CHAPTER 23
Attribute Descriptors

> Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.
>
> — Raymond Hettinger, Python core developer and guru[1]

Descriptors are a way of reusing the same access logic in multiple attributes. For example, field types in ORMs, such as the Django ORM and SQLAlchemy, are descriptors, managing the flow of data from the fields in a database record to Python object attributes and vice versa.

A descriptor is a class that implements a dynamic protocol consisting of the `__get__`, `__set__`, and `__delete__` methods. The `property` class implements the full descriptor protocol. As usual with dynamic protocols, partial implementations are OK. In fact, most descriptors we see in real code implement only `__get__` and `__set__`, and many implement only one of these methods.

Descriptors are a distinguishing feature of Python, deployed not only at the application level but also in the language infrastructure. User-defined functions are descriptors. We'll see how the descriptor protocol allows methods to operate as bound or unbound methods, depending on how they are called.

Understanding descriptors is key to Python mastery. This is what this chapter is about.

In this chapter we'll refactor the bulk food example we first saw in "Using a Property for Attribute Validation" on page 857, replacing properties with descriptors. This will make it easier to reuse the attribute validation logic across different classes. We'll

1 Raymond Hettinger, *Descriptor HowTo Guide*.

tackle the concepts of overriding and nonoverriding descriptors, and realize that Python functions are descriptors. Finally we'll see some tips about implementing descriptors.

What's New in This Chapter

The `Quantity` descriptor example in "LineItem Take #4: Automatic Naming of Storage Attributes" on page 887 was dramatically simplified thanks to the `__set_name__` special method added to the descriptor protocol in Python 3.6.

I removed the property factory example formerly in "LineItem Take #4: Automatic Naming of Storage Attributes" on page 887 because it became irrelevant: the point was to show an alternative way of solving the `Quantity` problem, but with the addition of `__set_name__`, the descriptor solution becomes much simpler.

The `AutoStorage` class that used to appear in "LineItem Take #5: A New Descriptor Type" on page 889 is also gone because `__set_name__` made it obsolete.

Descriptor Example: Attribute Validation

As we saw in "Coding a Property Factory" on page 865, a property factory is a way to avoid repetitive coding of getters and setters by applying functional programming patterns. A property factory is a higher-order function that creates a parameterized set of accessor functions and builds a custom property instance from them, with closures to hold settings like the `storage_name`. The object-oriented way of solving the same problem is a descriptor class.

We'll continue the series of `LineItem` examples where we left off, in "Coding a Property Factory" on page 865, by refactoring the `quantity` property factory into a `Quantity` descriptor class. This will make it easier to use.

LineItem Take #3: A Simple Descriptor

As we said in the introduction, a class implementing a `__get__`, a `__set__`, or a `__delete__` method is a descriptor. You use a descriptor by declaring instances of it as class attributes of another class.

We'll create a `Quantity` descriptor, and the `LineItem` class will use two instances of `Quantity`: one for managing the `weight` attribute, the other for `price`. A diagram helps, so take a look at Figure 23-1.

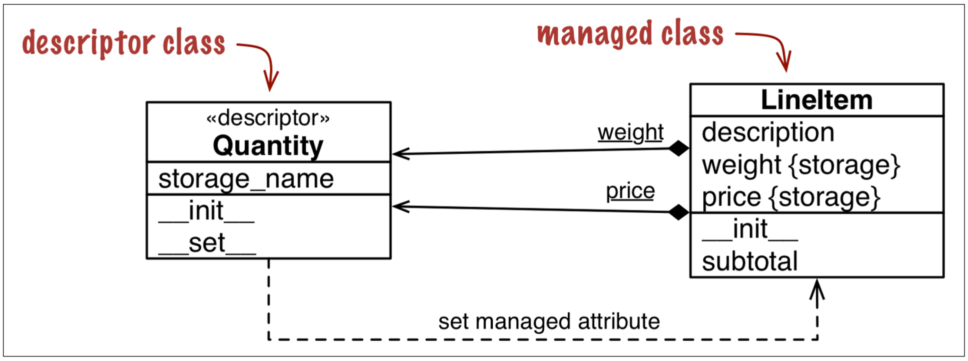


Figure 23-1. UML class diagram for `LineItem` using a descriptor class named `Quantity`. Underlined attributes in UML are class attributes. Note that weight and price are instances of `Quantity` attached to the `LineItem` class, but `LineItem` instances also have their own weight and price attributes where those values are stored.

Note that the word `weight` appears twice in Figure 23-1, because there are really two distinct attributes named `weight`: one is a class attribute of `LineItem`, the other is an instance attribute that will exist in each `LineItem` object. This also applies to `price`.

Terms to understand descriptors

Implementing and using descriptors involves several components, and it is useful to be precise when naming those components. I will use the following terms and definitions as I describe the examples in this chapter. They will be easier to understand once you see the code, but I wanted to put the definitions up front so you can refer back to them when needed.

Descriptor class
: A class implementing the descriptor protocol. That's `Quantity` in Figure 23-1.

Managed class
: The class where the descriptor instances are declared as class attributes. In Figure 23-1, `LineItem` is the managed class.

Descriptor instance
: Each instance of a descriptor class, declared as a class attribute of the managed class. In Figure 23-1, each descriptor instance is represented by a composition arrow with an underlined name (the underline means class attribute in UML). The black diamonds touch the `LineItem` class, which contains the descriptor instances.

Managed instance
: One instance of the managed class. In this example, `LineItem` instances are the managed instances (they are not shown in the class diagram).

Storage attribute
: An attribute of the managed instance that holds the value of a managed attribute for that particular instance. In Figure 23-1, the `LineItem` instance attributes `weight` and `price` are the storage attributes. They are distinct from the descriptor instances, which are always class attributes.

Managed attribute
: A public attribute in the managed class that is handled by a descriptor instance, with values stored in storage attributes. In other words, a descriptor instance and a storage attribute provide the infrastructure for a managed attribute.

It's important to realize that `Quantity` instances are class attributes of `LineItem`. This crucial point is highlighted by the mills and gizmos in Figure 23-2.

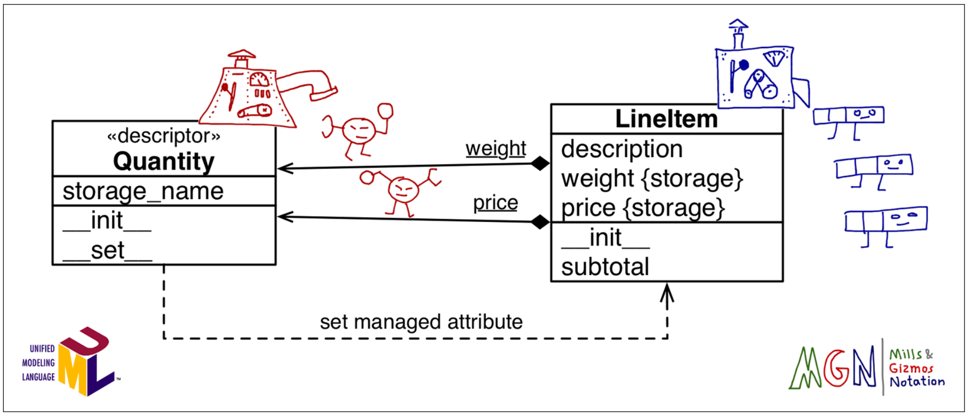


Figure 23-2. UML class diagram annotated with MGN (Mills & Gizmos Notation): classes are mills that produce gizmos—the instances. The `Quantity` mill produces two gizmos with round heads, which are attached to the `LineItem` mill: weight and price. The `LineItem` mill produces rectangular gizmos that have their own weight and price attributes where those values are stored.

Introducing Mills & Gizmos Notation

After explaining descriptors many times, I realized UML is not very good at showing relationships involving classes and instances, like the relationship between a managed class and the descriptor instances.[2] So I invented my own "language," the Mills & Gizmos Notation (MGN), which I use to annotate UML diagrams.

MGN is designed to make very clear the distinction between classes and instances. See Figure 23-3. In MGN, a class is drawn as a "mill," a complicated machine that produces gizmos. Classes/mills are always machines with levers and dials. The gizmos are the instances, and they look much simpler. When this book is rendered in color, gizmos have the same color as the mill that made it.

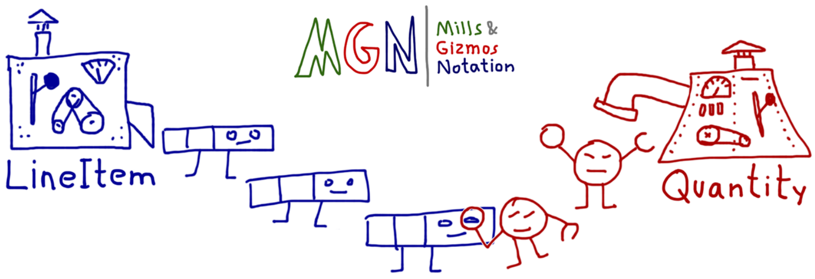


Figure 23-3. MGN sketch showing the `LineItem` class making three instances, and `Quantity` making two. One instance of `Quantity` is retrieving a value stored in a `LineItem` instance.

For this example, I drew `LineItem` instances as rows in a tabular invoice, with three cells representing the three attributes (`description`, `weight`, and `price`). Because `Quantity` instances are descriptors, they have a magnifying glass to `__get__` values, and a claw to `__set__` values. When we get to metaclasses, you'll thank me for these doodles.

Enough doodling for now. Here is the code: Example 23-1 shows the `Quantity` descriptor class, and Example 23-2 lists a new `LineItem` class using two instances of `Quantity`.

2 Classes and instances are drawn as rectangles in UML class diagrams. There are visual differences, but instances are rarely shown in class diagrams, so developers may not recognize them as such.

Example 23-1. bulkfood_v3.py: `Quantity` descriptor does not accept negative values

```
class Quantity:  ❶

    def __init__(self, storage_name):
        self.storage_name = storage_name  ❷

    def __set__(self, instance, value):  ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value  ❹
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

    def __get__(self, instance, owner):  ❺
        return instance.__dict__[self.storage_name]
```

❶ Descriptor is a protocol-based feature; no subclassing is needed to implement one.

❷ Each `Quantity` instance will have a `storage_name` attribute: that's the name of the storage attribute to hold the value in the managed instances.

❸ `__set__` is called when there is an attempt to assign to the managed attribute. Here, `self` is the descriptor instance (i.e., `LineItem.weight` or `LineItem.price`), `instance` is the managed instance (a `LineItem` instance), and `value` is the value being assigned.

❹ We must store the attribute value directly into `__dict__`; calling `set attr(instance, self.storage_name)` would trigger the `__set__` method again, leading to infinite recursion.

❺ We need to implement `__get__` because the name of the managed attribute may not be the same as the `storage_name`. The `owner` argument will be explained shortly.

Implementing `__get__` is necessary because a user could write something like this:

```
class House:
    rooms = Quantity('number_of_rooms')
```

In the `House` class, the managed attribute is `rooms`, but the storage attribute is `number_of_rooms`. Given a `House` instance named `chaos_manor`, reading and writing `chaos_manor.rooms` goes through the `Quantity` descriptor instance attached to `rooms`, but reading and writing `chaos_manor.number_of_rooms` bypasses the descriptor.

Note that `__get__` receives three arguments: `self`, `instance`, and `owner`. The `owner` argument is a reference to the managed class (e.g., `LineItem`), and it's useful if you want the descriptor to support retrieving a class attribute—perhaps to emulate Python's default behavior of retrieving a class attribute when the name is not found in the instance.

If a managed attribute, such as `weight`, is retrieved via the class like `LineItem.weight`, the descriptor `__get__` method receives `None` as the value for the `instance` argument.

To support introspection and other metaprogramming tricks by the user, it's a good practice to make `__get__` return the descriptor instance when the managed attribute is accessed through the class. To do that, we'd code `__get__` like this:

```
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.storage_name]
```

Example 23-2 demonstrates the use of `Quantity` in `LineItem`.

Example 23-2. bulkfood_v3.py: `Quantity` descriptors manage attributes in `LineItem`

```
class LineItem:
    weight = Quantity('weight')  ❶
    price = Quantity('price')  ❷

    def __init__(self, description, weight, price):  ❸
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ The first descriptor instance will manage the `weight` attribute.

❷ The second descriptor instance will manage the `price` attribute.

❸ The rest of the class body is as simple and clean as the original code in *bulkfood_v1.py* (Example 22-19).

The code in Example 23-2 works as intended, preventing the sale of truffles for $0:[3]

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

When coding descriptor `__get__` and `__set__` methods, keep in mind what the `self` and `instance` arguments mean: `self` is the descriptor instance, and `instance` is the managed instance. Descriptors managing instance attributes should store values in the managed instances. That's why Python provides the `instance` argument to the descriptor methods.

It may be tempting, but wrong, to store the value of each managed attribute in the descriptor instance itself. In other words, in the `__set__` method, instead of coding:

```
        instance.__dict__[self.storage_name] = value
```

the tempting, but bad, alternative would be:

```
        self.__dict__[self.storage_name] = value
```

To understand why this would be wrong, think about the meaning of the first two arguments to `__set__`: `self` and `instance`. Here, `self` is the descriptor instance, which is actually a class attribute of the managed class. You may have thousands of `LineItem` instances in memory at one time, but you'll only have two instances of the descriptors: the class attributes `LineItem.weight` and `LineItem.price`. So anything you store in the descriptor instances themselves is actually part of a `LineItem` class attribute, and therefore is shared among all `LineItem` instances.

A drawback of Example 23-2 is the need to repeat the names of the attributes when the descriptors are instantiated in the managed class body. It would be nice if the `LineItem` class could be declared like this:

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # remaining methods as before
```

As it stands, Example 23-2 requires naming each `Quantity` explicitly, which is not only inconvenient but dangerous. If a programmer copying and pasting code forgets to edit both names and writes something like `price = Quantity('weight')`, the

3 White truffles cost thousands of dollars per pound. Disallowing the sale of truffles for $0.01 is left as an exercise for the enterprising reader. I know a person who actually bought an $1,800 encyclopedia of statistics for $18 because of an error in an online store (not *Amazon.com* in this case).

program will misbehave badly, clobbering the value of `weight` whenever the `price` is set.

The problem is that—as we saw in Chapter 6—the righthand side of an assignment is executed before the variable exists. The expression `Quantity()` is evaluated to create a descriptor instance, and there is no way the code in the `Quantity` class can guess the name of the variable to which the descriptor will be bound (e.g., `weight` or `price`).

Thankfully, the descriptor protocol now supports the aptly named `__set_name__` special method. We'll see how to use it next.

Automatic naming of a descriptor storage attribute used to be a thorny issue. In the first edition of *Fluent Python*, I devoted several pages and lines of code in this chapter and the next to presenting different solutions, including the use of a class decorator, and then metaclasses in Chapter 24. This was greatly simplified in Python 3.6.

LineItem Take #4: Automatic Naming of Storage Attributes

To avoid retyping the attribute name in the descriptor instances, we'll implement `__set_name__` to set the `storage_name` of each `Quantity` instance. The `__set_name__` special method was added to the descriptor protocol in Python 3.6. The interpreter calls `__set_name__` on each descriptor it finds in a `class` body—if the descriptor implements it.[4]

In Example 23-3, the `LineItem` descriptor class doesn't need an `__init__`. Instead, `__set_item__` saves the name of the storage attribute.

Example 23-3. bulkfood_v4.py: `__set_name__` sets the name for each `Quantity` descriptor instance

```
class Quantity:

    def __set_name__(self, owner, name):  ❶
        self.storage_name = name          ❷

    def __set__(self, instance, value):   ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value
```

[4] More precisely, `__set_name__` is called by `type.__new__`—the constructor of objects representing classes. The `type` built-in is actually a metaclass, the default class of user-defined classes. This is hard to grasp at first, but rest assured: Chapter 24 is devoted to the dynamic configuration of classes, including the concept of metaclasses.

```
        else:
            msg = f'{self.storage_name} must be > 0'
            raise ValueError(msg)

    # no __get__ needed ❹

class LineItem:
    weight = Quantity()  ❺
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `self` is the descriptor instance (not the managed instance), `owner` is the managed class, and `name` is the name of the attribute of `owner` to which this descriptor instance was assigned in the class body of `owner`.

❷ This is what the `__init__` did in Example 23-1.

❸ The `__set__` method here is exactly the same as in Example 23-1.

❹ Implementing `__get__` is not necessary because the name of the storage attribute matches the name of the managed attribute. The expression `product.price` gets the `price` attribute directly from the `LineItem` instance.

❺ Now we don't need to pass the managed attribute name to the `Quantity` constructor. That was the goal for this version.

Looking at Example 23-3, you may think that's a lot of code just for managing a couple of attributes, but it's important to realize that the descriptor logic is now abstracted into a separate code unit: the `Quantity` class. Usually we do not define a descriptor in the same module where it's used, but in a separate utility module designed to be used across the application—even in many applications, if you are developing a library or framework.

With this in mind, Example 23-4 better represents the typical usage of a descriptor.

Example 23-4. bulkfood_v4c.py: `LineItem` definition uncluttered; the `Quantity` descriptor class now resides in the imported `model_v4c` module

```
import model_v4c as model  ❶
```

```
class LineItem:
    weight = model.Quantity()  ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Import the `model_v4c` module where `Quantity` is implemented.

❷ Put `model.Quantity` to use.

Django users will notice that Example 23-4 looks a lot like a model definition. It's no coincidence: Django model fields are descriptors.

Because descriptors are implemented as classes, we can leverage inheritance to reuse some of the code we have for new descriptors. That's what we'll do in the following section.

LineItem Take #5: A New Descriptor Type

The imaginary organic food store hits a snag: somehow a line item instance was created with a blank description, and the order could not be fulfilled. To prevent that, we'll create a new descriptor, `NonBlank`. As we design `NonBlank`, we realize it will be very much like the `Quantity` descriptor, except for the validation logic.

This prompts a refactoring, producing `Validated`, an abstract class that overrides the `__set__` method, calling a `validate` method that must be implemented by subclasses.

We'll then rewrite `Quantity`, and implement `NonBlank` by inheriting from `Validated` and just coding the `validate` methods.

The relationship among `Validated`, `Quantity`, and `NonBlank` is an application of the *template method* as described in the *Design Patterns* classic:

> A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.[5]

5 Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 326.

In Example 23-5, `Validated.__set__` is the template method and `self.validate` is the abstract operation.

Example 23-5. model_v5.py: the `Validated` ABC

```
import abc

class Validated(abc.ABC):

    def __set_name__(self, owner, name):
        self.storage_name = name

    def __set__(self, instance, value):
        value = self.validate(self.storage_name, value)  ❶
        instance.__dict__[self.storage_name] = value  ❷

    @abc.abstractmethod
    def validate(self, name, value):  ❸
        """return validated value or raise ValueError"""
```

❶ `__set__` delegates validation to the `validate` method…

❷ …then uses the returned `value` to update the stored value.

❸ `validate` is an abstract method; this is the template method.

Alex Martelli prefers to call this design pattern *Self-Delegation*, and I agree it's a more descriptive name: the first line of `__set__` self-delegates to `validate`.[6]

The concrete `Validated` subclasses in this example are `Quantity` and `NonBlank`, shown in Example 23-6.

Example 23-6. model_v5.py: `Quantity` and `NonBlank`, concrete `Validated` subclasses

```
class Quantity(Validated):
    """a number greater than zero"""

    def validate(self, name, value):  ❶
        if value <= 0:
            raise ValueError(f'{name} must be > 0')
        return value


class NonBlank(Validated):
    """a string with at least one non-space character"""
```

6 Slide #50 of Alex Martelli's "Python Design Patterns" talk. Highly recommended.

```
    def validate(self, name, value):
        value = value.strip()
        if not value:  ❷
            raise ValueError(f'{name} cannot be blank')
        return value  ❸
```

❶ Implementation of the template method required by the `Validated.validate` abstract method.

❷ If nothing is left after leading and trailing blanks are stripped, reject the value.

❸ Requiring the concrete `validate` methods to return the validated value gives them an opportunity to clean up, convert, or normalize the data received. In this case, `value` is returned without leading or trailing blanks.

Users of *model_v5.py* don't need to know all these details. What matters is that they get to use `Quantity` and `NonBlank` to automate the validation of instance attributes. See the latest `LineItem` class in Example 23-7.

Example 23-7. bulkfood_v5.py: `LineItem` using `Quantity` and `NonBlank` descriptors

```
import model_v5 as model  ❶

class LineItem:
    description = model.NonBlank()  ❷
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Import the `model_v5` module, giving it a friendlier name.

❷ Put `model.NonBlank` to use. The rest of the code is unchanged.

The `LineItem` examples we've seen in this chapter demonstrate a typical use of descriptors to manage data attributes. Descriptors like `Quantity` are called overriding descriptors because its `__set__` method overrides (i.e., intercepts and overrules) the setting of an instance attribute by the same name in the managed instance. However, there are also nonoverriding descriptors. We'll explore this distinction in detail in the next section.

Overriding Versus Nonoverriding Descriptors

Recall that there is an important asymmetry in the way Python handles attributes. Reading an attribute through an instance normally returns the attribute defined in the instance, but if there is no such attribute in the instance, a class attribute will be retrieved. On the other hand, assigning to an attribute in an instance normally creates the attribute in the instance, without affecting the class at all.

This asymmetry also affects descriptors, in effect creating two broad categories of descriptors, depending on whether the `__set__` method is implemented. If `__set__` is present, the class is an overriding descriptor; otherwise, it is a nonoverriding descriptor. These terms will make sense as we study descriptor behaviors in the next examples.

Observing the different descriptor categories requires a few classes, so we'll use the code in Example 23-8 as our test bed for the following sections.

Every `__get__` and `__set__` method in Example 23-8 calls `print_args` so their invocations are displayed in a readable way. Understanding `print_args` and the auxiliary functions `cls_name` and `display` is not important, so don't get distracted by them.

Example 23-8. descriptorkinds.py: simple classes for studying descriptor overriding behaviors

```
### auxiliary functions for display only ###

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return f'<class {obj.__name__}>'
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return f'<{cls_name(obj)} object>'

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print(f'-> {cls_name(args[0])}.__{name}__({pseudo_args})')


### essential classes for this example ###
```

```
class Overriding:  ❶
    """a.k.a. data descriptor or enforced descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)  ❷

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class OverridingNoGet:  ❸
    """an overriding descriptor without ``__get__``"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class NonOverriding:  ❹
    """a.k.a. non-data or shadowable descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)


class Managed:  ❺
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()

    def spam(self):  ❻
        print(f'-> Managed.spam({display(self)})')
```

❶ An overriding descriptor class with `__get__` and `__set__`.

❷ The `print_args` function is called by every descriptor method in this example.

❸ An overriding descriptor without a `__get__` method.

❹ No `__set__` method here, so this is a nonoverriding descriptor.

❺ The managed class, using one instance of each of the descriptor classes.

❻ The `spam` method is here for comparison, because methods are also descriptors.

In the following sections, we will examine the behavior of attribute reads and writes on the `Managed` class, and one instance of it, going through each of the different descriptors defined.

Overriding Descriptors

A descriptor that implements the `__set__` method is an *overriding descriptor*, because although it is a class attribute, a descriptor implementing `__set__` will override attempts to assign to instance attributes. This is how Example 23-3 was implemented. Properties are also overriding descriptors: if you don't provide a setter function, the default `__set__` from the `property` class will raise `AttributeError` to signal that the attribute is read-only. Given the code in Example 23-8, experiments with an overriding descriptor can be seen in Example 23-9.

Python contributors and authors use different terms when discussing these concepts. I adopted "overriding descriptor" from the book *Python in a Nutshell*. The official Python documentation uses "data descriptor," but "overriding descriptor" highlights the special behavior. Overriding descriptors are also called "enforced descriptors." Synonyms for nonoverriding descriptors include "nondata descriptors" or "shadowable descriptors."

Example 23-9. Behavior of an overriding descriptor

```
>>> obj = Managed()  ❶
>>> obj.over  ❷
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> Managed.over  ❸
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7  ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over  ❺
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> obj.__dict__['over'] = 8  ❻
>>> vars(obj)  ❼
{'over': 8}
>>> obj.over  ❽
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
```

❶ Create `Managed` object for testing.

❷ `obj.over` triggers the descriptor `__get__` method, passing the managed instance `obj` as the second argument.

❸ `Managed.over` triggers the descriptor `__get__` method, passing `None` as the second argument (`instance`).

❹ Assigning to `obj.over` triggers the descriptor `__set__` method, passing the value `7` as the last argument.

❺ Reading `obj.over` still invokes the descriptor `__get__` method.

❻ Bypassing the descriptor, setting a value directly to the `obj.__dict__`.

❼ Verify that the value is in the `obj.__dict__`, under the `over` key.

❽ However, even with an instance attribute named `over`, the `Managed.over` descriptor still overrides attempts to read `obj.over`.

Overriding Descriptor Without __get__

Properties and other overriding descriptors, such as Django model fields, implement both `__set__` and `__get__`, but it's also possible to implement only `__set__`, as we saw in Example 23-2. In this case, only writing is handled by the descriptor. Reading the descriptor through an instance will return the descriptor object itself because there is no `__get__` to handle that access. If a namesake instance attribute is created with a new value via direct access to the instance `__dict__`, the `__set__` method will still override further attempts to set that attribute, but reading that attribute will simply return the new value from the instance, instead of returning the descriptor object. In other words, the instance attribute will shadow the descriptor, but only when reading. See Example 23-10.

Example 23-10. Overriding descriptor without `__get__`

```
>>> obj.over_no_get  ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get  ❷
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7  ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get  ❹
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9  ❺
>>> obj.over_no_get  ❻
9
>>> obj.over_no_get = 7  ❼
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get  ❽
9
```

❶ This overriding descriptor doesn't have a `__get__` method, so reading `obj.over_no_get` retrieves the descriptor instance from the class.

❷ The same thing happens if we retrieve the descriptor instance directly from the managed class.

❸ Trying to set a value to `obj.over_no_get` invokes the `__set__` descriptor method.

❹ Because our `__set__` doesn't make changes, reading `obj.over_no_get` again retrieves the descriptor instance from the managed class.

❺ Going through the instance `__dict__` to set an instance attribute named `over_no_get`.

❻ Now that `over_no_get` instance attribute shadows the descriptor, but only for reading.

❼ Trying to assign a value to `obj.over_no_get` still goes through the descriptor set.

❽ But for reading, that descriptor is shadowed as long as there is a namesake instance attribute.

Nonoverriding Descriptor

A descriptor that does not implement `__set__` is a nonoverriding descriptor. Setting an instance attribute with the same name will shadow the descriptor, rendering it ineffective for handling that attribute in that specific instance. Methods and `@functools.cached_property` are implemented as nonoverriding descriptors. Example 23-11 shows the operation of a nonoverriding descriptor.

Example 23-11. Behavior of a nonoverriding descriptor

```
>>> obj = Managed()
>>> obj.non_over  ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
>>> obj.non_over = 7  ❷
>>> obj.non_over  ❸
7
>>> Managed.non_over  ❹
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over  ❺
>>> obj.non_over  ❻
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
```

❶ `obj.non_over` triggers the descriptor `__get__` method, passing `obj` as the second argument.

❷ `Managed.non_over` is a nonoverriding descriptor, so there is no `__set__` to interfere with this assignment.

❸ The `obj` now has an instance attribute named `non_over`, which shadows the namesake descriptor attribute in the `Managed` class.

❹ The `Managed.non_over` descriptor is still there, and catches this access via the class.

❺ If the `non_over` instance attribute is deleted…

❻ …then reading `obj.non_over` hits the `__get__` method of the descriptor in the class, but note that the second argument is the managed instance.

In the previous examples, we saw several assignments to an instance attribute with the same name as a descriptor, and different results according to the presence of a `__set__` method in the descriptor.

The setting of attributes in the class cannot be controlled by descriptors attached to the same class. In particular, this means that the descriptor attributes themselves can be clobbered by assigning to the class, as the next section explains.

Overwriting a Descriptor in the Class

Regardless of whether a descriptor is overriding or not, it can be overwritten by assignment to the class. This is a monkey-patching technique, but in Example 23-12 the descriptors are replaced by integers, which would effectively break any class that depended on the descriptors for proper operation.

Example 23-12. Any descriptor can be overwritten on the class itself

```
>>> obj = Managed()  ❶
>>> Managed.over = 1  ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over  ❸
(1, 2, 3)
```

❶ Create a new instance for later testing.

❷ Overwrite the descriptor attributes in the class.

❸ The descriptors are really gone.

Example 23-12 reveals another asymmetry regarding reading and writing attributes: although the reading of a class attribute can be controlled by a descriptor with `__get__` attached to the managed class, the writing of a class attribute cannot be handled by a descriptor with `__set__` attached to the same class.

In order to control the setting of attributes in a class, you have to attach descriptors to the class of the class—in other words, the metaclass. By default, the metaclass of user-defined classes is `type`, and you cannot add attributes to `type`. But in Chapter 24, we'll create our own metaclasses.

Let's now focus on how descriptors are used to implement methods in Python.

Methods Are Descriptors

A function within a class becomes a bound method when invoked on an instance because all user-defined functions have a `__get__` method, therefore they operate as descriptors when attached to a class. Example 23-13 demonstrates reading the `spam` method from the `Managed` class introduced in Example 23-8.

Example 23-13. A method is a nonoverriding descriptor

```
>>> obj = Managed()
>>> obj.spam  ❶
<bound method Managed.spam of <descriptorkinds.Managed object at 0x74c80c>>
>>> Managed.spam  ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7  ❸
>>> obj.spam
7
```

❶ Reading from `obj.spam` retrieves a bound method object.

❷ But reading from `Managed.spam` retrieves a function.

❸ Assigning a value to `obj.spam` shadows the class attribute, rendering the `spam` method inaccessible from the `obj` instance.

Functions do not implement `__set__`, therefore they are nonoverriding descriptors, as the last line of Example 23-13 shows.

The other key takeaway from Example 23-13 is that `obj.spam` and `Managed.spam` retrieve different objects. As usual with descriptors, the `__get__` of a function returns a reference to itself when the access happens through the managed class. But when the access goes through an instance, the `__get__` of the function returns a bound method object: a callable that wraps the function and binds the managed instance (e.g., `obj`) to the first argument of the function (i.e., `self`), like the `functools.partial` function does (as seen in "Freezing Arguments with functools.partial" on page 247). For a deeper understanding of this mechanism, take a look at Example 23-14.

Example 23-14. method_is_descriptor.py: a `Text` class, derived from `UserString`

```
import collections


class Text(collections.UserString):

    def __repr__(self):
        return 'Text({!r})'.format(self.data)

    def reverse(self):
        return self[::-1]
```

Now let's investigate the `Text.reverse` method. See Example 23-15.

Example 23-15. Experiments with a method

```
    >>> word = Text('forward')
    >>> word  ❶
    Text('forward')
    >>> word.reverse()  ❷
    Text('drawrof')
    >>> Text.reverse(Text('backward'))  ❸
    Text('drawkcab')
    >>> type(Text.reverse), type(word.reverse)  ❹
    (<class 'function'>, <class 'method'>)
    >>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')]))  ❺
    ['diaper', (30, 20, 10), Text('desserts')]
    >>> Text.reverse.__get__(word)  ❻
    <bound method Text.reverse of Text('forward')>
    >>> Text.reverse.__get__(None, Text)  ❼
    <function Text.reverse at 0x101244e18>
    >>> word.reverse  ❽
    <bound method Text.reverse of Text('forward')>
    >>> word.reverse.__self__  ❾
    Text('forward')
    >>> word.reverse.__func__ is Text.reverse  ❿
    True
```

❶ The `repr` of a `Text` instance looks like a `Text` constructor call that would make an equal instance.

❷ The `reverse` method returns the text spelled backward.

❸ A method called on the class works as a function.

❹ Note the different types: a `function` and a `method`.

❺ `Text.reverse` operates as a function, even working with objects that are not instances of `Text`.

❻ Any function is a nonoverriding descriptor. Calling its `__get__` with an instance retrieves a method bound to that instance.

❼ Calling the function's `__get__` with `None` as the `instance` argument retrieves the function itself.

❽ The expression `word.reverse` actually invokes `Text.reverse.__get__(word)`, returning the bound method.

❾ The bound method object has a `__self__` attribute holding a reference to the instance on which the method was called.

❿ The `__func__` attribute of the bound method is a reference to the original function attached to the managed class.

The bound method object also has a `__call__` method, which handles the actual invocation. This method calls the original function referenced in `__func__`, passing the `__self__` attribute of the method as the first argument. That's how the implicit binding of the conventional `self` argument works.

The way functions are turned into bound methods is a prime example of how descriptors are used as infrastructure in the language.

After this deep dive into how descriptors and methods work, let's go through some practical advice about their use.

Descriptor Usage Tips

The following list addresses some practical consequences of the descriptor characteristics just described:

Use `property` *to keep it simple*
: The `property` built-in creates overriding descriptors implementing `__set__` and `__get__` even if you do not define a setter method.[7] The default `__set__` of a property raises `AttributeError: can't set attribute`, so a property is the easiest way to create a read-only attribute, avoiding the issue described next.

7 A `__delete__` method is also provided by the `property` decorator, even if no deleter method is defined by you.

Read-only descriptors require `__set__`
: If you use a descriptor class to implement a read-only attribute, you must remember to code both `__get__` and `__set__`, otherwise setting a namesake attribute on an instance will shadow the descriptor. The `__set__` method of a read-only attribute should just raise `AttributeError` with a suitable message.[8]

Validation descriptors can work with `__set__` *only*
: In a descriptor designed only for validation, the `__set__` method should check the `value` argument it gets, and if valid, set it directly in the instance `__dict__` using the descriptor instance name as key. That way, reading the attribute with the same name from the instance will be as fast as possible, because it will not require a `__get__`. See the code for Example 23-3.

Caching can be done efficiently with `__get__` *only*
: If you code just the `__get__` method, you have a nonoverriding descriptor. These are useful to make some expensive computation and then cache the result by setting an attribute by the same name on the instance.[9] The namesake instance attribute will shadow the descriptor, so subsequent access to that attribute will fetch it directly from the instance `__dict__` and not trigger the descriptor `__get__` anymore. The `@functools.cached_property` decorator actually produces a nonoverriding descriptor.

Nonspecial methods can be shadowed by instance attributes
: Because functions and methods only implement `__get__`, they are nonoverriding descriptors. A simple assignment like `my_obj.the_method = 7` means that further access to `the_method` through that instance will retrieve the number 7—without affecting the class or other instances. However, this issue does not interfere with special methods. The interpreter only looks for special methods in the class itself, in other words, `repr(x)` is executed as `x.__class__.__repr__(x)`, so a `__repr__` attribute defined in `x` has no effect on `repr(x)`. For the same reason, the existence of an attribute named `__getattr__` in an instance will not subvert the usual attribute access algorithm.

The fact that nonspecial methods can be overridden so easily in instances may sound fragile and error prone, but I personally have never been bitten by this in more than 20 years of Python coding. On the other hand, if you are doing a lot of dynamic

8 Python is not consistent in such messages. Trying to change the `c.real` attribute of a `complex` number gets `AttributeError: readonly attribute`, but an attempt to change `c.conjugate` (a method of `complex`), results in `AttributeError: 'complex' object attribute 'conjugate' is read-only`. Even the spelling of "read-only" is different.

9 However, recall that creating instance attributes after the `__init__` method runs defeats the key-sharing memory optimization, as discussed in from "Practical Consequences of How dict Works" on page 102.

attribute creation, where the attribute names come from data you don't control (as we did in the earlier parts of this chapter), then you should be aware of this and perhaps implement some filtering or escaping of the dynamic attribute names to preserve your sanity.

The `FrozenJSON` class in Example 22-5 is safe from instance attribute shadowing methods because its only methods are special methods and the `build` class method. Class methods are safe as long as they are always accessed through the class, as I did with `FrozenJSON.build` in Example 22-5—later replaced by `__new__` in Example 22-6. The `Record` and `Event` classes presented in "Computed Properties" on page 845 are also safe: they implement only special methods, static methods, and properties. Properties are overriding descriptors, so they are not shadowed by instance attributes.

To close this chapter, we'll cover two features we saw with properties that we have not addressed in the context of descriptors: documentation and handling attempts to delete a managed attribute.

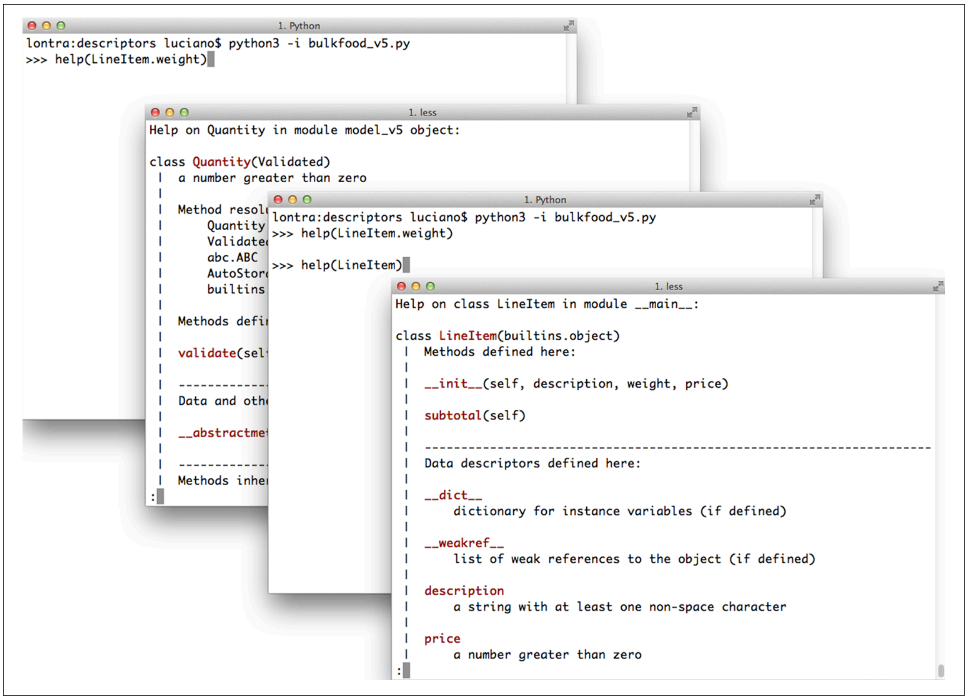
Descriptor Docstring and Overriding Deletion

The docstring of a descriptor class is used to document every instance of the descriptor in the managed class. Figure 23-4 shows the help displays for the `LineItem` class with the `Quantity` and `NonBlank` descriptors from Examples 23-6 and 23-7.

That is somewhat unsatisfactory. In the case of `LineItem`, it would be good to add, for example, the information that `weight` must be in kilograms. That would be trivial with properties, because each property handles a specific managed attribute. But with descriptors, the same `Quantity` descriptor class is used for `weight` and `price`.[10]

The second detail we discussed with properties, but have not addressed with descriptors, is handling attempts to delete a managed attribute. That can be done by implementing a `__delete__` method alongside or instead of the usual `__get__` and/or `__set__` in the descriptor class. I deliberately omitted coverage of `__delete__` because I believe real-world usage is rare. If you need this, please see the "Implementing Descriptors" section of the Python Data Model documentation. Coding a silly descriptor class with `__delete__` is left as an exercise to the leisurely reader.

10 Customizing the help text for each descriptor instance is surprisingly hard. One solution requires dynamically building a wrapper class for each descriptor instance.



```
lontra:descriptors luciano$ python3 -i bulkfood_v5.py
>>> help(LineItem.weight)
```

```
Help on Quantity in module model_v5 object:

class Quantity(Validated)
 |  a number greater than zero
 |
 |  Method resol
 |      Quantity
 |      Validate
 |      abc.ABC
 |      AutoStor
 |      builtins
 |
 |  Methods defi
 |
 |  validate(sel
 |
 |  ------------
 |  Data and oth
 |
 |  __abstractme
 |
 |  ------------
 |  Methods inhe
:
```

```
lontra:descriptors luciano$ python3 -i bulkfood_v5.py
>>> help(LineItem.weight)

>>> help(LineItem)
```

```
Help on class LineItem in module __main__:

class LineItem(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, description, weight, price)
 |
 |  subtotal(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  description
 |      a string with at least one non-space character
 |
 |  price
 |      a number greater than zero
:
```

Figure 23-4. Screenshots of the Python console when issuing the commands `help(LineItem.weight)` and `help(LineItem)`.

Chapter Summary

The first example of this chapter was a continuation of the `LineItem` examples from Chapter 22. In Example 23-2, we replaced properties with descriptors. We saw that a descriptor is a class that provides instances that are deployed as attributes in the managed class. Discussing this mechanism required special terminology, introducing terms such as *managed instance* and *storage attribute*.

In "LineItem Take #4: Automatic Naming of Storage Attributes" on page 887, we removed the requirement that `Quantity` descriptors were declared with an explicit `storage_name`, which was redundant and error prone. The solution was to implement the `__set_name__` special method in `Quantity`, to save the name of the managed property as `self.storage_name`.

"LineItem Take #5: A New Descriptor Type" on page 889 showed how to subclass an abstract descriptor class to share code while building specialized descriptors with some common functionality.

We then looked at the different behaviors of descriptors providing or omitting the `__set__` method, making the crucial distinction between overriding and nonoverriding descriptors, a.k.a. data and nondata descriptors. Through detailed testing we uncovered when descriptors are in control and when they are shadowed, bypassed, or overwritten.

Following that, we studied a particular category of nonoverriding descriptors: methods. Console experiments revealed how a function attached to a class becomes a method when accessed through an instance, by leveraging the descriptor protocol.

To conclude the chapter, "Descriptor Usage Tips" on page 900 presented practical tips, and "Descriptor Docstring and Overriding Deletion" on page 902 provided a brief look at how to document descriptors.

As noted in "What's New in This Chapter" on page 880, several examples in this chapter became much simpler thanks to the `__set_name__` special method of the descriptor protocol, added in Python 3.6. That's language evolution!

Further Reading

Besides the obligatory reference to the "Data Model" chapter, Raymond Hettinger's "Descriptor HowTo Guide" is a valuable resource—part of the HowTo collection in the official Python documentation.

As usual with Python object model subjects, Martelli, Ravenscroft, and Holden's *Python in a Nutshell*, 3rd ed. (O'Reilly) is authoritative and objective. Martelli also has a presentation titled "Python's Object Model," which covers properties and descriptors in depth (see the slides and video).

Beware that any coverage of descriptors written or recorded before PEP 487 was adopted in 2016 is likely to contain examples that are needlessly complicated today, because `__set_name__` was not supported in Python versions prior to 3.6.

For more practical examples, *Python Cookbook*, 3rd ed., by David Beazley and Brian K. Jones (O'Reilly), has many recipes illustrating descriptors, of which I want to highlight "6.12. Reading Nested and Variable-Sized Binary Structures," "8.10. Using Lazily Computed Properties," "8.13. Implementing a Data Model or Type System," and "9.9. Defining Decorators As Classes." The last recipe of which addresses deep issues with the interaction of function decorators, descriptors, and methods, explaining how a function decorator implemented as a class with `__call__` also needs to implement `__get__` if it wants to work with decorating methods as well as functions.

PEP 487—Simpler customization of class creation introduced the `__set_name__` special method, and includes an example of a validating descriptor.

Soapbox

The Design of self

The requirement to explicitly declare `self` as a first argument in methods is a controversial design decision in Python. After 23 years using the language, I am used to it. I think that decision is an example of "worse is better": a design philosophy described by computer scientist Richard P. Gabriel in "The Rise of Worse is Better". The first priority of this philosophy is "simplicity," which Gabriel presents as:

> The design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.

Python's explicit `self` embodies that design philosophy. The implementation is simple—elegant even—at the expense of the user interface: a method signature like `def zfill(self, width):` doesn't visually match the invocation `label.zfill(8)`.

Modula-3 introduced that convention with the same identifier `self`. But there is a key difference: in Modula-3, interfaces are declared separately from their implementation, and in the interface declaration the `self` argument is omitted, so from the user's perspective, a method appears in an interface declaration with the same explicit parameters used to call it.

Over time, Python's error messages related to method arguments became clearer. For a user-defined method with one argument besides `self`, if the user invokes `obj.meth()`, Python 2.7 raised:

```
TypeError: meth() takes exactly 2 arguments (1 given)
```

In Python 3, the confusing argument count is not mentioned, and the missing argument is named:

```
TypeError: meth() missing 1 required positional argument: 'x'
```

Besides the use of `self` as an explicit argument, the requirement to qualify every access to instance attributes with `self` is also criticized. See, for example, A. M. Kuchling's famous "Python Warts" post (archived); Kuchling himself is not so bothered by the `self` qualifier, but he mentions it—probably echoing opinions from the comp.lang.python group. I personally don't mind typing the `self` qualifier: it's good to distinguish local variables from attributes. My issue is with the use of `self` in the `def` statement.

Anyone who is unhappy about the explicit `self` in Python can feel a lot better by considering the baffling semantics of the implicit `this` in JavaScript. Guido had some good reasons to make `self` work as it does, and he wrote about them in "Adding Support for User-Defined Classes", a post on his blog, *The History of Python*.