Chapter 20

Mastering Functions in PL/SQL

Mayko Freitas da Silva

Introduction

In the world of PL/SQL programming, functions are like specialized tools in a toolbox. While procedures are designed to perform actions, functions are created to compute and return values. Think of a function as a calculator that takes in numbers, performs calculations, and gives you a result.

In this chapter, you will learn about two types of functions:

1. Nested Functions: These are functions defined within other PL/SQL blocks.
2. Stand-Alone Functions: These are functions created at the schema level.

By the end of this chapter, you'll be able to create and use both types of functions, making your PL/SQL code more efficient and organized.

Lab 20.1: Creating Nested Functions

Objectives

After completing this lab, you will be able to:

- Create and use nested functions in PL/SQL blocks
- Understand the structure and behavior of functions
- Use functions in different contexts within PL/SQL blocks

Understanding Nested Functions

A nested function is a function defined within another PL/SQL block, procedure, or function. It's like having a mini-calculator inside a larger program. Let's start with a simple example:

```
DECLARE
  v_salary_difference NUMBER;

  FUNCTION calculate_salary_difference (p_salary1 NUMBER, p_salary2 NUMBER)
  RETURN NUMBER
  IS
  BEGIN
    RETURN (p_salary1 - p_salary2);
  END calculate_salary_difference;

BEGIN
  -- Method 1: Assign function result to a variable
  v_salary_difference := calculate_salary_difference(15000, 10000);
  DBMS_OUTPUT.PUT_LINE('Salary difference (Method 1): ' || v_salary_difference);

  -- Method 2: Use function directly in DBMS_OUTPUT
  DBMS_OUTPUT.PUT_LINE('Salary difference (Method 2): ' || calculate_salary_difference(20000, 18000));
END;
/
```

In this example, we define a nested function called calculate_salary_difference. Let's break down its structure:

1. Function Header:

   ```
   FUNCTION calculate_salary_difference (p_salary1 NUMBER, p_salary2 NUMBER)
   RETURN NUMBER
   ```

   This line declares the function name, its parameters, and the type of value it will return (NUMBER in this case).

2. Function Body:

   ```
   IS
   BEGIN
     RETURN (p_salary1 - p_salary2);
   END calculate_salary_difference;
   ```

   The body contains the actual calculation and the RETURN statement that sends the result back.

3. Function Usage:

We demonstrate two ways to use the function:

- Assigning its result to a variable

Using it directly within another statement

When you run this code, you'll see output like this:

```
Salary difference (Method 1): 5000
Salary difference (Method 2): 2000
```

The RETURN Statement

The RETURN statement in a function is crucial. It does two things:

1. Specifies the value to be returned by the function
2. Immediately ends the function's execution

Let's modify our previous example to illustrate this:

```
DECLARE
  v_salary_difference NUMBER;

  FUNCTION calculate_salary_difference (p_salary1 NUMBER, p_salary2 NUMBER)
  RETURN NUMBER
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Calculating salary difference...');
    RETURN (p_salary1 - p_salary2);
    DBMS_OUTPUT.PUT_LINE('This line will never be executed');
  END calculate_salary_difference;

BEGIN
  v_salary_difference := calculate_salary_difference(15000, 10000);
  DBMS_OUTPUT.PUT_LINE('Salary difference: ' || v_salary_difference);
END;
/
```

When you run this code, you'll see:

```
Calculating salary difference...
Salary difference: 5000
```

Notice that the line "This line will never be executed" doesn't appear in the output. This is because the RETURN statement immediately ends the function's execution.

Multiple RETURN Statements

Functions can have multiple RETURN statements, but only one will be executed. Here's an example using employee data from the HR schema:

```
DECLARE
  v_salary_status VARCHAR2(20);

  FUNCTION get_salary_status (p_employee_id NUMBER)
  RETURN VARCHAR2
  IS
    v_salary NUMBER;
  BEGIN
    SELECT salary INTO v_salary
    FROM employees
    WHERE employee_id = p_employee_id;

    IF v_salary < 5000 THEN
      RETURN 'Low';
    ELSIF v_salary < 10000 THEN
      RETURN 'Medium';
    ELSE
      RETURN 'High';
    END IF;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN 'Employee not found';
  END get_salary_status;

BEGIN
  -- Test with different employee IDs
  DBMS_OUTPUT.PUT_LINE('Employee 103 salary status: ' || get_salary_status(103));
  DBMS_OUTPUT.PUT_LINE('Employee 150 salary status: ' || get_salary_status(150));
  DBMS_OUTPUT.PUT_LINE('Employee 999 salary status: ' || get_salary_status(999));
END;
/
```

This function, get_salary_status, demonstrates several important concepts:

1. It uses a SELECT INTO statement to fetch data from the employees table.
2. It has multiple RETURN statements based on different conditions.
3. It includes error handling for when an employee is not found.

When you run this code, you might see output like:

```
Employee 103 salary status: Medium
Employee 150 salary status: High
Employee 999 salary status: Employee not found
```

Using Functions in Conditional Statements

Functions can be very useful in conditional statements. Here's an example:

```
DECLARE
  FUNCTION is_manager (p_employee_id NUMBER)
  RETURN BOOLEAN
  IS
    v_direct_reports NUMBER;
  BEGIN
    SELECT COUNT(*)
    INTO v_direct_reports
    FROM employees
    WHERE manager_id = p_employee_id;

    RETURN (v_direct_reports > 0);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RETURN FALSE;
  END is_manager;

BEGIN
  IF is_manager(103) THEN
    DBMS_OUTPUT.PUT_LINE('Employee 103 is a manager');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Employee 103 is not a manager');
  END IF;

  IF NOT is_manager(105) THEN
    DBMS_OUTPUT.PUT_LINE('Employee 105 is not a manager');
  END IF;
END;
/
```

In this example, the is_manager function returns a BOOLEAN value, which is perfect for use in IF statements. It checks if an employee has any direct reports, indicating whether they are a manager.

Lab 20.2: Creating Stand-Alone Functions

Objectives

After completing this lab, you will be able to:

- Create stand-alone functions at the schema level
- Understand the syntax and options for creating stand-alone functions
- Use stand-alone functions in various contexts

Creating Stand-Alone Functions

Stand-alone functions are created at the schema level and can be used across different PL/SQL blocks, procedures, and other functions. Here's the general syntax:

```
CREATE [OR REPLACE] [EDITIONABLE | NONEDITIONABLE] FUNCTION function_name
   [(parameter_name [IN | OUT | IN OUT] datatype [, ...])]
RETURN return_datatype
[DETERMINISTIC | PIPELINED | PARALLEL_ENABLE | RESULT_CACHE]
{IS | AS}
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [function_name];
```

Let's create a stand-alone function using the HR schema:

```
CREATE OR REPLACE FUNCTION get_department_name
  (p_department_id IN departments.department_id%TYPE)
RETURN VARCHAR2
IS
  v_department_name departments.department_name%TYPE;
BEGIN
  SELECT department_name
  INTO v_department_name
  FROM departments
  WHERE department_id = p_department_id;

  RETURN v_department_name;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN 'Department not found';
END get_department_name;
/
```

This function, get_department_name, takes a department ID as input and returns the corresponding department name. Here's how you can use it:

```
DECLARE
  v_dept_name VARCHAR2(30);
BEGIN
  -- Using the function with an existing department
  v_dept_name := get_department_name(60);
  DBMS_OUTPUT.PUT_LINE('Department 60: ' || v_dept_name);

  -- Using the function with a non-existing department
  v_dept_name := get_department_name(999);
  DBMS_OUTPUT.PUT_LINE('Department 999: ' || v_dept_name);

  -- Using the function in a SQL statement
  SELECT employee_id, first_name, last_name, get_department_name(department_id) as dept_name
  FROM employees
  WHERE ROWNUM <= 5;
END;
/
```

This example demonstrates three ways to use the stand-alone function:

1. Assigning its result to a PL/SQL variable
2. Using it with invalid input to show error handling
3. Incorporating it in a SQL SELECT statement

Functions Returning Complex Data Types

Stand-alone functions can also return complex data types like records. Here's an example:

```
CREATE OR REPLACE FUNCTION get_employee_details
   (p_employee_id IN employees.employee_id%TYPE)
RETURN employees%ROWTYPE
IS
   v_emp_record employees%ROWTYPE;
BEGIN
   SELECT *
   INTO v_emp_record
   FROM employees
   WHERE employee_id = p_employee_id;

   RETURN v_emp_record;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      v_emp_record.first_name := 'Not';
      v_emp_record.last_name := 'Found';
      RETURN v_emp_record;
END get_employee_details;
/
```

This function returns an entire row from the employees table as a record. Here's how you can use it:

```
DECLARE
   v_emp_record employees%ROWTYPE;
BEGIN
   -- Get details for an existing employee
   v_emp_record := get_employee_details(103);
   DBMS_OUTPUT.PUT_LINE('Employee 103: ' || v_emp_record.first_name || ' ' || v_emp_record.last_name);
   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_record.salary);

   -- Try with a non-existing employee
   v_emp_record := get_employee_details(999);
   DBMS_OUTPUT.PUT_LINE('Employee 999: ' || v_emp_record.first_name || ' ' || v_emp_record.last_name);

   -- Using the function in a query
   SELECT e.employee_id,
          get_employee_details(e.employee_id).job_id as job,
          get_employee_details(e.employee_id).salary as salary
   FROM (SELECT employee_id FROM employees WHERE ROWNUM <= 5) e;
END;
/
```

This example shows how to work with the record returned by the function and how to use the function in a SQL query.

Summary

In this chapter, you've learned about creating and using functions in PL/SQL. We covered:

1. Nested functions: Defined within PL/SQL blocks
2. Stand-alone functions: Created at the schema level
3. Function structure: Parameters, RETURN clause, and function body
4. Using functions in various contexts: Assignments, conditions, and SQL statements
5. Functions returning complex data types like records

Functions are powerful tools in PL/SQL that allow you to encapsulate logic, improve code reusability, and create more modular and maintainable code.

By the Way

The concepts and examples provided in this chapter are foundational for working with functions in PL/SQL. To further enhance your understanding:

1. Experiment with creating functions that use different data types and complex logic.
2. Try combining multiple functions in a single PL/SQL block or SQL statement.
3. Explore how functions can be used to simplify complex queries or calculations in your database operations.

Remember, practice is key to mastering these concepts. Try modifying the examples or creating your own functions to solve specific problems using the HR schema or any other database you're working with.

Mastering Functions in PL/SQL