
HowTo - Guia de descritores

Release 3.10.19

**Guido van Rossum
and the Python development team**

outubro 16, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Introdução	3
1.1	Exemplo simples: um descritor que retorna uma constante	3
1.2	Pesquisas dinâmicas	3
1.3	Atributos gerenciados	4
1.4	Nomes personalizados	5
1.5	Pensamentos finais	6
2	Exemplo completamente prático	7
2.1	Classe Validator	7
2.2	Validadores personalizados	7
2.3	Aplicação prática	8
3	Tutorial técnico	9
3.1	Resumo	9
3.2	Definição e introdução	9
3.3	Protocolo Descriptor	9
3.4	Visão geral da invocação do descritor	10
3.5	Invocação de uma instância	10
3.6	Invocação de uma classe	11
3.7	Invocação de super	11
3.8	Resumo da lógica de invocação	11
3.9	Notificação automática de nome	12
3.10	Exemplo de ORM	12
4	Equivalentes de Python puro	13
4.1	Propriedades	13
4.2	Funções e métodos	14
4.3	Tipos de métodos	16
4.4	Métodos estáticos	16
4.5	Métodos de classe	17
4.6	Member objects and <code>__slots__</code>	18

Autor Raymond Hettinger

Contato [<python at rcn dot com>](mailto:python@rcn dot com)

Sumário

- *HowTo - Guia de descritores*
 - *Introdução*
 - * *Exemplo simples: um descritor que retorna uma constante*
 - * *Pesquisas dinâmicas*
 - * *Atributos gerenciados*
 - * *Nomes personalizados*
 - * *Pensamentos finais*
 - *Exemplo completamente prático*
 - * *Classe Validator*
 - * *Validadores personalizados*
 - * *Aplicação prática*
 - *Tutorial técnico*
 - * *Resumo*
 - * *Definição e introdução*
 - * *Protocolo Descriptor*
 - * *Visão geral da invocação do descritor*
 - * *Invocação de uma instância*
 - * *Invocação de uma classe*
 - * *Invocação de super*
 - * *Resumo da lógica de invocação*
 - * *Notificação automática de nome*
 - * *Exemplo de ORM*
 - *Equivalentes de Python puro*
 - * *Propriedades*
 - * *Funções e métodos*
 - * *Tipos de métodos*
 - * *Métodos estáticos*
 - * *Métodos de classe*
 - * *Member objects and __slots__*

Descritores permitem que os objetos personalizem a consulta, o armazenamento e a exclusão de atributos.

Este guia tem quatro seções principais:

- 1) A “introdução” oferece uma visão geral básica, movendo-se suavemente a partir de exemplos simples, adicionando um recurso de cada vez. Comece aqui se você for novo em descritores.
- 2) A segunda seção mostra um exemplo de descritor prático completo. Se você já conhece o básico, comece por aí.
- 3) A terceira seção fornece um tutorial mais técnico que aborda a mecânica detalhada de como os descritores funcionam. A maioria das pessoas não precisa desse nível de detalhe.

- 4) A última seção tem equivalentes puros de Python para descritores embutidos que são escritos em C. Leia isto se estiver curioso sobre como as funções se transformam em métodos vinculados ou sobre a implementação de ferramentas comuns como `classmethod()`, `staticmethod()`, `property()` e `__slots__`.

1 Introdução

Nesta introdução, começamos com o exemplo mais básico possível e, em seguida, adicionaremos novos recursos um por um.

1.1 Exemplo simples: um descritor que retorna uma constante

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

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

Para usar o descritor, ele deve ser armazenado como uma variável de classe em outra classe:

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

Uma sessão interativa mostra a diferença entre a pesquisa de atributo normal e a pesquisa de descritor:

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

Na pesquisa de atributo `a.x`, o operador ponto encontra `'x': 5` no dicionário de classe. Na pesquisa `a.y`, o operador ponto encontra uma instância de descritor, reconhecida por seu método `__get__`. Chamar esse método retorna `10`.

Observe que o valor `10` não é armazenado no dicionário da classe ou no dicionário da instância. Em vez disso, o valor `10` é calculado sob demanda.

Este exemplo mostra como funciona um descritor simples, mas não é muito útil. Para recuperar constantes, a pesquisa de atributo normal seria melhor.

Na próxima seção, criaremos algo mais útil, uma pesquisa dinâmica.

1.2 Pesquisas dinâmicas

Descritores interessantes normalmente executam cálculos em vez de retornar constantes:

```
import os

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

class Directory:
    size = DirectorySize()           # Descriptor instance
```

(continua na próxima página)

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

Uma sessão interativa mostra que a pesquisa é dinâmica – calcula respostas diferentes e atualizadas a cada vez:

```
>>> 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
```

Além de mostrar como os descritores podem executar cálculos, este exemplo também revela o propósito dos parâmetros para `__get__()`. O parâmetro *self* é *size*, uma instância de *DirectorySize*. O parâmetro *obj* é *g* ou *s*, uma instância de *Directory*. É o parâmetro *obj* que permite ao método `__get__()` aprender o diretório de destino. O parâmetro *objtype* é a classe *Directory*.

1.3 Atributos gerenciados

Um uso popular para descritores é gerenciar o acesso aos dados da instância. O descritor é atribuído a um atributo público no dicionário da classe, enquanto os dados reais são armazenados como um atributo privado no dicionário da instância. Os métodos `__get__()` e `__set__()` do descritor são disparados quando o atributo público é acessado.

No exemplo a seguir, *age* é o atributo público e *_age* é o atributo privado. Quando o atributo público é acessado, o descritor registra a pesquisa ou atualização:

```
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__()
```

Uma sessão interativa mostra que todo o acesso ao atributo gerenciado *age* é registrado, mas que o atributo regular *name* não é registrado:

```

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

```

Um grande problema com este exemplo é que o nome privado `_age` está conectado na classe `LoggedAgeAccess`. Isso significa que cada instância pode ter apenas um atributo registrado e que seu nome é imutável. No próximo exemplo, vamos corrigir esse problema.

1.4 Nomes personalizados

Quando uma classe usa descritores, ela pode informar a cada descritor sobre qual nome de variável foi usado.

Neste exemplo, a classe `Person` tem duas instâncias de descritor, `name` e `age`. Quando a classe `Person` é definida, ela faz uma função de retorno para `__set_name__()` em `LoggedAccess` para que os nomes dos campos possam ser registrados, dando a cada descritor o seu próprio `public_name` e `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):

```

(continua na próxima página)

```

self.name = name          # Calls the first descriptor
self.age = age            # Calls the second descriptor

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

```

Uma sessão interativa mostra que a classe `Person` chamou `__set_name__()` para que os nomes dos campos fossem registrados. Aqui chamamos `vars()` para pesquisar o descritor sem acioná-lo:

```

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

```

A nova classe agora registra acesso a *name* e *age*:

```

>>> 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

```

As duas instâncias *Person* contêm apenas os nomes privados:

```

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

```

1.5 Pensamentos finais

Um descritor é o que chamamos de qualquer objeto que define `__get__()`, `__set__()` ou `__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.)

Descritores são invocados pelo operador ponto durante a pesquisa de atributos. Se um descritor for acessado indiretamente com `vars(some_class)[descriptor_name]`, a instância do descritor é retornada sem invocá-lo.

Descritores só funcionam quando usados como variáveis de classe. Quando colocados em instâncias, eles não têm efeito.

A principal motivação para descritores é fornecer um gancho permitindo que objetos armazenados em variáveis de classe controlem o que acontece durante a pesquisa de atributos.

Tradicionalmente, a classe de chamada controla o que acontece durante a pesquisa. Descritores invertem esse relacionamento e permitem que os dados pesquisados tenham uma palavra a dizer sobre o assunto.

Descritores são usados em toda a linguagem. É como funções se transformam em métodos vinculados. Ferramentas comuns como `classmethod()`, `staticmethod()`, `property()` e `functools.cached_property()` são todas implementadas como descritores.

2 Exemplo completamente prático

Neste exemplo, criamos uma ferramenta prática e poderosa para localizar bugs de corrupção de dados notoriamente difíceis de encontrar.

2.1 Classe Validator

Um validador é um descritor para acesso de atributo gerenciado. Antes de armazenar quaisquer dados, ele verifica se o novo valor atende a várias restrições de tipo e intervalo. Se essas restrições não forem atendidas, ele levanta uma exceção para evitar corrupção de dados em sua origem.

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 Validadores personalizados

Vemos aqui três utilitários práticos de validação de dados:

- 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
```

(continua na próxima página)

```

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):
            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 Aplicação prática

Veja como os validadores de dados podem ser usados em uma classe real:

```

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

```

Os descritores impedem que instâncias inválidas sejam criadas:

```

>>> 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

```

(continua na próxima página)


```

Traceback (most recent call last):
...
ValueError: Expected 'mettle' 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
>>> 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 Tutorial técnico

O que se segue é um tutorial mais técnico sobre a mecânica e os detalhes de como os descritores funcionam.

3.1 Resumo

Define descritores, resume o protocolo e mostra como os descritores são chamados. Fornece um exemplo mostrando como mapeamentos relacionais de objetos funcionam.

Aprender sobre descritores não apenas fornece acesso a um conjunto de ferramentas maior, mas também cria uma compreensão mais profunda de como o Python funciona.

3.2 Definição e introdução

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.

O comportamento padrão para acesso a atributos é obter, definir ou excluir o atributo do dicionário de um objeto. Por exemplo, `a.x` tem uma cadeia de pesquisa começando com `a.__dict__['x']`, depois `type(a).__dict__['x']` e continuando pela ordem de resolução de métodos de `type(a)`. Se o valor pesquisado for um objeto que define um dos métodos descritores, o Python pode substituir o comportamento padrão e invocar o método descritor. Onde isso ocorre na cadeia de precedência depende de quais métodos descritores foram definidos.

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 Protocolo Descriptor

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

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

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

É só isso. Defina qualquer um desses métodos e um objeto é considerado um descritor e pode substituir o comportamento padrão ao ser pesquisado como um atributo.

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).

Descritores de dados e não dados diferem em como as substituições são calculadas com relação às entradas no dicionário de uma instância. Se o dicionário de uma instância tiver uma entrada com o mesmo nome de um descritor de dados, o descritor de dados terá precedência. Se o dicionário de uma instância tiver uma entrada com o mesmo nome de um descritor não dados, a entrada do dicionário terá precedência.

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 Visão geral da invocação do descritor

Um descritor pode ser chamado diretamente com `desc.__get__(obj)` ou `desc.__get__(None, cls)`.

Mas é mais comum que um descritor seja invocado automaticamente a partir do acesso ao atributo.

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.

Os detalhes da invocação dependem se `obj` é um objeto, classe ou instância de super.

3.5 Invocação de uma instância

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.

Se um descritor for encontrado para `a.x`, ele será invocado com: `desc.__get__(a, type(a))`.

A lógica para uma pesquisa pontilhada está em `object.__getattribute__()`. Aqui está um equivalente Python puro:

```
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 Invocação de uma classe

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.

Se um descritor for encontrado, ele será invocado com `desc.__get__(None, A)`.

The full C implementation can be found in `type_getattro()` and `_PyType_Lookup()` in `Objects/typeobject.c`.

3.7 Invocação de super

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

Uma pesquisa pontilhada como `super(A, obj).m` pesquisa `obj.__class__.__mro__` para a classe base B imediatamente após A e então retorna `B.__dict__['m'].__get__(obj, A)`. Se não for um descritor, m é retornado inalterado.

The full C implementation can be found in `super_getattro()` in `Objects/typeobject.c`. A pure Python equivalent can be found in [Guido's Tutorial](#).

3.8 Resumo da lógica de invocação

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

Os pontos importantes para lembrar são:

- Descriptors are invoked by the `__getattribute__()` method.
- As classes herdam esse maquinário de `object`, `type` ou `super()`.
- 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.
- Os descritores de dados sempre substituem os dicionários de instância.
- Descritores de não-dados podem ser substituídos pelos dicionários de instância.

3.9 Notificação automática de nome

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.

The implementation details are in `type_new()` and `set_names()` in `Objects/typeobject.c`.

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 Exemplo de ORM

O código a seguir é um esqueleto simplificado que mostra como os descritores de dados podem ser usados para implementar um [mapeamento relacional de objetos](#).

A ideia essencial é que os dados sejam armazenados em um banco de dados externo. As instâncias do Python só guardam chaves para as tabelas do banco de dados. Descritores cuidam de pesquisas ou atualizações:

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

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

Para usar os modelos, primeiro conecte ao banco de dados:

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

Uma sessão interativa mostra como os dados são recuperados do banco de dados e como eles podem ser atualizados:

```

>>> 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 Equivalentes de Python puro

O protocolo descritor é simples e oferece possibilidades interessantes. Vários casos de uso são tão comuns que foram pré-empacotados em ferramentas embutidas. Propriedades, métodos vinculados, métodos estáticos, métodos de classe e `__slots__` são todos baseados no protocolo descritor.

4.1 Propriedades

Chamar `property()` é uma maneira sucinta de construir um descritor de dados que dispare uma chamada de função ao acessar um atributo. Sua assinatura é:

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

A documentação mostra um uso típico para definir um atributo gerenciado `x`:

```

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'unreadable attribute {self._name}')
        return self.fget(obj)

```

(continua na próxima página)

```

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

def __delete__(self, obj):
    if self.fdel is None:
        raise AttributeError(f"can't delete attribute {self._name}")
    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
    return prop

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

```

O recurso embutido `property()` ajuda sempre que uma interface de usuário concede acesso a atributos e alterações subsequentes exigem a intervenção de um método.

Por exemplo, uma classe de planilha pode conceder acesso a um valor de célula por meio de `Cell('b10').value`. Melhorias subsequentes no programa exigem que a célula seja recalculada em cada acesso; no entanto, o programador não quer afetar o código do cliente existente acessando o atributo diretamente. A solução é encapsular o acesso ao atributo de valor em um descritor de dados de propriedade:

```

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 Funções e métodos

Os recursos orientados a objetos do Python são construídos sobre um ambiente baseado em funções. Usando descritores de não-dados, os dois são mesclados perfeitamente.

Funções armazenadas em dicionários de classe são transformadas em métodos quando invocadas. Métodos diferem de funções regulares apenas porque a instância do objeto é prefixada aos outros argumentos. Por convenção, a instância é chamada *self*, mas poderia ser chamada *this* ou qualquer outro nome de variável.trabalhar

Os métodos podem ser criados manualmente com `types.MethodType`, que é aproximadamente equivalente a:

```

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

    def __init__(self, func, obj):

```

(continua na próxima página)

```

self.__func__ = func
self.__self__ = obj

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)

```

A execução da classe a seguir no interpretador mostra como o descritor de função funciona na prática:

```

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

```

A função tem um atributo nome qualificado para dar suporte à introspecção:

```

>>> 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>>

```

Internamente, o método vinculado armazena a função subjacente e a instância vinculada:

```

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

>>> d.f.__self__
<__main__.D object at 0x1012e1f98>

```

Se você já se perguntou de onde vem *self* em métodos regulares ou de onde vem *cls* em métodos de classe, é isso!

4.3 Tipos de métodos

Descritores de não-dados fornecem um mecanismo simples para variações nos padrões usuais de vinculação de funções em métodos.

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)`.

Este gráfico resume a ligação e suas duas variantes mais úteis:

Transformação	Chamada de um objeto	Chamada de uma classe
função	<code>f(obj, *args)</code>	<code>f(*args)</code>
staticmethod	<code>f(*args)</code>	<code>f(*args)</code>
classmethod	<code>f(type(obj), *args)</code>	<code>f(cls, *args)</code>

4.4 Métodos estáticos

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattr__(c, "f")` or `object.__getattr__(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 static methods return the underlying function with no changes, the example calls are unexciting:

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

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

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

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

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

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

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


4.5 Métodos de classe

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

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

This behavior is useful whenever the method only needs to have a class reference and does not rely on data stored in a specific instance. One use for class methods is to create alternate class constructors. For example, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```
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
```

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

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

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

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

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

    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        if hasattr(type(self.f), '__get__'):
            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:

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

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

4.6 Member objects and `__slots__`

When a class defines `__slots__`, it replaces instance dictionaries with a fixed-length array of slot values. From a user point of view that has several effects:

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

    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: can't set attribute
>>> 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):
...
TypeError: No '__dict__' attribute on 'CP' instance to cache 'pi' property.

```

It is not possible to create an exact drop-in pure Python version of `__slots__` because it requires direct access to C structures and control over object memory allocation. However, we can build a mostly faithful simulation where the actual C structure for slots is emulated by a private `_slotvalues` list. Reads and writes to that private structure are managed by member descriptors:

```

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'

```

(continua na próxima página)

```
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)
```

The `object.__new__()` method takes care of creating instances that have slots instead of an instance dictionary. Here is a rough simulation in pure Python:

```
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'

    slot_names = ['x', 'y']

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

At this point, the metaclass has loaded member objects for `x` and `y`:

```
>>> 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'>}
```

When instances are created, they have a `slot_values` list where the attributes are stored:

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

Misspelled or unassigned attributes will raise an exception:

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