


Open in app ↗

Sign up

Sign in

Medium

 Search

Writing descriptors in Python 3.6+

Daw-Ran Liou · [Follow](#)

5 min read · Sep 24, 2017



Listen



Share



American football players “blocking” the kick, not “intercepting.”

This article is also cross-shared to [Dev.to](#) and [my blog](#).

Special thanks to [Luciano Ramalho](#). I learned most of the knowledge about descriptors from his workshop in PyBay 2017

Have you seen this code or maybe have written code like this?

```
from sqlalchemy import Column, Integer, String
class User(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String)
```

This code snippet partially comes from the tutorial of a popular ORM package called [SQLAlchemy](#). If you ever wonder why the attributes `id` and `name` aren't passed into the `__init__` method and bind to the instance like [regular class does](#), this post is for you.

This post starts with explaining descriptors, why to use them, how to write them in previous Python versions (≤ 3.5), and finally writing them in Python 3.6 with the new feature described in [PEP 487 — Simpler customisation of class creation](#)

If you are in a hurry or you just want to know what's new, scroll all the way down to the bottom of this article. You'll find the whole code.

What are descriptors

A great definition of descriptor is explained by Raymond Hettinger in [Descriptor HowTo Guide](#):

In general, a descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

There are three ways to access an attribute. Let's say we have the `a` attribute on the object `obj` :

1. To lookup its value, `some_variable = obj.a`,
2. To change its value, `obj.a = 'new value'`, or
3. To delete it, `del obj.a`

Python is dynamic and flexible to allow users intercept the above expression/statement and bind behaviors to them.

Why you want to use descriptors

Let's see an example:

```
class Order:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def total(self):
        return self.price * self.quantity

apple_order = Order('apple', 1, 10)
apple_order.total()
# 10
```

Despite the lack of proper documentation, there is a bug:

```
apple_order.quantity = -10
apple_order.total
# -10, too good of a deal!
```

Instead of using getter and setter methods and break the APIs, let's use property to enforce `quantity` be positive:

```

class Order:
    def __init__(self, name, price, quantity):
        self._name = name
        self.price = price
        self._quantity = quantity  # (1)

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        if value < 0:
            raise ValueError('Cannot be negative.')
        self._quantity = value  # (2)
    ...

apple_order.quantity = -10
# ValueError: Cannot be negative

```

We transformed `quantity` from a simple attribute to a non-negative property. Notice line (1) that the attribute are renamed to `_quantity` to avoid line (2) getting a `RecursionError`.

Are we done? Hell no. We forgot about the `price` attribute cannot be negative neither. It might be attempting to just create another property for `price`, but remember the DRY principle: when you find yourself doing the same thing twice, it's a good sign to extract the reusable code. Also, in our example, there might be more attributes need to be added into this class in the future. Repeating the code isn't fun for the writer or the reader. Let's see how to use descriptors to help us.

How to write descriptors

With the descriptors in place, our new class definition would become:

```

class Order:
    price = NonNegative('price')  # (3)
    quantity = NonNegative('quantity')

    def __init__(self, name, price, quantity):
        self._name = name
        self.price = price
        self.quantity = quantity

    def total(self):

```

```

        return self.price * self.quantity

apple_order = Order('apple', 1, 10)
apple_order.total()
# 10
apple_order.price = -10
# ValueError: Cannot be negative
apple_order.quantity = -10
# ValueError: Cannot be negative

```

Notice the class attributes defined before the `__init__` method? It's a lot like the SQLAlchemy example showed on the very beginning of this post. This is where we are heading. We need to define the `NonNegative` class and implement the descriptor protocols. Here's how:

```

class NonNegative:
    def __init__(self, name):
        self.name = name # (4)
    def __get__(self, instance, owner):
        return instance.__dict__[self.name] # (5)
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Cannot be negative.')
        instance.__dict__[self.name] = value # (6)

```

Line (4) : the `name` attribute is needed because when the `NonNegative` object is created on line (3), the assignment to attribute named `price` hasn't happen yet. Thus, we need to explicitly pass the name `price` to the initializer of the object to use as the key for the instance's `__dict__`.

Later, we'll see how in Python 3.6+ we can avoid the redundancy.

The redundancy could be avoid in earlier versions of Python, but I think this would take too much effort to explain and is not the purpose of this post. Thus, not included.

Line (5) and (6) : instead of using builtin function `getattr` and `setattr`, we need to reach into the `__dict__` object directly, because the builtins would be intercepted by the descriptor protocols too and cause the `RecursionError`.

Welcome to Python 3.6+

We are still repeating ourself in line (3). How do I get a cleaner API to use such

that we write:

```
class Order:
    price = NonNegative()
    quantity = NonNegative()

    def __init__(self, name, price, quantity):
        ...
```

Let's look at the new descriptor protocol in Python 3.6:

- `object.__set_name__(self, owner, name)`
- Called at the time the owning class owner is created. The descriptor has been assigned to name.

With this protocol, we could remove the `__init__` and bind the attribute name to the descriptor:

```
class NonNegative:
    ...
    def __set_name__(self, owner, name):
        self.name = name
```

To put all the codes together:

```
class NonNegative:
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Cannot be negative.')
        instance.__dict__[self.name] = value
    def __set_name__(self, owner, name):
        self.name = name

class Order:
    price = NonNegative()
    quantity = NonNegative()

    def __init__(self, name, price, quantity):
        self._name = name
        self._price = price
```

```
        self.quantity = quantity

    def total(self):
        return self.price * self.quantity

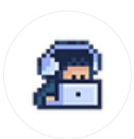
apple_order = Order('apple', 1, 10)
apple_order.total()
# 10
apple_order.price = -10
# ValueError: Cannot be negative
apple_order.quantity = -10
# ValueError: Cannot be negative
```

Conclusion

Python is a general purpose programming language. I love that it not only has very powerful features that are highly flexible and could possibly bend the language tremendously (e.g. Meta Classes,) but also has high-level APIs/protocols to serve 99% of the needs (e.g. Descriptors.) I believe there's the right tool for the job. Descriptors are clearly the right tool for binding behaviors to attributes. Although Meta Classes could potentially do the same thing, Descriptor could solve the problem more gracefully. It's also pleasing to see Python evolve for serving general people's needs better.

Here's my conclusion:

1. Python 3.6 is by far the greatest Python.
2. Descriptors are used to bind behaviors to accessing attributes.

[Python](#)[Software Development](#)[Software Engineering](#)[Follow](#)

Written by Daw-Ran Liou

581 Followers · 311 Following

Developer dawranliou.com

Responses (2)



Write a response

What are your thoughts?



Umesh

Apr 7, 2019



simple yet explained in a concise manner



[Reply](#)



Felix Stephen

Jan 4, 2019

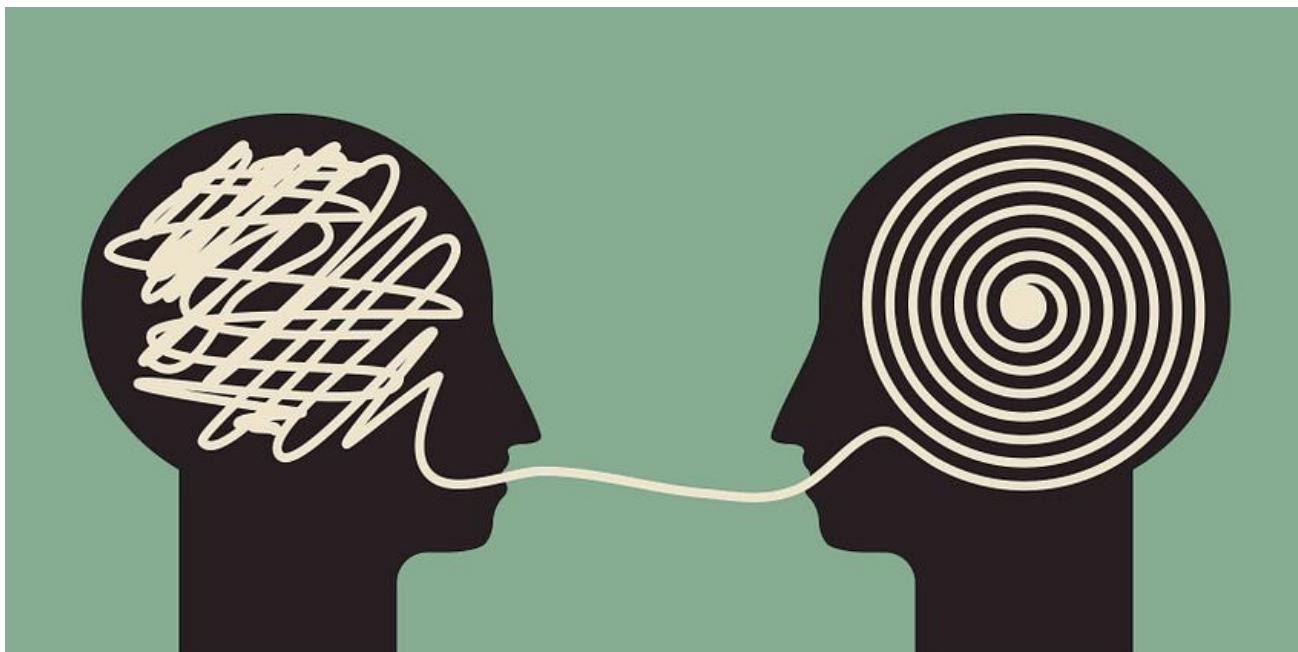


Great :)



[Reply](#)

More from Daw-Ran Liou

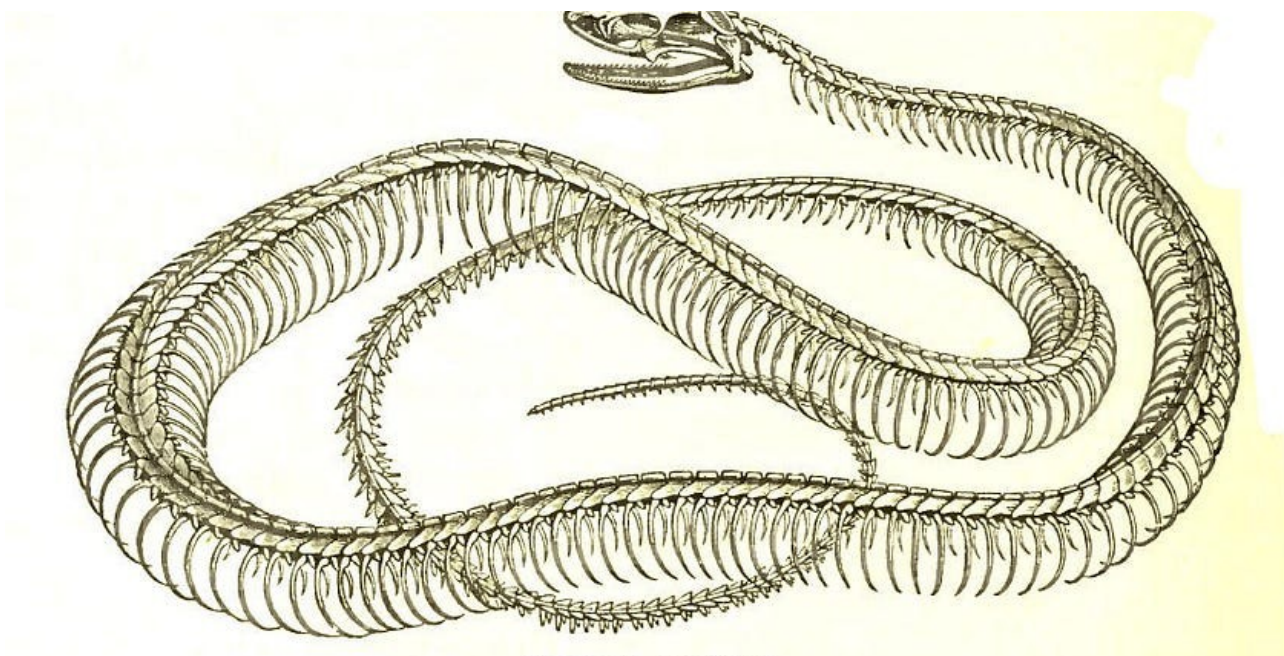


 In Python Pandemonium by Daw-Ran Liou

You (Probably) Don't Need For-Loops

The original title was “Never Write For-Loops Again” but I think it misled people to think that for-loops are bad. This wasn't my intent...

Nov 18, 2016  1.2K  22



 Daw-Ran Liou


Getting Started with Python Internals

...e what's happening **right now**

 349  1 

Daw-Ran Liou @dawranliou

...ple for @dawranliou

 Daw-Ran Liou

Twitter scraper tutorial with Python: Requests, BeautifulSoup, and Selenium — Part 1

Inspired by Bruce, my friend's take-home interview question, I started this bite-size project to play around with some of the most popular...


Mar 26, 2016  422  6



 Daw-Ran Liou

Composing namedtuple from namedtuples

Sharing the recipe of composing namedtuple from namedtuples.

Nov 2, 2017  56



See all from Daw-Ran Liou

Recommended from Medium



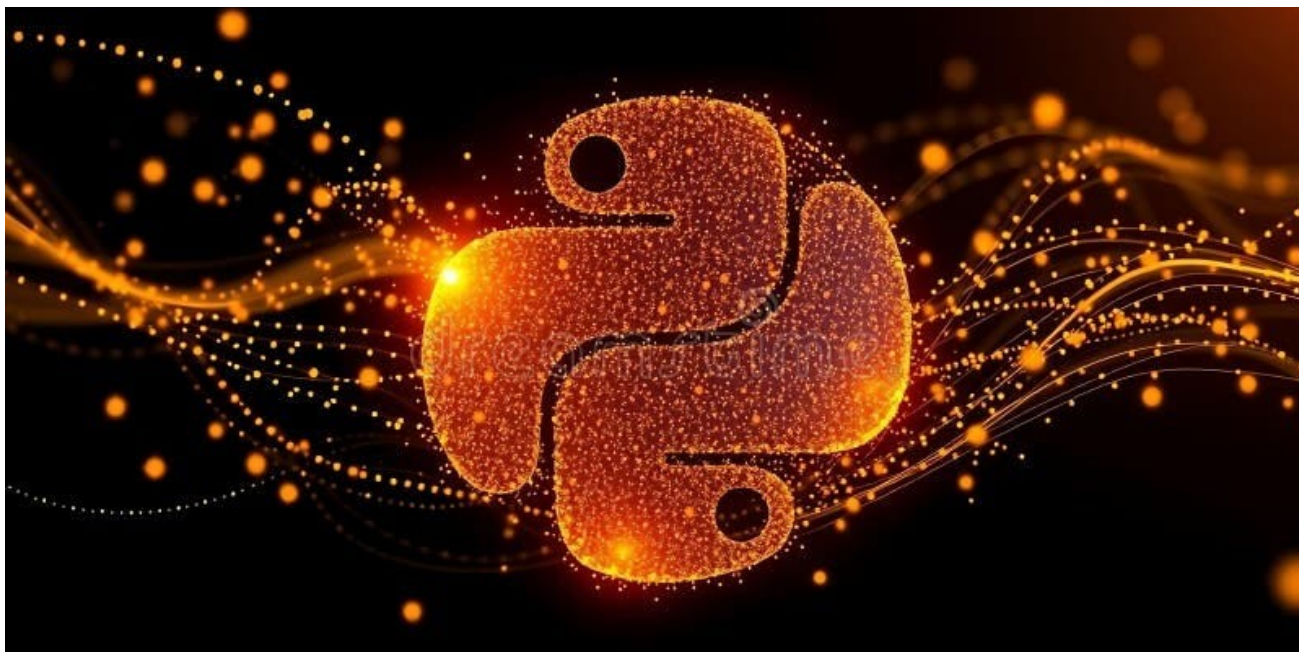
In Pythonic AF by Aysha R

I Tried Writing Python in VS Code and PyCharm—Here's What I Found

One felt like a smart coding companion. The other felt like assembling IKEA furniture blindfolded.

★ 6d ago  684  61






Ethanza

10 Python Tips and Tricks to Write Cleaner, Faster Code





Python's simplicity and versatility make it a favorite among developers, but mastering its nuances can elevate your code from functional to...

Apr 11



 In TechToFreedom by Yang Zhou


11 Outdated Python Modules That You Should Never Use Again

Tool	uv Equivalent	Improvement	
  792  23	<code>uv pip install</code>	8–10x faster	
<code>/ venv</code>	<code>uv venv</code>	Instant creation	
<code>e</code>	<code>uv pip compile</code>	Faster resolution and lockfile support	
	<code>uv pip sync</code>	Reproducible installs	
<code>ts.txt</code>	<code>uv</code> uses <code>requirements.lock</code>	Modern lockfile support	

 Nafiul Khan Earth

NextGen Python Development using uv

What is uv?

5d ago  4



```
class MagicExample:
    def __init__(self, value):
        self.value = value # Initialize the instance variable

    def __str__(self):
        return f"MagicExample with value {self.value}" # String representation

# Creating an instance of MagicExample
obj = MagicExample(10)

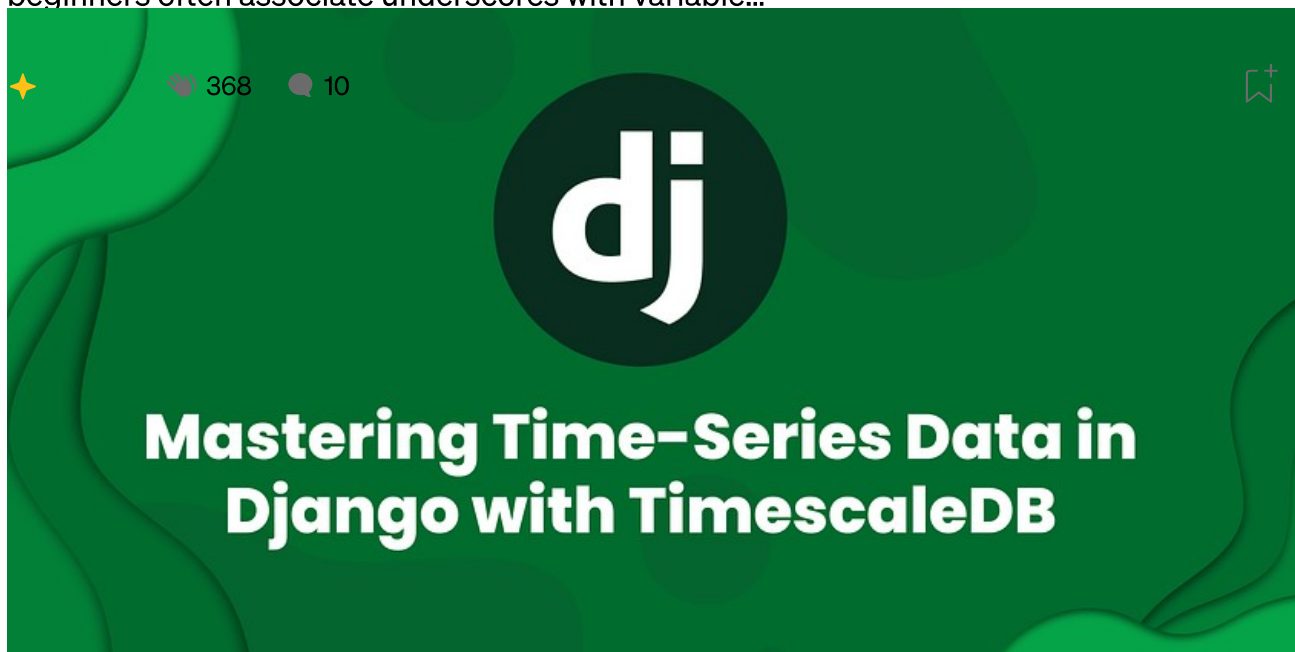
# Printing the object, which calls __str__()
print(obj)
```



In Coding Nexus by Algo Insights

90% of Python programmers don't know these seven uses of underscore

Underscores (_) in Python serve more purposes than most developers realize. While beginners often associate underscores with variable...



In Django Unleashed by Sanjay Prajapati

Mastering Time-Series Data in Django with TimescaleDB 🚀

🚀 Learn how to store & query time-series data in Django using TimescaleDB. Optimize PostgreSQL for real-time analytics! 📊



Mar 25 🖱️ 28

[See more recommendations](#)