Getters and Setters: Manage Attributes in Python

by Leodanis Pozo Ramos Nov 09, 2022 9 Comments

intermediate python

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Getters and Setters in Python**

If you come from a language like Java or C++, then you're probably used to writing **getter** and **setter** methods for every attribute in your classes. These methods allow you to access and mutate private attributes while maintaining **encapsulation**. In Python, you'll typically expose attributes as part of your public API and use **properties** when you need attributes with functional behavior.

Even though properties are the Pythonic way to go, they can have some practical drawbacks. Because of this, you'll find some situations where getters and setters are preferable over properties.

In this tutorial, you'll:

- Write **getter** and **setter** methods in your classes
- Replace getter and setter methods with **properties**
- Explore **other tools** to replace getter and setter methods in Python
- Decide when **setter** and **getter** methods can be the **right tool for the job**

To get the most out of this tutorial, you should be familiar with Python object-oriented programming. It'll also be a plus if you have basic knowledge of Python properties and descriptors.

Source Code: Click here to get the free source code that shows you how and when to use getters, setters, and properties in Python.

Getting to Know Getter and Setter Methods

When you define a class in object-oriented programming (OOP), you'll likely end up with some instance and class attributes. These attributes are just variables that you can access through the instance, the class, or both.

Attributes hold the internal state of objects. In many cases, you'll need to access and mutate this state, which involves accessing and mutating the attributes. Typically, you'll have at least two ways to access and mutate attributes. You can either:

1. Access and mutate the attribute **directly**
2. Use **methods** to access and mutate the attribute

If you expose the attributes of a class to your users, then those attributes automatically become part of the class's public API. They'll be **public attributes**, which means that your users will directly access and mutate the attributes in their code.

Having an attribute that's part of a class's API will become a problem if you need to change the internal implementation of the attribute itself. A clear example of this issue is when you want to turn a **stored** attribute into a **computed** one. A stored attribute will immediately respond to access and mutation operations by just retrieving and storing data, while a computed attribute will run computations before such operations.

The problem with regular attributes is that they can't have an *internal implementation* because they're just variables. So, changing an attribute's internal implementation will require converting the attribute into a method, which will probably break your users' code. Why? Because they'll have to change attribute access and mutation operations into method calls throughout their codebase if they want the code to continue working.

To deal with this kind of issue, some programming languages, like Java and C++, require you to provide methods for manipulating the attributes of your classes. These methods are commonly known as **getter** and **setter** methods. You can also find them referred to as accessor and mutator methods.

What Are Getter and Setter Methods?

Getter and setter methods are quite popular in many object-oriented programming languages. So, it's pretty likely that you've heard about them already. As a rough definition, you can say that getters and setters are:

- **Getter:** A method that allows you to *access* an attribute in a given class
- **Setter:** A method that allows you to *set* or *mutate* the value of an attribute in a class

In OOP, the getter and setter pattern suggests that public attributes should be used only when you're sure that no one will ever need to attach behavior to them. If an attribute is likely to change its internal implementation, then you should use getter and setter methods.

Implementing the getter and setter pattern requires:

1. Making your attributes non-public
2. Writing getter and setter methods for each attribute

For example, say that you need to write a `Label` class with text and font attributes. If you were to use getter and setter methods to manage these attributes, then you'd write the class like in the following code:

Python

```
# label.py

class Label:
    def __init__(self, text, font):
        self._text = text
        self._font = font

    def get_text(self):
        return self._text

    def set_text(self, value):
        self._text = value

    def get_font(self):
        return self._font

    def set_font(self, value):
        self._font = value
```

In this example, the constructor of `Label` takes two arguments, `text` and `font`. These arguments are stored in the `._text` and `._font` non-public instance attributes, respectively.

Then you define getter and setter methods for both attributes. Typically, getter methods return the target attribute's value, while setter methods take a new value and assign it to the underlying attribute.

> **Note:** Python doesn't have the notion of access modifiers, such as `private`, `protected`, and `public`, to restrict access to attributes and methods in a class. 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 should use the well-established Python convention of prefixing the name with an underscore (_).

> 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.

You can use your `Label` class like in the examples below:

Python >>>

```
>>> from label import Label

>>> label = Label("Fruits", "JetBrains Mono NL")
>>> label.get_text()
'Fruits'

>>> label.set_text("Vegetables")

>>> label.get_text()
'Vegetables'

>>> label.get_font()
'JetBrains Mono NL'
```

`Label` hides its attributes from public access and exposes getter and setter methods instead. You can use these methods when you need to access or mutate the class's attributes, which are non-public and therefore not part of the class API, as you already know.

Where Do Getter and Setter Methods Come From?

To understand where getter and setter methods come from, get back to the `Label` example and say that you want to automatically store the label's text in uppercase letters. Unfortunately, you can't simply add this behavior to a regular attribute like `.text`. You can only add behavior through methods, but converting a public attribute into a method will introduce a **breaking change** in your API.

So, what can you do? Well, in Python, you'll most likely use a property, as you'll learn soon. However, programming languages like Java and C++ don't support property-like constructs, or their properties aren't quite like Python properties.

That's why these languages encourage you *never to expose your attributes as part of your public APIs*. Instead, you must *provide getter and setter methods*, which offer a quick way to change the internal implementation of your attributes without changing your public API.

Encapsulation is another fundamental topic related to the origin of getter and setter methods. Essentially, this principle refers to bundling data with the methods that operate on that data. This way, access and mutation operations will be done through methods exclusively.

The principle also has to do with restricting direct access to an object's attributes, which will prevent exposing implementation details or violating state invariance.

To provide `Label` with the newly required functionality in Java or C++, you must use getter and setter methods from the beginning. How can you apply the getter and setter pattern to solve the problem in Python?

Consider the following version of `Label`:

Python

```
# label.py

class Label:
    def __init__(self, text, font):
        self.set_text(text)
        self.font = font

    def get_text(self):
        return self._text

    def set_text(self, value):
        self._text = value.upper()  # Attached behavior
```

In this updated version of `Label`, you provide getter and setter methods for the label's text. The attribute holding the text is **non-public** because it has a leading underscore on its name, `._text`. The setter method does the input transformation, converting the text into uppercase letters.

Now you can use your `Label` class like in the following code snippet:

Python >>>

```
>>> from label import Label

>>> label = Label("Fruits", "JetBrains Mono NL")
>>> label.get_text()
'FRUITS'

>>> label.set_text("Vegetables")
>>> label.get_text()
'VEGETABLES'
```

Cool! You've successfully added the required behavior to your label's text attribute. Now your setter method has a true goal instead of just assigning a new value to the target attribute. It has the goal of adding extra behavior to the `._text` attribute.

Even though the getter and setter pattern is quite common in other programming languages, that's not the case in Python.

Adding getter and setter methods to your classes can considerably increase the number of lines in your code. Getters and setters also follow a repetitive and boring pattern that'll require extra time to complete. This pattern can be error-prone and tedious. You'll also find that the immediate functionality gained from all this additional code is often zero.

All this sounds like something that Python developers wouldn't want to do in their code. In Python, you'll probably write the `Label` class like in the following snippet:

Python >>>

```
>>> class Label:
...     def __init__(self, text, font):
...         self.text = text
...         self.font = font
...
```

Here, `.text`, and `.font` are public attributes and are exposed as part of the class's API. This means that your users can and will change their value whenever they like:

Python >>>

```
>>> label = Label("Fruits", "JetBrains Mono NL")
>>> label.text
'Fruits'

>>> # Later...
>>> label.text = "Vegetables"
>>> label.text
'Vegetables'
```

Exposing attributes like `.text` and `.font` is common practice in Python. So, your users will directly access and mutate this kind of attribute in their code.

Making your attributes public, like in the above example, is a common practice in Python. In these cases, switching to getters and setters will introduce breaking changes. So, how do you deal with situations that require adding behavior to your attributes? The Pythonic way to do this is to replace attributes with properties.

Using Properties Instead of Getters and Setters: The Python Way

The Pythonic way to attach behavior to an attribute is to turn the attribute itself into a **property**. Properties pack together methods for getting, setting, deleting, and documenting the underlying data. Therefore, properties are special attributes with additional behavior.

You can use properties in the same way that you use regular attributes. When you access a property, its attached getter method is automatically called. Likewise, when you mutate the property, its setter method gets called. This behavior provides the means to attach functionality to your attributes without introducing breaking changes in your code's API.

As an example of how properties can help you attach behavior to attributes, say that you need an `Employee` class as part of an employee management system. You start with the following bare-bones implementation:

Python

```
# employee.py

class Employee:
    def __init__(self, name, birth_date):
        self.name = name
        self.birth_date = birth_date

    # Implementation...
```

This class's constructor takes two arguments, the name and date of birth of the employee at hand. These attributes are directly stored in two instance attributes, `.name` and `.birth_date`.

You can start using the class right away:

Python >>>

```
>>> from employee import Employee

>>> john = Employee("John", "2001-02-07")

>>> john.name
'John'
>>> john.birth_date
'2001-02-07'

>>> john.name = "John Doe"
>>> john.name
'John Doe'
```

`Employee` allows you to create instances that give you direct access to the associated name and birth date. Note that you can also mutate the attributes by using direct assignments.

As your project evolves, you have new requirements. You need to store the employee's name in uppercase letters and turn the birth date into a `date` object. To meet these requirements without breaking your API with getter and setter methods for `.name` and `.birth_date`, you can use properties:

Python

```
# employee.py

from datetime import date

class Employee:
    def __init__(self, name, birth_date):
        self.name = name
        self.birth_date = birth_date

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

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

    @property
    def birth_date(self):
        return self._birth_date

    @birth_date.setter
    def birth_date(self, value):
        self._birth_date = date.fromisoformat(value)
```

In this enhanced version of `Employee`, you turn `.name` and `.birth_date` into properties using the `@property` decorator. Now each attribute has a getter and a setter method named after the attribute itself. Note that the setter of `.name` turns the input name into uppercase letters. Similarly, the setter of `.birth_date` automatically converts the input date into a `date` object for you.

As mentioned before, a neat feature of properties is that you can use them as regular attributes:

Python >>>

```
>>> from employee import Employee

>>> john = Employee("John", "2001-02-07")

>>> john.name
'JOHN'

>>> john.birth_date
datetime.date(2001, 2, 7)

>>> john.name = "John Doe"
>>> john.name
'JOHN DOE'
```

Cool! You've added behavior to the `.name` and `.birth_date` attributes without affecting your class's API. With properties, you've gained the ability to refer to these attributes as you would to regular attributes. Behind the scenes, Python takes care of running the appropriate methods for you.

You must avoid breaking your user's code by introducing changes in your APIs. Python's `@property` decorator is the Pythonic way to do that. Properties are officially recommended in PEP 8 as the right way to deal with attributes that need functional behavior:

> For simple public data attributes, it's best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax. (Source)

Python's properties have a lot of potential use cases. For example, you can use properties to create read-only, read-write, and write-only attributes in an elegant and straightforward manner. Properties allow you to delete and document the underlying attributes and more. More importantly, properties allow you to make regular attributes behave like managed attributes with attached behavior without changing the way you work with them.

Because of properties, Python developers tend to design their classes' APIs using a few guidelines:

- Use **public attributes** whenever appropriate, even if you expect the attribute to require functional behavior in the future.
- **Avoid** defining **setter** and **getter** methods for your attributes. You can always turn them into properties if needed.
- Use **properties** when you need to **attach behavior** to attributes and keep using them as regular attributes in your code.
- **Avoid side effects** in properties because no one would expect operations like assignments to cause any side effects.

Python's properties are cool! Because of that, people tend to overuse them. In general, you should only use properties when you need to add extra processing on top of a specific attribute. Turning all your attributes into properties will be a waste of your time. It may also imply performance and maintainability issues.

Replacing Getters and Setters With More Advanced Tools

Up to this point, you've learned how to create bare-bones getter and setter methods to manage the attributes of your classes. You've also learned that properties are the Pythonic way to approach the problem of adding functional behavior to existing attributes.

In the following sections, you'll learn about other tools and techniques that you can use to replace getter and setter methods in Python.

Python's Descriptors

Descriptors are an advanced Python feature that allows you to create attributes with attached behaviors in your classes. To create a descriptor, you need to use the **descriptor protocol**, especially the `.__get__()` and `.__set__()` special methods.

Descriptors are pretty similar to properties. In fact, a property is a special type of descriptor. However, regular descriptors are more powerful than properties and can be reused through different classes.

To illustrate how to use descriptors to create attributes with functional behavior, say that you need to continue developing your `Employee` class. This time, you need an attribute to store the date on which an employee started to work for the company:

Python

```
# employee.py

from datetime import date

class Employee:
    def __init__(self, name, birth_date, start_date):
        self.name = name
        self.birth_date = birth_date
        self.start_date = start_date

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

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

    @property
    def birth_date(self):
        return self._birth_date

    @birth_date.setter
    def birth_date(self, value):
        self._birth_date = date.fromisoformat(value)

    @property
    def start_date(self):
        return self._start_date

    @start_date.setter
    def start_date(self, value):
        self._start_date = date.fromisoformat(value)
```

In this update, you added another property to `Employee`. This new property will allow you to manage the start date of each employee. Again, the setter method converts the date from a string to a `date` object.

This class works as expected. However, it starts to look repetitive and boring. So, you decide to refactor the class. You notice that you're doing the same operation in both date-related attributes, and you think of using a descriptor to pack the repetitive functionality:

Python

```python
# employee.py

from datetime import date

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

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

    def __set__(self, instance, value):
        instance.__dict__[self._name] = date.fromisoformat(value)

class Employee:
    birth_date = Date()
    start_date = Date()

    def __init__(self, name, birth_date, start_date):
        self.name = name
        self.birth_date = birth_date
        self.start_date = start_date

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

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

This code is cleaner and less repetitive than its previous version. In this update, you create a `Date` descriptor to manage date-related attributes. The descriptor has a `.__set_name__()` method that automatically stores the attribute name. It also has `.__get__()` and `.__set__()` methods that work as the attribute's getter and setter, respectively.

The two implementations of `Employee` in this section work similarly. Go ahead and give them a try!

In general, if you find yourself cluttering your classes with similar property definitions, then you should consider using a descriptor instead.

The `.__setattr__()` and `.__getattr__()` Methods

Another way to replace traditional getter and setter methods in Python is to use the `.__setattr__()` and `.__getattr__()` special methods to manage your attributes. Consider the following example, which defines a `Point` class. The class automatically converts the input coordinates into floating-point numbers:

Python

```
# point.py

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

    def __getattr__(self, name: str):
        return self.__dict__[f"_{name}"]

    def __setattr__(self, name, value):
        self.__dict__[f"_{name}"] = float(value)
```

The initializer of `Point` takes two coordinates, `x` and `y`. The `.__getattr__()` method returns the coordinate represented by `name`. To do this, the method uses the instance namespace dictionary, `.__dict__`. Note that the attribute's final name will have an underscore preceding whatever you pass in `name`. Python automatically calls `.__getattr__()` whenever you access an attribute of `Point` using the dot notation.

The `.__setattr__()` method adds or updates attributes. In this example, `.__setattr__()` operates on each coordinate and converts it into a floating-point number using the built-in `float()` function. Again, Python calls `.__setattr__()` whenever you run an assignment operation on any attribute of the containing class.

Here's how this class works in practice:

Python >>>

```
>>> from point import Point

>>> point = Point(21, 42)

>>> point.x
21.0
>>> point.y
42.0

>>> point.x = 84
>>> point.x
84.0

>>> dir(point)
['__class__', '__delattr__', ..., '_x', '_y']
```

Your `Point` class automatically converts coordinate values into floating-point numbers. You can access the coordinates, `x` and `y`, as you would any other regular attribute. However, access and mutation operations go through `.__getattr__()` and `.__setattr__()`, respectively.

Note that `Point` allows you to access the coordinates as public attributes. However, it stores them as non-public attributes. You can confirm this behavior with the built-in `dir()` function.

The example in this section is a bit exotic, and you probably won't use something similar in your code. However, the tools that you've used in the example allow you to perform validations or transformations on attribute access and mutation, just like getter and setter methods do.

In a sense, `.__getattr__()` and `.__setattr__()` are kind of a generic implementation of the getter and setter pattern. Under the hood, these methods work as getters and setters that support regular attribute access and mutation in Python.

Deciding Whether to Use Getters and Setters or Properties in Python

In real-world coding, you'll find a few use cases where getter and setter methods can be preferred over properties, even though properties are generally the Pythonic way to go.

For example, getter and setter methods may be better suited to deal with situations in which you need to:

- **Run costly transformations** on attribute access or mutation
- Take **extra arguments** and **flags**
- Use inheritance
- Raise exceptions related to attribute access and mutation
- Facilitate integration in **heterogeneous** development **teams**

In the following sections, you'll dive into these use cases and why getter and setter methods can be better than properties to approach such cases.

Avoiding Slow Methods Behind Properties

You should avoid hiding slow operations behind a Python property. The users of your APIs will expect attribute access and mutation to perform like regular variable access and mutation. In other words, users will expect these operations to happen *instantaneously* and without *side effects*.

Going too far away from that expectation will make your API odd and unpleasant to use, violating the least surprise principle.

Additionally, if your users repeatedly access and mutate your attributes in a loop, then their code can involve too much overhead, which may produce huge and *unexpected* performance issues.

In contrast, traditional getter and setter methods make it *explicit* that accessing or mutating a given attribute happens through a method call. Indeed, your users will be aware that calling a method can take time, and the performance of their code can vary significantly because of that.

Making such facts explicit in your APIs can help minimize your users' surprise when they access and mutate your attributes in their code.

In short, if you're going to use a property to manage an attribute, then make sure that the methods behind the property are fast and don't cause side effects. In contrast, if you're dealing with slow accessor and mutator methods, then favor traditional getters and setters over properties.

Taking Extra Arguments and Flags

Unlike Python properties, traditional getter and setter methods allow for more flexible attribute access and mutation. For example, say you have a `Person` class with a `.birth_date` attribute. This attribute should be constant during a person's lifetime. Therefore, you decide that the attribute will be read-only.

However, because human error exists, you'll face cases in which someone makes a mistake when entering the date of birth of a given person. You can solve this problem by providing a setter method that takes a `force` flag, like in the example below:

Python

```
# person.py

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

    def get_birth_date(self):
        return self._birth_date

    def set_birth_date(self, value, force=False):
        if force:
            self._birth_date = value
        else:
            raise AttributeError("can't set birth_date")
```

You provide traditional getter and setter methods for the `.birth_date` attribute in this example. The setter method takes an extra argument called `force`, which allows you to force the modification of a person's date of birth.

> **Note:** Traditional setter methods typically don't take more than one argument. The example above may look odd or even incorrect to some developers. However, its intention is to showcase a technique that can be useful in some situations.

Here's how this class works:

Python >>>

```
>>> from person import Person

>>> jane = Person("Jane Doe", "2000-11-29")
>>> jane.name
'Jane Doe'

>>> jane.get_birth_date()
'2000-11-29'

>>> jane.set_birth_date("2000-10-29")
Traceback (most recent call last):
    ...
AttributeError: can't set birth_date

>>> jane.set_birth_date("2000-10-29", force=True)
>>> jane.get_birth_date()
'2000-10-29'
```

When you try to modify Jane's date of birth using `.set_birth_date()` without setting `force` to `True`, you get an `AttributeError` signaling that the attribute can't be set. In contrast, if you set `force` to `True`, then you'll be able to update Jane's date of birth to correct any errors that occurred when the date was entered.

It's important to note that Python properties don't accept extra arguments in their setter methods. They just accept the value to be set or updated.

Using Inheritance: Getter and Setters vs Properties

One issue with Python properties is that they don't do well in inheritance scenarios. For example, say that you need to extend or modify the getter method of a property in a subclass. In practice, there's no safe way to do this. You can't just override the getter method and expect the rest of the property's functionality to remain the same as in the parent class.

This issue occurs because the getter and setter methods are hidden inside the property. They're not inherited independently but as a whole. Therefore, when you override the getter method of a property inherited from a parent class, you override the whole property, including its setter method and the rest of its internal components.

As an example, consider the following class hierarchy:

Python

```
# person.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

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

In this example, you override the getter method of the `.name` property in `Employee`. This way, you're implicitly overriding the whole `.name` property, including its setter functionality:

Python >>>

```
>>> from person import Employee

>>> jane = Employee("Jane")

>>> jane.name
'JANE'

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

Now `.name` is a read-only property because the setter method of the parent class wasn't inherited but was overridden by a completely new property. You don't want that, do you? How can you solve this inheritance issue?

If you use traditional getter and setter methods, then the issue won't happen:

Python

```python
# person.py

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

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

class Employee(Person):
    def get_name(self):
        return super().get_name().upper()
```

This version of `Person` provides independent getter and setter methods. `Employee` subclasses `Person`, overriding the getter method for the name attribute. This fact doesn't affect the setter method, which `Employee` successfully inherits from its parent class, `Person`.

Here's how this new version of `Employee` works:

Python >>>

```
>>> from person import Employee

>>> jane = Employee("Jane")

>>> jane.get_name()
'JANE'

>>> jane.set_name("Jane Doe")
>>> jane.get_name()
'JANE DOE'
```

Now `Employee` is completely functional. The overridden getter method works as expected. The setter method also works because it was successfully inherited from `Person`.

Raising Exceptions on Attribute Access or Mutation

Most of the time, you won't expect an assignment statement like `obj.attribute = value` to raise an exception. In contrast, you can expect methods to raise exceptions in response to errors. In this regard, traditional getter and setter methods are more explicit than properties.

For example, `site.url = "123"` doesn't look like something that can raise an exception. It looks and should behave like a regular attribute assignment. On the other hand, `site.set_url("123")` does look like something that can raise an exception, perhaps a `ValueError`, because the input value isn't a valid URL for a website. In this example, the setter method is more explicit. It clearly expresses the code's possible behavior.

As a rule of thumb, avoid raising exceptions from your Python properties unless you're using a property to provide read-only attributes. If you ever need to raise exceptions on attribute access or mutation, then you should consider using getter and setter methods instead of properties.

In these cases, using getters and setters will reduce the user's surprise and make your code more aligned with common practices and expectations.

Facilitating Team Integration and Project Migration

Providing getter and setter methods is common practice in many well-established programming languages. If you're working on a Python project with a team of developers who come from other language backgrounds, then it's pretty likely that the getter and setter pattern will look more familiar to them than Python properties.

In this type of heterogeneous team, using getters and setters can facilitate the integration of new developers into the team.

Using the getter and setter pattern can also promote API consistency. It allows you to provide an API based on method calls rather than an API that combines method calls with direct attribute access and mutation.

Often, when a Python project grows, you may need to migrate the project from Python to another language. The new language may not have properties, or they may not behave as Python properties do. In these situations, using traditional getters and setters from the beginning would make future migrations less painful.

In all of the above situations, you should consider using traditional getter and setter methods instead of properties in Python.

Conclusion

Now you know what **getter** and **setter** methods are and where they come from. These methods allow access and mutation of attributes while avoiding API changes. However, they're not so popular in Python because of the existence of properties. Properties allow you to add behavior to your attributes while avoiding breaking changes in your APIs.

Even though properties are the Pythonic way to replace traditional getters and setters, properties can have some practical drawbacks that you can overcome with getters and setters.

In this tutorial, you've learned how to:

- Write **getter** and **setter** methods in Python
- Use Python **properties** to replace getter and setter methods
- Use Python **tools**, like descriptors, to replace getters and setters
- Decide on when **setter** and **getter** methods can be the **right tool for the job**

With all this knowledge, you can now decide when to use either getter and setter methods or properties in your Python classes.

Source Code: Click here to get the free source code that shows you how and when to use getters, setters, and properties in Python.

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Getters and Setters in Python**

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

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

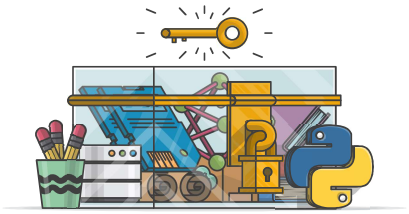
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Geir Arne

Kate

Philipp



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

Level Up Your Python Skills »

What Do You Think?

Rate this article:

Tweet Share Share Email

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: intermediate python

Recommended Video Course: Getters and Setters in Python