ディスクリプタ HowTo ガイド

リリース *2.7.17*

Guido van Rossum
and the Python development team

12 月 24, 2019

Python Software Foundation
Email: docs@python.org

目 次

著者 Raymond Hettinger

問い合わせ先 <python at rcn dot com>

目次

第1章　概要

ディスクリプタを定義し、プロトコルを要約し、ディスクリプタがどのように呼び出されるか示します。カスタムのディスクリプタや、関数、プロパティ、静的メソッド、クラスメソッドを含む、いくつかの組み込み Python ディスクリプタを考察します。等価な pure Python 版やサンプルアプリケーションを与えることにより、それぞれがどのように働くかを示します。

ディ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。

第2章　定義と導入

一般に、ディスクリプタは ”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、ディ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、それはディ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、ディ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、代わりにディ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、どのディスクリプタメソッドが定義されているかに依ります。なお、ディスクリプタは、新スタイルのオブジェクトまたはクラスにのみ呼び出されます (あるクラスは、それが `object` または `type` を継承していれば、新スタイルです)。

ディスクリプタは、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。ディスクリプタは、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。

第3章　ディ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、オブジェクトはディ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。

あるオブジェクトが `__get__()` と `__set__()` の両方を定義していたら、それはデータディスクリプタとみなされます。 `__get__()` だけを定義しているディスクリプタは、非データディスクリプタと呼ばれます (これらは典型的にはメソッドに使われますが、他の使い方も出来ます)。

データディスクリプタと非データディスクリプタでは、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、データディスクリプタと同名の項目があれば、データディスクリプタの方が優先されます。インスタンスの辞書に、非データディ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。

読み込み専用のデータディ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__()` メソッドをプレースホルダとして定義すれば、データディスクリプタにするのに十分です。

第4章　デスクリプタの呼び出し

ディスクリプタは、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、ディ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。どちらにしても、ディスクリプタは新スタイルのオブジェクトやクラスにのみ働きます。クラスは、それが `object` のサブクラスであるなら新スタイルです。

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.__getattribute__()` にあります。データディ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ィ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.__getattribute__()` にあります。 pure Python では、このようになります:

```
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
```

憶えておくべき重要な点は:

- ディスクリプタは `__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、自動的なディスクリプタの呼び出しが行われなくなる
- `__getattribute__()` は新スタイルのクラスとオブジェクトにのみ使える。
- `object.__getattribute__()` と `type.__getattribute__()` では、 `__get__()` の呼び出しが異なる。
- データディスクリプタは、必ずインスタンス辞書をオーバーライドする。
- 非データディスクリプタは、インスタンス辞書にオーバーライドされることがある。

`super()` が返したオブジェクトにはさらにディスクリプタを起動するカスタム `__getattribute__()` メソッドがあります。 `super(B, obj).m()` の呼び出しは、 `B` に続き基底クラス `A` を見つけるために `obj.__class__.__mro__` を探索し、 `A.__dict__['m'].__get__(obj, B)` を返します。ディスクリプタ

ではない場合、 m が変更されずに返されます。辞書になければ、 m は object.__getattribute__() を使用した探索に戻ります。

なお、Python 2.2 では、 m がデータディスクリプタなら、 super(B, obj).m() は __get__() を呼び出すだけです。 Python 2.3 では、旧スタイルクラスが呼び出されなければ、非データディスクリプタも呼び出されます。実装の詳細は、 Objects/typeobject.c 内の super_getattro() を参照してください。

上述の詳細は、ディ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、ディスクリプタの呼び出しを無効にできます。

第5章　ディスクリプタの例

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ィ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、このディスクリプタは、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

    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、束縛および非束縛のメソッド、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。

第6章　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、データディ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.")
```

ディ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__)
```

(次のページに続く)

(前のページからの続き)

```
    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 データディスクリプ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)
```

第7章　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。非データディスクリプタを使って、この 2 つはシームレスに組み合わされています。

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` や `lambda` を使って書かれます。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。

メソッドの呼び出しをサポートするために、関数の `__get__()` メソッドは属性アクセス時にメソッドを束縛します。これにより、すべての関数は、それが呼び出されたのがオブジェクトかクラスかによって、束縛か非束縛メソッドを返す非データデスクリプタになります。pure Python では、これはこのようにはたらきます:

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

インタプリタを起動すると、この関数ディ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()
>>> 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>>
```

この出力は、束縛メソッドと非束縛メソッドは 2 つの異なる型であることを示しています。これらは、異なる型として実装することも出来ますが、 Objects/classobject.c における `PyMethod_Type` の実際の C 実装は 1 つのオブジェクトで、 `im_self` が *NULL* (C での `None` と同等のもの) に設定されているかどうかで決まる 2 つの異なる表現を持ってるのです。

同様に、メソッドオブジェクトを呼び出すことの効果も、 `im_self` フィールドに依ります。設定されていれば (束縛を意味し)、期待通り (`im_func` フィールドに保存されている) 元の関数が、第一引数をインスタンスとして、呼び出されます。非束縛なら、すべての引数がそのまま元の関数に渡されます。`instancemethod_call()` の実際の C 実装は、型チェックがあるため、もう少しだけ複雑です。

第8章　静的メソッドとクラスメソッド

非データディスクリプタは、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。この非データディスクリプタは、 `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
関数	f(obj, *args)	f(*args)
静的メソッド	f(*args)	f(*args)
クラスメソッド	f(type(obj), *args)	f(klass, *args)

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

非データディスクリプタプロトコルを使うと、pure Python 版の `staticmethod()` は以下のようになります:

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

(次のページに続く)

(前のページからの続き)

```
    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):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        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}
```

非データディスクリプ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
```