
Descriptor HowTo Guide

リリース 3.11.13

Guido van Rossum and the Python development team

7月 07, 2025

目次

1	入門	4
1.1	シンプルな例: 定数を返すデスクリプタ	4
1.2	動的なルックアップ	5
1.3	管理された属性	5
1.4	名前のカスタマイズ	7
1.5	まとめ	8
2	実践的なサンプル	9
2.1	バリデータクラス	9
2.2	カスタムバリデータ	10
2.3	実践的なアプリケーション	11
3	技術的なチュートリアル	12
3.1	概要	12
3.2	定義と導入	12
3.3	デスクリプタプロトコル	13
3.4	デスクリプタ呼び出しの概要	13
3.5	インスタンスからの呼び出し	13
3.6	クラスからの呼び出し	15
3.7	super からの呼び出し	15
3.8	呼び出しロジックのまとめ	15
3.9	自動的の名前の伝達	16
3.10	ORM のサンプル	16
4	ピュア Python の等価実装	17
4.1	プロパティ	17
4.2	関数とメソッド	19

4.3	メソッドの種類	21
4.4	静的メソッド	21
4.5	クラスメソッド	22
4.6	メンバーオブジェクトと <code>__slots__</code>	24

著者

Raymond Hettinger

問い合わせ先

<python at rcn dot com>

目次

- *Descriptor HowTo Guide*
 - 入門
 - * シンプルな例: 定数を返すデスクリプタ
 - * 動的なルックアップ
 - * 管理された属性
 - * 名前のカスタマイズ
 - * まとめ
 - 実践的なサンプル
 - * バリデータクラス
 - * カスタムバリデータ
 - * 実践的なアプリケーション
 - 技術的なチュートリアル
 - * 概要
 - * 定義と導入
 - * デスクリプタプロトコル
 - * デスクリプタ呼び出しの概要
 - * インスタンスからの呼び出し

- * クラスからの呼び出し
- * *super* からの呼び出し
- * 呼び出しロジックのまとめ
- * 自動的の名前の伝達
- * *ORM* のサンプル
- ピュア *Python* の等価実装
 - * プロパティ
 - * 関数とメソッド
 - * メソッドの種類
 - * 静的メソッド
 - * クラスメソッド
 - * メンバーオブジェクトと `__slots__`

デスクリプタを使うと、オブジェクトの属性検索、保存、削除をカスタマイズできます。

本ガイドは4つの大項目から構成されています。

- 1) 最初の「入門」では基本の紹介をして、シンプルな例から少しずつ概念を説明していきます。もしデスクリプタに触れるのが初めての場合はここから学んでいきましょう。
- 2) 2つ目のセクションでは完全で実践的なデスクリプタのサンプルを紹介します。もしデスクリプタの基本を理解しているのであれば、ここから読み始めても良いでしょう。
- 3) 3番目のセクションはより技巧的なチュートリアルを通じて、デスクリプタがどのようなメカニズムで動作しているのかを紹介します。多くの人はこのレベルの理解は不要でしょう。
- 4) 最後のセクションは、Cで書かれた組み込みのデスクリプタをピュア Python で実現する方法を紹介します。このセクションがどのように関数が束縛メソッドになるのか、`classmethod()` や `staticmethod()`, `property()`, `__slots__` といった一般的なツールの実装がどのようなになっているのか興味を持った方読んでください。

1 入門

この入門では、なるべく基礎的なところから開始し、新しい要素を一つずつ紹介していきます。

1.1 シンプルな例: 定数を返すデスクリプタ

The `Ten` class is a descriptor whose `__get__()` method always returns the constant 10:

```
class Ten:
    def __get__(self, obj, objtype=None):
        return 10
```

このデスクリプタを使うには他のクラスのクラス変数として保存します:

```
class A:
    x = 5                # Regular class attribute
    y = Ten()            # Descriptor instance
```

インタラクティブセッションを使って、通常の属性ルックアップと、デスクリプタのルックアップの違いを見てみましょう。

```
>>> a = A()            # Make an instance of class A
>>> a.x                # Normal attribute lookup
5
>>> a.y                # Descriptor lookup
10
```

`a.x` 属性ルックアップではドット演算子がクラス辞書の中から `'x': 5` を見つけます。`a.y` ルックアップではドット演算子は `__get__` メソッドを持つデスクリプタインスタンスを取得します。そのメソッドを呼び出すと、10 を返します。

10 はクラスの辞書にも、インスタンスの辞書にも格納されていません。その代わり、10 はオンデマンドに計算されます。

このサンプルは、シンプルなデスクリプタがどのように動作するかを説明するためのものですが、実用的ではありません。定数を返す場合、通常の属性アクセスの方が良いでしょう。

次のセクションでは、より実践的な動的なルックアップを作成します。

1.2 動的なルックアップ

単に定数を返すよりも、なにか処理を実行するデスクリプタの方が面白いでしょう。

```
import os

class DirectorySize:

    def __get__(self, obj, objtype=None):
        return len(os.listdir(obj.dirname))

class Directory:

    size = DirectorySize()           # Descriptor instance

    def __init__(self, dirname):
        self.dirname = dirname      # Regular instance attribute
```

インタラクティブセッションで見たように、この探索は動的です。実行ごとに異なる計算を行い、毎回答えが更新されます。

```
>>> s = Directory('songs')
>>> g = Directory('games')
>>> s.size           # The songs directory has twenty files
20
>>> g.size           # The games directory has three files
3
>>> os.remove('games/chess')    # Delete a game
>>> g.size           # File count is automatically updated
2
```

Besides showing how descriptors can run computations, this example also reveals the purpose of the parameters to `__get__()`. The *self* parameter is *size*, an instance of *DirectorySize*. The *obj* parameter is either *g* or *s*, an instance of *Directory*. It is the *obj* parameter that lets the `__get__()` method learn the target directory. The *objtype* parameter is the class *Directory*.

1.3 管理された属性

A popular use for descriptors is managing access to instance data. The descriptor is assigned to a public attribute in the class dictionary while the actual data is stored as a private attribute in the instance dictionary. The descriptor's `__get__()` and `__set__()` methods are triggered when the public attribute is accessed.

このサンプルでは *age* はパブリックな属性で、*_age* はプライベートな属性です。パブリックな属性がアクセスされると、デスクリプタはルックアップや更新のログ出力をします。

```

import logging

logging.basicConfig(level=logging.INFO)

class LoggedAgeAccess:

    def __get__(self, obj, objtype=None):
        value = obj._age
        logging.info('Accessing %r giving %r', 'age', value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', 'age', value)
        obj._age = value

class Person:

    age = LoggedAgeAccess()           # Descriptor instance

    def __init__(self, name, age):
        self.name = name             # Regular instance attribute
        self.age = age               # Calls __set__()

    def birthday(self):
        self.age += 1               # Calls both __get__() and __set__()

```

インタラクティブセッションの結果で見てわかるように、管理された属性の `age` のログ出力が行われますが、通常の属性の `name` はログ出力されません。

```

>>> mary = Person('Mary M', 30)           # The initial age update is logged
INFO:root:Updating 'age' to 30
>>> dave = Person('David D', 40)
INFO:root:Updating 'age' to 40

>>> vars(mary)                             # The actual data is in a private attribute
{'name': 'Mary M', '_age': 30}
>>> vars(dave)
{'name': 'David D', '_age': 40}

>>> mary.age                               # Access the data and log the lookup
INFO:root:Accessing 'age' giving 30
30
>>> mary.birthday()                        # Updates are logged as well
INFO:root:Accessing 'age' giving 30
INFO:root:Updating 'age' to 31

>>> dave.name                             # Regular attribute lookup isn't logged
'David D'

```

(次のページに続く)

(前のページからの続き)

```
>>> dave.age                                     # Only the managed attribute is logged
INFO:root:Accessing 'age' giving 40
40
```

このサンプル実装には大きな問題があります。プライベートな名前の `_age` が `LoggedAgeAccess` クラスの中でハードコードされています。そのため、どのインスタンスも1つしかこのログ出力される属性が作れず、名前を変更できません。次のサンプルでこの問題を解決していきます。

1.4 名前のカスタマイズ

クラスがデスクリプタを使うときに、クラスはそれぞれのデスクリプタにどのような変数名を使うかを通知します。

In this example, the `Person` class has two descriptor instances, `name` and `age`. When the `Person` class is defined, it makes a callback to `__set_name__()` in `LoggedAccess` so that the field names can be recorded, giving each descriptor its own `public_name` and `private_name`:

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r', self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', self.public_name, value)
        setattr(obj, self.private_name, value)

class Person:

    name = LoggedAccess()           # First descriptor instance
    age = LoggedAccess()           # Second descriptor instance

    def __init__(self, name, age):
        self.name = name           # Calls the first descriptor
        self.age = age             # Calls the second descriptor
```

(次のページに続く)

```
def birthday(self):
    self.age += 1
```

An interactive session shows that the `Person` class has called `__set_name__()` so that the field names would be recorded. Here we call `vars()` to look up the descriptor without triggering it:

```
>>> vars(vars(Person)['name'])
{'public_name': 'name', 'private_name': '_name'}
>>> vars(vars(Person)['age'])
{'public_name': 'age', 'private_name': '_age'}
```

新しいクラスは `name` と `age` の両方でログを出力します。

```
>>> pete = Person('Peter P', 10)
INFO:root:Updating 'name' to 'Peter P'
INFO:root:Updating 'age' to 10
>>> kate = Person('Catherine C', 20)
INFO:root:Updating 'name' to 'Catherine C'
INFO:root:Updating 'age' to 20
```

2つの `Person` のインスタンスは、プライベート名しか保持していません。

```
>>> vars(pete)
{'_name': 'Peter P', '_age': 10}
>>> vars(kate)
{'_name': 'Catherine C', '_age': 20}
```

1.5 まとめ

A descriptor is what we call any object that defines `__get__()`, `__set__()`, or `__delete__()`.

Optionally, descriptors can have a `__set_name__()` method. This is only used in cases where a descriptor needs to know either the class where it was created or the name of class variable it was assigned to. (This method, if present, is called even if the class is not a descriptor.)

デスクリプタは属性のルックアップ中に、ドット演算子によって呼び出されます。`vars(some_class)[descriptor_name]` という書き方によって間接的にデスクリプタがアクセスされると、デスクリプタを実行することなく、デスクリプタのインスタンスが返されます。

デスクリプタはクラス変数として利用したときにだけ動作します。インスタンスに設定しても効果はありません。

デスクリプタは、クラス変数として格納されているオブジェクトが属性ルックアップ中に発生することを制御できるように、フックを提供することが主なモチベーションとなっています。

伝統的には、呼び出し側のクラスが、ルックアップ中に発生することを制御してきましたが、デスクリプタはその関係を逆転させ、検索されているデータが問題に介入できるようにします。

デスクリプタは、Python のさまざまなところで利用されています。これにより、単なる関数は、インスタンスに束縛されたメソッドになります。`classmethod()`、`staticmethod()`、`property()`、`functools.cached_property()` といったツールもすべて、デスクリプタとして実装されています。

2 実践的なサンプル

このサンプルでは、発見が極めて難しいデータ破損バグを特定するのに利用可能な、実践的でパワフルなツールを紹介します。

2.1 バリデータクラス

バリデータは管理された属性アクセスのためのデスクリプタです。データを格納する前に、新しい値が型が適合しているか、値の範囲制限に適合しているかを検証します。もし制限に合わなければ、例外を送出し、データが不正な状態になるのを防ぎます。

This `Validator` class is both an abstract base class and a managed attribute descriptor:

```
from abc import ABC, abstractmethod

class Validator(ABC):

    def __set_name__(self, owner, name):
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        return getattr(obj, self.private_name)

    def __set__(self, obj, value):
        self.validate(value)
        setattr(obj, self.private_name, value)

    @abstractmethod
    def validate(self, value):
        pass
```

Custom validators need to inherit from `Validator` and must supply a `validate()` method to test various restrictions as needed.

2.2 カスタムバリデータ

ここでは3つの実践的なバリデーションのユーティリティのサンプルを紹介します。

- 1) `OneOf` verifies that a value is one of a restricted set of options.
- 2) `Number` verifies that a value is either an `int` or `float`. Optionally, it verifies that a value is between a given minimum or maximum.
- 3) `String` verifies that a value is a `str`. Optionally, it validates a given minimum or maximum length. It can validate a user-defined `predicate` as well.

```
class OneOf(Validator):

    def __init__(self, *options):
        self.options = set(options)

    def validate(self, value):
        if value not in self.options:
            raise ValueError(f'Expected {value!r} to be one of {self.options!r}')

class Number(Validator):

    def __init__(self, minvalue=None, maxvalue=None):
        self.minvalue = minvalue
        self.maxvalue = maxvalue

    def validate(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError(f'Expected {value!r} to be an int or float')
        if self.minvalue is not None and value < self.minvalue:
            raise ValueError(
                f'Expected {value!r} to be at least {self.minvalue!r}'
            )
        if self.maxvalue is not None and value > self.maxvalue:
            raise ValueError(
                f'Expected {value!r} to be no more than {self.maxvalue!r}'
            )

class String(Validator):

    def __init__(self, minsize=None, maxsize=None, predicate=None):
        self.minsize = minsize
        self.maxsize = maxsize
        self.predicate = predicate

    def validate(self, value):
        if not isinstance(value, str):
```

(次のページに続く)

(前のページからの続き)

```
        raise TypeError(f'Expected {value!r} to be an str')
    if self.minsize is not None and len(value) < self.minsize:
        raise ValueError(
            f'Expected {value!r} to be no smaller than {self.minsize!r}'
        )
    if self.maxsize is not None and len(value) > self.maxsize:
        raise ValueError(
            f'Expected {value!r} to be no bigger than {self.maxsize!r}'
        )
    if self.predicate is not None and not self.predicate(value):
        raise ValueError(
            f'Expected {self.predicate} to be true for {value!r}'
        )
```

2.3 実践的なアプリケーション

次のコードは実際のクラスでデータバリデータを使う方法のサンプルです。

```
class Component:

    name = String(minsize=3, maxsize=10, predicate=str.isupper)
    kind = OneOf('wood', 'metal', 'plastic')
    quantity = Number(minvalue=0)

    def __init__(self, name, kind, quantity):
        self.name = name
        self.kind = kind
        self.quantity = quantity
```

これらのデスクリプタは、不正インスタンスが作成されるのを防ぎます。

```
>>> Component('Widget', 'metal', 5)           # Blocked: 'Widget' is not all uppercase
Traceback (most recent call last):
...
ValueError: Expected <method 'isupper' of 'str' objects> to be true for 'Widget'

>>> Component('WIDGET', 'metle', 5)           # Blocked: 'metle' is misspelled
Traceback (most recent call last):
...
ValueError: Expected 'metle' to be one of {'metal', 'plastic', 'wood'}

>>> Component('WIDGET', 'metal', -5)          # Blocked: -5 is negative
Traceback (most recent call last):
...
ValueError: Expected -5 to be at least 0
```

(次のページに続く)

(前のページからの続き)

```
>>> Component('WIDGET', 'metal', 'V')    # Blocked: 'V' isn't a number
Traceback (most recent call last):
...
TypeError: Expected 'V' to be an int or float

>>> c = Component('WIDGET', 'metal', 5)  # Allowed: The inputs are valid
```

3 技術的なチュートリアル

これから先はデスクリプタのメカニズムや動作の詳細について、よりテクニカルなチュートリアルになります。

3.1 概要

デスクリプタの定義、プロトコルのサマリー、デスクリプタがどのように呼び出されるかの説明、OR マッパーがどのように動作するかのサンプルを提示します。

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 の仕組みについてのより深い理解が得られます。

3.2 定義と導入

In general, a descriptor is an attribute value that has one of the methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an attribute, it is said to be a descriptor.

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)` のメソッド探索順序に従ったさらなる探索へと連鎖します。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。

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. Descriptors simplify the underlying C code and offer a flexible set of new tools for everyday Python programs.

3.3 デスクリプタプロトコル

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

```
descr.__set__(self, obj, value)
```

```
descr.__delete__(self, obj)
```

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 often 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。

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.

3.4 デスクリプタ呼び出しの概要

`desc.__get__(obj)` や `desc.__get__(None, cls)` を使うとデスクリプタを直接呼び出すことができます。

しかし、属性アクセスの際に自動的に呼び出されるのが、一般的なデスクリプタの呼び出し方法です。

The expression `obj.x` looks up the attribute `x` in the chain of namespaces for `obj`. If the search finds a descriptor outside of the instance `__dict__`, its `__get__()` method is invoked according to the precedence rules listed below.

呼び出しの詳細は、`obj` がオブジェクトかクラスか、`super` のインスタンスかに依ります。

3.5 インスタンスからの呼び出し

Instance lookup scans through a chain of namespaces giving data descriptors the highest priority, followed by instance variables, then non-data descriptors, then class variables, and lastly `__getattr__()` if it is provided.

`a.x` に対するデスクリプタが見つかった場合、`desc.__get__(a, type(a))` という形式で呼び出されます。

ドットによるルックアップのロジックは `object.__getattribute__()` の中で行われます。ピュア Python で表した等価なロジックは次の通りです。

```

def find_name_in_mro(cls, name, default):
    "Emulate _PyType_Lookup() in Objects/typeobject.c"
    for base in cls.__mro__:
        if name in vars(base):
            return vars(base)[name]
    return default

def object_getattribute(obj, name):
    "Emulate PyObject_GenericGetAttr() in Objects/object.c"
    null = object()
    objtype = type(obj)
    cls_var = find_name_in_mro(objtype, name, null)
    descr_get = getattr(type(cls_var), '__get__', null)
    if descr_get is not null:
        if (hasattr(type(cls_var), '__set__')
            or hasattr(type(cls_var), '__delete__')):
            return descr_get(cls_var, obj, objtype)    # data descriptor
    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name]                        # instance variable
    if descr_get is not null:
        return descr_get(cls_var, obj, objtype)        # non-data descriptor
    if cls_var is not null:
        return cls_var                                # class variable
    raise AttributeError(name)

```

Note, there is no `__getattr__()` hook in the `__getattribute__()` code. That is why calling `__getattribute__()` directly or with `super().__getattribute__` will bypass `__getattr__()` entirely.

Instead, it is the dot operator and the `getattr()` function that are responsible for invoking `__getattr__()` whenever `__getattribute__()` raises an `AttributeError`. Their logic is encapsulated in a helper function:

```

def getattr_hook(obj, name):
    "Emulate slot_tp_getattr_hook() in Objects/typeobject.c"
    try:
        return obj.__getattribute__(name)
    except AttributeError:
        if not hasattr(type(obj), '__getattr__'):
            raise
        return type(obj).__getattr__(obj, name)    # __getattr__

```

3.6 クラスからの呼び出し

The logic for a dotted lookup such as `A.x` is in `type.__getattribute__()`. The steps are similar to those for `object.__getattribute__()` but the instance dictionary lookup is replaced by a search through the class's method resolution order.

`a.x` に対するデスクリプタが見つかった場合、`desc.__get__(None, A)` という形式で呼び出されます。

完全な C での実装は `Objects/typeobject.c` 中の `c:func:!type_getattro` と `_PyType_Lookup()` を参照してください。

3.7 super からの呼び出し

The logic for super's dotted lookup is in the `__getattribute__()` method for object returned by `super()`.

`super(A, obj).m` のようなドットを使ったルックアップは `obj.__class__.__mro__` を探索して、`A` の直前のクラス `B` をまず探し、`B.__dict__['m'].__get__(obj, A)` を返します。もしデスクリプタでなければ `m` を変更せずに返します。

完全な C での実装は `Objects/typeobject.c` 中の `super_getattro()` を参照してください。純粋な Python での同等なコードは [Guido's Tutorial](#) を参照してください。

3.8 呼び出しロジックのまとめ

The mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`.

憶えておくべき重要な点は:

- Descriptors are invoked by the `__getattribute__()` method.
- クラスはこの機構を `object`、`type`、`super()` から継承する
- Overriding `__getattribute__()` prevents automatic descriptor calls because all the descriptor logic is in that method.
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`. The first includes the instance and may include the class. The second puts in `None` for the instance and always includes the class.
- 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

3.9 自動的の名前の伝達

Sometimes it is desirable for a descriptor to know what class variable name it was assigned to. When a new class is created, the `type` metaclass scans the dictionary of the new class. If any of the entries are descriptors and if they define `__set_name__()`, that method is called with two arguments. The *owner* is the class where the descriptor is used, and the *name* is the class variable the descriptor was assigned to.

実装の詳細は `Objects/typeobject.c` の中の `type_new()` と `set_names()` を参照してください。

Since the update logic is in `type.__new__()`, notifications only take place at the time of class creation. If descriptors are added to the class afterwards, `__set_name__()` will need to be called manually.

3.10 ORM のサンプル

以下のコードは、オブジェクト関係マッピングを実装するのにデータディスクリプタを使う方法を示した、簡略化された骨格です。

基本的な考え方は、データが外部のデータベースに保存することです。Python インスタンスは、データベースのテーブルへのキーのみを保持します。ディスクリプタがルックアップや更新をします。

```
class Field:

    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM {owner.table} WHERE {owner.key}=?;'
        self.store = f'UPDATE {owner.table} SET {name}=? WHERE {owner.key}=?;'

    def __get__(self, obj, objtype=None):
        return conn.execute(self.fetch, [obj.key]).fetchone()[0]

    def __set__(self, obj, value):
        conn.execute(self.store, [value, obj.key])
        conn.commit()
```

We can use the `Field` class to define `models` that describe the schema for each table in a database:

```
class Movie:
    table = 'Movies'           # Table name
    key = 'title'              # Primary key
    director = Field()
    year = Field()

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

class Song:
```

(次のページに続く)

(前のページからの続き)

```
table = 'Music'
key = 'title'
artist = Field()
year = Field()
genre = Field()

def __init__(self, key):
    self.key = key
```

モデルを使用するにはまずデータベースに接続します:

```
>>> import sqlite3
>>> conn = sqlite3.connect('entertainment.db')
```

次のインタラクティブセッションでは、データベースからのデータの取得と、更新方法を示しています。

```
>>> Movie('Star Wars').director
'George Lucas'
>>> jaws = Movie('Jaws')
>>> f'Released in {jaws.year} by {jaws.director}'
'Released in 1975 by Steven Spielberg'

>>> Song('Country Roads').artist
'John Denver'

>>> Movie('Star Wars').director = 'J.J. Abrams'
>>> Movie('Star Wars').director
'J.J. Abrams'
```

4 ピュア Python の等価実装

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組み込みに追加さ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、クラスメソッド、`__slots__` は、全てデスク립タプロトコルに基づいています。

4.1 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립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
```

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:
    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.")

```

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

```

class Property:
    "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
        self._name = ''

    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError(f"property '{self._name}' has no getter")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError(f"property '{self._name}' has no setter")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError(f"property '{self._name}' has no deleter")
        self.fdel(obj)

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

    def setter(self, fset):
        prop = type(self)(self.fget, fset, self.fdel, self.__doc__)
        prop._name = self._name

```

(次のページに続く)

(前のページからの続き)

```
    return prop

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

組み込みの `property()` 関数は、ユーザインター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:
    ...

    @property
    def value(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
```

Either the built-in `property()` or our `Property()` equivalent would work in this example.

4.2 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。

クラス辞書に格納された関数は、実行時にメソッドに変換されます。メソッドと通常の間数はオブジェクトのインスタンスが他の引数よりも前に渡される点だけが異なります。監修ではインスタンスは `self` という名前が使われますが、`this` や他の変数名も使えます。

メソッドは、ほぼ次のコードと同等な `types.MethodType` を使えば手動で作れます。

```
class MethodType:
    "Emulate PyMethod_Type in Objects/classobject.c"

    def __init__(self, func, obj):
        self.__func__ = func
        self.__self__ = obj
```

(次のページに続く)

(前のページからの続き)

```
def __call__(self, *args, **kwargs):
    func = self.__func__
    obj = self.__self__
    return func(obj, *args, **kwargs)
```

To support automatic creation of methods, functions include the `__get__()` method for binding methods during attribute access. This means that functions are non-data descriptors that return bound methods during dotted lookup from an instance. Here's how it works:

```
class Function:
    ...

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

次のクラスを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働くかを見られます:

```
class D:
    def f(self, x):
        return x
```

この関数はイントロスペクションをサポートするために `qualified name` も持っています。

```
>>> D.f.__qualname__
'D.f'
```

Accessing the function through the class dictionary does not invoke `__get__()`. Instead, 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 interesting behavior occurs during dotted access from an instance. The dotted lookup calls `__get__()` which returns a bound method object:

```
>>> d = D()
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

内部では、束縛メソッドはオリジナルの関数と束縛されたインスタンスを保持しています。

```
>>> d.f.__func__
<function D.f at 0x00C45070>

>>> d.f.__self__
<__main__.D object at 0x00B18C90>
```

もし、通常のメソッドの `self` や、クラスメソッドの `cls` がどこから来るの疑問に思っている人がいるとしたら、これがその答えになります！

4.3 メソッドの種類

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。

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 `cls.f(*args)` becomes `f(*args)`.

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	<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(cls, *args)</code>

4.4 静的メソッド

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.__getattr__`(`c`, "f") や `object.__getattr__`(`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。

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`.

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:
    @staticmethod
    def f(x):
        return x * 10
```

```
>>> E.f(3)
30
>>> E().f(3)
30
```

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

```
import functools

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

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

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

    def __call__(self, *args, **kwds):
        return self.f(*args, **kwds)
```

The `functools.update_wrapper()` call adds a `__wrapped__` attribute that refers to the underlying function. Also it carries forward the attributes necessary to make the wrapper look like the wrapped function: `__name__`, `__qualname__`, `__doc__`, and `__annotations__`.

4.5 クラスメソッド

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 F:
    @classmethod
    def f(cls, x):
```

(次のページに続く)

(前のページからの続き)

```
return cls.__name__, x
```

```
>>> F.f(3)
('F', 3)
>>> F().f(3)
('F', 3)
```

この振る舞いは、関数がクラス参照のみを必要とし、特定のインスタンスに保存されたデータに依存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、代わりにクラスコンストラクタを作ることです。例えば、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(dict):
    @classmethod
    def fromkeys(cls, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = cls()
        for key in iterable:
            d[key] = value
        return d
```

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:

```
>>> d = Dict.fromkeys('abracadabra')
>>> type(d) is Dict
True
>>> d
{'a': None, 'b': None, 'r': 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:

```
import functools

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

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

    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        if hasattr(type(self.f), '__get__'):
            # This code path was added in Python 3.9
            # and was deprecated in Python 3.11.
```

(次のページに続く)

```

    return self.f.__get__(cls, cls)
return MethodType(self.f, cls)

```

The code path for `hasattr(type(self.f), '__get__')` was added in Python 3.9 and makes it possible for `classmethod()` to support chained decorators. For example, a classmethod and property could be chained together. In Python 3.11, this functionality was deprecated.

```

class G:
    @classmethod
    @property
    def __doc__(cls):
        return f'A doc for {cls.__name__!r}'

```

```

>>> G.__doc__
"A doc for 'G'"

```

`ClassMethod` 内の `functools.update_wrapper()` の呼び出しは、根底にある関数を参照する `__wrapped__` 属性を追加します。また、ラッパーがそのラップされた関数のように見えるよう、必要な属性を追加します: `__name__`、`__qualname__`、`__doc__`、`__annotations__`。

4.6 メンバーオブジェクトと `__slots__`

クラスが `__slots__` を定義すると、インスタンスの辞書が固定超のスロットの値の配列と置き換えられます。ユーザーの目線から見ると、いくつかの変化があります。

1. Provides immediate detection of bugs due to misspelled attribute assignments. Only attribute names specified in `__slots__` are allowed:

```

class Vehicle:
    __slots__ = ('id_number', 'make', 'model')

```

```

>>> auto = Vehicle()
>>> auto.id_nubmer = 'VYE483814LQEX'
Traceback (most recent call last):
...
AttributeError: 'Vehicle' object has no attribute 'id_nubmer'

```

2. Helps create immutable objects where descriptors manage access to private attributes stored in `__slots__`:

```

class Immutable:

    __slots__ = ('_dept', '_name')           # Replace the instance dictionary

```


(前のページからの続き)

```
def __init__(self, dept, name):
    self._dept = dept          # Store to private attribute
    self._name = name          # Store to private attribute

    @property                  # Read-only descriptor
    def dept(self):
        return self._dept

    @property                  # Read-only descriptor
    def name(self):
        return self._name
```

```
>>> mark = Immutable('Botany', 'Mark Watney')
>>> mark.dept
'Botany'
>>> mark.dept = 'Space Pirate'
Traceback (most recent call last):
...
AttributeError: property 'dept' of 'Immutable' object has no setter
>>> mark.location = 'Mars'
Traceback (most recent call last):
...
AttributeError: 'Immutable' object has no attribute 'location'
```

3. Saves memory. On a 64-bit Linux build, an instance with two attributes takes 48 bytes with `__slots__` and 152 bytes without. This *flyweight design pattern* likely only matters when a large number of instances are going to be created.

4. Improves speed. Reading instance variables is 35% faster with `__slots__` (as measured with Python 3.10 on an Apple M1 processor).

5. Blocks tools like `functools.cached_property()` which require an instance dictionary to function correctly:

```
from functools import cached_property

class CP:
    __slots__ = ()          # Eliminates the instance dict

    @cached_property        # Requires an instance dict
    def pi(self):
        return 4 * sum((-1.0)**n / (2.0*n + 1.0)
                        for n in reversed(range(100_000)))
```

```
>>> CP().pi
Traceback (most recent call last):
```

(次のページに続く)

```
...
TypeError: No '__dict__' attribute on 'CP' instance to cache 'pi' property.
```

`__slots__` は C 構造体に直接アクセスし、オブジェクトのメモリ確保を制御する必要があるため、ピュア Python バージョンを作成することはできません。しかし、プライベートの `__slotvalues` リストを使って C 構造体のスロットをほぼ忠実にエミュレートすることはできます。そのプライベートな構造体への読み書きはメンバーのデスク립タによって管理されます。

```
null = object()

class Member:

    def __init__(self, name, clsname, offset):
        'Emulate PyMemberDef in Include/structmember.h'
        # Also see descr_new() in Objects/descrobject.c
        self.name = name
        self.clsname = clsname
        self.offset = offset

    def __get__(self, obj, objtype=None):
        'Emulate member_get() in Objects/descrobject.c'
        # Also see PyMember_GetOne() in Python/structmember.c
        if obj is None:
            return self
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        return value

    def __set__(self, obj, value):
        'Emulate member_set() in Objects/descrobject.c'
        obj._slotvalues[self.offset] = value

    def __delete__(self, obj):
        'Emulate member_delete() in Objects/descrobject.c'
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        obj._slotvalues[self.offset] = null

    def __repr__(self):
        'Emulate member_repr() in Objects/descrobject.c'
        return f'<Member {self.name!r} of {self.clsname!r}>'
```

The `type.__new__()` method takes care of adding member objects to class variables:

```

class Type(type):
    'Simulate how the type metaclass adds member objects for slots'

    def __new__(mcls, clsname, bases, mapping, **kwargs):
        'Emulate type_new() in Objects/typeobject.c'
        # type_new() calls PyTypeReady() which calls add_methods()
        slot_names = mapping.get('slot_names', [])
        for offset, name in enumerate(slot_names):
            mapping[name] = Member(name, clsname, offset)
        return type.__new__(mcls, clsname, bases, mapping, **kwargs)

```

object.__new__() メソッドはインスタンス辞書の代わりにスロットを持つインスタンスの作成の処理をします。おおよばにピュア Python でのシミュレーションしたコードがこちらです。

```

class Object:
    'Simulate how object.__new__() allocates memory for __slots__'

    def __new__(cls, *args, **kwargs):
        'Emulate object_new() in Objects/typeobject.c'
        inst = super().__new__(cls)
        if hasattr(cls, 'slot_names'):
            empty_slots = [null] * len(cls.slot_names)
            object.__setattr__(inst, '_slotvalues', empty_slots)
        return inst

    def __setattr__(self, name, value):
        'Emulate _PyObject_GenericSetAttrWithDict() Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in cls.slot_names:
            raise AttributeError(
                f'{cls.__name__!r} object has no attribute {name!r}'
            )
        super().__setattr__(name, value)

    def __delattr__(self, name):
        'Emulate _PyObject_GenericSetAttrWithDict() Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in cls.slot_names:
            raise AttributeError(
                f'{cls.__name__!r} object has no attribute {name!r}'
            )
        super().__delattr__(name)

```

To use the simulation in a real class, just inherit from Object and set the metaclass to Type:

```

class H(Object, metaclass=Type):
    'Instance variables stored in slots'

```

(次のページに続く)

```

slot_names = ['x', 'y']

def __init__(self, x, y):
    self.x = x
    self.y = y

```

この時点ではメタクラスは x と y のメンバーオブジェクトをロードします。

```

>>> from pprint import pp
>>> pp(dict(vars(H)))
{'__module__': '__main__',
 '__doc__': 'Instance variables stored in slots',
 'slot_names': ['x', 'y'],
 '__init__': <function H.__init__ at 0x7fb5d302f9d0>,
 'x': <Member 'x' of 'H'>,
 'y': <Member 'y' of 'H'>}

```

インスタンスが作成されると、属性が保持される `slot_values` のリストをインスタンスが保持します:

```

>>> h = H(10, 20)
>>> vars(h)
{'_slotvalues': [10, 20]}
>>> h.x = 55
>>> vars(h)
{'_slotvalues': [55, 20]}

```

スペルミスや、未アサインの属性は例外を送出します。

```

>>> h.xz
Traceback (most recent call last):
...
AttributeError: 'H' object has no attribute 'xz'

```