, onde você pode estar assumindo que um valor é sempre um `str`, quando na verdade também pode ser `None`.

`Optional[Something]` é na verdade um atalho para `Union[Something, None]`, eles são equivalentes.

Isso também significa que no Python 3.10, você pode utilizar `Something | None`:

Python 3.10+

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

Python 3.8+

```
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

```
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")
```

Utilizando Union ou Optional

Se você está utilizando uma versão do Python abaixo da 3.10, aqui vai uma dica do meu ponto de vista bem **subjetivo**:

- 🚫 Evite utilizar `Optional[SomeType]`
- No lugar, ✨ **use** `Union[SomeType, None]` ✨.

Ambos são equivalentes, e no final das contas, eles são o mesmo. Mas eu recomendaria o `Union` ao invés de `Optional` porque a palavra **Optional** parece implicar que o valor é opcional, quando na verdade significa "isso pode ser `None`", mesmo que ele não seja opcional e ainda seja obrigatório.

Eu penso que `Union[SomeType, None]` é mais explícito sobre o que ele significa.

Isso é apenas sobre palavras e nomes. Mas estas palavras podem afetar como os seus colegas de trabalho pensam sobre o código.

Por exemplo, vamos pegar esta função:

Python 3.8+

```
from typing import Optional
```

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

🔗 Other versions and variants

Python 3.10+

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

O parâmetro `name` é definido como `Optional[str]`, mas ele **não é opcional**, você não pode chamar a função sem o parâmetro:

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

O parâmetro `name` **ainda é obrigatório** (não *opcional*) porque ele não possui um valor padrão. Mesmo assim, `name` aceita `None` como valor:

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

A boa notícia é, quando você estiver no Python 3.10 você não precisará se preocupar mais com isso, pois você poderá simplesmente utilizar o `|` para definir uniões de tipos:

Python 3.10+

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

🔗 Other versions and variants

Python 3.8+

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

E então você não precisará mais se preocupar com nomes como `Optional` e `Union`. 😎

Tipos genéricos

Esses tipos que usam parâmetros de tipo entre colchetes são chamados **tipos genéricos** ou **genéricos**. Por exemplo:

Python 3.10+

Você pode utilizar os mesmos tipos internos como genéricos (com colchetes e tipos dentro):

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

E o mesmo como no Python 3.8, do módulo `typing`:

- `Union`
- `Optional` (o mesmo que com o 3.8)
- ...entro outros.

No Python 3.10, como uma alternativa para a utilização dos genéricos `Union` e `Optional`, você pode usar a barra vertical (`|`) para declarar uniões de tipos. Isso é muito melhor e mais simples.

Python 3.9+

Você pode utilizar os mesmos tipos internos como genéricos (com colchetes e tipos dentro):

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

E o mesmo como no Python 3.8, do módulo `typing`:

- `Union`
- `Optional`
- ...entro outros.

Python 3.8+

- `List`
- `Tuple`
- `Set`
- `Dict`
- `Union`
- `Optional`
- ...entro outros.

Classes como tipos

Você também pode declarar uma classe como o tipo de uma variável.

Digamos que você tenha uma classe `Person`, com um nome:

Python 3.8+

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

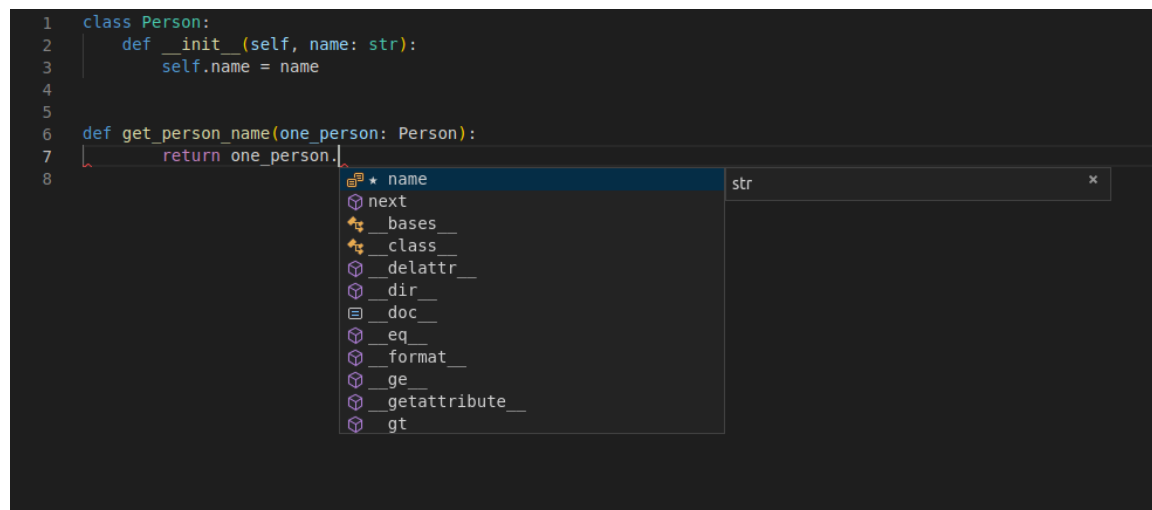
```
def get_person_name(one_person: Person):  
    return one_person.name
```

Então você pode declarar que uma variável é do tipo `Person`:

Python 3.8+

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

E então, novamente, você recebe todo o suporte do editor:



Perceba que isso significa que "`one_person`" é uma **instância** da classe `Person`.

Isso não significa que "`one_person`" é a **classe** chamada `Person`.

Modelos Pydantic

O **Pydantic** ([↗](#)) é uma biblioteca Python para executar a validação de dados.

Você declara a "forma" dos dados como classes com atributos.

E cada atributo tem um tipo.

Em seguida, você cria uma instância dessa classe com alguns valores e ela os validará, os converterá para o tipo apropriado (se for esse o caso) e fornecerá um objeto com todos os dados.

E você recebe todo o suporte do editor com esse objeto resultante.

Retirado dos documentos oficiais dos Pydantic:

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] = []


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+

```
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+

```
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
```

**Informação**

Para saber mais sobre o [Pydantic](#), verifique a sua documentação [↔].

O **FastAPI** é todo baseado em Pydantic.

Você verá muito mais disso na prática no [Tutorial - Guia do usuário](#) [↔].

**Dica**

O Pydantic tem um comportamento especial quando você usa `Optional` ou `Union[Something, None]` sem um valor padrão. Você pode ler mais sobre isso na documentação do Pydantic sobre [campos Opcionais Obrigatórios](#) [↔].

Type Hints com Metadados de Anotações

O Python possui uma funcionalidade que nos permite incluir **metadados adicionais** nos type hints utilizando `Annotated`.

Python 3.9+

No Python 3.9, `Annotated` é parte da biblioteca padrão, então você pode importá-lo de `typing`.

```
from typing import Annotated

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

Python 3.8+

Em versões abaixo do Python 3.9, você importa `Annotated` de `typing_extensions`.

Ele já estará instalado com o **FastAPI**.

```
from typing_extensions import Annotated

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

O Python em si não faz nada com este `Annotated`. E para editores e outras ferramentas, o tipo ainda é `str`.

Mas você pode utilizar este espaço dentro do `Annotated` para fornecer ao **FastAPI** metadata adicional sobre como você deseja que a sua aplicação se comporte.

O importante aqui de se lembrar é que o **primeiro type parameter** que você informar ao `Annotated` é o **tipo de fato**. O resto é apenas metadado para outras ferramentas.

Por hora, você precisa apenas saber que o `Annotated` existe, e que ele é Python padrão. 😎

Mais tarde você verá o quão **poderoso** ele pode ser.



Dica

O fato de que isso é **Python padrão** significa que você ainda obtém a **melhor experiência de desenvolvedor possível** no seu editor, com as ferramentas que você utiliza para analisar e refatorar o seu código, etc. ✨

E também que o seu código será muito compatível com diversas outras ferramentas e bibliotecas Python. 🚀

Type hints no FastAPI

O **FastAPI** aproveita esses type hints para fazer várias coisas.

Com o **FastAPI**, você declara parâmetros com type hints e obtém:

- **Suporte ao editor.**
- **Verificações de tipo.**

... e o **FastAPI** usa as mesmas declarações para:

- **Definir requisitos:** dos parâmetros de rota, parâmetros da consulta, cabeçalhos, corpos, dependências, etc.
- **Converter dados:** da solicitação para o tipo necessário.
- **Validar dados:** provenientes de cada solicitação:
 - Gerando **erros automáticos** retornados ao cliente quando os dados são inválidos.
- **Documentar** a API usando OpenAPI:
 - que é usado pelas interfaces de usuário da documentação interativa automática.

Tudo isso pode parecer abstrato. Não se preocupe. Você verá tudo isso em ação no [Tutorial - Guia do usuário ↗](#).

O importante é que, usando tipos padrão de Python, em um único local (em vez de adicionar mais classes, decoradores, etc.), o **FastAPI** fará muito trabalho para você.

Informação

Se você já passou por todo o tutorial e voltou para ver mais sobre os tipos, um bom recurso é a "cheat sheet" do `mypy` [\[↗\]](#).