Notes on Python Decorators

🔹 What are Decorators?

- A **decorator** is a wrapper around a function.
- They allow us to **add extra functionality** to an existing function **without modifying its code**.
- Think of it like adding *sprinkles on coffee* – it enhances or changes slightly but the base remains the same.

🔹 Why use Decorators?

- To **reuse code** for tasks like:
    - Logging
    - Authorization checks
    - Measuring execution time
    - Caching results
- Helps in **keeping functions clean** while still applying extra behavior.

🔹 Basic Idea

1. A decorator is a **function that takes another function as input**.
2. Inside, it defines a **wrapper function**.
3. The wrapper can:
    - Run some code **before** the main function.

    - Call the original function.
    - Run some code **after** the main function.
4. Return the wrapper instead of the original function.

🔹 Syntax

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function runs")
        result = func(*args, **kwargs)   # call the original function
        print("After function runs")
        return result
    return wrapper
```

Apply using `@`:

```
@my_decorator
def greet():
    print("Hello from decorators class!")

greet()
```

Output:

```
Before function runs
Hello from decorators class!
After function runs
```

🔹 Problem with Decorators

- When you decorate, the function's **metadata** changes.
  Example:

```
print(greet.__name__)
```

Instead of `greet`, it may show `wrapper`.

🔹 Solution → `functools.wraps`

- Use `wraps` from `functools` to preserve metadata.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before function runs")
        result = func(*args, **kwargs)
        print("After function runs")
        return result
    return wrapper
```

Now:

```
print(greet.__name__)  # outputs: greet
```

🔹 Key Takeaways

- Decorators = wrappers around functions.
- Add functionality **before and after execution**.
- Use `@decorator_name` instead of manually wrapping.
- Use `functools.wraps` to preserve function’s **name & metadata**.

Key Takeaways from the Video

1. Decorators in Python

- A decorator is a function that **wraps another function** to extend or modify its behavior.
- You can use decorators from libraries (Django, FastAPI, etc.) or make your own.

---

2. `@wraps` Usage

- Always import it from `functools`:

```
from functools import wraps
```

- When creating a custom decorator, use `@wraps(func)` inside the decorator **to preserve metadata** (`__name__`, `__doc__`, etc.) of the original function.
- Example:

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

---

3. Handling Arguments

- Use `*args` and `**kwargs` in the wrapper to accept **any number of positional and keyword arguments**, so your decorator works with any function:

```
def wrapper(*args, **kwargs):
    return func(*args, **kwargs)
```

---

4. Logging Example

- Before calling the function, print a message.
- After calling the function, print another message.
- This pattern is useful for debugging or tracking function calls.

```
@log_activity
def brew_chai(type_of_tea):
    print(f"Brewing {type_of_tea}")
```

Output:

```
Calling brew_chai...
Brewing masala chai
Finished brew_chai
```

---

5. Benefits

- You can **add logging** (or other behavior) **without touching the original function**.
- Works with default values and extra keyword arguments easily.

Key Concepts Covered

1. **Purpose of the Decorator**
    - Restricts access to a function so that only users with a specific role (e.g., `admin`) can execute it.
    - Useful in frameworks like Django for access control, e.g., restricting certain views or APIs.
2. **Basic Decorator Structure**
    - Import `wraps` from `functools` to preserve metadata of the decorated function.
    - Define a **wrapper function** inside the decorator to add the additional behavior.
    - The wrapper can accept:
        - A **specific argument** if you know it (like `user_role`).
        - `*args` and `**kwargs` if the number of parameters is variable.
3. **Checking User Role**
    - Inside the wrapper, check if the `user_role` is `"admin"`.
    - If not, print an **access denied message**.
    - If it is, call the original function and return its result.
4. **Returning the Wrapper**
    - Always return the wrapper from the decorator to make it functional.

Using the Decorator

```
@require_admin
def access_tea_inventory(user_role):
    print("Access granted to tea inventory")
```

5.
6. **Common Pitfalls**

    - Forgetting to explicitly return something in the wrapper can sometimes cause unexpected behavior.
    - In newer Python versions, if the wrapper doesn't return, it implicitly returns `None`, so the code may still run.
    - It's safer to explicitly `return None` when access is denied.

7. **Behavior Observed**
    - Running the decorated function with a non-admin role prints "Access denied".
    - Running it with `"admin"` executes the function normally.
    - Explicit returns make the code more predictable and maintainable.

✅ Takeaways

- Role-based decorators are simple yet powerful for access control.
- Always use `wraps` to maintain the original function's metadata.
- Explicit returns in the wrapper prevent subtle bugs, especially in older Python versions or complex decorators.