No.	Questions
	What is Decorators?

Ques. What is Decorators?

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are usually called before the definition of a function you want to decorate.
- Decorators are used to add some design patterns to a function without changing its structure.
- A decorator function is a function that accepts a function as parameter and return a function(decorator ek function hai jo as a argument leta bhi function hai and return bhi function karta hai).
- Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it.

```
# A simple decorator function
def decorator(func):

    def wrapper():
        print("Before calling the function.")
        func()
        print("After calling the function.")
    return wrapper

# Applying the decorator to a function
@decorator

def greet():
    print("Hello, Mohit!")

greet()

Output:-
Before calling the function.
Hello, Mohit!
After calling the function.
```

Explanation:

- decorator takes the greet function as an argument.
- It returns a new function (wrapper) that first prints a message, calls greet() and then prints another message.
- The @decorator syntax is a shorthand for greet = decorator(greet).

Syntax of Decorator Parameters

```
def decorator_name(func):
    def wrapper(*args, **kwargs):
        # Add functionality before the original function call
        result = func(*args, **kwargs)
        # Add functionality after the original function call
        return result
    return wrapper


@decorator_name
def function_to_decorate():
    # Original function code
    pass
```

Explanation of Parameters(Syntax of Decorator Parameters)

1. decorator_name(func):
    1. decorator_name: This is the name of the decorator function.
    2. func: This parameter represents the function being decorated. When you use a decorator, the decorated function is passed to this parameter.
2. wrapper(*args, **kwargs):
    1. wrapper: This is a nested function inside the decorator. It wraps the original function, adding additional functionality.
    2. *args: This collects any positional arguments passed to the decorated function into a tuple.
    3. **kwargs: This collects any keyword arguments passed to the decorated function into a dictionary.
    4. The wrapper function allows the decorator to handle functions with any number and types of arguments.
3. @decorator_name:
    1. This syntax applies the decorator to the function_to_decorate function. It is equivalent to writing function_to_decorate = decorator_name(function_to_decorate).

- 2nd Type

```
def decor(fun1):
    def inner():
        print('befor inhance')
        fun1()
        print('after inhance')
    return inner

@decor
def num():
    print('hello mohit')
```

```
num()

Output:-
befor inhance
hello mohit
after inhance
```

```
def num_decor(num):
    def inner():
        a = num()
        add = a + 5
        return add
    return inner

@num_decor
def num():
    return 10

print(num())

Output:- 15

# Example:-
def num_decor(num):
    def inner():
        a = num()
        add = a + 5
        return add
    return inner

def num():
    return 10

result = num_decor(num)
print(result())

Output:- 15
```

- Upper case decorater

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper
```

```
def say_hi():
    return 'hello there'

# call decorater
decorate = uppercase_decorator(say_hi)
print(decorate())

# -------2nd Opton call decorater----------
@uppercase_decorator
def say_hi():
    return 'hello there'

print(say_hi())
--------------------------

Output:-'HELLO THERE'
```

Higher-Order Functions

- In Python, higher-order functions are functions that take one or more functions as arguments, return a function as a result or do both. Essentially, a higher-order function is a function that operates on other functions.
- **Key Properties of Higher-Order Functions:**
    - Taking functions as arguments: A higher-order function can accept other functions as parameters.
    - Returning functions: A higher-order function can return a new function that can be called later.

```
# A higher-order function that takes another function as an argument
def fun(f, x):
    return f(x)

# A simple function to pass
def square(x):
    return x * x

# Using apply_function to apply the square function
res = fun(square, 5)
print(res)

# Output:
25
```