# Practice Questions

## Decorators (10 Questions)

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

def decorator(func):

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

print("Function started")

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

print("Function ended")

Return result

return wrapper

@decorator

def welcome():

print("hi, abhi")

welcome()

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

def multiply(func):

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

return func(\*args, \*\*kwargs) \*2

return wrapper

@multiply

def num():

return 10

print(num())

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

def log\_name(func):

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

print(f"call function: {func.\_\_name\_\_}")

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

return wrapper

@log\_name

def welcome():

print("hi, abhi")

welcome()

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

def check(func):

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

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

print("The function has 2 arguments")

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

else:

return "function needs to have 2 arguments"

return wrapper

@check

def add(a,b):

return a+b

print(add(5,5))

print(add(20))

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):

nonlocal count

count += 1

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

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

return wrapper

@count\_calls

def message():

print("Hello Abhigna!")

message()

message()

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

def run\_once(func):

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

if not wrapper.is\_run:

wrapper.is\_run = True

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

else:

print("Function can only run once.")

wrapper.is\_run = False

return wrapper

@run\_once

def welcome():

print("Welcome!")

welcome()

welcome()

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

def check\_auth(func):

def wrapper(is\_authenticated):

if is\_authenticated:

return func()

else:

return "Access Denied"

return wrapper

@check\_auth

def show\_data():

return "Here is your data"

print(show\_data(True))

print(show\_data(False))

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

def repeat(n):

def decorator(func):

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

for x in range(n):

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

return wrapper

return decorator

@repeat(5)

def message():

print("HELLO")

message()

1. 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"Executed in {duration is:.5f} seconds")

return result

return wrapper

@timer

def sleep\_func():

time.sleep(2)

sleep\_func()

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

def modify(func):

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

try:

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

except Exception as e:

print(f"Error,{e}")

return None

return wrapper

@modify

def divide(a,b):

return a/b

print(divide(2,3))

print(divide(20,0))

## Logging (10 Questions)

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

def logger(func):

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

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

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

return wrapper

@logger

def welcome():

print("Hello!,Abhi")

welcome()

1. 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 log message")

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

def log(func):

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

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

print(f"{func.\_\_name\_\_} called with {args}, gives {result}")

return result

return wrapper

@log

def multiply(a, b):

return a \* b

multiply(5, 5)

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

def calc\_log(func):

def wrapper(a, b):

result = func(a, b)

print(f"{func.\_\_name\_\_}({a}, {b}) = {result}")

return result

return wrapper

@calc\_log

def add(a, b): return a + b

@calc\_log

def sub(a, b): return a - b

@calc\_log

def mul(a, b): return a \* b

@calc\_log

def div(a, b): return a / b

add(5, 2)

sub(10, 3)

mul(12,4)

div(10,5)

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

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

logging.warning("this is warning message")

logging.error("this is an error")

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

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()

1. 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()

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 authorize(user\_role):

def decorator(func):

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

if user\_role == "admin":

print(f"access granted to {user\_role}")

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

else:

print(f"denied to {user\_role}")

return "unauthorized access"

return wrapper

return decorator

@authorize("admin")

def view\_dashboard():

return "dashboard data"

@authorize("guest")

def view\_settings():

return "settings data"

print((view\_dashboard()))

print(view\_settings())

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

def view\_reports(user):

return 'view\_reports' in user.get('permissions', [])

user1 = {

"username": "admin",

"permissions": ["view\_reports", "settings"]

}

user2 = {

"username": "guest",

"permissions": ["read\_reports"]

}

print(view\_reports(user1))

print(view\_reports(user2))

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

def block\_inactive(func):

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

if user.get("status") == "inactive":

print(f"Access denied for {user['username']}: user inactive.")

return

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

return wrapper

@block\_inactive

def access\_feature(user):

print(f"{user['username']} accessed the feature.")

user1 = {"username": "Abhi", "status": "active"}

user2 = {"username": "Esha", "status": "inactive"}

access\_feature(user1)

access\_feature(user2)

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

ef authorize(email\_allowed):

def decorator(func):

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

if user\_email == email\_allowed:

print(f"access granted to {user\_email}")

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

else:

print(f"access denied to {user\_email}")

# func()

return "unauthorized Access"

return wrapper

return decorator

@authorize("admin@cgi.com")

def view\_dashboard(user\_email):

return "this is the dashboard"

@authorize("guest")

def view\_settings(user\_email):

return " can change/access settings"

print(view\_dashboard("admin@stud.com"))

print(view\_settings("guest@stud.com"))

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

def token\_check(valid\_tokens):

def decorator(func):

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

if user.get("token") not in valid\_tokens:

print(f"Access denied: Invalid token")

return

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

return wrapper

return decorator

@token\_check(["abcd123", "token456"])

def access(user):

print(f"{user['username']} accessed secured data.")

user1 = {"username": "Abhi", "token": "abcd123"}

user2 = {"username": "Harini", "token": "azby789"}

access(user1)

access(user2)

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

def premium(func):

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

if user.get("subscription") != "premium":

print(f"Access denied for {user.get('username', ' ')}, Premium required.")

return

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

return wrapper

@premium

def access(user):

print(f"{user['username']} is allowed premium access.")

user1 = {"username": "vibha", "subscription": "premium"}

user2 = {"username": "bhargav", "subscription": "free"}

access(user1)

access(user2)

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

def authenticated(func):

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

if not user.get("is\_authenticated",False):

print(f"{user.get('username', 'Unknown')} is not authenticated.")

return

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

return wrapper

@authenticated

def data(user):

print(f"200 OK:{user['username']},it is your authenticated data.")

user1 = {"username": "abhi", "is\_authenticated": True}

user2 = {"username": "rahul", "is\_authenticated": False}

data(user1)

data(user2)

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

import datetime

def working\_hours(start\_hour=9,end\_hour=15):

def decorator(func):

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

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

if current\_hour < 9 or current\_hour > 15:

print(f"Access denied to ({start\_hour}- {end\_hour}),outside working hours.")

return

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

return wrapper

return decorator

@working\_hours(start\_hour=9,end\_hour=15)

def access(user):

print(f"{user['username']} accessed during working hours.")

user = {"username": "rahul"}

access(user)

1. Write a decorator that logs unauthorized access attempts.

def unauthorized\_access(func):

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

if not user.get("authorized"):

print(f"LOG Unauthorized access attempted}")

return "Access denied"

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

return wrapper

@unauthorized\_access

def authorized(user):

print(f"{user['username']} ,This is your confidential data.")

user1 = {"username": "abhi", "authorized": True}

user2 = {"username": "bhumi", "authorized": False}

authorized(user1)

authorized(user2)

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

def country\_allowed(allowed\_countries):

def decorator(func):

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

if user.get("country") not in allowed\_countries:

print(f"Access denied for {user.get('username')} from {user.get('country')}.")

return

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

return wrapper

return decorator

@country\_allowed(allowed\_countries=["IN", "US"])

def access(user):

print(f"Access allowed for {user['username']} from {user['country']}")

user1 = {"username": "abhi", "country": "IN"}

user2 = {"username": "esha", "country": "CN"}

access(user1)

access(user2)

## Lambda Functions (10 Questions)

1. Write a lambda function to square a number.

square= lambda x:x\*\*2

print(square(7))

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

num=[1,2,3,4,5]

even=list(filter(lambda x: x%2==0,num))

print(even)

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

num=[1,2,3,4,5]

cube=list(map(lambda x: x\*\*3,num))

print(cube)

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

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

print(is\_palindrome("madam"))

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

students=[('abhi',85),('bhumi',90),("esha",75)]

sorted\_students=sorted(students,key=lambda x:x[1])

print(sorted\_students)

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

from functools import reduce

n=7

factorial= reduce(lambda a,b:a\*b,range(1,n+1)

print(factorial)

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

is\_divisible=lambda x:x%3==0 and x%5==0

print(is\_divisible(50))

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

string=[“cgi”, ”employee”]

upper\_case=list(map(lambda x:x.upper( ),string))

print(upper\_case)

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

students=[{"name":'abhi',"age":85},{"name":'bhumi',"age":90},{"name":"esha","age":75}]

sorted\_students=sorted(students,key=lambda x:x['age'])

print(sorted\_students)

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

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

print(max(20,50))