Python and SQL Questions

# Python Questions

## Question 1: Validate Email Addresses

Write a Python function named `validate\_email` that takes a string as input and returns `True` if the string is a valid email address, and `False` otherwise. An email is considered valid if it follows the pattern `username@domain.extension`, where:  
- `username` is a combination of alphanumeric characters, underscores (\_), or periods (.).  
- `domain` is a combination of alphanumeric characters.  
- `extension` is two to four alphabets long.  
  
Use regular expressions to validate the email and a try-except block to handle any potential errors during the validation process.

import re

def validate\_email(email):

pattern = r'^[\w\-.]+@[A-Za-z0-9]+\.[A-Za-z]{2,4}$'

try:

if re.match(pattern, email):

return True

else:

return False

except Exception as e:

print("An error occurred during email validation:", e)

return False

print(validate\_email("vishva@gmail.com"))

print(validate\_email("vishva@gmail.c"))

print(validate\_email("vishv@@gmail.com"))

print(validate\_email("vishva@12.com"))

print(validate\_email("vishva@12.c1"))

print(validate\_email("AvBvishvaqqqqqqwe@gmailasdffffffs.com"))

**Output:**

**True**

**False**

**False**

**True**

**False**

**True**

## Question 2: Extract Dates from Text

Write a Python function called `extract\_dates` that accepts a string containing text and returns a list of dates found in the text. The dates could be in the format `DD/MM/YYYY`, `DD-MM-YYYY`, or `DD.MM.YYYY`. Your function should use regular expressions to find these patterns and return the list of dates in the format `YYYY-MM-DD`. Use try-except blocks to gracefully handle any errors in the conversion process, especially with invalid date formats.

import re

from datetime import datetime

def extract\_dates(text):

try:

date\_formats = r'\b(?:\d{2}[\/.-]\d{2}[\/.-]\d{4})\b'

dates = re.findall(date\_formats, text)

formatted\_dates = []

for date in dates:

try:

formatted\_date = datetime.strptime(date, '%d/%m/%Y').strftime('%Y-%m-%d')

formatted\_dates.append(formatted\_date)

except ValueError:

pass

try:

formatted\_date = datetime.strptime(date, '%d-%m-%Y').strftime('%Y-%m-%d')

formatted\_dates.append(formatted\_date)

except ValueError:

pass

try:

formatted\_date = datetime.strptime(date, '%d.%m.%Y').strftime('%Y-%m-%d')

formatted\_dates.append(formatted\_date)

except ValueError:

pass

return formatted\_dates

except Exception as e:

print(f"An error occurred: {e}")

return []

# Example usage:

text = "This is a sample text with dates 20/02/2024, 27-02-2024, and 03.03.2024."

dates = extract\_dates(text)

print("Dates found:", dates)

**Output:**

Dates found: ['2024-02-20', '2024-02-27', '2024-03-03']

## Question 3: Password Strength Checker

Implement a Python function named `check\_password\_strength` that takes a password string as input and returns `Strong`, `Medium`, or `Weak` based on the following criteria:  
- A strong password must contain at least one uppercase letter, one lowercase letter, one digit, one special character (`!@#$%^&\*()-\_+=`), and be at least 8 characters long.  
- A medium password meets all the above criteria except for the presence of a special character.  
- A weak password does not meet one or more of the other criteria.  
  
Use regular expressions to evaluate the password strength. Employ a try-except block to catch and handle any unexpected errors during the process.

def check\_password\_strength(password):

pattern1 = r'^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d)(?=.\*[!@#$%^&\*()\-\_+=]).{8,}$'

pattern2 = r'^(?=.\*[A-Z])(?=.\*[a-z])(?=.\*\d).{8,}$'

try:

if re.match(pattern1, password):

return "Strong"

elif re.match(pattern2, password):

return ("Medium")

else:

return "Weak"

except Exception as e:

print("An error occurred during password validation:", e)

return "Re-enter the password"

print(check\_password\_strength("hello1234G"))

print(check\_password\_strength("Hello@"))

print(check\_password\_strength("Hello123!adas"))

**Output:**

Medium

Weak

Strong

## Question 4: Log File Error Extraction

Suppose you have a log file with mixed content, and you want to extract all error messages. Write a Python script named `extract\_errors` that reads text from a given log file path and returns a list of strings containing only the error messages. Assume error messages in the log start with the timestamp in the format `[DD/MM/YYYY HH:MM:SS] ERROR:` followed by the error description.  
  
Use regular expressions to identify and extract the error messages. Implement try-except blocks to handle scenarios like the file not existing, permission issues, or any other IO errors during the file reading process.

import re

def extract\_errors(log\_file\_path):

try:

with open(log\_file\_path, 'r') as file:

log\_content = file.read()

error\_pattern = r'\[\d{2}/\d{2}/\d{4} \d{2}:\d{2}:\d{2}\] ERROR: (.+?)\n'

errors = re.findall(error\_pattern, log\_content)

return errors

except FileNotFoundError:

print(f"Error: File '{log\_file\_path}' not found.")

return []

except PermissionError:

print(f"Error: Permission denied while accessing '{log\_file\_path}'.")

return []

except Exception as e:

print(f"An error occurred: {e}")

return []

# Example usage:

log\_file\_path = r'C:/Users/Lenovo/Downloads/sample\_log.txt'

errors = extract\_errors(log\_file\_path)

print("Errors found:", errors)

**Output:**

Errors found: ['Connection failed: timeout', 'Database query error: table not found', 'Server crashed unexpectedly']

# SQL Questions

## Question 5: Basic INNER JOIN

Write an SQL query to fetch all records that have matching values in both tables. You have two tables, `Employees` and `Departments`, where `Employees.DepartmentID` matches `Departments.DepartmentID`. Select all employees who are in departments.

SELECT Employees.\*

FROM Employees

INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;

## Question 6: LEFT JOIN with Condition

Write an SQL query to fetch all employees and their department names. Use a LEFT JOIN to include employees who do not belong to any department. Assume the tables `Employees` and `Departments` where `Employees.DepartmentID` matches `Departments.DepartmentID`. For employees without a department, display the department name as 'No Department'.

SELECT Employees.EmployeeID, Employees.EmployeeName,

COALESCE(Departments.DepartmentName, 'No Department') AS DepartmentName

FROM Employees

LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;

## Question 7: Complex JOIN with Multiple Conditions

Write an SQL query to find all employees who work in the same department as 'John Doe'. The tables `Employees` (with columns `EmployeeID`, `Name`, `DepartmentID`) and `Departments` (with columns `DepartmentID`, `DepartmentName`) are given. Use JOIN operations to solve this, ensuring you include employees from any department where at least one person shares the same `DepartmentID` as 'John Doe'.

SELECT e.EmployeeID, e.Name, e.DepartmentID

FROM Employees e

INNER JOIN Employees john ON e.DepartmentID = john.DepartmentID

WHERE john.Name = 'John Doe';