Day -5 Morning Assessment

# Decorators

1. def decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 print('function started')  
 result = func(\*args, \*\*kwargs)  
 print('function finished')  
 return result  
 return wrapper  
  
@decorator  
def greet(name):  
 print('Hello', name)  
greet('Vidya')

2. def decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 result = func(\*args, \*\*kwargs)  
 return result\*2  
 return wrapper  
  
@decorator  
def multiply( ):  
 return 9  
print(multiply())

3. def log\_function\_name(function):  
 def wrapper(\*args, \*\*kwargs):  
 print(f'{function.\_\_name\_\_}')  
 return function(\*args, \*\*kwargs)  
 return wrapper

4. def check\_two\_args(func):  
 def wrapper(\*args, \*\*kwargs):  
 if len(args)+len(kwargs) != 2:  
 print("more than 2 arguments")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
@check\_two\_args  
def add(a, b ,c):  
 return a+b  
  
print(add(2, 3,4))

o/p: more than 2 arguments

5

5. 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 add(a, b):  
 return a + b  
@count\_calls  
def subtract(a, b):  
 return a - b  
print(add(2, 3))  
print(subtract(2, 3))

o/p: add has been called 1 times

5

subtract has been called 1 times

-1

6. def run\_once(func):  
 has\_run = False  
 def wrapper(\*args, \*\*kwargs):  
 nonlocal has\_run  
 if has\_run:  
 print(f'{func.\_\_name\_\_} already run')  
 return None  
 has\_run = True  
 return func(\*args, \*\*kwargs)  
 return wrapper

7. def require\_authentication(func):  
   def wrapper(\*args, \*\*kwargs):  
       if not kwargs.get('is\_authenticated', False):  
           print("User is not authenticated.")  
           return None  
       return func(\*args, \*\*kwargs)  
   return wrapper

8. def repeat(n):  
   def decorator(func):  
       def wrapper(\*args, \*\*kwargs):  
           for \_ in range(n):  
               func(\*args, \*\*kwargs)  
       return wrapper  
   return decorator

9. import time  
  
def measure\_time(func):  
   def wrapper(\*args, \*\*kwargs):  
       start = time.time()  
       result = func(\*args, \*\*kwargs)  
       end = time.time()  
       print(f"{func.\_\_name\_\_} took {end - start:.4f} seconds.")  
       return result  
   return wrapper

10. def suppress\_exceptions(func):  
   def wrapper(\*args, \*\*kwargs):  
       try:  
           return func(\*args, \*\*kwargs)  
       except Exception as e:  
           print(f"Exception caught in {func.\_\_name\_\_}: {e}")  
           return None  
   return wrapper

# Logging

11.   
def simple\_log(func):  
   def wrapper(\*args, \*\*kwargs):  
       print(f"Function {func.\_\_name\_\_} was called")  
       return func(\*args, \*\*kwargs)  
   return wrapper

12.   
import logging  
# Configure logger  
logging.basicConfig(filename='app.log', level=logging.INFO, format='%(asctime)s - %(message)s')  
  
def file\_logger\_func():  
   logging.info("This is a log message to a file.")

13. Function that logs both arguments and return values  
def log\_args\_and\_return(func):  
   def wrapper(\*args, \*\*kwargs):  
       logging.info(f"Calling {func.\_\_name\_\_} with args={args}, kwargs={kwargs}")  
       result = func(\*args, \*\*kwargs)  
       logging.info(f"{func.\_\_name\_\_} returned {result}")  
       return result  
   return wrapper

14. def calculator(op, a, b):  
   if op == 'add':  
       result = a + b  
   elif op == 'subtract':  
       result = a - b  
   elif op == 'multiply':  
       result = a \* b  
   elif op == 'divide':  
       result = a / b  
   else:  
       raise ValueError("Invalid operation")  
  
   logging.info(f"{op}({a}, {b}) = {result}")  
   return result

15. Configure logger to log only warnings and errors  
logging.basicConfig(level=logging.WARNING, format='%(levelname)s: %(message)s')  
  
# Example:  
logging.debug("This won't show")  
logging.warning("This is a warning")  
logging.error("This is an error")

16.import time  
def log\_execution\_time(func):  
   def wrapper(\*args, \*\*kwargs):  
       start = time.time()  
       result = func(\*args, \*\*kwargs)  
       duration = time.time() - start  
       logging.info(f"{func.\_\_name\_\_} took {duration:.4f} seconds")  
       return result  
   return wrapper

17. def log\_exceptions(func):  
   def wrapper(\*args, \*\*kwargs):  
       try:  
           return func(\*args, \*\*kwargs)  
       except Exception as e:  
           logging.exception(f"Exception occurred in {func.\_\_name\_\_}")  
           raise

   return wrapper

18. def log\_ip(func):  
   def wrapper(\*args, \*\*kwargs):  
       ip = kwargs.get('ip', 'unknown')  
       logging.info(f"{func.\_\_name\_\_} called from IP: {ip}")  
       return func(\*args, \*\*kwargs)  
   return wrapper

19. from logging.handlers import RotatingFileHandler  
logger = logging.getLogger("RotatingLogger")  
logger.setLevel(logging.INFO)  
  
handler = RotatingFileHandler("rotating.log", maxBytes=2000, backupCount=3)  
formatter = logging.Formatter('%(asctime)s - %(message)s')  
handler.setFormatter(formatter)  
  
logger.addHandler(handler)  
logger.info("Rotating log message")

20. def log\_processing\_time(func):  
   def wrapper(\*args, \*\*kwargs):  
       start\_time = time.strftime('%Y-%m-%d %H:%M:%S')  
       logging.info(f"{func.\_\_name\_\_} started at {start\_time}")  
  
       result = func(\*args, \*\*kwargs)  
  
       end\_time = time.strftime('%Y-%m-%d %H:%M:%S')  
       logging.info(f"{func.\_\_name\_\_} ended at {end\_time}")  
       return result  
   return wrapper

# Authorization

21. def require\_admin(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('role') != 'admin':  
           return "Access denied: Admins only."  
       return func(user, \*args, \*\*kwargs)  
   return wrapper

22. def has\_view\_reports\_permission(user):  
   return 'view\_reports' in user.get('permissions', [])

23. def block\_if\_inactive(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('status') == 'inactive':  
           print("Access blocked: User is inactive.")  
           return  
       return func(user, \*args, \*\*kwargs)  
   return wrapper  
24.   
AUTHORIZED\_EMAILS = {'admin@example.com', 'user@domain.com'}  
  
def email\_authorized(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('email') in AUTHORIZED\_EMAILS:  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: Unauthorized email.")  
   return wrapper  
25.   
def valid\_token\_required(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       token = user.get('token')  
       if token and token.startswith("valid-"):  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: Invalid token.")  
   return wrapper  
26def premium\_required(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('subscription') == 'premium':  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: Premium members only.")  
   return wrapper  
27. def authenticated\_only(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('is\_authenticated'):  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: User not authenticated.")  
   return wrapper

28.   
import datetime  
  
def within\_working\_hours(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       current\_hour = datetime.datetime.now().hour  
       if 9 <= current\_hour < 17:  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: Outside working hours.")  
   return wrapper  
29.   
import logging  
  
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')  
  
def log\_unauthorized\_access(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if not user.get('is\_authenticated'):  
           logging.warning(f"Unauthorized access attempt by user: {user}")  
           print("Access denied: Not authenticated.")  
           return  
       return func(user, \*args, \*\*kwargs)  
   return wrapper

30.   
ALLOWED\_COUNTRIES = {'IN', 'US', 'UK'}  
  
def country\_restricted(func):  
   def wrapper(user, \*args, \*\*kwargs):  
       if user.get('country') in ALLOWED\_COUNTRIES:  
           return func(user, \*args, \*\*kwargs)  
       else:  
           print("Access denied: Country not allowed.")  
   return wrapper

# LAMBDA FUNCTIONS

31. square= lambda num:num\*num  
print(square(10))

32. list1=[7,9,10,4,3,2]  
even\_numbers=list(filter(lambda x:x%2==0,list1))  
print(even\_numbers)

o/p: [10, 4, 2]

33. list1=[7,9,10,4,3,2]  
numbers=list(map(lambda x:x\*\*3,list1))  
print(numbers)

o/p: [343, 729, 1000, 64, 27, 8]

34. list1=['hi','pop','append']  
palindrome=list(filter(lambda x:x[::-1]==x,list1))  
print(palindrome)

o/p: [‘pop’]

35. data = [(1,3),(4,1),(2,2)]  
sorted\_data = sorted(data, key=lambda x: x[1])  
print(sorted\_data)

o/p: [(4, 1), (2, 2), (1, 3)]

36. from functools import reduce  
def factorial(n):  
 return reduce(lambda x, y: x\*y, range(1, n+1))  
  
print(factorial(5))

o/p: 120

37. list1=[3,4,78,99,34,56]  
numbers=list(filter(lambda x:x%3==0 and x%5==0,list1))  
print(numbers)

o/p: [ ]

38. list1=['hi','hello','how']  
string=list(map(lambda x:x.upper(),list1))  
print(string)

o/p: ['HI', 'HELLO', 'HOW']

39.people=[{'name':'Alice','age':30},{'name':'Bob','age':40},{'name':'Charlie','age':50}]  
sorted\_people=sorted(people,key=lambda x:x['name'])  
print(sorted\_people)

o/p: [{'name': 'Alice', 'age': 30}, {'name': 'Bob', 'age': 40}, {'name': 'Charlie', 'age': 50}]

40. max\_of\_two = lambda a, b: a if a > b else b  
print(max\_of\_two(1, 2))  
print(max\_of\_two(1, 6))

o/p: 2

6