CHAPTER 9
A Pythonic Object

> Never, ever use two leading underscores. This is annoyingly private.[1]
>
> — Ian Bicking
> *Creator of pip, virtualenv, Paste and many other projects*

Thanks to the Python data model, your user-defined types can behave as naturally as the built-in types. And this can be accomplished without inheritance, in the spirit of *duck typing*: you just implement the methods needed for your objects to behave as expected.

In previous chapters, we presented the structure and behavior of many built-in objects. We will now build user-defined classes that behave as real Python objects.

This chapter starts where Chapter 1 ended, by showing how to implement several special methods that are commonly seen in Python objects of many different types.

In this chapter, we will see how to:

- Support the built-in functions that produce alternative object representations (e.g., `repr()`, `bytes()`, etc).
- Implement an alternative constructor as a class method.
- Extend the format mini-language used by the `format()` built-in and the `str.format()` method.
- Provide read-only access to attributes.
- Make an object hashable for use in sets and as `dict` keys.
- Save memory with the use of `__slots__`.

1. From the Paste Style Guide.

We'll do all that as we develop a simple two-dimensional Euclidean vector type.

The evolution of the example will be paused to discuss two conceptual topics:

- How and when to use the `@classmethod` and `@staticmethod` decorators.
- Private and protected attributes in Python: usage, conventions, and limitations.

Let's get started with the object representation methods.

Object Representations

Every object-oriented language has at least one standard way of getting a string representation from any object. Python has two:

`repr()`
: Return a string representing the object as the developer wants to see it.

`str()`
: Return a string representing the object as the user wants to see it.

As you know, we implement the special methods `__repr__` and `__str__` to support `repr()` and `str()`.

There are two additional special methods to support alternative representations of objects: `__bytes__` and `__format__`. The `__bytes__` method is analogous to `__str__`: it's called by `bytes()` to get the object represented as a byte sequence. Regarding `__format__`, both the built-in function `format()` and the `str.format()` method call it to get string displays of objects using special formatting codes. We'll cover `__bytes__` in the next example, and `__format__` after that.

> If you're coming from Python 2, remember that in Python 3 `__repr__`, `__str__`, and `__format__` must always return Unicode strings (type `str`). Only `__bytes__` is supposed to return a byte sequence (type `bytes`).

Vector Class Redux

In order to demonstrate the many methods used to generate object representations, we'll use a `Vector2d` class similar to the one we saw in Chapter 1. We will build on it in this and future sections. Example 9-1 illustrates the basic behavior we expect from a `Vector2d` instance.

Example 9-1. Vector2d instances have several representations

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)  ❶
```

```
3.0 4.0
>>> x, y = v1  ❷
>>> x, y
(3.0, 4.0)
>>> v1  ❸
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))  ❹
>>> v1 == v1_clone  ❺
True
>>> print(v1)  ❻
(3.0, 4.0)
>>> octets = bytes(v1)  ❼
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)  ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0))  ❾
(True, False)
```

❶ The components of a `Vector2d` can be accessed directly as attributes (no getter method calls).

❷ A `Vector2d` can be unpacked to a tuple of variables.

❸ The `repr` of a `Vector2d` emulates the source code for constructing the instance.

❹ Using `eval` here shows that the `repr` of a `Vector2d` is a faithful representation of its constructor call.[2]

❺ `Vector2d` supports comparison with `==`; this is useful for testing.

❻ `print` calls `str`, which for `Vector2d` produces an ordered pair display.

❼ `bytes` uses the `__bytes__` method to produce a binary representation.

❽ `abs` uses the `__abs__` method to return the magnitude of the `Vector2d`.

❾ `bool` uses the `__bool__` method to return `False` for a `Vector2d` of zero magnitude or `True` otherwise.

`Vector2d` from Example 9-1 is implemented in *vector2d_v0.py* (Example 9-2). The code is based on Example 1-2, but the infix operators will be implemented in Chapter 13—except for `==` (which is useful for testing). At this point, `Vector2d` uses several special methods to provide operations that a Pythonista expects in a well-designed object.

Example 9-2. vector2d_v0.py: methods so far are all special methods

```
from array import array
import math
```

2. I used `eval` to clone the object here just to make a point about `repr`; to clone an instance, the `copy.copy` function is safer and faster.

```python
class Vector2d:
    typecode = 'd'  ❶

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

    def __iter__(self):
        return (i for i in (self.x, self.y))  ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)  ❹

    def __str__(self):
        return str(tuple(self))  ❺

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +  ❻
                bytes(array(self.typecode, self)))  ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other)  ❽

    def __abs__(self):
        return math.hypot(self.x, self.y)  ❾

    def __bool__(self):
        return bool(abs(self))  ❿
```

❶ `typecode` is a class attribute we'll use when converting `Vector2d` instances to/from `bytes`.

❷ Converting `x` and `y` to `float` in `__init__` catches errors early, which is helpful in case `Vector2d` is called with unsuitable arguments.

❸ `__iter__` makes a `Vector2d` iterable; this is what makes unpacking work (e.g, `x, y = my_vector`). We implement it simply by using a generator expression to yield the components one after the other.[3]

❹ `__repr__` builds a string by interpolating the components with `{!r}` to get their `repr`; because `Vector2d` is iterable, `*self` feeds the `x` and `y` components to `format`.

❺ From an iterable `Vector2d`, it's easy to build a `tuple` for display as an ordered pair.

3. This line could also be written as `yield self.x; yield.self.y`. I have a lot more to say about the `__iter__` special method, generator expressions, and the `yield` keyword in Chapter 14.

❻ To generate bytes, we convert the typecode to bytes and concatenate…

❼ …bytes converted from an array built by iterating over the instance.

❽ To quickly compare all components, build tuples out of the operands. This works for operands that are instances of Vector2d, but has issues. See the following warning.

❾ The magnitude is the length of the hypotenuse of the triangle formed by the x and y components.

❿ `__bool__` uses `abs(self)` to compute the magnitude, then converts it to bool, so 0.0 becomes False, nonzero is True.

> Method `__eq__` in Example 9-2 works for Vector2d operands but also returns True when comparing Vector2d instances to other iterables holding the same numeric values (e.g., `Vector(3, 4) == [3, 4]`). This may be considered a feature or a bug. Further discussion needs to wait until Chapter 13, when we cover operator overloading.

We have a fairly complete set of basic methods, but one obvious operation is missing: rebuilding a Vector2d from the binary representation produced by `bytes()`.

An Alternative Constructor

Because we can export a Vector2d as bytes, naturally we need a method that imports a Vector2d from a binary sequence. Looking at the standard library for inspiration, we find that array.array has a class method named .frombytes that suits our purpose—we saw it in "Arrays" on page 48. We adopt its name and use its functionality in a class method for Vector2d in *vector2d_v1.py* (Example 9-3).

Example 9-3. Part of vector2d_v1.py: this snippet shows only the frombytes class method, added to the Vector2d definition in vector2d_v0.py (Example 9-2)

```
    @classmethod  ❶
    def frombytes(cls, octets):  ❷
        typecode = chr(octets[0])  ❸
        memv = memoryview(octets[1:]).cast(typecode)  ❹
        return cls(*memv)  ❺
```

❶ Class method is modified by the classmethod decorator.

❷ No self argument; instead, the class itself is passed as cls.

❸ Read the typecode from the first byte.

❹ Create a `memoryview` from the `octets` binary sequence and use the `typecode` to cast it.[4]

❺ Unpack the `memoryview` resulting from the cast into the pair of arguments needed for the constructor.

Because we just used a `classmethod` decorator, and it is very Python-specific, let's have a word about it.

classmethod Versus staticmethod

The `classmethod` decorator is not mentioned in the Python tutorial, and neither is `staticmethod`. Anyone who has learned OO in Java may wonder why Python has both of these decorators and not just one of them.

Let's start with `classmethod`. Example 9-3 shows its use: to define a method that operates on the class and not on instances. `classmethod` changes the way the method is called, so it receives the class itself as the first argument, instead of an instance. Its most common use is for alternative constructors, like `frombytes` in Example 9-3. Note how the last line of `frombytes` actually uses the `cls` argument by invoking it to build a new instance: `cls(*memv)`. By convention, the first parameter of a class method should be named `cls` (but Python doesn't care how it's named).

In contrast, the `staticmethod` decorator changes a method so that it receives no special first argument. In essence, a static method is just like a plain function that happens to live in a class body, instead of being defined at the module level. Example 9-4 contrasts the operation of `classmethod` and `staticmethod`.

Example 9-4. Comparing behaviors of classmethod and staticmethod

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args  # ❶
...     @staticmethod
...     def statmeth(*args):
...         return args  # ❷
...
>>> Demo.klassmeth()  # ❸
(<class '__main__.Demo'>,)
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth()   # ❹
()
```

4. We had a brief introduction to `memoryview`, explaining its `.cast` method in "Memory Views" on page 51.

```
>>> Demo.statmeth('spam')
('spam',)
```

❶ `klassmeth` just returns all positional arguments.

❷ `statmeth` does the same.

❸ No matter how you invoke it, `Demo.klassmeth` receives the `Demo` class as the first argument.

❹ `Demo.statmeth` behaves just like a plain old function.

> The `classmethod` decorator is clearly useful, but I've never seen a compelling use case for `staticmethod`. If you want to define a function that does not interact with the class, just define it in the module. Maybe the function is closely related even if it never touches the class, so you want to them nearby in the code. Even so, defining the function right before or after the class in the same module is close enough for all practical purposes.[5]

Now that we've seen what `classmethod` is good for (and that `staticmethod` is not very useful), let's go back to the issue of object representation and see how to support formatted output.

Formatted Displays

The `format()` built-in function and the `str.format()` method delegate the actual formatting to each type by calling their `.__format__(format_spec)` method. The `format_spec` is a formatting specifier, which is either:

- The second argument in `format(my_obj, format_spec)`, or
- Whatever appears after the colon in a replacement field delimited with `{}` inside a format string used with `str.format()`

For example:

```
>>> brl = 1/2.43  # BRL to USD currency conversion rate
>>> brl
0.4115226337448559
>>> format(brl, '0.4f')  # ❶
'0.4115'
```

5. Leonardo Rochael, one of the technical reviewers of this book disagrees with my low opinion of `staticmethod`, and recommends the blog post "The Definitive Guide on How to Use Static, Class or Abstract Methods in Python" by Julien Danjou as a counter-argument. Danjou's post is very good; I do recommend it. But it wasn't enough to change my mind about `staticmethod`. You'll have to decide for yourself.

```
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl)  # ❷
'1 BRL = 0.41 USD'
```

❶ Formatting specifier is `'0.4f'`.

❷ Formatting specifier is `'0.2f'`. The `'rate'` substring in the replacement field is called the field name. It's unrelated to the formatting specifier, but determines which argument of `.format()` goes into that replacement field.

The second callout makes an important point: a format string such as `'{0.mass:5.3e}'` actually uses two separate notations. The `'0.mass'` to the left of the colon is the `field_name` part of the replacement field syntax; the `'5.3e'` after the colon is the formatting specifier. The notation used in the formatting specifier is called the Format Specification Mini-Language.

> If `format()` and `str.format()` are new to you, classroom experience has shown that it's best to study the `format()` function first, which uses just the Format Specification Mini-Language. After you get the gist of that, read Format String Syntax to learn about the `{:}` replacement field notation, used in the `str.format()` method (including the `!s`, `!r`, and `!a` conversion flags).

A few built-in types have their own presentation codes in the Format Specification Mini-Language. For example—among several other codes—the `int` type supports `b` and `x` for base 2 and base 16 output, respectively, while `float` implements `f` for a fixed-point display and `%` for a percentage display:

```
>>> format(42, 'b')
'101010'
>>> format(2/3, '.1%')
'66.7%'
```

The Format Specification Mini-Language is extensible because each class gets to interpret the `format_spec` argument as it likes. For instance, the classes in the `datetime` module use the same format codes in the `strftime()` functions and in their `__format__` methods. Here are a couple examples using the `format()` built-in and the `str.format()` method:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

If a class has no `__format__`, the method inherited from `object` returns `str(my_object)`. Because `Vector2d` has a `__str__`, this works:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

However, if you pass a format specifier, `object.__format__` raises `TypeError`:

```
>>> format(v1, '.3f')
Traceback (most recent call last):
  ...
TypeError: non-empty format string passed to object.__format__
```

We will fix that by implementing our own format mini-language. The first step will be to assume the format specifier provided by the user is intended to format each `float` component of the vector. This is the result we want:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Example 9-5 implements `__format__` to produce the displays just shown.

Example 9-5. Vector2d.format method, take #1

```
# inside the Vector2d class

def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self)  # ❶
    return '({}, {})'.format(*components)  # ❷
```

❶ Use the `format` built-in to apply the `fmt_spec` to each vector component, building an iterable of formatted strings.

❷ Plug the formatted strings in the formula `'(x, y)'`.

Now let's add a custom formatting code to our mini-language: if the format specifier ends with a `'p'`, we'll display the vector in polar coordinates: `<r, θ>`, where `r` is the magnitude and θ (theta) is the angle in radians. The rest of the format specifier (whatever comes before the `'p'`) will be used as before.

When choosing the letter for the custom format code I avoided overlapping with codes used by other types. In Format Specification Mini-Language we see that integers use the codes `'bcdoxXn'`, floats use `'eEfFgGn%'`, and strings use `'s'`. So I picked `'p'` for polar coordinates. Because each class interprets these codes independently, reusing a code letter in a custom format for a new type is not an error, but may be confusing to users.

To generate polar coordinates we already have the `__abs__` method for the magnitude, and we'll code a simple `angle` method using the `math.atan2()` function to get the angle. This is the code:

```
    # inside the Vector2d class

    def angle(self):
        return math.atan2(self.y, self.x)
```

With that, we can enhance our `__format__` to produce polar coordinates. See Example 9-6.

Example 9-6. Vector2d.format method, take #2, now with polar coordinates

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):  ❶
        fmt_spec = fmt_spec[:-1]  ❷
        coords = (abs(self), self.angle())  ❸
        outer_fmt = '<{}, {}>'  ❹
    else:
        coords = self  ❺
        outer_fmt = '({}, {})'  ❻
    components = (format(c, fmt_spec) for c in coords)  ❼
    return outer_fmt.format(*components)  ❽
```

❶ Format ends with `'p'`: use polar coordinates.

❷ Remove `'p'` suffix from `fmt_spec`.

❸ Build `tuple` of polar coordinates: `(magnitude, angle)`.

❹ Configure outer format with angle brackets.

❺ Otherwise, use `x, y` components of `self` for rectangular coordinates.

❻ Configure outer format with parentheses.

❼ Generate iterable with components as formatted strings.

❽ Plug formatted strings into outer format.

With Example 9-6, we get results similar to these:

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

As this section shows, it's not hard to extend the format specification mini-language to support user-defined types.

Now let's move to a subject that's not just about appearances: we will make our `Vector2d` hashable, so we can build sets of vectors, or use them as `dict` keys. But before we can do that, we must make vectors immutable. We'll do what it takes next.

A Hashable Vector2d

As defined, so far our `Vector2d` instances are unhashable, so we can't put them in a `set`:

```
>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
```

To make a `Vector2d` hashable, we must implement `__hash__` (`__eq__` is also required, and we already have it). We also need to make vector instances immutable, as we've seen in "What Is Hashable?" on page 65.

Right now, anyone can do `v1.x = 7` and there is nothing in the code to suggest that changing a `Vector2d` is forbidden. This is the behavior we want:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

We'll do that by making the `x` and `y` components read-only properties in Example 9-7.

Example 9-7. vector2d_v3.py: only the changes needed to make Vector2d immutable are shown here; see full listing in Example 9-9

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)  ❶
        self.__y = float(y)

    @property  ❷
    def x(self):  ❸
        return self.__x  ❹

    @property  ❺
    def y(self):
        return self.__y
```

```python
    def __iter__(self):
        return (i for i in (self.x, self.y))  ❻

    # remaining methods follow (omitted in book listing)
```

❶ Use exactly two leading underscores (with zero or one trailing underscore) to make an attribute private.[6]

❷ The `@property` decorator marks the getter method of a property.

❸ The getter method is named after the public property it exposes: `x`.

❹ Just return `self.__x`.

❺ Repeat same formula for `y` property.

❻ Every method that just reads the `x`, `y` components can stay as they were, reading the public properties via `self.x` and `self.y` instead of the private attribute, so this listing omits the rest of the code for the class.

> `Vector.x` and `Vector.y` are examples of read-only properties. Read/write properties will be covered in Chapter 19, where we dive deeper into the `@property`.

Now that our vectors are reasonably immutable, we can implement the `__hash__` method. It should return an `int` and ideally take into account the hashes of the object attributes that are also used in the `__eq__` method, because objects that compare equal should have the same hash. The `__hash__` special method documentation suggests using the bitwise XOR operator (`^`) to mix the hashes of the components, so that's what we do. The code for our `Vector2d.__hash__` method is really simple, as shown in Example 9-8.

Example 9-8. vector2d_v3.py: implementation of hash

```python
    # inside class Vector2d:

    def __hash__(self):
        return hash(self.x) ^ hash(self.y)
```

With the addition of the `__hash__` method, we now have hashable vectors:

```python
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
```

6. This is not how Ian Bicking would do it; recall the quote at the start of the chapter. The pros and cons of private attributes are the subject of the upcoming "Private and "Protected" Attributes in Python" on page 262.

```
(7, 384307168202284039)
>>> set([v1, v2])
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```

> It's not strictly necessary to implement properties or otherwise protect the instance attributes to create a hashable type. Implementing `__hash__` and `__eq__` correctly is all it takes. But the hash value of an instance is never supposed to change, so this provides an excellent opportunity to talk about read-only properties.

If you are creating a type that has a sensible scalar numeric value, you may also implement the `__int__` and `__float__` methods, invoked by the `int()` and `float()` constructors—which are used for type coercion in some contexts. There's also a `__complex__` method to support the `complex()` built-in constructor. Perhaps `Vector2d` should provide `__complex__`, but I'll leave that as an exercise for you.

We have been working on `Vector2d` for a while, showing just snippets, so Example 9-9 is a consolidated, full listing of *vector2d_v3.py*, including all the doctests I used when developing it.

Example 9-9. vector2d_v3.py: the full monty

```
"""
A two-dimensional vector class

    >>> v1 = Vector2d(3, 4)
    >>> print(v1.x, v1.y)
    3.0 4.0
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector2d(3.0, 4.0)
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets
    b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector2d(0, 0))
    (True, False)


Test of ``.frombytes()`` class method:
```

```
    >>> v1_clone = Vector2d.frombytes(bytes(v1))
    >>> v1_clone
    Vector2d(3.0, 4.0)
    >>> v1 == v1_clone
    True


Tests of ``format()`` with Cartesian coordinates:

    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Tests of the ``angle`` method::

    >>> Vector2d(0, 0).angle()
    0.0
    >>> Vector2d(1, 0).angle()
    0.0
    >>> epsilon = 10**-8
    >>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
    True
    >>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
    True


Tests of ``format()`` with polar coordinates:

    >>> format(Vector2d(1, 1), 'p')  # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector2d(1, 1), '.3ep')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector2d(1, 1), '0.5fp')
    '<1.41421, 0.78540>'


Tests of `x` and `y` read-only properties:

    >>> v1.x, v1.y
    (3.0, 4.0)
    >>> v1.x = 123
    Traceback (most recent call last):
      ...
    AttributeError: can't set attribute


Tests of hashing:
```

```
    >>> v1 = Vector2d(3, 4)
    >>> v2 = Vector2d(3.1, 4.2)
    >>> hash(v1), hash(v2)
    (7, 384307168202284039)
    >>> len(set([v1, v2]))
    2

"""

from array import array
import math

class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __hash__(self):
        return hash(self.x) ^ hash(self.y)

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))
```

```
    def angle(self):
        return math.atan2(self.y, self.x)

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('p'):
            fmt_spec = fmt_spec[:-1]
            coords = (abs(self), self.angle())
            outer_fmt = '<{}, {}>'
        else:
            coords = self
            outer_fmt = '({}, {})'
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(*components)

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(*memv)
```

To recap, in this and the previous sections, we saw some essential special methods that you may want to implement to have a full-fledged object. Of course, it is a bad idea to implement all of these methods if your application has no real use for them. Customers don't care if your objects are "Pythonic" or not.

As coded in Example 9-9, `Vector2d` is a didactic example with a laundry list of special methods related to object representation, not a template for every user-defined class.

In the next section, we'll take a break from `Vector2d` to discuss the design and drawbacks of the private attribute mechanism in Python—the double-underscore prefix in `self.__x`.

Private and "Protected" Attributes in Python

In Python, there is no way to create private variables like there is with the `private` modifier in Java. What we have in Python is a simple mechanism to prevent accidental overwriting of a "private" attribute in a subclass.

Consider this scenario: someone wrote a class named `Dog` that uses a `mood` instance attribute internally, without exposing it. You need to subclass `Dog` as `Beagle`. If you create your own `mood` instance attribute without being aware of the name clash, you will clobber the `mood` attribute used by the methods inherited from `Dog`. This would be a pain to debug.

To prevent this, if you name an instance attribute in the form `__mood` (two leading underscores and zero or at most one trailing underscore), Python stores the name in the instance `__dict__` prefixed with a leading underscore and the class name, so in the

Dog class, __mood becomes _Dog__mood, and in Beagle it's _Beagle__mood. This language feature goes by the lovely name of *name mangling*.

Example 9-10 shows the result in the Vector2d class from Example 9-7.

Example 9-10. Private attribute names are "mangled" by prefixing the _ and the class name

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

Name mangling is about safety, not security: it's designed to prevent accidental access and not intentional wrongdoing (Figure 9-1 illustrates another safety device).

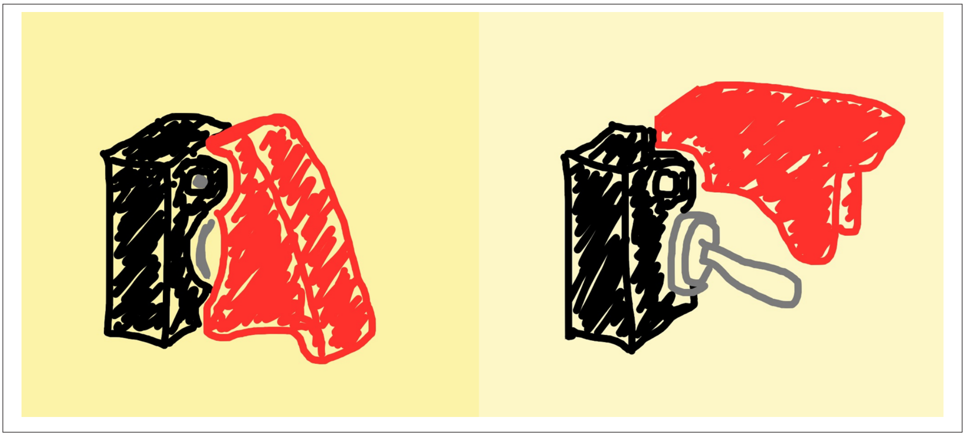

Figure 9-1. A cover on a switch is a safety device, not a security one: it prevents accidental activation, not malicious use

Anyone who knows how private names are mangled can read the private attribute directly, as the last line of Example 9-10 shows—that's actually useful for debugging and serialization. They can also directly assign a value to a private component of a Vector2d by simply writing v1._Vector__x = 7. But if you are doing that in production code, you can't complain if something blows up.

The name mangling functionality is not loved by all Pythonistas, and neither is the skewed look of names written as self.__x. Some prefer to avoid this syntax and use just one underscore prefix to "protect" attributes by convention (e.g., self._x). Critics of the automatic double-underscore mangling suggest that concerns about accidental attribute clobbering should be addressed by naming conventions. This is the full quote from the prolific Ian Bicking, cited at the beginning of this chapter:

> Never, ever use two leading underscores. This is annoyingly private. If name clashes are a concern, use explicit name mangling instead (e.g., `_MyThing_blahblah`). This is essentially the same thing as double-underscore, only it's transparent where double underscore obscures.[7]

The single underscore prefix has no special meaning to the Python interpreter when used in attribute names, but it's a very strong convention among Python programmers that you should not access such attributes from outside the class.[8] It's easy to respect the privacy of an object that marks its attributes with a single `_`, just as it's easy respect the convention that variables in `ALL_CAPS` should be treated as constants.

Attributes with a single `_` prefix are called "protected" in some corners of the Python documentation.[9] The practice of "protecting" attributes by convention with the form `self._x` is widespread, but calling that a "protected" attribute is not so common. Some even call that a "private" attribute.

To conclude: the `Vector2d` components are "private" and our `Vector2d` instances are "immutable"—with scare quotes—because there is no way to make them really private and immutable.[10]

We'll now come back to our `Vector2d` class. In this final section, we cover a special attribute (not a method) that affects the internal storage of an object, with potentially huge impact on the use of memory but little effect on its public interface: `__slots__`.

Saving Space with the __slots__ Class Attribute

By default, Python stores instance attributes in a per-instance `dict` named `__dict__`. As we saw in "Practical Consequences of How dict Works" on page 90, dictionaries have a significant memory overhead because of the underlying hash table used to provide fast access. If you are dealing with millions of instances with few attributes, the `__slots__` class attribute can save a lot of memory, by letting the interpreter store the instance attributes in a `tuple` instead of a `dict`.

7. From the Paste Style Guide.
8. In modules, a single `_` in front of a top-level name does have an effect: if you write `from mymod import *` the names with a `_` prefix are not imported from `mymod`. However, you can still write `from mymod import _privatefunc`. This is explained in the Python Tutorial, section 6.1. More on Modules.
9. One example is in the gettext module docs.
10. If this state of affairs depresses you, and makes you wish Python was more like Java in this regard, don't read my discussion of the relative strength of the Java `private` modifier in "Soapbox" on page 272.

A `__slots__` attribute inherited from a superclass has no effect. Python only takes into account `__slots__` attributes defined in each class individually.

To define `__slots__`, you create a class attribute with that name and assign it an iterable of `str` with identifiers for the instance attributes. I like to use a `tuple` for that, because it conveys the message that the `__slots__` definition cannot change. See Example 9-11.

Example 9-11. vector2d_v3_slots.py: the slots attribute is the only addition to Vector2d

```
class Vector2d:
    __slots__ = ('__x', '__y')

    typecode = 'd'

    # methods follow (omitted in book listing)
```

By defining `__slots__` in the class, you are telling the interpreter: "These are all the instance attributes in this class." Python then stores them in a tuple-like structure in each instance, avoiding the memory overhead of the per-instance `__dict__`. This can make a huge difference in memory usage if your have millions of instances active at the same time.

If you are handling millions of objects with numeric data, you should really be using NumPy arrays (see "NumPy and SciPy" on page 52), which are not only memory-efficient but have highly optimized functions for numeric processing, many of which operate on the entire array at once. I designed the `Vector2d` class just to provide context when discussing special methods, because I try to avoid vague `foo` and `bar` examples when I can.

Example 9-12 shows two runs of a script that simply builds a `list`, using a list comprehension, with 10,000,000 instances of `Vector2d`. The *mem_test.py* script takes the name of a module with a `Vector2d` class variant as command-line argument. In the first run, I am using `vector2d_v3.Vector2d` (from Example 9-7); in the second run, the `__slots__` version of `vector2d_v3_slots.Vector2d` is used.

Example 9-12. mem_test.py creates 10 million Vector2d instances using the class defined in the named module (e.g., vector2d_v3.py)

```
$ time python3 mem_test.py vector2d_v3.py
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,623,808
```

```
  Final RAM usage:  1,558,482,944

real  0m16.721s
user  0m15.568s
sys 0m1.149s
$ time python3 mem_test.py vector2d_v3_slots.py
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,718,016
  Final RAM usage:    655,466,496

real  0m13.605s
user  0m13.163s
sys 0m0.434s
```

As Example 9-12 reveals, the RAM footprint of the script grows to 1.5 GB when instance `__dict__` is used in each of the 10 million `Vector2d` instances, but that is reduced to 655 MB when `Vector2d` has a `__slots__` attribute. The `__slots__` version is also faster. The *mem_test.py* script in this test basically deals with loading a module, checking memory usage, and formatting results. The code is not really relevant here so it's in Appendix A, Example A-4.

> When `__slots__` is specified in a class, its instances will not be allowed to have any other attributes apart from those named in `__slots__`. This is really a side effect, and not the reason why `__slots__` exists. It's considered bad form to use `__slots__` just to prevent users of your class from creating new attributes in the instances if they want to. `__slots__` should used for optimization, not for programmer restraint.

It may be possible, however, to "save memory and eat it too": if you add the `'__dict__'` name to the `__slots__` list, your instances will keep attributes named in `__slots__` in the per-instance tuple, but will also support dynamically created attributes, which will be stored in the usual `__dict__`. Of course, having `'__dict__'` in `__slots__` may entirely defeat its purpose, depending on the number of static and dynamic attributes in each instance and how they are used. Careless optimization is even worse than premature optimization.

There is another special per-instance attribute that you may want to keep: the `__weakref__` attribute is necessary for an object to support weak references (covered in "Weak References" on page 236). That attribute is present by default in instances of user-defined classes. However, if the class defines `__slots__`, and you need the instances to be targets of weak references, then you need to include `'__weakref__'` among the attributes named in `__slots__`.

To summarize, `__slots__` has some caveats and should not be abused just for the sake of limiting what attributes can be assigned by users. It is mostly useful when working with tabular data such as database records where the schema is fixed by definition and the datasets may be very large. However, if you do this kind of work often, you must check out not only NumPy, but also the pandas data analysis library, which can handle nonnumeric data and import/export to many different tabular data formats.

The Problems with __slots__

To summarize, `__slots__` may provide significant memory savings if properly used, but there are a few caveats:

- You must remember to redeclare `__slots__` in each subclass, because the inherited attribute is ignored by the interpreter.
- Instances will only be able to have the attributes listed in `__slots__`, unless you include `'__dict__'` in `__slots__` (but doing so may negate the memory savings).
- Instances cannot be targets of weak references unless you remember to include `'__weakref__'` in `__slots__`.

If your program is not handling millions of instances, it's probably not worth the trouble of creating a somewhat unusual and tricky class whose instances may not accept dynamic attributes or may not support weak references. Like any optimization, `__slots__` should be used only if justified by a present need and when its benefit is proven by careful profiling.

The last topic in this chapter has to do with overriding a class attribute in instances and subclasses.

Overriding Class Attributes

A distinctive feature of Python is how class attributes can be used as default values for instance attributes. In `Vector2d` there is the `typecode` class attribute. It's used twice in the `__bytes__` method, but we read it as `self.typecode` by design. Because `Vector2d` instances are created without a `typecode` attribute of their own, `self.typecode` will get the `Vector2d.typecode` class attribute by default.

But if you write to an instance attribute that does not exist, you create a new instance attribute—e.g., a `typecode` instance attribute—and the class attribute by the same name is untouched. However, from then on, whenever the code handling that instance reads `self.typecode`, the instance `typecode` will be retrieved, effectively shadowing the class attribute by the same name. This opens the possibility of customizing an individual instance with a different `typecode`.

The default `Vector2d.typecode` is `'d'`, meaning each vector component will be represented as an 8-byte double precision float when exporting to `bytes`. If we set the `typecode` of a `Vector2d` instance to `'f'` prior to exporting, each component will be exported as a 4-byte single precision float. Example 9-13 demonstrates.

> We are discussing adding a custom instance attribute, therefore Example 9-13 uses the `Vector2d` implementation without `__slots__` as listed in Example 9-9.

Example 9-13. Customizing an instance by setting the typecode attribute that was formerly inherited from the class

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
>>> len(dumpd)  # ❶
17
>>> v1.typecode = 'f'  # ❷
>>> dumpf = bytes(v1)
>>> dumpf
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf)  # ❸
9
>>> Vector2d.typecode  # ❹
'd'
```

❶ Default `bytes` representation is 17 bytes long.

❷ Set `typecode` to `'f'` in the `v1` instance.

❸ Now the `bytes` dump is 9 bytes long.

❹ `Vector2d.typecode` is unchanged; only the `v1` instance uses `typecode` `'f'`.

Now it should be clear why the `bytes` export of a `Vector2d` is prefixed by the `typecode`: we wanted to support different export formats.

If you want to change a class attribute you must set it on the class directly, not through an instance. You could change the default `typecode` for all instances (that don't have their own `typecode`) by doing this:

```
>>> Vector2d.typecode = 'f'
```

However, there is an idiomatic Python way of achieving a more permanent effect, and being more explicit about the change. Because class attributes are public, they are inherited by subclasses, so it's common practice to subclass just to customize a class data attribute. The Django class-based views use this technique extensively. Example 9-14 shows how.

Example 9-14. The ShortVector2d is a subclass of Vector2d, which only overwrites the default typecode

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d):  # ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27)  # ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035)  # ❸
>>> len(bytes(sv))  # ❹
9
```

❶ Create `ShortVector2d` as a `Vector2d` subclass just to overwrite the `typecode` class attribute.

❷ Build `ShortVector2d` instance `sv` for demonstration.

❸ Inspect the `repr` of `sv`.

❹ Check that the length of the exported bytes is 9, not 17 as before.

This example also explains why I did not hardcode the `class_name` in `Vector2d.__repr__`, but instead got it from `type(self).__name__`, like this:

```
    # inside class Vector2d:

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)
```

If I had hardcoded the `class_name`, subclasses of `Vector2d` like `ShortVector2d` would have to overwrite `__repr__` just to change the `class_name`. By reading the name from the `type` of the instance, I made `__repr__` safer to inherit.

This ends our coverage of implementing a simple class that leverages the data model to play well with the rest of Python—offering different object representations, implementing a custom formatting code, exposing read-only attributes, and supporting `hash()` to integrate with sets and mappings.

Chapter Summary

The aim of this chapter was to demonstrate the use of special methods and conventions in the construction of a well-behaved Pythonic class.

Is *vector2d_v3.py* (Example 9-9) more Pythonic than *vector2d_v0.py* (Example 9-2)? The `Vector2d` class in *vector2d_v3.py* certainly exhibits more Python features. But whether the first or the last `Vector2d` implementation is more idiomatic depends on the context where it would be used. Tim Peter's Zen of Python says:

> Simple is better than complex.

A Pythonic object should be as simple as the requirements allow—and not a parade of language features.

But my goal in expanding the `Vector2d` code was to provide context for discussing Python special methods and coding conventions. If you look back at Table 1-1, the several listings in this chapter demonstrated:

- All string/bytes representation methods: `__repr__`, `__str__`, `__format__`, and `__bytes__`.
- Several methods for converting an object to a number: `__abs__`, `__bool__`, `__hash__`.
- The `__eq__` operator, to test `bytes` conversion and to enable hashing (along with `__hash__`).

While supporting conversion to `bytes` we also implemented an alternative constructor, `Vector2d.frombytes()`, which provided the context for discussing the decorators `@classmethod` (very handy) and `@staticmethod` (not so useful, module-level functions are simpler). The `frombytes` method was inspired by it's namesake in the `array.array` class.

We saw that the Format Specification Mini-Language is extensible by implementing a `__format__` method that does some minimal parsing of `format_spec` provided to the `format(obj, format_spec)` built-in or within replacement fields `'{:«format_spec»}'` in strings used with the `str.format` method.

In preparation to make `Vector2d` instances hashable, we made an effort to make them immutable, at least preventing accidental changes by coding the `x` and `y` attributes as private, and exposing them as read-only properties. We then implemented `__hash__` using the recommended technique of xor-ing the hashes of the instance attributes.

We then discussed the memory savings and the caveats of declaring a `__slots__` attribute in `Vector2d`. Because using `__slots__` is somewhat tricky, it really makes sense only when handling a very large number of instances—think millions of instances, not just thousands.

The last topic we covered was the overriding of a class attribute accessed via the instances (e.g., `self.typecode`). We did that first by creating an instance attribute, and then by subclassing and overwriting at the class level.

Throughout the chapter, I mentioned how design choices in the examples were informed by studying the API of standard Python objects. If this chapter can be summarized in one sentence, this is it:

> To build Pythonic objects, observe how real Python objects behave.
>
> — Ancient Chinese proverb

Further Reading

This chapter covered several special methods of the data model, so naturally the primary references are the same as the ones provided in Chapter 1, which gave a high-level view of the same topic. For convenience, I'll repeat those four earlier recommendations here, and add a few other ones:

"Data Model" chapter of The Python Language Reference
: Most of the methods we used in this chapter are documented in "3.3.1. Basic customization".

Python in a Nutshell, 2nd Edition, by Alex Martelli
: Excellent coverage of the data model, even if only Python 2.5 is covered (in the second edition). The fundamental concepts are all the same and most of the Data Model APIs haven't changed at all since Python 2.2, when built-in types and user-defined classes became more compatible.

Python Cookbook, 3rd Edition, by David Beazley and Brian K. Jones
: Very modern coding practices demonstrated through recipes. Chapter 8, "Classes and Objects" in particular has several solutions related to discussions in this chapter.

Python Essential Reference, 4th Edition, by David Beazley
: Covers the data model in detail in the context of Python 2.6 and Python 3.

In this chapter, we covered every special method related to object representation, except `__index__`. It's used to coerce an object to an integer index in the specific context of sequence slicing, and was created to solve a need in NumPy. In practice, you and I are not likely to need to implement `__index__` unless we decide to write a new numeric data type, and we want it to be usable as arguments to `__getitem__`. If you are curious about it, A.M. Kuchling's What's New in Python 2.5 has a short explanation, and PEP 357 — Allowing Any Object to be Used for Slicing details the need for `__index__`, from the perspective of an implementor of a C-extension, Travis Oliphant, the lead author of NumPy.

An early realization of the need for distinct string representations for objects appeared in Smalltalk. The 1996 article "How to Display an Object as a String: printString and displayString" by Bobby Woolf discusses the implementation of the `printString` and `displayString` methods in that language. From that article, I borrowed the pithy descriptions "the way the developer wants to see it" and "the way the user wants to see it" when defining `repr()` and `str()` in "Object Representations" on page 248.

Soapbox

Properties Help Reduce Upfront Costs

In the initial versions of `Vector2d`, the `x` and `y` attributes were public, as are all Python instance and class attributes by default. Naturally, users of vectors need to be able to access its components. Although our vectors are iterable and can be unpacked into a pair of variables, it's also desirable to be able to write `my_vector.x` and `my_vector.y` to get each component.

When we felt the need to avoid accidental updates to the `x` and `y` attributes, we implemented properties, but nothing changed elsewhere in the code and in the public interface of `Vector2d`, as verified by the doctests. We are still able to access `my_vector.x` and `my_vector.y`.

This shows that we can always start our classes in the simplest possible way, with public attributes, because when (or if) we later need to impose more control with getters and setters, these can be implemented through properties without changing any of the code that already interacts with our objects through the names (e.g., `x` and `y`) that were initially simple public attributes.

This approach is the opposite of that encouraged by the Java language: a Java programmer cannot start with simple public attributes and only later, if needed, implement properties, because they don't exist in the language. Therefore, writing getters and setters is the norm in Java—even when those methods do nothing useful—because the API cannot evolve from simple public attributes to getters and setters without breaking all code that uses those attributes.

In addition, as our technical reviewer Alex Martelli points out, typing getter/setter calls everywhere is goofy. You have to write stuff like:

```
>>> my_object.set_foo(my_object.get_foo() + 1)
```

Just to do this:

```
>>> my_object.foo += 1
```

Ward Cunningham, inventor of the wiki and an Extreme Programming pioneer, recommends asking "What's the simplest thing that could possibly work?" The idea is to focus on the goal.[11] Implementing setters and getters up front is a distraction from the goal. In Python, we can simply use public attributes knowing we can change them to properties later, if the need arises.

Safety Versus Security in Private Attributes

> Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.
>
> — Larry Wall
> *Creator of Perl*

Python and Perl are polar opposites in many regards, but Larry and Guido seem to agree on object privacy.

Having taught Python to many Java programmers over the years, I've found a lot of them put too much faith in the privacy guarantees that Java offers. As it turns out, the Java `private` and `protected` modifiers normally provide protection against accidents only (i.e., safety). They can only guarantee security against malicious intent if the application is deployed with a security manager, and that seldom happens in practice, even in corporate settings.

To prove my point, I like to show this Java class (Example 9-15).

Example 9-15. Confidential.java: a Java class with a private field named secret

```java
public class Confidential {

    private String secret = "";

    public Confidential(String text) {
        secret = text.toUpperCase();
    }
}
```

In Example 9-15, I store the `text` in the `secret` field after converting it to uppercase, just to make it obvious that whatever is in that field will be in all caps.

The actual demonstration consists of running *expose.py* with Jython. That script uses introspection ("reflection" in Java parlance) to get the value of a private field. The code is in Example 9-16.

Example 9-16. expose.py: Jython code to read the content of a private field in another class

```python
import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
```

11. See "Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V".

```
secret_field.setAccessible(True)  # break the lock!
print 'message.secret =', secret_field.get(message)
```

If you run Example 9-16, this is what you get:

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

The string `'TOP SECRET TEXT'` was read from the `secret` private field of the `Confidential` class.

There is no black magic here: *expose.py* uses the Java reflection API to get a reference to the private field named `'secret'`, and then calls `'secret_field.setAccessible(True)'` to make it readable. The same thing can be done with Java code, of course (but it takes more than three times as many lines to do it; see the file *Expose.java* in the *Fluent Python* code repository).

The crucial call `.setAccessible(True)` will fail only if the Jython script or the Java main program (e.g., `Expose.class`) is running under the supervision of a SecurityManager. But in the real world, Java applications are rarely deployed with a SecurityManager—except for Java applets (remember those?).

My point is: in Java too, access control modifiers are mostly about safety and not security, at least in practice. So relax and enjoy the power Python gives you. Use it responsibly.