Python Decorators

This video provides a tutorial on Python decorators.

What are Decorators?

• Decorators are functions that take another function as an argument, modify it, and then return the modified function.
• The video's host explains that decorators are used to add functionality to an existing function without changing its code. He gives an example of adding "Good Morning" and "Thanks for using this function" messages before and after a function's primary task.

How to create a Decorator

The video provides an example of creating a decorator named `greet`:

• A decorator function takes a function as an argument. Inside `greet`, another function `mffx` (modified effects) is defined. `mffx` is the wrapper function that contains the new functionality, such as printing "Good Morning" and "Thanks for using this function," and calls the original function.
• The decorator then returns the wrapper function `mffx`.

Using a Decorator

The video demonstrates two ways to use the decorator:

Using the `@` symbol

The `@` symbol is syntactic sugar that simplifies the process. Placing `@greet` above a function definition, like `hello()`, automatically decorates it.

```
@greet
def hello():
    print("Hello World")

hello()
```

Explicitly calling the decorator

You can pass the function to the decorator and then call the returned function.

```
greet(hello)()
```

Handling Function Arguments

The video addresses a common issue where a decorated function might take arguments. The host shows that to handle this, the inner wrapper function must accept arguments using `*args` and `**kwargs` and then pass them to the original function call.

```
def greet(fx):
    def mffx(*args, **kwargs):
        print("Good Morning")
        fx(*args, **kwargs)
        print("Thanks for using this function")
    return mffx
```

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

add(1, 2)
```