Custom Decorator Creation

A decorator is a **function that takes a function and returns a new function.**

python

```python
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

OUTPUT:

```
Before function call

Hello!

After function call
```

If the decorated function has arguments, the wrapper must accept `*args, **kwargs`.

python

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before call")
        result = func(*args, **kwargs)
        print("After call")
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f"Hello {name}!")

greet("Alice")
```

@my_decorator just memorizes the function named “greet”

Wrapper is triggered when “greet” function is called

*args become “Alice”

*args are used in the memorized

Pre-existing Decorators (not limited to what are shown below)

`@staticmethod`

- Defines a method that does **not access the instance (`self`) or class (`cls`)**.
- Callable on the class or instance.

python

```python
class MyClass:
    @staticmethod
    def greet(name):
        return f"Hello, {name}"

print(MyClass.greet("Alice"))  # Hello, Alice
```

`@classmethod`

- Defines a method that takes the **class itself (`cls`) as the first argument**, instead of the instance.

python Copy code

```python
class MyClass:
    count = 0

    @classmethod
    def increment(cls):
        cls.count += 1
        return cls.count

print(MyClass.increment())  # 1
```

`@abstractmethod` (from `abc` module)

- Used in **abstract base classes**; forces subclasses to implement the method.

python

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

`@property`

- Turns a method into a **read-only attribute**.
- Can be combined with `@<property>.setter` and `@<property>.deleter` for full property control.

python Copy code

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

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        self._radius = value

c = Circle(5)
print(c.radius)  # 5
c.radius = 10
```