Exercises: Functions and Procedures

This document defines the **exercise assignments** for the PostgreSQL course @ Software University.

Submit your solutions in the SoftUni Judge Contest.

1. User-defined Function Full Name

Create a **PostgreSQL function** named **"fn_full_name"** that takes two arguments: **"first_name"** and **"last_name"**. The function should return the **"Full Name"** as a single string, with the first name and last name capitalized and separated by a space. If either **"first_name"** or **"last_name"** is null, the function should return null or only the non-null name capitalized.

*** Note, you have the option to utilize the **INITCAP** function that transforms a string expression into a correct capitalization format, where the **first letter of every word is capitalized** and the **rest of the letters are in lowercase**.

For this task, please only submit your **user-defined function** in the Judge system.

Example

Input	Output
fn_full_name('fred', 'sanford')	Fred Sanford
fn_full_name('', 'SIMPSONS')	Simpsons
fn_full_name('JOHN', '')	John
fn_full_name(NULL, NULL)	[null]

2. User-defined Function Future Value

As a financial analyst for a bank, your job is to develop a PostgreSQL function that computes the **future value of an investment**. Your objective is to create a function called **"fn_calculate_future_value"**, which requires three inputs: **"initial_sum"**, **"yearly_interest_rate"**, and **"number_of_years"**.

- **"initial_sum"** - represents the amount of money initially invested;
- **"yearly_interest_rate"** - is the annual interest rate represented as a decimal;
- **"number_of_years"** - represents the duration for which the investment will earn interest;
- the function should output the future value of the investment, **rounded to four decimal places**.

You may employ the given formula to finish this task.

$$future\_value = \text{initial\_sum} \times ((1 + \text{yearly\_interest\_rate})^{\text{number\_of\_years}})$$

For this task, please only submit your **user-defined function** in the Judge system.

Example

Input	Output	Comments
fn_calculate_future_value (1000, 0.1, 5)	1610.5100	**initial_sum:** 1000 **yearly_interest_rate:** 10% **number_of_years:** 5

fn_calculate_future_value(2500, 0.30, 2)	4225.0000	
fn_calculate_future_value(500, 0.25, 10)	4656.6128	

3. User-defined Function Is Word Comprised

As part of a database of words for a language-learning app, you need to create a function that will check if a word is composed of a given set of letters. The function should return true if the word can be formed from the set of letters, and false otherwise.

Your task is to create the function **"fn_is_word_comprised"** with the following requirements:

- The function should take two input parameters: **"set_of_letters"** and **"word"**. Both parameters should be of type **VARCHAR(50)**;
- The function should return a **BOOLEAN** value indicating whether the word can be composed from the given set of letters;
- The function should be **case-insensitive**, i.e., it should treat upper-case and lower-case letters as the same;
- The function should be able to handle **words containing spaces or special characters** but should **ignore them** when checking if the word can be composed of the set of letters.

*** Note, one approach to checking whether a given word is composed of the letters in a given set is to use a **WHILE LOOP** to iterate through each character of the word and verify whether that character exists in the set of letters.

For this task, please only submit your **user-defined function** in the Judge system.

Example

Input	Output
fn_is_word_comprised('ois tmiah%f', 'halves')	false
fn_is_word_comprised('ois tmiah%f', 'Sofia')	true
fn_is_word_comprised('bobr', 'Rob')	true
fn_is_word_comprised('papopep', 'toe')	false
fn_is_word_comprised('R@o!B$B', 'Bob')	true

To prepare for the upcoming tasks, you need to create a database called ***diablo_db*** and open its query tool. You can download the ***07-Exercise-Database-Programmability-diablo_db.sql*** file from the course instance and import it into the query tab of your database. Once imported, execute the **queries provided** in the file. The schema and tables available in the ***diablo_db*** database will be used for the tasks that follow.

4. Game Over

You are working for a gaming company, and you need to create a PostgreSQL function to query the status of the ongoing games. Your task is to write a function called **"fn_is_game_over"** which takes one parameter: **"is_game_over"**, a **BOOLEAN** value indicating whether the game is over or not.

- The function should return a table with three columns: **"name"** of the game **(VARCHAR(50))**, **"game_type_id"** (**INT**), and **"is_finished"** (**BOOLEAN**);
- The function should retrieve all the rows from the **"games"** table where the **"is_finished"** column matches the **"is_game_over"** parameter.

For this task, please only submit your **user-defined function** in the Judge system.

Example

Input	name	game_type_id	is_finished
fn_is_game_over(true)	Apple	3	true
	Lisbon	1	true
	Ablajeck	2	true
	…	…	…
	Lotte World	4	true
	Victoria Peak	1	true
fn_is_game_over(false)	Aithusa	2	false
	Acid green	5	false
	Broadway	5	false
	…	…	…
	Pompeii	1	false
	Versailles	1	false

5. Difficulty Level

Write a **PostgreSQL function** that accepts a **"level"** parameter and returns the corresponding **"difficulty_level"** based on the following criteria:

- if **"level"** is **less than or equal to 40**, the **"difficulty_level"** is "**Normal Difficulty**";
- if **"level"** is **between 41 and 60 (inclusive)**, the **"difficulty_level"** is "**Nightmare Difficulty**";
- if **"level"** is **greater than 60**, the **"difficulty_level"** is "**Hell Difficulty**".

Next, write a **SQL query** that retrieves the **"user_id"**, **"level"**, **"cash"**, and **"difficulty_level"** of all users in the **"users_games"** table. Use the **"fn_difficulty_level()"** function to calculate the difficulty level for each user. Sort the result by **"user_id"** in **ascending order**.

For this task, please only submit your **user-defined function** in the Judge system.

Example

user_id	level	cash	difficulty_level
1	22	7396.0000	Normal Difficulty
	20	6699.0000	Normal Difficulty
2	67	5826.0000	Hell Difficulty
2	81	8862.0000	Hell Difficulty
…	…	…	…

4	55	7582.0000	Nightmare Difficulty
…	…	…	…
6	50	9490.0000	Nightmare Difficulty
71	21	5615.0000	Normal Difficulty
71	4	6424.0000	Normal Difficulty

6. * Cash in User Games Odd Rows

Write a PostgreSQL function named **"fn_cash_in_users_games"** that takes a parameter **"game_name"** of type **VARCHAR(50)**. The function should return a **table with one column named "total_cash"** of type **NUMERIC**.
The function should calculate the **total cash of the odd rows** of the **"users_games"** table that belong to the game specified by the **"game_name"** parameter. Rows should be ordered by **"cash"** in **descending order**.
The function should then **round the result to two decimal places**.

*** Hint, you can use the **ROW_NUMBER** function to get the ranking of each row based on certain order criteria. The function assigns a unique integer value to each row based on the order of the column or columns specified in the **OVER()** clause. This can help implement paging or filtering in your queries, or for grouping and summarizing data.

For this task, please only submit your **user-defined function** in the Judge system.

Example

Input	Output
fn_cash_in_users_games('Love in a mist')	8585.00
fn_cash_in_users_games('Delphinium Pacific Giant')	6921.00

Begin by creating a database called ***bank_db*** and then launch its query tool. After that, download the ***07-Exercise-Database-Programmability-bank_db.sql*** file from the course instance, import it into the query tab of your database, and execute the queries provided in the file. Once you've executed the queries, take some time to familiarize yourself with the ***bank_db*** database and get to know its schema and tables.

7. Retrieving Account Holders**

The purpose of this task is to create a PostgreSQL stored procedure named **"sp_retrieving_holders_with_balance_higher_than"** that accepts a **numeric** input parameter called **"searched_balance"**. The procedure will calculate the **"total_balance"** for each account **"holder"** by summing up the balances of all their accounts. It will then compare the **"total_balance"** with the **"searched_balance"** and **return a list of people** who have a total amount of money greater than the **"searched_balance"**.

To achieve this, the procedure will **iterate** through each **"holder"** in the **"account_holders"** table, **sorted by** their **"first_name"** and **"last_name"**. If the **"total_balance"** for an account holder is greater than the **"searched_balance"**, the procedure will **raise a notification** in the **format "First Name Last Name - Total Balance"**.

There is no need to submit the solution to the Judge system for this problem.

Example

This is the outcome for the **"searched_balance"** of **200000.**

```
NOTICE: Monika Miteva - 565649.2000
NOTICE: Petar Kirilov - 245656.2300
NOTICE: Petko Petkov Junior - 6546543.2300
NOTICE: Susan Cane - 5585351.2400
NOTICE: Zlatko Zlatyov - 1112627.9000
```

8. Deposit Money

As an employee at a bank, you are required to develop a PostgreSQL stored procedure th at will enable you to **deposit money into an existing account**. The stored procedure should be named **"sp_deposit_money"** and should have two parameters:

- **"account_id"** - an **integer** that **identifies the account** where the money is to be deposited;
- **"money_amount"** - a **numeric value** with a maximum **precision of four digits after the decimal point**, which specifies the **amount of money to be deposited**.

The stored procedure should **increase the account's "balance" by adding the deposited amount of money**. After that, it should **commit the transaction** to guarantee that the modifications are saved.

For this task, please only submit your **stored procedure** in the Judge system.

Example

This is the outcome for the account with an **"id"** of **1** and a **deposited amount** of **200**.

id	account_holder_id	balance
1	1	323.1200

This is the outcome for the account with an **"id"** of **10** and a **deposited amount** of **500**.

id	account_holder_id	balance
10	2	1043.3000

This is the outcome for the account with an **"id"** of **14** and a **deposited amount** of **1000**.

id	account_holder_id	balance
14	4	1001.2300

9. Withdraw Money

Your task is to create a stored procedure in PostgreSQL called **"sp_withdraw_money"**, which will withdraw money from an existing account. The procedure should have two parameters:

- **"account_id"** - an **integer** representing the **"id"** of the account to withdraw money from;
- **"money_amount"** - a **numeric** value with a precision of up to four digits after the decimal point representing the amount of money to withdraw.

The procedure should update the account's **"balance"** by **subtracting the withdrawn amount of money** only if the **account has enough balance** to make the transaction. If the account **balance is not sufficient to withdraw the specified amount**, it should **raise a notice indicating the insufficient balance**. Finally, the procedure should **commit the transaction** to ensure that the changes are saved.

For this task, please only submit your **stored procedure** in the Judge system.

Example

This is the outcome for the account with an **"id"** of **3** and a **deposited amount** of **5050.7500**.

id	account_holder_id	balance
3	12	6541492.4800

This is the outcome for the account with an **"id"** of **6** and a **deposited amount** of **5437.0000**.

```
NOTICE: Insufficient balance to withdraw 5437.0000
```

10. Money Transfer

Your new assignment involves creating a stored procedure in PostgreSQL, which will allow for the transfer of a specific amount of money from one account to another. The procedure will have three parameters:

- **"sender_id"** - an **integer** that represents the **"id"** of the **account from** which the money will be transferred;
- **"receiver_id"** - an **integer** that represents the **"id"** of the **account to** which the money will be transferred;
- **"amount"** - a **numeric** value that has a precision of up to four decimal places, which represents the **amount of money to be transferred**.

Initially, the procedure will try to withdraw the given amount of money from the account associated with the **"sender_id"** by invoking the previously defined stored procedure **"sp_withdraw_money"**. If the withdrawal operation is successful, the procedure will subsequently deposit the same amount into the account linked with the **"receiver_id"** by using the stored procedure **"sp_deposit_money"**.

In the event of any errors occurring during the transaction, the procedure will **ROLLBACK** the entire transaction to ensure data integrity is maintained.

*** Note, to ensure the successful execution of your **"sp_transfer_money"** procedure, it is important to keep in mind that you **should not** include any **COMMIT** or **ROLLBACK** statements inside the **"sp_deposit_money"** and **"sp_withdraw_money"** procedures. This is because executing these statements will commit or roll back the transaction before the **"sp_transfer_money"** procedure is able to do so, **causing unexpected results**.

For this task, please only submit your **stored procedure** in the Judge system.

Example

This is the outcome for the account with a **"sender_id"** of **5**, **"receiver_id"** of **1**, and an **"amount"** of **5000.0000**.

id	account_holder_id	balance

1	1	5323.1200
5	11	31521.2000

This is the outcome for the account with a **"sender_id"** of **10**, **"receiver_id"** of **2**, and an **"amount"** of **1043.9000**.

```
NOTICE: Insufficient balance to withdraw 1043.9000
```

This is the outcome for the account with a **"sender_id"** of **13**, **"receiver_id"** of **15**, and an **"amount"** of **400.9000**.

id	account_holder_id	balance
13	5	5034.4200
15	6	401.0900

11. Delete Procedure

The procedure named **"sp_retrieving_holders_with_balance_higher_than"** is no longer required, and it can be removed.
For this task, please only submit your **stored procedure** in the Judge system.

12. Log Accounts Trigger

Create a table called **"logs"** with columns **"id"**, **"account_id"**, **"old_sum"**, and **"new_sum"**.

Then, create a PostgreSQL function called **"trigger_fn_insert_new_entry_into_logs"** that takes no arguments and returns a trigger. Inside the function, write an SQL query that **inserts a new row into** the **"logs"** table, with the **"account_id"** value set to the **"id"** value of the old row, the **"old_sum"** value set to the **"balance"** value of the old row, and the **"new_sum"** value set to the **"balance"** value of the new row. Finally, **return the new row**.

After that, create a trigger called **"account_balance_change"** that activates **"trigger_fn_insert_new_entry_into_logs"** after an update is made to the **"accounts"** table, but only when the **"balance" value of the old row is different from the "balance" value of the new row**.

For this assignment, kindly submit only your **"CREATE TABLE" query and the trigger function** in the Judge system.

Example

Before update

id	account_id	balance
1	1	5323.1200

After update

id	account_id	old_sum	new_sum	Comments
1	1	5323.1200	150.0000	First update the account table: UPDATE accounts SET balance = 150.00 WHERE "id" = 1; Then to see the result you can: SELECT * FROM logs;

13. Notification Email on Balance Change

Your task is to implement a notification system for bank customers that sends an email whenever there is a change in their account balance. To accomplish this, you need to create a trigger called **"tr_send_email_on_balance_change"** that will be activated **whenever there is an update** made to the **"logs"** table resulting in a **change in the account balance**. The email sent to the customer should contain:

- their **"account_id"**;
- the **subject** of the email should read **"Balance change for account: {account_id}"**;
- **body** should include **"On {date} your balance was changed from {old} to {new}."**

To create the necessary infrastructure, you should first create a **"notification_emails"** table with the following columns: **"id"**, **"recipient_id"**, **"subject"**, and **"body"**.

Next, create a trigger function called **"trigger_fn_send_email_on_balance_change"** that inserts a new row into the **"notification_emails"** table whenever a row is updated in the **"logs"** table. This function should take no arguments and should return the type **TRIGGER**.

For this assignment, kindly submit only your **"CREATE TABLE" query and the trigger function** in the Judge system.

Example

Modify the **"logs"** table by increasing the **"new_sum"** value by **100** for the record with an **"account_id"** of **1**.

id	recipient_id	subject	body
1	1	Balance change for account: 1	On 2023-08-19 your balance was changed from 5323.1200 to 250.0000.

** This task is not required to be submitted to the Judge system and will not be considered in the final result.