# Practice Questions on Python Decorators, Logging, Authorization, and Lambda Functions (Easy to Intermediate Level)

## Decorators (10 Questions)

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

def log\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print("Function started")

result = func(\*args, \*\*kwargs)

print("Function ended")

return result

return wrapper

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

def multiple(no):

def wrapper():

print("the multiple of 2")

res = no()

return res\*2

return wrapper

@multiple

def performa():

return 5

print(performa())

1. 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 greet():

print("Hello!")

greet()

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

def check\_two\_args(func):

def wrapper(\*args, \*\*kwargs):

if len(args) + len(kwargs) != 2:

print("Function must be called with exactly 2 arguments.")

return func(\*args, \*\*kwargs)

return wrapper

@check\_two\_args

def add(a, b):

return a + b

print(add(3, 5)) # Valid

print(add(3))# error

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

def count\_calls(func):

count = 0

def wrapper(\*args, \*\*kwargs):

count += 1

print(f"{func.\_\_name\_\_} has been called {count} times.")

return func(\*args, \*\*kwargs)

return wrapper

@count\_calls

def say\_hello():

print("Hello!")

say\_hello()

say\_hello()

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

def run\_once(func):

has\_run = False

def wrapper(\*args, \*\*kwargs):

if not has\_run:

has\_run = True

return func(\*args, \*\*kwargs)

else:

print("Function has already been run once.")

return wrapper

@run\_once

def initialize():

print("Initialization done.")

initialize()

initialize()

1. Write a decorator to check if a user is authenticated (pass is\_authenticated=True as a keyword argument).

def require\_authentication(func):

def wrapper(\*args, \*\*kwargs):

if not kwargs.get('is\_authenticated', False):

print("User not authenticated.")

return

return func(\*args, \*\*kwargs)

return wrapper

@require\_authentication

def access\_dashboard(user):

print(f"{user}'s dashboard accessed.")

access\_dashboard("Alice", is\_authenticated=True)

access\_dashboard("Bob", is\_authenticated=False)

1. 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\_hi():

print("Hi!")

say\_hi()

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

import time

def measure\_time(func):

def wrapper(\*args, \*\*kwargs):

start = time.time()

result = func(\*args, \*\*kwargs)

end = time.time()

print(f"{func.\_\_name\_\_} executed in {end - start:.4f} seconds")

return result

return wrapper

@measure\_time

def slow\_function():

time.sleep(1)

print("Done!")

slow\_function()

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

def suppress\_errors(func):

def wrapper(\*args, \*\*kwargs):

try:

return func(\*args, \*\*kwargs)

except Exception as e:

print(f"Exception caught: {e}")

return None

return wrapper

@suppress\_errors

def divide(a, b):

return a / b

print(divide(10, 2)) # Output: 5.0

print(divide(10, 0)) # Output: None (division by zero)

## Logging (10 Questions)

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

def simple\_logger(func):

def wrapper(\*args, \*\*kwargs):

print(f"{func.\_\_name\_\_} was called")

return func(\*args, \*\*kwargs)

return wrapper

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

import logging

logging.basicConfig(filename='app.log', level=logging.INFO)

def log\_to\_file():

logging.info("This is a log message to file.")

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

def log\_args\_and\_return(func):

def wrapper(\*args, \*\*kwargs):

print(f"Arguments: {args}, {kwargs}")

result = func(\*args, \*\*kwargs)

print(f"Return: {result}")

return result

return wrapper

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

import logging

logging.basicConfig(level=logging.INFO)

def add(a, b):

result = a + b

logging.info(f"Adding {a} + {b} = {result}")

return result

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

import logging

logging.basicConfig(filename='app.log',level = logging.INFO)

logging.info("this ia a log message")

logging.basicConfig(filename='warnings.log',level = logging.WARNING)

logging.warning("this is a warning message")

logging.error("this is an error")

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

import logging

import time

def log\_exec\_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

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

import logging

logging.basicConfig(level=logging.ERROR)

def log\_exceptions(func):

def wrapper(\*args, \*\*kwargs):

try:

return func(\*args, \*\*kwargs)

except Exception as e:

logging.error(f"Exception in {func.\_\_name\_\_}: {e}")

raise

return wrapper

1. 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()

1. 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")

1. 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)

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

def require\_admin(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('role') == 'admin':

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Only admins are allowed.")

return wrapper

1. Create a function that checks if the user has permission 'view\_reports'.

def has\_permission(user\_permissions):

return 'view\_reports' in user\_permissions

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

def block\_inactive\_users(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('status') == 'inactive':

raise PermissionError("Inactive users cannot perform this action.")

return func(\*args, \*\*kwargs)

return wrapper

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

AUTHORIZED\_EMAILS = ['admin@example.com', 'user@example.com']

def check\_email(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('email') in AUTHORIZED\_EMAILS:

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Email not authorized.")

return wrapper

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

def check\_token(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('token') == 'valid\_token':

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Invalid token.")

return wrapper

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

def require\_premium(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('subscription') == 'premium':

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Premium subscription required.")

return wrapper

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

def require\_auth(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('authenticated'):

return func(\*args, \*\*kwargs)

else:

return {"error": "User not authenticated"}

return wrapper

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

import datetime

def working\_hours\_only(func):

def wrapper(\*args, \*\*kwargs):

current\_hour = datetime.datetime.now().hour

if 9 <= current\_hour <= 17:

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Access only allowed during working hours (9AM–5PM).")

return wrapper

1. Write a decorator that logs unauthorized access attempts.

def log\_unauthorized\_access(func):

def wrapper(\*args, \*\*kwargs):

if not kwargs.get('authorized', False):

print(f"Unauthorized access attempt to {func.\_\_name\_\_}")

raise PermissionError("Unauthorized access.")

return func(\*args, \*\*kwargs)

return wrapper

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

ALLOWED\_COUNTRIES = ['US', 'IN', 'UK']

def restrict\_by\_country(func):

def wrapper(\*args, \*\*kwargs):

if kwargs.get('country') in ALLOWED\_COUNTRIES:

return func(\*args, \*\*kwargs)

else:

raise PermissionError("Access restricted in your region.")

return wrapper

## Lambda Functions (10 Questions)

1. Write a lambda function to square a number.

square = lambda x: x \*\* 2

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

nums = [1, 2, 3, 4, 5, 6]

even\_nums = list(filter(lambda x: x % 2 == 0, nums))

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

nums = [1, 2, 3, 4]

cubed = list(map(lambda x: x \*\* 3, nums))

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

is\_palindrome = lambda s: s == s[::-1]

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

data = [(1, 3), (2, 1), (4, 2)]

sorted\_data = sorted(data, key=lambda x: x[1])

1. 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))

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

divisible = lambda x: x % 3 == 0 and x % 5 == 0

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

words = ['hello', 'world']

upper\_words = list(map(lambda s: s.upper(), words))

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

people = [{'name': 'Alice', 'age': 25}, {'name': 'Bob', 'age': 20}]

sorted\_people = sorted(people, key=lambda x: x['age'])

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

maximum = lambda a, b: a if a > b else b