Properties in Python

What Are Properties?

A **property** in Python allows you to access a method like an attribute.

Normally, an attribute looks like this:

```
user.name
```

A method looks like this:

```
user.get_name()
```

With a property, you can create controlled behavior behind normal attribute syntax:

```
user.name
```

The key idea is:

> **A property lets an attribute-like interface perform method-like logic.**

Why Do Properties Matter?

Suppose we have a `Student` class:

```
class Student:
    def __init__(self, marks):
        self.marks = marks
```

Anyone can change the value:

```
student.marks = -100
```

That may create invalid object state.

We could use a getter and setter:

```
student.get_marks()
student.set_marks(80)
```

But Python's property system lets us keep the cleaner syntax:

```
student.marks
student.marks = 80
```

while still controlling what happens internally.

The Basic Property Pattern

A property is commonly created using:

```
@property
```

Example:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

    @property
```

```
    def marks(self):
        return self._marks
```

Now:

```
student = Student(85)

print(student.marks)
```

Output:

```
85
```

Notice that we did **not** write:

```
student.marks()
```

The method behaves like an attribute because of `@property`.

How `@property` Works

This:

```
@property
def marks(self):
    return self._marks
```

creates a property named:

```
marks
```

When Python sees:

```
student.marks
```

it calls the property's getter internally.

Conceptually:

```
student.marks
    ↓
@property
    ↓
marks()
    ↓
self._marks
```

You get the convenience of attribute access with the control of a method.

Why Use `_marks` Instead of `marks`?

Consider:

```
class Student:
    @property
    def marks(self):
        return self.marks
```

This creates a problem.

When `self.marks` is accessed, Python calls the property again:

marks

marks

marks

...

This causes recursive access.

Instead, use a separate internal attribute:

self._marks

So:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

    @property
    def marks(self):
        return self._marks
```

Here:

marks

property

_marks

Properties With Setters

A getter lets you read a property.

A **setter** lets you control what happens when someone assigns a value.

Example:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

    @property
    def marks(self):
        return self._marks

    @marks.setter
    def marks(self, value):
        self._marks = value
```

Now:

```
student = Student(80)

print(student.marks)

student.marks = 90

print(student.marks)
```

Output:

```
80
```

`90`

The assignment:

`student.marks = 90`

automatically calls:

`marks.setter`

Properties Allow Validation

This is where properties become especially useful.

Suppose marks must be between `0` and `100`.

```
class Student:
    def __init__(self, marks):
        self.marks = marks

    @property
    def marks(self):
        return self._marks

    @marks.setter
    def marks(self, value):
        if not 0 <= value <= 100:
            raise ValueError("Marks must be between 0 and 100")

        self._marks = value
```

Now:

```
student = Student(85)
```

works.

But:

```
student.marks = 150
```

raises:

```
ValueError: Marks must be between 0 and 100
```

The property protects the object's state.

Property vs Normal Attribute

Normal attribute

```
class Student:
    def __init__(self, marks):
        self.marks = marks
```

Anyone can assign:

```
student.marks = -50
```

No validation exists.

Property

```
class Student:
    def __init__(self, marks):
```

```
        self._marks = marks

    @property
    def marks(self):
        return self._marks

    @marks.setter
    def marks(self, value):
        if value < 0:
            raise ValueError("Marks cannot be negative")

        self._marks = value
```

Now assignment is controlled.

Read Only Properties

Sometimes you want a value to be readable but not directly changeable.

You can create a property without a setter.

Example:

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

    @property
    def area(self):
        return 3.14159 * self._radius ** 2
```

Now:

```
circle = Circle(5)

print(circle.area)
```

Output:

```
78.53975
```

But:

```
circle.area = 100
```

raises an error because there is no setter.

The property is effectively **read-only**.

Computed Properties

A property does not have to simply return stored data.

It can calculate a value.

Example:

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

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

Now:

```
rectangle = Rectangle(10, 5)

print(rectangle.area)
```

Output:

```
50
```

There is no `area` value stored separately.

It is calculated whenever the property is accessed.

Why Is This Better Than `get_area()`?

You could write:

```
rectangle.get_area()
```

But:

```
rectangle.area
```

often reads more naturally because `area` represents a property of the rectangle.

Compare:

```
print(rectangle.get_area())
```

with:

```
print(rectangle.area)
```

The second version communicates:

> "Area is a characteristic of this object."

Properties Can Hide Implementation Details

Suppose you initially store a user's full name:

```
class User:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"
```

The user writes:

```
user.full_name
```

The internal implementation is hidden.

Later, you could change how the name is calculated without changing the code that uses:

`user.full_name`

This is an important software design benefit.

Properties and Encapsulation

Properties are closely connected to **encapsulation**.

Encapsulation means controlling how an object's internal state is accessed or changed.

For example:

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance

    @property
    def balance(self):
        return self._balance
```

The balance can be read:

`print(account.balance)`

But changing it can be controlled through specific methods:

```
def deposit(self, amount):
    if amount <= 0:
        raise ValueError("Amount must be positive")

    self._balance += amount
```

This is usually better than allowing arbitrary changes:

```
account.balance = -50000
```

Property vs Getter and Setter Methods

Traditional getter/setter style:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

    def get_marks(self):
        return self._marks

    def set_marks(self, value):
        if 0 <= value <= 100:
            self._marks = value
```

Usage:

```
student.get_marks()
student.set_marks(90)
```

Property style:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

    @property
    def marks(self):
        return self._marks

    @marks.setter
    def marks(self, value):
        if 0 <= value <= 100:
            self._marks = value
```

Usage:

```
student.marks
student.marks = 90
```

Python generally favors the property approach when attribute-style access makes sense.

Why Properties Are Useful for API Design

Suppose you initially expose:

```
user.name
```

Later you realize you need validation or transformation.

Without properties, you might need to change the public interface.

With a property, you can keep:

`user.name`

while changing the implementation behind it.

For example:

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

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

Now the caller still uses:

`user.name`

but the implementation can normalize the value.

This is sometimes called preserving a stable interface.

Setter With Validation

A common property pattern is:

```
class Product:
    def __init__(self, price):
        self.price = price

    @property
```

```
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price cannot be negative")

        self._price = value
```

Notice something important:

```
self.price = price
```

inside `__init__` uses the setter.

This means the same validation is applied during object creation and later updates.

Setter With Type Checking

Properties can also validate data types.

```
class User:
    def __init__(self, age):
        self.age = age

    @property
    def age(self):
        return self._age

    @age.setter
```

```
    def age(self, value):
        if not isinstance(value, int):
            raise TypeError("Age must be an integer")

        if value < 0:
            raise ValueError("Age cannot be negative")

        self._age = value
```

Now:

```
user = User(25)
```

works.

But:

```
user.age = "twenty five"
```

raises a `TypeError`.

Setter With Transformation

A setter can also normalize incoming data.

Example:

```
class User:
    def __init__(self, username):
        self.username = username
```

```
    @property
    def username(self):
        return self._username

    @username.setter
    def username(self, value):
        self._username = value.strip().lower()
```

Now:

```
user = User("  Hafsa  ")

print(user.username)
```

Output:

```
hafsa
```

The caller provides:

```
"  Hafsa  "
```

but the object stores:

```
"hafsa"
```

Deleting a Property

Properties can also define a deleter using:

```
@name.deleter
```

Example:

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

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

    @name.deleter
    def name(self):
        del self._name
```

Now:

```
user = User("Hafsa")

del user.name
```

calls the property deleter.

A deleter is useful only when deletion is meaningful for the object's design.

You do not need to define one for every property.

Complete Getter, Setter and Deleter

A property can support all three operations:

```
class User:
```

```
    def __init__(self, name):
        self._name = name

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

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")

        self._name = value

    @name.deleter
    def name(self):
        del self._name
```

This gives:

```
user.name
    
getter

user.name = "Hafsa"
    
setter

del user.name
    
deleter
```

Properties With Dependent Values

Properties are especially useful when one value depends on another.

Example:

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

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

    @property
    def perimeter(self):
        return 2 * (self.width + self.height)
```

Now:

```
rectangle = Rectangle(10, 5)

print(rectangle.area)
print(rectangle.perimeter)
```

Output:

```
50
30
```

Neither `area` nor `perimeter` needs to be stored separately.

Properties and Object State

Be careful when a property performs expensive work.

This:

```
@property
def report(self):
    # expensive database query
```

looks like a simple attribute:

```
object.report
```

but could actually perform significant work.

Properties should generally behave like attributes from the user's perspective.

If an operation is expensive, has side effects, or performs a substantial action, a normal method may communicate the behavior more clearly:

```
object.generate_report()
```

rather than:

```
object.report
```

Property vs Method

Use a **property** when the value conceptually behaves like an attribute.

Good examples:

```
user.full_name
rectangle.area
account.balance
product.price
```

Use a **method** when an action is being performed.

Good examples:

```
user.send_email()
account.withdraw(100)
report.generate()
order.cancel()
```

A useful mental model:

```
Property → What something IS / HAS

Method → What something DOES
```

This is not an absolute rule, but it is a useful design guideline.

Properties and Inheritance

Properties can be inherited like other class attributes and methods.

Example:

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

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

A child class can use it:

```
class Student(Person):
    pass
```

Now:

```
student = Student("Hafsa")

print(student.name)
```

The inherited property works normally.

Overriding a Property

A child class can also override a property.

```
class Person:
    @property
    def role(self):
        return "Person"


class Student(Person):
```

```
    @property
    def role(self):
        return "Student"
```

Now:

```
person = Person()
student = Student()

print(person.role)
print(student.role)
```

Output:

```
Person
Student
```

This is another example of polymorphic behavior.

`property()` Without Decorators

The `@property` syntax is the most readable approach.

But properties can also be created using the built-in `property()` function.

Example:

```
class Student:
    def __init__(self, marks):
        self._marks = marks

```

```
    def get_marks(self):
        return self._marks

    def set_marks(self, value):
        if not 0 <= value <= 100:
            raise ValueError("Marks must be between 0 and 100")

        self._marks = value

    marks = property(get_marks, set_marks)
```

This works:

```
student.marks = 90
print(student.marks)
```

But the decorator syntax is generally easier to read:

```
@property
def marks(self):
    return self._marks
```

The Decorator Syntax Explained

When you write:

```
@property
def marks(self):
    return self._marks
```

Python creates a property object.

Then:

```
@marks.setter
def marks(self, value):
    self._marks = value
```

adds the setter behavior to that property.

So the name:

```
marks
```

represents the complete property interface.

A Realistic haas.dev Example

Imagine a resource object:

```
class Resource:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    @property
    def reading_time(self):
        return max(1, self.pages // 5)
```

Now:

```
resource = Resource(
    "Python OOP Guide",
```

```
    25
)

print(resource.reading_time)
```

Output:

```
5
```

The reading time is derived from the number of pages.

The caller does not need to know how the value is calculated.

They simply use:

```
resource.reading_time
```

A More Complete Resource Example

```
class Resource:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    @property
    def pages(self):
        return self._pages

    @pages.setter
    def pages(self, value):
        if not isinstance(value, int):
            raise TypeError("Pages must be an integer")
```

```
        if value <= 0:
            raise ValueError("Pages must be greater than zero")

        self._pages = value

    @property
    def reading_time(self):
        return max(1, self.pages // 5)
```

Usage:

```
resource = Resource("Python OOP Guide", 25)

print(resource.pages)
print(resource.reading_time)
```

The class now provides:

```
Stored state:
_pages

Controlled property:
pages

Computed property:
reading_time
```

Properties and Backward Compatibility

One powerful reason to use properties is that they allow you to change implementation without necessarily changing how other code interacts with your class.

Suppose you start with:

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

Other code uses:

```
user.name
```

Later you need validation.

You can change the implementation to:

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

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

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")

        self._name = value
```

The calling code can still use:

```
user.name
```

This is one reason properties are useful for designing stable class interfaces.

Common Mistake: Infinite Recursion

Incorrect:

```
class User:
    @property
    def name(self):
        return self.name
```

The property calls itself repeatedly.

Correct:

```
class User:
    @property
    def name(self):
        return self._name
```

Use a separate internal storage attribute.

Common Mistake: Forgetting the Setter

Suppose you have:

```
class Student:
    @property
    def marks(self):
```

```
        return self._marks
```

Then:

```
student.marks = 90
```

will fail because the property has no setter.

If the property should be writable, add:

```
@marks.setter
```

If it should be read-only, not having a setter is intentional.

Common Mistake: Putting Side Effects in Getters

Avoid surprising behavior such as:

```
@property
def balance(self):
    send_email()
    charge_database()
    return self._balance
```

Someone reading:

```
account.balance
```

normally expects to retrieve information, not trigger unrelated actions.

Properties should generally have predictable attribute-like behavior.

Common Mistake: Using Properties Everywhere

Not every attribute needs a property.

This is often unnecessary:

```
class User:
    @property
    def name(self):
        return self._name

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

If there is no validation, transformation, calculation, or controlled access requirement, a normal attribute may be simpler:

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

Do not add abstraction without a reason.

Common Mistake: Using a Property for an Action

Avoid:

```
@property
def send_email(self):
    ...
```

if accessing the property actually sends an email.

Use:

```
def send_email(self):
    ...
```

because sending an email is an action.

Properties and Dataclasses

Python's `dataclasses` are useful for classes primarily storing data.

A property can still be added when derived or controlled behavior is needed.

For example:

```
from dataclasses import dataclass


@dataclass
class Rectangle:
    width: float
    height: float

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

Usage:

```
rectangle = Rectangle(10, 5)

print(rectangle.area)
```

Output:

```
50
```

Properties and dataclasses can therefore work together.

Properties in Real Applications

Properties are commonly useful for:

Validation

```
user.age = 25
```

while ensuring the value is valid.

Computed values

```
order.total
```

Read-only information

```
user.id
```

Normalization

```
user.email
```

Encapsulation

```
account.balance
```

Stable APIs

Allowing callers to keep using:

```
object.value
```

even when internal implementation changes.

A Simple Decision Framework

When designing a class, ask:

Is it just simple stored data?

Use a normal attribute.

```
self.name = name
```

Does reading it require calculation?

Consider a property.

```
@property
def total(self):
    ...
```

Does assigning it require validation?

Use a property setter.

```
@value.setter
def value(self, value):
    ...
```

Should users be able to read it but not change it?

Use a read-only property.

Does accessing it perform a significant action?

Prefer a method.

```
generate_report()
```

Mini Project: Product

Create a `Product` class with:

```
name
price
quantity
```

Requirements:

1. Price cannot be negative.
2. Quantity cannot be negative.
3. `total_price` should be calculated automatically.
4. `price` and `quantity` should use properties.

Expected usage:

```
product = Product("Python Guide", 100, 3)
```

```
print(product.price)
print(product.quantity)
print(product.total_price)
```

Expected:

```
100
3
300
```

Changing:

```
product.quantity = 5
```

should automatically make:

```
product.total_price
```

return:

```
500
```

---

5 Minute Challenge

Build a `Temperature` class.

It should store Celsius internally:

```
temperature = Temperature(25)
```

Provide:

```
temperature.celsius
temperature.fahrenheit
```

`fahrenheit` should be a computed property.

Formula:

```
F = C × 9/5 + 32
```

So:

```
temperature = Temperature(25)

print(temperature.celsius)
print(temperature.fahrenheit)
```

should produce:

```
25
77
```

Then add a setter for Celsius that prevents temperatures below absolute zero:

```
-273.15°C
```

Exercises

Exercise 1

Create a `Circle` class with:

`radius`
`area`
`circumference`

Make `area` and `circumference` properties.

Exercise 2

Create a `BankAccount` class.

Use a property for:

`balance`

The balance should be readable but not directly assignable.

Provide methods:

`deposit()`
`withdraw()`

for changing the balance.

Exercise 3

Create a `User` class.

Create a property:

`email`

The setter should:

- require a string
- remove surrounding spaces
- convert it to lowercase
- reject an empty value

Exercise 4

Create a `Rectangle` class.

Use properties for:

`area`
`perimeter`

Neither value should be stored directly.

Exercise 5

Create a `Resource` class for haas.dev.

Include:

`title`

`pages`
`reading_time`

Requirements:

- `pages` must be a positive integer.
- `reading_time` should be calculated automatically.
- `reading_time` should be read-only.

Mini Quiz

1. What does `@property` allow?

A. A method to be accessed like an attribute
B. A class to inherit another class
C. A function to become global
D. A variable to become constant

Answer: A

---

2. Which decorator creates a setter?

A. `@property.setter`
B. `@name.setter`
C. `@setter`
D. `@setproperty`

Answer: B

---

3. Why is `_name` commonly used inside a property?

A. It is required by Python
B. It provides separate internal storage
C. It makes the variable constant
D. It automatically validates data

Answer: B

---

4. What happens if a property has no setter?

A. It becomes writable automatically
B. Assignment normally raises an error
C. Python deletes the property
D. The property becomes a method

Answer: B

---

5. Which is usually more appropriate for an action?

A. Property
B. Method
C. Class attribute
D. Getter

Answer: B

---

6. Which is a good property example?

A. `user.send_email`
B. `order.cancel`
C. `rectangle.area`
D. `database.connect`

Answer: C

Interview Questions

Beginner

1. What is a property in Python?

A property allows a method to be accessed using attribute syntax.

2. What is `@property`?

It is a built-in decorator used to define a property's getter.

3. Why are properties useful?

They allow controlled access, validation, computed values, and stable interfaces while keeping simple attribute-style syntax.

Intermediate

4. What is the purpose of a property setter?

A setter controls what happens when a value is assigned to the property.

5. How do you create a read-only property?

Define the property with `@property` but do not define a setter.

6. Why do properties commonly use `_attribute` internally?

The underscore provides separate internal storage and prevents the property from recursively accessing itself.

Advanced

7. What is the difference between a property and a method?

A property represents attribute-like information, while a method generally represents an action or operation.

8. Why are properties useful for API design?

They allow the implementation behind an attribute-like interface to change without necessarily changing the code that uses the class.

9. Can properties be overridden in subclasses?

Yes. A subclass can define a property with the same name and provide different behavior.

10. Can properties be used with dataclasses?

Yes. A dataclass can contain properties for computed or controlled behavior.

Properties Cheat Sheet

Basic property

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

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

Property with setter

```
@property
def age(self):
    return self._age

@age.setter
def age(self, value):
    if value < 0:
        raise ValueError("Invalid age")

    self._age = value
```

Read-only property

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

Computed property

```
@property
```

```
def total(self):
    return self.price * self.quantity
```

Deleter

```
@name.deleter
def name(self):
    del self._name
```

Properties vs Attributes vs Methods

Feature	Attribute	Property	Method
Syntax	`obj.name`	`obj.name`	`obj.do_something()`
Can calculate value	Usually no	Yes	Yes
Validation on assignment	No	Yes	Yes
Read-only	Not by default	Yes	Usually
Represents	Stored data	Attribute-like behavior	Action/operation
Example	`user.name`	`rectangle.area`	`account.withdraw()`

Key Takeaways

1. A **property** lets a method behave like an attribute.
2. `@property` defines the getter.
3. `@name.setter` defines controlled assignment.
4. A property can validate incoming values.
5. A property can calculate values dynamically.
6. A property without a setter can be effectively read-only.
7. `_attribute` is commonly used as internal storage.
8. Properties are closely related to encapsulation.
9. Use properties when something conceptually behaves like an attribute.
10. Use methods for actions and operations.
11. Properties can help maintain a stable public interface while implementation changes internally.
12. Do not use properties everywhere; simple data can remain simple attributes.

The core mental model is:

```
Normal attribute:
object.value

Property:
object.value
    ↓
getter logic
    ↓
stored or calculated result

Assignment:
object.value = new_value
    ↓
setter logic
    ↓
```

`validation / transformation / storage`

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app