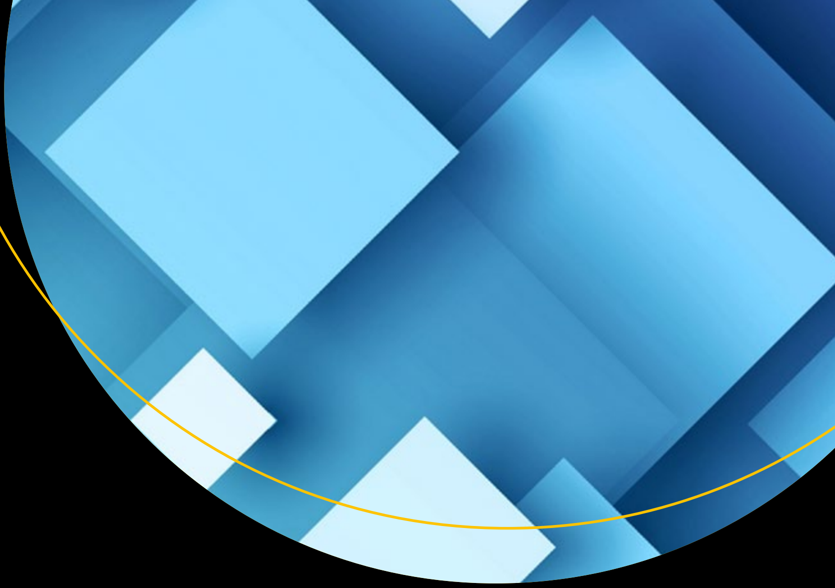

Python Descriptors

Jacob Zimmerman

Apress®

Python Descriptors

Jacob Zimmerman

Apress®

Python Descriptors

Jacob Zimmerman
Placeholder, New York, USA

ISBN-13 (pbk): 978-1-4842-2504-2 ISBN-13 (electronic): 978-1-4842-2505-9
DOI 10.1007/978-1-4842-2505-9

Library of Congress Control Number: 2016961318

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Jacob Zimmerman is a blogger, gamer (tabletop more so than video games), and programmer who was born and raised in Wisconsin. He has a twin brother who could also be considered to have all those traits.

Jacob has his own programming blog that focuses on Java, Kotlin, and Python programming, Programming Ideas with Jake. He also writes for a gaming blog with his brother-in-law called Ramblings of Jacob and Delos.

His brother writes a JavaScript blog called JoeZimJS and works with his best friend on a gaming Youtube channel called Bork & Zim Gaming, which Jacob helps out with on occasion.

Programming Ideas with Jake
`http://programmingideaswithjake.wordpress.com/`
Ramblings of Jacob and Delos
`http://www.ramblingsofjacobanddelos.com/`
JoeZimJS
`http://www.joezimjs.com`

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and health care providers.

Acknowledgments

In order to be sure that I had gotten everything right - it would really suck for a "comprehensive guide" to be missing a big chunk of functionality or to get anything wrong - I enlisted the help of some Python experts. In return for their help, I let them introduce themselves to you here. That's not all I did in return, but it's all you're going to see :)

Emanuel Barry is a self-taught Python programmer who loves pushing the language to its limits as well as explore its darkest corners. He has to do a lot of proofreading and editing for a local non-for-profit organization, and decided to combine his love of Python and knowledge sharing with his background in proofreading to help make this book even better. He can often be found in the shadows of the mailing lists or the issue tracker, as well as the Python IRC channel, as Vgr.

Chris Angelico has played around with Python since the late 90s, getting more serious with the language in the mid 2000s. As a PEP Editor and active participant in the various mailing lists, he keeps well up to date with what's new and upcoming in the language, and also shares that knowledge with fledgling students in the Thinkful tutoring/mentoring program. When not coding in Python, he is often found wordsmithing for a Dungeons & Dragons campaign, or exploring the linguistic delights of Alice in Wonderland and similar works. If you find a subtle Alice reference in this text, blame him!

https://github.com/Rosuav

Kevin Mackay is a software engineer who has been programming in Python since 2010 and is currently working at BBC, improving the Taster platform. He is enthusiastic about open source software and occasionally contributes to the 3D graphics application, Blender. He can be found on the Python IRC channel as yakca, or hiking on a mountain somewhere in Scotland.

Introduction

Python is a remarkable language with many surprisingly powerful features baked into it. Generators, metaclasses, and decorators are some of those, but descriptors are the topic of this guide.

Code Samples

All code samples are written in Python 3, since that is the most recent version, but all the ideas and principles taught in this book apply to Python 2 as well, as long as you're using new style classes. There is one thing that doesn't work quite properly in PyPy, which is discussed in the section that first looks at data and non-data descriptors.

The Descriptor Tools Library

Written alongside this book was a library, called descriptor_tools, which can be installed with pip. It contains the fruition of a lot of the ideas and helpers to make it easier to implement them all. It's an open source project with a public GitHub repository[1].

Note: superscript letters like the one at the end of the previous line are in reference to the bibliography at the back of the book, which includes URLs to the referenced site.

Conventions in This Book

When the text mentions "class" and "instance" in a general sense, they refer to a class that has a descriptor attribute and to instances of such classes, respectively. All other classes and instances will be referred to more specifically.

PART I

About Descriptors

This part provides a deep explanation of what descriptors are, how they work, and how they're used. Enough information will be given to be able to look at any descriptor and understand how it works and why it works that way.

It's not difficult to figure out how to create descriptors from the information given, but little to no guidance is given to help with this. Part 2 will cover that with a bunch of options for creating new descriptors.

CHAPTER 1

What Is a Descriptor?

Put very simply, a descriptor is a class that can be used to call a method with simple dot notation, also referred to as attribute access, but there's obviously more to it than that. It's difficult to really explain beyond that without digging a little into how they're implemented. So, here's a high-level view of the descriptor protocol.

A descriptor implements at least one of these three methods: `__get__()`, `__set__()`, or `__delete__()`. Each of these methods has a list of parameters that are needed, which will be discussed later in the book, and each is called by a different sort of access of the attribute the descriptor represents. Doing simple `a.x` access will call the `__get__()` method of `x`, setting the attribute using `a.x = value` will call the `__set__()` method of `x`, and using `del a.x` will call, as expected, the `__delete__()` method of `x`.

As stated earlier, only one of the methods needs to be implemented in order to be considered a descriptor, but any number of them can be implemented. And, depending on the descriptor type and which methods are implemented, not implementing certain methods can restrict certain types of attribute access or provide an interesting alternative behavior for them. There are two types of descriptors based on which sets of these methods are implemented: data and nondata.

Data Descriptors vs. Nondata Descriptors

A data descriptor implements at least `__set__()` or `__delete__()`, but can include both. They also often include `__get__()` since it's rare to want to set something without also being able to get it too. You *can* get the value, even if the descriptor doesn't include `__get__()`, but it's either a roundabout process or the descriptor writes it to the instance. That will be discussed more later in the book.

A nondata descriptor *only* implements `__get__()`. If it adds `__set__()` or `__delete__()` to its method list, it becomes a data descriptor.

Unfortunately, the PyPy interpreter (up to version 2.4.0; the fix is in the next version, which hadn't been released yet at the time of writing) gets this a little bit wrong. It doesn't take `__delete__()` into consideration until it knows that it's a data descriptor, and PyPy

Electronic supplementary material The online version of this chapter (doi: 10.1007/978-1-4842-2505-9_1) contains supplementary material, which is available to authorized users.

doesn't believe something is a data descriptor unless `__set__()` is implemented. Luckily, since a majority of data descriptors implement `__set__()`, this rarely becomes a problem.

It may seem like the distinction is pointless, but it is not. It comes into play upon attribute look up. This will be discussed more later in the book, but basically the distinction is based on the types of uses it provides.

The Use of Descriptors by Python

It is worth noting that descriptors are an inherent part of how Python works. Python is known to be a multiparadigm language, and as such it supports paradigms such as functional programming, imperative programming, and object-oriented programming (among others). This book does not attempt to go into depth about the different paradigms, only the object-oriented programming paradigm will be observed. Descriptors are used implicitly in Python for the language's object-oriented mechanisms. As it will be explained in the chapters that follow, methods are implemented using descriptors. As you may guess from reading this, it is because of descriptors that object-oriented programming is possible in Python. Descriptors are very powerful and advanced, and this book aims to teach Python programmers how to use them fully.

Summary

Descriptors occupy a large part of the Python language, as they can replace attribute access with method calls and even restrict which types of attribute access is allowed. Now that you have a broad idea of how descriptors are implemented as well as their use by the language, let's dig a little deeper yet and gain a better understanding of how they work.

CHAPTER 2

The Descriptor Protocol

In order to get a better idea of what descriptors are good for, it is best to finish showing the full descriptor protocol. It's time to see the full signature of the protocol's methods and what the parameters are.

__get__(self, instance, owner)

This method is clearly the method for retrieving whatever data or object the descriptor is meant to maintain. Obviously, `self` is a parameter, since it's a method. Also, it receives `instance` and/or `owner`.

Let's start with `owner`, which is the class that the descriptor is accessed from, or the class of the instance it's being accessed from. When you make the call `A.x`, where `x` is a descriptor object with `__get__()`, it's called with an `owner` and the `instance` is set to `None`. So the lookup gets effectively transformed into `A.__dict__['x'].__get__(None, A)`. This lets the descriptor know that `__get__()` is being called from a class, not an instance. The descriptor `owner` is also often written to have a default value of `None`, but that's largely an optimization that only built-in descriptors use.

Now, on to the other parameters. The parameter `instance` is the instance that the descriptor is being accessed from, if it is being accessed from an instance. As mentioned earlier, if `None` is passed into `instance`, the descriptor knows that it's being called from the class level. But, if `instance` is *not* `None`, then it tells the descriptor which instance it's being called from. So an `a.x` call will be effectively translated to `type(a).__dict__['x'].__get__(a, type(a))`. Notice that it still receives the instance's class. Notice also that the call still starts with `type(a)`, not just `a`, because descriptors are stored on classes. In order to be able to apply per-instance as well as per-class functionality, descriptors are given `instance` *and* `owner` (the class of the instance). How this translation and application happens will be discussed later in the book.

Remember that this applies to `__set__()` and `__delete__()` as well and `self` is an instance of the descriptor itself. It is not the instance that the descriptor is being called from, but instead, the `instance` parameter is the instance the descriptor is being called from. This may sound confusing at first, but don't worry if you don't understand for now, everything will be explained further.

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_2

The `__get__()` method is the only one that bothers to get the class separately. That's because it's the only method on nondata descriptors, which are generally made at a class level. The built-in decorator `classmethod` is implemented using descriptors and the `__get__()` method. In that case, it will make use of the `owner` parameter alone.

__set__(self, instance, value)

As mentioned earlier, `__set__()` does not have an `owner` parameter that accepts a class, and `__set__()` does not need it, since data descriptors are generally designed for storing per-instance data. Even if the data were being stored on a per-class level, it should be stored internally without needing to reference the class.

The parameter `self` should be self-explanatory now; so the next parameter is `instance`. This is the same as it is for the `__get__()` method. In this case, though, your initial call is `a.x = someValue`, which is then translated into `type(a).__dict__['x'].__set__(a, someValue)`.

The last parameter is `value`, which is the value the attribute is being assigned.

One thing to note: when setting an attribute that is currently a descriptor from the class level, it will replace the descriptor with whatever is being set. For example, `A.x = someValue` does not get translated to anything; `someValue` replaces the descriptor object stored in `x`. To act on the class, see the note below).

__delete__(self, instance)

After having learned about the `__get__()` and `__set__()` methods, `__delete__()` should be easy to figure out. The parameters `self` and `instance` are the same as in the other methods, but this method is invoked when `del a.x` is called and is translated to `type(a).__dict__['x'].__delete__(a)`.

Do not accidentally name it `__del__()` as that won't work as intended; `__del__()` would be the destructor of the descriptor instance, not of the attribute stored within.

It must be noted that, again, `__delete__()` does not work from the class level, just like `__set__()`. Using `del` from the class level will remove the descriptor from the class's dictionary rather than calling the descriptor's `__delete__()` method.

■ **Note** If you want a descriptor's `__set__()` or `__delete__()` methods to work from the class level, the descriptor must be created on the class's metaclass. When doing so, everything that refers to `owner` is referring to the metaclass, while a reference to `instance` refers to the class. After all, classes are just instances of metaclasses. The section on metadescriptors will explain that in greater detail.

Summary

That's the sum total of the descriptor protocol. With a basic idea of how it works, let's get a high-level view of the types of things that can be done with descriptors.

CHAPTER 3

What Are Descriptors Good For?

Data Encapsulation

One of the most useful aspects of descriptors is their ability to encapsulate data so well. With descriptors, you have a simple way of accessing an attribute via attribute access notation (`a.x`) while allowing more complex actions to happen in the background.

Reuse of Read/Write Patterns

Using specialized descriptors, you can reuse code that you used with reading and/or writing of attributes. These can be used for repetitious attributes within the same class or attribute types shared by other classes.

Lazy Instantiation

You can use descriptors to define a really simple syntax for lazily instantiating an attribute. There will be code provided for a nice lazy attribute implementation later in the book.

Properties (made with `property` built in, which is implemented using descriptors) that only have a getter provided can be considered lazy attributes. For example, if a `Circle` object stores only the radius, a property that calculates the area of the circle could be considered lazy.

Validation

Many descriptors are written simply to make sure that data being passed in conform to the class's or attribute's invariants. Such descriptors can usually also be designed as handy decorators.

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_3

Writing for the Class Level

Because descriptors are stored at the class scope instead of the instance scope, it allows you to do more robust things at the class level. For instance, descriptors are what makes `classmethod` and `staticmethod` work, which will be explained shortly.

Hiding Function Calls Within Attribute Access

This is actually more of the definition of how descriptors work than what they're good for, but it's worth mentioning because abstracting how it works helps to spur the mind to come up with other possible uses.

Basically, a descriptor is there to be a function that is called simply by accessing it as an attribute. Using simple attribute access calls `__get__()`; attempting to assign an object to that attribute delegates to `__set__()` on the descriptor; and using `del` redirects to `__delete__()`. This was already mentioned earlier, but restating it this way may help you remember it for devising use cases.

Triggering Actions

Descriptors can be used to trigger certain actions when the attribute is accessed. For example, the observer pattern can be implemented in a per-attribute sense to trigger calls to the observer whenever an attribute is changed.

Summary

I have explained how descriptors allow for a variety of nice features that are extremely difficult or tedious to implement otherwise. In fact, Python comes with a few baked-in descriptors that already make developers' lives easier. These will be explored in the next chapter.

CHAPTER 4

Descriptors in the Standard Library

There are three basic, well-known, built-in descriptors: `property`, `classmethod`, and `staticmethod`.

Many people who know about `property` also know that it is a descriptor, and it's probably the first descriptor they learn about. This is what usually drives them to learn more about descriptors. But a lot of people don't know that `classmethod` and `staticmethod` are also descriptors. They appear to be super magical constructs built in to the language that no one could reproduce in pure Python. Once you have an understanding of descriptors, though, their basic implementation becomes relatively obvious. In fact, an example will be provided for all three in simplified, pure Python code.

This chapter will explain that *all* methods are actually implemented with descriptors. Normal methods are actually done "magically," since the descriptor creation is implicit, but it's still not entirely magical because it's done using a language construct that anyone could create.

`property`

This book doesn't include instructions for how to use the `property` class and decorator; the focus here is on understanding and creating descriptors. The official documentation for using `property` can be found in Python's documentation.[1]

Of all the descriptors out there, property is likely the most versatile. This is because it doesn't really do anything, but rather allows the user to inject their wanted functionality into code by providing their own getters, setters, and deleters.

[1]Python documentation on property: `http://tinyurl.com/ljsmxck`

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_4

To get a better idea of how it works, here is a simplified pure Python implementation of property:

```
class property:
    def __init__(self, fget=None, fset=None, fdel=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel

    def __get__(self, instance, owner):
        if instance is None:
            return self
        elif self.fget is None:
            raise AttributeError("unreadable attribute")
        else:
            return self.fget(instance)

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        else:
            self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        else:
            self.fdel(instance)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel)
```

As stated before and can be seen in the code section, the property class has almost no real functionality of its own; it simply delegates to the functions given to it. When a function is not provided for a certain method to delegate to, property assumes that it is a forbidden action and raises an AttributeError with an appropriate message.

A nice thing about the property class is that it largely just accepts methods, and even the constructor, which can be given all three methods at once, is capable of being called with just one, or even none. Because of this, the constructor and other methods can be used as decorators in a very convenient syntax. Check out the Python documentation[2] to learn more about it.

Omitted from this code example is the doc functionality, where it sets its own __doc__ property based on what is passed in through the __init__() doc parameter or using __doc__ from fget if nothing is given. Also omitted is the code that sets other attributes on property, such as __name__, in order to make it appear even more like a simple attribute. But this does not seem important enough to worry about, since the focus here is more on the main functionality.

classmethod

The descriptor classmethod can be used as a decorator, but, unlike property, there's no good reason *not* to use it as one. The use of the descriptor classmethod is an interesting concept that doesn't exist in many other languages (if any). Python's type system, which uses classes as objects, makes use of classmethod easy and worthwhile.

Here's the Python code for classmethod:

```
class classmethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        return functools.partial(self.func, owner)
```

That's all there is to it. The descriptor classmethod is a nondata descriptor, so it only implements __get__(). This __get__() method completely ignores the instance parameter because, as the class name implies, the method has nothing to do with an instance of the class and only deals with the class itself. What's really nice is the fact that this can still be called from an instance without any issues.

Why does the __get__() method return a functools.partial object with the owner passed in? To understand this, think about the parameter list of a function marked as a classmethod. The first parameter is the class parameter, usually named cls. This parameter is filled in the call to partial so that the returned function can be called with just the arguments the user wants to explicitly provide. The true implementation doesn't use partial, but works similarly.

Again, the code that sets __name__ or __doc__ is omitted in order to simply show how the main functionality works.

[2]Python documentation on property: http://tinyurl.com/ljsmxck.

staticmethod

The descriptor staticmethod is strange in that it's a method that is really just a function, but it is "attached" to a class. Being part of the class doesn't give it much extra effectiveness other than showing users that it is associated with that class and giving it a more specific namespace. Also, interestingly, because staticmethod and classmethod are implemented using descriptors, they're inherited by subclasses.

Some other languages, such as Java, only really need the staticmethod concept because all functions must be methods associated with a class.

Knowing that, the implementation of staticmethod becomes easy to realize:

```
class staticmethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        return self.func
```

Essentially, staticmethod just accepts a function and then returns it when `__get__()` is called.

Regular Methods

As I stated earlier, regular methods implicitly use descriptors as well. In fact, all functions can be used as methods. This is because functions are nondata descriptors as well as callables.

Here is a Python implementation that roughly shows how a function looks:

```
class function:
    def __call__(self, *args, **kwargs):
        # do something

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        else:
            return functools.partial(self, instance)
```

This is not a very accurate representation; the return statements are a bit off. When you access a method from an instance without calling it, the returned object isn't a partial object, it is a "bound method." A bound method is one that already has self "bound" to it, but it has yet to be called, passing in the other arguments if needed. When it's called from the class, it only returns the function itself. In Python 2, this was an "unbound method," which is basically the same thing. This idea of creating "unbound" versions when instance is None will be discussed later, so please keep it in mind.

Summary

In this chapter, I have explained the built-in uses of descriptors that many Python developers use quite often. Now that you’ve seen some examples, let’s get a closer look at how they work, digging into the real differences between data and nondata descriptors.

CHAPTER 5

Attribute Access and Descriptors

In Chapter X it was stated that attribute access calls get transformed into descriptor calls, but it was not stated how. The simple answer is that `__getattribute__()` does it, but there's more to it than that, so this answer is also entirely insufficient. This simplification is helpful, since it tells you that, if you want to customize `__getattribute__()` and have it continue working with descriptors, you need it to delegate to the default implementation with `super()`.

It is useful to know that descriptor access isn't *actually* transformed; the transformations are effectively just the call used by `__getattribute__()` in the end.

Instance Access

The method `__getattribute__()` has an order of priority that describes where to look for attributes and how to react to them. That priority is why there is a formal definition between data descriptors and nondata descriptors. Here is that list of priorities:

1. Data descriptors
2. Instance attributes
3. Nondata descriptors and class attributes
4. The function `__getattr__` (might be called separately from `__getattribute__`)

The first thing `__getattribute__()` does is look in the class dictionary for the attribute. If it's not found, it works its way down the method resolution order (MRO) of classes (the subclasses in a linear order) to continue looking for it. If it's still not found, it'll move to the next priority. If it *is* found, it checks to see if it is a data descriptor. If it's not, it moves on to the next priority. If it turns out to be a data descriptor, it'll call `__get__()` and return the result, assuming it has a `__get__()` method. If it doesn't have a `__get__()` method, then it moves on to the next priority.

That's a lot of ifs, and that's just within the first priority to determine whether a viable data descriptor is available to work with. Luckily, the next priority is simpler.

Next in the priority list is checking the instance dictionary. If it exists there, it simply returns that. Otherwise, it moves to the next priority.

In this priority, it checks through the class dictionary again, working its way down the MRO list if needed. If nothing is found, it moves to the next priority. Otherwise, it checks the found object to see if it's a nondata descriptor. If so, it transforms the call to the descriptor's `__get__()` and returns the result. Otherwise, it simply returns the object. Again, if the descriptor doesn't define a `__get__()` method, the descriptor object itself will be returned.

If all else has failed up to this point, it checks with `__getattr__()` for any possible custom behavior regarding attribute access. If there's nothing, an `AttributeError` is raised.

With this complicated definition, Python users should be grateful that a lot of work has been put into optimizing this access algorithm to the point that it's remarkably fast.

Figure 5-1 is a flowchart showing how descriptors are accessed, with blue bands denoting each priority.

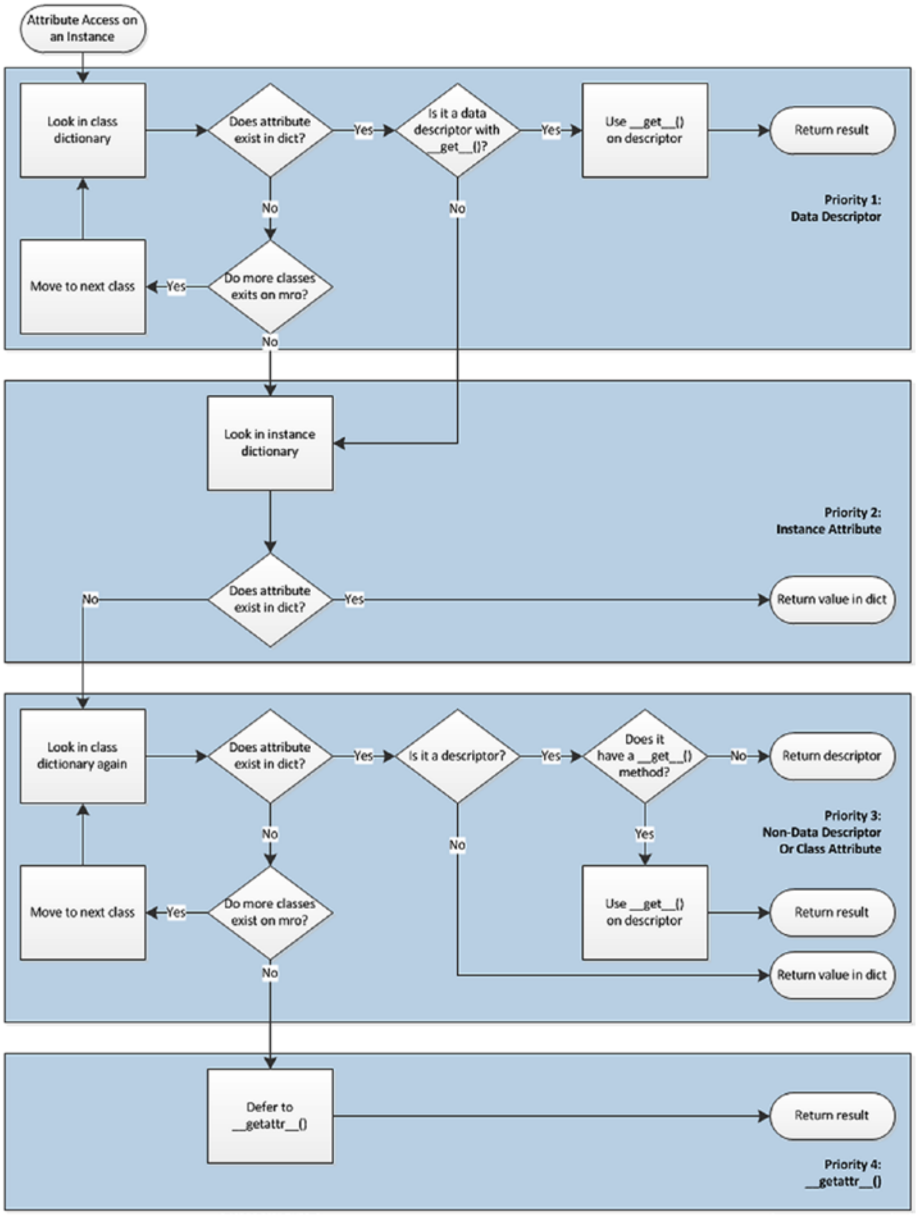
Attribute Access on an Instance
Look in class dictionary
Does attribute exist in dict?
Yes
Is it a data descriptor with __get__()?
Yes
Use __get__() on descriptor
Return result
No
No
Priority 1: Data Descriptor
Move to next class
Yes
Do more classes exits on mro?
No
Look in instance dictionary
Priority 2: Instance Attribute
No
Does attribute exist in dict?
Yes
Return value in dict
Look in class dictionary again
Does attribute exist in dict?
Yes
Is it a descriptor?
Yes
Does it have a __get__() method?
No
Return descriptor
No
No
Yes
Priority 3: Non-Data Descriptor Or Class Attribute
Move to next class
Yes
Do more classes exist on mro?
Use __get__() on descriptor
Return result
No
Return value in dict
Defer to __getattr__()
Return result
Priority 4: __getattr__()

Figure 5-1. *Flowchart showing how descriptors are accessed*

Class Access

In the common case where the class's metaclass is `type` or there are no new attributes on the metaclass, class access can be viewed in a simplified way compared to instance access. It doesn't even have a priority list. It still uses `__getattribute__()`, but it's the one defined on its metaclass. It simply searches through the class dictionary, progressing through the MRO as needed. If found, it checks to see if it's a descriptor with the `__get__()` method. If so, it makes the proper call and returns the result. Otherwise, it just returns the object. At the class level, though, it doesn't care if the descriptor is data or nondata; if the descriptor has a `__get__()` method, the method is used. Figure 5-2 is a diagram of this process.

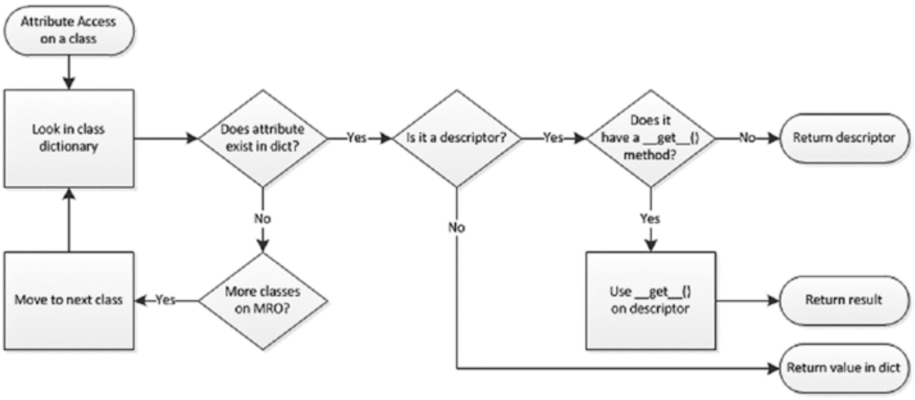

Figure 5-2. *Process of*

If nothing was found, an `AttributeError` is raised.

Unfortunately, if there are new attributes on the metaclass, this simplification is unhelpful, since they might be used in the look up. In fact, class access looks almost exactly like instance access (replacing "class" with "metaclass" and "instance" with "class") with one big difference. Instead of checking just the current instance/class dictionary, it checks through the MRO of it as well, which would be the hierarchy of its metaclasses. It also still treats descriptors on the class as descriptors, rather than automatically returning the descriptor object. Figure 5-3 shows the full class access diagram, with all the priority levels.

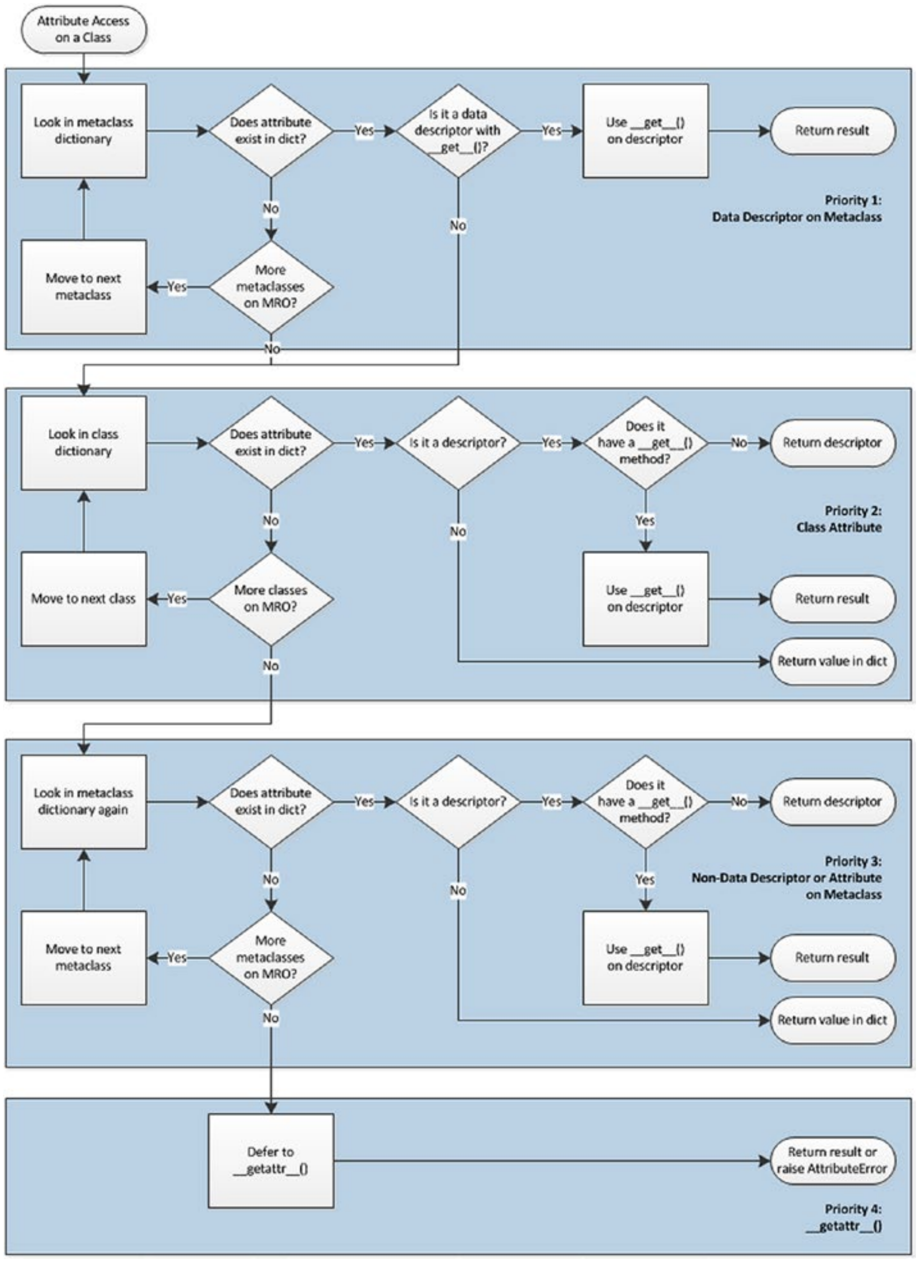
Attribute Access on a Class
Look in metaclass dictionary
Does attribute exist in dict?
Yes
Is it a data descriptor with __get__()?
Yes
Use __get__() on descriptor
Return result
No
No
Priority 1: Data Descriptor on Metaclass
Move to next metaclass
Yes
More metaclasses on MRO?
No
Look in class dictionary
Does attribute exist in dict?
Yes
Is it a descriptor?
Yes
Does it have a __get__() method?
No
Return descriptor
No
No
Yes
Priority 2: Class Attribute
Move to next class
Yes
More classes on MRO?
Use __get__() on descriptor
Return result
No
Return value in dict
Look in metaclass dictionary again
Does attribute exist in dict?
Yes
Is it a descriptor?
Yes
Does it have a __get__() method?
No
Return descriptor
No
No
Yes
Priority 3: Non-Data Descriptor or Attribute on Metaclass
Move to next metaclass
Yes
More metaclasses on MRO?
Use __get__() on descriptor
Return result
No
Return value in dict
Defer to __getattr__()
Return result or raise AttributeError
Priority 4: __getattr__()

Figure 5-3. *Class action with all of the priority levels*

Set and Delete Calls

Setting and deleting calls are done just a little bit differently. If the required `__set__()` or `__delete__()` method doesn't exist, and it's a data descriptor, an `AttributeError` is raised. The other difference is the fact that setting and deleting calls never get beyond the instance priority. If the attribute doesn't exist on the instance, setting will add it and deleting will raise an `AttributeError`.

Figure 5-4 is a flowchart depicting what happens for setting and deleting calls.

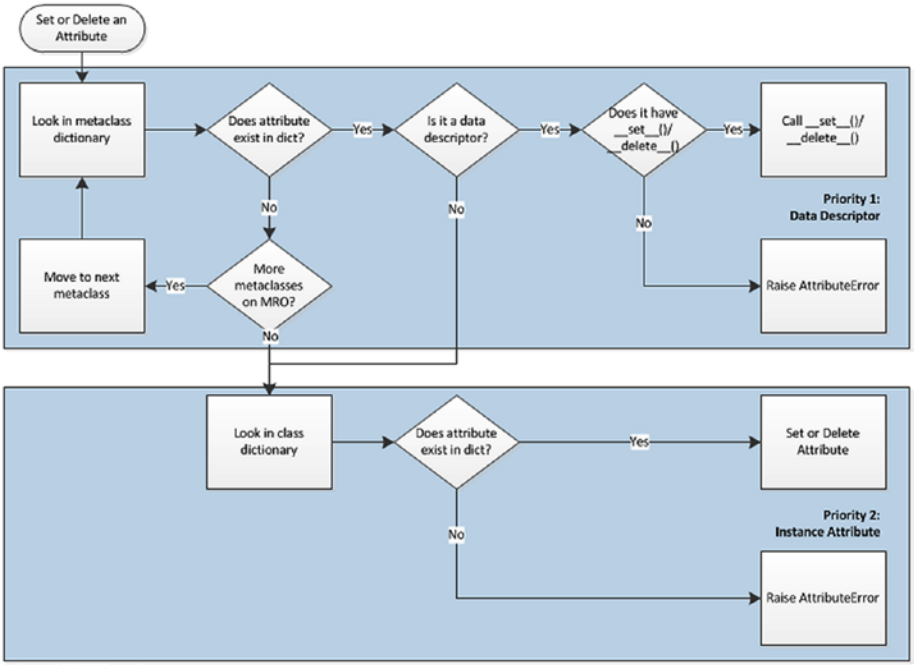
Set or Delete an Attribute
Look in metaclass dictionary
Does attribute exist in dict?
Yes
Is it a data descriptor?
Yes
Does it have __set__()/ __delete__()
Yes
Call __set__()/ __delete__()
No
No
No
Priority 1: Data Descriptor
Move to next metaclass
Yes
More metaclasses on MRO?
Raise AttributeError
No
Look in class dictionary
Does attribute exist in dict?
Yes
Set or Delete Attribute
No
Priority 2: Instance Attribute
Raise AttributeError

Figure 5-4. *Setting and deleting calls*

The Reasoning Behind Data vs. Nondata Descriptors

Now that the difference between data and nondata descriptors has been explained, it should be explained *why* these two versions exist.

The first place to look at is the built-in use cases for each type within the language and standard library. The prime example of a data descriptor is `property`. As its name suggests, its purpose is to create properties for classes (replace getter and setter methods with a syntax that looks like simple attribute use). That means class-level access is not intended since properties represent fields on an instance.

Meanwhile, the primary use case for nondata descriptors is decorating methods for different usages (`classmethod`, `staticmethod`, and especially the implicit descriptor used for normal methods). While these can be called from instances (and normal methods *should* be called from instances), they're not meant to be *set* or *deleted* from instances. Methods are assignments on the class. A function can be assigned to an instance, but it doesn't make it a method, since `self` is not automatically provided as the first argument when called. Also, when it comes to the magic dunder methods (methods with two leading and two trailing underscores) being called through the normal, "magical" way, Python is optimized to look directly on the class, skipping over anything that may have been assigned to the instance.

Summary

Rarely is it useful to know the *full* depth of what is happening behind the scenes of attribute calls, and even knowing the basic priority list rarely comes into play, since descriptors generally do what is obvious, once you understand how they're accessed. There are times, though, when the priority list, and possibly even the full depth, will help in understanding why a descriptor isn't working as hoped or how to set up a descriptor to do a more complicated task.

PART II

Making Descriptors

Finally, the fun part has arrived! Despite the simplicity of the descriptor protocol, there are so many ways that a descriptor can be used and created that, even though Part 1 was pretty lengthy, this part will be even longer.

Part 1 can be enough to help you create descriptors, but it doesn’t give tips, patterns, or real guidance for doing so. Part 2 is filled to the brim with those.

CHAPTER 6

Which Methods Are Needed?

When designing a descriptor, it must be decided which methods will be included. It can sometimes help to decide right away if the descriptor should be a data or nondata descriptor, but sometimes it works better to first "discover" which kind of descriptor it is.

The method `__delete__()` is rarely ever needed, even if it is a data descriptor. That doesn't mean it shouldn't ever be included, however. If the descriptor is going to be released into open domain, it wouldn't hurt to add the `__delete__()` method simply for completeness for cases when a user decides to call `del` on it. If you don't, an `AttributeError` will be raised upon trying to delete it.

The method `__get__()` is almost always needed, for data *and* nondata descriptors. It is required for nondata descriptors, and the typical case where `__get__()` isn't required for data descriptors is if `__set__()` assigns the data into the instance dictionary under the same name as the descriptor (what I call set-it-and-forget-it descriptors). Otherwise, it is almost always needed for retrieving the data that is set in a data descriptor, so unless the data are assigned to the instance to be automatically retrieved without `__get__()` or the data are write-only, a `__get__()` method would be necessary. It should be kept in mind that if a descriptor doesn't have a `__get__()` method and `instance` doesn't have anything in `__dict__` under the same name as the descriptor, the actual descriptor object itself will be returned.

Just like `__delete__()`, `__set__()` is only used for data descriptors. Unlike `__delete__()`, `__set__()` is not regarded as unnecessary. Seeing that `__delete__()` is unused in most common cases, `__set__()` is nearly a requirement for creating data descriptors (which need either `__set__()` or `__delete__()`). If the descriptor's status as data or nondata is being "discovered," generally `__set__()` is the deciding factor. Even if the data are meant to be read-only, `__set__()` should be included to raise an `AttributeError` in order to enforce the read-only nature.

When __get__( ) Is Called Without instance

It often happens that a descriptor's `__get__()` method is the most complicated method on it because there are two different ways it can be called: with or without an `instance` argument (although "without" means that `None` is given instead of an instance).

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_6

When the descriptor is a class-level descriptor (usually nondata), implementing `__get__()` without using `instance` is trivial, since that's the intended use. But when a descriptor is meant for instance-level use and the descriptor is not being called from an instance, it can be difficult to figure out what to do.

Here, I present a few options.

Raise Exception or Return self

The first thing that may come to mind is to raise an exception, since class-level access is not intended, but this should be avoided. A common programming style in Python is called EAFP, meaning that it is "Easier to Ask for Forgiveness than for Permission." What this means is that, just because something isn't used as intended doesn't mean that usage should be disallowed. If the use will hurt invariants and cause problems, it's fine; otherwise, there are other, better options to consider. The conventional solution is to simply return `self`. If the descriptor is being accessed from the class level, it's likely that the user realizes that it's a descriptor and wants to work with it. Doing so can be a sign of inappropriate use, but Python allows freedom, and so should its users, to a point. The property built-in will return `self` (the property object) if accessed is from the class, as an example.

Unbound Attributes

Another solution, which is used by methods, is to have an unbound version of the attribute be returned. When accessing a function from the class level, the function's `__get__()` detects that it does not have an instance, so it just returns the function itself. In Python 2, it actually returned an unbound method, which is where the name comes from. In Python 3, though, they changed it to just the function.

This can work for noncallable attributes as well. It's a little strange, since it turns the attribute into a callable that must receive an instance to return the value. This makes it into a specific attribute look up, akin to `len()` and `iter()`, where you just need to pass in the instance to receive the wanted value.

Here is a stripped-down `__get__()` implementation that works this way:

```
def __get__(self, instance, owner):
    if instance is None:
        def unboundattr(inst):
            return self.__get__(inst, owner)
        return unboundattr
    else:
        ...
```

The inner `unboundattr()` function can use the same implementation code within it that the `else` block does, but it was desirable to show off a universal way to define `unboundattr()` for the sake of completeness, and it leads to less duplicate code by simply calling itself again. Here's a reusable implementation, though, which can be used in any descriptor:

```
class UnboundAttribute:
    def __init__(self, descriptor, owner):
        self.descriptor = descriptor
        self.owner = owner

    def __call__(self, instance):
        return self.descriptor.__get__(instance, self.owner)
```

Using this class, a `__get__()` method that uses unbound attributes can be implemented like this:

```
def __get__(self, instance, owner):
    if instance is None:
        return UnboundAttribute(self, owner)
    else:
        ...
```

The original version relies on closures around `self` and `owner`, which removes its reusability, other than through copying and pasting. But the class takes those two variables in with its constructor to store on a new instance.

The really interesting (and useful) thing about this technique is that the unbound attribute can be passed into a higher-order function that receives a function, such as `map()`. It prevents you from having to write a getter method or ugly lambda. For example, if there was a class like this:

```
class Class:
    attr = UnbindableDescriptor()
```

a `map()` call to a list of `Class` objects would look like this:

```
result = map(lambda c: c.attr, aList)
```

but could be replaced with this:

```
result = map(Class.attr, aList)
```

Instead of passing in a lambda to do the work of accessing the attribute of the `Class` instances, `Class.attr` is passed in, which returns the unbound version of the attribute, a function that receives the instance in order to look up the attribute on the descriptor. In essence, the descriptor provides an implicit getter method to the reference of the attribute.

This is a very useful technique for implementing a descriptor's `__get__()` method, but it has one major drawback: returning `self` is so prevalent that not doing so is highly unexpected. Hopefully, this idea gets some traction in the programming community and becomes the new standard. Also, as will be discussed in Chapter 8 on read-only descriptors, there needs to be a way to access the descriptor object. Luckily, all you need to do is get the `descriptor` attribute from the returned `UnboundAttribute`.

Although it's not the expected behavior, the built-in `function` descriptor already does this, so it won't be too difficult for them to get used to it. People expect unbound method functions when accessing them from the class level, so applying this convention to attributes shouldn't be a huge stretch for them.

Summary

This chapter discussed the decision-making process behind building general descriptors and figuring out which methods you'll want and possibly using unbound attributes with `__get__()`. In the next chapter, I'll dig into even more design decisions that have to be made, at least when it comes to storing values with descriptors.

CHAPTER 7

Storing the Attributes

Now that all the preliminaries are out of the way, it is time to see the part of descriptors that is actually useful: actually storing the attributes that the descriptor represents.

Class-Level Storage

Class-level storage is easy; it's normal storage on the descriptor. As an example, here is a descriptor that creates a basic class-level variable:

```
class ClassAttr:
    def __init__(self, value):
        self.value = value

    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = value
```

This descriptor saves a value on itself as a typical instance attribute, which is simply returned in the `__get__()` method, ignoring whether `instance` is provided or not, since it's a class-level attribute. This attribute can also be accessed through an instance, but making any change to it from the instance will apply the change to every instance of the class. Unfortunately, due to `__set__()` not being called when a descriptor is accessed from the class level, the variable storing the descriptor will be reassigned to the new value, rather than it being passed to `__set__()`.

For more details about making class-level descriptors that `__set__()` and `__delete__()` can be used on, check out the section at the end of this chapter about metadescriptors.

Descriptors aren't just for class-level attributes, though; they're used for instance-level attributes too. There are two broad strategies for storing instance-level attributes with descriptors:

- on the descriptor
- in the instance dictionary

Each strategy has some hurdles to clear for a reusable descriptor. When storing it on the descriptor, there are hurdles as to how to store it without problems. As for storing the attributes on the instance dictionary, the difficulty comes from trying to figure out what name to store it under in the dictionary to avoid clashing.

Storing Data on the Descriptor

As shown before, saving a simple value on the descriptor is how a class-level value is stored. What must be done to store a value on a per-instance basis in one place? What is needed is some way to map an instance to its attribute value. Well, another name for a mapping is a dictionary. Maybe a dictionary would work. Here's what using a dictionary for its storage might look like.

```
class Descriptor:
    def __init__(self):
        self.storage = {}

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

    def __set__(self, instance, value):
        self.storage[instance] = value

    def __delete__(self, instance):
        del self.storage[instance]
```

The `__get__()` method could have returned an unbound attribute, but for the sake of brevity and removing distractions, descriptor examples in the book will return `self` instead.

The `dict` in the code example has solved our problem. Unfortunately, there are a couple shortcomings to using a plain old `dict` for the job.

The first shortcoming to address is memory leaks. A typical `dict` will store the instance used as the key long after the object should have been otherwise garbage collected from lack of use. This is fine for short-lived programs that won't use a lot of memory and if the instances don't suffer from the second shortcoming mentioned later.

Let's look at how to get around the first problem first. The descriptor needs a way to stop caring about instances that are no longer in use. The `weakref` module provides just that. Weak references allows variables to reference an instance as long as there is a normal reference to it somewhere, but allows it to be garbage collected otherwise. They also allow you to specify behavior that will run as soon as the reference is removed.

The module also provides a few collections that are designed to remove items from themselves as the items are garbage collected. Of those, we want to look at a `WeakKeyDictionary`. A `WeakKeyDictionary` keeps a weak reference to its key, and therefore once the instance that is used as the key is no longer in use, the dictionary cleans the key-value entry out.

So, here's the example again, this time using the `WeakKeyDictionary`.

```
from weakref import WeakKeyDictionary
class Descriptor:
    def __init__(self):
        self.storage = WeakKeyDictionary()

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

    def __set__(self, instance, value):
        self.storage[instance] = value

    def __delete__(self, instance):
        del self.storage[instance]
```

Every change between the previous example and this one has been made bold, and this shows that there really isn't much of a difference. The only difference is that the special dictionary needs to be imported and a `WeakKeyDictionary` needs to be created instead of the normal `dict`. This is a very easy upgrade to make, and many descriptor guides stop here. It works in most situations, so it isn't a bad solution.

Unfortunately, it still suffers from the other shortcoming that a regular `dict` does: it doesn't support unhashable types.

To use an object as a key in a `dict`, it must be hashable. There are a few built-in types that cannot be hashed, namely the mutable collections (`list`, `set`, and `dict`), and maybe a few more. Any object that is mutable (values inside can be changed) and overrides `__eq__()` to compare internal values, the object must be unhashable. If the object is changed in a way that changes equality, suddenly the hash code changes so that it can't be looked up as a dictionary key. Thus, such mutable objects are generally advised to mark themselves as unhashable using `__hash__ = None`. Overriding `__eq__()` will do this automatically; overriding `__hash__` should therefore only be done if equality is constant.

If it weren't for Python providing default implementations of `__eq__()` and `__hash__()` (equality is the same as identity - an object is equal to itself, and nothing else), most objects wouldn't be hashable and thus supported for descriptors using a hashing collection. Luckily, this means that types are hashable by default, but there are still many unhashable types out there.

Again, the `WeakKeyDictionary` is *not* a bad solution; it just doesn't cover all possibilities. Much of the time, it is good enough, but it generally advised not to use it for public libraries, at least not without good warnings in the documentation. After all, the descriptor protocol provides ways to set and delete attributes, so they should support instances of mutable classes.

There needs to be a solution that doesn't suffer from this problem, and there is. The simplest solution is to use the `instance`'s `id` as the key instead of the instance itself. Hooray! Now the dictionary doesn't hold onto unused `instances` anymore, and it doesn't require the classes to be hashable.

Here's what that solution would look like.

```
class Descriptor:
    def __init__(self):
        self.storage = {}

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return self.storage[id(instance)]

    def __set__(self, instance, value):
        self.storage[id(instance)] = value

    def __delete__(self, instance):
        del self.storage[id(instance)]
```

The example switches back to a normal `dict`, so the changes mentioned are based off the differences between this example and the first one again, rather than comparing to the previous one. Every time the storage is being accessed, it's being accessed by `id(instance)` instead of just `instance`.

This seems like a pretty good solution, since it doesn't suffer from either of the problems of the previous two solutions. But it's not a good solution. It doesn't suffer from exactly the same problems of the previous solutions, but it still suffers from a memory leak. Yes, the dictionary no longer stores the instances, so those aren't being kept, but there's no mechanism to clear useless ids from the dictionary. In fact, there's a chance (probably a very tiny one, but a chance nonetheless) that a new instance of the class may be created at the same memory address of an older instance, so the new instance has an attribute equal to the old one until it's changed (if possible, since some descriptors are set up as read-only attributes).

This still doesn't solve the on-descriptor storage problem, but it leads in the right direction. What is needed is a storage system that works like a dictionary, with `instance` as the key, but uses `id(instance)` instead of `hash(instance)` for storage. But, it also needs to clean itself out if an instance is no longer in use.

Since such a thing isn't built in, it will have to be custom-made. Here is that custom dictionary, designed specifically for this book.

```
import weakref

class DescriptorStorage:
    def __init__(self, **kwargs):
        self.storage = {}
        for k, v in kwargs.items():
            self.__setitem__(k, v)
    def __getitem__(self, item):
        return self.storage[id(item)]

    def __setitem__(self, key, value):
        self.storage[id(key)] = value
        weakref.finalize(key, self.storage.__delitem__, id(key))

    def __delitem__(self, key):
        del self.storage[id(key)]
```

The real version obviously has more methods, such as `__iter__`, `__len__`, etc., but the main three uses for storage with a descriptor are implemented here.

This may look a little bit complicated , but it's actually fairly simple. The basics of it is that there is a facade class that acts like a dictionary, delegating most functionality to an inner dictionary, but transforming the given keys to their `id`s. The only real difference is that, in `__setitem__()`, this new class creates a `finalize` weak reference, which takes a reference, a function, and any arguments to send to that function when the reference is garbage collected. In this case, it removes the item (again, stored using `id()`) from the internal dictionary.

The keys to how this storage class works are using an id as the key (which means the instances do not need to be hashable) and weak reference callbacks (which remove unused objects from the dictionary). In essence, this class is a `WeakKeyDictionary` that internally uses the id of the given key as the actual key.

Storing the attribute in the descriptor safely takes a lot more consideration than most people ever actually put into it, but now there is a nice, catch-all solution for doing that. The first two solutions are imperfect, but not useless, though. If the use case for the descriptor allows for the use of either of those solutions, it wouldn't hurt to consider them. They are viable enough for many cases and are likely to be slightly more performant than the custom storage system provided here. For public libraries, though, either the custom dictionary or a solution from the following section should be considered.

Storing on the Instance Dictionary

It's often better to store the data on the instance instead of within the descriptor, provided that a worthwhile strategy for deriving the key can be found. This is because it doesn't require an additional object for storage; the `instance`'s built-in `__dict__` is used. However, some classes will define `__slots__`, and, as such, will not have a `__dict__` attribute to mess with. While this is a good idea regardless, it's worth mentioning that it may fail in some cases.

If you want to make a descriptor safe with `__slots__` while still defaulting to using `__dict__`, you may want to create some sort of alternative that uses on-descriptor storage when a boolean flag is set on creation. There are plenty of ways to implement that, whether using a factory that chooses a different descriptor if the flag is set or the class within has alternate paths based on the flag value. Another, simpler alternative is to document the name that the descriptor stores its values under so that users of the descriptor who want to use `__slots__` can prepare a slot for it. This requires that the descriptor does direct instance attribute setting rather than getting `__dict__` first, though.

Another way to go about this (which doesn't require explicitly asking the user) is to check if the class has a `__dict__` attribute; if it does, then simply use it, but if it doesn't, you'll need to store it on the descriptor instance directly. Checking for the existence of `__slots__` is unreliable as subclasses may not define `__slots__` (while the base class does), so they will have both a `__dict__` attribute and a `__slots__` attribute. It is also possible to define a `__dict__` slot.

Storing the data on the instance using `__dict__` is easy (though often verbose, since referencing the attribute as `a.__dict__['x']` is often needed instead of `a.x` in order to avoid recursively calling the descriptor), as the following example will show. It's a simple example with a location of where to store the data being hard-coded as `"desc_store"`.

```
class InstanceStoringDescriptorBasic:
    location = "desc_store"

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

    def __set__(self, instance, value):
        instance.__dict__[self.location] = value

    def __delete__(self, instance):
        del instance.__dict__[self.location]
```

As shown, it is pretty easy to store on the instance. But why should the values be accessed via `__dict__` and not simple dot notation? There are actually plenty of situations where using dot notation would work just fine. In fact, it works in most situations. The only time that there's a problem is when the data descriptor has the same name that is being used for storage in the dictionary. Often, this case pops up because the descriptor is purposely storing the attribute under its own name, which is almost guaranteed to prevent name conflicts. But it's still possible that an outside data descriptor has the same name as where the main descriptor is trying to store its data. In order to avoid this, it is preferable to always directly reference the instance's `__dict__`. Another good reason is that it makes it more explicit and obvious where the data is being stored.

The next thing to be figured out is how the descriptor knows where to store the data. Hopefully it's obvious that hard-coding a location is a bad idea; it prevents multiple instances of that type of descriptor from being used on the same class without them all contending for the same place.

Asking for the Location

The simplest way to get a location is to ask for it in the constructor. A descriptor like that would look something like this:

```
class GivenNameInstanceStoringDescriptor:
    def __init__(self, location):
        self.location = location

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

    def __set__(self, instance, value):
        instance.__dict__[self.location] = value

    def __delete__(self, instance):
        del instance.__dict__[self.location]
```

The only real difference between this one and the previous one is that it has an `__init__()` method that receives the preferred location name from the user instead of hard-coding the location. In fact, the rest of the code is exactly the same.

Asking for the location to store the attribute value is easy when it comes to creating the descriptor, but is tedious for the user to use and can even be dangerous in the event that the location is required to have the same name as the descriptor, since the user can mess that up. Such is the case with set-it-and-forget-it descriptors, such as the following descriptor which is a descriptor used for validating data using the function provided.

```
class Validated:
    def __init__(self, location, validator):
        self.location = location
        self.validator = validator

    def __set__(self, instance, value):
        if self.validator(value):
            instance.__dict__[self.location] = value
        else:
            raise ValueError("not a valid value for " + self.location)
```

In this `Validated` descriptor, `__init__()` asks for the location to store the real data at. Since this is a set-it-and-forget-it descriptor that lets the instance handle retrieval instead of providing a `__get__()`, the location that the user provides must be the same as the descriptor's name on the class in order for the descriptor to work as intended. For example, if a class was accidentally written like this:

```
class A:
    validatedAttr = Validated('validatedAttribute', validatorFunc)
```

`validatedAttr` is all screwed up. To set it, the user writes `a.validatedAttr = someValue`, but retrieving it requires the user to write `instOfAClass.validatedAttribute`. This may not seem all that bad since it can be fixed easily, but these are the types of bugs that can often be very difficult to figure out and can take a long time to notice. Also, why should the user be required to write in the location when it can be derived somehow?

Set-it-and-forget-it Descriptors

Now set-it-and-forget-it descriptors can finally be explained. Of the three methods in the descriptor protocol, these descriptors generally only implement __set__(), as seen in the example. That's not always the case, though. For example the following lazy initialization descriptor only uses __get__().

```
class lazy:
    def __init__(self, func):
        self.func = func
    def __get__(self, instance, owner):
        if instance is None:
            return self
        value = self.func(instance)
        instance.__dict__[func.__name__] = value
        return value
```

This lazy descriptor is also a decorator over a function, which it replaces and uses to do the lazy initialization. In this case, and in the case of other set-it-and-forget-it descriptors, the descriptor sets the value directly onto the instance, using the same name the descriptor is referenced by. This allows the descriptor to either be a non-data descriptor that is never used more than once - as in the case of `lazy` - or to be a data descriptor that has no need to implement `__get__()`, which is the case with most set-it-and-forget-it descriptors. In many cases, set-it-and-forget-it descriptors can increase lookup speeds by just looking in the instance or even provide other optimizations, like the `lazy` descriptor.

Indirectly Asking for the Location

Something else can be noted about the `lazy` descriptor from the set-it-and-forget-it section, and that's how it was able to determine where to store the attribute; it pulled it from the function that it decorated.

This is a great way to indirectly ask for the name of the descriptor. Since the descriptor, initialized as a decorator, is provided with a function which the descriptor is replacing, using that function's name to look up that name for a place to store the information on `instance`.

Name Mangling

Using the name directly like that, though, can be dangerous for non-data, non-set-it-and-forget-it descriptors, since setting it directly to that location would override its own access (which `lazy` actually intended to have happen). When building a non-data descriptor that doesn't want to write over itself - though the chances are probably pretty slim for that situation to come up - it is best to do some "name mangling" when storing the data. To do so, just add an underscore or two at the beginning of the name. Using at least two leading underscores and at most one trailing underscore causes Python to add its own mangling to the name; using one leading underscore simply signals that the attribute is "private". There's an incredibly low chance that the name is already taken on the `instance`.

Next, what can be done if asking the user for the name is a bad idea and the descriptor isn't also a decorator? How does a descriptor determine its name then? There are several options, and the first one that will be discussed is how a descriptor can try to dig up its own name.

Fetching the Name

It would seem so simple to just look up what a descriptor's name is, but, like any object, a descriptor could be assigned to multiple variables with different names. No, a more roundabout way of discovering one's own name is required.

■ **Note** Inspiration for this technique is attributed to The Zipline Show on YouTube, specifically their video about descriptors[3]. This technique shows up around 22 minutes in. They may have gotten the technique from the book they mention at the beginning of the video, but I took the idea from them, not the book.

The original version of this technique that I adapted a little used the following code.

```
def name_of(self, instance):
    for attr in type(instance).__dict__:
        if attr.startswith('__'): continue
        obj = type(instance).__dict__[attr]
        if obj is self:
            self.name = self.mangle(attr)
            break
```

This method is meant to be added to any descriptor in order to look up its name. If the descriptor's `name` attribute isn't set, the descriptor just runs this method to set it. On the second to last line, it sends the name to a name mangler - which just makes sure it starts with two underscores - instead of using the name as it is. As mentioned in the name mangling section, this may be necessary, but not always.

There's a problem with this method, though: it doesn't handle subclasses. If a class with this descriptor is subclassed and an instance of that subclass tries to use the descriptor before an instance of the original class does, it will fail to look up its name.

This is because the descriptor is on the original class, not the subclass, but the `name_of()` method looks in the class' dictionary for itself. The subclass will not have the descriptor in its dictionary.

Not to worry, though. The version in the library solves this problem by using `dir()` to get all the names of attributes, including from superclasses, and then it delegates those to a function that digs into the `__dict__` of each class on the MRO until it finds what it's looking for. I also removed the name mangling function, allowing you to use that only as necessary. Lastly, it doesn't bother with ignoring attributes that start with a double underscore. Such a check may actually be slower than accessing the attribute and comparing identity, but even if it's not, it largely just clutters the code. Plus, you never know; your descriptor may be used in place of a special method.

The final result looks like this:

```
def name_of(descriptor, owner):
    return first(attr for attr in dir(owner)
                 if (get_descriptor(owner, attr) is descriptor))

def first(iter):
    return next(iter, None)

def get_descriptor(cls, descname):
    selected_class = first(clss for clss in cls.__mro__
                           if descname in clss.__dict__)
    return selected_class.__dict__[descname]
```

Python 3.2 also added a new function in the inspect module called getattr_static() which works just like getattr() except that it doesn't activate a descriptor's __get__() method upon lookup. You could replace the call to get_descriptor() with getattr_static() and it would work the same.

Keying on the ID

Another thing that can be done for relatively safe places to store on the instance is to use the `id()` of the descriptor to generate a location on the instance, somehow. It may seem strange, but the `id` of the descriptor can be used as the key on the instance's location. It seems strange that a non-string can be used as the key in an instance dictionary, but it can.

Unfortunately, it can only be accessed directly via `instance.__dict__[id(desc)]` and not via dot notation or `get/set/hasattr()`. This may actually seem like a plus, since it prevents unwanted access to the attribute, but it also messes up `dir(instance)`, which raises an exception when it finds a non-string key.

On the plus side, it's impossible for this location to clash with user-defined attributes, since those must be strings, and this is an integer, but causing `dir()` to fail is undesirable, so a different solution must be found. Defining a __dir__ method would be overkill and inappropriate in most cases. However, the aggressive programmer could call `object.__dir__()` and remove the `id()` from the list before returning it. As stated, however, this is overkill.

A simple solution is to change the id into a string, i.e. `str(id(desc))` instead of just `id(desc)`. This fixes the `dir()` problem and also opens up the use of `get/set/hasattr()` while still preventing dot notation access, since it's an invalid Python identifier. The likelihood of name clashes is still extremely low, so this is still an acceptable solution.

■ **Note** An interesting little twist of `str(id(desc))` is to use the hexadecimal value, as `hex(id(desc))` instead of the straight string version of the number, preferably removing `'0x'` at the beginning, such as `hex(id(desc))[2:]`. The benefit of this is that the hex string will generally be shorter, which shortens the time needed to calculate the hash value (which is done on lookup and assignment in `__dict__`) by a tiny bit. Yes, the amount of time needed to calculate the hex value is greater than that of calculating the plain string value, but that only needs to be done once, whereas attribute lookup is likely to happen many times. It's a tiny optimization and may not even be worth noting.

There's no good reason to add acceptable characters to the front of the key in order to support dot notation, since dot notation requires the user to know what the name is going to be ahead of time, which they can't since the name changes every time the program is run when using `id()` to derive it. There are other restrictions that a consistently-changing key imposes, one of which is that it makes serialization and deserialization (pickling and unpickling, respectively, done with the `pickle` module, are one of those ways, among others) a little more difficult.

If it's desirable to be able to derive some sort of information from the save location, additional information can be added to the key. For example, the descriptor's class name could be added to the front of the key, for example `type(self).__name__ + str(id(self))`. This gives users who use `dir()` to look through the names on the instance to have some clue as to what that name refers to, especially if there are multiple descriptors that base their name on `id()` on the instance.

Letting the User Take Care of It

The title of this section may sound like it's about asking the user for the name in the descriptor's constructor, but that's not it at all. Instead, this is referring to the approach `property` uses.

One could say that `property` "cheats" by simply assigning functions that you give it to its different methods. It acts as the ultimate descriptor by being almost infinitely customizable, and that's largely what it is. The biggest descriptor-y thing it can't do is become a non-data descriptor (since it defines all three methods of the descriptor protocol), which is fine, since that doesn't work with the intent anyway. Also, the functions fed to the descriptor don't have easy access to the descriptor's internals, so there's a limit to what can be done there.

Interestingly, a large percentage of descriptors could be written using `property` - and actually work better, since there would be no difficulties in figuring out where to save the data - but it certainly has major setbacks. The biggest of those is the lack of DRYness

(Don't Repeat Yourself; DRYness is the lack of unnecessary repeated code) when it comes to reusing the same descriptor idea. If the same code has to largely be rewritten many times for the same effect with `property`, it should be turned into a custom descriptor that encapsulates the repeated part. Sadly, it isn't likely to be a really easy copy-over because of the fact of storing a value. If the descriptor doesn't need to figure that out, though, which is sometimes the case, then the conversion is much easier.

In summary, `property` is a highly versatile descriptor, and even makes some things extremely easy (namely the difficult thing this entire chapter was about), but they're not easily reusable. Custom descriptors are the best solution for that, hence why this book exists!

There aren't many use cases out there for recreating "storage" the way that `property` does it, but there are enough use cases for extending what property does in little ways to make it worthwhile to look into.

Metadescriptors

The restrictions of descriptors and their use with classes can be quite the pain, limiting some of the possibilities that could be wanted from descriptors, such as class constants. It turns out that there *is* a way around it, and that solution will be affectionately called metadescriptors in this book (hopefully the idea and name spreads throughout the advanced Python community :)).

The reason they are called metadescriptors is because the descriptors, instead of being stored on the classes, are stored on metaclasses. This causes metaclasses to take the place of `owner` while classes take the place of `instance`. Technically, that's all there really is to metadescriptors. It's not even required for a descriptor to be specially designed in order for it to be a metadescriptor.

While the idea of metadescriptors is actually pretty simple, the restrictions around metaclasses can make the use of metadescriptors more difficult. The biggest restriction that must be noted is the fact that no class can be derived from more than one metaclass, whether that is specified directly on the class or having multiple subclasses have different metaclasses. Don't forget that, even if there is no metaclass specified, a class is still being derived from the `type` metaclass.

Because of this, choosing to use metadescriptors must be done with caution. Luckily, if the codebase is following the guideline of preferring composition over inheritance, this is less likely to be a problem.

For a good example of a metadescriptor, check out the `ClassConstant` metadescriptor near the end of the next chapter.

Summary

In this chapter, we've looked at a bunch of examples of techniques for storing values in descriptors, looking at options for storing on the descriptor as well as on the instances themselves. Now that we know the basics that apply to a majority of descriptors, we'll start looking at some other relatively common functionality and how it can be implemented.

CHAPTER 8

Read-Only Descriptors

There are many good uses for read-only - or immutable - property descriptors. In fact, there is a lot to back up the idea of having everything be effectively immutable. Unfortunately, due to Python's inherent lack of being able to make anything *actually* immutable, interpreter optimization isn't one of those possible benefits with Python.

There are plenty of other benefits to immutability, but those are beyond the scope of this book. The point of this chapter is to show how a descriptor can make instance-level properties be effectively immutable.

A first stab at making a read-only descriptor might be to not give it a `__set__()` method, but that only works if there's a `__delete__()` method. If there's no `__delete__()` method either, it becomes a non-data descriptor that is checked after instance variables, meaning the assigned variable will be saved on the instance and be the one that is retrieved after the assignment.

No, to truly keep users from assigning new values, `__set__()` is required, but it obviously can't do its normal job. So, what can it do? It can raise an exception. `AttributeError` is probably the best option of the built-in exceptions, but the functionality is almost unique enough to make a custom exception. It's up to you, but the examples use `AttributeError`.

Now that the attribute can't be changed, how does one supply it with its original value? Trying to send it in through the descriptor's constructor would simply end up with the same value for every instance. There needs to be some sort of back door. Three different techniques will be discussed: set-once, secret-set, and forced-set.

Set-Once Descriptors

A set-once descriptor is the most restrictive of the three read-only properties in that it most strongly restricts the number of assignments to once per instance under it.

Set-once descriptors work simply by checking whether a value is already set and acting accordingly. If it's already assigned, it raises an exception; if it's not, then it sets it.

For example, this is what the basic `__set__()` method would look like if the descriptor was using on-descriptor storage in the instance attribute, `storage`.

```
def __set__(self, instance, value):
    if instance in self.storage:
        raise AttributeError("Cannot set new value on read-only property")
    self.storage[instance] = value
```

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_8

First, it checks to see if there's already a value set for the instance. If there is, it raises an `AttributeError`. Otherwise, it sets the value. Simple.

Of the three read-only descriptors, it's also the simplest to use, since it's set the same way descriptors are normally set: using simple assignment. The others each have a roundabout way of getting the value set. Also, because of it having a typical use for setting the value, it's also the easiest to make versatile.

Secret-Set Descriptors

Secret-set descriptors use a "secret" method on the descriptor to initialize the value. The method uses the same parameters as `__set__()` and sets the value exactly the way `__set__()` would do with a normal descriptor.

To have access to that method, the actual descriptor object needs to be accessed. Accessing the descriptor object is the hard part. With the current general standard of returning `self` in the `__get__()` method when no `instance` is provided, getting the descriptor from the instance is as easy as `type(a).x`. Even with returning unbound attributes, this is possible, though requiring an extra step. You may recall that `UnboundAttribute` has a `descriptor` attribute of its own. So, the lookup becomes just a little longer. Instead of just `type(a).x`, it becomes `type(a).x.descriptor`. Once you have access to the descriptor object, all that needs to be done is to call the "secret" set method. Here's an example of a class using a secret-set descriptor called `ROValue` in the `__init__()` method.

```
class A:
    val = ROValue()

    def __init__(self, value):
        type(self).val.set(self, value)
```

The descriptor is accessed, then `set()` - the descriptor's 'secret' set method - is called to initialize the value for the instance. This is more verbose than `self.val = value`, but it works.

In the library, there are some helper functions (some of which are kind of universal) that can be used. The one that is most guaranteed to work in every case (including instance attributes) is `setattribute(instance, attr_name, value)`. There are also some optional parameters with default values that can be set for specifying the specific behavior, but the defaults will try everything (including techniques not show here yet) until something works.

Forced-Set Descriptors

The way that forced-set descriptors work is, instead of using an entirely new method as a back door, it still uses `__set__()`, but with a twist. Instead of just the typical 3 parameters (self, instance, and value), it has a fourth with a default value. This parameter is forced=False. This makes it so that the built-in way of calling `__set__()` will not cause the value to be set. Rather, the descriptor object needs to be accessed and have `__set__()`

called explicitly with the additional forced=True argument. So, if ROValue was a forced-set descriptor instead the previous secret-set one, the basic `__set__()` method would look like this:

```
def __set__(self, instance, value, forced=False):
    if not forced:
        raise AttributeError("Cannot set new value on read-only property")
    # setter implementation here
```

Now the `__set__()` method makes sure that the `forced` parameter is set to `True`. If it's not, then the method fails like any other read only descriptor should. If it *is* `True`, though, then the method knows to let it pass and actually set the value.

If a descriptor is truly only meant to be written to on object initialization, using the set-once descriptor is the best choice. It's harder for users of the descriptor to thwart the read-only nature of the set-once descriptor than it is for the other two options. Choosing between either of the other two is a matter of preference. Some may find that altering the signature of a "magic" method doesn't sit well with them, though some may enjoy the lack of a need for another method. Some may actually prefer the additional method, since they may already be using it as shown in some examples in Chapter 11: Reusing the Wheel. For the most part, choosing between the secret-set and forced-set descriptor designs is just about preference.

Class Constants

Class constants are very much like read-only descriptors except that, when done properly, they don't need to be set-once; instead, they're set upon creation. This requires a little bit of tweaking, though.

First, it must be realized that a descriptor for a class constant must be implemented as a metadescriptor instead of a normal one. Second, every class that has constants will likely have its own set of constants, which means each of those classes will need a custom metaclass just for itself. The reason for that will be explored in depth along with the implementation further below.

To begin, here's the actual descriptor that will be used.

```
class Constant:
   def __init__(self, value):
       self.value = value

   def __get__(self, instance, owner):
       return self.value

   def __set__(self, instance, value):
       raise AttributeError("Cannot change a constant")

   def __delete__(self, instance):
       raise AttributeError("Cannot delete a constant")
```

It's an extremely simple descriptor, receiving a value in its constructor and returning it with a `__get__()` call and raising an `AttributeError` if someone attempts to delete or change the value.

To use this descriptor, though, it must be placed within a metaclass, which must then have a class to derive from it. For an example, here is an instance of a metaclass and class holding several mathematical constants.

```
class MathMeta(type):
    PI = Constant(3.14159)
    e = Constant(2.71828)
    GOLDEN_RATIO = Constant(1.61803)

class Math(metaclass=MathMeta):
    pass
```

Now `PI`, `e`, and the `GOLDEN_RATIO` are constants on the `Math` class. The only way to mess with them is through the metaclass. The only real downside to using a metadescriptor for this is the fact the constants can no longer be accessed through instances of classes with the constant. This isn't really a problem though, since many other languages never permitted that kind of access to begin with.

There's another tiny problem with it: as of writing this, there aren't any Python IDEs that can provide auto-completion for the constants.

So, now that there's a `Constant` metadescriptor and it's understood how to use it, I will now channel my inner Raymond Hettinger by saying, "There must be a better way!"

There *is* a better way. Python allows for dynamically defining classes and metaclasses, and if they're created within a function, that definition can be reused dynamically over and over again. Here's how.

```
def withConstants(**kwargs):
    class MetaForConstants(type):
        pass
    for k, v in kwargs.items():
        MetaForConstants.__dict__[k] = Constant(v)
    return MetaForConstants
```

This function creates a metaclass using each given keyword argument as a new `Constant` and returns the metaclass. Here's what the new `Math` class definition would look like with this function instead of the fully written metaclass.

```
class Math(metaclass=withConstants(PI=3.14159, e=2.71828, GOLDEN_
RATIO=1.61803)):
    pass
```

There! Now, just by setting the resulting metaclass as `Math`'s metaclass, it has the constants provided by the keyword arguments given to `withConstants()`.

Summary

This chapter has examined several different techniques behind making descriptors for read-only attributes (or, at least, read-only-ish attributes). One thing to note in all of this is that none of the techniques actually make it impossible to change the values; they only make it difficult to do so, requiring extra steps in order to signify to the user that doing so is not what was intended. Such is the way of Python; after all, we're all consenting adults here.

CHAPTER 9

Writing `__delete__()`

This is going to be a short chapter, since there isn't really all that much to say, but it didn't really fit in any of the other chapters. Also, `__get__()` and `__set__()` sort of got their own chapters.

Most descriptor tutorials don't even mention what to do with `__delete__()`, and they often don't even have the method on their example descriptors.

If a descriptor is being used only internally (as opposed to being in a public library) and `del` is never called within the internal code, then there is no point in implementing a `__delete__()` method. But in a public library, there is no way to know whether or not users are going to use `del` on the descriptor attributes. Because of that, it is generally safest to include working `__delete__()` methods on data descriptors in a library. How those methods look depends on how the attributes are actually stored.

For internal storage, it deletes the entry from the `dict`:

```
del self.storage[instance]
```

For external storage, it simply deletes from the instance dictionary:

```
del instance.__dict__[name]
```

If the descriptor doesn't represent a stored value, do nothing. There's truly very little variation in what `__delete__()` methods look like, other than the additional functionality a descriptor may have.

When a descriptor wraps a function as a decorator, using `__delete__()` should probably not delete that function; that would revoke its access to all instances. Instead, either do nothing, raise an exception, or skip the implementation entirely.

Summary

We've seen that `__delete__()` is a pretty simple method to implement, but deciding whether to actually implement it or not can be a difficult decision. In the end, though, it will be used so little that implementing it can probably be put off until it's needed. The default behavior of raising an exception due to lack of implementation should get you by until then.

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_9

CHAPTER 10

Descriptors are Classes Too

It's time for some more advanced stuff with descriptors. Actually, it's not really advanced, since it's stuff that pertains to all classes. There won't be a very in-depth look at much in this chapter, just a reminder that features normally available to classes are available to descriptors as well.

Inheritance

Descriptors can inherit and be inherited from other classes (which will generally be other descriptors or else descriptor utilities/helpers). Using inheritance, descriptors can be built using pre-built mix-ins and template classes that already implement the base functionality wanted for storing the attribute. In fact, a suite of these are discussed in the next chapter and fully provided in the library. Just as an example, a base class can be created that takes care of the minor details of using on-descriptor storage that the derived specialization can delegate to. Again, there's more about this idea in the next chapter with full code examples in the library.

More Methods

A descriptor can have more methods than just that of the descriptor protocol and `__init__()`. This was shown with secret-set descriptors which have a back door method, like `set()`.

Externally-used methods like that should be limited, since access to these methods should be limited too, but using internally-used 'protected' and 'private' methods (methods whose names start with _ and __, respectively) that are used only within the class are fair game. Also, implementing `__str__()` and `__repr__()` is a good idea too. It's rarely useful or necessary to implement `__eq__()` or `__hash__()`, as descriptors themselves are rarely compared or stored in a hashed collection as a key.

As usual, the internally-used methods' main use is to keep the code of other methods cleaner, splitting up the work of a method into smaller refined chunks.

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_10

Optional/Default Parameters

Just like in the forced-set descriptors, optional/default parameters can be added to the protocol methods. Since users providing alternative arguments still requires them to get the descriptor object and call the protocol methods directly, this should be limited, just like additional externally-used methods.

Additionally, it should be limited for the sake of composition and inheritance. If the class providing the optional parameter gets wrapped or subclassed, the new class has to either know about the optional parameter or provide a `**kwargs` parameter and pass it down the line, as will be seen in much of the provided code in the library.

■ **Note** I have an additional reason to avoid the optional parameters: it just feels wrong. The descriptor protocol has its set of parameters for the methods, and it feels wrong to try and add additional optional parameters.

Descriptors on Descriptors

Since descriptors are classes, descriptors can have descriptors on them too! Since most descriptors are useful for easing the calls of those descriptor attributes, this is rarely helpful since the descriptor objects themselves are almost never accessed from the outside, just their protocol methods.

If the descriptor on the descriptor is mostly providing nice, reusable logic that keeps the outer descriptor's implementation simpler, then there's definitely nothing wrong with using it.

Passing an Instance Around

No one ever said that a descriptor had to create a new instance for each and every class it was put on. An instance of a descriptor can be created outside of a class definition, then assigned to a class attribute of multiple classes.

This can possibly be a good idea to save a little bit of space when storing on the descriptor, since it will only have the overhead of a single dictionary instead of one per class. In fact, this works just fine in most cases. This should only be done with a little consideration, though, as there might be hidden effects. Plus it can also be confusing, especially due to its unusualness.

Descriptors Just Abstract Method Calls

Basically, a descriptor is just a simpler way to do certain method calls. Those method calls don't *have* to work in a `property`-ish way, getting and/or setting a certain value.

The `__get__()` descriptor method can essentially replace any method on a class that takes no parameters and returns an object. What's more, it doesn't even need to return anything, since not returning anything makes it return `None`. The `__set__()` descriptor method can be a replacement for any method that has a single parameter and doesn't return anything. The `__delete__()` method replaces methods with no parameters and doesn't return anything.

While a descriptor *can* be used in these ways, using a descriptor this way is very likely to be unintuitive to users of the descriptor, largely due to the fact that the syntax seems strange for many of those cases, especially in the case of `__delete__()`.

Summary

Anything that can be done with any other class can be done with a descriptor, including things not brought up here. Though much of it *can* be done without any real downsides, there is rarely a need for many of the features, but it doesn't hurt to keep all of this in mind when writing your descriptors.

CHAPTER 11

Reusing the Wheel

Whenever possible and sensible, one should try to avoid reinventing the wheel. This chapter goes over a set of classes to use as superclasses and strategies to help build new descriptors a little faster. Only barebones code is presented here; the full code examples are in the library.

Storage Solutions

The first code examples cover storage "strategies" (which I'm calling Solutions) that a descriptor can use for its storage. These strategies can be hard-coded into new descriptors or be passed into the descriptor's initializer to be chosen on a case-by-case basis. Only two basic strategies will be shown here; the rest can be found in the library.

```
class OnDescriptorStorageSolution:
    def __init__(self):
        self.storage = DescriptorStorage()

    def get(self, instance):
        return self.storage[instance]

    def set(self, instance, value):
        self.storage[instance] = value

    def delete(self, instance):
        del self.storage[instance]

class NameGetter:
    def __init__(self, name_lookup_strategy):
        self.lookup_strategy = name_lookup_strategy
        self.name = None

    def __call__(self, instance, descriptor):
        if self.name is None:
            self.name = self.lookup_strategy(instance, descriptor)
        return self.name
```

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9_11

```
class OnInstanceStorageSolution:
    def __init__(self, name_lookup_strategy):
        self.name_getter = NameGetter(name_lookup_strategy)

    def get(self, instance):
        return instance.__dict__[self.name_getter(instance, self)]

    def set(self, instance, value):
        instance.__dict__[self.name_getter(instance, self)] = value

    def delete(self, instance):
        del instance.__dict__[self.name_getter(instance, self)]
```

Clearly, these storage solutions are designed for per instance storage. This is due to two reasons: 1) per class storage is trivial and therefore doesn't need pre-built solutions; 2) per instance storage is much more common.

The `NameGetter` class and its use might be just a little confusing. As stated in the chapter about storage, the most difficult thing for storing on the instances is figuring out how to find the name of where to store, so the `OnInstanceStorageSolution` class takes in a `name_lookup_strategy`. This strategy is just a function that accepts `instance` and the descriptor and returns the name to store at. The strategy accepts those two parameters because those are the only pieces of information guaranteed that can be used for the lookup, and they're also required for doing lookup via `name_of()` as mentioned earlier in the book. If the name is already decided, the lookup strategy can simply be a lambda that takes `*args` and returns the decided name.

`NameGetter` isn't technically required to do the work necessary, but is used to cache the name after the name has been calculated. That way, the lookup method doesn't need to be called more than once; it's called once, then stored for quick returns on subsequent lookups.

Now that storage solutions have been shown, here are some example descriptors using or prepared to be supplied with a storage solution object (delete methods are elided for simplicity's sake).

```
class ExampleHardCodedStrategy:
    def __init__(self):
        self.storage = OnDescriptorStorageSolution()

    def __get__(self, instance, owner):
        # any pre-fetch logic
        value = self.storage.get(instance)
        # any post-fetch logic
        return value

    def __set__(self, instance, value):            # any pre-set logic
        self.storage.set(instance, value)
```

```
class ExampleOpenStrategy:
    def __init__(self, storage_solution):
        self.storage = storage_solution

    def __get__(self, instance, owner):
        # any pre-fetch logic
        value = self.storage.get(instance)
        # any post-fetch logic
        return value

    def __set__(self, instance, value):
        # any pre-set logic
        self.storage.set(instance, value)
```

These strategies could also be subclassed, making the strategy methods be more of template-called methods. For example:

```
class ExampleSubclassingStrategy(OnDescriptorStorageSolution):
    def __get__(self, instance, owner):
        # any pre-fetch logic
        value = self.get(instance) # calls the solution method on itself
        # any post-fetch logic
        return value

    def __set__(self, instance, value):
        # any pre-set logic
        self.set(instance, value) # same here
```

Using the storage solutions this way is a cleaner way of hard-coding the solution. Note that the code calls `self.get()` and `self.set()` and not `super().get()` and `super().set()`. Either one will work in this situation, but using self is more flexible. Leaving it as-is will allow for subclasses of this class to easily modify `set()` and `get()`.

Read-Only Solutions

Another utility class that can be built is a wrapper that can turn ANY other descriptor into a read-only descriptor. Here's an example using the set-once style.

```
class ReadOnly:
    def __init__(self, wrapped):
        self.wrapped = wrapped
        self.setInstances = set()

    def __set__(self, instance, value):
        if instance in setInstances:
            raise AttributeError("Cannot set new value on read-only
            property")
        else:
            setInstances.add(instance)
            self.wrapped.__set__(instance, value)

    ef __getattr__(self, item):
        # redirects any calls other than __set__ to the wrapped descriptor
           return getattr(self.wrapped, item)

def readOnly(deco):  # a decorator for wrapping other decorator descriptors
    def wrapper(func):
        return ReadOnly(deco(func))
    return wrapper
```

It even includes a decorator decorator for decorating descriptors being used as decorators. (Yo dawg; I heard you like decorators, so I put decorators in your decorators.) This isn't meant for wrapping just any decorators; it's only meant for wrapping decorators that produce descriptors. It's not likely to be used often, since most descriptors that are created from decorators are non-data descriptors, making the ReadOnly wrapping not very useful. But it doesn't hurt to have it anyway, just in case; especially after claiming it can wrap ANY other descriptor.

It can be noted that ReadOnly only implements the __set__() method of the descriptor protocol. This is because it's the only one that it covers. It uses __getattr__() in order to redirect calls to potential __get__() and __delete__() methods because it doesn't know which ones might be implemented.

ReadOnly doesn't have to be implemented as a wrapper. In fact, there is a strategy-based implementation, along with implementations of all the other read-only styles in the library.

Simple Unbound Attributes

Reusable code can be created for making the __get__() method return unbound attributes when instance isn't provided rather than returning the descriptor, too. It can be done via a wrapper class like the previous ReadOnly wrapper, via inheritance, or even a method decorator.

```
def binding(get):
    @wraps(get)
    def wrapper(self, instance, owner):
        if instance is None:
            return UnboundAttribute(self, owner)
        else:
            return get(self, instance, owner)
    return wrapper
```

This simple decorator can be used inside a descriptor easily:

```
class Descriptor:
    # other implementation details
    @binding
    def __get__(self, instance, owner):
        # implementation that assumes instance not None
```

By simply adding the call to the decorator, you can simplify the code you have to write, ignoring writing anything that has to deal with the possibility of `instance` being `None`, other than the decorator.

There's also an object decorator (i.e. a Gang of Four Decorator) version in the library so that any existing descriptor can be transformed to return unbound attributes. For example, if a user wants to use attribute binding with an existing descriptor that doesn't provide them, they could do something like this:

```
class MyClass:
    @Binding
    @property
    def myProp(self):
        # gets the actual property
```

Binding is a class that wraps entire Now property can be used with unbound attributes (with some caveats: if you continue on, defining a setter for `myProp`, myProp will be replaced with a *new* property object; only add the `@Binding` call to *last* method decorated with the `property`). With descriptors that aren't being used as decorators, it would actually look like this:

```
class MyClass:
    myProp = Binding(SomeDescriptor(...))
```

There is no version that works with inheritance since calling either of the decorators is easier than trying to create a superclass for the new descriptor to inherit from.

Summary

This is all the categories of helpful code provided in the library, but it is by no means the only actual pieces of code there. There are a ton of helpful pieces there to help you build your own descriptors, to mix and match certain pieces into a cohesive whole descriptor where you need to do minimal work to add in your core logic amongst the rest of it.

In this chapter, we've seen how reusable pieces can be made that can make implementing descriptors a little quicker and easier, as well as a little bit more standardized. As mentioned, all of these tools (and more) will be available in the library as well as on GitHub. Hopefully, they will help make your lives easier when you try to create your own descriptors.

CHAPTER 12

Other Uses of Descriptors in the World

Much of the usefulness of descriptors covered in this book was just using them as specialized properties. While this is one of the primary purposes of descriptors, it's not all that they can do, though even the more innovative uses still largely serve that purpose.

SQLAlchemy[4]

This is probably the best-known library that uses descriptors for some of its stronger powers. When using the declarative mapping style for data classes, the use of the `Column` descriptor allows users to specify all sorts of database metadata about the column that the attribute represents, including the data type, column name, whether it's a primary key, etc.

That Column class also has a ton of other methods that are used when creating queries around the data, such as the ordering methods, `__lt__()`, `__gt__()`, etc. and what table it's in.

Jigna

Jigna is a library that provides a kind of bridge between Python and JavaScript, allowing you to write Python code that creates web pages, including single-page applications. Using Trait descriptors, it can create two-way data bindings, generating AngularJS code that works with HTML pages.

The use is extremely innovative and powerful and it's all thanks to descriptors that it can be as easy to use as it is.

For more information, visit its GitHub repository[5] or check out the presentation the creator gave at EuroPython 2014[6].

Elk

Elk is Python library that is almost all descriptors, allowing for classes to be defined in a more strict fashion. Every attribute for instances is meant to be defined in the class with an `ElkAttribute` descriptor. Some examples of what can be done with `ElkAttributes` are:

- setting an attribute as required
- making lazy attributes
- delegate to the methods on the attribute
- make an attribute read-only
- automatic constructor creation

There are other features in the library, attempting to make the tedious parts of class definition a little easier, and they can be seen in its documentation[7].

Validators

This isn't a specific instance of what’s out there, but rather a well-known use for descriptors. For example, if an attribute needs to be a string that follows a certain regex pattern, a descriptor can be created that takes the regex, and every time a value is set into the descriptor, it validates that the new value fits the validation.

There are a bunch of different validation descriptors that can be written that allow a class to maintain its invariants.

Summary

Now you've seen some really cool uses for descriptors. Also, this is the end of the book, so I suggest you go out there and make your own really awesome descriptors. Go, and make the Python community an even more awesome place.

Bibliography

1. GitHub repo of Descriptor Tools
   https://github.com/sad2project/descriptor-tools
2. Python documentation on property
   http://tinyurl.com/ljsmxck
3. The Zipline Show about descriptors
   https://www.youtube.com/watch?v=xYBVjVEJtEg
4. SQLAchemy site http://www.sqlalchemy.org/
5. Jigna GitHub repo https://github.com/enthought/jigna
6. Jigna presentation at EuroPython 2014
   https://www.youtube.com/watch?v=KHSXq5jfv_4
7. Elk documentation http://frasertweedale.github.io/elk/

J. Zimmerman, *Python Descriptors*, DOI 10.1007/978-1-4842-2505-9

Index