Advanced Python Programming

# LESSON 5: Properties & Basic Descriptors

**Why This Matters:** Every time you write `obj.attribute`, Python is working behind the scenes to make that simple syntax possible. Understanding properties and descriptors is like understanding how a car engine works - it makes you a better driver and helps you build more robust, maintainable code.

**The Problem We're Solving:** Imagine you're building a `Temperature` class. You start with a simple attribute, but then you need validation (no temperatures below absolute zero), computed properties (Fahrenheit from Celsius), and change notifications. Without properties, you'd need ugly getter/setter methods everywhere. Properties let you keep the simple `obj.temperature` syntax while adding powerful behavior behind the scenes.

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand the mental model of properties as "smart attributes"
- Create properties for validation and computed values
- Understand the basic descriptor protocol that powers properties
- Apply properties to create clean, maintainable class interfaces

Lesson Outline:

## I. The Mental Model: Properties as "Smart Attributes" (10 min)

**Think of properties like a helpful assistant sitting between you and your data:**

- **Normal attribute**: You ask for temperature, you get exactly what's stored (even if it's wrong)
- **Property**: You ask for temperature, your assistant checks if it makes sense, converts it if needed, or computes it fresh

**The Property Mental Model - Three Types of Assistants:**

1. **The Validator Assistant** - Checks data before storing it
   - "Before I save this temperature, let me make sure it's not below absolute zero"
2. **The Calculator Assistant** - Computes values on demand
   - "You want Fahrenheit? Let me calculate that from the Celsius value"
3. **The Transformer Assistant** - Modifies how data is accessed or stored
   - "You're setting the name? Let me clean it up and log this change"

Let's see this in action:

```python
# The Problem: Without properties, you need ugly methods
class TemperatureBad:
    def __init__(self, celsius):
        self._celsius = celsius

    def get_celsius(self):  # Ugly getter
        return self._celsius

    def set_celsius(self, value):  # Ugly setter
        if value < -273.15:
            raise ValueError("Below absolute zero!")
        self._celsius = value

    def get_fahrenheit(self):  # More ugly getters
        return (self._celsius * 9/5) + 32

# Usage is ugly
temp = TemperatureBad(25)
print(temp.get_celsius())  # Ugly
temp.set_celsius(30)       # Ugly
print(temp.get_fahrenheit())  # Ugly

# The Solution: Properties make it beautiful
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius  # Uses the property setter automatically!

    @property
    def celsius(self):
        """The Validator Assistant - clean access"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """The Validator Assistant - checks before storing"""
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """The Calculator Assistant - computes on demand"""
        return (self.celsius * 9/5) + 32

    @property
    def kelvin(self):
        """The Calculator Assistant - another computed property"""
        return self.celsius + 273.15

    def __repr__(self):
        return f"Temperature({self.celsius}C)"

# Usage is beautiful and natural
temp = Temperature(25)
print(temp.celsius)     # 25 - looks like a simple attribute!
print(temp.fahrenheit)  # 77.0 - computed automatically!
print(temp.kelvin)      # 298.15 - computed automatically!

temp.celsius = 100      # Validation happens automatically!
print(temp.fahrenheit)  # 212.0 - updated computation!

# The assistant protects us from bad data
try:
    temp.celsius = -300  # The assistant says "No way!"
except ValueError as e:
    print(f"Protected by our assistant: {e}")
```

**Key Insight:** Properties make your class interface clean while adding powerful behavior behind the scenes. Your users get simple attribute access, but you get validation, computation, and control.

---

#### II. Common Property Patterns for Real Applications (15 min)

##### Pattern 1: The Data Validator - "Bouncer at the Door"

Think of validation properties like a bouncer at a club - they check everyone before letting them in.

```
class User:
    def __init__(self, username, email, age):
        self.username = username  # Uses property setters
        self.email = email
        self.age = age

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

    @username.setter
    def username(self, value):
        """The Bouncer: checks username rules"""
        if not isinstance(value, str):
            raise TypeError("Username must be a string")
        if len(value) < 3:
            raise ValueError("Username must be at least 3 characters")
        if not value.replace('_', '').isalnum():
            raise ValueError("Username can only contain letters, numbers,
and underscores")
        self._username = value

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        """The Bouncer: checks email format"""
        if not isinstance(value, str):
            raise TypeError("Email must be a string")
        if '@' not in value or '.' not in value.split('@')[-1]:
            raise ValueError("Invalid email format")
        self._email = value.lower()  # Normalize to lowercase

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

    @age.setter
    def age(self, value):
        """The Bouncer: checks age constraints"""
        if not isinstance(value, int):
            raise TypeError("Age must be an integer")
        if value < 0:
            raise ValueError("Age cannot be negative")
        if value > 150:
            raise ValueError("Age seems unrealistic")
        self._age = value

    def __repr__(self):
        return f"User(username={self.username!r}, email={self.email!r},
age={self.age})"

# The bouncer protects your data
user = User("john_doe", "John.Doe@EXAMPLE.COM", 25)
print(user)  # User(username='john_doe', email='john.doe@example.com',
age=25)

# The bouncer rejects bad data
try:
    user.username = "xy"  # Too short!
except ValueError as e:
    print(f"Bouncer rejected: {e}")

try:
    user.email = "not-an-email"  # Bad format!
except ValueError as e:
    print(f"Bouncer rejected: {e}")

try:
    user.age = -5  # Impossible age!
except ValueError as e:
    print(f"Bouncer rejected: {e}")
```

**Pattern 2: The Smart Calculator - "Personal Math Assistant"**

These properties compute values on demand, always giving you fresh, accurate results.

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

    @property
    def area(self):
        """Smart Calculator: always gives current area"""
        return self.width * self.height

    @property
    def perimeter(self):
        """Smart Calculator: always gives current perimeter"""
        return 2 * (self.width + self.height)

    @property
    def diagonal(self):
        """Smart Calculator: uses Pythagorean theorem"""
        return (self.width ** 2 + self.height ** 2) ** 0.5

    @property
    def aspect_ratio(self):
        """Smart Calculator: width to height ratio"""
        return self.width / self.height

    def __repr__(self):
        return f"Rectangle({self.width} x {self.height})"

# The calculator always gives current values
rect = Rectangle(10, 5)
print(f"Rectangle: {rect}")
print(f"Area: {rect.area}")              # 50
print(f"Perimeter: {rect.perimeter}")    # 30
print(f"Diagonal: {rect.diagonal:.2f}")  # 11.18

# Change dimensions - calculations update automatically!
rect.width = 20
print(f"New area: {rect.area}")          # 100 (updated automatically!)
print(f"New diagonal: {rect.diagonal:.2f}")  # 20.62 (updated
automatically!)
```

**Pattern 3: The Data Transformer - "Smart Assistant Who Cleans Up"**

These properties clean and transform data as it goes in or comes out.

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        """Transformer: cleans and standardizes names"""
        if not value:
            raise ValueError("First name cannot be empty")
        # Clean and standardize: strip whitespace, title case
        self._first_name = value.strip().title()

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        """Transformer: cleans and standardizes names"""
        if not value:
            raise ValueError("Last name cannot be empty")
        self._last_name = value.strip().title()

    @property
    def full_name(self):
        """Transformer: combines names into full name"""
        return f"{self.first_name} {self.last_name}"

    @property
    def initials(self):
        """Transformer: creates initials"""
        return f"{self.first_name[0]}.{self.last_name[0]}."

    @property
    def display_name(self):
        """Transformer: creates display-friendly name"""
        return f"{self.last_name}, {self.first_name}"

# The transformer cleans up messy input
person = Person("  john  ", "  DOE  ")  # Messy input
print(f"Cleaned first name: {person.first_name}")  # "John" (cleaned!)
print(f"Cleaned last name: {person.last_name}")    # "Doe" (cleaned!)
print(f"Full name: {person.full_name}")            # "John Doe"
print(f"Initials: {person.initials}")              # "J.D."
print(f"Display name: {person.display_name}")      # "Doe, John"

# All transformations happen automatically
person.first_name = "  JANE  "  # Messy input again
print(f"Updated full name: {person.full_name}")    # "Jane Doe" (auto-
updated!)
```

---

### III. Properties for Caching and Performance (10 min)

#### The Mental Model: "Lazy but Smart Employee"

Imagine you have an employee who's really good at their job but also lazy. They'll only do work when absolutely necessary, but once they do it, they remember the result so they don't have to do it again. That's cached properties!

**Pattern: The Cached Calculator - "Do Once, Remember Forever"**

```
class DataProcessor:
    """Demonstrates cached properties for expensive operations."""

    def __init__(self, data):
        self._data = data
        self._stats_cache = None  # The employee's memory
        self._data_version = 0    # Track if data changed

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, value):
        """When data changes, clear the cache (employee forgets old
work)"""
        self._data = value
        self._stats_cache = None  # Clear cache
        self._data_version += 1   # Increment version

    @property
    def statistics(self):
        """Lazy employee: only calculates when needed, remembers result"""
        if self._stats_cache is None:
            print("Computing statistics... (this takes time)")
            # Simulate expensive computation
            import time
            time.sleep(0.5)  # Pretend this is really slow

            # Do the actual work
            self._stats_cache = {
                'count': len(self._data),
                'sum': sum(self._data),
                'mean': sum(self._data) / len(self._data),
                'max': max(self._data),
                'min': min(self._data)
            }
            print("Statistics computed and cached!")
        else:
            print("Using cached statistics (instant!)")

        return self._stats_cache

    @property
    def summary(self):
        """Another property that depends on the cached statistics"""
        stats = self.statistics  # Might use cache or compute fresh
        return f"Data: {stats['count']} items, mean={stats['mean']:.2f},
range=[{stats['min']}, {stats['max']}]"

# Watch the lazy employee in action
processor = DataProcessor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

print("First request for statistics:")
print(processor.statistics)  # Employee works hard (slow)

print("\nSecond request for statistics:")
print(processor.statistics)  # Employee uses memory (fast)

print("\nGetting summary (uses cached stats):")
print(processor.summary)     # Employee uses memory again (fast)

print("\nChanging data...")
processor.data = [10, 20, 30, 40, 50]  # Employee forgets old work

print("\nFirst request after change:")
print(processor.statistics)  # Employee works hard again (slow)

print("\nSecond request after change:")
print(processor.statistics)  # Employee uses new memory (fast)
```

**Pattern: Smart Property Factory - "Assembly Line for Properties"**

When you need many similar properties, create a factory:

```
def cached_property(expensive_function):
    """Factory that creates cached properties - like training employees"""
    cache_name = f'_cached_{expensive_function.__name__}'

    def property_getter(self):
        # Check if the lazy employee has done this work before
        if not hasattr(self, cache_name):
            print(f"Computing {expensive_function.__name__}...")
            # Train the employee to do the work
            result = expensive_function(self)
            # Employee remembers the result
            setattr(self, cache_name, result)
        else:
            print(f"Using cached {expensive_function.__name__}")

        return getattr(self, cache_name)

    return property(property_getter)

class CircleAnalyzer:
    """Multiple cached properties using our factory."""

    def __init__(self, radius):
```

```python
        self.radius = radius

    @cached_property
    def area(self):
        """Expensive area calculation (pretend it's complex)"""
        import time
        time.sleep(0.2)  # Simulate complexity
        return 3.14159 * self.radius ** 2

    @cached_property
    def circumference(self):
        """Expensive circumference calculation"""
        import time
        time.sleep(0.2)  # Simulate complexity
        return 2 * 3.14159 * self.radius

    @cached_property
    def diameter(self):
        """Expensive diameter calculation"""
        import time
        time.sleep(0.2)  # Simulate complexity
        return 2 * self.radius

    def clear_cache(self):
        """Fire all the lazy employees - make them forget everything"""
        for attr in list(self.__dict__.keys()):
            if attr.startswith('_cached_'):
                delattr(self, attr)
        print("All caches cleared!")

# Multiple lazy employees working for you
circle = CircleAnalyzer(5)

print("Getting area (first time):")
print(f"Area: {circle.area}")  # Employee 1 works

print("\nGetting area (second time):")
print(f"Area: {circle.area}")  # Employee 1 remembers

print("\nGetting circumference (first time):")
print(f"Circumference: {circle.circumference}")  # Employee 2 works

print("\nGetting all values (using cache):")
print(f"Area: {circle.area}")           # Employee 1 remembers
print(f"Circumference: {circle.circumference}")  # Employee 2 remembers
print(f"Diameter: {circle.diameter}")   # Employee 3 works (first time)

circle.clear_cache()  # Fire everyone
print("\nAfter clearing cache:")
print(f"Area: {circle.area}")  # Employee 1 has to work again
```

---

### IV. Understanding What Makes Properties Work (15 min)

#### The Mental Model: "Looking Behind the Curtain"

Properties feel like magic, but they're built on a simple foundation called the "descriptor protocol." Think of it like this: when you access `obj.attribute`, Python asks the attribute, "Hey, do you want to handle this yourself?" If the attribute knows how to handle its own access (by having special methods), it gets to control what happens.

#### The Basic Descriptor Protocol - Three Questions Python Asks:

1. **"Someone wants to get your value"** `__get__` method
2. **"Someone wants to set your value"** `__set__` method
3. **"Someone wants to delete you"** `__delete__` method

Let's build our own simple property to understand how it works:

```python
class SimpleProperty:
    """A basic property implementation to show how properties work."""

    def __init__(self, getter=None, setter=None):
        self.getter = getter
        self.setter = setter
        self.name = None

    def __set_name__(self, owner, name):
        """Python calls this when the descriptor is assigned to a
class."""
        self.name = name
        self.private_name = f'_{name}'

    def __get__(self, instance, owner):
        """Python calls this when someone accesses the attribute."""
        if instance is None:
            return self  # Accessed from class, return the descriptor

        if self.getter:
            return self.getter(instance)
        else:
            # Default behavior: get from private attribute
            return getattr(instance, self.private_name, None)

    def __set__(self, instance, value):
        """Python calls this when someone assigns to the attribute."""
        if self.setter:
            self.setter(instance, value)
        else:
            # Default behavior: store in private attribute
            setattr(instance, self.private_name, value)

# Let's see our simple property in action
class Temperature:
    def __init__(self, celsius):
        self.celsius = celsius

    def _get_celsius(self):
        print("Getting celsius via custom getter")
        return self._celsius

    def _set_celsius(self, value):
        print(f"Setting celsius via custom setter: {value}")
        if value < -273.15:
            raise ValueError("Below absolute zero!")
        self._celsius = value

    # Use our custom property
    celsius = SimpleProperty(_get_celsius, _set_celsius)

# Test it out
temp = Temperature(25)
print(f"Temperature: {temp.celsius}")  # Calls __get__
temp.celsius = 30                      # Calls __set__
print(f"New temperature: {temp.celsius}")

try:
    temp.celsius = -300  # Our setter validates!
except ValueError as e:
    print(f"Validation caught: {e}")
```

**Real Properties are Just Syntactic Sugar**

The `@property` decorator is just a nicer way to create descriptors:

```python
class BetterTemperature:
    def __init__(self, celsius):
        self.celsius = celsius

    @property
    def celsius(self):
        """This becomes the __get__ method"""
        print("Getting celsius via @property")
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """This becomes the __set__ method"""
        print(f"Setting celsius via @property: {value}")
        if value < -273.15:
            raise ValueError("Below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Read-only computed property (no setter)"""
```

```python
            return (self.celsius * 9/5) + 32

        @property
        def kelvin(self):
            """Another read-only computed property"""
            return self.celsius + 273.15

    # The @property syntax is just easier to read and write
    temp = BetterTemperature(25)
    print(f"Celsius: {temp.celsius}")
    print(f"Fahrenheit: {temp.fahrenheit}")
    print(f"Kelvin: {temp.kelvin}")

    temp.celsius = 100
    print(f"Boiling point - Fahrenheit: {temp.fahrenheit}")

    # Can't set computed properties (no setter defined)
    try:
        temp.fahrenheit = 500
    except AttributeError as e:
        print(f"Can't set computed property: {e}")
```

**When to Use Properties vs Simple Attributes**

**Use simple attributes when:**

- Data needs no validation or computation
- Performance is critical (properties add small overhead)
- The interface is internal/private

**Use properties when:**

- You need validation on assignment
- Values are computed from other data
- You want to add behavior later without breaking existing code
- You need to log or monitor access

```python
class SmartThermostat:
    """Example showing when to use each approach."""

    def __init__(self, target_temp, room_temp):
        # Simple attributes - no special behavior needed
        self.device_id = "THERMO_001"
        self.manufacturer = "SmartHome Inc"
        self.firmware_version = "2.1.4"

        # Properties - these need validation/computation
        self.target_temperature = target_temp
        self.current_temperature = room_temp

    @property
    def target_temperature(self):
        return self._target_temp

    @target_temperature.setter
    def target_temperature(self, value):
        """Validate temperature range for safety"""
        if not isinstance(value, (int, float)):
            raise TypeError("Temperature must be a number")
        if not (10 <= value <= 35):  # Celsius range
            raise ValueError("Target temperature must be between 10C and
35C")
        self._target_temp = value
        print(f"Target temperature set to {value}C")

    @property
    def current_temperature(self):
        return self._current_temp

    @current_temperature.setter
    def current_temperature(self, value):
        """Update current temperature and check if action needed"""
        self._current_temp = value
        if abs(value - self.target_temperature) > 2:
            print(f"Temperature difference detected: {value}C vs target
{self.target_temperature}C")

    @property
    def heating_needed(self):
        """Computed property - no storage needed"""
        return self.current_temperature < self.target_temperature - 1

    @property
    def cooling_needed(self):
        """Computed property - no storage needed"""
        return self.current_temperature > self.target_temperature + 1

    @property
    def status(self):
        """Computed status based on current conditions"""
        if self.heating_needed:
            return "HEATING"
        elif self.cooling_needed:
            return "COOLING"
        else:
            return "MAINTAINING"

# See the difference in action
thermostat = SmartThermostat(22, 18)
print(f"Device: {thermostat.device_id}")  # Simple attribute
print(f"Status: {thermostat.status}")     # Computed property
print(f"Heating needed: {thermostat.heating_needed}")  # Computed property

# Update conditions
```

```
thermostat.current_temperature = 25  # Triggers validation and logic
print(f"New status: {thermostat.status}")
```

---

## V. Recap & Best Practices (10 min)

### The Big Picture: Properties Transform Your Code Design

Properties are one of Python's most elegant features because they solve a fundamental problem: how do you add smart behavior to attributes without breaking existing code or making your interface ugly?

**Mental Model Summary:**

- **Simple attributes**: Direct storage, like a basic box
- **Properties**: Smart assistants that validate, compute, or transform data
- **Descriptors**: The underlying machinery that makes properties work

**When to Use Properties - The Decision Tree:**

1. **Start with simple attributes** - Don't over-engineer early
2. **Add properties when you need:**
    - Validation (bouncer at the door)
    - Computed values (smart calculator)
    - Data transformation (smart cleanup assistant)
    - Caching for expensive operations (lazy but smart employee)
    - Logging or monitoring access

**Essential Best Practices:**

```
class BestPracticesExample:
    """Demonstrates property best practices."""

    def __init__(self, name, age):
        # Use the property setters even in __init__
        self.name = name  # Triggers validation
        self.age = age    # Triggers validation

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

    @name.setter
    def name(self, value):
        """Best Practice: Clear error messages"""
        if not isinstance(value, str):
            raise TypeError(f"Name must be a string, got
{type(value).__name__}")
        if not value.strip():
            raise ValueError("Name cannot be empty or just whitespace")
        # Best Practice: Clean and normalize data
        self._name = value.strip().title()

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

    @age.setter
    def age(self, value):
        """Best Practice: Comprehensive validation"""
        if not isinstance(value, int):
            raise TypeError(f"Age must be an integer, got
{type(value).__name__}")
        if value < 0:
            raise ValueError("Age cannot be negative")
        if value > 150:
            raise ValueError("Age must be reasonable (0-150)")

        self._age = value

    @property
    def display_name(self):
        """Best Practice: Computed properties should be obviously read-
only"""
        return f"{self.name} (age {self.age})"

    # Best Practice: Don't add setters to computed properties
    # (no @display_name.setter)

    def __repr__(self):
        """Best Practice: Use properties in __repr__ for consistency"""
        return f"Person(name={self.name!r}, age={self.age})"
```

**Common Gotchas and How to Avoid Them:**

```
class CommonMistakes:
    """Examples of what NOT to do."""

    def __init__(self, value):
        # MISTAKE: Forgetting to use property in __init__
        self._value = value  # Bypasses validation!
        # CORRECT: self.value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, val):
        if val < 0:
            raise ValueError("Must be positive")
        self._value = val
    # MISTAKE: Using properties for expensive operations without caching
    @property
    def expensive_calculation(self):
        """This recalculates every time - bad!"""
        import time
        time.sleep(1)  # Expensive operation
        return sum(range(1000000))

    # CORRECT: Cache expensive calculations
    @property
    def better_calculation(self):
        """Cache expensive results"""
        if not hasattr(self, '_cached_calc'):
            import time
            time.sleep(1)  # Expensive operation
            self._cached_calc = sum(range(1000000))
        return self._cached_calc

# MISTAKE: Creating setters for obviously computed properties
class BadCircle:
    def __init__(self, radius):
        self.radius = radius

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

    @area.setter  # BAD: This is confusing!
    def area(self, value):
        # What does it mean to "set" an area?
        self.radius = (value / 3.14159) ** 0.5

# CORRECT: Keep computed properties read-only
class GoodCircle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        """Read-only computed property"""
        return 3.14159 * self.radius ** 2

    # No setter - it's obviously computed from radius
```

**Quick Reference Patterns:**

```
# Pattern 1: Validation Property
@property
def attr(self):
    return self._attr
```

```python
    @attr.setter
    def attr(self, value):
        if not self._is_valid(value):
            raise ValueError("Invalid value")
        self._attr = value

    # Pattern 2: Computed Property (read-only)
    @property
    def computed_attr(self):
        return self._calculate_from_other_attrs()

    # Pattern 3: Cached Property
    @property
    def expensive_attr(self):
        if not hasattr(self, '_cached_expensive'):
            self._cached_expensive = self._expensive_calculation()
        return self._cached_expensive

    # Pattern 4: Transforming Property
    @property
    def clean_attr(self):
        return self._attr

    @clean_attr.setter
    def clean_attr(self, value):
        self._attr = self._clean_and_normalize(value)
```

**Key Takeaways:**

- Properties make your interfaces clean and Pythonic
- Start simple, add properties when you need smart behavior
- Use clear error messages and comprehensive validation
- Keep computed properties read-only
- Cache expensive calculations
- Properties should feel natural and unsurprising to users

**Final Practice Question:**

Which of these is the best design for a `Product` class with name and price?

A. Use simple attributes for everything B. Use properties for validation, simple attributes for storage C. Use properties for name (validation), price (validation), and total_value (computed) D. Use descriptors for everything

(Answer: C. Use properties strategically - validation where needed, computation for derived values, simple attributes where appropriate.)