Guía práctica de uso de los descriptores

Versión 3.8.20

Guido van Rossum
and the Python development team

septiembre 08, 2024

Python Software Foundation
Email: docs@python.org

Índice general

Autor Raymond Hettinger

Contacto <python at rcn dot com>

Contenido

1 Resumen

Definir los descriptores, resumir el protocolo y mostrar como los descriptores son llamados. Estudiar un descriptor personalizado y varios descriptores de Python incorporados, incluidas funciones, propiedades, métodos estáticos y métodos de clase. Mostrar como funciona cada uno proporcionando un equivalente puro de Python y un ejemplo de aplicación.

Aprender acerca de los descriptores no solo brinda acceso a un conjunto de herramientas mayor, sino que genera una comprensión más profunda de como funciona Python y una apreciación sobre la elegancia de su diseño.

2 Definición e introducción

En general, un descriptor es un atributo de objeto con «comportamiento vinculante», dónde el acceso al atributo ha sido reemplazado por métodos en el protocolo del descriptor. Esos métodos son `__get__()`, `__set__()` y `__delete__()`. Si alguno de esos métodos está definido para un objeto, se dice que es un descriptor.

El comportamiento predeterminado para el acceso a los atributos es obtener, establecer o eliminar el atributo del diccionario de un objeto. Por ejemplo, `a.x` tiene una cadena de búsqueda que comienza con `a.__dict__['x']`, luego `type(a).__dict__['x']` y continúa a través de las clases base de `type(a)` excluyendo metaclases. Si el valor buscado es un objeto que define uno de los métodos del descriptor, entonces Python puede anular el comportamiento predeterminado e invocar el método del descriptor en su lugar. El lugar donde ésto ocurre en la cadena de precedencia depende de qué métodos del descriptor fueron definidos.

Los descriptores son un potente protocolo de propósito general. Son el mecanismo detrás de las propiedades, métodos, métodos estáticos, métodos de clase y `super()`. Se utilizan en todo Python para implementar las clases de nuevo estilo introducidas en la versión 2.2. Los descriptores simplifican el código C subyacente y ofrecen un conjunto flexible de nuevas herramientas para los programas de Python cotidianos.

3 Protocolo descriptor

```
descr.__get__(self, obj, type=None) -> value

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

Eso es todo lo que hay que hacer. Si se define cualquiera de estos métodos, el objeto se considera un descriptor y puede anular el comportamiento predeterminado al ser buscado como un atributo.

Si un objeto define `__set__()` o `__delete__()`, se considera un descriptor de datos. Los descriptores que solo definen `__get__()` se denominan descriptores de no-datos (normalmente se utilizan para métodos, pero son posibles otros usos).

Los descriptores de datos y de no-datos difieren en como se calculan las anulaciones con respecto a las entradas en el diccionario de una instancia. Si el diccionario de una instancia tiene una entrada con el mismo nombre que un descriptor de datos, el descriptor de datos tiene prioridad. Si el diccionario de una instancia tiene una entrada con el mismo nombre que un descriptor de no-datos, la entrada del diccionario tiene prioridad.

Para crear un descriptor de datos de solo lectura, se define tanto `__get__()` como `__set__()` donde `__set__()` lanza un error `AttributeError` cuando es llamado. Definir el método `__set__()` de forma que lance una excepción genérica es suficiente para convertirlo en un descriptor de datos.

4 Invocar descriptores

Un descriptor puede ser llamado directamente mediante el nombre de su método. Por ejemplo `d.__get__(obj)`.

Alternativamente, es más común que un descriptor se invoque automáticamente al acceder a un atributo. Por ejemplo, `obj.d` busca `d` en el diccionario de `obj`. Si `d` define el método `__get__()`, entonces se invoca `d.__get__(obj)` de acuerdo con las reglas de precedencia que se enumeran a continuación.

Los detalles de la invocación dependen de si `obj` es un objeto o una clase.

Para los objetos, el mecanismo se encuentra en `object.__getattribute__()` que transforma `b.x` en `type(b).__dict__['x'].__get__(b, type(b))`. La implementación funciona a través de una cadena de precedencia que da a los descriptores de datos prioridad sobre las variables de instancia, a las variables de instancia prioridad sobre los descriptores de no-datos y asigna la prioridad más baja a `__getattr__()` si se proporciona. La implementación completa en C se puede encontrar en `PyObject_GenericGetAttr()` en Objects/object.c.

Para clases, el mecanismo se define en `type.__getattribute__()` que transforma `B.x` en `B.__dict__['x'].__get__(None, B)`. En Python puro, quedaría así:

```
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
```

Los puntos importantes a recordar son:

- los descriptores son invocados por el método `__getattribute__()`
- redefinir `__getattribute__()` evita las llamadas automáticas al descriptor
- `object.__getattribute__()` y `type.__getattribute__()` realizan diferentes llamadas a `__get__()`.
- los descriptores de datos siempre anulan los diccionarios de instancia.
- los descriptores de no-datos pueden ser reemplazados por los diccionarios de instancia.

El objeto devuelto por `super()` también tiene un método personalizado `__getattribute__()` para poder invocar descriptores. La búsqueda de atributo `super(B, obj).m` busca `obj.__class__.__mro__` para la clase base `A` inmediatamente después de `B` y luego devuelve `A.__dict__['m'].__get__(obj, B)`. Si no es un descriptor, se devuelve `m` sin cambios. Si no está en el diccionario, `m` revierte a una búsqueda usando `object.__getattribute__()`.

Los detalles de la implementación están en `super_getattro()` en Objects/typeobject.c y un equivalente puro de Python se puede encontrar en el Guido's Tutorial.

Los detalles anteriores muestran que el mecanismo para los descriptores está incrustado en los métodos `__getattribute__()` para `object`, `type` y `super()`. Las clases heredan este mecanismo cuando derivan de `object` o mediante una metaclase que proporcione funcionalidades similares. Del mismo modo, las clases pueden desactivar la invocación del descriptor redefiniendo `__getattribute__()`.

5 Ejemplo de descriptor

El siguiente código crea una clase cuyos objetos son descriptores de datos que imprimen un mensaje para cada lectura o escritura. Redefinir `__getattribute__()` es un enfoque alternativo que podría hacer esto para cada atributo. Sin embargo, este descriptor es útil para monitorizar solo algunos atributos elegidos:

```
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
```

El protocolo es simple y ofrece interesantes posibilidades. Varios casos de uso son tan comunes que se han empaquetado en llamadas a funciones individuales. Las propiedades, los métodos vinculados, los métodos estáticos y los métodos de clase se basan en el protocolo descriptor.

6 Propiedades

Llamar a `property()` es una forma sucinta de construir un descriptor de datos que desencadena llamadas a funciones al acceder a un atributo. Su firma es:

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

La documentación muestra un uso típico para definir un atributo administrado `x`:

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

Para ver cómo se implementa `property()` en términos del protocolo descriptor, aquí hay un equivalente puro de 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__)
```

La función incorporada `property()` es de ayuda cuando una interfaz de usuario ha otorgado acceso a atributos y luego los cambios posteriores requieren la intervención de un método.

Por ejemplo, una clase de hoja de cálculo puede otorgar acceso al valor de una celda a través de `Cell('b10').value`. Las mejoras posteriores del programa requieren que la celda se vuelva a calcular en cada acceso; sin embargo, el programador no quiere afectar al código de cliente existente que accede al atributo directamente. La solución es envolver el acceso al valor del atributo en un descriptor de datos mediante una propiedad:

```
class Cell(object):
    . . .
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
    value = property(getvalue)
```

7 Funciones y métodos

Las características orientadas a objetos de Python se basan en un entorno basado en funciones. Usando descriptores de no-datos, ambas se combinan perfectamente.

Los diccionarios de clase almacenan los métodos como funciones. En una definición de clase, los métodos se escriben usando `def` o `lambda`, las herramientas habituales para crear funciones. Los métodos solo difieren de las funciones regulares en que el primer argumento está reservado para la instancia del objeto. Por convención en Python, la referencia de instancia se llama *self* pero puede llamarse *this* o cualquier otro nombre de variable.

Para admitir llamadas a métodos, las funciones incluyen el método `__get__()` para vincular métodos durante el acceso a atributos. Esto significa que todas las funciones son descriptores de no-datos que devuelven métodos enlazados cuando se invocan desde un objeto. En Python puro, funciona así:

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

Ejecutar el intérprete muestra como funciona el descriptor de función en la práctica:

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()

# Access through the class dictionary does not invoke __get__.
# It just returns the underlying function object.
>>> D.__dict__['f']
<function D.f at 0x00C45070>

# Dotted access from a class calls __get__() which just returns
# the underlying function unchanged.
>>> D.f
<function D.f at 0x00C45070>

# The function has a __qualname__ attribute to support introspection
>>> D.f.__qualname__
'D.f'

# Dotted access from an instance calls __get__() which returns the
# function wrapped in a bound method object
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>

# Internally, the bound method stores the underlying function and
# the bound instance.
>>> d.f.__func__
<function D.f at 0x1012e5ae8>
>>> d.f.__self__
<__main__.D object at 0x1012e1f98>
```

8 Métodos estáticos y métodos de clase

Los descriptores de no-datos proporcionan un mecanismo simple para variaciones de los patrones habituales para vincular funciones en métodos.

En resumen, las funciones tienen un método `__get__()` para que se puedan convertir en un método cuando se accede a ellas como atributos. El descriptor de no-datos transforma una llamada a `obj.f(*args)` en `f(obj, *args)`. Llamar a `klass.f(*args)` se convierte en `f(*args)`.

Este cuadro resume el enlace (*binding*) y sus dos variantes más útiles:

Transformación	Llamado desde un objeto	Llamado desde una clase
función	f(obj, *args)	f(*args)
método estático	f(*args)	f(*args)
método de clase	f(type(obj), *args)	f(klass, *args)

Los métodos estáticos devuelven la función subyacente sin cambios. Llamar a `c.f` o `C.f` es equivalente a una búsqueda directa en `object.__getattribute__(c, "f")` o en `object.__getattribute__(C, "f")`. Como resultado, la función se vuelve idénticamente accesible desde un objeto o una clase.

Buenos candidatos para ser métodos estáticos son los métodos que no hacen referencia a la variable `self`.

Por ejemplo, un paquete de estadística puede incluir una clase contenedora para datos experimentales. La clase proporciona métodos normales para calcular el promedio, la media, la mediana y otras estadísticas descriptivas que dependen de los datos. Sin embargo, puede haber funciones útiles que están relacionadas conceptualmente pero que no dependen de los datos. Por ejemplo, `erf(x)` es una práctica rutinaria de conversión que surge en el trabajo estadístico pero que no depende directamente de un conjunto de datos en particular. Se puede llamar desde un objeto o la clase: `s.erf(1.5) --> .9332` o `Sample.erf(1.5) --> .9332`.

Dado que los métodos estáticos devuelven la función subyacente sin cambios, las llamadas de ejemplo carecen de interés:

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

Usando el protocolo descriptor de no-datos, una versión pura de Python de `staticmethod()` se vería así:

```
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
```

A diferencia de los métodos estáticos, los métodos de clase anteponen la referencia de clase a la lista de argumentos antes de llamar a la función. Este formato es el mismo si quien llama es un objeto o una clase:

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

(continué en la próxima página)

(proviene de la página anterior)

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

Este comportamiento es útil siempre que la función solo necesite tener una referencia de clase y no preocuparse por los datos subyacentes. Un uso de los métodos de clase es crear constructores de clase alternativos. En Python 2.3, el método de clase `dict.fromkeys()` crea un nuevo diccionario a partir de una lista de claves. El equivalente puro de Python es:

```
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)
```

Ahora se puede construir un nuevo diccionario de claves únicas así:

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

Usando el protocolo descriptor de no-datos, una implementación pura en Python de `classmethod()` se vería así:

```
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
```