# PL/pgSQL Functions

CODEUS_

## PL/pgSQL: SQL + Procedural Logic

- Combines SQL's data handling with procedural language features (variables, conditions, loops).
- Allows embedding complex logic directly into the database.
- Native language for PostgreSQL for efficient execution.

## Key Advantages of PL/pgSQL Functions

- **Encapsulation:** Grouping complex logic.
- **Reusability:** Write once, use many times.
- **Performance:** Can reduce network traffic and allow server-side optimizations.
- **Security:** Control data access through functions.

# Anatomy of a PL/pgSQL Function

**Key Components:**

- **CREATE FUNCTION...**: Defines the function.
- **RETURNS return_type**: Specifies the function's output data type.
- **AS $$ ... $$**: Delimits the function body (dollar quoting is common).
- **DECLARE**: (Optional) Section for local variable declarations.
    - Example: v_count INTEGER := 0;
- **BEGIN ... END;**: The main block containing executable SQL and procedural statements.
- **RETURN result_expression**: Returns a value from the function (if not returning VOID).
- **EXCEPTION WHEN ...**: (Optional) Handles errors that occur within the BEGIN...END block.
- **LANGUAGE plpgsql**: Specifies PL/pgSQL as the function's language.

```
CREATE [OR REPLACE] FUNCTION function_name (parameters)
RETURNS return_type AS $$
DECLARE
  -- variable declarations (e.g., v_some_variable INTEGER;)
BEGIN
  -- function logic
  RETURN result_expression;
EXCEPTION
  WHEN condition THEN
    -- error handling
END;
$$ LANGUAGE plpgsql;
```

CODEUS_

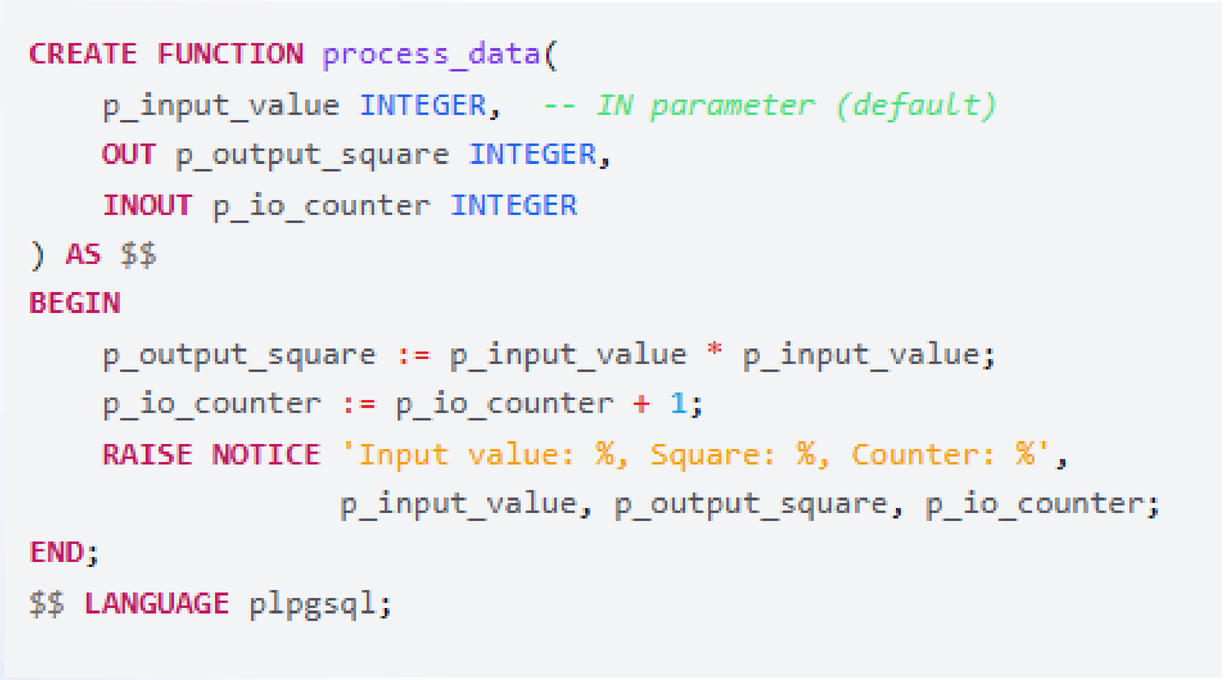
## Inputs and Outputs: Examples

**Parameter Modes:**

- IN (default): Input value.
- OUT: Output value, assigned within the function.
- INOUT: Input value, can be modified and returned.

**Common Return Types:**

- Scalar types, SETOF record_type or TABLE(...), VOID.



```sql
CREATE FUNCTION process_data(
    p_input_value INTEGER,  -- IN parameter (default)
    OUT p_output_square INTEGER,
    INOUT p_io_counter INTEGER
) AS $$
BEGIN
    p_output_square := p_input_value * p_input_value;
    p_io_counter := p_io_counter + 1;
    RAISE NOTICE 'Input value: %, Square: %, Counter: %',
                 p_input_value, p_output_square, p_io_counter;
END;
$$ LANGUAGE plpgsql;
```

## Making Decisions: The IF Statement

- **Syntax:**

```
IF condition THEN
  -- statements if condition is true
ELSIF other_condition THEN
  -- statements if other_condition is true
ELSE
  -- statements if all conditions are false
END IF;
```

- **Example:**

```
CREATE FUNCTION get_discount_type(p_amount NUMERIC)
RETURNS TEXT AS $$
DECLARE
  v_discount_type TEXT;
BEGIN
  IF p_amount > 1000 THEN
    v_discount_type := 'Large discount (15%)';
  ELSIF p_amount > 500 THEN
    v_discount_type := 'Medium discount (10%)';
  ELSIF p_amount > 100 THEN
    v_discount_type := 'Small discount (5%)';
  ELSE
    v_discount_type := 'No discount';
  END IF;
  RETURN v_discount_type;
END;
$$ LANGUAGE plpgsql;
```

CODEUS_

## Alternative Decisions: The CASE Statement

- **Simple CASE:**

```
CASE expression
  WHEN value1 THEN statements1
  WHEN value2 THEN statements2
  ...
  ELSE else_statements
END CASE;
```

- **Searched CASE:**

```
CASE
  WHEN condition1 THEN statements1
  WHEN condition2 THEN statements2
  ...
  ELSE else_statements
END CASE;
```

- **Example (Searched CASE):**

```
CREATE FUNCTION get_season(p_month INTEGER)
RETURNS TEXT AS $$
BEGIN
  RETURN CASE
    WHEN p_month IN (12, 1, 2) THEN 'Winter'
    WHEN p_month IN (3, 4, 5) THEN 'Spring'
    WHEN p_month IN (6, 7, 8) THEN 'Summer'
    WHEN p_month IN (9, 10, 11) THEN 'Autumn'
    ELSE 'Invalid month'
  END CASE;
END;
$$ LANGUAGE plpgsql;
```

## Repeating Actions: The FOR Loop

- **FOR with range:**

```
FOR counter IN [REVERSE] start_value .. end_value [BY step] LOOP
  -- statements
END LOOP;
```

- **FOR with query (iterate over rows):**

```
FOR record_variable IN SQL_query LOOP
  -- statements, access fields via record_variable.column_name
END LOOP;
```

- **Examples:**

```
-- FOR with range
CREATE FUNCTION sum_numbers(p_limit INTEGER) RETURNS INTEGER AS $$
DECLARE
  v_sum INTEGER := 0;
BEGIN
  FOR i IN 1 .. p_limit LOOP
    v_sum := v_sum + i;
  END LOOP;
  RETURN v_sum;
END;
$$ LANGUAGE plpgsql;
```

```
-- FOR with query
CREATE FUNCTION list_usernames() RETURNS VOID AS $$
DECLARE
  rec RECORD;
BEGIN
  FOR rec IN SELECT username, email FROM users ORDER BY username LOOP
    RAISE NOTICE 'User: %, Email: %', rec.username, rec.email;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

CODEUS_

## Repeating by Condition: The WHILE Loop

- **Syntax:**

```
WHILE condition LOOP
  -- statements
  -- (important that the condition eventually becomes false, otherwise infinite loop)
END LOOP;
```

- **Example:**

```
CREATE FUNCTION countdown(p_start_value INTEGER) RETURNS VOID AS $$
DECLARE
  v_counter INTEGER := p_start_value;
BEGIN
  WHILE v_counter > 0 LOOP
    RAISE NOTICE '%...', v_counter;
    v_counter := v_counter - 1;
    -- For demonstration, a small delay can be added if needed
    -- PERFORM pg_sleep(0.5); -- 0.5 second delay
  END LOOP;
  RAISE NOTICE 'Start!';
END;
$$ LANGUAGE plpgsql;
```

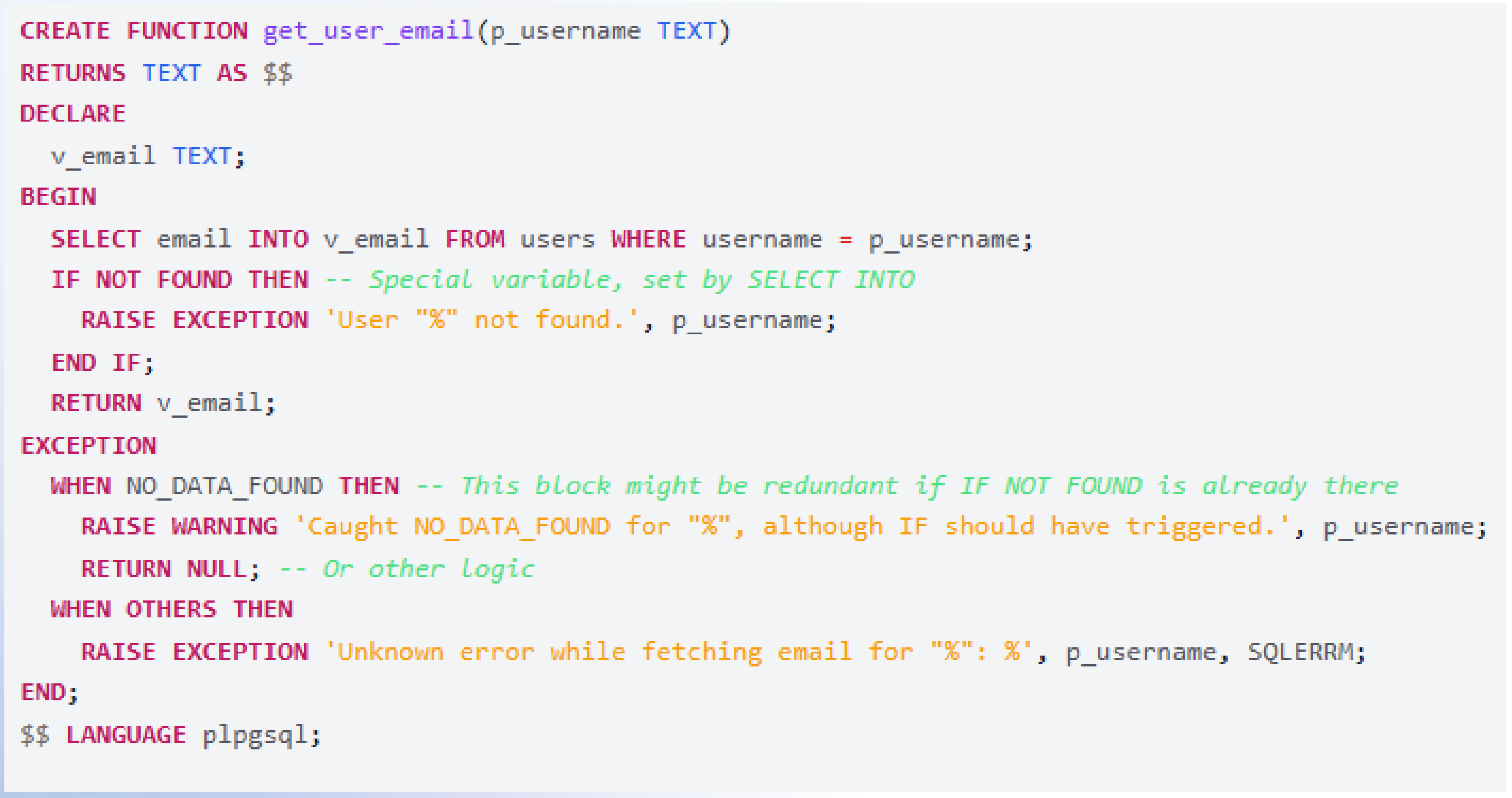
# Managing Errors Gracefully

- **EXCEPTION Block:**

  Catches errors from the BEGIN block.

- **Raising Errors:**

  RAISE EXCEPTION (stops),
  RAISE NOTICE (informs).



```sql
CREATE FUNCTION get_user_email(p_username TEXT)
RETURNS TEXT AS $$
DECLARE
  v_email TEXT;
BEGIN
  SELECT email INTO v_email FROM users WHERE username = p_username;
  IF NOT FOUND THEN -- Special variable, set by SELECT INTO
    RAISE EXCEPTION 'User "%" not found.', p_username;
  END IF;
  RETURN v_email;
EXCEPTION
  WHEN NO_DATA_FOUND THEN -- This block might be redundant if IF NOT FOUND is already there
    RAISE WARNING 'Caught NO_DATA_FOUND for "%", although IF should have triggered.', p_username;
    RETURN NULL; -- Or other logic
  WHEN OTHERS THEN
    RAISE EXCEPTION 'Unknown error while fetching email for "%": %', p_username, SQLERRM;
END;
$$ LANGUAGE plpgsql;
```

## Returning Multiple Rows

- **Concept:** Functions that return a set of rows.
- **Declaration:** RETURNS SETOF data_type or RETURNS TABLE (...).
- **Returning Data:** RETURN QUERY query; or RETURN NEXT expression;.

```
CREATE FUNCTION get_products_by_category(p_category_name TEXT)
RETURNS TABLE(product_id INTEGER, product_name TEXT, price NUMERIC) AS $$
BEGIN
  RETURN QUERY
    SELECT p.id, p.name, p.price
    FROM products p
    JOIN categories c ON p.category_id = c.id
    WHERE c.name = p_category_name;
END;
$$ LANGUAGE plpgsql;
```

CODEUS_

# Building Queries on the Fly

- **EXECUTE Command:**

Allows constructing and executing SQL queries as strings.
This is useful when table or column names are not known until runtime.

```sql
EXECUTE format('SELECT * FROM %I WHERE status = %L', target_table_name, status_value);
```

```sql
CREATE FUNCTION get_table_row_count(p_table_name TEXT)
RETURNS BIGINT AS $$
DECLARE
    v_query TEXT;
    v_count BIGINT;
BEGIN
    -- Securely construct the query using format()
    -- %I is for identifiers (like table or column names) - it quotes them if necessary.
    v_query := format('SELECT COUNT(*) FROM %I', p_table_name);
    RAISE NOTICE 'Executing: %', v_query;

    EXECUTE v_query INTO v_count; -- Execute the dynamic query and store result in v_count

    RETURN v_count;
EXCEPTION
    WHEN undefined_table THEN -- Example of specific error handling
        RAISE WARNING 'Table not found: %', p_table_name;
        RETURN -1; -- Indicate an error
    WHEN OTHERS THEN
        RAISE WARNING 'Error executing dynamic SQL for table %: %', p_table_name, SQLERRM;
        RETURN -1;
END;
$$ LANGUAGE plpgsql;

-- Example call (assuming a table 'my_users' exists):
-- SELECT get_table_row_count('my_users');
-- Example call (for a non-existent table):
-- SELECT get_table_row_count('non_existent_table');
```

## Writing Good Functions

- **Clear Naming:** Use descriptive names.
- **Simplicity:** Smaller, focused functions.
- **Error Handling:** Implement checks.
- **Security with Dynamic SQL:** Sanitize inputs.
- **Testing:** Verify with various data.

# Thank you

- Author: Denys Kuchmei
- My LinkedIn: https://www.linkedin.com/in/denys-kuchmei-7a8259208/
- Date: May 2025
- Join Codeus community in Discord
- Join Codeus community in LinkedIn