In Python, there are multiple ways to implement **getters and setters** in a class. Here are the main approaches:

1️⃣ Using `@property` (Recommended & Pythonic)

This is the **most Pythonic way** of defining getters and setters.

python
CopyEdit

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

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

    @name.setter
    def name(self, new_name):  # Setter
        if not isinstance(new_name, str):
            raise ValueError("Name must be a string")
        self._name = new_name

# Usage
p = Person("Alice")
print(p.name)  # Calls the getter

p.name = "Bob"  # Calls the setter
print(p.name)

# p.name = 123  # Raises ValueError
```

✅ **Pros:**
✔ More readable and **Pythonic**
✔ Looks like accessing an attribute (`p.name` instead of `p.get_name()`)
✔ Encapsulates logic inside the property

2️⃣ Using Traditional Getter & Setter Methods (Java-Style)

This is an **older, non-Pythonic way** but is still valid.

python
CopyEdit

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

    def get_name(self):  # Getter
        return self._name

    def set_name(self, new_name):  # Setter
        if not isinstance(new_name, str):
            raise ValueError("Name must be a string")
        self._name = new_name

# Usage
p = Person("Alice")
print(p.get_name())  # Calls the getter

p.set_name("Bob")  # Calls the setter
print(p.get_name())
```

⚠ **Cons:**

- **Less Pythonic** (Python prefers properties over explicit getter/setter methods)
- **More verbose**

3️⃣ Using `property()` Function (Alternative)

Instead of using `@property`, you can manually create a property using the `property()` function.

python

CopyEdit

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

    def get_name(self):  # Getter
        return self._name

    def set_name(self, new_name):  # Setter
        if not isinstance(new_name, str):
            raise ValueError("Name must be a string")
        self._name = new_name

    # Creating a property manually
    name = property(get_name, set_name)

# Usage
p = Person("Alice")
print(p.name)  # Calls the getter

p.name = "Bob"  # Calls the setter
print(p.name)
```

✅ **Pros:**
✔ Similar to `@property`, but more explicit

⚠ **Cons:**

- **Less readable** than using `@property`

4️⃣ Using `@property` with `@name.getter` (Explicit Getter)

This is useful if you're also defining a setter and want to make it clear.

python
CopyEdit

```
class Person:
    def __init__(self, name):
```

```
        self._name = name

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

    @name.getter
    def name(self):
        """Explicit getter (same as @property)"""
        return self._name

    @name.setter
    def name(self, new_name):
        """Setter"""
        if not isinstance(new_name, str):
            raise ValueError("Name must be a string")
        self._name = new_name

# Usage
p = Person("Alice")
print(p.name)  # Calls the getter

p.name = "Bob"  # Calls the setter
print(p.name)
```

✅ **Pros:**
✔ More explicit when using multiple decorators like `@setter` or `@deleter`.

5️⃣ Using `__getattr__` and `__setattr__` (Advanced & Dynamic)

This method dynamically controls attribute access.

python
CopyEdit

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

    def __getattr__(self, attr):  # Getter
        if attr == "name":
            return self._name
        raise AttributeError(f"{attr} not found")

    def __setattr__(self, attr, value):  # Setter
        if attr == "name":
            if not isinstance(value, str):
                raise ValueError("Name must be a string")
            self.__dict__["_name"] = value  # Prevent recursion
        else:
            self.__dict__[attr] = value  # Default behavior

# Usage
p = Person("Alice")
print(p.name)  # Calls __getattr__

p.name = "Bob"  # Calls __setattr__
print(p.name)
```

✅ **Pros:**
✔ **More flexible**, can handle multiple attributes dynamically
⚠ **Cons:**

- **Overkill** for simple cases
- **Harder to debug**

6️⃣ Using `dataclasses` (For Simple Cases)

If you just need a simple class with validation, **dataclasses** can be used.

python
CopyEdit

```
from dataclasses import dataclass
```

```
@dataclass
class Person:
    name: str

# Usage
p = Person("Alice")
print(p.name)

p.name = "Bob"
print(p.name)
```

✅ **Pros:**
✔ **Automatic `__init__`, `__repr__`, etc.**
⚠ **Cons:**

- No built-in validation (you'd need `__post_init__`)

🎯 Summary of Different Approaches

Method	Pythonic ?	Readability	Validation ?	Use Case
`@property` (Recommended)	✅ Yes	✅ High	✅ Yes	Best for most cases
Traditional `get_name()`/`set_name()`	❌ No	❌ Low	✅ Yes	Java-style, not recommended
`property()` function	❌ No	❌ Medium	✅ Yes	Alternative to `@property`
`@name.getter`	✅ Yes	✅ High	✅ Yes	If using `@setter/@deleter`
`__getattr__` & `__setattr__`	❌ No	❌ Complex	✅ Yes	Dynamic attribute handling
`dataclasses`	✅ Yes	✅ High	❌ No	Simple cases

🚀 Best Practices

✅ **Use `@property` for clean, readable, and Pythonic code**
✅ **Use explicit getters/setters (`@radius.getter`) only when necessary**
✅ **Use `__getattr__` only when dealing with dynamic attributes**
✅ **Use `dataclasses` for simple cases where validation isn't required**

Would you like me to demonstrate a real-world example of using getters and setters? 🚀