



Understanding Python's Mapping Type

(2025-05-08 11:32:01)

User

python Mapping类型

Assistant

Searched codebase "python Mapping类型" • **16** results

File	Lines
ch3-3-2.py	L1-38
ch3-3-2.py	L38-70
pdf_dict/optional.md	L63-67
pdf_dict/optional.md	L1-63
test.py	L1-8
ch3.2.py	L16-35
ch3.2.py	L1-16
test.py	L8-11
ch2.py	L1-38
ch3-3.py	L1-12
ch3-3-2.py	L70-88
ch2.py	L38-51

File	Lines
ch3-3-2.py	L88-92
ch2.ipynb	L1-1
.cursorindexingignore	L1-4
.gitignore	L1-3

在Python中，`Mapping` 是 `typing` 模块中的一个抽象基类，用于表示类似字典的映射类型。它是所有映射类型（如字典）的基类。

`Mapping` 类型的主要特点：

1. 表示键值对集合的抽象接口
2. 是一个只读的映射类型，不包含修改操作（如果需要可修改的映射类型，应使用 `MutableMapping`）
3. 通常用于函数参数类型提示，表示函数接受任何类似字典的对象

在代码示例中我们可以看到 `Mapping` 的使用：

```
from typing import Mapping, Any

@property
def _identifying_params(self) -> Mapping[str, Any]:
    return {
        "model_name": self.model_name,
        "model_folder_path": self.model_folder_path,
        **self._get_model_default_parameters
    }
```

`Mapping` 的优势和使用场景：

1. 相比直接使用 `dict` 类型提示，`Mapping` 更加抽象和灵活
2. 可以接受任何符合映射接口的类型，不仅限于Python的内置 `dict`
3. 当函数只需要读取映射内容而不修改时，使用 `Mapping` 比 `dict` 更准确

在使用时，`Mapping` 通常需要两个类型参数：键的类型和值的类型，例如：

- `Mapping[str, int]` - 键为字符串，值为整数的映射
- `Mapping[str, Any]` - 键为字符串，值可以是任何类型的映射

如果需要可修改的映射类型（可以添加、删除或修改键值对），应该使用 `MutableMapping`。

User

可以接受任何符合映射接口的类型，不仅限于Python的内置`dict`，请详细解释一下

Assistant

在Python中，`Mapping` 类型可以接受任何符合映射接口的类型，这是因为Python遵循"鸭子类型"（duck typing）原则，只要某个对象表现得像映射，实现了映射所需的接口方法，就可以被视为映射类型。这不仅限于内置的 `dict` 类型，还包括许多其他类型。

符合映射接口的类型包括：

1. 内置的字典类型：`dict`
2. `collections`模块中的映射类型：
 - `collections.defaultdict` - 当键不存在时提供默认值的字典
 - `collections.OrderedDict` - 保持插入顺序的字典
 - `collections.Counter` - 计数器，用于统计可哈希对象
 - `collections.ChainMap` - 将多个字典链接在一起
3. 第三方库中的映射类型：
 - 许多数据库接口提供的类似字典的结果对象
 - `pandas.DataFrame` 的某些访问方法
4. 自定义类：任何实现了以下关键方法的类：
 - `__getitem__` - 通过键获取值
 - `__iter__` - 允许迭代所有键
 - `__len__` - 返回项目数量

- `__contains__` - 检查键是否存在

示例自定义映射类：

```
class MyMapping:
    def __init__(self, data=None):
        self._data = data or {}

    def __getitem__(self, key):
        return self._data[key]

    def __iter__(self):
        return iter(self._data)

    def __len__(self):
        return len(self._data)

    def __contains__(self, key):
        return key in self._data

    def keys(self):
        return self._data.keys()
```

使用 `Mapping` 类型提示的优势是 ‘它更准确地表达了函数的意图——函数只需要一个能够通过键查找值的对象’ 而不一定要求是特定的 `dict` 类型。这使得代码更加灵活 ‘能够接受各种兼容的数据结构’。

User

当函数只需要读取映射内容而不修改时 ‘使用 `Mapping` 比 `dict` 更准确’ 请详细解释一下

Assistant

使用 `Mapping` 类型而非 `dict` 在函数只需读取映射内容时更准确，主要有以下几个原因：

1. 接口契约更精确

`Mapping` 是一个只读接口，明确表明函数只会读取数据而不会修改它。相比之下，`dict` 类型暗示该对象可能会被修改（如添加、删除或更新键值对）。

2. 遵循最小权限原则

在函数参数类型中使用 `Mapping` 遵循了软件设计中的最小权限原则 - 函数应该只获取完成其工作所需的最小权限。如果函数只需要读取映射而不需要修改它，使用 `Mapping` 更准确地表达了这一意图。

3. 提高代码灵活性

使用 `Mapping` 允许函数接受任何提供映射接口的对象，而不仅限于 `dict`。这使得函数更加灵活，可以处理各种实现了映射接口的数据结构。

4. 更好的接口设计

通过使用 `Mapping`，你传达了函数对参数的真实需求 - 它只需要一个可以通过键查找值的接口，无需其他功能。

举例说明：

```
from typing import Mapping, Dict

# 使用Mapping表示只读意图
def print_user_info(user_data: Mapping[str, str]) -> None:
    """打印用户信息（只读取，不修改）"""
    print(f"姓名: {user_data.get('name', '未知')}")
    print(f"年龄: {user_data.get('age', '未知')}")

# 使用Dict可能暗示函数会修改数据
def update_user_info(user_data: Dict[str, str], new_age: str) -> Dict[str, str]:
    """更新并返回用户信息（读取并修改）"""
    user_data['age'] = new_age
    return user_data
```

这种类型提示不仅使代码自文档化，还能帮助静态类型检查工具（如 `mypy`）识别潜在的错误，如在只读函数中尝试修改映射的操作。

使用Mapping允许函数接受任何提供映射接口的对象，而不仅限于dict。这使得函数更加灵活，可以处理各种实现了映射接口的数据结构。举例详细解释下

Assistant

使用 Mapping 的灵活性示例

通过使用 Mapping 类型，函数可以接受各种符合映射接口的对象，下面通过实际例子详细说明：

示例函数

```
from typing import Mapping, Any

def summarize_data(data: Mapping[str, Any]) -> str:
    """汇总映射中的数据"""
    items = len(data)
    keys = ", ".join(list(data.keys())[:3]) + ("..." if items > 3 else "")
    return f"包含{items}项数据，键包括: {keys}"
```

可接受的各种映射类型

1. 标准字典

```
result = summarize_data({"name": "张三", "age": 30, "city": "上海"})
# 输出: "包含3项数据，键包括: name, age, city"
```

2. 有序字典

```
from collections import OrderedDict

user_info = OrderedDict([
    ("name", "李四"),
    ("email", "lisi@example.com"),
    ("phone", "12345678901"),
    ("address", "北京市")
])

result = summarize_data(user_info)
# 输出: "包含4项数据, 键包括: name, email, phone..."
```

3. 默认字典

```
from collections import defaultdict

access_counts = defaultdict(int)
access_counts["home_page"] = 1503
access_counts["about_page"] = 722
access_counts["contact_page"] = 492

result = summarize_data(access_counts)
# 输出: "包含3项数据, 键包括: home_page, about_page, contact_page"
```

4. 自定义映射类

```
class ConfigMap:
    def __init__(self, config_file):
        self._data = self._load_config(config_file)

    def _load_config(self, file):
        # 假设从文件加载配置
        return {"db_host": "localhost", "db_port": 5432, "timeout": 30}

    def __getitem__(self, key):
        return self._data[key]

    def __iter__(self):
        return iter(self._data)

    def __len__(self):
        return len(self._data)

    def keys(self):
        return self._data.keys()

# 创建自定义映射对象
config = ConfigMap("config.ini")
result = summarize_data(config)
# 输出: "包含3项数据, 键包括: db_host, db_port, timeout"
```

5. 数据库结果对象

```
# 使用SQLAlchemy的Row对象 (类似字典的接口)
from sqlalchemy import create_engine, text

engine = create_engine("sqlite:///memory:")
with engine.connect() as conn:
    result = conn.execute(text("SELECT 'value' as key"))
    row = result.mappings().first() # 返回类似字典的对象
    summary = summarize_data(row)
# 可以处理数据库查询结果
```


6. 环境变量包装器

```
import os

class EnvVarMap:
    def __getitem__(self, key):
        return os.environ[key]

    def __iter__(self):
        return iter(os.environ)

    def __len__(self):
        return len(os.environ)

    def keys(self):
        return os.environ.keys()

# 环境变量的映射视图
env_vars = EnvVarMap()
result = summarize_data(env_vars)
# 输出环境变量的摘要信息
```

这种灵活性使函数可以处理各种来源和实现的数据结构，而无需修改函数本身。这遵循了“面向接口编程”而非“面向实现编程”的原则，提高了代码的可重用性和可维护性。
