Advanced `@staticmethod` Examples

1. Utility Methods in a Math Library

Static methods are used when you want logical grouping of related utility functions without needing to instantiate the class.

```
class MathLibrary:
    @staticmethod
    def factorial(n):
        if n == 0 or n == 1:
            return 1
        return n * MathLibrary.factorial(n - 1)

    @staticmethod
    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a

    @staticmethod
    def is_prime(n):
        if n < 2:
            return False
        for i in range(2, int(n ** 0.5) + 1):
            if n % i == 0:
                return False
        return True

# Usage
print(MathLibrary.factorial(5))  # Output: 120
print(MathLibrary.gcd(48, 18))   # Output: 6
print(MathLibrary.is_prime(13)) # Output: True
```

- **Clarification**:
    - These methods operate solely on their inputs and outputs, independent of any class or instance attributes.
    - Grouping them under `MathLibrary` provides logical organization.

---

2. Static Methods in a Validator Class

Static methods can be used for input validation.

```
class Validator:
    @staticmethod
    def is_email_valid(email):
        import re
        return bool(re.match(r"[^@]+@[^@]+\.[^@]+", email))

    @staticmethod
    def is_password_strong(password):
        return len(password) >= 8 and any(char.isdigit() for char in password)

# Usage
print(Validator.is_email_valid("test@example.com"))  # Output: True
print(Validator.is_password_strong("Abc123"))       # Output: False
```

- **Clarification**:
    - Utility-like functionality that fits conceptually within the class but doesn't require class context.

Advanced `@classmethod` Examples

1. Alternate Constructors

A common use of `@classmethod` is to create alternate constructors for the class.

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

    @classmethod
    def from_string(cls, emp_string):
        name, salary = emp_string.split(',')
        return cls(name, int(salary))

# Usage
emp = Employee.from_string("Alice,50000")
print(emp.name)   # Output: Alice
print(emp.salary) # Output: 50000
```

- **Clarification**:
    - The `from_string` method is an alternate constructor that processes custom input formats to instantiate the class.

---

2. Managing Class-Level State

Class methods can manage shared state or perform logic based on class-level data.

```
class Configuration:
    _settings = {}

    @classmethod
    def set_setting(cls, key, value):
        cls._settings[key] = value

    @classmethod
    def get_setting(cls, key):
        return cls._settings.get(key, None)

# Usage
Configuration.set_setting("theme", "dark")
Configuration.set_setting("language", "English")

print(Configuration.get_setting("theme"))    # Output: dark
print(Configuration.get_setting("language")) # Output: English
```

- **Clarification**:
    - Class methods provide a centralized way to manage class-level state, useful in scenarios like global configuration settings.

---

3. Subclass-Friendly Behavior

Class methods are ideal for ensuring behavior extends naturally to subclasses.

```
class Animal:
    species = "Generic Animal"

    @classmethod
    def describe_species(cls):
        return f"This is a {cls.species}."

class Dog(Animal):
    species = "Dog"

class Cat(Animal):
    species = "Cat"
```

```
# Usage
print(Dog.describe_species())  # Output: This is a Dog.
print(Cat.describe_species())  # Output: This is a Cat.
```

- **Clarification**:
  - Using `cls` ensures the method refers to the correct subclass rather than the base class.

Comparison Recap

Here's a quick reference for when to use `@staticmethod` vs `@classmethod`:

Aspect	`@staticmethod`	`@classmethod`
Access to class (`cls`)	No	Yes
Access to instance (`self`)	No	No
Typical Use Cases	Utility methods, validation	Alternate constructors, managing class-level state
Relation to Class Context	Doesn't need class context	Operates within the class context

---

Combining `@staticmethod` and `@classmethod`

```
class BankAccount:
    _interest_rate = 0.05

    def __init__(self, balance):
        self.balance = balance

    @classmethod
    def set_interest_rate(cls, rate):
        cls._interest_rate = rate

    @staticmethod
    def is_valid_amount(amount):
        return amount > 0

    def calculate_interest(self):
        if BankAccount.is_valid_amount(self.balance):
```

```
            return self.balance * self._interest_rate
        return 0

# Usage
BankAccount.set_interest_rate(0.07)
account = BankAccount(1000)
print(account.calculate_interest())  # Output: 70.0
```

- **Explanation**:
    - `@staticmethod`: Used to validate inputs (e.g., `is_valid_amount`).
    - `@classmethod`: Used to manage class-level attributes (e.g., `_interest_rate`).
    - Instance methods combine their functionality for practical use.

Key Takeaways

- **`@staticmethod`**:
    - Use for pure utility methods.
    - No reliance on class or instance data.
- **`@classmethod`**:
    - Use for alternate constructors or managing shared state.
    - Relies on class-level context (`cls`).