Descriptor HowTo Guide

Release 3.9.0

Guido van Rossum
and the Python development team

October 25, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 Primer **3**
1.1 Simple example: A descriptor that returns a constant . . . 3
1.2 Dynamic lookups . . . 3
1.3 Managed attributes . . . 4
1.4 Customized Names . . . 5
1.5 Closing thoughts . . . 6

2 Complete Practical Example **7**
2.1 Validator class . . . 7
2.2 Custom validators . . . 7
2.3 Practical use . . . 8

3 Technical Tutorial **9**
3.1 Abstract . . . 9
3.2 Definition and Introduction . . . 9
3.3 Descriptor Protocol . . . 9
3.4 Invoking Descriptors . . . 10
3.5 Automatic Name Notification . . . 11
3.6 Descriptor Example . . . 11
3.7 Properties . . . 12
3.8 Functions and Methods . . . 13
3.9 Static Methods and Class Methods . . . 15

Author Raymond Hettinger

Contact <python at rcn dot com>

Contents

- *Descriptor HowTo Guide*

- *Primer*
  - *Simple example: A descriptor that returns a constant*
  - *Dynamic lookups*
  - *Managed attributes*
  - *Customized Names*
  - *Closing thoughts*
- *Complete Practical Example*
  - *Validator class*
  - *Custom validators*
  - *Practical use*
- *Technical Tutorial*
  - *Abstract*
  - *Definition and Introduction*
  - *Descriptor Protocol*
  - *Invoking Descriptors*
  - *Automatic Name Notification*
  - *Descriptor Example*
  - *Properties*
  - *Functions and Methods*
  - *Static Methods and Class Methods*

Descriptors let objects customize attribute lookup, storage, and deletion.

This HowTo guide has three major sections:

1) The "primer" gives a basic overview, moving gently from simple examples, adding one feature at a time. It is a great place to start.
2) The second section shows a complete, practical descriptor example. If you already know the basics, start there.
3) The third section provides a more technical tutorial that goes into the detailed mechanics of how descriptors work. Most people don't need this level of detail.

1 Primer

In this primer, we start with the most basic possible example and then we'll add new capabilities one by one.

1.1 Simple example: A descriptor that returns a constant

The `Ten` class is a descriptor that always returns the constant `10`:

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

To use the descriptor, it must be stored as a class variable in another class:

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

An interactive session shows the difference between normal attribute lookup and descriptor lookup:

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

In the `a.x` attribute lookup, the dot operator finds the value `5` stored in the class dictionary. In the `a.y` descriptor lookup, the dot operator calls the descriptor's `__get__()` method. That method returns `10`. Note that the value `10` is not stored in either the class dictionary or the instance dictionary. Instead, the value `10` is computed on demand.

This example shows how a simple descriptor works, but it isn't very useful. For retrieving constants, normal attribute lookup would be better.

In the next section, we'll create something more useful, a dynamic lookup.

1.2 Dynamic lookups

Interesting descriptors typically run computations instead of doing lookups:

```
import os

class DirectorySize:

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

class Directory:

    size = DirectorySize()                # Descriptor

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

An interactive session shows that the lookup is dynamic — it computes different, updated answers each time:

```
>>> g = Directory('games')
>>> s = Directory('songs')
>>> g.size                                  # The games directory has three files
3
>>> os.system('touch games/newfile')        # Add a fourth file to the directory
0
>>> g.size
4
>>> s.size                                  # The songs directory has twenty files
20
```

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 *obj* parameter that lets the `__get__()` method learn the target directory. The *objtype* parameter is the class *Directory*.

1.3 Managed attributes

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.

In the following example, *age* is the public attribute and *_age* is the private attribute. When the public attribute is accessed, the descriptor logs the lookup or update:

```
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

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

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

An interactive session shows that all access to the managed attribute *age* is logged, but that the regular attribute *name* is not logged:

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

(continues on next page)

(continued from previous page)

```
>>> 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
```

One major issue with this example is the private name *_age* is hardwired in the *LoggedAgeAccess* class. That means that each instance can only have one logged attribute and that its name is unchangeable. In the next example, we'll fix that problem.

1.4 Customized Names

When a class uses descriptors, it can inform each descriptor about what variable name was used.

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, force=True)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = f'_{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
```

(continues on next page)

(continued from previous page)

```
    age = LoggedAccess()                # Second descriptor

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

    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 lookup 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'}
```

The new class now logs access to both *name* and *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
```

The two *Person* instances contain only the private names:

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

1.5 Closing thoughts

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 is created or the name of class variable it was assigned to.

Descriptors get invoked by the dot operator during attribute lookup. If a descriptor is accessed indirectly with `vars(some_class)[descriptor_name]`, the descriptor instance is returned without invoking it.

Descriptors only work when used as class variables. When put in instances, they have no effect.

The main motivation for descriptors is to provide a hook allowing objects stored in class variables to control what happens during dotted lookup.

Traditionally, the calling class controls what happens during lookup. Descriptors invert that relationship and allow the data being looked-up to have a say in the matter.

Descriptors are used throughout the language. It is how functions turn into bound methods. Common tools like `classmethod()`, `staticmethod()`, `property()`, and `functools.cached_property()` are all implemented as descriptors.

2 Complete Practical Example

In this example, we create a practical and powerful tool for locating notoriously hard to find data corruption bugs.

2.1 Validator class

A validator is a descriptor for managed attribute access. Prior to storing any data, it verifies that the new value meets various type and range restrictions. If those restrictions aren't met, it raises an exception to prevent data corruption at its source.

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 = f'_{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 subclass from `Validator` and supply a `validate()` method to test various restrictions as needed.

2.2 Custom validators

Here are three practical data validation utilities:

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. Optionally, it can test for another 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):
```

(continues on next page)

(continued from previous page)

```
    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):
            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 Practical use

Here's how the data validators can be used in a real class:

```
class Component:

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

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

The descriptors prevent invalid instances from being created:

```
Component('WIDGET', 'metal', 5)      # Allowed.
Component('Widget', 'metal', 5)      # Blocked: 'Widget' is not all uppercase
Component('WIDGET', 'metle', 5)      # Blocked: 'metle' is misspelled
Component('WIDGET', 'metal', -5)     # Blocked: -5 is negative
Component('WIDGET', 'metal', 'V')    # Blocked: 'V' isn't a number
```

3 Technical Tutorial

What follows is a more technical tutorial for the mechanics and details of how descriptors work.

3.1 Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in Python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.

3.2 Definition and Introduction

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)`. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

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 Descriptor Protocol

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

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

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

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines `__set__()` or `__delete__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are typically used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

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 Invoking Descriptors

A descriptor can be called directly by its method name. For example, `d.__get__(obj)`.

But it is more common for a descriptor to be invoked automatically from attribute access. The expression `obj.d` looks up `d` in the dictionary of `obj`. If `d` defines the method `__get__()`, then `d.__get__(obj)` is invoked according to the precedence rules listed below.

The details of invocation depend on whether `obj` is an object, class, or instance of super.

Objects: The machinery is in `object.__getattribute__()`.

It transforms `b.x` into `type(b).__dict__['x'].__get__(b, type(b))`.

The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to `__getattr__()` if provided.

The full C implementation can be found in `PyObject_GenericGetAttr()` in Objects/object.c.

Classes: The machinery is in `type.__getattribute__()`.

It transforms `A.x` into `A.__dict__['x'].__get__(None, A)`.

In pure Python, it looks like this:

```
def __getattribute__(cls, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(cls, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, cls)
    return v
```

Super: The machinery is in the custom `__getattribute__()` method for object returned by `super()`.

The attribute lookup `super(A, obj).m` searches `obj.__class__.__mro__` for the base class `B` immediately following `A` and then returns `B.__dict__['m'].__get__(obj, A)`.

If not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

The implementation details are in `super_getattro()` in Objects/typeobject.c. A pure Python equivalent can be found in Guido's Tutorial.

Summary: The details listed above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`.

The important points to remember are:

- Descriptors are invoked by the `__getattribute__()` method.
- Classes inherit this machinery from `object`, `type`, or `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.

- Data descriptors always override instance dictionaries.
- Non-data descriptors may be overridden by instance dictionaries.

3.5 Automatic Name Notification

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, the *name* is 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.

3.6 Descriptor Example

The following code is simplified skeleton showing how data descriptors could be used to implement an object relational mapping.

The essential idea is that instances only hold keys to a database table. The actual data is stored in an external table that is being dynamically updated:

```
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` 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
```

An interactive session shows how data is retrieved from the database and how it can be updated:

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

>>> 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'
```

The descriptor protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound methods, static methods, and class methods are all based on the descriptor protocol.

3.7 Properties

Calling `property()` is a succinct way of building a data descriptor that triggers function calls upon access to an attribute. Its signature is:

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

The documentation shows a typical use to define a managed attribute `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

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)
```

(continues on next page)

(continued from previous page)

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

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

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

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

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

The `property()` builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```
class Cell:
    ...

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

3.8 Functions and Methods

Python's object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Functions stored in class dictionaries get turned into methods when invoked. Methods only differ from regular functions in that the object instance is prepended to the other arguments. By convention, the instance is called *self* but could be called *this* or any other variable name.

Methods can be created manually with `types.MethodType` which is roughly equivalent to:

```
class Method:
    "Emulate Py_MethodType in Objects/classobject.c"

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

(continues on next page)

(continued from previous page)

```
    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 which 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 types.MethodType(self, obj)
```

Running the following class in the interpreter shows how the function descriptor works in practice:

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

The function has a qualified name attribute to support introspection:

```
>>> 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>>
```

Internally, the bound method stores the underlying function and the bound instance:

```
>>> d.f.__func__
<function D.f at 0x1012e5ae8>

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

If you have ever wondered where *self* comes from in regular methods or where *cls* comes from in class methods, this is it!

3.9 Static Methods and Class Methods

Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

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)`.

This chart summarizes the binding and its two most useful variants:

Transformation	Called from an object	Called from a class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(cls, *args)

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattribute__(c, "f")` or `object.__getattribute__(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):
        print(x)

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

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
```

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
```

(continues on next page)

(continued from previous page)

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

This behavior is useful whenever the function only needs to have a class reference and does not care about any underlying data. One use for class methods is to create alternate class constructors. The classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```
class 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:

```
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': 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(obj, '__get__'):
            return self.f.__get__(cls)
        return types.MethodType(self.f, cls)
```

The code path for `hasattr(obj, '__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}'
```