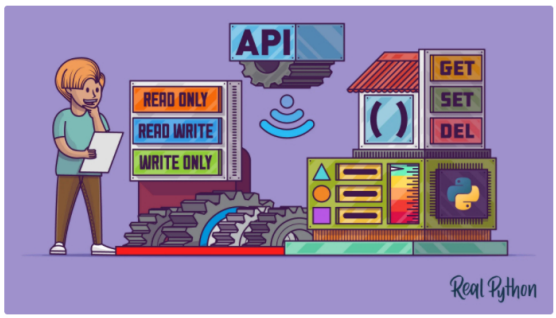

Python's property(): Add Managed Attributes to Your Classes

by Leodanis Pozo Ramos Oct 13, 2021 4 Comments best-practices intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

With Python's `property()`, you can create **managed attributes** in your classes. You can use managed attributes, also known as **properties**, when you need to modify their internal implementation without changing the public API of the class. Providing stable APIs can help you avoid breaking your users' code when they rely on your classes and objects.

Properties are arguably the most popular way to create managed attributes quickly and in the purest Pythonic style.

In this tutorial, you'll learn how to:

- Create **managed attributes** or **properties** in your classes
- Perform **lazy attribute evaluation** and provide **computed attributes**
- Avoid **setter** and **getter** methods to make your classes more Pythonic
- Create **read-only**, **read-write**, and **write-only** properties
- Create consistent and **backward-compatible APIs** for your classes

You'll also write a few practical examples that use `property()` for validating input data, computing attribute values dynamically, logging your code, and more. To get the most out of this tutorial, you should know the basics of object-oriented programming and decorators in Python.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

Managing Attributes in Your Classes

When you define a class in an object-oriented programming language, you'll probably end up with some instance and class attributes. In other words, you'll end up with variables that are accessible through the instance, class, or even both, depending on the language. Attributes represent or hold the internal state of a given object, which you'll often need to access and mutate.

Typically, you have at least two ways to manage an attribute. Either you can access and mutate the attribute directly or you can use **methods**. Methods are functions attached to a given class. They provide the behaviors and actions that an object can perform with its internal data and attributes.

If you expose your attributes to the user, then they become part of the public API of your classes. Your user will access and mutate them directly in their code. The problem comes when you need to change the internal implementation of a given attribute.

Table of Contents
- Managing Attributes in Your Classes
- Getting Started With Python's property()
- Providing Read-Only Attributes
- Creating Read-Write Attributes
- Providing Write-Only Attributes
- Putting Python's property() Into Action
- Overriding Properties in Subclasses
- Conclusion

Mark as Completed
Tweet Share Email

Say you're working on a `Circle` class. The initial implementation has a single attribute called `.radius`. You finish coding the class and make it available to your end users. They start using `Circle` in their code to create a lot of awesome projects and applications. Good job!

Now suppose that you have an important user that comes to you with a new requirement. They don't want `Circle` to store the radius any longer. They need a public `.diameter` attribute.

At this point, removing `.radius` to start using `.diameter` could break the code of some of your end users. You need to manage this situation in a way other than removing `.radius`.

Programming languages such as Java and C++ encourage you to never expose your attributes to avoid this kind of problem. Instead, you should provide **getter** and **setter** methods, also known as accessors and mutators, respectively. These methods offer a way to change the internal implementation of your attributes without changing your public API.

> **Note:** Getter and setter methods are often considered an anti-pattern and a signal of poor object-oriented design. The main argument behind this proposition is that these methods break encapsulation. They allow you to access and mutate the components of your objects.

In the end, these languages need getter and setter methods because they don't provide a suitable way to change the internal implementation of an attribute if a given requirement changes. Changing the internal implementation would require an API modification, which can break your end users' code.

The Getter and Setter Approach in Python

Technically, there's nothing that stops you from using getter and setter methods in Python. Here's how this approach would look:

Python

```
# point.py

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def get_x(self):
        return self._x

    def set_x(self, value):
        self._x = value

    def get_y(self):
        return self._y

    def set_y(self, value):
        self._y = value
```

In this example, you create `Point` with two **non-public attributes** `._x` and `._y` to hold the Cartesian coordinates of the point at hand.

> **Note:** Python doesn't have the notion of access modifiers, such as `private`, `protected`, and `public`, to restrict access to attributes and methods. In Python, the distinction is between **public** and **non-public** class members.
>
> If you want to signal that a given attribute or method is non-public, then you have to use the well-known Python convention of prefixing the name with an underscore (_). That's the reason behind the naming of the attributes `._x` and `._y`.
>
> Note that this is just a convention. It doesn't stop you and other programmers from accessing the attributes using **dot notation**, as in `obj._attr`. However, it's bad practice to violate this convention.

To access and mutate the value of either `._x` or `._y`, you can use the corresponding getter and setter methods. Go ahead and save the above definition of `Point` in a Python module and import the class into your interactive shell.

Here's how you can work with `Point` in your code:

Python >>>

```
>>> from point import Point

>>> point = Point(12, 5)
>>> point.get_x()
12
>>> point.get_y()
5

>>> point.set_x(42)
>>> point.get_x()
42

>>> # Non-public attributes are still accessible
>>> point._x
42
>>> point._y
5
```

With `.get_x()` and `.get_y()`, you can access the current values of `._x` and `._y`. You can use the setter method to store a new value in the corresponding managed attribute. From this code, you can confirm that Python doesn't restrict access to non-public attributes. Whether or not you do so is up to you.

The Pythonic Approach

Even though the example you just saw uses the Python coding style, it doesn't look Pythonic. In the example, the getter and setter methods don't perform any further processing with `._x` and `._y`. You can rewrite `Point` in a more concise and Pythonic way:

```
Python >>>
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> point = Point(12, 5)
>>> point.x
12
>>> point.y
5

>>> point.x = 42
>>> point.x
42
```

This code uncovers a fundamental principle. Exposing attributes to the end user is normal and common in Python. You don't need to clutter your classes with getter and setter methods all the time, which sounds pretty cool! However, how can you handle requirement changes that would seem to involve API changes?

Unlike Java and C++, Python provides handy tools that allow you to change the underlying implementation of your attributes without changing your public API. The most popular approach is to turn your attributes into **properties**.

> **Note:** Another common approach to provide managed attributes is to use descriptors. In this tutorial, however, you'll learn about properties.

Properties represent an intermediate functionality between a plain attribute (or field) and a method. In other words, they allow you to create methods that behave like attributes. With properties, you can change how you compute the target attribute whenever you need to do so.

For example, you can turn both `.x` and `.y` into properties. With this change, you can continue accessing them as attributes. You'll also have an underlying method holding `.x` and `.y` that will allow you to modify their internal implementation and perform actions on them right before your users access and mutate them.

> **Note:** Properties aren't exclusive to Python. Languages such as JavaScript, C#, Kotlin, and others also provide tools and techniques for creating properties as class members.

The main advantage of Python properties is that they allow you to expose your attributes as part of your public API. If you ever need to change the underlying implementation, then you can turn the attribute into a property at any time without much pain.

In the following sections, you'll learn how to create properties in Python.

Getting Started With Python's `property()`

Python's `property()` is the Pythonic way to avoid formal getter and setter methods in your code. This function allows you to turn class attributes into **properties** or **managed attributes**. Since `property()` is a built-in function, you can use it without importing anything. Additionally, `property()` was implemented in C to ensure optimal performance.

> **Note:** It's common to refer to `property()` as a built-in function. However, `property` is a class designed to work as a function rather than as a regular class. That's why most Python developers call it a function. That's also the reason why `property()` doesn't follow the Python convention for naming classes.
>
> This tutorial follows the common practices of calling `property()` a function rather than a class. However, in some sections, you'll see it called a class to facilitate the explanation.

With `property()`, you can attach getter and setter methods to given class attributes. This way, you can handle the internal implementation for that attribute without exposing getter and setter methods in your API. You can also specify a way to handle attribute deletion and provide an appropriate docstring for your properties.

Here's the full signature of `property()`:

```
Python
property(fget=None, fset=None, fdel=None, doc=None)
```

The first two arguments take function objects that will play the role of getter (`fget`) and setter (`fset`) methods. Here's a summary of what each argument does:

Argument	Description
`fget`	Function that returns the value of the managed attribute
`fset`	Function that allows you to set the value of the managed attribute
`fdel`	Function to define how the managed attribute handles deletion
`doc`	String representing the property's docstring

The return value of `property()` is the managed attribute itself. If you access the managed attribute, as in `obj.attr`, then Python automatically calls `fget()`. If you assign a new value to the attribute, as in `obj.attr = value`, then Python calls `fset()` using the input `value` as an argument. Finally, if you run a `del obj.attr` statement, then Python automatically calls `fdel()`.

> **Note:** The first three arguments to `property()` take function objects. You can think of a function object as the function name without the calling pair of parentheses.

You can use `doc` to provide an appropriate docstring for your properties. You and your fellow programmers will be able to read that docstring using Python's `help()`. The `doc` argument is also useful when you're working with code editors and IDEs that support docstring access.

You can use `property()` either as a function or a decorator to build your properties. In the following two sections, you'll learn how to use both approaches. However, you should know up front that the decorator approach is more popular in the Python community.

Creating Attributes With `property()`

You can create a property by calling `property()` with an appropriate set of arguments and assigning its return value to a class attribute. All the arguments to `property()` are optional. However, you typically provide at least a **setter function**.

The following example shows how to create a `Circle` class with a handy property to manage its radius:

Python

```
# circle.py

class Circle:
    def __init__(self, radius):
        self._radius = radius

    def _get_radius(self):
        print("Get radius")
        return self._radius

    def _set_radius(self, value):
        print("Set radius")
        self._radius = value

    def _del_radius(self):
        print("Delete radius")
        del self._radius

    radius = property(
        fget=_get_radius,
        fset=_set_radius,
        fdel=_del_radius,
        doc="The radius property."
    )
```

In this code snippet, you create `Circle`. The class initializer, `.__init__()`, takes `radius` as an argument and stores it in a non-public attribute called `._radius`. Then you define three non-public methods:

1. **`._get_radius()`** returns the current value of `._radius`
2. **`._set_radius()`** takes `value` as an argument and assigns it to `._radius`
3. **`._del_radius()`** deletes the instance attribute `._radius`

Once you have these three methods in place, you create a class attribute called `.radius` to store the property object. To initialize the property, you pass the three methods as arguments to `property()`. You also pass a suitable docstring for your property.

In this example, you use keyword arguments to improve the code readability and prevent confusion. That way, you know exactly which method goes into each argument.

To give `Circle` a try, run the following code in your Python shell:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42.0)

>>> circle.radius
Get radius
42.0

>>> circle.radius = 100.0
Set radius
>>> circle.radius
Get radius
100.0

>>> del circle.radius
Delete radius
>>> circle.radius
Get radius
Traceback (most recent call last):
    ...
AttributeError: 'Circle' object has no attribute '_radius'

>>> help(circle)
Help on Circle in module __main__ object:

class Circle(builtins.object)
    ...
 |  radius
 |      The radius property.
```

The `.radius` property hides the non-public instance attribute `._radius`, which is now your managed attribute in this example. You can access and assign `.radius` directly. Internally, Python automatically calls `._get_radius()` and `._set_radius()` when needed. When you execute `del circle.radius`, Python calls `._del_radius()`, which deletes the underlying

Execute `del Circle.radius`, Python calls `._del_radius()`, which deletes the underlying `._radius`.

Using `lambda` Functions as Getter Methods Show/Hide

Properties are **class attributes** that manage **instance attributes**. You can think of a property as a collection of methods bundled together. If you inspect `.radius` carefully, then you can find the raw methods you provided as the `fget`, `fset`, and `fdel` arguments:

Python >>>

```
>>> from circle import Circle

>>> Circle.radius.fget
<function Circle._get_radius at 0x7fba7e1d7d30>

>>> Circle.radius.fset
<function Circle._set_radius at 0x7fba7e1d78b0>

>>> Circle.radius.fdel
<function Circle._del_radius at 0x7fba7e1d7040>

>>> dir(Circle.radius)
[..., '__get__', ..., '__set__', ...]
```

You can access the getter, setter, and deleter methods in a given property through the corresponding `.fget`, `.fset`, and `.fdel`.

Properties are also **overriding descriptors**. If you use `dir()` to check the internal members of a given property, then you'll find `.__set__()` and `.__get__()` in the list. These methods provide a default implementation of the descriptor protocol.

> **Note:** If you want to better understand the internal implementation of `property` as a class, then check out the pure Python `Property` class described in the documentation.

The default implementation of `.__set__()`, for example, runs when you don't provide a custom setter method. In this case, you get an `AttributeError` because there's no way to set the underlying property.

Using `property()` as a Decorator

Decorators are everywhere in Python. They're functions that take another function as an argument and return a new function with added functionality. With a decorator, you can attach pre- and post-processing operations to an existing function.

When Python 2.2 introduced `property()`, the decorator syntax wasn't available. The only way to define properties was to pass getter, setter, and deleter methods, as you learned before. The decorator syntax was added in Python 2.4, and nowadays, using `property()` as a decorator is the most popular practice in the Python community.

The decorator syntax consists of placing the name of the decorator function with a leading `@` symbol right before the definition of the function you want to decorate:

Python

```
@decorator
def func(a):
    return a
```

In this code fragment, `@decorator` can be a function or class intended to decorate `func()`. This syntax is equivalent to the following:

Python

```
def func(a):
    return a

func = decorator(func)
```

The final line of code reassigns the name `func` to hold the result of calling `decorator(func)`. Note that this is the same syntax you used to create a property in the section above.

Python's `property()` can also work as a decorator, so you can use the `@property` syntax to create your properties quickly:

Python

```
# circle.py

class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """The radius property."""
        print("Get radius")
        return self._radius

    @radius.setter
    def radius(self, value):
        print("Set radius")
        self._radius = value

    @radius.deleter
    def radius(self):
        print("Delete radius")
        del self._radius
```

This code looks pretty different from the getter and setter methods approach. `Circle` now

looks more Pythonic and clean. You don't need to use method names such as `._get_radius()`, `._set_radius()`, and `._del_radius()` anymore. Now you have three methods with the same clean and descriptive attribute-like name. How is that possible?

The decorator approach for creating properties requires defining a first method using the public name for the underlying managed attribute, which is `.radius` in this case. This method should implement the getter logic. In the above example, lines 7 to 11 implement that method.

Lines 13 to 16 define the setter method for `.radius`. In this case, the syntax is fairly different. Instead of using `@property` again, you use `@radius.setter`. Why do you need to do that? Take another look at the `dir()` output:

```
Python >>>
>>> dir(Circle.radius)
[..., 'deleter', ..., 'getter', 'setter']
```

Besides `.fget`, `.fset`, `.fdel`, and a bunch of other special attributes and methods, `property` also provides `.deleter()`, `.getter()`, and `.setter()`. These three methods each return a new property.

When you decorate the second `.radius()` method with `@radius.setter` (line 13), you create a new property and reassign the class-level name `.radius` (line 8) to hold it. This new property contains the same set of methods of the initial property at line 8 with the addition of the new setter method provided on line 14. Finally, the decorator syntax reassigns the new property to the `.radius` class-level name.

The mechanism to define the deleter method is similar. This time, you need to use the `@radius.deleter` decorator. At the end of the process, you get a full-fledged property with the getter, setter, and deleter methods.

Finally, how can you provide suitable docstrings for your properties when you use the decorator approach? If you check `Circle` again, you'll note that you already did so by adding a docstring to the getter method on line 9.

The new `Circle` implementation works the same as the example in the section above:

```
Python >>>
>>> from circle import Circle

>>> circle = Circle(42.0)

>>> circle.radius
Get radius
42.0

>>> circle.radius = 100.0
Set radius
>>> circle.radius
Get radius
100.0

>>> del circle.radius
Delete radius
>>> circle.radius
Get radius
Traceback (most recent call last):
    ...
AttributeError: 'Circle' object has no attribute '_radius'

>>> help(circle)
Help on Circle in module __main__ object:

class Circle(builtins.object)
    ...
 |  radius
 |      The radius property.
```

You don't need to use a pair of parentheses for calling `.radius()` as a method. Instead, you can access `.radius` as you would access a regular attribute, which is the primary use of properties. They allow you to treat methods as attributes, and they take care of calling the underlying set of methods automatically.

Here's a recap of some important points to remember when you're creating properties with the decorator approach:

- The `@property` decorator must decorate the **getter method**.
- The docstring must go in the **getter method**.
- The **setter and deleter methods** must be decorated with the name of the getter method plus `.setter` and `.deleter`, respectively.

Up to this point, you've created managed attributes using `property()` as a function and as a decorator. If you check your `Circle` implementations so far, then you'll note that their getter and setter methods don't add any real extra processing on top of your attributes.

In general, you should avoid turning attributes that don't require extra processing into properties. Using properties in those situations can make your code:

- Unnecessarily verbose
- Confusing to other developers
- Slower than code based on regular attributes

Unless you need something more than bare attribute access, don't write properties. They're a waste of CPU time, and more importantly, they're a waste of *your* time. Finally, you should avoid writing explicit getter and setter methods and then wrapping them in a property. Instead, use the `@property` decorator. That's currently the most Pythonic way to go.

Providing Read-Only Attributes

Probably the most elementary use case of `property()` is to provide **read-only attributes** in your classes. Say you need an immutable `Point` class that doesn't allow the user to mutate the original value of its coordinates, `x` and `y`. To achieve this goal, you can create `Point` like in the following example:

Python

```
# point.py

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

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

    @property
    def y(self):
        return self._y
```

Here, you store the input arguments in the attributes `._x` and `._y`. As you already learned, using the leading underscore (_) in names tells other developers that they're non-public attributes and shouldn't be accessed using dot notation, such as in `point._x`. Finally, you define two getter methods and decorate them with `@property`.

Now you have two read-only properties, `.x` and `.y`, as your coordinates:

Python >>>

```
>>> from point import Point

>>> point = Point(12, 5)

>>> # Read coordinates
>>> point.x
12
>>> point.y
5

>>> # Write coordinates
>>> point.x = 42
Traceback (most recent call last):
    ...
AttributeError: can't set attribute
```

Here, `point.x` and `point.y` are bare-bone examples of read-only properties. Their behavior relies on the underlying descriptor that `property` provides. As you already saw, the default `.__set__()` implementation raises an `AttributeError` when you don't define a proper setter method.

You can take this implementation of `Point` a little bit further and provide explicit setter methods that raise a custom exception with more elaborate and specific messages:

Python

```
# point.py

class WriteCoordinateError(Exception):
    pass

class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

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

    @x.setter
    def x(self, value):
        raise WriteCoordinateError("x coordinate is read-only")

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

    @y.setter
    def y(self, value):
        raise WriteCoordinateError("y coordinate is read-only")
```

In this example, you define a custom exception called `WriteCoordinateError`. This exception allows you to customize the way you implement your immutable `Point` class. Now, both setter methods raise your custom exception with a more explicit message. Go ahead and give your improved `Point` a try!

Creating Read-Write Attributes

You can also use `property()` to provide managed attributes with **read-write** capabilities. In practice, you just need to provide the appropriate getter method ("read") and setter method ("write") to your properties in order to create read-write managed attributes.

Say you want your `Circle` class to have a `.diameter` attribute. However, taking the radius and the diameter in the class initializer seems unnecessary because you can compute the one using the other. Here's a `Circle` that manages `.radius` and `.diameter` as read-write attributes:

Python

```
# circle.py

import math

class Circle:
    def __init__(self, radius):
        self.radius = radius
```

```
    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        self._radius = float(value)

    @property
    def diameter(self):
        return self.radius * 2

    @diameter.setter
    def diameter(self, value):
        self.radius = value / 2
```

Here, you create a `Circle` class with a read-write `.radius`. In this case, the getter method just returns the radius value. The setter method converts the input value for the radius and assigns it to the non-public `._radius`, which is the variable you use to store the final data.

There is a subtle detail to note in this new implementation of `Circle` and its `.radius` attribute. In this case, the class initializer assigns the input value to the `.radius` property directly instead of storing it in a dedicated non-public attribute, such as `._radius`.

Why? Because you need to make sure that every value provided as a radius, including the initialization value, goes through the setter method and gets converted to a floating-point number.

`Circle` also implements a `.diameter` attribute as a property. The getter method computes the diameter using the radius. The setter method does something curious. Instead of storing the input diameter `value` in a dedicated attribute, it calculates the radius and writes the result into `.radius`.

Here's how your `Circle` works:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42)
>>> circle.radius
42.0

>>> circle.diameter
84.0

>>> circle.diameter = 100
>>> circle.diameter
100.0

>>> circle.radius
50.0
```

Both `.radius` and `.diameter` work as normal attributes in these examples, providing a clean and Pythonic public API for your `Circle` class.

Providing Write-Only Attributes

You can also create **write-only** attributes by tweaking how you implement the getter method of your properties. For example, you can make your getter method raise an exception every time a user accesses the underlying attribute value.

Here's an example of handling passwords with a write-only property:

Python

```
# users.py

import hashlib
import os

class User:
    def __init__(self, name, password):
        self.name = name
        self.password = password

    @property
    def password(self):
        raise AttributeError("Password is write-only")

    @password.setter
    def password(self, plaintext):
        salt = os.urandom(32)
        self._hashed_password = hashlib.pbkdf2_hmac(
            "sha256", plaintext.encode("utf-8"), salt, 100_000
        )
```

The initializer of `User` takes a username and a password as arguments and stores them in `.name` and `.password`, respectively. You use a property to manage how your class processes the input password. The getter method raises an `AttributeError` whenever a user tries to retrieve the current password. This turns `.password` into a write-only attribute:

Python >>>

```
>>> from users import User

>>> john = User("John", "secret")

>>> john._hashed_password
b'b\xc7^ai\x9f3\xd2g ... \x89^-\x92\xbe\xe6'

>>> john.password
Traceback (most recent call last):
    ...
```

```
AttributeError: Password is write-only

>>> john.password = "supersecret"
>>> john._hashed_password
b'\xe91$\x9f\xaf\x9d ... b\xe8\xc8\xfcaU\r_'
```

In this example, you create `john` as a `User` instance with an initial password. The setter method hashes the password and stores it in `._hashed_password`. Note that when you try to access `.password` directly, you get an `AttributeError`. Finally, assigning a new value to `.password` triggers the setter method and creates a new hashed password.

In the setter method of `.password`, you use `os.urandom()` to generate a 32-byte random string as your hashing function's salt. To generate the hashed password, you use `hashlib.pbkdf2_hmac()`. Then you store the resulting hashed password in the non-public attribute `._hashed_password`. Doing so ensures that you never save the plaintext password in any retrievable attribute.

Putting Python's `property()` Into Action

So far, you've learned how to use Python's `property()` built-in function to create managed attributes in your classes. You used `property()` as a function and as a decorator and learned about the differences between these two approaches. You also learned how to create read-only, read-write, and write-only attributes.

In the following sections, you'll code a few examples that will help you get a better practical understanding of common use cases of `property()`.

Validating Input Values

One of the most common use cases of `property()` is building managed attributes that validate the input data before storing or even accepting it as a secure input. Data validation is a common requirement in code that takes input from users or other information sources that you consider untrusted.

Python's `property()` provides a quick and reliable tool for dealing with input data validation. For example, thinking back to the `Point` example, you may require the values of `.x` and `.y` to be valid numbers. Since your users are free to enter any type of data, you need to make sure that your point only accepts numbers.

Here's an implementation of `Point` that manages this requirement:

Python

```python
# point.py

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

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

    @x.setter
    def x(self, value):
        try:
            self._x = float(value)
            print("Validated!")
        except ValueError:
            raise ValueError('"x" must be a number') from None

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

    @y.setter
    def y(self, value):
        try:
            self._y = float(value)
            print("Validated!")
        except ValueError:
            raise ValueError('"y" must be a number') from None
```

The setter methods of `.x` and `.y` use `try ... except` blocks that validate input data using the Python EAFP style. If the call to `float()` succeeds, then the input data is valid, and you get `Validated!` on your screen. If `float()` raises a `ValueError`, then the user gets a `ValueError` with a more specific message.

> **Note:** In the example above, you use the syntax `raise ... from None` to hide internal details related to the context in which you're raising the exception. From the end user's viewpoint, these details can be confusing and make your class look unpolished.
>
> Check out the section on the `raise` statement in the documentation for more information about this topic.

It's important to note that assigning the `.x` and `.y` properties directly in `.__init__()` ensures that the validation also occurs during object initialization. Not doing so is a common mistake when using `property()` for data validation.

Here's how your `Point` class works now:

Python >>>

```
>>> from point import Point

>>> point = Point(12, 5)
Validated!
Validated!
>>> point.x
12.0
>>> point.y
5.0

>>> point.x = 42
Validated!
```

```
>>> point.x
42.0

>>> point.y = 100.0
Validated!
>>> point.y
100.0

>>> point.x = "one"
Traceback (most recent call last):
    ...
ValueError: "x" must be a number

>>> point.y = "1o"
Traceback (most recent call last):
    ...
ValueError: "y" must be a number
```

If you assign `.x` and `.y` values that `float()` can turn into floating-point numbers, then the validation is successful, and the value is accepted. Otherwise, you get a `ValueError`.

This implementation of `Point` uncovers a fundamental weakness of `property()`. Did you spot it?

That's it! You have repetitive code that follows specific patterns. This repetition breaks the DRY (Don't Repeat Yourself) principle, so you would want to refactor this code to avoid it. To do so, you can abstract out the repetitive logic using a descriptor:

Python

```python
# point.py

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

    def __get__(self, instance, owner):
        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        try:
            instance.__dict__[self._name] = float(value)
            print("Validated!")
        except ValueError:
            raise ValueError(f'"{self._name}" must be a number') from None

class Point:
    x = Coordinate()
    y = Coordinate()

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

Now your code is a bit shorter. You managed to remove repetitive code by defining `Coordinate` as a descriptor that manages your data validation in a single place. The code works just like your earlier implementation. Go ahead and give it a try!

In general, if you find yourself copying and pasting property definitions all around your code or if you spot repetitive code like in the example above, then you should consider using a proper descriptor.

Providing Computed Attributes

If you need an attribute that builds its value dynamically whenever you access it, then `property()` is the way to go. These kinds of attributes are commonly known as **computed attributes**. They're handy when you need them to look like eager attributes, but you want them to be lazy.

The main reason for creating eager attributes is to optimize computation costs when you access the attribute often. On the other hand, if you rarely use a given attribute, then a lazy property can postpone its computation until needed, which can make your programs more efficient.

Here's an example of how to use `property()` to create a computed attribute `.area` in a `Rectangle` class:

Python

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height
```

In this example, the `Rectangle` initializer takes `width` and `height` as arguments and stores them in regular instance attributes. The read-only property `.area` computes and returns the area of the current rectangle every time you access it.

Another common use case of properties is to provide an auto-formatted value for a given attribute:

Python

```python
class Product:
    def __init__(self, name, price):
        self._name = name
        self._price = float(price)

    @property
```

```
    def price(self):
        return f"${self._price:,.2f}"
```

In this example, `.price` is a property that formats and returns the price of a particular product. To provide a currency-like format, you use an f-string with appropriate formatting options.

> **Note:** This example uses floating-point numbers to represent currencies, which is bad practice. Instead, you should use `decimal.Decimal` from the standard library.

As a final example of computed attributes, say you have a `Point` class that uses `.x` and `.y` as Cartesian coordinates. You want to provide polar coordinates for your point so that you can use them in a few computations. The polar coordinate system represents each point using the distance to the origin and the angle with the horizontal coordinate axis.

Here's a Cartesian coordinates `Point` class that also provides computed polar coordinates:

Python

```
# point.py

import math

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

    @property
    def distance(self):
        return round(math.dist((0, 0), (self.x, self.y)))

    @property
    def angle(self):
        return round(math.degrees(math.atan(self.y / self.x)), 1)

    def as_cartesian(self):
        return self.x, self.y

    def as_polar(self):
        return self.distance, self.angle
```

This example shows how to compute the distance and angle of a given `Point` object using its `.x` and `.y` Cartesian coordinates. Here's how this code works in practice:

Python >>>

```
>>> from point import Point

>>> point = Point(12, 5)

>>> point.x
12
>>> point.y
5

>>> point.distance
13
>>> point.angle
22.6

>>> point.as_cartesian()
(12, 5)
>>> point.as_polar()
(13, 22.6)
```

When it comes to providing computed or lazy attributes, `property()` is a pretty handy tool. However, if you're creating an attribute that you use frequently, then computing it every time can be costly and wasteful. A good strategy is to cache them once the computation is done.

Caching Computed Attributes

Sometimes you have a given computed attribute that you use frequently. Constantly repeating the same computation may be unnecessary and expensive. To work around this problem, you can cache the computed value and save it in a non-public dedicated attribute for further reuse.

To prevent unexpected behaviors, you need to think of the mutability of the input data. If you have a property that computes its value from constant input values, then the result will never change. In that case, you can compute the value just once:

Python

```
# circle.py

from time import sleep

class Circle:
    def __init__(self, radius):
        self.radius = radius
        self._diameter = None

    @property
    def diameter(self):
        if self._diameter is None:
            sleep(0.5)  # Simulate a costly computation
            self._diameter = self.radius * 2
        return self._diameter
```

Even though this implementation of `Circle` properly caches the computed diameter, it has the drawback that if you ever change the value of `.radius`, then `.diameter` won't return a correct value:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42.0)
```

```
>>> circle.radius
42.0

>>> circle.diameter  # With delay
84.0
>>> circle.diameter  # Without delay
84.0

>>> circle.radius = 100.0
>>> circle.diameter  # Wrong diameter
84.0
```

In these examples, you create a circle with a radius equal to `42.0`. The `.diameter` property computes its value only the first time you access it. That's why you see a delay in the first execution and no delay in the second. Note that even though you change the value of the radius, the diameter stays the same.

If the input data for a computed attribute mutates, then you need to recalculate the attribute:

Python

```python
# circle.py

from time import sleep

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        self._diameter = None
        self._radius = value

    @property
    def diameter(self):
        if self._diameter is None:
            sleep(0.5)  # Simulate a costly computation
            self._diameter = self._radius * 2
        return self._diameter
```

The setter method of the `.radius` property resets `._diameter` to `None` every time you change the value of the radius. With this little update, `.diameter` recalculates its value the first time you access it after every mutation of `.radius`:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42.0)

>>> circle.radius
42.0
>>> circle.diameter  # With delay
84.0
>>> circle.diameter  # Without delay
84.0

>>> circle.radius = 100.0
>>> circle.diameter  # With delay
200.0
>>> circle.diameter  # Without delay
200.0
```

Cool! `Circle` works correctly now! It computes the diameter the first time you access it and also every time you change the radius.

Another option to create cached properties is to use `functools.cached_property()` from the standard library. This function works as a decorator that allows you to transform a method into a cached property. The property computes its value only once and caches it as a normal attribute during the lifetime of the instance:

Python

```python
# circle.py

from functools import cached_property
from time import sleep

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @cached_property
    def diameter(self):
        sleep(0.5)  # Simulate a costly computation
        return self.radius * 2
```

Here, `.diameter` computes and caches its value the first time you access it. This kind of implementation is suitable for those computations in which the input values don't mutate. Here's how it works:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42.0)
>>> circle.diameter  # With delay
84.0
>>> circle.diameter  # Without delay
84.0

>>> circle.radius = 100
>>> circle.diameter  # Wrong diameter
84.0

>>> # Allow direct assignment
```

```
>>> circle.diameter = 200
>>> circle.diameter  # Cached value
200
```

When you access `.diameter`, you get its computed value. That value remains the same from this point on. However, unlike `property()`, `cached_property()` doesn't block attribute mutations unless you provide a proper setter method. That's why you can update the diameter to `200` in the last couple of lines.

If you want to create a cached property that doesn't allow modification, then you can use `property()` and `functools.cache()` like in the following example:

Python

```
# circle.py

from functools import cache
from time import sleep

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    @cache
    def diameter(self):
        sleep(0.5) # Simulate a costly computation
        return self.radius * 2
```

This code stacks `@property` on top of `@cache`. The combination of both decorators builds a cached property that prevents mutations:

Python >>>

```
>>> from circle import Circle

>>> circle = Circle(42.0)

>>> circle.diameter  # With delay
84.0
>>> circle.diameter  # Without delay
84.0

>>> circle.radius = 100
>>> circle.diameter
84.0

>>> circle.diameter = 200
Traceback (most recent call last):
    ...
AttributeError: can't set attribute
```

In these examples, when you try to assign a new value to `.diameter`, you get an `AttributeError` because the setter functionality comes from the internal descriptor of `property`.

Logging Attribute Access and Mutation

Sometimes you need to keep track of what your code does and how your programs flow. A way to do that in Python is to use `logging`. This module provides all the functionality you would require for logging your code. It'll allow you to constantly watch the code and generate useful information about how it works.

If you ever need to keep track of how and when you access and mutate a given attribute, then you can take advantage of `property()` for that, too:

Python

```
# circle.py

import logging

logging.basicConfig(
    format="%(asctime)s: %(message)s",
    level=logging.INFO,
    datefmt="%H:%M:%S"
)

class Circle:
    def __init__(self, radius):
        self._msg = '"radius" was %s. Current value: %s'
        self.radius = radius

    @property
    def radius(self):
        """The radius property."""
        logging.info(self._msg % ("accessed", str(self._radius)))
        return self._radius

    @radius.setter
    def radius(self, value):
        try:
            self._radius = float(value)
            logging.info(self._msg % ("mutated", str(self._radius)))
        except ValueError:
            logging.info('validation error while mutating "radius"')
```

Here, you first import `logging` and define a basic configuration. Then you implement `Circle` with a managed attribute `.radius`. The getter method generates log information every time you access `.radius` in your code. The setter method logs each mutation that you perform on `.radius`. It also logs those situations in which you get an error because of bad input data.

Here's how you can use `Circle` in your code:

```
Python >>>

>>> from circle import Circle

>>> circle = Circle(42.0)

>>> circle.radius
14:48:59: "radius" was accessed. Current value: 42.0
42.0

>>> circle.radius = 100
14:49:15: "radius" was mutated. Current value: 100

>>> circle.radius
14:49:24: "radius" was accessed. Current value: 100
100

>>> circle.radius = "value"
15:04:51: validation error while mutating "radius"
```

Logging useful data from attribute access and mutation can help you debug your code. Logging can also help you identify sources of problematic data input, analyze the performance of your code, spot usage patterns, and more.

Managing Attribute Deletion

You can also create properties that implement deleting functionality. This might be a rare use case of `property()`, but having a way to delete an attribute can be handy in some situations.

Say you're implementing your own tree data type. A tree is an abstract data type that stores elements in a hierarchy. The tree components are commonly known as **nodes**. Each node in a tree has a parent node, except for the root node. Nodes can have zero or more children.

Now suppose you need to provide a way to delete or clear the list of children of a given node. Here's an example that implements a tree node that uses `property()` to provide most of its functionality, including the ability to clear the list of children of the node at hand:

```
Python

# tree.py

class TreeNode:
    def __init__(self, data):
        self._data = data
        self._children = []

    @property
    def children(self):
        return self._children

    @children.setter
    def children(self, value):
        if isinstance(value, list):
            self._children = value
        else:
            del self.children
            self._children.append(value)

    @children.deleter
    def children(self):
        self._children.clear()

    def __repr__(self):
        return f'{self.__class__.__name__}("{self._data}")'
```

In this example, `TreeNode` represents a node in your custom tree data type. Each node stores its children in a Python list. Then you implement `.children` as a property to manage the underlying list of children. The deleter method calls `.clear()` on the list of children to remove them all:

```
Python >>>

>>> from tree import TreeNode

>>> root = TreeNode("root")
>>> child1 = TreeNode("child 1")
>>> child2 = TreeNode("child 2")

>>> root.children = [child1, child2]

>>> root.children
[TreeNode("child 1"), TreeNode("child 2")]

>>> del root.children
>>> root.children
[]
```

Here, you first create a `root` node to start populating the tree. Then you create two new nodes and assign them to `.children` using a list. The `del` statement triggers the internal deleter method of `.children` and clears the list.

Creating Backward-Compatible Class APIs

As you already know, properties turn method calls into direct attribute lookups. This feature allows you to create clean and Pythonic APIs for your classes. You can expose your attributes publicly without the need for getter and setter methods.

If you ever need to modify how you compute a given public attribute, then you can turn it into a property. Properties make it possible to perform extra processing, such as data validation, without having to modify your public APIs.

Suppose you're creating an accounting application and you need a base class to manage currencies. To this end, you create a `Currency` class that exposes two attributes, `.units` and `.cents`:

```
Python

class Currency:
```

```
    def __init__(self, units, cents):
        self.units = units
        self.cents = cents

    # Currency implementation...
```

This class looks clean and Pythonic. Now say that your requirements change, and you decide to store the total number of cents instead of the units and cents. Removing `.units` and `.cents` from your public API to use something like `.total_cents` would break more than one client's code.

In this situation, `property()` can be an excellent option to keep your current API unchanged. Here's how you can work around the problem and avoid breaking your clients' code:

Python

```
# currency.py

CENTS_PER_UNIT = 100

class Currency:
    def __init__(self, units, cents):
        self._total_cents = units * CENTS_PER_UNIT + cents

    @property
    def units(self):
        return self._total_cents // CENTS_PER_UNIT

    @units.setter
    def units(self, value):
        self._total_cents = self.cents + value * CENTS_PER_UNIT

    @property
    def cents(self):
        return self._total_cents % CENTS_PER_UNIT

    @cents.setter
    def cents(self, value):
        self._total_cents = self.units * CENTS_PER_UNIT + value

    # Currency implementation...
```

Now your class stores the total number of cents instead of independent units and cents. However, your users can still access and mutate `.units` and `.cents` in their code and get the same result as before. Go ahead and give it a try!

When you write something upon which many people are going to build, you need to guarantee that modifications to the internal implementation don't affect how end users work with your classes.

Overriding Properties in Subclasses

When you create Python classes that include properties and release them in a package or library, you should expect your users to do a lot of different things with them. One of those things could be **subclassing** them to customize their functionalities. In these cases, your users have to be careful and be aware of a subtle gotcha. If you partially override a property, then you lose the non-overridden functionality.

For example, suppose you're coding an `Employee` class to manage employee information in your company's internal accounting system. You already have a class called `Person`, and you think about subclassing it to reuse its functionalities.

`Person` has a `.name` attribute implemented as a property. The current implementation of `.name` doesn't meet the requirement of returning the name in uppercase letters. Here's how you end up solving this:

Python

```
# persons.py

class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    # Person implementation...

class Employee(Person):
    @property
    def name(self):
        return super().name.upper()

    # Employee implementation...
```

In `Employee`, you override `.name` to make sure that when you access the attribute, you get the employee name in uppercase:

Python >>>

```
>>> from persons import Employee, Person

>>> person = Person("John")
>>> person.name
'John'
>>> person.name = "John Doe"
>>> person.name
'John Doe'
```

```
>>> employee = Employee("John")
>>> employee.name
'JOHN'
```

Great! `Employee` works as you need! It returns the name using uppercase letters. However, subsequent tests uncover an unexpected behavior:

```
Python                                                    >>>
>>> employee.name = "John Doe"
Traceback (most recent call last):
    ...
AttributeError: can't set attribute
```

What happened? Well, when you override an existing property from a parent class, you override the whole functionality of that property. In this example, you reimplemented the getter method only. Because of that, `.name` lost the rest of the functionality from the base class. You don't have a setter method any longer.

The idea is that if you ever need to override a property in a subclass, then you should provide all the functionality you need in the new version of the property at hand.

Conclusion

A **property** is a special type of class member that provides functionality that's somewhere in between regular attributes and methods. Properties allow you to modify the implementation of instance attributes without changing the public API of the class. Being able to keep your APIs unchanged helps you avoid breaking code your users wrote on top of older versions of your classes.

Properties are the Pythonic way to create **managed attributes** in your classes. They have several use cases in real-world programming, making them a great addition to your skill set as a Python developer.

In this tutorial, you learned how to:

- Create **managed attributes** with Python's `property()`
- Perform **lazy attribute evaluation** and provide **computed attributes**
- Avoid **setter** and **getter** methods with properties
- Create **read-only**, **read-write**, and **write-only** attributes
- Create consistent and **backward-compatible APIs** for your classes

You also wrote several practical examples that walked you through the most common use cases of `property()`. Those examples include input data validation, computed attributes, logging your code, and more.

About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» More about Leodanis

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Bartosz

 Martin

 Sadie

Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate python