# Descriptor HowTo Guide

Release 3.5.7

Guido van Rossum
and the Python development team

## Contents

**Author** Raymond Hettinger

**Contact** <python at rcn dot com>

Contents

- Descriptor HowTo Guide
    - Abstract
    - Definition and Introduction
    - Descriptor Protocol
    - Invoking Descriptors

# 1 Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.

# 2 Definition and Introduction

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are _ _get_ _(), _ _set_ _(), and _ _delete_ _(). If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, a.x has a lookup chain starting with a._ _dict_ _['x'], then type(a)._ _dict_ _['x'], and continuing through the base classes of type(a) excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and super(). They are used throughout Python itself to implement the new style classes introduced in version 2.2. Descriptors simplify the underlying C-code and offer a flexible set of new tools for everyday Python programs.

# 3 Descriptor Protocol

descr._ _get_ _(self, obj, type=None) --> value

descr._ _set_ _(self, obj, value) --> None

descr._ _delete_ _(self, obj) --> None

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines both _ _get_ _() and _ _set_ _(), it is considered a data descriptor. Descriptors that only define _ _get_ _() are called non-data descriptors (they are typically used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data

descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

To make a read-only data descriptor, define both __get__() and __set__() with the __set__() raising an AttributeError when called. Defining the __set__() method with an exception raising placeholder is enough to make it a data descriptor.

# 4 Invoking Descriptors

A descriptor can be called directly by its method name. For example, d.__get__(obj).

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, obj.d looks up d in the dictionary of obj. If d defines the method __get__(), then d.__get__(obj) is invoked according to the precedence rules listed below.

The details of invocation depend on whether obj is an object or a class.

For objects, the machinery is in object.__getattribute__() which transforms b.x into type(b).__dict__['x'].__get__(b, type(b)). The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to __getattr__() if provided. The full C implementation can be found in PyObject_GenericGetAttr() in Objects/object.c.

For classes, the machinery is in type.__getattribute__() which transforms B.x into B.__dict__['x'].__get__(None, B). In pure Python, it looks like:

```python
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

The important points to remember are:

- descriptors are invoked by the __getattribute__() method
- overriding __getattribute__() prevents automatic descriptor calls
- object.__getattribute__() and type.__getattribute__() make different calls to __get__().
- data descriptors always override instance dictionaries.
- non-data descriptors may be overridden by instance dictionaries.

The object returned by super() also has a custom __getattribute__() method for invoking descriptors. The call super(B, obj).m() searches obj.__class__.__mro__ for the base class A immediately following B and then returns A.__dict__['m'].__get__(obj, B). If not a descriptor, m is returned unchanged. If not in the dictionary, m reverts to a search using object.__getattribute__().

The implementation details are in super_getattro() in Objects/typeobject.c. and a pure Python equivalent can be found in Guido's Tutorial.

The details above show that the mechanism for descriptors is embedded in the __getattribute__() methods for object, type, and super(). Classes inherit this machinery when they derive from object or if they have a meta-class providing similar functionality. Likewise, classes can turn-off descriptor invocation by overriding __getattribute__().

# 5 Descriptor Example

The following code creates a class whose objects are data descriptors which print a message for each get or set. Overriding __getattribute__() is alternate approach that could do this for every attribute. However, this descriptor is useful for monitoring just a few chosen attributes:

```python
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound and unbound methods, static methods, and class methods are all based on the descriptor protocol.

# 6 Properties

Calling property() is a succinct way of building a data descriptor that triggers function calls upon access to an attribute. Its signature is:

```python
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

The documentation shows a typical use to define a managed attribute x:

```python
class C(object):
    def getx(self): return self.__x
```

4

```python
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

To see how property() is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```python
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

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

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

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

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

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

The property() builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through Cell('b10').value. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```python
class Cell(object):
    ...
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
```

```
        return self._value
    value = property(getvalue)
```

# 7 Functions and Methods

Python's object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Class dictionaries store methods as functions. In a class definition, methods are written using def and lambda, the usual tools for creating functions. The only difference from regular functions is that the first argument is reserved for the object instance. By Python convention, the instance reference is called self but may be called this or any other variable name.

To support method calls, functions include the __get__() method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound or unbound methods depending whether they are invoked from an object or a class. In pure python, it works like this:

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj)
```

Running the interpreter shows how the function descriptor works in practice:

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f          # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f          # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

The output suggests that bound and unbound methods are two different types. While they could have been implemented that way, the actual C implementation of PyMethod_Type in Objects/classobject.c is a single object with two different representations depending on whether the im_self field is set or is NULL (the C equivalent of None).

Likewise, the effects of calling a method object depend on the im_self field. If set (meaning bound), the original function (stored in the im_func field) is called as expected with the first argument set to the instance. If unbound, all of the arguments are passed unchanged to the original function. The actual C implementation of instancemethod_call() is only slightly more complex in that it includes some type checking.

# 8 Static Methods and Class Methods

Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

To recap, functions have a __get__() method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an obj.f(*args) call into f(obj, *args). Calling klass.f(*args) becomes f(*args).

This chart summarizes the binding and its two most useful variants:

| Transformation | Called from an Object | Called from a Class |
|---|---|---|
| function | f(obj, *args) | f(*args) |
| staticmethod | f(*args) | f(*args) |
| classmethod | f(type(obj), *args) | f(klass, *args) |

Static methods return the underlying function without changes. Calling either c.f or C.f is the equivalent of a direct lookup into object.__getattribute__(c, "f") or object.__getattribute__(C, "f"). As a result, the function becomes identically accessible from either an object or a class.

Good candidates for static methods are methods that do not reference the self variable.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are conceptually related but do not depend on the data. For instance, erf(x) is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: s.erf(1.5) --> .9332 or Sample.erf(1.5) --> .9332.

Since staticmethods return the underlying function with no changes, the example calls are unexciting:

```
>>> class E(object):
...     def f(x):
...         print(x)
...     f = staticmethod(f)
...
>>> print(E.f(3))
3
>>> print(E().f(3))
3
```

Using the non-data descriptor protocol, a pure Python version of staticmethod() would look like this:

```
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:

```
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f = classmethod(f)
...
>>> print(E.f(3))
('E', 3)
```

```
>>> print(E().f(3))
('E', 3)
```

This behavior is useful whenever the function only needs to have a class reference and does not care about any underlying data. One use for classmethods is to create alternate class constructors. In Python 2.3, the classmethod dict.fromkeys() creates a new dictionary from a list of keys. The pure Python equivalent is:

```python
class Dict(object):
    . . .
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)
```

Now a new dictionary of unique keys can be constructed like this:

```python
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of classmethod() would look like this:

```python
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```