Python Decorators - Complete Guide (Basic to Advanced)

1. What is a Decorator?

A decorator is a function that takes another function as input and adds additional functionality to it without changing its original structure.

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

@decorator
def greet():
    print("Hello!")

greet()
```

2. Wrapper Functions & Closures

The wrapper is an inner function that wraps around the original one. It forms a closure, meaning it can access variables from the outer decorator function even after the outer function is done executing.

```
def outer(x):
    def inner():
        print(f"Accessing x from outer: {x}")
    return inner

closure_func = outer(10)
closure_func()
```

3. Handling *args and **kwargs

To handle any number of arguments in the decorator, use *args and **kwargs in your wrapper function.

```
def smart_decorator(func):
    def wrapper(*args, **kwargs):
        print("Arguments:", args, kwargs)
        return func(*args, **kwargs)
    return wrapper
```

```
@smart_decorator
def add(a, b):
    print(a + b)

add(5, 3)
```

4. Using functools.wraps()

wraps(func) from functools preserves the original function's metadata like __name__, __doc__, etc., when wrapping it with another function.

```
from functools import wraps

def logging_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
```

5. Stacked (Multiple) Decorators

Decorators can be layered. The decorator closest to the function is applied first.

```
def deco1(func):
    def wrapper(*args):
        print("Decorator 1")
        return func(*args)
    return wrapper

def deco2(func):
    def wrapper(*args):
        print("Decorator 2")
        return func(*args)
    return wrapper

@deco1
@deco2
def show(a, b):
```

```
    print(a + b)

show(2, 3)
```

6. Logging to File with Decorator

Python's logging module can write logs to a file, capturing arguments, return values, and errors using decorators.

```
import logging
from functools import wraps

logging.basicConfig(filename='log.txt', level=logging.INFO)

def log_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        logging.info(f"Running {func.__name__} with {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper
```

7. Stateful Decorator (Call Counter)

You can add attributes to the wrapper function itself to maintain state like number of calls.

```
from functools import wraps

def count_calls(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        wrapper.call_count += 1
        print(f"Call #{wrapper.call_count}")
        return func(*args, **kwargs)
    wrapper.call_count = 0
    return wrapper

@count_calls
def say_hi():
    print("Hi!")
```

```
say_hi()
say_hi()
```

8. Decorators with Arguments

Decorators can take parameters if you wrap them inside another function.

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

9. Class-Based Decorators

Using classes as decorators helps when you want to maintain complex state or configuration.

```
class CallCounter:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"Called {self.count} times")
        return self.func(*args, **kwargs)

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

say_hello()
say_hello()
```