
YASH
Technologies
SM
More than what you think.
Python Decorator
© 2017 YASH Technologies | www.yash.com | Confidential


4
Function in python
Python Function
Con 1: In python everything treated as object including function.
A function is an object. Because of that, a function can be assigned to a variable. The function can be accessed from that variable.
Ex:
def my_function():
print('I am a function.')
# Assign the function to a variable without parenthesis. We don't want to execute the function.
description = my_function
description()


5
Function in python
Ex:
def fun1():
print("Hello")
print(fun1)
print(id(fun1))
Con 2: Function Aliasing
def fun1():
print("Hello")
fun1() #Here fun1 pointing to fun object.
f2=fun1 #Now f2 also printing to same fun1 object.
print(id(fun1))
print(id(f2))
#Now we can call this function either by using
#f2 or by using fun1
f2()

Function in python : del

Ex 2:

```
def fun1():
    print("Hello")
fun1()  #Here fun1 pointing to fun object
f2=fun1 #Now f2 also pointing to same fun1 object
print(id(fun1))
print(id(f2))
#Now we can call this function either by using
#f2 or by using fun1
f2()
del f2
#Here i am deleting wish function
#But here we are only deleting reference variable
fun1()
f2() # Now it is invalid because we deleted it.
```

YASH Technologies

More than what you thi

Nested Function in python

Con 3: Nested function. A function inside another function is called nested function.

```
def outer():
    print("This is outer fun")
    def inner():
        print("This is Inner")
    inner()
    return inner
k=outer()
k()
```

Function in python

First-Class Objects

In Python, functions are first-class objects. This means that **functions can be passed around and used as arguments**, just like any other object (string, int, float, list, and so on).

That's why, we can assign a function to a variable, pass it to a function or return it from a function. Just like any other variable.

```
def say_hello(name):
    return f"Hello {name}"

def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

def greet_bob(greeter_func):
    return greeter_func("Aarvi")
```

Python Function as a argument

Con 4: We can pass function as a argument to another function.

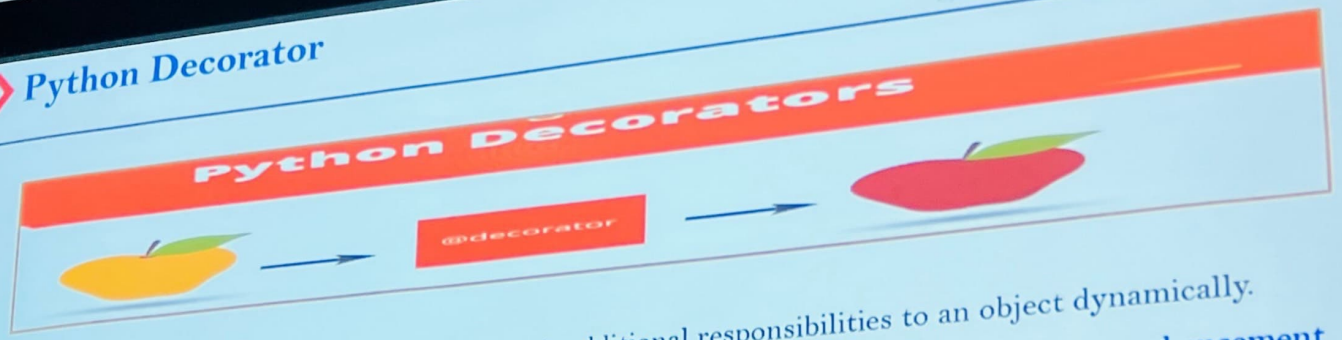
Ex: filter() map()

Returning Functions From Functions

Python also allows you to use functions as return values. The following example returns one of the inner functions from the outer parent() function:

```
def parent(num):
  def first_child():
    return "Hi, I am Emma"

  def second_child():
    return "Call me Liam"

  if num == 1:
    return first_child
  else:
    return second_child
```

Python Decorator



In python, decorator is a design pattern that adds additional responsibilities to an object dynamically.

Decorator is a function; Decorator is always going to take a function as argument and do some enhancement and return some output function.

This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

A decorator in Python is a function that modifies the behaviour of another function without changing its actual code.

Basic Concept

A decorator is a function that takes another function as an argument, adds some functionality, and returns another function without altering the source code of the original function.

Basic Syntax

```
@decorator
def function():
    pass
```

This is equivalent to:

```
def function():
    pass
function = decorator(function)
```

Why Do We Need Decorators?

Decorators help in:

- **Code reusability** — common logic (like logging, validation, authentication) can be reused across multiple functions.
- **Separation of concerns** — keeps "core business logic" clean.
- **Pre-processing or post-processing** — you can execute some code *before* or *after* a function runs.

YASH Technologies
More than what you think.

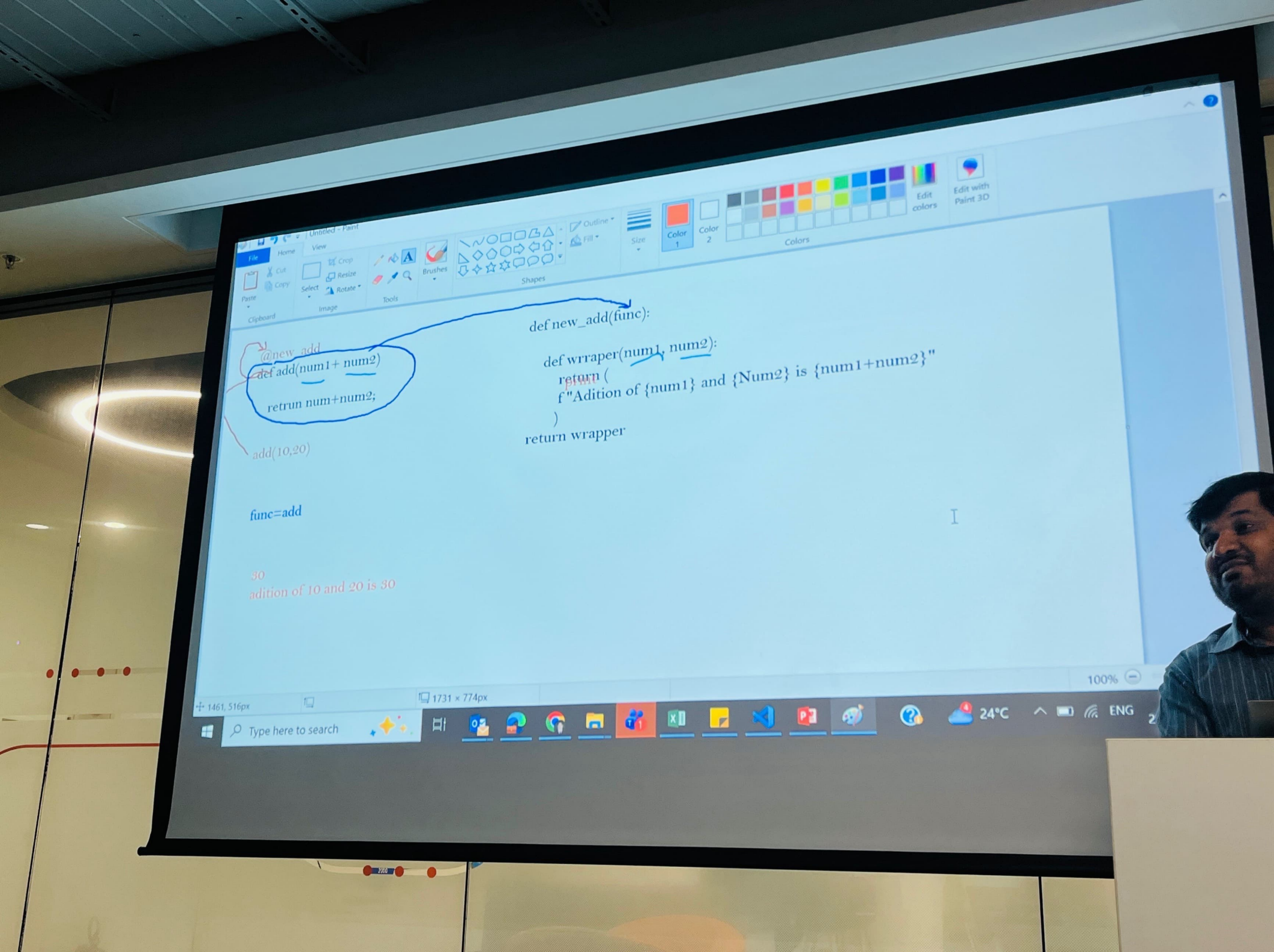
Simple Example

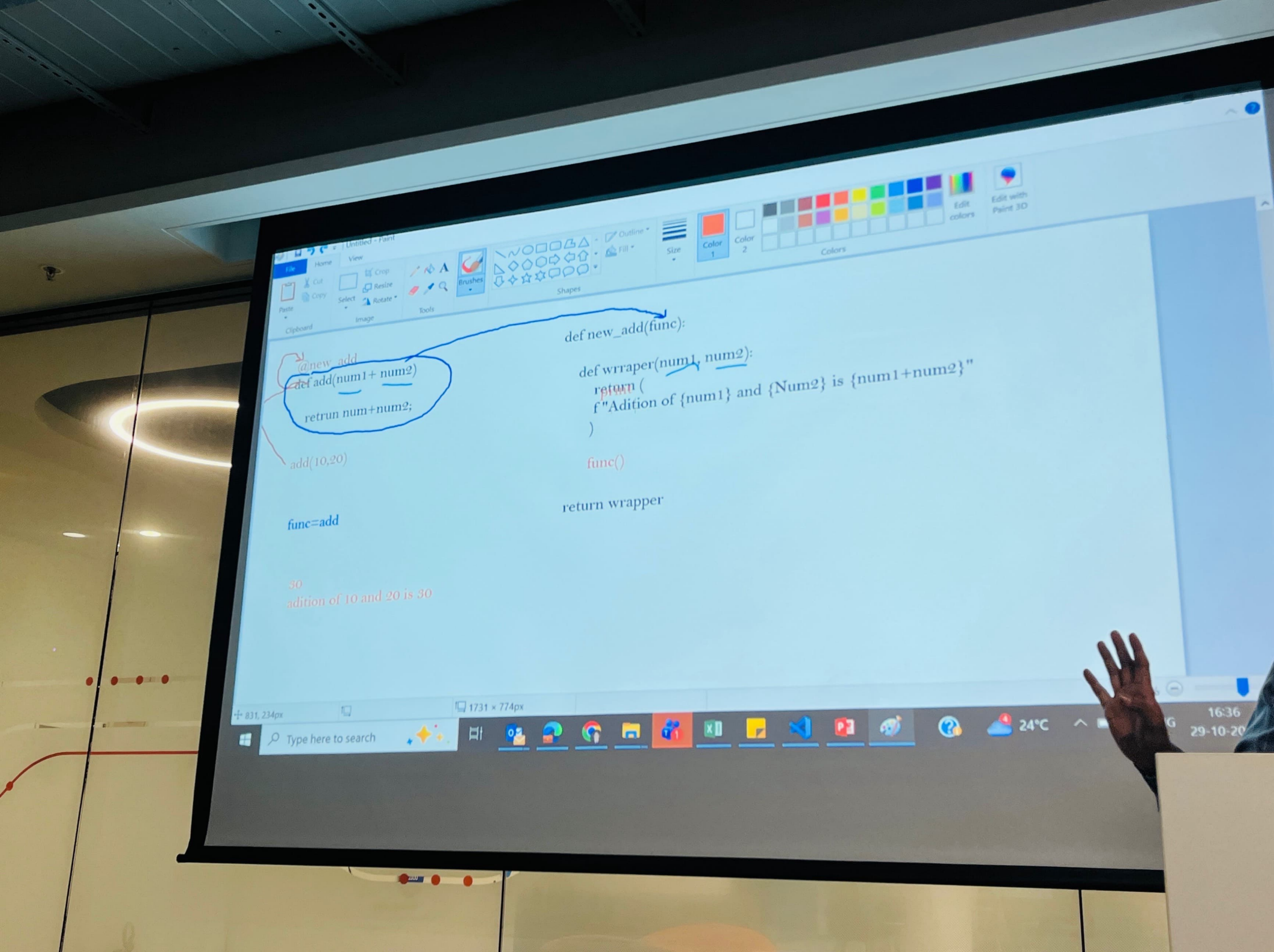
```
def simple_decorator(func):
    def wrapper():
        print("Something is happening before the function is
              called.")
        func()
        print("Something is happening after the function is c
              alled.")
    return wrapper
```

```
@simple_decorator
def say_hello():
    print("Hello!")
say_hello()
```

How It Works

- my_decorator is a function that takes func as a parameter
- Inside, it defines a wrapper function that adds behavior before and after calling func
- The decorator returns the wrapper function
- When you use @my_decorator, it's like writing: say_hello = my_decorator(say_hello)


Untitled - Paint
@new_add
def add(num1+ num2)
retrun num+num2;
add(10,20)
func=add
30
adition of 10 and 20 is 30
def new_add(func):
def wrraper(num1, num2):
return (
f"Adition of {num1} and {Num2} is {num1+num2}"
)
return wrapper
1461, 516px
1731 × 774px
100%
Type here to search
24°C
ENG


@new_add
def add(num1+ num2)
retrun num+num2;
add(10,20)
func=add
30
adition of 10 and 20 is 30
def new_add(func):
def wrraper(num1, num2):
return (
f"Adition of {num1} and {Num2} is {num1+num2}"
)
func()
return wrapper
Type here to search
24°C
16:36
29-10-20

Decorators with Arguments

To handle functions with arguments, use *args and **kwargs:

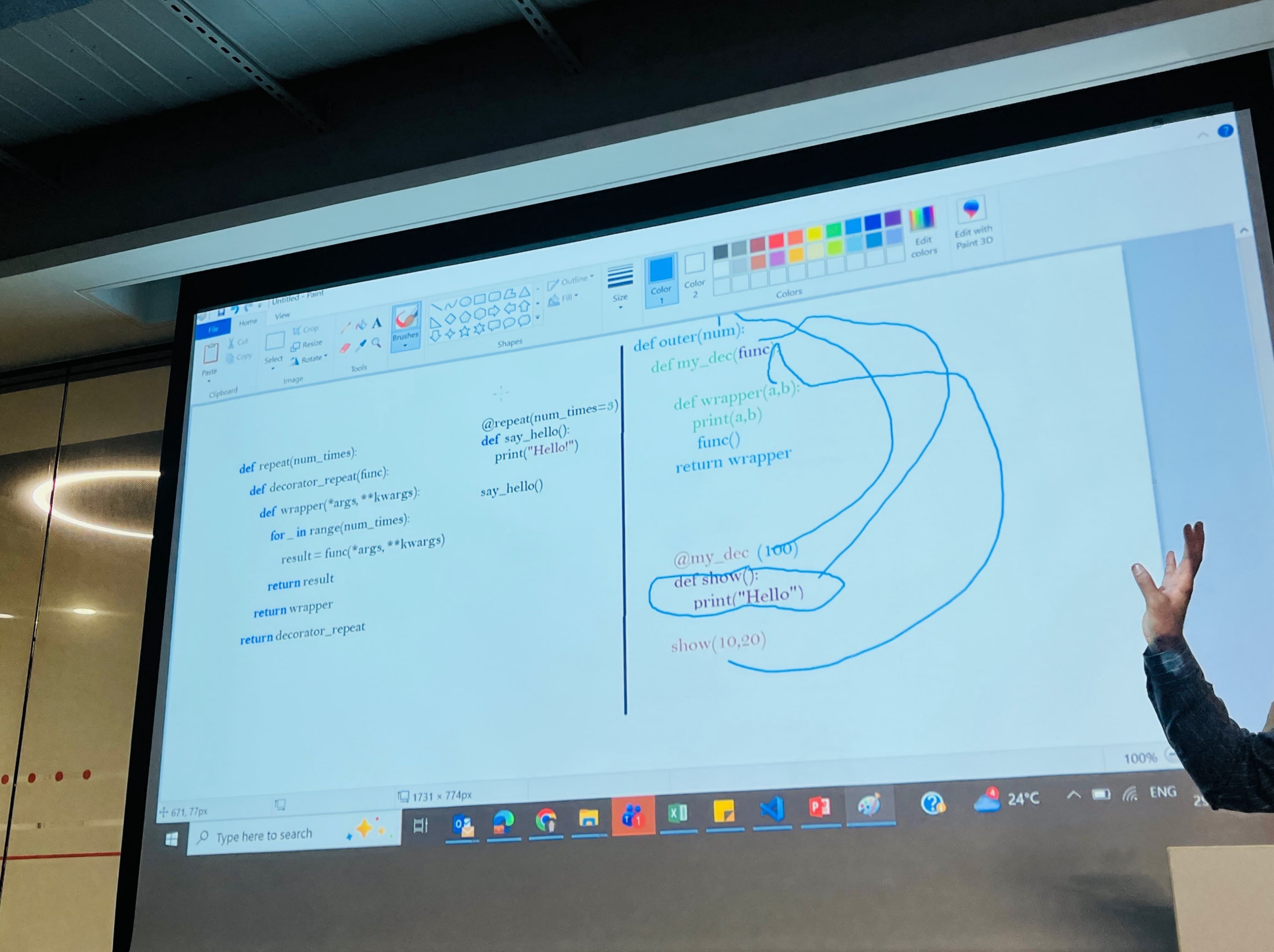
```
def decorator_with_args(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        print(f"Arguments: {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"Function {func.__name__} completed")
        return result
    return wrapper
```

```
@decorator_with_args
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Ayush"))
print(greet("Amit", greeting="Hi"))
```

Decorators with Their Own Arguments

```
def repeat(num_times):
    def decorator_repeat(func):
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator_repeat
```

```
@repeat(num_times=3)
def say_hello():
    print("Hello!")

say_hello()
```


Untitled - Paint
File
Home
View
Paste
Cut
Copy
Clipboard
Select
Crop
Resize
Rotate
Image
Tools
Brushes
Shapes
Outline
Fill
Size
Color 1
Color 2
Colors
Edit colors
Edit with Paint 3D
def repeat(num_times):
def decorator_repeat(func):
def wrapper(*args, **kwargs):
for _ in range(num_times):
result = func(*args, **kwargs)
return result
return wrapper
return decorator_repeat
@repeat(num_times=3)
def say_hello():
print("Hello!")
say_hello()
def outer(num):
def my_dec(func):
def wrapper(a,b):
print(a,b)
func()
return wrapper
@my_dec (100)
def show():
print("Hello")
show(10,20)
100%
671, 77px
1731 × 774px
Type here to search
24°C
ENG

What is @functools.wraps?

When you create a decorator, it **wraps** the original function with a new one. But this can cause the original function's metadata (like its name, docstring, etc.) to be **lost**.

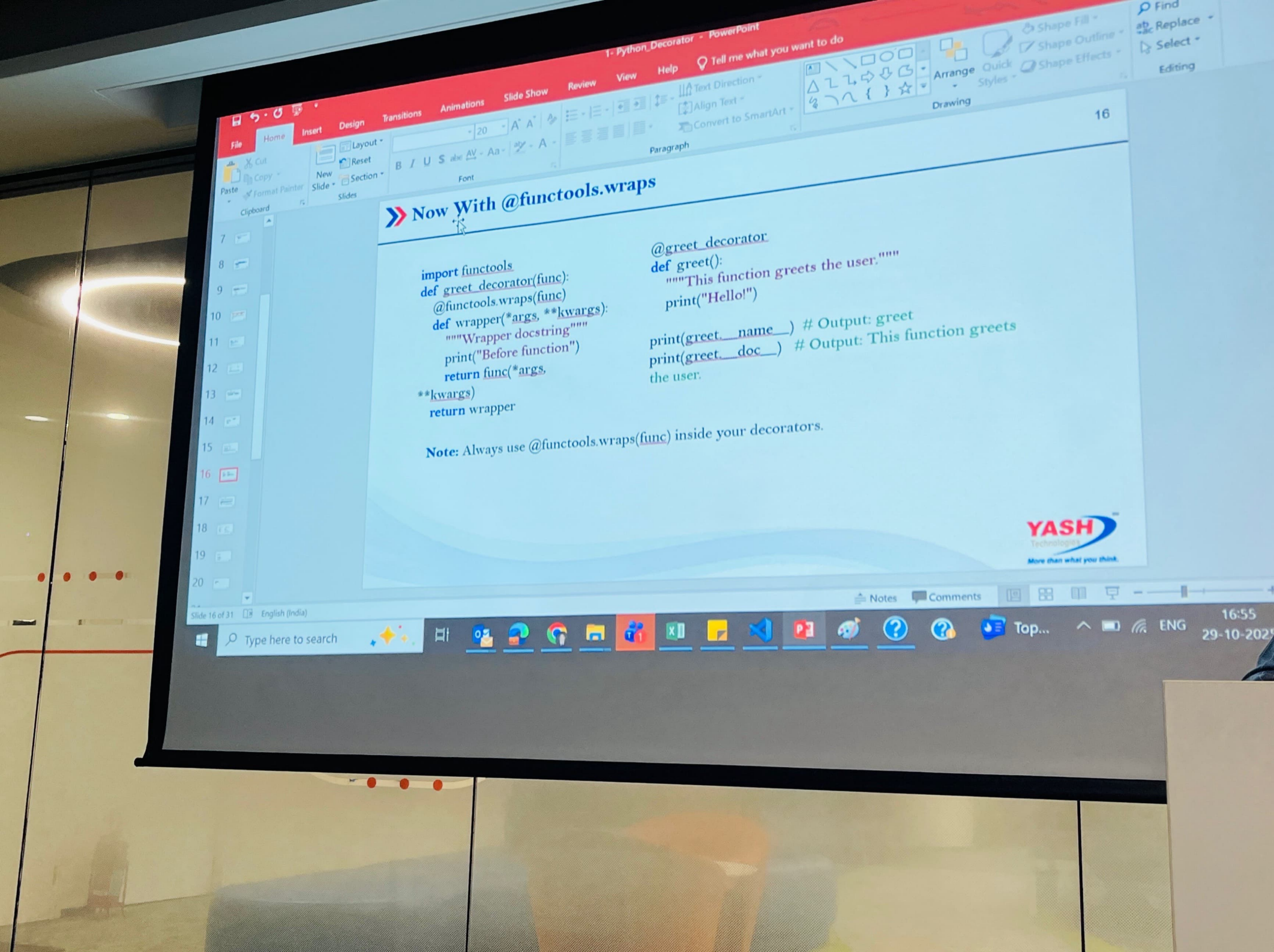
@functools.wraps is used to **preserve** that metadata.

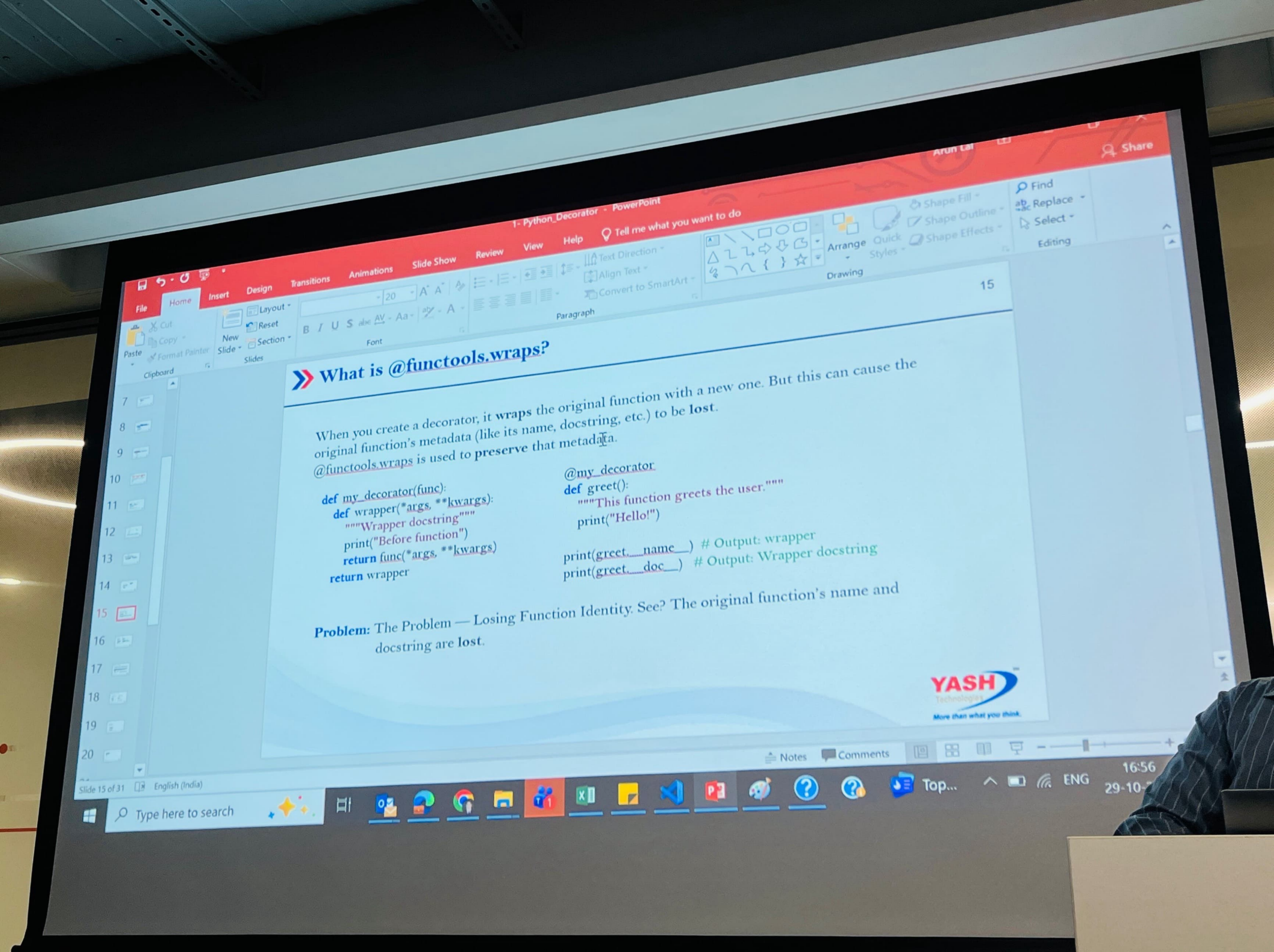
```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        """Wrapper docstring"""
        print("Before function")
        return func(*args, **kwargs)
    return wrapper
```

```
@my_decorator
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__name__)  # Output: wrapper
print(greet.__doc__)   # Output: Wrapper docstring
```

Problem: The Problem — Losing Function Identity. See? The original function's name and docstring are **lost**.

YASH Technologies
More than what you think


1- Python_Decorator - PowerPoint
Tell me what you want to do
Now With @functools.wraps
16
import functools
def greet_decorator(func):
@functools.wraps(func)
def wrapper(*args, **kwargs):
"""Wrapper docstring"""
print("Before function")
return func(*args,
**kwargs)
return wrapper
@greet_decorator
def greet():
"""This function greets the user."""
print("Hello!")
print(greet.__name__) # Output: greet
print(greet.__doc__) # Output: This function greets
the user.
Note: Always use @functools.wraps(func) inside your decorators.
YASH
Technologies
Move than what you think.
Notes
Comments
Slide 16 of 31
English (India)
Type here to search
ENG
16:55
29-10-202


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
15
What is @functools.wraps?
When you create a decorator, it wraps the original function with a new one. But this can cause the original function's metadata (like its name, docstring, etc.) to be lost.
@functools.wraps is used to preserve that metadata.
def my_decorator(func):
def wrapper(*args, **kwargs):
"""Wrapper docstring"""
print("Before function")
return func(*args, **kwargs)
return wrapper
@my_decorator
def greet():
"""This function greets the user."""
print("Hello!")
print(greet.__name__) # Output: wrapper
print(greet.__doc__) # Output: Wrapper docstring
Problem: The Problem — Losing Function Identity. See? The original function's name and docstring are lost.
YASH
More than what you think.
Notes
Comments
Slide 15 of 31
English (India)
Type here to search
ENG
16:56

Now With @functools.wraps

```
import functools
def greet_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        """Wrapper docstring"""
        print("Before function")
        return func(*args,
**kwargs)
    return wrapper

@greet_decorator
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__name__) # Output: greet
print(greet.__doc__) # Output: This function greets
the user.
```

Note: Always use @functools.wraps(func) inside your decorators.

Python Decorator Chaining

Decorator chaining (or stacking) is the process of applying multiple decorators to a single function. The decorators are executed in a specific order, creating a "wrapper chain" around the original function.

Basic Concept

When you stack decorators, they form layers around your function, like an onion. The decorator closest to the function definition is applied first, and the one farthest is applied last.

Syntax

```
@decorator1
@decorator2
@decorator3def
my_function(): pass
```

This is equivalent to:

```
def my_function():
    pass
my_function=decorator1(decorator2(decorator3(my_function))
```

YASH Technologies
More than what you think.

17

Slide 17 of 31 English (India)

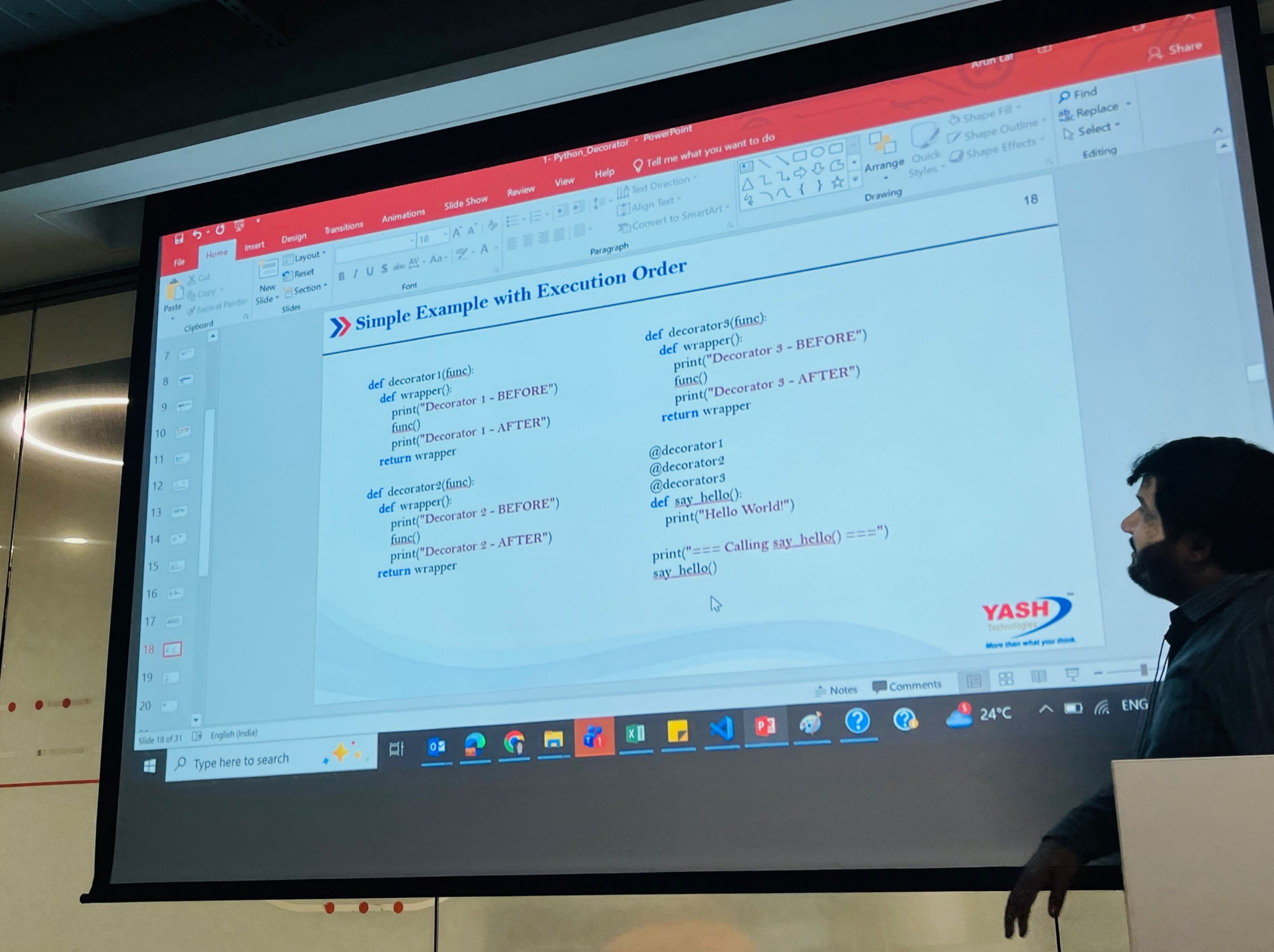
Notes Comments

Type here to search

Earn...

ENG

16:56
29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
Simple Example with Execution Order
18
def decorator1(func):
def wrapper():
print("Decorator 1 - BEFORE")
func()
print("Decorator 1 - AFTER")
return wrapper
def decorator2(func):
def wrapper():
print("Decorator 2 - BEFORE")
func()
print("Decorator 2 - AFTER")
return wrapper
def decorator3(func):
def wrapper():
print("Decorator 3 - BEFORE")
func()
print("Decorator 3 - AFTER")
return wrapper
@decorator1
@decorator2
@decorator3
def say_hello():
print("Hello World!")
print("=== Calling say_hello() ===")
say_hello()
YASH
Technologies
Move than what you think
Notes
Comments
Slide 18 of 31
English (India)
Type here to search
24°C
ENG

1- Python_Decorator - PowerPoint

19

Execution Flow:

Note: The closest decorator to the function runs first when the function is called

1. decorator3 wraps say_hello
2. decorator2 wraps the result from decorator3
3. decorator1 wraps the result from decorator2
4. When called, execution goes: decorator1 → decorator2 → decorator3 → original function

Output:

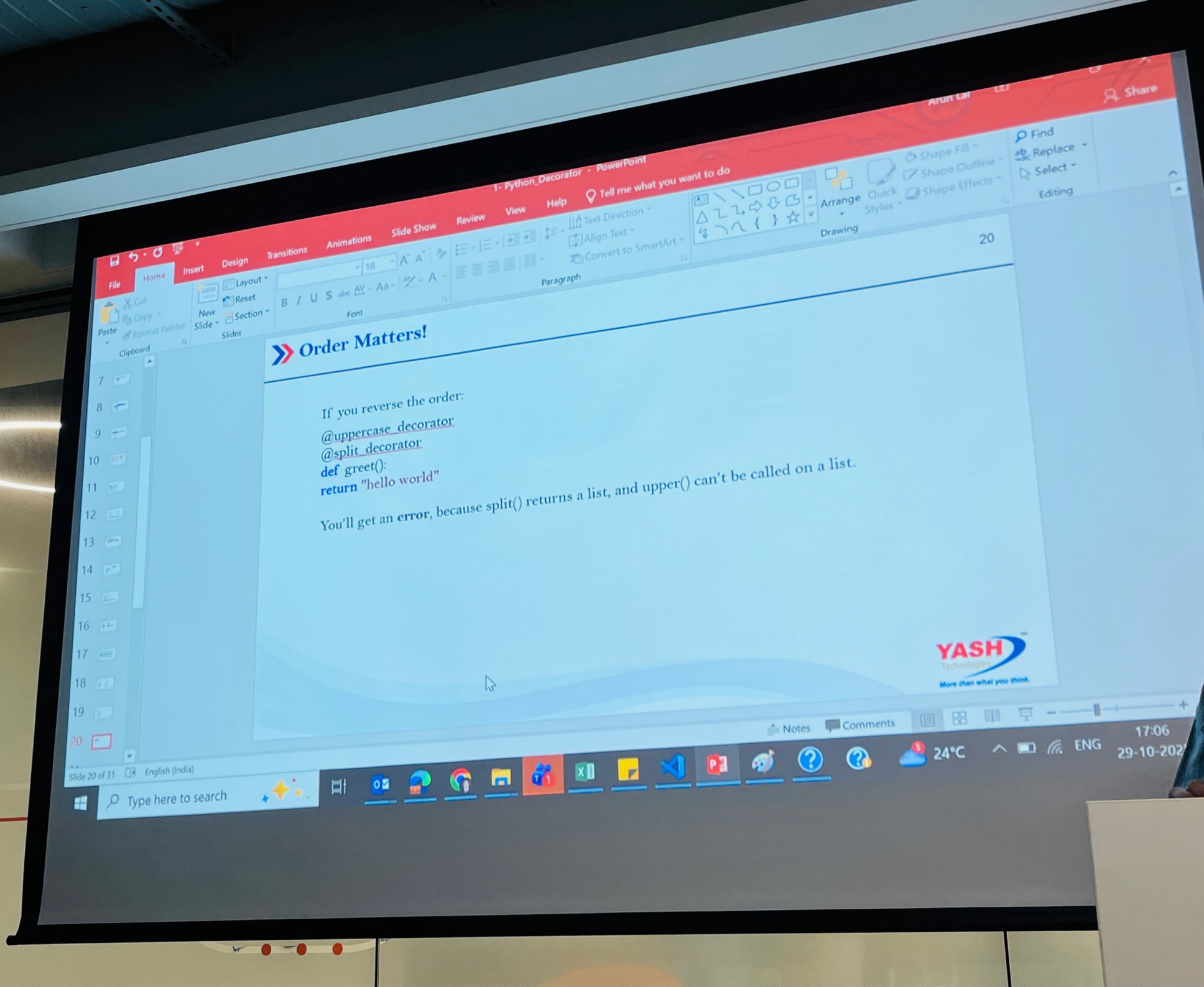
```
=== Calling say_hello() ===
Decorator 1 - BEFORE
Decorator 2 - BEFORE
Decorator 3 - BEFORE
Hello World!
Decorator 3 - AFTER
Decorator 2 - AFTER
Decorator 1 - AFTER
```

YASH Technologies
More than what you think.

Slide 19 of 31 English (India)

Type here to search

24°C ENG 17:06 29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
20
Order Matters!
If you reverse the order:
@uppercase_decorator
@split_decorator
def greet():
return "hello world"
You'll get an error, because split() returns a list, and upper() can't be called on a list.
YASH
Notes
Comments
Slide 20 of 31
English (India)
Type here to search
24°C
ENG
17:06

1- Python_Decorator - PowerPoint

Class-Based Decorators

A Python class decorator is a class that wraps a function inside an instance of the class, allowing you to modify or extend the original function's behavior without changing the source code. Here's how to apply it. It needs two special methods:

- __init__: Called when the decorator is applied (receives the function)
- __call__: Called when the decorated function is executed

```
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("Before function call")
        result = self.func(*args, **kwargs)
        print("After function call")
        return result
```

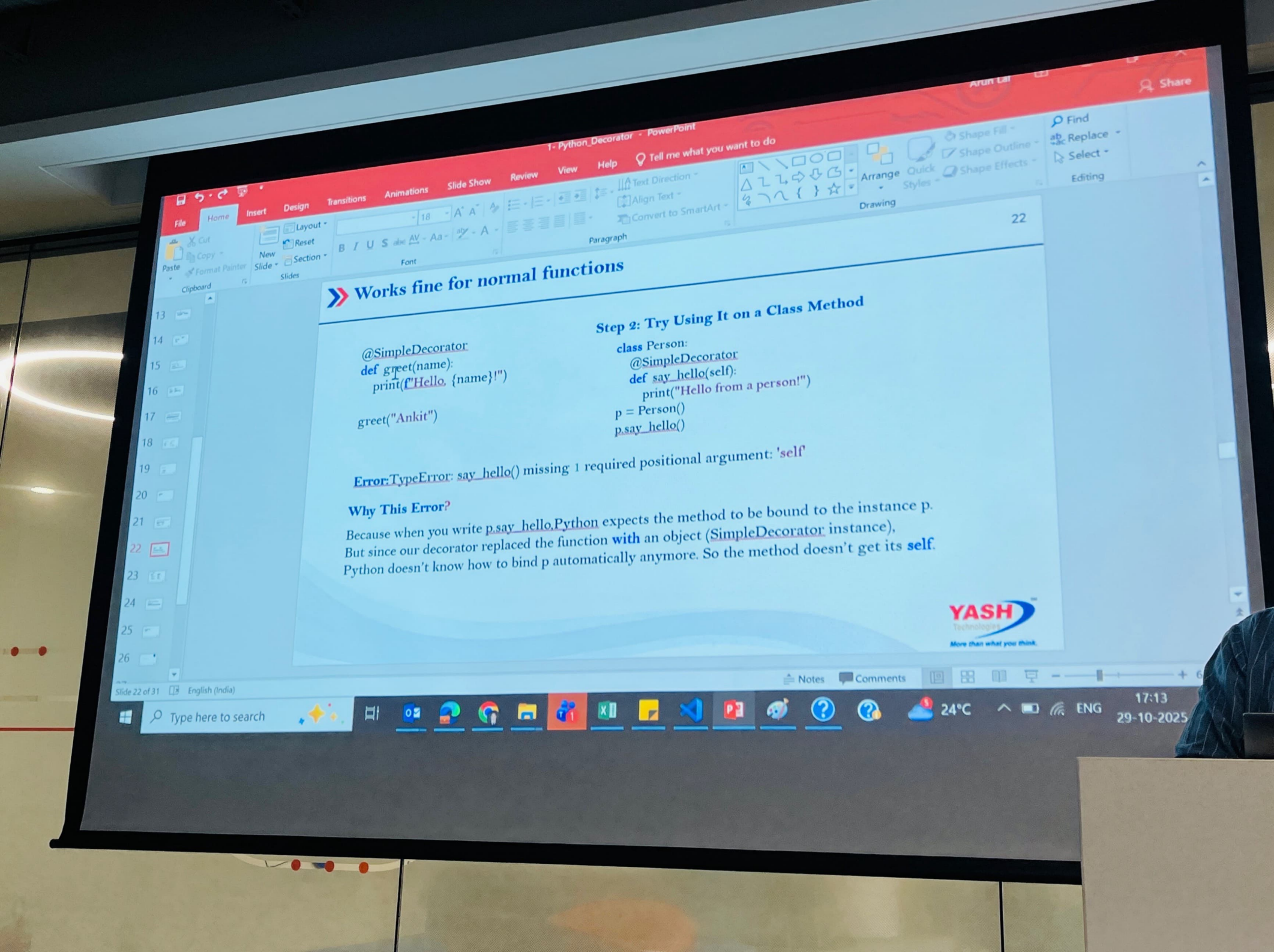
```
@MyDecorator
def greet(name):
    print(f"Hello, {name}!")

greet("Ankit")
```

YASH Technologies
More than what you think.

Slide 21 of 31 English (India)

Type here to search

24°C ENG 17:06 29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
22
Works fine for normal functions
@SimpleDecorator
def greet(name):
print(f"Hello, {name}!")
greet("Ankit")
Step 2: Try Using It on a Class Method
class Person:
@SimpleDecorator
def say_hello(self):
print("Hello from a person!")
p = Person()
p.say_hello()
Error:TypeError: say_hello() missing 1 required positional argument: 'self'
Why This Error?
Because when you write p.say_hello,Python expects the method to be bound to the instance p.
But since our decorator replaced the function with an object (SimpleDecorator instance),
Python doesn't know how to bind p automatically anymore. So the method doesn't get its self.
YASH
Technologies
More than what you think.
Notes
Comments
Slide 22 of 31
English (India)
Type here to search
24°C
ENG
17:13
29-10-2025

Fix It Using __get__

```python
from functools import partial
class BetterDecorator:
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, objtype=None):
        # When accessed from instance, bind 'obj'
        #  (the instance)
        return partial(self.__call__, obj)

    def __call__(self, *args, **kwargs):
        print(" Before function call")
        result = self.func(*args, **kwargs)
        print(" After function call")
        return result
```

Now let's test again

```python
class Person:
    @BetterDecorator
    def say_hello(self):
        print("Hello from a person!")
p = Person()
p.say_hello()
```

Output:

```
Before function call
Hello from a person!
After function call
```

YASH Technologies
More than what you think

1 - Python_Decorator - PowerPoint

Tell me what you want to do

Share

24

What Happened Behind the Scenes:

When you do p.say_hello, Python sees that say_hello is a BetterDecorator object.
It calls its __get__() method.

__get__() returns partial(**self.__call__**, obj) —
a function that will automatically **pass** obj (the instance p)
as the first argument when called.

Note: __get__ in a class-based decorator is what makes it work correctly with **instance methods** — it ensures the self of the class is automatically passed to the decorated function.

Note: Conclusion, while some may class based decorators prefer, I personally do not prefer class based decorator because of the extra boilerplate code to fix the doc strings.

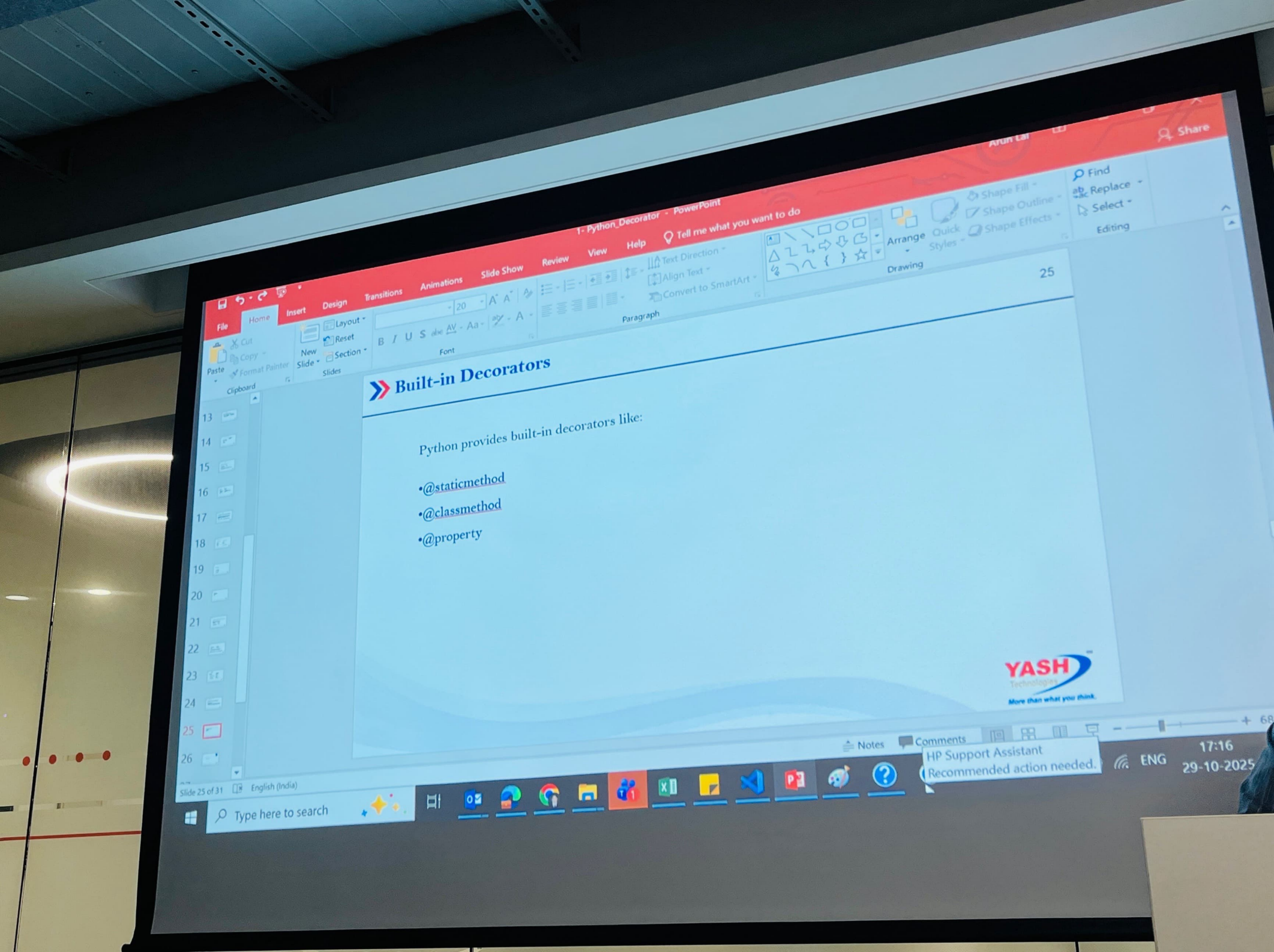
YASH Technologies
More than what you think

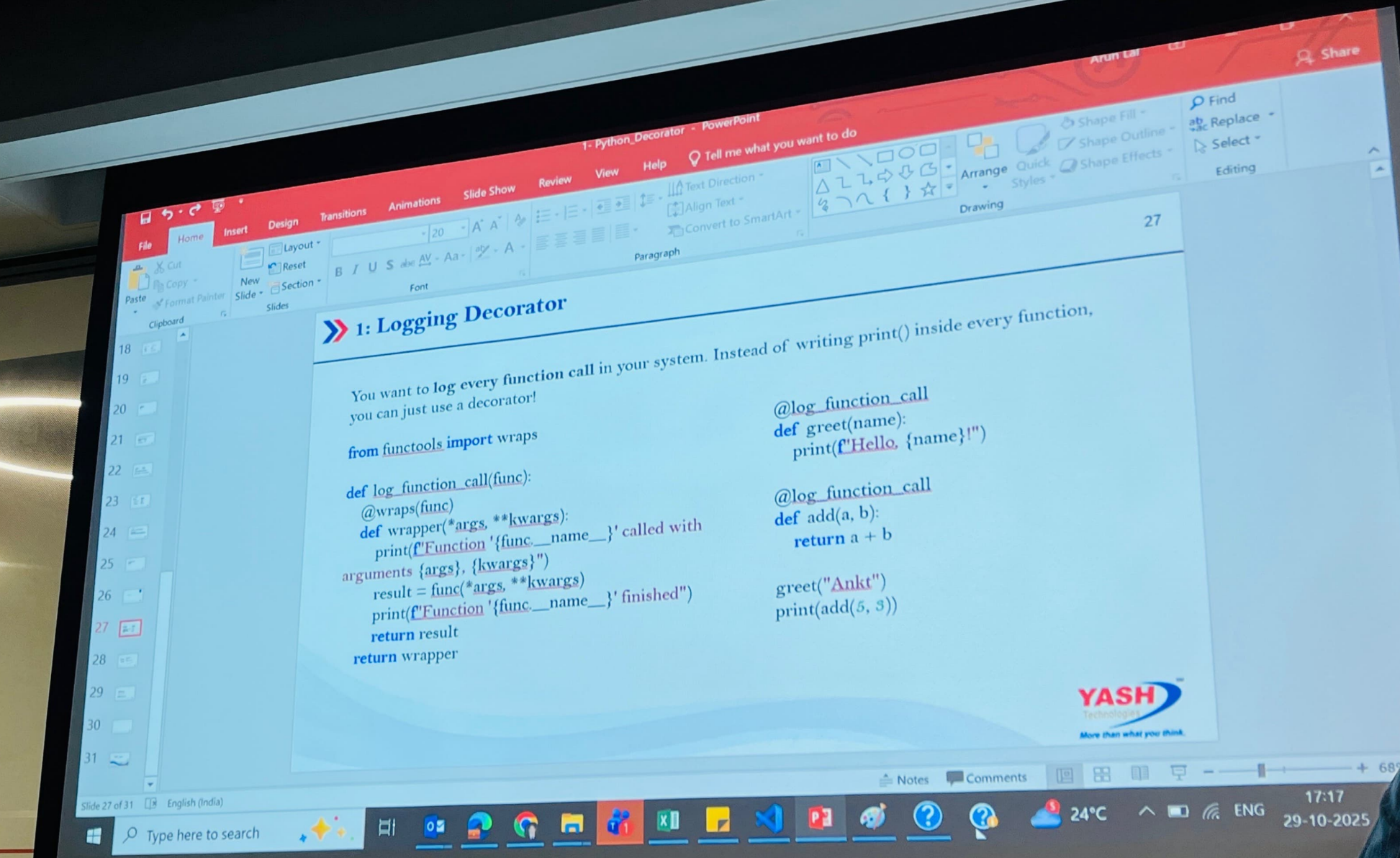
Slide 24 of 31 English (India)

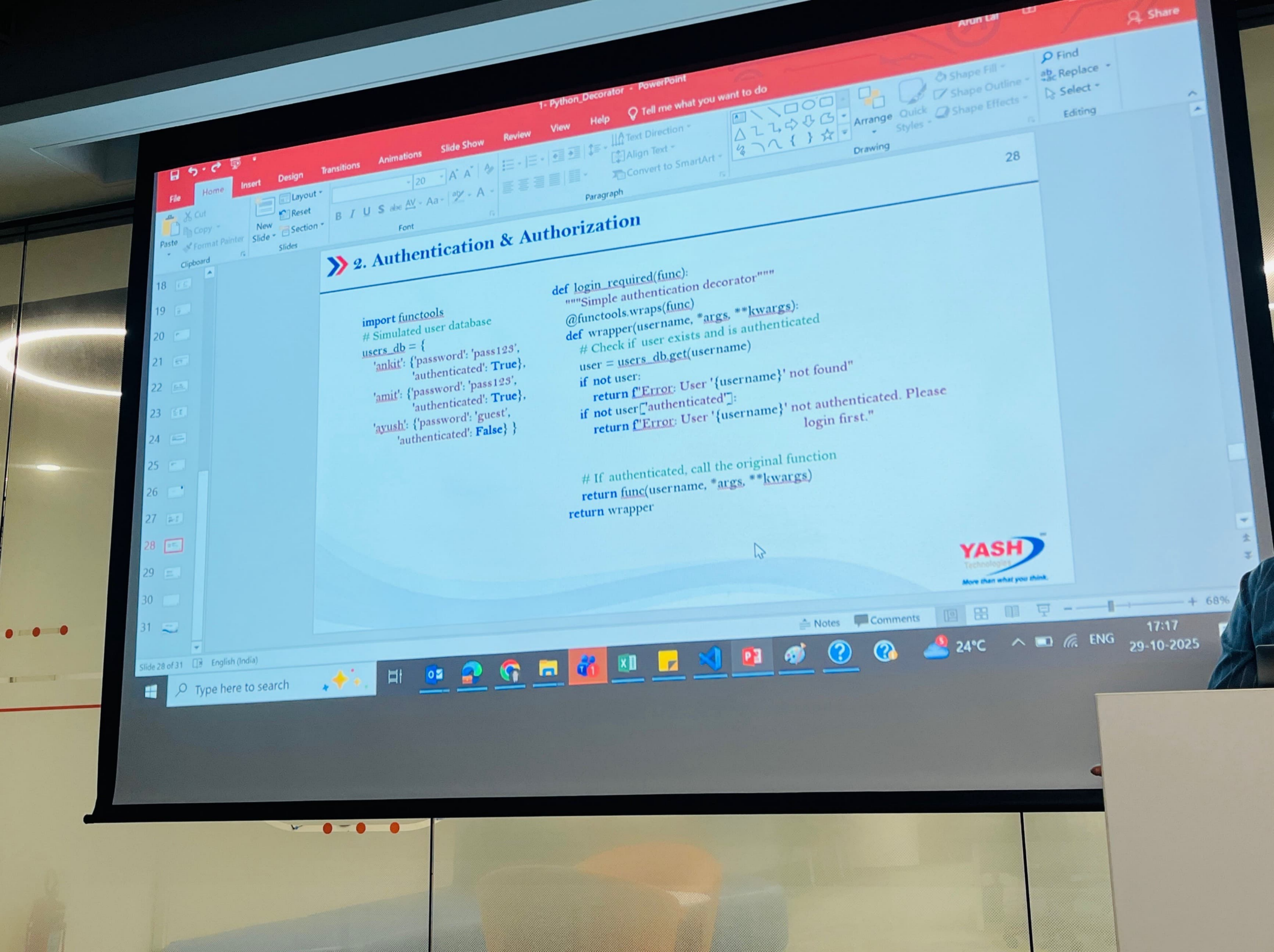
Notes Comments 68%

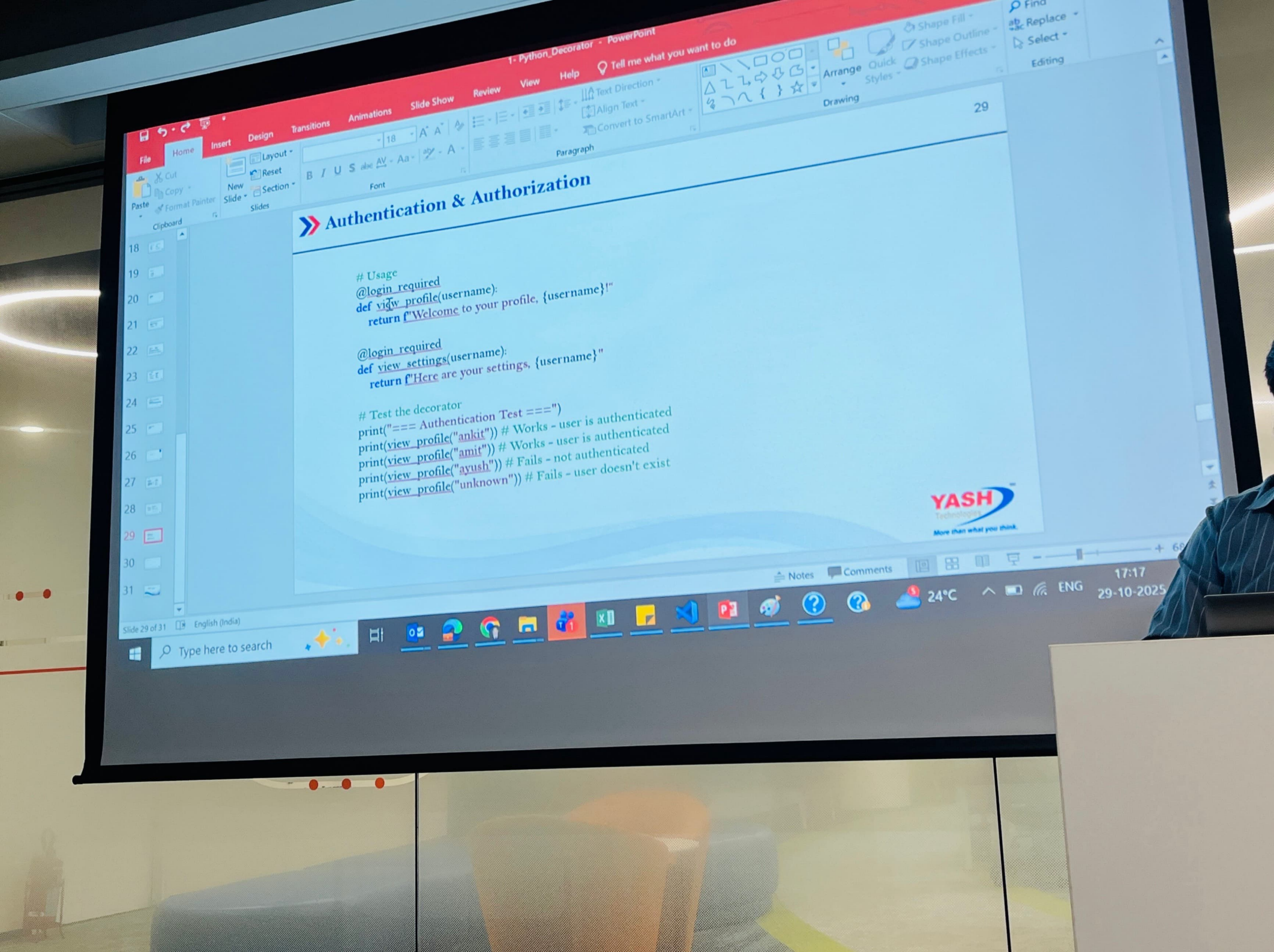
Type here to search

24°C ENG 17:16 29-10-2025


1 - Python_Decorator - PowerPoint
Tell me what you want to do
Share
Built-in Decorators
25
Python provides built-in decorators like:
•@staticmethod
•@classmethod
•@property
YASH
Technologies
More than what you think.
Slide 25 of 31
English (India)
Notes
Comments
HP Support Assistant
Recommended action needed.
Type here to search
ENG
17:16
29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
27
1: Logging Decorator
You want to log every function call in your system. Instead of writing print() inside every function, you can just use a decorator!
from functools import wraps
def log_function_call(func):
@wraps(func)
def wrapper(*args, **kwargs):
print(f"Function '{func.__name__}' called with arguments {args}, {kwargs}")
result = func(*args, **kwargs)
print(f"Function '{func.__name__}' finished")
return result
return wrapper
@log_function_call
def greet(name):
print(f"Hello, {name}!")
@log_function_call
def add(a, b):
return a + b
greet("Ankt")
print(add(5, 3))
YASH Technologies
More than what you think.
Notes
Comments
Slide 27 of 31
English (India)
Type here to search
24°C
ENG
17:17
29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
Share
28
2. Authentication & Authorization
import functools
# Simulated user database
users_db = {
'ankit': {'password': 'pass123',
'authenticated': True},
'amit': {'password': 'pass123',
'authenticated': True},
'ayush': {'password': 'guest',
'authenticated': False} }
def login_required(func):
"""Simple authentication decorator"""
@functools.wraps(func)
def wrapper(username, *args, **kwargs):
# Check if user exists and is authenticated
user = users_db.get(username)
if not user:
return f"Error: User '{username}' not found"
if not user["authenticated"]:
return f"Error: User '{username}' not authenticated. Please
login first."
If authenticated, call the original function
return func(username, *args, **kwargs)
return wrapper
YASH
Technologies
Move than what you think
Notes
Comments
68%
Slide 28 of 31
English (India)
Type here to search
24°C
ENG
17:17
29-10-2025


1- Python_Decorator - PowerPoint
Tell me what you want to do
29
Authentication & Authorization
Usage
@login_required
def view_profile(username):
return f"Welcome to your profile, {username}!"
@login_required
def view_settings(username):
return f"Here are your settings, {username}"
# Test the decorator
print("=== Authentication Test ===")
print(view_profile("ankit")) # Works - user is authenticated
print(view_profile("amit")) # Works - user is authenticated
print(view_profile("ayush")) # Fails - not authenticated
print(view_profile("unknown")) # Fails - user doesn't exist
YASH
Technologies
Notes
Comments
24°C
ENG
17:17
29-10-2025
Type here to search

Logging

```
@log_calls
def add(x, y):
    return x + y
add(2, 3)
```

output:

```
INFO:root:Calling add with args (2, 3) and kwargs {}
INFO:root:Function add returned 5
```

YASH Technologies
More than what you think.

Caching

Decorators can be used to cache the results of expensive function calls, which can improve performance by avoiding redundant computations. Here's an example:

```
def cache(func):
    cached_results = {}
    def wrapper(*args):
        if args in cached_results:
            return cached_results[args]
        result = func(*args)
        cached_results[args] = result
        return result
    return wrapper
```

```
@cache
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
fibonacci(10)
```

This will return the result of fibonacci(10) (55) and cache the results of previous calls to fibonacci.

YASH Technologies
More than what you think.

Caching

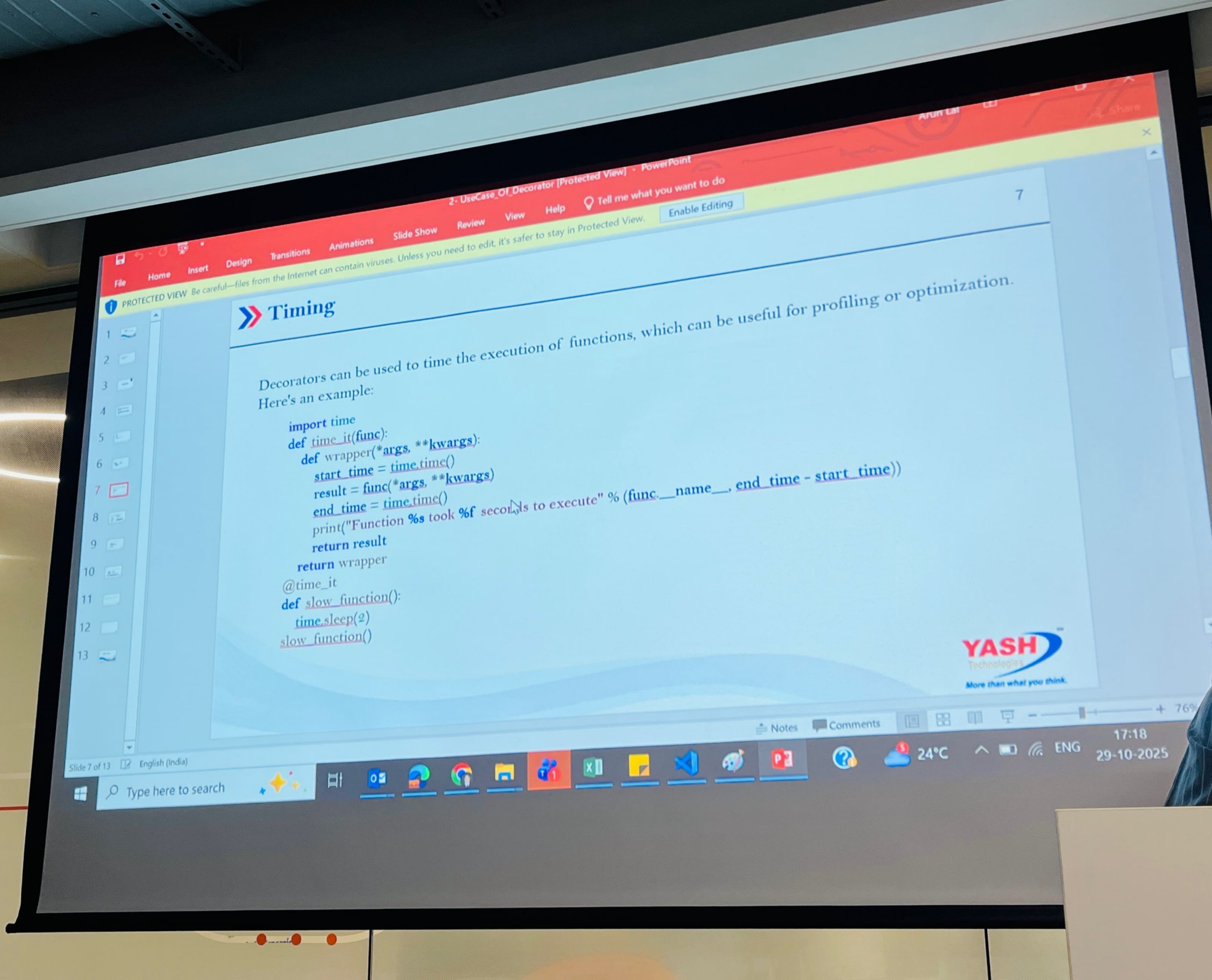
Decorators can be used to cache the results of expensive function calls, which can improve performance by avoiding redundant computations. Here's an example:

```
def cache(func):
    cached_results = {}
    def wrapper(*args):
        if args in cached_results:
            return cached_results[args]
        result = func(*args)
        cached_results[args] = result
        return result
    return wrapper
```

```
@cache
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
fibonacci(10)
```

This will return the result of fibonacci(10) (55) and cache the results of previous calls to fibonacci.

YASH Technologies
More than what you think.


2- UseCase_Of_Decorator [Protected View] - PowerPoint
PROTECTED VIEW Be careful—files from the Internet can contain viruses. Unless you need to edit, it's safer to stay in Protected View. Enable Editing
7
Timing
Decorators can be used to time the execution of functions, which can be useful for profiling or optimization.
Here's an example:
import time
def time_it(func):
def wrapper(*args, **kwargs):
start_time = time.time()
result = func(*args, **kwargs)
end_time = time.time()
print("Function %s took %f seconds to execute" % (func.__name__, end_time - start_time))
return result
return wrapper
@time_it
def slow_function():
time.sleep(2)
slow_function()
YASH Technologies
More than what you think.
Notes
Comments
Slide 7 of 13
English (India)
Type here to search
24°C
ENG
17:18
29-10-2025

Method Chaining

Decorators can be used to implement method chaining in Python. Method chaining is a programming pattern where multiple methods are called on an object in a single statement, with each method returning the object itself. This can make code more concise and readable.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def set_name(self, name):
        self.name = name
        return self

    def set_age(self, age):
        self.age = age
        return self

    def __str__(self):
        return f"{self.name}, {self.age}"

person = Person("Alice", 30)
person.set_name("Bob").set_age(40)
print(person) # Output: Bob, 40
```

In this example, the set_name and set_age methods return the Person object itself, which allows them to be chained together in a single statement.

YASH Technologies
More than what you think.

Authentication

Decorators can be used to implement authentication or authorization checks in Python. This can be useful for restricting access to certain parts of an application or API.

```
def requires_authentication(func):
    def wrapper(*args, **kwargs):
        # Check if the user is authenticated
        if not is_authenticated():
            raise Exception("User not authenticated")
        return func(*args, **kwargs)
    return wrapper

@requires_authentication
def delete_user(user_id):
    # Delete the user from the database
    pass
```

YASH Technologies
More than what you think.

2- UseCase_Of_Decorator [Protected View] - PowerPoint

File Home Insert Design Transitions Animations Slide Show Review View Help Tell me what you want to do

PROTECTED VIEW Be careful—files from the Internet can contain viruses. Unless you need to edit, it's safer to stay in Protected View. Enable Editing

11

Conclusion

Decorators are a powerful tool in Python that can be used for a variety of purposes, including method chaining, authentication, and memorization. By using decorators, you can add functionality to functions in a flexible and composable way, which can make your code more modular and easier to maintain.

YASH Technologies
More than what you think.

Notes Comments

Slide 11 of 13 English (India)

Type here to search

24°C ENG 17:18 29-10-2025