Decorators (10 Questions)

1. Write a decorator to print 'Function started' before a function runs and 'Function ended' after it runs.

```
def startendfunction(func):
    def wrapper(*args, **kwargs):
        print("Function started")
        result = func(*args, **kwargs)
        print("Function ended")
        return result
    return wrapper
@startendfunction
def welcome():
    print("Hello!")
welcome()
```

2. Create a decorator that multiplies the return value of a function by 2.

```
def multiply_func(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) * 2
    return wrapper
@multiply_func
def get_number():
    return 5
print(get_number())
```

3. Write a decorator that logs the name of the function being called.

```
def log_function_name(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
@log_function_name
def welcome():
    print("Welcome!")
welcome()
```

4. Create a decorator to check if the function is called with exactly 2 arguments.

```
def check_args(func):
    def wrapper(*args):
        if len(args) != 2:
            print("Function needs exactly 2 arguments")
        else:
            return func(*args)
    return wrapper
```

5. Write a decorator that counts and prints how many times the function has been called.

```
def call_counter(func):
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        print(f"Function has been called {count} times")
        return func(*args, **kwargs)
    return wrapper
@call_counter
def welcome():
    print("Hi!")
welcome()
welcome()
```

6. Write a decorator that restricts a function from running more than once.

```
def run_once(func):
    has_run = False
    def wrapper(*args, **kwargs):
        nonlocal has_run
        if not has_run:
            has_run = True
            return func(*args, **kwargs)
        print("Function already executed once.")
    return wrapper
@run_once
def welcome():
    print("Hello!")
welcome()
welcome()
```

7. Write a decorator to check if a user is authenticated (pass is_authenticated=True as a keyword argument).

```
def authenticated(func):
    def wrapper(*args, **kwargs):
        if kwargs.get("is_authenticated"):
            return func(*args, **kwargs)
        else:
            print("User not authenticated.")
    return wrapper
@authenticated
def user(**kwargs):
    print("Welcome user!")
user(is_authenticated=True)
```

8. Create a decorator with arguments that repeats the function n times.

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
@repeat(3)
def say_hello():
    print("Hello!")
say_hello()
```

9. Write a decorator that measures the execution time of a function.

```
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Execution time: {end - start:.4f} seconds")
        return result
    return wrapper
@timer
def task():
    time.sleep(1)
    print("Task completed")
task()
```

10. Write a decorator that modifies a function to return None if it raises any exception.

```
def remove_errors(func):
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            print(f"Error: {e}")
            return None
    return wrapper
@remove_errors
def divide(a, b):
    return a / b
print(divide(10, 2))
print(divide(10, 0))
```

Logging (10 Questions)

11. Write a simple logging function that logs to the console when a function is called.

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"[LOG] {func.__name__} called.")
        return func(*args, **kwargs)
    return wrapper
@log_call
def welcome():
    print("Hello!")
welcome()
```

12. Create a logger using the logging module that logs messages to a file.

```
import logging
logging.basicConfig(filename='app.log', level=logging.INFO)
logging.info("This is an info message")
```

13. Write a function that logs both arguments and return values.

```
def log_args(func):
    def wrapper(*args, **kwargs):
        print(f"Function called with arguments: {args} and keyword arguments: {kwargs}")
        result = func(*args, **kwargs)
        print(f"Function returned: {result}")
        return result
    return wrapper
@log_args
def add(a, b):
    return a + b
add(3, 4)
```

14. Add logging to a calculator function that logs each operation.

```
def calculator(a, b, op):
    if op == '+':
        result = a + b
    elif op == '-':
        result = a - b
    elif op == '*':
        result = a * b
    elif op == '/':
        result = a / b
    print(f"{a} {op} {b} = {result}")
    return result
calculator(5, 2, '+')
```

15. Configure a logger to log only warnings and errors.

```
import logging
logging.basicConfig(level=logging.WARNING)
logging.debug("This is a debug")
logging.info("This is info")
logging.warning("This is warning")
logging.error("This is error")
```

16. Write a decorator that logs the execution time of a function.

```
import logging
import time
logging.basicConfig(filename='timing.log',level = logging.INFO)
def log_execution_time(func):
    def wrapper(*args,**kwargs):
        start = time.time()
        result =func(*args,**kwargs)
        end = time.time()
        logging.info(f"{func.__name__} took {end-start:.4f} seconds")
        return result
    return wrapper
@log_execution_time
def sayhello():
    print("hello")
    print("a+b")
    print("a-b")
    print("a*b")
    print("a/b")
    print("seeing the execution time")
sayhello()
```

17. Use logging to log uncaught exceptions in a function.

```
import logging
logging.basicConfig(filename='exceptions.log',level = logging.ERROR)
def error_handled_function():
   try:
      1/0
   except Exception as e:
      logging.error("error occured: %s,e")
      raise
error_handled_function()
```

18. Write a logger that logs the user IP address when a function is called.

```
import logging
logging.basicConfig(filename='user_ip.log',level = logging.INFO)
def log_user_ip(ip):
  def decorator(func):
    def wrapper(*args,**kwargs):
      logging.info(f"function {func.__name__} called by user with IP  {ip}")
      return func(*args,**kwargs)
    return wrapper
  return decorator
@log_user_ip("192.168.0.100")
def sayhello():
  print("hello")
sayhello()
```

19. Create a rotating file logger using logging.handlers.

```
import logging
from logging.handlers import RotatingFileHandler
logger = logging.getLogger('rotating_logger')
logger.setLevel(logging.INFO)
handler = RotatingFileHandler('rotate.log',maxBytes=1000,backupCount=2)
logger.addHandler(handler)
logger.info("this ia a rotating log message")
```

20. Write a decorator that logs the start and end time of a data processing function.

```
import logging
import time
logging.basicConfig(filename='process.log',level = logging.INFO)
def log_start_end(func):
    def wrapper(*args,**kwargs):
      logging.info(f"function {func.__name__} started")
      result = func(*args,**kwargs)
    return wrapper
```

```
@log_start_end
def sayhello():
  print("processing data....")
  time.sleep(20)
  print("data processed successfully")
sayhello()
```

Authorization (10 Questions)

21. Write an authorization decorator that allows only users with the role 'admin'.

```
def admin(func):
    def wrapper(*args, **kwargs):
        if kwargs.get('role') == 'admin':
            return func(*args, **kwargs)
        else:
            print("Access denied")
    return wrapper
```

22. Create a function that checks if the user has permission 'view_reports'.

```
def permission(user):
    return 'view_reports' in user.get('permissions', [])
user1 = {'permissions': ['view_reports', 'edit_reports']}
user2 = {'permissions': ['edit_reports']}
print(permission(user1))
print(permission(user2))
```

23. Write a decorator that blocks a function call if the user’s status is 'inactive'.

```
def block_if_inactive(func):
    def wrapper(user_status,*args, **kwargs):
        if user_status.lower() == "inactive":
            print(f"access denied :user status is '{user_status}.function '{func.__name__}' will not be executed.")

        return func(user_status,*args, **kwargs)
    return wrapper
@block_if_inactive
```

```
def view_profile(user_status):
    print(f"user profile is being viewed.status: {user_status}")
view_profile('inactive')
view_profile('active')
```

24. Implement a decorator that checks if a user email is in the authorized list.

```
authorized_emails = ['user1@example.com', 'admin@example.com', 'test@example.com']
def email_list(func):
    def wrapper(*args, **kwargs):
        email = kwargs.get('email')
        if email in authorized_emails:
            print(f"Access granted to: {email}")
            return func(*args, **kwargs)
        else:
            print(f"Access denied for: {email}")
    return wrapper
@email_list
def view_dashboard(**kwargs):
    print("Welcome to the dashboard!")
view_dashboard(email='admin@example.com')
```

25. Write a decorator that checks if the user token is valid.

```
valid_token = "abc123token"
def check_token(func):
    def wrapper(*args, **kwargs):
        token = kwargs.get('token')
        if token == valid_token:
            print("Valid token")
            return func(*args, **kwargs)
        else:
            print("Invalid token. Access denied.")
    return wrapper
@check_token
```

```
def access_data(**kwargs):
    print("Data Fetched.")
access_data(token='abc123token')
```

26. Write a decorator that allows only users with subscription 'premium' to access a function.

```
def premium(func):
    def wrapper(user):
        if user.get('subscription') == 'premium':
            return func(user)
        else:
            print("Upgrade to premium to access this feature.")
    return wrapper
@premium
def watch_hd_movies(user):
    print("Enjoy your movie!")
watch_hd_movies({'subscription': 'premium'})
```

27. Simulate an API call where only authenticated users can access data using a decorator.

```
def auth(func):
    def wrapper(*args, **kwargs):
        if kwargs.get("authenticated", False):
            return func(*args, **kwargs)
        else:
            return "Access Denied: User not authenticated"
    return wrapper
@auth
def get_data(*args, **kwargs):
    return "Here is your protected data!"
print(get_data(authenticated=True))
```

28. Write a decorator that denies access if the user tries to access outside working hours.

```
from datetime import datetime
def work_hours(func):
    def wrapper(*args, **kwargs):
        current = datetime.now().hour
        if 9 <= current < 18:
            return func(*args, **kwargs)
        else:
            print("Access allowed only during working hours.")
    return wrapper
@work_hours
def access_tool():
    print("Tool accessed!")
access_tool()
```

29. Write a decorator that logs unauthorized access attempts.

```
def log_unauthorized_access(func):
    def wrapper(*args, **kwargs):
        if kwargs.get('allowed', False):
            return func(*args, **kwargs)
        else:
            print("Unauthorized access attempt logged.")
            return "Access Denied"
    return wrapper
@log_unauthorized_access
def view_report(*args, **kwargs):
    return "Data"
print(view_report(allowed=True))
print(view_report(allowed=False))
```

30. Create a decorator that restricts access to functions based on country code.

```
def allow_country(allowed_countries):
    def decorator(func):
        def wrapper(*args, **kwargs):
            country = kwargs.get('country')
            if country in allowed_countries:
                return func(*args, **kwargs)
        else:
            print(f"Access denied for country: {country}")
            return "Access Denied"
    return wrapper
  return decorator
@allow_country(['IN', 'US', 'UK'])  # Only these country codes are allowed
def access_service(*args, **kwargs):
  return "You have access to the service!"
print(access_service(country='IN'))
print(access_service(country='CA'))
```

Lambda Functions (10 Questions)

31. Write a lambda function to square a number.

```
square = lambda x: x**2
print(square(5))
```

32. Use filter() with lambda to get all even numbers from a list.

```
nums = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)
```

33. Use map() with lambda to get the cube of all numbers in a list.

```
nums = [1, 2, 3]
cubes = list(map(lambda x: x**3, nums))
print(cubes)
```

34. Write a lambda function to check if a string is a palindrome.

```
is_palindrome = lambda s: s == s[::-1]
print(is_palindrome("madam"))
print(is_palindrome("hello"))
```

35. Sort a list of tuples based on the second element using lambda.

```
pairs = [(1, 3), (2, 1), (4, 2)]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)
```

36. Use reduce() with lambda to calculate the factorial of a number.

```
from functools import reduce
factorial = lambda n: reduce(lambda x, y: x*y, range(1, n+1))
print(factorial(5))
```

37. Write a lambda function to check if a number is divisible by both 3 and 5.

```
div_3_5 = lambda x: x % 3 == 0 and x % 5 == 0
print(div_3_5(15))
print(div_3_5(10))
```

38. Use map() and lambda to convert a list of strings to uppercase.

```
words = ['apple', 'banana']
uppercased = list(map(lambda x: x.upper(), words))
print(uppercased)
```

39. Use lambda inside sorted() to sort a list of dictionaries by the 'age' key.

```
people = [{'name': 'A', 'age': 30}, {'name': 'B', 'age': 25}]
sorted_people = sorted(people, key=lambda x: x['age'])
print(sorted_people)
```

40. Write a lambda function that returns the maximum of two numbers.

```
max_two = lambda a, b: a if a > b else b
print(max_two(4, 7))
```