
Guia de descritores

Release 3.13.2

Guido van Rossum and the Python development team

fevereiro 17, 2025

Python Software Foundation

Email: docs@python.org

Sumário

1	Primer	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@rcn dot com>

Sumário

- *Guia de descritores*
 - *Primer*
 - * *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) O “primer” 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 Primer

Neste primer, 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

A classe `Ten` é um descritor cujo método `__get__()` sempre retorna a constante 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                    # Atributo de classe comum
    y = Ten()               # Instância de descritor
```

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

```
>>> a = A()                # Cria uma instância da classe A
>>> a.x                    # Pesquisa de atributo normal
5
>>> a.y                    # Pesquisa de descritor
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()    # Instância de descritor

    def __init__(self, dirname):
        self.dirname = dirname  # Atributo de instância regular
```

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                                     # O jogo de músicas tem vinte arquivos
20
>>> g.size                                     # O diretório de jogos tem três arquivos
3
>>> os.remove('games/chess')                  # Exclui um jogo
>>> g.size                                     # Contagem de arquivos é atualizada
↪ automaticamente
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()           # Instância de descritor

    def __init__(self, name, age):
        self.name = name             # Atributo de instância regular
        self.age = age               # Chama __set__()

    def birthday(self):
        self.age += 1                # Chama __get__() e __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)          # A atualização inicial de idade é
↪ registrada
INFO:root:Updating 'age' to 30
```

(continua na próxima página)

```

>>> dave = Person('David D', 40)
INFO:root:Updating 'age' to 40

>>> vars(mary)                                     # Os dados estão em um atributo privado
{'name': 'Mary M', '_age': 30}
>>> vars(dave)
{'name': 'David D', '_age': 40}

>>> mary.age                                       # Acessa os dados e registra a pesquisa
INFO:root:Accessing 'age' giving 30
30
>>> mary.birthday()                               # Atualizações são registradas também
INFO:root:Accessing 'age' giving 30
INFO:root:Updating 'age' to 31

>>> dave.name                                     # Pesquisa de atributo regular não é
↳ registrada
'David D'
>>> dave.age                                       # Apenas o atributo gerenciado é registrado
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()           # Primeira instância de descritor
    age = LoggedAccess()           # Segunda instância de descritor

```

(continua na próxima página)

```
def __init__(self, name, age):
    self.name = name          # Chama o primeiro descritor
    self.age = age            # Chama o segundo descritor

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__()`.

Opcionalmente, os descritores podem ter um método `__set_name__()`. Isso é usado somente em casos em que um descritor precisa saber a classe onde foi criado ou o nome da variável de classe à qual foi atribuído. (Este método, se presente, é chamado mesmo se a classe não for um descritor.)

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.

Esta classe `Validator` é uma classe base abstrata e um descritor de atributo gerenciado:

```
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
```

Validadores personalizados precisam herdar de `Validator` e devem fornecer um método `validate()` para testar várias restrições conforme necessário.

2.2 Validadores personalizados

Vemos aqui três utilitários práticos de validação de dados:

- 1) `OneOf` verifica se um valor é um de um conjunto restrito de opções.
- 2) `Number` verifica se um valor é um `int` ou `float`. Opcionalmente, ele verifica se um valor está entre um mínimo ou máximo dado.
- 3) `String` verifica se um valor é um `str`. Opcionalmente, ele valida um comprimento mínimo ou máximo dado. Ele pode validar um `predicado` definido pelo usuário também.

```
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):
```

(continua na próxima página)

```

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):
            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)      # Bloqueado: 'Widget' não está todo em_
```

(continua na próxima página)


```

↪letras maiúsculas
Traceback (most recent call last):
...
ValueError: Expected <method 'isupper' of 'str' objects> to be true for 'Widget'

>>> Component('WIDGET', 'metle', 5)      # Bloqueado: 'metle' contém um erro de_
↪escrita
Traceback (most recent call last):
...
ValueError: Expected 'metle' to be one of {'metal', 'plastic', 'wood'}

>>> Component('WIDGET', 'metal', -5)      # Bloqueado: -5 é negativo
Traceback (most recent call last):
...
ValueError: Expected -5 to be at least 0

>>> Component('WIDGET', 'metal', 'V')     # Bloqueado: 'V' não é um número
Traceback (most recent call last):
...
TypeError: Expected 'V' to be an int or float

>>> c = Component('WIDGET', 'metal', 5)   # Permitido: As entradas são válidas

```

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

Em geral, um descritor é um valor de atributo que tem um dos métodos no protocolo do descritor. Esses métodos são `__get__()`, `__set__()` e `__delete__()`. Se qualquer um desses métodos for definido para um atributo, ele é dito ser um descritor.

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.

Descritores são um protocolo poderoso de propósito geral. Eles são o mecanismo por trás de propriedades, métodos, métodos estáticos, métodos de classe e `super()`. Eles são usados em todo o Python. Descritores simplificam o código C subjacente e oferecem um conjunto flexível de novas ferramentas para programas Python do dia a dia.

3.3 Protocolo Descriptor

```

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

descr.__set__(self, obj, value)

descr.__delete__(self, obj)

```

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.

Se um objeto define `__set__()` ou `__delete__()`, ele é considerado um descritor de dados. Descritores que definem apenas `__get__()` são chamados de descritores não-dados (eles são frequentemente usados para métodos, mas outros usos são possíveis).

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.

Para criar um descritor de dados somente leitura, defina `__get__()` e `__set__()` com o `__set__()` levantando `AttributeError` quando chamado. Definir o método `__set__()` com um espaço reservado levantando uma exceção é o suficiente para torná-lo um descritor de dados.

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.

A expressão `obj.x` procura o atributo `x` na cadeia de espaços de nomes para `obj`. Se a busca encontrar um descritor fora da instância `__dict__`, seu método `__get__()` é invocado de acordo com as regras de precedência listadas abaixo.

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

3.5 Invocação de uma instância

A pesquisa de instância verifica uma cadeia de espaços de nomes, dando aos descritores de dados a maior prioridade, seguidos por variáveis de instância, depois descritores que não são de dados, depois variáveis de classe e, por último, `__getattr__()`, se fornecido.

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.__getattr__()`. 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)  # descritor de dados
    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name]  # variável de instância
    if descr_get is not null:
        return descr_get(cls_var, obj, objtype)  # descritor de não dados
    if cls_var is not null:
```

(continua na próxima página)

```

return cls_var                                # variável de classe
raise AttributeError(name)

```

Note que não há nenhum gancho `__getattr__()` no código `__getattribute__()`. É por isso que chamar `__getattribute__()` diretamente ou com `super().__getattribute__` ignorará `__getattr__()` completamente.

Em vez disso, é o operador ponto e a função `getattr()` que são responsáveis por invocar `__getattr__()` sempre que `__getattribute__()` levanta um `AttributeError`. A lógica deles é encapsulada em uma função auxiliar:

```

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

A lógica para uma pesquisa pontilhada como `A.x` está em `type.__getattribute__()`. Os passos são similares aos de `object.__getattribute__()` mas a pesquisa do dicionário de instância é substituída por uma pesquisa através da ordem de resolução de métodos da classe.

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

A implementação completa em C pode ser encontrada em `type_getattro()` e `_PyType_Lookup()` em `Objects/typeobject.c`.

3.7 Invocação de super

A lógica para a pesquisa pontilhada de `super` está no método `__getattribute__()` para o objeto retornado por `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.

A implementação completa em C pode ser encontrada em `super_getattro()` em `Objects/typeobject.c`. Um equivalente em Python puro pode ser encontrado no [Tutorial do Guido](#).

3.8 Resumo da lógica de invocação

O mecanismo para descritores está incorporado nos métodos `__getattribute__()` para `object`, `type` e `super()`.

Os pontos importantes para lembrar são:

- Descritores são invocados pelo método `__getattribute__()`.
- As classes herdam esse maquinário de `object`, `type` ou `super()`.
- Substituir `__getattribute__()` impede chamadas automáticas do descritor porque toda a lógica do descritor está nesse método.
- `object.__getattribute__()` e `type.__getattribute__()` fazem chamadas diferentes para `__get__()`. O primeiro inclui a instância e pode incluir a classe. O segundo coloca `None` para a instância e sempre inclui a classe.
- 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

Às vezes, é desejável que um descritor saiba a qual nome de variável de classe ele foi atribuído. Quando uma nova classe é criada, a metaclasses `type` varre o dicionário da nova classe. Se qualquer uma das entradas for descritor e se eles definirem `__set_name__()`, esse método será chamado com dois argumentos. O *owner* é a classe onde o descritor é usado, e o *name* é a variável de classe à qual o descritor foi atribuído.

Os detalhes de implementações estão em `type_new()` e `set_names()` em [Objects/typeobject.c](#).

Como a lógica de atualização está em `type.__new__()`, as notificações só ocorrem no momento da criação da classe. Se descritores forem adicionados à classe posteriormente, `__set_name__()` precisará ser chamado manualmente.

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()
```

Podemos usar a classe `Field` para definir [modelos](#) que descrevem o esquema de cada tabela em um banco de dados:

```
class Movie:
    table = 'Movies'                # Nome da tabela
    key = 'title'                   # Chave primária
    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 dispara 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.")
```

Para ver como `property()` é implementada em termos do protocolo descritor, aqui está um equivalente Python puro que implementa a maior parte da funcionalidade principal:

```
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

    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
```

(continua na próxima página)

```

    return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError
        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__)

```

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

```

Tanto o `property()` embutida quanto nosso equivalente `Property()` funcionariam neste exemplo.

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):
        self.__func__ = func
        self.__self__ = obj

```

(continua na próxima página)

```

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

def __getattr__(self, name):
    "Emulate method_getset() in Objects/classobject.c"
    if name == '__doc__':
        return self.__func__.__doc__
    return object.__getattr__(self, name)

def __getattr__(self, name):
    "Emulate method_getattro() in Objects/classobject.c"
    return getattr(self.__func__, name)

def __get__(self, obj, objtype=None):
    "Emulate method_descr_get() in Objects/classobject.c"
    return self

```

Para dar suporte à criação automática de métodos, as funções incluem o método `__get__()` para vincular métodos durante o acesso ao atributo. Isso significa que as funções são descritores de não-dados que retornam métodos vinculados durante a pesquisa pontilhada de uma instância. Veja como funciona:

```

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):
        return self

class D2:
    pass

```

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

```

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

```

Acessar a função por meio do dicionário de classes não invoca `__get__()`. Em vez disso, ele apenas retorna o objeto da função subjacente:

```

>>> D.__dict__['f']
<function D.f at 0x00C45070>

```

O acesso pontilhado de uma classe chama `__get__()` que apenas retorna a função subjacente inalterada:

```

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

```

O comportamento interessante ocorre durante o acesso pontuado de uma instância. A pesquisa pontilhada chama `__get__()` que retorna um objeto do método vinculado:

```
>>> 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 0x00B18C90>
```

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) --> 0.9332` or `Sample.erf(1.5) --> 0.9332`.

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:


```
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 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:

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

The `functools.update_wrapper()` call in `ClassMethod` 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__`.

4.6 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'
```

(continua na próxima página)

```
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):
...
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
```

(continua na próxima página)

```

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)

```

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(

```

(continua na próxima página)

(continuação da página anterior)

```
        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'
```