
デスクリプタ HowTo ガイド

リリース 3.8.20

Guido van Rossum
and the Python development team

9月 08, 2024

目次

1	概要	2
2	定義と導入	2
3	デスクリプタプロトコル	3
4	デスクリプタの呼び出し	3
5	デスクリプタの例	4
6	プロパティ	5
7	関数とメソッド	7
8	静的メソッドとクラスメソッド	8

著者 Raymond Hettinger

問い合わせ先 <python at rcn dot com>

目次

- デスクリプタ *HowTo* ガイド
 - 概要

- 定義と導入
- デスクリプタプロトコル
- デスクリプタの呼び出し
- デスクリプタの例
- プロパティ
- 関数とメソッド
- 静的メソッドとクラスメソッド

1 概要

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.

デスクリプタについて学ぶことにより、新しいツールセットが使えるようになるだけでなく、Python の仕組みや、洗練された設計のアプリケーションについてのより深い理解が得られます。

2 定義と導入

一般に、デスクリプタは ” 束縛動作 (binding behavior) ” をもつオブジェクト属性で、その属性アクセスが、デスクリプタプロトコルのメソッドによってオーバーライドされたものです。このメソッドは、`__get__()`、`__set__()`、および `__delete__()` です。これらのメソッドのいずれかが、オブジェクトに定義されていれば、それはデスクリプタと呼ばれます。

属性アクセスのデフォルトの振る舞いは、オブジェクトの辞書の属性の取得、設定、削除です。例えば `a.x` は、まず `a.__dict__['x']`、それから `type(a).__dict__['x']`、さらに `type(a)` のメタクラスを除く基底クラスへと続くというように探索が連鎖します。見つかった値が、デスクリプタメソッドのいずれかを定義しているオブジェクトなら、Python はそのデフォルトの振る舞いをオーバーライドし、代わりにデスクリプタメソッドを呼び出します。これがどの連鎖順位で行われるかは、どのデスクリプタメソッドが定義されているかに依ります。

デスクリプタは、強力な、多目的のプロトコルです。これはプロパティ、メソッド、静的メソッド、クラスメソッド、そして `super()` の背後にある機構です。これはバージョン 2.2 で導入された新スタイルクラスを実装するために、Python のいたるところで使われています。デスクリプタは、基幹にある C コードを簡潔にし、毎日の Python プログラムに、柔軟な新しいツール群を提供します。

3 デスクリプタプロトコル

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

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

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

これで全てです。これらのメソッドのいずれかを定義すれば、オブジェクトはデスクリプタとみなされ、探索された際のデフォルトの振る舞いをオーバーライドできます。

If an object defines `__set__()` or `__delete__()`, 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).

データデスクリプタと非データデスクリプタでは、オーバーライドがインスタンスの辞書のエントリに関してどのように計算されるかが異なります。インスタンスの辞書に、データデスクリプタと同名の項目があれば、データデスクリプタの方が優先されます。インスタンスの辞書に、非データデスクリプタと同名の項目があれば、辞書の項目の方が優先されます。

読み出し専用のデータデスクリプタを作るには、`__get__()` と `__set__()` の両方を定義し、`__set__()` が呼び出されたときに `AttributeError` が送出されるようにしてください。例外を送出する `__set__()` メソッドをブレースホルダとして定義すれば、データデスクリプタにするのに十分です。

4 デスクリプタの呼び出し

デスクリプタは、メソッド名で直接呼ぶことも出来ます。例えば、`d.__get__(obj)` です。

または、一般的に、デスクリプタは属性アクセスから自動的に呼び出されます。例えば、`obj.d` は `obj` の辞書から `d` を探索します。`d` がメソッド `__get__()` を定義していたら、以下に列挙する優先順位に従って、`d.__get__(obj)` が呼び出されます。

呼び出しの詳細は、`obj` がオブジェクトかクラスかに依ります。

オブジェクトでは、その機構は `b.x` を `type(b).__dict__['x'].__get__(b, type(b))` に変換する `object.__getattr__()` にあります。データデスクリプタの優先度はインスタンス変数より高く、インスタンス変数の優先度は非データデスクリプタより高く、(提供されていれば) `__getattr__()` の優先度が最も低いように実装されています。完全な C 実装は、`Objects/object.c` の `PyObject_GenericGetAttr()` で見つかります。

クラスでは、その機構は `B.x` を `B.__dict__['x'].__get__(None, B)` に変換する `type.__getattr__()` にあります。pure Python では、このようになります:

```
def __getattr__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
```

(次のページに続く)

```

v = object.__getattribute__(self, key)
if hasattr(v, '__get__'):
    return v.__get__(None, self)
return v

```

憶えておくべき重要な点は:

- デスクリプタは `__getattribute__()` メソッドによって呼び出される
- `__getattribute__()` をオーバーライドすると、自動的なデスクリプタの呼び出しが行われなくなる
- `object.__getattribute__()` と `type.__getattribute__()` では、`__get__()` の呼び出しが異なる。
- データデスクリプタは、必ずインスタンス辞書をオーバーライドする。
- 非データデスクリプタは、インスタンス辞書にオーバーライドされることがある。

The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The attribute lookup `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__()`.

実装の詳細は、[Objects/typeobject.c](#) の `super_getattro()` と、[Guido's Tutorial](#) にある等価な pure Python 版を参照してください。

上述の詳細は、デスクリプタの機構が、`__getattribute__()` メソッドに埋めこまれ、`object`, `type`, そして `super()` に使われているということを表しています。クラスは、`object` から導出されたとき、または、同じような機能を提供するメタクラスをもつとき、この機構を継承します。同様に、`__getattribute__()` をオーバーライドすることで、デスクリプタの呼び出しを無効にできます。

5 デスクリプタの例

以下のコードは、オブジェクトが取得と設定のたびにメッセージを表示するデータデスクリプタであるようなクラスを生成します。代わりに `__getattribute__()` をオーバーライドすると、全ての属性に対してこれができます。しかし、このデスクリプタは、少数の選ばれた属性を監視するのに便利です:

```

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

```

(次のページに続く)

```

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

```

このプロトコルは単純ですが、ワクワクする可能性も秘めています。ユースケースの中には、あまりに一般的なので個別の関数の呼び出しにまとめられたものもあります。プロパティ、束縛のメソッド、静的メソッド、そしてクラスメソッドは、全てデスクリプタプロトコルに基づいています。

6 プロパティ

`property()` を呼び出すことで、属性へアクセスすると関数の呼び出しを引き起こす、データデスクリプタを簡潔に組み立てられます。シグネチャはこうです:

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

このドキュメントでは、管理された属性 `x` を定義する典型的な使用法を示します:

```

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.")

```

デスクリプタの見地から `property()` がどのように実装されているかを見るために、等価な Python 版をここに

挙げます:

```
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__)
```

組み込みの `property()` 関数は、ユーザインタフェースへの属性アクセスが与えられ、続く変更がメソッドの介入を要求するときに役立ちます。

例えば、スプレッドシートクラスが、`Cell('b10').value` でセルの値を取得できるとします。続く改良により、プログラムがアクセスの度にセルの再計算をすることを要求しました。しかしプログラマは、その属性に直接アクセスする既存のクライアントコードに影響を与えたくありません。この解決策は、`property` データデスク립タ内に値属性へのアクセスをラップすることです:

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

7 関数とメソッド

Python のオブジェクト指向機能は、関数に基づく環境の上に構築されています。非データデスクリプタを使って、この 2 つはシームレスに組み合わせられています。

クラス辞書は、メソッドを関数として保存します。クラス定義内で、メソッドは、関数を使うのに便利なツール、`def` か `lambda` を使って書かれます。メソッドの標準の関数との唯一の違いは、第一引数がオブジェクトインスタンスのために予約されていることです。Python の慣習では、このインスタンスの参照は `self` と呼ばれますが、`this` その他の好きな変数名で呼び出せます。

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 methods when they are invoked from an object. In pure Python, it works like this:

```
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)
```

インタプリタを起動すると、この関数デスクリプタが実際にどうはたらくかを見られます:

```
>>> 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
```

(次のページに続く)

```
# 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 静的メソッドとクラスメソッド

非データデスクリプタは、関数をメソッドに束縛する、各種の一般的なパターンに、単純な機構を提供します。

まとめると、関数は `__get__()` メソッドを持ち、属性としてアクセスされたとき、メソッドに変換されます。この非データデスクリプタは、`obj.f(*args)` の呼び出しを `f(obj, *args)` に変換します。`klass.f(*args)` を呼び出すと `f(*args)` になります。

このチャートは、束縛と、その 2 つの異なる便利な形をまとめています:

変換	オブジェクトから呼び出される	クラスから呼び出される
function	<code>f(obj, *args)</code>	<code>f(*args)</code>
静的メソッド	<code>f(*args)</code>	<code>f(*args)</code>
クラスメソッド	<code>f(type(obj), *args)</code>	<code>f(klass, *args)</code>

静的メソッドは、下にある関数をそのまま返します。`c.f` や `C.f` は、`object.__getattribute__(c, "f")` や `object.__getattribute__(C, "f")` を直接探索するのと同じです。結果として、関数はオブジェクトとクラスから同じようにアクセスできます。

静的メソッドにすると良いのは、`self` 変数への参照を持たないメソッドです。

例えば、統計パッケージに、実験データのコンテナがあるとします。そのクラスは、平均、メジアン、その他の、データに依る記述統計を計算する標準メソッドを提供します。しかし、概念上は関係があっても、データには依ら

ないような便利な関数もあります。例えば、`erf(x)` は統計上の便利な変換ルーチンですが、特定のデータセットに直接には依存しません。これは、オブジェクトからでもクラスからでも呼び出せます: `s.erf(1.5) --> .9332` または `Sample.erf(1.5) --> .9332`。

静的メソッドは下にある関数をそのまま返すので、呼び出しの例は面白くありません:

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

非データデスク립タプロトコルを使うと、pure Python 版の `staticmethod()` は以下ようになります:

```
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
```

静的メソッドとは違って、クラスメソッドは関数を呼び出す前にクラス参照を引数リストの先頭に加えます。このフォーマットは、呼び出し元がオブジェクトでもクラスでも同じです:

```
>>> 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)
```

この振る舞いは、関数がクラス参照のみを必要とし、下にあるデータを考慮しないときに便利です。クラスメソッドの使い方の一つは、代わりのクラスコンストラクタを作ることです。Python 2.3 では、クラスメソッド `dict.fromkeys()` は新しい辞書をキーのリストから生成します。等価な pure Python 版は:

```
class Dict(object):
    . . .
    def fromkeys(klass, iterable, value=None):
```

(次のページに続く)

```

    "Emulate dict_fromkeys() in Objects/dictobject.c"
    d = class()
    for key in iterable:
        d[key] = value
    return d
fromkeys = classmethod(fromkeys)

```

これで一意なキーを持つ新しい辞書が以下のように構成できます:

```

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

```

非データデスク립タプロトコルを使った、`classmethod()` の pure Python 版はこのようになります:

```

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

```