Python 类型提示简介

Python 3.6+ 版本加入了对"类型提示"的支持。

这些**"类型提示"**是一种新的语法（在 Python 3.6 版本加入）用来声明一个变量的类型。

通过声明变量的类型，编辑器和一些工具能给你提供更好的支持。

这只是一个关于 Python 类型提示的**快速入门 / 复习**。它仅涵盖与 **FastAPI** 一起使用所需的最少部分...实际上只有很少一点。

整个 **FastAPI** 都基于这些类型提示构建，它们带来了许多优点和好处。

但即使你不会用到 **FastAPI**，了解一下类型提示也会让你从中受益。

!!! note 如果你已经精通 Python，并且了解关于类型提示的一切知识，直接跳到下一章节吧。

动机

让我们从一个简单的例子开始：

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

运行这段程序将输出：

```
John Doe
```

这个函数做了下面这些事情：

- 接收 `first_name` 和 `last_name` 参数。
- 通过 `title()` 将每个参数的第一个字母转换为大写形式。
- 中间用一个空格来拼接它们。

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

修改示例

这是一个非常简单的程序。

现在假设你将从头开始编写这段程序。

在某一时刻，你开始定义函数，并且准备好了参数...。

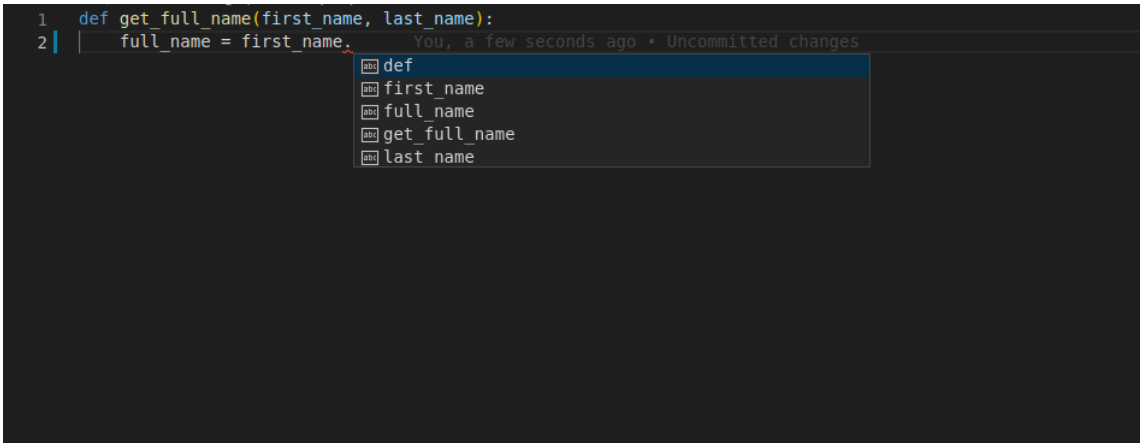
现在你需要调用一个"将第一个字母转换为大写形式的方法"。

等等，那个方法是什么来着？ `upper`？还是 `uppercase`？ `first_uppercase`？ `capitalize`？

然后你尝试向程序员老手的朋友——编辑器自动补全寻求帮助。

输入函数的第一个参数 `first_name`，输入点号（ `.` ）然后敲下 `Ctrl+Space` 来触发代码补全。

但遗憾的是并没有起什么作用：



添加类型

让我们来修改上面例子的一行代码。

我们将把下面这段代码中的函数参数从：

```
    first_name, last_name
```

改成：

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

就是这样。

这些就是"类型提示"：

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

这和声明默认值是不同的，例如：

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

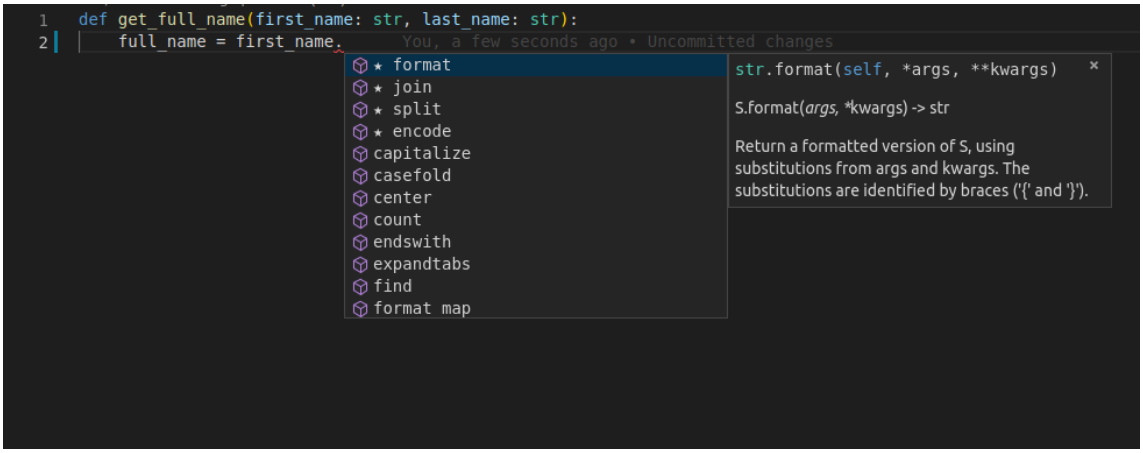
这两者不一样。

我们用的是冒号（`:`），不是等号（`=`）。

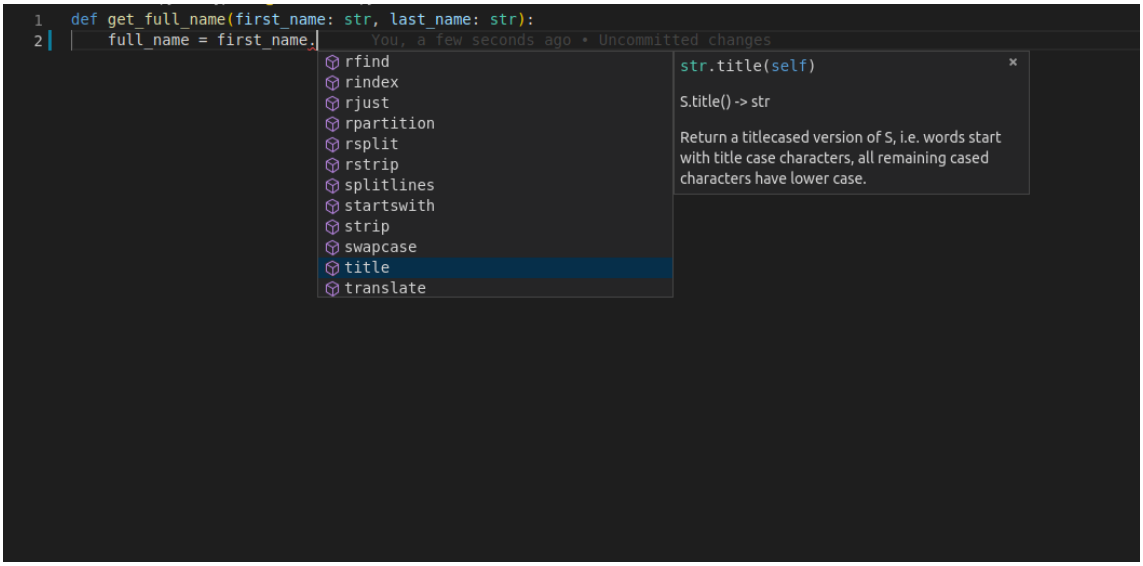
而且添加类型提示一般不会改变原来的运行结果。

现在假设我们又一次正在创建这个函数，这次添加了类型提示。

在同样的地方，通过 `Ctrl+Space` 触发自动补全，你会发现：



这样，你可以滚动查看选项，直到你找到看起来眼熟的那个：

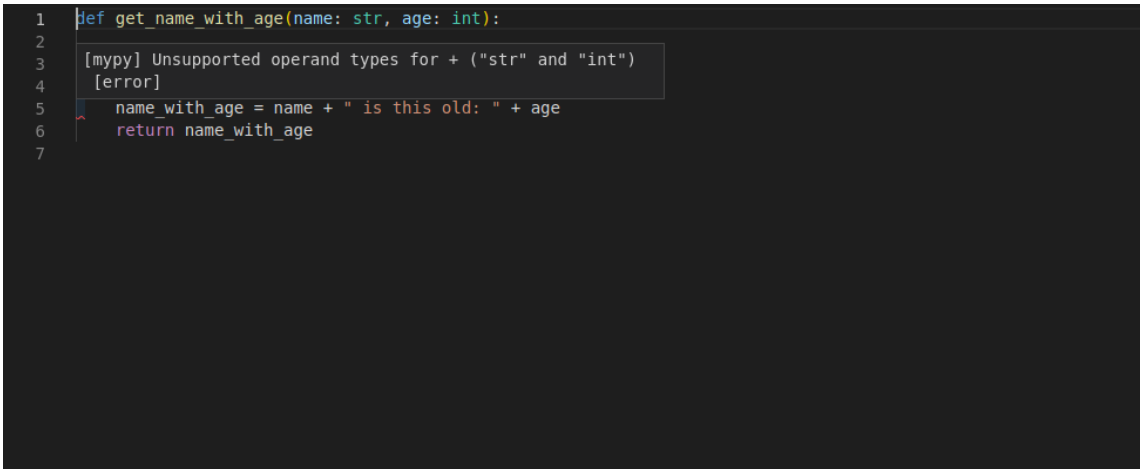


更多动机

下面是一个已经有类型提示的函数：

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

因为编辑器已经知道了这些变量的类型，所以不仅能对代码进行补全，还能检查其中的错误：



现在你知道了必须先修复这个问题，通过 `str(age)` 把 `age` 转换成字符串：

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

声明类型

你刚刚看到的就是声明类型提示的主要场景。用于函数的参数。

这也是你将在 **FastAPI** 中使用它们的主要场景。

简单类型

不只是 `str`，你能够声明所有的标准 Python 类型。

比如以下类型：

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

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

嵌套类型

有些容器数据结构可以包含其他的值，比如 `dict`、`list`、`set` 和 `tuple`。它们内部的值也会拥有自己的类型。

你可以使用 Python 的 `typing` 标准库来声明这些类型以及子类型。

它专门用来支持这些类型提示。

列表

例如，让我们来定义一个由 `str` 组成的 `list` 变量。

从 `typing` 模块导入 `List`（注意是大写的 `L`）：

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

同样以冒号（`:`）来声明这个变量。

输入 `List` 作为类型。

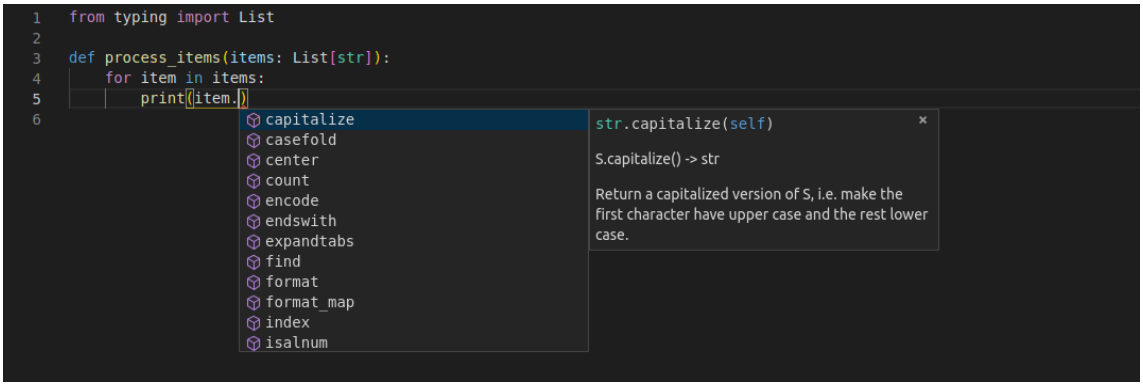
由于列表是带有"子类型"的类型，所以我们把子类型放在方括号中：

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

这表示："变量 `items` 是一个 `list`，并且这个列表里的每一个元素都是 `str`"。

这样，即使在处理列表中的元素时，你的编辑器也可以提供支持。

没有类型，几乎是不可能实现下面这样：



注意，变量 `item` 是列表 `items` 中的元素之一。

而且，编辑器仍然知道它是一个 `str`，并为此提供了支持。

元组和集合

声明 `tuple` 和 `set` 的方法也是一样的：

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

这表示：

- 变量 `items_t` 是一个 `tuple`，其中的每个元素都是 `int` 类型。
- 变量 `items_s` 是一个 `set`，其中的每个元素都是 `bytes` 类型。

字典

定义 `dict` 时，需要传入两个子类型，用逗号进行分隔。

第一个子类型声明 `dict` 的所有键。

第二个子类型声明 `dict` 的所有值：

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

这表示：

- 变量 `prices` 是一个 `dict`：
  - 这个 `dict` 的所有键为 `str` 类型（可以看作是字典内每个元素的名称）。
  - 这个 `dict` 的所有值为 `float` 类型（可以看作是字典内每个元素的价格）。

类作为类型

你也可以将类声明为变量的类型。

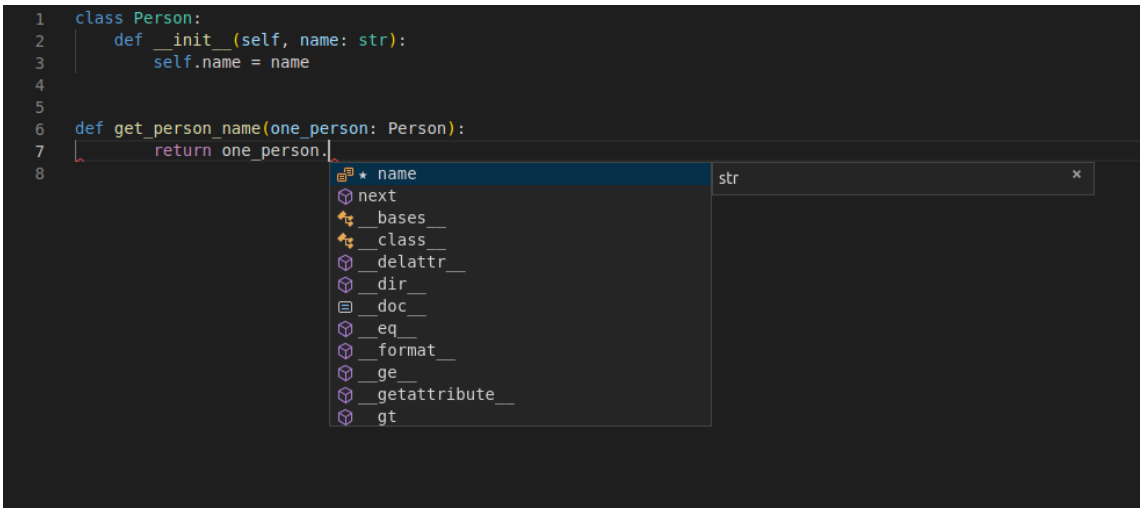
假设你有一个名为 `Person` 的类，拥有 name 属性：

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

接下来，你可以将一个变量声明为 `Person` 类型：

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

然后，你将再次获得所有的编辑器支持：



Pydantic 模型

Pydantic 是一个用来用来执行数据校验的 Python 库。

你可以将数据的"结构"声明为具有属性的类。

每个属性都拥有类型。

接着你用一些值来创建这个类的实例，这些值会被校验，并被转换为适当的类型（在需要的情况下），返回一个包含所有数据的对象。

然后，你将获得这个对象的所有编辑器支持。

下面的例子来自 Pydantic 官方文档：

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

!!! info 想进一步了解 Pydantic，请阅读其文档.

整个 **FastAPI** 建立在 Pydantic 的基础之上。

实际上你将在 教程 - 用户指南{.internal-link target=_blank} 看到很多这种情况。

FastAPI 中的类型提示

FastAPI 利用这些类型提示来做下面几件事。

使用 **FastAPI** 时用类型提示声明参数可以获得：

- **编辑器支持。**
- **类型检查。**

...并且 **FastAPI** 还会用这些类型声明来：

- **定义参数要求**：声明对请求路径参数、查询参数、请求头、请求体、依赖等的要求。
- **转换数据**：将来自请求的数据转换为需要的类型。
- **校验数据**： 对于每一个请求：
    - 当数据校验失败时自动生成**错误信息**返回给客户端。
- 使用 OpenAPI **记录** API：
    - 然后用于自动生成交互式文档的用户界面。

听上去有点抽象。不过不用担心。你将在 教程 - 用户指南{.internal-link target=_blank} 中看到所有的实战。

最重要的是，通过使用标准的 Python 类型，只需要在一个地方声明（而不是添加更多的类、装饰器等），**FastAPI** 会为你完成很多的工作。

!!! info 如果你已经阅读了所有教程，回过头来想了解有关类型的更多信息，来自 `mypy` 的"速查表"是不错的资源。