CHAPTER 20

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.[1]
>
> — Raymond Hettinger
> *Python core developer and guru*

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 SQL Alchemy 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 protocol consisting of the `__get__`, `__set__`, and `__delete__` methods. The `property` class implements the full descriptor protocol. As usual with 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. Besides properties, other Python features that leverage descriptors are methods and the `classmethod` and `staticmethod` decorators. Understanding descriptors is key to Python mastery. This is what this chapter is about.

Descriptor Example: Attribute Validation

As we saw in "Coding a Property Factory" on page 611, 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

1. Raymond Hettinger, Descriptor HowTo Guide.

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 it, in "Coding a Property Factory" on page 611, by refactoring the `quantity` property factory into a `Quantity` descriptor class.

LineItem Take #3: A Simple Descriptor

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 20-1.

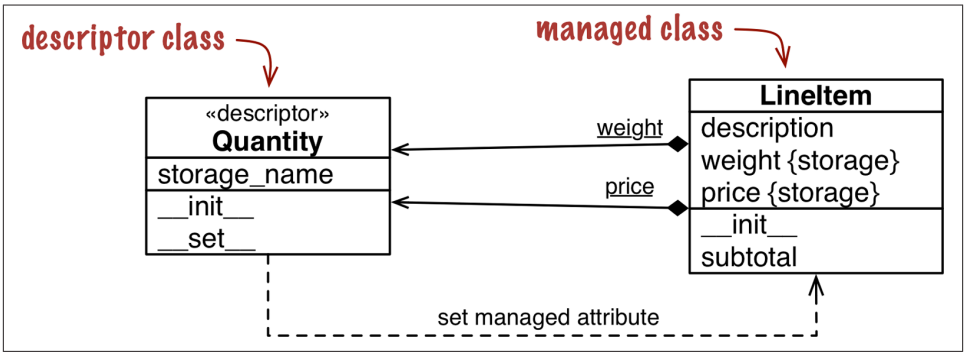

Figure 20-1. UML class diagram for LineItem using a descriptor class named Quantity. Underlined attributes in UML are class attributes. Note that weight and price are instances of Quantity attached to the LineItem class, but LineItem instances also have their own weight and price attributes where those values are stored.

Note that the word `weight` appears twice in Figure 20-1, 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`.

From now on, I will use the following definitions:

Descriptor class
: A class implementing the descriptor protocol. That's `Quantity` in Figure 20-1.

Managed class
: The class where the descriptor instances are declared as class attributes—`LineItem` in Figure 20-1.

Descriptor instance
: Each instance of a descriptor class, declared as a class attribute of the managed class. In Figure 20-1, 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 will be the managed instances (they are not shown in the class diagram).

Storage attribute
: An attribute of the managed instance that will hold the value of a managed attribute for that particular instance. In Figure 20-1, the `LineItem` instance attributes `weight` and `price` will be 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 will be 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 20-2.

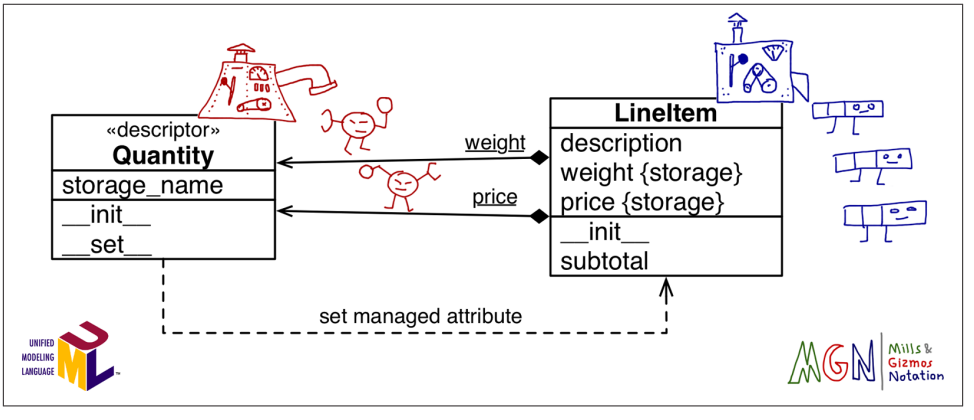

Figure 20-2. UML class diagram annotated with MGN (Mills & Gizmos Notation): classes are mills that produce gizmos—the instances. The Quantity mill produces two red gizmos, which are attached to the LineItem mill: weight and price. The LineItem mill produces blue gizmos that have their own weight and 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 20-3. 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. A gizmo is the same color as the mill that made it.

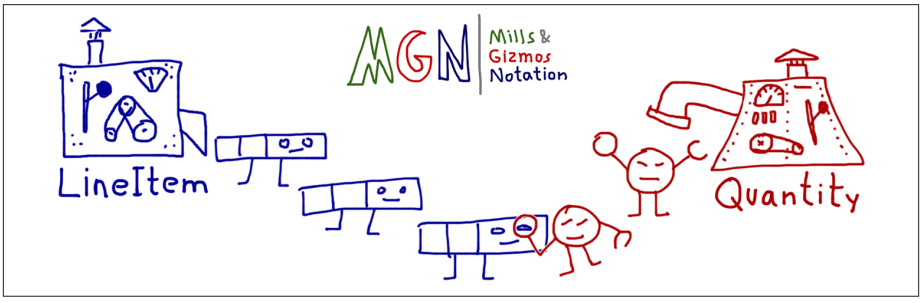

Figure 20-3. MGN sketch showing the LineItem class making three instances, and Quantity making two. One instance of Quantity is retrieving a value stored in a LineItem instance.

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 20-1 shows the `Quantity` descriptor class and a new `LineItem` class using two instances of `Quantity`.

Example 20-1. bulkfood_v3.py: quantity descriptors manage attributes in LineItem

```
class Quantity:    ❶

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

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.

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


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

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

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

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 attribute that will 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 Here, we must handle the managed instance `__dict__` directly; trying to use the `setattr` built-in would trigger the `__set__` method again, leading to infinite recursion.

❺ The first descriptor instance is bound to the `weight` attribute.

❻ The second descriptor instance is bound to the `price` attribute.

❼ The rest of the class body is as simple and clean as the original code in *bulkfood_v1.py* (Example 19-15).

In Example 20-1, each managed attribute has the same name as its storage attribute, and there is no special getter logic, so `Quantity` doesn't need a `__get__` method.

The code in Example 20-1 works as intended, preventing the sale of truffles for $0:[3]

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).

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

When coding a `__set__` method, you must 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: `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 20-1 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
```

The problem is that—as we saw in Chapter 8—the righthand side of an assignment is executed before the variable exists. The expression `Quantity()` is evaluated to create a descriptor instance, and at this time 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`).

As it stands, Example 20-1 requires naming each `Quantity` explicitly, which is not only inconvenient but dangerous: if a programmer copy and pasting code forgets to edit both

names and writes something like `price = Quantity('weight')`, the program will misbehave badly, clobbering the value of `weight` whenever the `price` is set.

A not-so-elegant but workable solution to the repeated name problem is presented next. Better solutions require either a class decorator or a metaclass, so I'll leave them for Chapter 21.

LineItem Take #4: Automatic Storage Attribute Names

To avoid retyping the attribute name in the descriptor declarations, we'll generate a unique string for the `storage_name` of each `Quantity` instance. Figure 20-4 shows the updated UML diagram for the `Quantity` and `LineItem` classes.

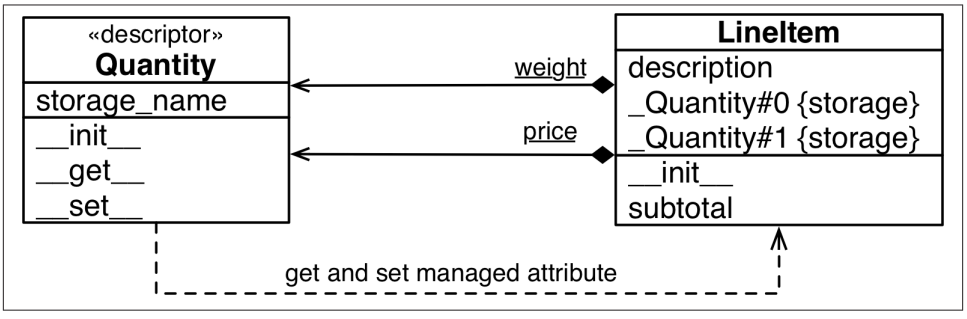

Figure 20-4. UML class diagram for Example 20-2. Now Quantity has both get and set methods, and LineItem instances have storage attributes with generated names: _Quantity#0 and _Quantity#1.

To generate the `storage_name`, we start with a `'_Quantity#'` prefix and concatenate an integer: the current value of a `Quantity.__counter` class attribute that we'll increment every time a new `Quantity` descriptor instance is attached to a class. Using the hash character in the prefix guarantees the `storage_name` will not clash with attributes created by the user using dot notation, because `nutmeg._Quantity#0` is not valid Python syntax. But we can always get and set attributes with such "invalid" identifiers using the `getattr` and `setattr` built-in functions, or by poking the instance `__dict__`. Example 20-2 shows the new implementation.

Example 20-2. bulkfood_v4.py: each Quantity descriptor gets a unique storage_name

```
class Quantity:
    __counter = 0    ❶

    def __init__(self):
        cls = self.__class__    ❷
        prefix = cls.__name__
        index = cls.__counter
```

```
        self.storage_name = '_{}#{}'.format(prefix, index)   ❸
        cls.__counter += 1   ❹

    def __get__(self, instance, owner):   ❺
        return getattr(instance, self.storage_name)   ❻

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)   ❼
        else:
            raise ValueError('value must be > 0')


class LineItem:
    weight = Quantity()   ❽
    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 `__counter` is a class attribute of `Quantity`, counting the number of `Quantity` instances.

❷ `cls` is a reference to the `Quantity` class.

❸ The `storage_name` for each descriptor instance is unique because it's built from the descriptor class name and the current `__counter` value (e.g., `_Quantity#0`).

❹ Increment `__counter`.

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

❻ Use the `getattr` built-in function to retrieve the value from the `instance`.

❼ Use the `setattr` built-in to store the value in the `instance`.

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

Here we can use the higher-level `getattr` and `setattr` built-ins to store the value—instead of resorting to `instance.__dict__`—because the managed attribute and the storage attribute have different names, so calling `getattr` on the storage attribute will not trigger the descriptor, avoiding the infinite recursion discussed in Example 20-1.

If you test *bulkfood_v4.py*, you can see that the `weight` and `price` descriptors work as expected, and the storage attributes can also be read directly, which is useful for debugging:

```
>>> from bulkfood_v4 import LineItem
>>> coconuts = LineItem('Brazilian coconut', 20, 17.95)
>>> coconuts.weight, coconuts.price
(20, 17.95)
>>> getattr(raisins, '_Quantity#0'), getattr(raisins, '_Quantity#1')
(20, 17.95)
```

If we wanted to follow the convention Python uses to do name mangling (e.g., `_LineItem__quantity0`) we'd have to know the name of the managed class (i.e., `LineItem`), but the body of a class definition runs before the class itself is built by the interpreter, so we don't have that information when each descriptor instance is created. However, in this case, there is no need to include the managed class name to avoid accidental overwriting in subclasses: the descriptor class `__counter` will be incremented every time a new descriptor is instantiated, guaranteeing that each storage name will be unique across all managed classes.

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 handy when the descriptor is used to get attributes from the class. 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. This explains the `Attribute` error in the next console session:

```
>>> from bulkfood_v4 import LineItem
>>> LineItem.weight
Traceback (most recent call last):
  ...
  File ".../descriptors/bulkfood_v4.py", line 54, in __get__
    return getattr(instance, self.storage_name)
AttributeError: 'NoneType' object has no attribute '_Quantity#0'
```

Raising `AttributeError` is an option when implementing `__get__`, but if you choose to do so, the message should be fixed to remove the confusing mention of `NoneType` and `_Quantity#0`, which are implementation details. A better message would be `"'LineItem' class has no such attribute"`. Ideally, the name of the missing attribute should be spelled out, but the descriptor doesn't know the name of the managed attribute in this example, so we can't do better at this point.

On the other hand, to support introspection and other metaprogramming tricks by the user, it's a good practice to make `__get__` return the descriptor instance when the man-

aged attribute is accessed through the class. Example 20-3 is a minor variation of Example 20-2, adding a bit of logic to `Quantity.__get__`.

Example 20-3. bulkfood_v4b.py (partial listing): when invoked through the managed class, get returns a reference to the descriptor itself

```
class Quantity:
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self    ❶
        else:
            return getattr(instance, self.storage_name)    ❷

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)
        else:
            raise ValueError('value must be > 0')
```

❶ If the call was not through an instance, return the descriptor itself.

❷ Otherwise, return the managed attribute value, as usual.

Trying out Example 20-3, this is what we see:

```
>>> from bulkfood_v4b import LineItem
>>> LineItem.price
<bulkfood_v4b.Quantity object at 0x100721be0>
>>> br_nuts = LineItem('Brazil nuts', 10, 34.95)
>>> br_nuts.price
34.95
```

Looking at Example 20-2, 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 framework.

With this in mind, Example 20-4 better represents the typical usage of a descriptor.

Example 20-4. 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, giving it a friendlier name.

❷ Put `model.Quantity` to use.

Django users will notice that Example 20-4 looks a lot like a model definition. It's no coincidence: Django model fields are descriptors.

As implemented so far, the `Quantity` descriptor works pretty well. Its only real drawback is the use of generated storage names like `_Quantity#0`, making debugging hard for the users. But automatically assigning storage names that resemble the managed attribute names requires a class decorator or a metaclass, topics we'll defer to Chapter 21.

Because descriptors are defined in 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.

Property Factory Versus Descriptor Class

It's not hard to reimplement the enhanced descriptor class of Example 20-2 by adding a few lines to the property factory shown in Example 19-24. The `__counter` variable presents a difficulty, but we can make it persist across invocations of the factory by defining it as an attribute of factory function object itself, as shown in Example 20-5.

Example 20-5. bulkfood_v4prop.py: same functionality as Example 20-2 with a property factory instead of a descriptor class

```
def quantity():    ❶
    try:
        quantity.counter += 1    ❷
```

```
    except AttributeError:
        quantity.counter = 0    ❸

    storage_name = '_{}:{}'.format('quantity', quantity.counter)    ❹

    def qty_getter(instance):    ❺
        return getattr(instance, storage_name)

    def qty_setter(instance, value):
        if value > 0:
            setattr(instance, storage_name, value)
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)
```

❶ No `storage_name` argument.

❷ We can't rely on class attributes to share the `counter` across invocations, so we define it as an attribute of the `quantity` function itself.

❸ If `quantity.counter` is undefined, set it to `0`.

❹ We also don't have instance attributes, so we create `storage_name` as a local variable and rely on closures to keep them alive for later use by `qty_getter` and `qty_setter`.

❺ The remaining code is identical to Example 19-24, except here we can use the `getattr` and `setattr` built-ins instead of fiddling with `instance.__dict__`.

So, which do you prefer? Example 20-2 or Example 20-5?

I prefer the descriptor class approach mainly for two reasons:

- A descriptor class can be extended by subclassing; reusing code from a factory function without copying and pasting is much harder.
- It's more straightforward to hold state in class and instance attributes than in function attributes and closures as we had to do in Example 20-5.

On the other hand, when I explain Example 20-5, I don't feel the urge to draw mills and gizmos. The property factory code does not depend on strange object relationships evidenced by descriptor methods having arguments named `self` and `instance`.

To summarize, the property factory pattern is simpler in some regards, but the descriptor class approach is more extensible. It's also more widely used.

LineItem Take #5: A New Descriptor Type

he 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.

Reflecting on the functionality of `Quantity`, we note it does two different things: it takes care of the storage attributes in the managed instances, and it validates the value used to set those attributes. This prompts a refactoring, producing two base classes:

`AutoStorage`
: Descriptor class that manages storage attributes automatically.

`Validated`
: `AutoStorage` abstract subclass 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. Figure 20-5 depicts the setup.

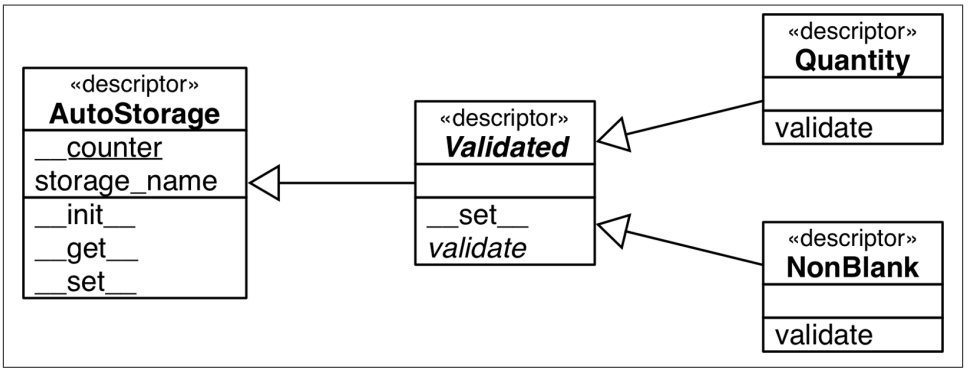

Figure 20-5. A hierarchy of descriptor classes. The AutoStorage base class manages the automatic storage of the attribute, Validated handles validation by delegating to an abstract validate method, Quantity and NonBlank are concrete subclasses of Validated.

The relationship between `Validated`, `Quantity`, and `NonBlank` is an application of the Template Method design pattern. In particular, the `Validated.__set__` is a clear example of what the Gang of Four describe as a template method:

> A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.[4]

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

In this case, the abstract operation is validation. Example 20-6 lists the implementation of the classes in Figure 20-5.

Example 20-6. model_v5.py: the refactored descriptor classes

```
import abc


class AutoStorage:   ❶
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)

    def __set__(self, instance, value):
        setattr(instance, self.storage_name, value)   ❷


class Validated(abc.ABC, AutoStorage):   ❸

    def __set__(self, instance, value):
        value = self.validate(instance, value)   ❹
        super().__set__(instance, value)   ❺

    @abc.abstractmethod
    def validate(self, instance, value):   ❻
        """return validated value or raise ValueError"""


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

    def validate(self, instance, value):
        if value <= 0:
            raise ValueError('value must be > 0')
        return value


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

    def validate(self, instance, value):
```

```
        value = value.strip()
        if len(value) == 0:
            raise ValueError('value cannot be empty or blank')
        return value    ❽
```

❶ `AutoStorage` provides most of the functionality of the former `Quantity` descriptor…

❷ …except validation.

❸ `Validated` is abstract but also inherits from `AutoStorage`.

❹ `__set__` delegates validation to a `validate` method…

❺ …then uses the returned `value` to invoke `__set__` on a superclass, which performs the actual storage.

❻ In this class, `validate` is an abstract method.

❼ `Quantity` and `NonBlank` inherit from `Validated`.

❽ 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, the `value` is returned stripped of leading and 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 20-7.

Example 20-7. 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. Such a descriptor is also called an overriding descriptor because its `__set__` method overrides (i.e., interrupts and overrules) the setting of an 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 defined. Observing the different behaviors requires a few classes, so we are going to use the code in Example 20-8 as our testbed for the following sections.

Every `__get__` and `__set__` method in Example 20-8 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 20-8. 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 '<class {}>'.format(obj.__name__)
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return '<{} object>'.format(cls_name(obj))

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
```

```
    print('-> {}.__{}__({})'.format(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('-> Managed.spam({})'.format(display(self)))
```

❶ A typical 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 Descriptor

A descriptor that implements the `__set__` method is called 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 20-2 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 20-8, experiments with an overriding descriptor can be seen in Example 20-9.

Example 20-9. Behavior of an overriding descriptor: obj.over is an instance of Overriding (Example 20-8)

```
>>> 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__

Usually, overriding descriptors implement both `__set__` and `__get__`, but it's also possible to implement only `__set__`, as we saw in Example 20-1. 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 20-10.

Example 20-10. Overriding descriptor without get: obj.over_no_get is an instance of OverridingNoGet (Example 20-8)

```
>>> 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

If a descriptor does not implement `__set__`, then it's 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 are implemented as nonoverriding descriptors. Example 20-11 shows the operation of a nonoverriding descriptor.

Example 20-11. Behavior of a nonoverriding descriptor: obj.non_over is an instance of NonOverriding (Example 20-8)

```
>>> 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.

Python contributors and authors use different terms when discussing these concepts. Overriding descriptors are also called data descriptors or enforced descriptors. Nonoverriding descriptors are also known as nondata descriptors or shadowable descriptors.

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 20-12 the descriptors are replaced by integers, which would effectively break any class that depended on the descriptors for proper operation.

Example 20-12. 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 20-12 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 21, 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 because all user-defined functions have a `__get__` method, therefore they operate as descriptors when attached to a class. Example 20-13 demonstrates reading the `spam` method from the `Managed` class introduced in Example 20-8.

Example 20-13. 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.

Because functions do not implement `__set__`, they are nonoverriding descriptors, as the last line of Example 20-13 shows.

The other key takeaway from Example 20-13 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 159).

For a deeper understanding of this mechanism, take a look at Example 20-14.

Example 20-14. 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 20-15.

Example 20-15. 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.

⓾ 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 actually creates overriding descriptors implementing both `__set__` and `__get__`, even if you do not define a setter method. 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.

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.[5]

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 20-1.

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

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. 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.

Nonspecial methods can be shadowed by instance attributes
: Because functions and methods only implement `__get__`, they do not handle attempts at setting instance attributes with the same name, so 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 15 years of Python coding. On the other hand, if you are doing a lot of dynamic 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 19-6 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 19-6—later replaced by `__new__` in Example 19-7. The `Record` class (Examples 19-9 and 19-11) and subclasses are also safe: they use only special methods, class methods, static methods, and properties. Properties are data descriptors, so cannot be overridden 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. See Figure 20-6 for the help displays for the `LineItem` class with the `Quantity` and `NonBlank` descriptors from Examples 20-6 and 20-7.

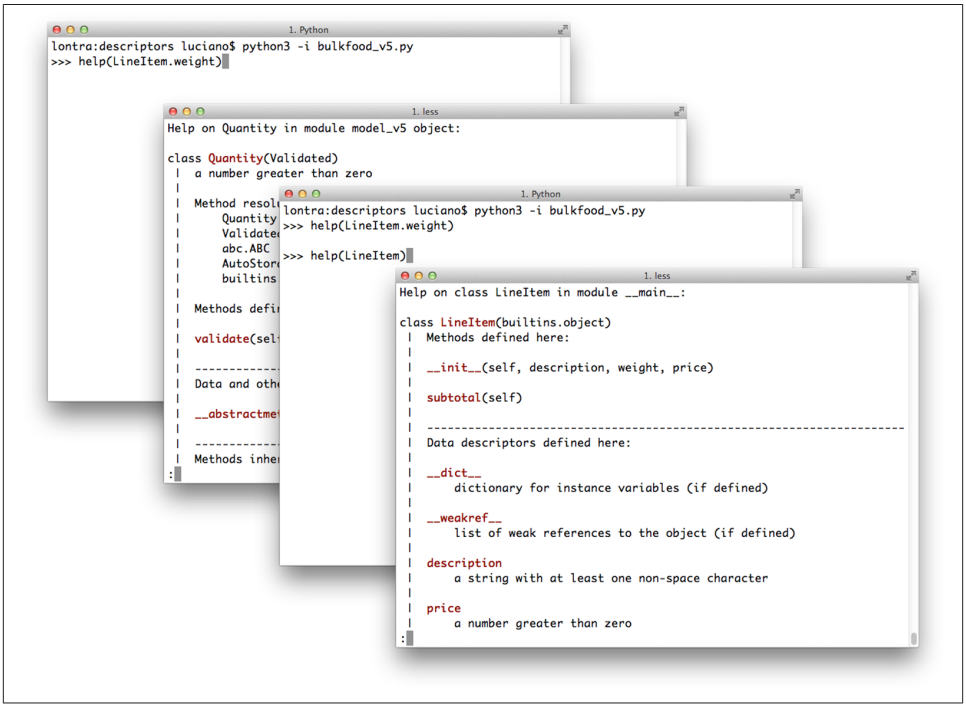

Figure 20-6. Screenshots of the Python console when issuing the commands help(LineItem.weight) and help(LineItem)

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`.[6]

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

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

descriptor class. Coding a silly descriptor class with `__delete__` is left as an exercise to the leisurely reader.

Chapter Summary

The first example of this chapter was a continuation of the `LineItem` examples from Chapter 19. In Example 20-1, 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 Storage Attribute Names" on page 631, we removed the requirement that `Quantity` descriptors were declared with an explicit `storage_name`, which was redundant and error-prone, because that name should always match the attribute name on the left of the assignment in the descriptor instantiation. The solution was to generate unique `storage_names` by combining the descriptor class name with a counter at the class level (e.g., `'_Quantity#1'`).

Next, we compared the code size, strengths, and weaknesses of a descriptor class with a property factory built on functional programming idioms. The latter works perfectly well and is simpler in some ways, but the former is more flexible and is the standard solution. A key advantage of the descriptor class was exploited in "LineItem Take #5: A New Descriptor Type" on page 637: subclassing to share code while building specialized descriptors with some common functionality.

We then looked at the different behavior of descriptors providing or omitting the `__set__` method, making the crucial distinction between overriding and nonoverriding 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 testing 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 648 provided a brief look at how descriptor deletion and documentation work.

Throughout this chapter, we faced a few issues that only class metaprogramming can solve, and we deferred those to Chapter 21.

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, Alex Martelli's *Python in a Nutshell*, 2E (O'Reilly) is authoritative and objective, even if somewhat dated: the key mechanisms discussed in this chapter were introduced in Python 2.2, long before the 2.5 version covered by that book. Martelli also has a presentation titled *Python's Object Model*, which covers properties and descriptors in depth (slides, video). Highly recommended.

For Python 3 coverage with practical examples, *Python Cookbook, 3E* 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 latter 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.

Soapbox

The Problem with self

"Worse is Better" is a design philosophy described by Richard P. Gabriel in "The Rise of Worse is Better". The first priority of this philosophy is "Simplicity," which Gabriel states 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.

I believe the requirement to explicitly declare `self` as a first argument in methods is an application of "Worse is Better" in Python. 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 `pobox.zfill(8)`.

Modula-3 introduced that convention—and the use of the `self` identifier—but there is a 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 exactly with the same number of explicit arguments it takes.

One improvement in this regard has been the error messages: for a user-defined method with one argument besides `self`, if the user invokes `obj.meth()`, Python 2.7 raises `TypeError: meth() takes exactly 2 arguments (1 given)`, but in Python 3.4 the message is less confusing, sidestepping the issue of the argument count and naming the missing argument: `meth() missing 1 required positional argument: 'x'`.

Besides the use of `self` as an explicit argument, the requirement to qualify all access to instance attributes with `self` is also criticized.[7] 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. But I got used to it.

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.

7. 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 `comp.lang.python`.

CHAPTER 21
Class Metaprogramming

> [Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).[1]
>
> — Tim Peters
> *Inventor of the timsort algorithm and prolific Python contributor*

Class metaprogramming is the art of creating or customizing classes at runtime. Classes are first-class objects in Python, so a function can be used to create a new class at any time, without using the `class` keyword. Class decorators are also functions, but capable of inspecting, changing, and even replacing the decorated class with another class. Finally, metaclasses are the most advanced tool for class metaprogramming: they let you create whole new categories of classes with special traits, such as the abstract base classes we've already seen.

Metaclasses are powerful, but hard to get right. Class decorators solve many of the same problems more simply. In fact, metaclasses are now so hard to justify in real code that my favorite motivating example lost much of its appeal with the introduction of class decorators in Python 2.6.

Also covered here is the distinction between import time and runtime: a crucial prerequisite for effective Python metaprogramming.

This is an exciting topic, and it's easy to get carried away. So I must start this chapter with the following admonition:

If you are not authoring a framework, you should not be writing metaclasses—unless you're doing it for fun or to practice the concepts.

1. Message to comp.lang.python, subject: "Acrimony in c.l.p.". This is another part of the same message from December 23, 2002, quoted in the Preface. The TimBot was inspired that day.

We'll get started by reviewing how to create a class at runtime.

A Class Factory

The standard library has a class factory that we've seen several times in this book: `collections.namedtuple`. It's a function that, given a class name and attribute names creates a subclass of `tuple` that allows retrieving items by name and provides a nice `__repr__` for debugging.

Sometimes I've felt the need for a similar factory for mutable objects. Suppose I'm writing a pet shop application and I want to process data for dogs as simple records. It's bad to have to write boilerplate like this:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Boring… the field names appear three times each. All that boilerplate doesn't even buy us a nice `repr`:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Taking a hint from `collections.namedtuple`, let's create a `record_factory` that creates simple classes like `Dog` on the fly. Example 21-1 shows how it should work.

Example 21-1. Testing record_factory, a simple class factory

```
>>> Dog = record_factory('Dog', 'name weight owner')   ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex   ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex   ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex)   ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32   ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
>>> Dog.__mro__   ❻
(<class 'factories.Dog'>, <class 'object'>)
```

❶ Factory signature is similar to that of `namedtuple`: class name, followed by attribute names in a single string, separated by spaces or commas.

❷ Nice `repr`.

❸ Instances are iterable, so they can be conveniently unpacked on assignment…

❹ …or when passing to functions like `format`.

❺ A record instance is mutable.

❻ The newly created class inherits from `object`—no relationship to our factory.

The code for `record_factory` is in Example 21-2.[2]

Example 21-2. record_factory.py: a simple class factory

```
def record_factory(cls_name, field_names):
    try:
        field_names = field_names.replace(',', ' ').split()    ❶
    except AttributeError:  # no .replace or .split
        pass  # assume it's already a sequence of identifiers
    field_names = tuple(field_names)    ❷

    def __init__(self, *args, **kwargs):    ❸
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self):    ❹
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self):    ❺
        values = ', '.join('{}={!r}'.format(*i) for i
                           in zip(self.__slots__, self))
        return '{}({})'.format(self.__class__.__name__, values)

    cls_attrs = dict(__slots__ = field_names,    ❻
                     __init__  = __init__,
                     __iter__  = __iter__,
                     __repr__  = __repr__)

    return type(cls_name, (object,), cls_attrs)    ❼
```

❶ Duck typing in practice: try to split `field_names` by commas or spaces; if that fails, assume it's already an iterable, with one name per item.

❷ Build a tuple of attribute names, this will be the `__slots__` attribute of the new class; this also sets the order of the fields for unpacking and `__repr__`.

❸ This function will become the `__init__` method in the new class. It accepts positional and/or keyword arguments.

2. Thanks to my friend J.S. Bueno for suggesting this solution.

❹ Implement an `__iter__`, so the class instances will be iterable; yield the field values in the order given by `__slots__`.

❺ Produce the nice `repr`, iterating over `__slots__` and `self`.

❻ Assemble dictionary of class attributes.

❼ Build and return the new class, calling the `type` constructor.

We usually think of `type` as a function, because we use it like one, e.g., `type(my_object)` to get the class of the object—same as `my_object.__class__`. However, `type` is a class. It behaves like a class that creates a new class when invoked with three arguments:

```
MyClass = type('MyClass', (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2})
```

The three arguments of `type` are named `name`, `bases`, and `dict`—the latter being a mapping of attribute names and attributes for the new class. The preceding code is functionally equivalent to this:

```
class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2
```

The novelty here is that the instances of `type` are classes, like `MyClass` here, or the `Dog` class in Example 21-1.

In summary, the last line of `record_factory` in Example 21-2 builds a class named by the value of `cls_name`, with `object` as its single immediate superclass and with class attributes named `__slots__`, `__init__`, `__iter__`, and `__repr__`, of which the last three are instance methods.

We could have named the `__slots__` class attribute anything else, but then we'd have to implement `__setattr__` to validate the names of attributes being assigned, because for our record-like classes we want the set of attributes to be always the same and in the same order. However, recall that the main feature of `__slots__` is saving memory when you are dealing with millions of instances, and using `__slots__` has some drawbacks, discussed in "Saving Space with the __slots__ Class Attribute" on page 264.

Invoking `type` with three arguments is a common way of creating a class dynamically. If you peek at the source code for `collections.namedtuple`, you'll see a different approach: there is `_class_template`, a source code template as a string, and the `namedtuple` function fills its blanks calling `_class_template.format(…)`. The resulting source code string is then evaluated with the `exec` built-in function.

It's good practice to avoid `exec` or `eval` for metaprogramming in Python. These functions pose serious security risks if they are fed strings (even fragments) from untrusted sources. Python offers sufficient introspection tools to make `exec` and `eval` unnecessary most of the time. However, the Python core developers chose to use `exec` when implementing `namedtuple`. The chosen approach makes the code generated for the class available in the `._source` attribute.

Instances of classes created by `record_factory` have a limitation: they are not serializable—that is, they can't be used with the `dump/load` functions from the `pickle` module. Solving this problem is beyond the scope of this example, which aims to show the `type` class in action in a simple use case. For the full solution, study the source code for `collections.namedtuple`; search for the word "pickling."

A Class Decorator for Customizing Descriptors

When we left the `LineItem` example in "LineItem Take #5: A New Descriptor Type" on page 637, the issue of descriptive storage names was still pending: the value of attributes such as `weight` was stored in an instance attribute named `_Quantity#0`, which made debugging a bit hard. You can retrieve the storage name from a descriptor in Example 20-7 with the following lines:

```
>>> LineItem.weight.storage_name
'_Quantity#0'
```

However, it would be better if the storage names actually included the name of the managed attribute, like this:

```
>>> LineItem.weight.storage_name
'_Quantity#weight'
```

Recall from "LineItem Take #4: Automatic Storage Attribute Names" on page 631 that we could not use descriptive storage names because when the descriptor is instantiated it has no way of knowing the name of the managed attribute (i.e., the class attribute to which the descriptor will be bound, such as `weight` in the preceding examples). But once the whole class is assembled and the descriptors are bound to the class attributes, we can inspect the class and set proper storage names to the descriptors. This could be done in the `__new__` method of the `LineItem` class, so that by the time the descriptors are used in the `__init__` method, the correct storage names are set. The problem of using `__new__` for that purpose is wasted effort: the logic of `__new__` will run every time a new `LineItem` instance is created, but the binding of the descriptor to the managed attribute will never change once the `LineItem` class itself is built. So we need to set the

storage names when the class is created. That can be done with a class decorator or a metaclass. We'll do it first in the easier way.

A class decorator is very similar to a function decorator: it's a function that gets a class object and returns the same class or a modified one.

In Example 21-3, the `LineItem` class will be evaluated by the interpreter and the resulting class object will be passed to the `model.entity` function. Python will bind the global name `LineItem` to whatever the `model.entity` function returns. In this example, `model.entity` returns the same `LineItem` class with the `storage_name` attribute of each descriptor instance changed.

Example 21-3. bulkfood_v6.py: LineItem using Quantity and NonBlank descriptors

```
import model_v6 as model

@model.entity  ❶
class LineItem:
    description = model.NonBlank()
    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 The only change in this class is the added decorator.

Example 21-4 shows the implementation of the decorator. Only the new code at the bottom of *model_v6.py* is listed here; the rest of the module is identical to *model_v5.py* (Example 20-6).

Example 21-4. model_v6.py: a class decorator

```
def entity(cls):  ❶
    for key, attr in cls.__dict__.items():  ❷
        if isinstance(attr, Validated):  ❸
            type_name = type(attr).__name__
            attr.storage_name = '_{}#{}'.format(type_name, key)  ❹
    return cls  ❺
```

❶ Decorator gets class as argument.

❷ Iterate over `dict` holding the class attributes.

❸ If the attribute is one of our `Validated` descriptors…

❹ …set the `storage_name` to use the descriptor class name and the managed attribute name (e.g., `_NonBlank#description`).

❺ Return the modified class.

The doctests in *bulkfood_v6.py* prove that the changes are successful. For example, Example 21-5 shows the names of the storage attributes in a `LineItem` instance.

Example 21-5. bulkfood_v6.py: doctests for new storage_name descriptor attributes

```
>>> raisins = LineItem('Golden raisins', 10, 6.95)
>>> dir(raisins)[:3]
['_NonBlank#description', '_Quantity#price', '_Quantity#weight']
>>> LineItem.description.storage_name
'_NonBlank#description'
>>> raisins.description
'Golden raisins'
>>> getattr(raisins, '_NonBlank#description')
'Golden raisins'
```

That's not too complicated. Class decorators are a simpler way of doing something that previously required a metaclass: customizing a class the moment it's created.

A significant drawback of class decorators is that they act only on the class where they are directly applied. This means subclasses of the decorated class may or may not inherit the changes made by the decorator, depending on what those changes are. We'll explore the problem and see how it's solved in the following sections.

What Happens When: Import Time Versus Runtime

For successful metaprogramming, you must be aware of when the Python interpreter evaluates each block of code. Python programmers talk about "import time" versus "runtime" but the terms are not strictly defined and there is a gray area between them. At import time, the interpreter parses the source code of a *.py* module in one pass from top to bottom, and generates the bytecode to be executed. That's when syntax errors may occur. If there is an up-to-date *.pyc* file available in the local `__pycache__`, those steps are skipped because the bytecode is ready to run.

Although compiling is definitely an import-time activity, other things may happen at that time, because almost every statement in Python is executable in the sense that they potentially run user code and change the state of the user program. In particular, the `import` statement is not merely a declaration[3] but it actually runs all the top-level code of the imported module when it's imported for the first time in the process—further imports of the same module will use a cache, and only name binding occurs then. That

3. Contrast with the `import` statement in Java, which is just a declaration to let the compiler know that certain packages are required.

top-level code may do anything, including actions typical of "runtime", such as connecting to a database.[4] That's why the border between "import time" and "runtime" is fuzzy: the `import` statement can trigger all sorts of "runtime" behavior.

In the previous paragraph, I wrote that importing "runs all the top-level code," but "top-level code" requires some elaboration. The interpreter executes a `def` statement on the top level of a module when the module is imported, but what does that achieve? The interpreter compiles the function body (if it's the first time that module is imported), and binds the function object to its global name, but it does not execute the body of the function, obviously. In the usual case, this means that the interpreter defines top-level functions at import time, but executes their bodies only when—and if—the functions are invoked at runtime.

For classes, the story is different: at import time, the interpreter executes the body of every class, even the body of classes nested in other classes. Execution of a class body means that the attributes and methods of the class are defined, and then the class object itself is built. In this sense, the body of classes is "top-level code": it runs at import time.

This is all rather subtle and abstract, so here is an exercise to help you see what happens when.

The Evaluation Time Exercises

Consider a script, *evaltime.py*, which imports a module *evalsupport.py*. Both modules have several `print` calls to output markers in the format `<[N]>`, where `N` is a number. The goal of this pair of exercises is to determine when each of theses calls will be made.

Students have reported these exercises are helpful to better appreciate how Python evaluates the source code. Do take the time to solve them with paper and pencil before looking at "Solution for scenario #1" on page 664.

The listings are Examples 21-6 and 21-7. Grab paper and pencil and—without running the code—write down the markers in the order they will appear in the output, in two scenarios:

Scenario #1

The module *evaltime.py* is imported interactively in the Python console:

```
>>> import evaltime
```

4. I'm not saying starting a database connection just because a module is imported is a good idea, only pointing out it can be done.

Scenario #2

The module *evaltime.py* is run from the command shell:

```
$ python3 evaltime.py
```

Example 21-6. evaltime.py: write down the numbered <[N]> markers in the order they will appear in the output

```
from evalsupport import deco_alpha

print('<[1]> evaltime module start')


class ClassOne():
    print('<[2]> ClassOne body')

    def __init__(self):
        print('<[3]> ClassOne.__init__')

    def __del__(self):
        print('<[4]> ClassOne.__del__')

    def method_x(self):
        print('<[5]> ClassOne.method_x')

    class ClassTwo(object):
        print('<[6]> ClassTwo body')


@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')

    def method_y(self):
        print('<[8]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[9]> ClassFour body')

    def method_y(self):
        print('<[10]> ClassFour.method_y')


if __name__ == '__main__':
    print('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print('<[12]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
```

```
    four.method_y()


print('<[14]> evaltime module end')
```

Example 21-7. evalsupport.py: module imported by evaltime.py

```
print('<[100]> evalsupport module start')

def deco_alpha(cls):
    print('<[200]> deco_alpha')

    def inner_1(self):
        print('<[300]> deco_alpha:inner_1')

    cls.method_y = inner_1
    return cls

# BEGIN META_ALEPH
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
# END META_ALEPH

print('<[700]> evalsupport module end')
```

Solution for scenario #1

Example 21-8 is the output of importing the *evaltime.py* module in the Python console.

Example 21-8. Scenario #1: importing evaltime in the Python console

```
>>> import evaltime
<[100]> evalsupport module start  ❶
<[400]> MetaAleph body  ❷
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body  ❸
<[6]> ClassTwo body  ❹
<[7]> ClassThree body
<[200]> deco_alpha  ❺
<[9]> ClassFour body
<[14]> evaltime module end  ❻
```

❶ All top-level code in `evalsupport` runs when the module is imported; the `deco_alpha` function is compiled, but its body does not execute.

❷ The body of the `MetaAleph` function does run.

❸ The body of every class is executed…

❹ …including nested classes.

❺ The decorator function runs after the body of the decorated `ClassThree` is evaluated.

❻ In this scenario, the `evaltime` is imported, so the `if __name__ == '__main__':` block never runs.

Notes about scenario #1:

1. This scenario is triggered by a simple `import evaltime` statement.
2. The interpreter executes every class body of the imported module and its dependency, `evalsupport`.
3. It makes sense that the interpreter evaluates the body of a decorated class before it invokes the decorator function that is attached on top of it: the decorator must get a class object to process, so the class object must be built first.
4. The only user-defined function or method that runs in this scenario is the `deco_alpha` decorator.

Now let's see what happens in scenario #2.

Solution for scenario #2

Example 21-9 is the output of running `python evaltime.py`.

Example 21-9. Scenario #2: running evaltime.py from the shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body
<[6]> ClassTwo body
<[7]> ClassThree body
<[200]> deco_alpha
<[9]> ClassFour body  ❶
<[11]> ClassOne tests .............................
<[3]> ClassOne.__init__  ❷
<[5]> ClassOne.method_x
<[12]> ClassThree tests ............................
<[300]> deco_alpha:inner_1  ❸
```

```
<[13]> ClassFour tests ..........................
<[10]> ClassFour.method_y
<[14]> evaltime module end
<[4]> ClassOne.__del__  ❹
```

❶ Same output as Example 21-8 so far.

❷ Standard behavior of a class.

❸ `ClassThree.method_y` was changed by the `deco_alpha` decorator, so the call `three.method_y()` runs the body of the `inner_1` function.

❹ The `ClassOne` instance bound to `one` global variable is garbage-collected only when the program ends.

The main point of scenario #2 is to show that the effects of a class decorator may not affect subclasses. In Example 21-6, `ClassFour` is defined as a subclass of `ClassThree`. The `@deco_alpha` decorator is applied to `ClassThree`, replacing its `method_y`, but that does not affect `ClassFour` at all. Of course, if the `ClassFour.method_y` did invoke the `ClassThree.method_y` with `super(…)`, we would see the effect of the decorator, as the `inner_1` function executed.

In contrast, the next section will show that metaclasses are more effective when we want to customize a whole class hierarchy, and not one class at a time.

Metaclasses 101

A metaclass is a class factory, except that instead of a function, like `record_factory` from Example 21-2, a metaclass is written as a class. Figure 21-1 depicts a metaclass using the Mills & Gizmos Notation: a mill producing another mill.

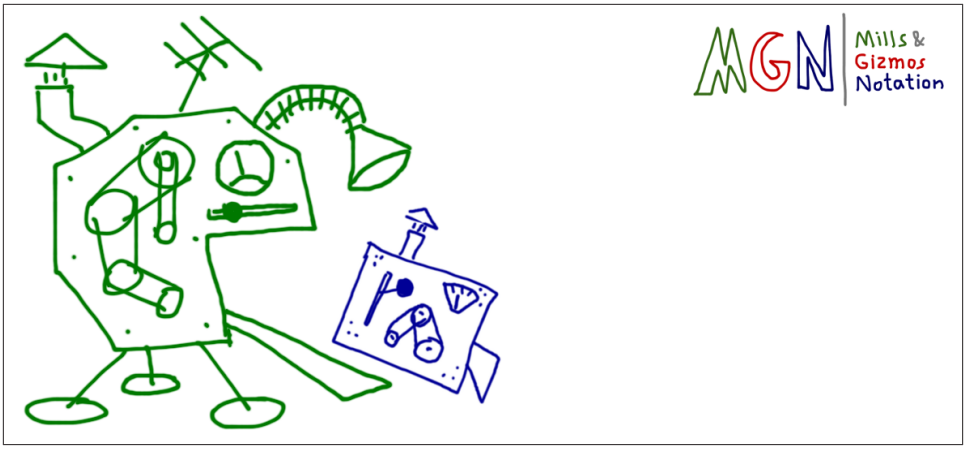

Figure 21-1. A metaclass is a class that builds classes

Consider the Python object model: classes are objects, therefore each class must be an instance of some other class. By default, Python classes are instances of `type`. In other words, `type` is the metaclass for most built-in and user-defined classes:

```
>>> 'spam'.__class__
<class 'str'>
>>> str.__class__
<class 'type'>
>>> from bulkfood_v6 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

To avoid infinite regress, `type` is an instance of itself, as the last line shows.

Note that I am not saying that `str` or `LineItem` inherit from `type`. What I am saying is that `str` and `LineItem` are instances of `type`. They all are subclasses of `object`. Figure 21-2 may help you confront this strange reality.

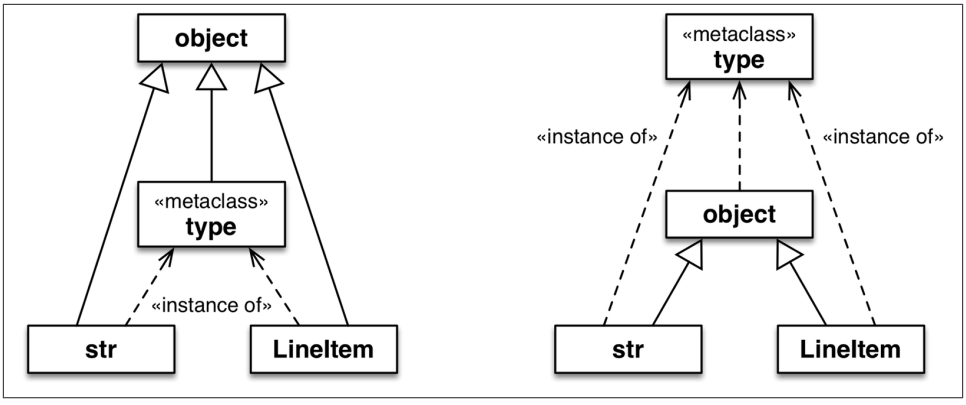

Figure 21-2. Both diagrams are true. The left one emphasizes that str, type, and LineItem are subclasses of object. The right one makes it clear that str, object, and LineItem are instances of type, because they are all classes.

The classes `object` and `type` have a unique relationship: `object` is an instance of `type`, and `type` is a subclass of `object`. This relationship is "magic": it cannot be expressed in Python because either class would have to exist before the other could be defined. The fact that `type` is an instance of itself is also magical.

Besides `type`, a few other metaclasses exist in the standard library, such as `ABCMeta` and `Enum`. The next snippet shows that the class of `collections.Iterable` is `abc.ABCMeta`. The class `Iterable` is abstract, but `ABCMeta` is not—after all, `Iterable` is an instance of `ABCMeta`:

```
>>> import collections
>>> collections.Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> abc.ABCMeta.__class__
<class 'type'>
>>> abc.ABCMeta.__mro__
(<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>)
```

Ultimately, the class of `ABCMeta` is also `type`. Every class is an instance of `type`, directly or indirectly, but only metaclasses are also subclasses of `type`. That's the most important relationship to understand metaclasses: a metaclass, such as `ABCMeta`, inherits from `type` the power to construct classes. Figure 21-3 illustrates this crucial relationship.

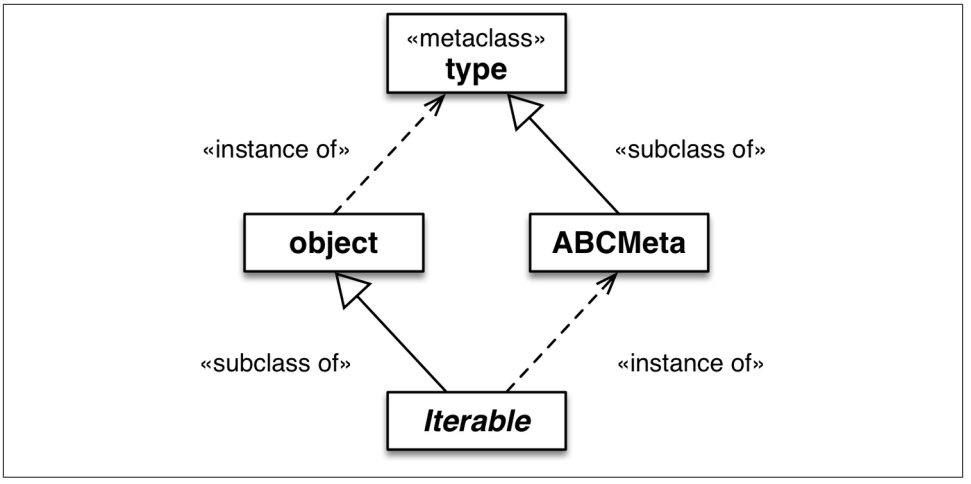

Figure 21-3. Iterable is a subclass of object and an instance of ABCMeta. Both object and ABCMeta are instances of type, but the key relationship here is that ABCMeta is also a subclass of type, because ABCMeta is a metaclass. In this diagram, Iterable is the only abstract class.

The important takeaway here is that all classes are instances of `type`, but metaclasses are also subclasses of `type`, so they act as class factories. In particular, a metaclass can customize its instances by implementing `__init__`. A metaclass `__init__` method can do everything a class decorator can do, but its effects are more profound, as the next exercise demonstrates.

The Metaclass Evaluation Time Exercise

This is a variation of "The Evaluation Time Exercises" on page 662. The *evalsupport.py* module is the same as Example 21-7, but the main script is now *evaltime_meta.py*, listed in Example 21-10.

Example 21-10. evaltime_meta.py: ClassFive is an instance of the MetaAleph metaclass

```
from evalsupport import deco_alpha
from evalsupport import MetaAleph

print('<[1]> evaltime_meta module start')


@deco_alpha
class ClassThree():
    print('<[2]> ClassThree body')

    def method_y(self):
        print('<[3]> ClassThree.method_y')
```

```
class ClassFour(ClassThree):
    print('<[4]> ClassFour body')

    def method_y(self):
        print('<[5]> ClassFour.method_y')


class ClassFive(metaclass=MetaAleph):
    print('<[6]> ClassFive body')

    def __init__(self):
        print('<[7]> ClassFive.__init__')

    def method_z(self):
        print('<[8]> ClassFive.method_y')


class ClassSix(ClassFive):
    print('<[9]> ClassSix body')

    def method_z(self):
        print('<[10]> ClassSix.method_y')


if __name__ == '__main__':
    print('<[11]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[12]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
    print('<[13]> ClassFive tests', 30 * '.')
    five = ClassFive()
    five.method_z()
    print('<[14]> ClassSix tests', 30 * '.')
    six = ClassSix()
    six.method_z()

print('<[15]> evaltime_meta module end')
```

Again, grab pencil and paper and write down the numbered `<[N]>` markers in the order they will appear in the output, considering these two scenarios:

Scenario #3
: The module *evaltime_meta.py* is imported interactively in the Python console.

Scenario #4
: The module *evaltime_meta.py* is run from the command shell.

Solutions and analysis are next.

Solution for scenario #3

Example 21-11 shows the output of importing *evaltime_meta.py* in the Python console.

Example 21-11. Scenario #3: importing evaltime_meta in the Python console

```
>>> import evaltime_meta
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__  ❶
<[9]> ClassSix body
<[500]> MetaAleph.__init__  ❷
<[15]> evaltime_meta module end
```

❶ The key difference from scenario #1 is that the `MetaAleph.__init__` method is invoked to initialize the just-created `ClassFive`.

❷ And `MetaAleph.__init__` also initializes `ClassSix`, which is a subclass of `ClassFive`.

The Python interpreter evaluates the body of `ClassFive` but then, instead of calling `type` to build the actual class body, it calls `MetaAleph`. Looking at the definition of `MetaAleph` in Example 21-12, you'll see that the `__init__` method gets four arguments:

`self`
: That's the class object being initialized (e.g., `ClassFive`)

`name`, `bases`, `dic`
: The same arguments passed to `type` to build a class

Example 21-12. evalsupport.py: definition of the metaclass MetaAleph from Example 21-7

```
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
```

When coding a metaclass, it's conventional to replace `self` with `cls`. For example, in the `__init__` method of the metaclass, using `cls` as the name of the first argument makes it clear that the instance under construction is a class.

The body of `__init__` defines an `inner_2` function, then binds it to `cls.method_z`. The name `cls` in the signature of `MetaAleph.__init__` refers to the class being created (e.g., `ClassFive`). On the other hand, the name `self` in the signature of `inner_2` will eventually refer to an instance of the class we are creating (e.g., an instance of `ClassFive`).

Solution for scenario #4

Example 21-13 shows the output of running `python evaltime.py` from the command line.

Example 21-13. Scenario #4: running evaltime_meta.py from the shell

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__
<[9]> ClassSix body
<[500]> MetaAleph.__init__
<[11]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1  ❶
<[12]> ClassFour tests ..............................
<[5]> ClassFour.method_y  ❷
<[13]> ClassFive tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2  ❸
<[14]> ClassSix tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2  ❹
<[15]> evaltime_meta module end
```

❶ When the decorator is applied to `ClassThree`, its `method_y` is replaced by the `inner_1` method…

❷ But this has no effect on the undecorated `ClassFour`, even though `ClassFour` is a subclass of `ClassThree`.

❸ The `__init__` method of `MetaAleph` replaces `ClassFive.method_z` with its `inner_2` function.

❹ The same happens with the `ClassFive` subclass, `ClassSix`: its `method_z` is replaced by `inner_2`.

Note that `ClassSix` makes no direct reference to `MetaAleph`, but it is affected by it because it's a subclass of `ClassFive` and therefore it is also an instance of `MetaAleph`, so it's initialized by `MetaAleph.__init__`.

Further class customization can be done by implementing `__new__` in a metaclass. But more often than not, implementing `__init__` is enough.

We can now put all this theory in practice by creating a metaclass to provide a definitive solution to the descriptors with automatic storage attribute names.

A Metaclass for Customizing Descriptors

Back to the `LineItem` examples. It would be nice if the user did not have to be aware of decorators or metaclasses at all, and could just inherit from a class provided by our library, like in Example 21-14.

Example 21-14. bulkfood_v7.py: inheriting from model.Entity can work, if a metaclass is behind the scenes

```
import model_v7 as model

class LineItem(model.Entity):    ❶
    description = model.NonBlank()
    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 `LineItem` is a subclass of `model.Entity`.

Example 21-14 looks pretty harmless. No strange syntax to be seen at all. However, it only works because *model_v7.py* defines a metaclass, and `model.Entity` is an instance of that metaclass. Example 21-15 shows the implementation of the `Entity` class in the *model_v7.py* module.

Example 21-15. model_v7.py: the EntityMeta metaclass and one instance of it, Entity

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)   ❶
        for key, attr in attr_dict.items():   ❷
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)

class Entity(metaclass=EntityMeta):   ❸
    """Business entity with validated fields"""
```

❶ Call `__init__` on the superclass (`type` in this case).

❷ Same logic as the `@entity` decorator in Example 21-4.

❸ This class exists for convenience only: the user of this module can just subclass `Entity` and not worry about `EntityMeta`—or even be aware of its existence.

The code in Example 21-14 passes the tests in Example 21-3. The support module, *model_v7.py*, is harder to understand than *model_v6.py*, but the user-level code is simpler: just inherit from `model_v7.entity` and you get custom storage names for your `Validated` fields.

Figure 21-4 is a simplified depiction of what we just implemented. There is a lot going on, but the complexity is hidden inside the `model_v7` module. From the user perspective, `LineItem` is simply a subclass of `Entity`, as coded in Example 21-14. This is the power of abstraction.

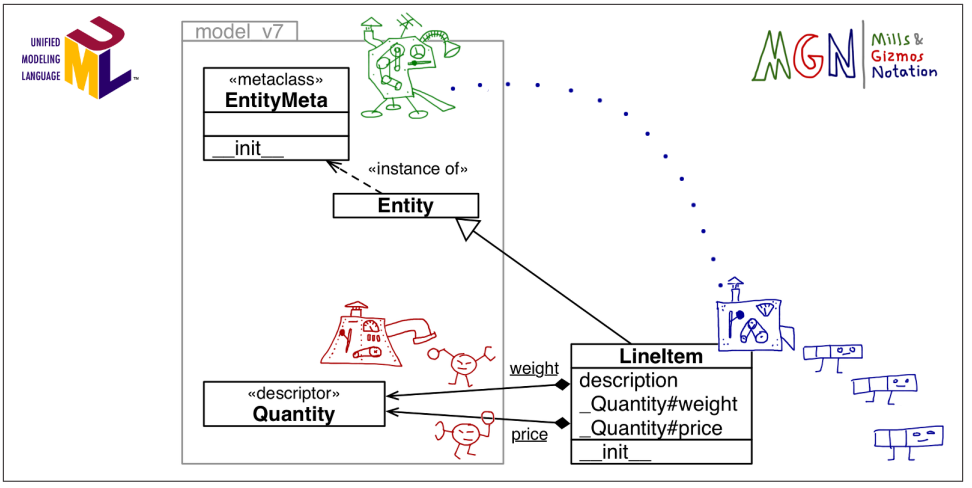

Figure 21-4. UML class diagram annotated with MGN (Mills & Gizmos Notation): the EntityMeta meta-mill builds the LineItem mill. Configuration of the descriptors (e.g., weight and price) is done by EntityMeta.__init__. Note the package boundary of model_v7.

Except for the syntax for linking a class to the metaclass,[5] everything written so far about metaclasses applies to versions of Python as early as 2.2, when Python types underwent a major overhaul. The next section covers a feature that is only available in Python 3.

The Metaclass ___prepare___ Special Method

In some applications it's interesting to be able to know the order in which the attributes of a class are defined. For example, a library to read/write CSV files driven by user-defined classes may want to map the order of the fields declared in the class to the order of the columns in the CSV file.

As we've seen, both the `type` constructor and the `__new__` and `__init__` methods of metaclasses receive the body of the class evaluated as a mapping of names to attributes. However, by default, that mapping is a `dict`, which means the order of the attributes as they appear in the class body is lost by the time our metaclass or class decorator can look at them.

The solution to this problem is the `__prepare__` special method, introduced in Python 3. This special method is relevant only in metaclasses, and it must be a class method (i.e., defined with the `@classmethod` decorator). The `__prepare__` method is invoked

5. Recall from "ABC Syntax Details" on page 328 that in Python 2.7 the `__metaclass__` class attribute is used, and the `metaclass=` keyword argument is not supported in the class declaration.

by the interpreter before the `__new__` method in the metaclass to create the mapping that will be filled with the attributes from the class body. Besides the metaclass as first argument, `__prepare__` gets the name of the class to be constructed and its tuple of base classes, and it must return a mapping, which will be received as the last argument by `__new__` and then `__init__` when the metaclass builds a new class.

It sounds complicated in theory, but in practice, every time I've seen `__prepare__` being used it was very simple. Take a look at Example 21-16.

Example 21-16. model_v8.py: the EntityMeta metaclass uses prepare, and Entity now has a field_names class method

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    @classmethod
    def __prepare__(cls, name, bases):
        return collections.OrderedDict()  ❶

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []  ❷
        for key, attr in attr_dict.items():  ❸
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)
                cls._field_names.append(key)  ❹


class Entity(metaclass=EntityMeta):
    """Business entity with validated fields"""

    @classmethod
    def field_names(cls):  ❺
        for name in cls._field_names:
            yield name
```

❶ Return an empty `OrderedDict` instance, where the class attributes will be stored.

❷ Create a `_field_names` attribute in the class under construction.

❸ This line is unchanged from the previous version, but `attr_dict` here is the `OrderedDict` obtained by the interpreter when it called `__prepare__` before calling `__init__`. Therefore, this `for` loop will go over the attributes in the order they were added.

❹ Add the name of each `Validated` field found to `_field_names`.

❺ The `field_names` class method simply yields the names of the fields in the order they were added.

With the simple additions made in Example 21-16, we are now able to iterate over the `Validated` fields of any `Entity` subclass using the `field_names` class method. Example 21-17 demonstrates this new feature.

Example 21-17. bulkfood_v8.py: doctest showing the use of field_names—no changes are needed in the LineItem class; field_names is inherited from model.Entity

```
    >>> for name in LineItem.field_names():
    ...     print(name)
    ...
    description
    weight
    price
```

This wraps up our coverage of metaclasses. In the real world, metaclasses are used in frameworks and libraries that help programmers perform, among other tasks:

- Attribute validation
- Applying decorators to many methods at once
- Object serialization or data conversion
- Object-relational mapping
- Object-based persistency
- Dynamic translation of class structures from other languages

We'll now have a brief overview of methods defined in the Python data model for all classes.

Classes as Objects

Every class has a number of attributes defined in the Python data model, documented in "4.13. Special Attributes" of the "Built-in Types" chapter in the *Library Reference*. Three of those attributes we've seen several times in the book already: `__mro__`, `__class__`, and `__name__`. Other class attributes are:

`cls.__bases__`
: The tuple of base classes of the class.

`cls.__qualname__`
: A new attribute in Python 3.3 holding the qualified name of a class or function, which is a dotted path from the global scope of the module to the class definition. For example, in Example 21-6, the `__qualname__` of the inner class `ClassTwo` is the string `'ClassOne.ClassTwo'`, while its `__name__` is just `'ClassTwo'`. The specification for this attribute is PEP-3155 — Qualified name for classes and functions.

`cls.__subclasses__()`
: This method returns a list of the immediate subclasses of the class. The implementation uses weak references to avoid circular references between the superclass and its subclasses—which hold a strong reference to the superclasses in their `__bases__` attribute. The method returns the list of subclasses that currently exist in memory.

`cls.mro()`
: The interpreter calls this method when building a class to obtain the tuple of superclasses that is stored in the `__mro__` attribute of the class. A metaclass can override this method to customize the method resolution order of the class under construction.

None of the attributes mentioned in this section are listed by the `dir(…)` function.

With this, our study of class metaprogramming ends. This is a vast topic and I only scratched the surface. That's why we have "Further Reading" sections in this book.

Chapter Summary

Class metaprogramming is about creating or customizing classes dynamically. Classes in Python are first-class objects, so we started the chapter by showing how a class can be created by a function invoking the `type` built-in metaclass.

In the next section, we went back to the `LineItem` class with descriptors from Chapter 20 to solve a lingering issue: how to generate names for the storage attributes that reflected the names of the managed attributes (e.g., `_Quantity#price` instead of `_Quantity#1`). The solution was to use a class decorator, essentially a function that gets a just-built class and has the opportunity to inspect it, change it, and even replace it with a different class.

We then moved to a discussion of when different parts of the source code of a module actually run. We saw that there is some overlap between the so-called "import time" and "runtime," but clearly a lot of code runs triggered by the `import` statement. Understanding what runs when is crucial, and there are some subtle rules, so we used the evaluation-time exercises to cover this topic.

The following subject was an introduction to metaclasses. We saw that all classes are instances of `type`, directly or indirectly, so that is the "root metaclass" of the language. A variation of the evaluation-time exercise was designed to show that a metaclass can

customize a hierarchy of classes—in contrast with a class decorator, which affects a single class and may have no impact on its descendants.

The first practical application of a metaclass was to solve the issue of the storage attribute names in `LineItem`. The resulting code is a bit trickier than the class decorator solution, but it can be encapsulated in a module so that the user merely subclasses an apparently plain class (`model.Entity`) without being aware that it is an instance of a custom metaclass (`model.EntityMeta`). The end result is reminiscent of the ORM APIs in Django and SQLAlchemy, which use metaclasses in their implementations but don't require the user to know anything about them.

The second metaclass we implemented added a small feature to `model.EntityMeta`: a `__prepare__` method to provide an `OrderedDict` to serve as the mapping from names to attributes. This preserves the order in which those attributes are bound in the body of the class under construction, so that metaclass methods like `__new__` and `__init__` can use that information. In the example, we implemented a `_field_names` class attribute, which made possible an `Entity.field_names()` so users could retrieve the `Validated` descriptors in the same order they appear in the source code.

The last section was a brief overview of attributes and methods available in all Python classes.

Metaclasses are challenging, exciting, and—sometimes—abused by programmers trying to be too clever. To wrap up, let's recall Alex Martelli's final advice from his essay "Waterfowl and ABCs" on page 314:

> And, *don't* define custom ABCs (or metaclasses) in production code… if you feel the urge to do so, I'd bet it's likely to be a case of "all problems look like a nail"-syndrome for somebody who just got a shiny new hammer—you (and future maintainers of your code) will be much happier sticking with straightforward and simple code, eschewing such depths.
>
> — Alex Martelli

Wise words from a man who is not only a master of Python metaprogramming but also an accomplished software engineer working on some of the largest mission-critical Python deployments in the world.

Further Reading

The essential references for this chapter in the Python documentation are "3.3.3. Customizing class creation" in the "Data Model" chapter of The Python Language Reference, the `type` class documentation in the "Built-in Functions" page, and "4.13. Special Attributes" of the "Built-in Types" chapter in the *Library Reference*. Also, in the *Library Reference*, the `types` module documentation covers two functions that are new in

Python 3.3 and are designed to help with class metaprogramming: `types.new_class(…)` and `types.prepare_class(…)`.

Class decorators were formalized in PEP 3129 - Class Decorators, written by Collin Winter, with the reference implementation authored by Jack Diederich. The PyCon 2009 talk "Class Decorators: Radically Simple" (video), also by Jack Diederich, is a quick introduction to the feature.

Python in a Nutshell, 2E by Alex Martelli features outstanding coverage of metaclasses, including a `metaMetaBunch` metaclass that aims to solve the same problem as our simple `record_factory` from Example 21-2 but is much more sophisticated. Martelli does not address class decorators because the feature appeared later than his book. Beazley and Jones provide excellent examples of class decorators and metaclasses in their *Python Cookbook, 3E* (O'Reilly). Michael Foord wrote an intriguing post titled "Meta-classes Made Easy: Eliminating self with Metaclasses". The subtitle says it all.

For metaclasses, the main references are PEP 3115 — Metaclasses in Python 3000, in which the `__prepare__` special method was introduced and Unifying types and classes in Python 2.2, authored by Guido van Rossum. The text applies to Python 3 as well, and it covers what were then called the "new-style" class semantics, including descriptors and metaclasses. It's a must-read. One of the references cited by Guido is *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*, by Ira R. Forman and Scott H. Danforth (Addison-Wesley, 1998), a book to which he gave 5 stars on Amazon.com, adding the following review:

> **This book contributed to the design for metaclasses in Python 2.2**
>
> Too bad this is out of print; I keep referring to it as the best tutorial I know for the difficult subject of cooperative multiple inheritance, supported by Python via the `super()` function.[6].

For Python 3.5—in alpha as I write this—PEP 487 - Simpler customization of class creation puts forward a new special method, `__init_subclass__` that will allow a regular class (i.e., not a metaclass) to customize the initialization of its subclasses. As with class decorators, `__init_subclass__` will make class metaprogramming more accessible and also make it that much harder to justify the deployment of the nuclear option —metaclasses.

If you are into metaprogramming, you may wish Python had the ultimate metaprogramming feature: syntactic macros, as offered by Elixir and the Lisp family of languages. Be careful what you wish for. I'll just say one word: MacroPy.

6. Amazon.com catalog page for *Putting Metaclasses to Work*. You can still buy it used. I bought it and found it a hard read, but I will probably go back to it later.

Soapbox

I will start the last soapbox in the book with a long quote from Brian Harvey and Matthew Wright, two computer science professors from the University of California (Berkeley and Santa Barbara). In their book, *Simply Scheme*, Harvey and Wright wrote:

> There are two schools of thought about teaching computer science. We might caricature the two views this way:
>
> 1. **The conservative view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to discipline their work in such a way that 500 mediocre programmers can join together and produce a program that correctly meets its specification.
> 2. **The radical view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to expand their minds so that the programs can fit, by learning to think in a vocabulary of larger, more powerful, more flexible ideas than the obvious ones. Each unit of programming thought must have a big payoff in the capabilities of the program.[7]
>
> — Brian Harvey and Matthew Wright
> *Preface to Simply Scheme*

Harvey and Wright's exaggerated descriptions are about teaching computer science, but they also apply to programming language design. By now, you should have guessed that I subscribe to the "radical" view, and I believe Python was designed in that spirit.

The property idea is a great step forward compared to the accessors-from-the-start approach practically demanded by Java and supported by Java IDEs generating getters/setters with a keyboard shortcut. The main advantage of properties is to let us start our programs simply exposing attributes as public—in the spirit of *KISS*—knowing a public attribute can become a property at any time without much pain. But the descriptor idea goes way beyond that, providing a framework for abstracting away repetitive accessor logic. That framework is so effective that essential Python constructs use it behind the scenes.

Another powerful idea is functions as first-class objects, paving the way to higher-order functions. Turns out the combination of descriptors and higher-order functions enable the unification of functions and methods. A function's `__get__` produces a method object on the fly by binding the instance to the `self` argument. This is elegant.[8]

7. Brian Harvey and Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. Full text available at Berkeley.edu.
8. *Machine Beauty* by David Gelernter (Basic Books) is an intriguing short book about elegance and aesthetics in works of engineering, from bridges to software.

Finally, we have the idea of classes as first-class objects. It's an outstanding feat of design that a beginner-friendly language provides powerful abstractions such as class decorators and full-fledged, user-defined metaclasses. Best of all: the advanced features are integrated in a way that does not complicate Python's suitability for casual programming (they actually help it, under the covers). The convenience and success of frameworks such as Django and SQLAlchemy owes much to metaclasses, even if many users of these tools aren't aware of them. But they can always learn and create the next great library.

I haven't yet found a language that manages to be easy for beginners, practical for professionals, and exciting for hackers in the way that Python is. Thanks, Guido van Rossum and everybody else who makes it so.