Understanding Python Decorators with Real-Life Examples

1. Basic Decorator
Imagine a decorator as a food topping like cheese on a pizza—it enhances the taste but doesn't change the pizza itself.

```
def decorator_example(func):
    def wrapper():
        print("Before the function runs...")
        func()
        print("After the function runs...")
    return wrapper

@decorator_example  # Applying the decorator
def say_hello():
    print("Hello, World!")

say_hello()
```

2. Calling a Decorator in Another Way
Instead of using @decorator, we can manually call the decorator function.

```
def normal_function():
    print("I am a normal function!")

# Applying decorator manually
decorated_function = decorator_example(normal_function)
decorated_function()
```

3. Decorator with Arguments
A decorator can also accept arguments, just like a pizza with extra toppings.

```
def repeat(n):
    def decorator(func):
```

```
        def wrapper():
            for _ in range(n):
                func()
        return wrapper
    return decorator

@repeat(3)  # Calling function 3 times
def greet():
    print("Good Morning!")

greet()
```

4. Decorator for Functions with Arguments
If a function takes arguments, the decorator must handle them too.

```
def smart_divide(func):
    def wrapper(a, b):
        if b == 0:
            print("Oops! Division by zero is not allowed.")
            return
        return func(a, b)
    return wrapper

@smart_divide
def divide(a, b):
    print(f"Result: {a / b}")

divide(10, 2)
divide(5, 0)  # This will be handled gracefully
```

5. Multiple Decorators
You can stack multiple decorators, just like layering multiple toppings on a pizza.

```
def bold(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

def italic(func):
    def wrapper():
        return "<i>" + func() + "</i>"
```

```
    return wrapper

@bold
@italic
def styled_text():
    return "Decorators are cool!"

print(styled_text())  # Output: <b><i>Decorators are cool!</i></b>
```

Summary:
- Decorators enhance functions without modifying their actual code.
- They can be called using @ or manually.
- They can accept arguments.
- Multiple decorators can be stacked.

Decorators are like magic toppings that make Python functions even more powerful! 🚀