**Defining Properties in Classes in Python**

In Python, **properties** in classes allow us to define controlled access to instance attributes. They are useful when we need to encapsulate data, ensuring that attributes are properly managed (e.g., validating values before assignment).

**1. Using the property() Function**

The property() function can be used to define properties in a class.

**Example:**

class Person:

def \_\_init\_\_(self, name):

self.\_name = name # Private attribute (convention: underscore prefix)

def get\_name(self):

return self.\_name

def set\_name(self, value):

if not isinstance(value, str):

raise ValueError("Name must be a string")

self.\_name = value

def del\_name(self):

print("Deleting name...")

del self.\_name

# Creating a property

name = property(get\_name, set\_name, del\_name, "Property for person's name")

# Usage

p = Person("Alice")

print(p.name) # Calls get\_name()

p.name = "Bob" # Calls set\_name()

del p.name # Calls del\_name()

**Explanation:**

* property(get\_name, set\_name, del\_name, docstring) creates a property.
* When p.name is accessed, get\_name() is called.
* When p.name is assigned a value, set\_name() is called.
* When del p.name is executed, del\_name() is called.

**2. Using the @property Decorator (Recommended)**

A cleaner way to define properties is using the @property decorator.

**Example:**

class Person:

def \_\_init\_\_(self, name):

self.\_name = name # Private attribute

@property

def name(self):

"""Getter method for name"""

return self.\_name

@name.setter

def name(self, value):

"""Setter method for name"""

if not isinstance(value, str):

raise ValueError("Name must be a string")

self.\_name = value

@name.deleter

def name(self):

"""Deleter method for name"""

print("Deleting name...")

del self.\_name

# Usage

p = Person("Alice")

print(p.name) # Calls the getter

p.name = "Bob" # Calls the setter

del p.name # Calls the deleter

**Advantages of Using @property Decorator:**

1. **More Readable & Pythonic**: It avoids explicit calls to property().
2. **Encapsulation & Validation**: You can enforce rules (e.g., name must be a string).
3. **Maintains Attribute-like Access**: Allows seamless usage of p.name instead of p.get\_name().

**3. Read-Only Properties**

If you only define a getter method (@property) and do not provide a setter, the attribute becomes **read-only**.

**Example:**

class Circle:

def \_\_init\_\_(self, radius):

self.\_radius = radius

@property

def radius(self):

return self.\_radius # Read-only

# Usage

c = Circle(5)

print(c.radius) # Allowed

c.radius = 10 # Error! Attribute cannot be set

**Explanation:**

* radius is defined with @property but no @radius.setter, making it immutable.

**4. Computed Properties**

Properties can be used to define **computed attributes**.

**Example:**

import math

class Circle:

def \_\_init\_\_(self, radius):

self.\_radius = radius

@property

def area(self):

"""Computed property for area"""

return math.pi \* self.\_radius \*\* 2 # No setter, derived from radius

# Usage

c = Circle(5)

print(c.area) # Computed value: 78.54

c.area = 100 # Error! Read-only computed property

**Explanation:**

* area is a computed property (depends on \_radius).
* No setter is provided, so it cannot be modified directly.

**Summary**

| **Method** | **Usage** |
| --- | --- |
| property(get, set, del, doc) | Traditional way (less readable) |
| @property | Defines a getter method |
| @property + @setter | Defines a writable property |
| @property + @deleter | Allows property deletion |
| Read-only property | Only defines @property, no setter |
| Computed property | Returns a derived value (e.g., area) |

Using @property is the **preferred** and **Pythonic** way to define properties in Python.