Python: how to create getters and setters for classes with decorators

In this article we will talk about the decorators used to specify getters and setters in Python classes.

Let's imagine we need to implement a class that uses pymongo to work with a MongoDB database.

The first design choice we face is how to store the database connection URL and the client instance using that URL. A simple solution is to use a class attribute and a public attribute like this:

```
import pymongo
from pymongo import MongoClient

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

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

It makes sense that the client instance is kept public in order to make it available elsewhere in our code base.

At this point we have to set the references to the database and to the collection of documents. In this case we choose to keep these attributes private.

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

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

A developer who comes from a language where the OO paradigm follows the traditional Java design might be tempted to add the following methods to implement getters and setters.

```
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]
```

This code works fine, but as soon as we try to use it, we immediately realize that its design is not usable and is not "pythonic":

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

We can achieve exactly the same result by using the `@property` decorator instead to create getters that allow us to access the class member without having to invoke a method directly.

```
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
```

Now we can access the two attributes from outside the class simply with `db.database` and `db.collection`. To create setters instead, we use the name of the methods just defined as decorators by specifying the `setter` attribute, like this:

```
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]
```

Now the code seen above that used direct method invocation can be rewritten as follows:

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

In short, Python offers a different way to set a class's getters and setters that differs from traditional OO languages.