Python: usare i decorator per i getter e setter delle classi

In questo articolo parleremo dei decorator usati per specificare getter e setter nelle classi Python.

Immaginiamo di dover implementare una classe che utilizzi pymongo per interfacciarsi con un database MongoDB.

La prima scelta che dobbiamo affrontare a livello di design è relativa alla memorizzazione dell'URL di connessione al database e l'istanza del client che utilizza tale URL. Una semplice soluzione è quella di usare un attributo di classe e un attributo pubblico nel seguente modo:

```
import pymongo
from pymongo import MongoClient

class Database():
    connection_str = 'mongodb://localhost:27017/'

    def __init__(self):
        self.client =
MongoClient(self.connection_str)
```

Ha senso al livello di design che il client sia pubblico in modo da renderlo disponibile in altre parti del nostro code base.

A questo punto dobbiamo impostare i riferimenti al database e alla collezione di documenti selezionati. In questo caso scegliamo di mantenere questi attributi privati.

```
class Database():
    connection_str = 'mongodb://localhost:27017/'

    def __init__(self):
        self.client =
MongoClient(self.connection_str)
        self._collection = None
        self._database = None
```

Chi proviene da un linguaggio in cui il paradigma OO segue l'insegnamento di Java, potrebbe essere tentato di aggiungere i seguenti metodi per implementare i getter e setter.

```
class Database():
    connection_str = 'mongodb://localhost:27017/'

    def __init__(self):
        self.client =
MongoClient(self.connection_str)
        self._collection = None
        self._database = None

    def get_database(self):
        return self._database

    def set_database(self, name):
        self._database = self.client[name]

    def get_collection(self):
        return self._collection

    def set_collection(self, name):
```

```
        self._collection = self._database[name]
```

Questo codice funziona correttamente, ma non appena proviamo ad usarlo, ci accorgiamo subito che il suo design non è usabile e meno che mai "pythonico":

```
db = Database()
db.set_database('test')
db.set_collection('data')
```

Possiamo ottenere esattamente lo stesso risultato usando invece il decorator `@property` per creare dei getter che ci permettono di accedere al membro della classe senza dover invocare un metodo direttamente.

```
class Database():
    connection_str = 'mongodb://localhost:27017/'

    def __init__(self):
        self.client =
MongoClient(self.connection_str)
        self._collection = None
        self._database = None

    @property
    def database(self):
        return self._database

    @property
    def collection(self):
        return self._collection
```

Ora possiamo accedere ai due attributi dall'esterno della classe semplicemente con `db.database` e `db.collection`. Per creare invece dei setter, usiamo il nome dei metodi appena definiti come altrettanti decorator specificando l'attributo `setter`, in questo modo:

```
class Database():
    connection_str = 'mongodb://localhost:27017/'

    def __init__(self):
        self.client =
MongoClient(self.connection_str)
        self._collection = None
        self._database = None

    @property
    def database(self):
        return self._database

    @database.setter
    def database(self, name):
        self._database = self.client[name]

    @property
    def collection(self):
        return self._collection

    @collection.setter
    def collection(self, name):
        self._collection = self._database[name]
```

Ora il codice visto in precedenza che utilizzava l'invocazione diretta dei metodi può essere riscritto come segue:

```
db = Database()
db.database = 'test'
db.collection = 'data'
```

In conclusione, Python offre una modalità diversa per impostare i getter e setter di una classe che differisce dai tradizionali linguaggi OO.