

UNIT 7

Functions and Packages

(5 Hours) — Database Programming with PL/SQL

This unit covers two further PL/SQL program units. A function is, like a procedure, a named block that can be called by other programs — but unlike a procedure, it always computes and returns a single value, so it can be used directly inside expressions. A package is a schema object that bundles related procedures, functions, variables, and types together under one name, giving PL/SQL a way to organise large applications into well-defined, reusable modules.

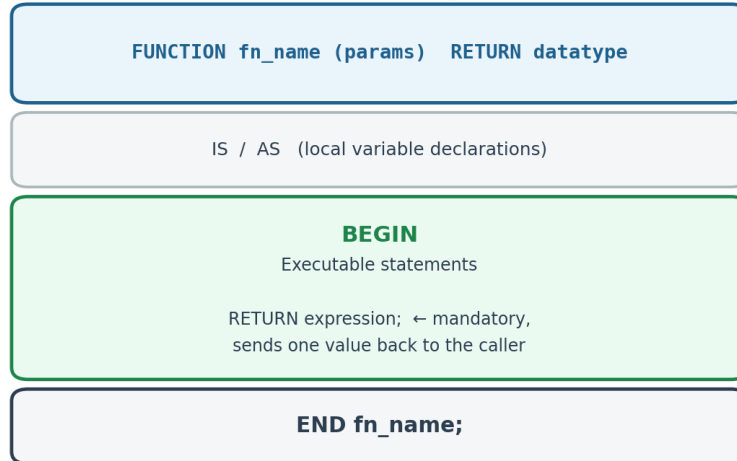
7.1 Functions

A function is a named, stored PL/SQL block that performs a computation and returns exactly one value to its caller through a mandatory **RETURN** statement. Because a function evaluates to a value, it can be used anywhere an expression is allowed — in a **SELECT** list, a **WHERE** clause, an assignment, or inside another PL/SQL expression — which is the key difference from a procedure.

Aspect	Procedure	Function
Purpose	Performs an action	Computes and returns a value
RETURN statement	Optional, without a value (exits early)	Mandatory, must return a value of the declared type
Can appear in an expression?	No	Yes — e.g. inside a SELECT list
Typical call	<code>EXECUTE proc_name(...);</code> or <code>proc_name(...);</code> in a block	<code>var := fn_name(...);</code> or <code>SELECT fn_name(...) FROM ...;</code>

7.1.1 Creating and Invoking a Function

Structure of a PL/SQL Function



A function must return exactly one value via RETURN, and can be called wherever an expression is valid.

Fig 7.1 — Structure of a PL/SQL function

A function is created with **CREATE [OR REPLACE] FUNCTION**, followed by a parameter list and a mandatory **RETURN datatype** clause in the header that states the data type of the value the function will send back.

Syntax — Creating a Function

```
CREATE OR REPLACE FUNCTION function_name
  (parameter1 IN datatype,
   parameter2 IN datatype)
  RETURN return_datatype
IS
  -- local variable declarations (optional)
BEGIN
  -- executable statements
  RETURN expression;          -- mandatory
EXCEPTION
  -- exception-handling statements (optional)
END function_name;
/
```

Example 7.1 — A Function that Calculates Annual Salary

Create a function that takes a monthly salary as input and returns the equivalent annual salary, then call it in two different ways.

Syntax

```
CREATE OR REPLACE FUNCTION annual_salary (p_monthly IN NUMBER)
    RETURN NUMBER
IS
    v_annual NUMBER;
BEGIN
    v_annual := p_monthly * 12;
    RETURN v_annual;
END annual_salary;
/
-- Invoking from an anonymous block:
DECLARE
    v_result NUMBER;
BEGIN
    v_result := annual_salary(45000);
    DBMS_OUTPUT.PUT_LINE('Annual salary: ' || v_result);
END;
/
-- Invoking directly inside a SQL statement:
SELECT employee_id, annual_salary(salary) AS yearly_pay
FROM employees;
```

Explanation

annual_salary declares its return type (NUMBER) right in the header. Inside the block it computes v_annual and hands it back with RETURN v_annual, which ends the function immediately. Because the function returns a value, it can be assigned to a variable in a PL/SQL block or called directly in a SQL SELECT list.

Note: A function can contain more than one RETURN statement along different execution paths, but exactly one of them will execute before control leaves the function — a function can never fall through to END without returning a value.

7.1.2 Optimizing Function Execution

Calling a PL/SQL function — especially from within SQL — has a cost, so several techniques help reduce unnecessary re-execution and context switching between the SQL and PL/SQL engines:

Technique	How it helps
PRAGMA UDF	Tells the compiler the function is intended mainly for use inside SQL statements, allowing it to generate more efficient code for that context.
DETERMINISTIC keyword	Marks a function as always returning the same output for the same input; Oracle can then cache and reuse results instead of re-executing the function for repeated calls, e.g. inside a query.

Technique	How it helps
Scalar subquery caching	Oracle automatically caches the results of repeated identical scalar-subquery function calls within the same SQL statement, avoiding duplicate execution.
Minimizing context switches	Reducing calls that alternate between SQL and PL/SQL engines (each switch has overhead) — for example, by moving logic into a single SQL expression instead of a function called per row.
Avoiding functions in WHERE clauses on indexed columns	Wrapping an indexed column in a function typically prevents the optimizer from using the index; a function-based index can be created instead if this is unavoidable.

Example 7.2 — Using DETERMINISTIC for Optimization

A function that converts a currency code to its full name never changes for a given input, so it is safe — and faster — to mark it DETERMINISTIC.

Syntax

```
CREATE OR REPLACE FUNCTION currency_name (p_code IN VARCHAR2)
  RETURN VARCHAR2
  DETERMINISTIC
IS
BEGIN
  CASE p_code
    WHEN 'USD' THEN RETURN 'US Dollar';
    WHEN 'NPR' THEN RETURN 'Nepalese Rupee';
    WHEN 'INR' THEN RETURN 'Indian Rupee';
    ELSE RETURN 'Unknown';
  END CASE;
END currency_name;
/
```

Note: DETERMINISTIC is a promise to the optimizer, not an enforced rule — marking a function that is not truly deterministic (e.g. one that reads changing table data) can lead to stale, incorrect cached results.

7.2 Packages

A package is a schema object that groups logically related PL/SQL types, variables, constants, cursors, procedures, and functions into a single named unit. Packages have exactly two parts, compiled and stored separately:

Package Specification vs Package Body

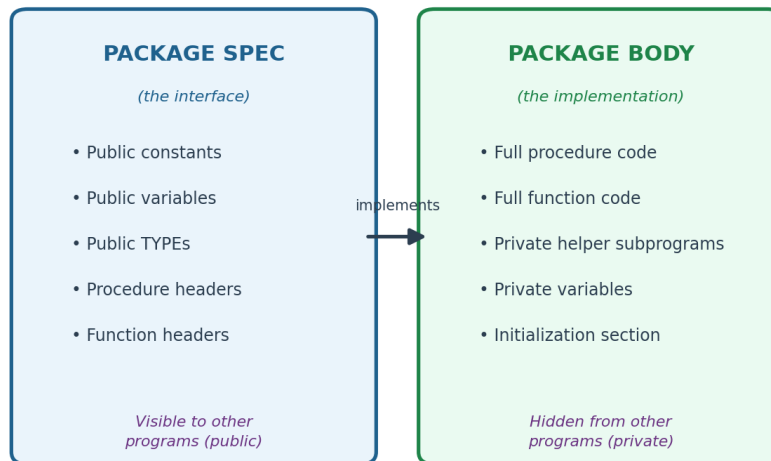


Fig 7.2 — The package specification defines the interface; the package body implements it

Part	Keyword	Contains
Specification (spec)	CREATE [OR REPLACE] PACKAGE	The public interface only — headers of procedures/functions, public types, constants, and variables that other programs are allowed to see and use.
Body	CREATE [OR REPLACE] PACKAGE BODY	The full implementation of every subprogram declared in the spec, plus any private helper subprograms or variables not visible outside the package.

7.2.1 Creating a Package

Example 7.3 — A Simple Package for Employee Operations

Create a package spec that exposes one procedure and one function, then a package body that implements both.

Syntax

```
-- Package specification
CREATE OR REPLACE PACKAGE emp_pkg
IS
    PROCEDURE give_raise (p_emp_id IN NUMBER, p_amount IN NUMBER);
    FUNCTION  get_salary (p_emp_id IN NUMBER) RETURN NUMBER;
END emp_pkg;
/

-- Package body
CREATE OR REPLACE PACKAGE BODY emp_pkg
IS
    PROCEDURE give_raise (p_emp_id IN NUMBER, p_amount IN NUMBER)
    IS
    BEGIN
        UPDATE employees
        SET salary = salary + p_amount
        WHERE employee_id = p_emp_id;
    END give_raise;
    FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER
    IS
        v_salary NUMBER;
    BEGIN
        SELECT salary INTO v_salary
        FROM employees WHERE employee_id = p_emp_id;
        RETURN v_salary;
    END get_salary;
END emp_pkg;
/

-- Calling package subprograms using dot notation:
BEGIN
    emp_pkg.give_raise(100, 2000);
    DBMS_OUTPUT.PUT_LINE(emp_pkg.get_salary(100));
END;
/
```

Explanation

Other programs see only what emp_pkg's specification exposes — the two headers. The body supplies the real code for each, and is compiled as a separate object. Both subprograms are called using package_name.subprogram_name dot notation.

7.2.2 Extending a Package

A package can be extended by adding new public subprograms, constants, or types to its specification, or by adding new private helpers to its body — without breaking programs that already use it, as long as the existing public interface is preserved. Extension is done using

CREATE OR REPLACE PACKAGE and CREATE OR REPLACE PACKAGE BODY again, this time with the additional elements included.

Example 7.4 — Extending emp_pkg with a New Function

Add a new function, department_total, to the existing emp_pkg without touching give_raise or get_salary.

Syntax

```
CREATE OR REPLACE PACKAGE emp_pkg
IS
    PROCEDURE give_raise (p_emp_id IN NUMBER, p_amount IN NUMBER);
    FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER;
    FUNCTION department_total (p_dept_id IN NUMBER) RETURN NUMBER; -- ne
w
END emp_pkg;
/
CREATE OR REPLACE PACKAGE BODY emp_pkg
IS
    PROCEDURE give_raise (p_emp_id IN NUMBER, p_amount IN NUMBER)
    IS BEGIN
        UPDATE employees SET salary = salary + p_amount
        WHERE employee_id = p_emp_id;
    END give_raise;
    FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER
    IS
        v_salary NUMBER;
    BEGIN
        SELECT salary INTO v_salary FROM employees
        WHERE employee_id = p_emp_id;
        RETURN v_salary;
    END get_salary;
    FUNCTION department_total (p_dept_id IN NUMBER) RETURN NUMBER -- new
    IS
        v_total NUMBER;
    BEGIN
        SELECT SUM(salary) INTO v_total FROM employees
        WHERE department_id = p_dept_id;
        RETURN v_total;
    END department_total;
END emp_pkg;
/
```

Note: Because the spec must be re-created with CREATE OR REPLACE, any dependent objects (procedures, views, etc.) that reference the package are marked INVALID and automatically recompiled the next time they are used.

7.2.3 Package Instantiation and Initialization

A package's variables and cursors are not created the moment the database starts — they come into existence only when the package is first referenced within a session. This is called instantiation. If the package body contains an executable section after all subprogram declarations (a block with no header, just BEGIN...END), that section runs exactly once per session, immediately after instantiation — this is called initialization.

Package Instantiation and Initialization

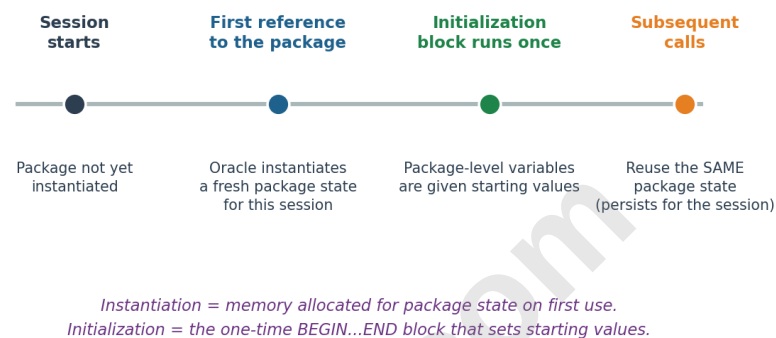


Fig 7.3 — A package is instantiated on first reference; its initialization block then runs once per session

Example 7.5 — A Package with an Initialization Section

Track how many times `give_raise` has been called during the current session, using a package-level variable set up by an initialization block.

Syntax

```
CREATE OR REPLACE PACKAGE BODY emp_pkg
IS
    v_call_count NUMBER;          -- package-level (persists for the session)
    PROCEDURE give_raise (p_emp_id IN NUMBER, p_amount IN NUMBER)
    IS
    BEGIN
        UPDATE employees SET salary = salary + p_amount
        WHERE employee_id = p_emp_id;
        v_call_count := v_call_count + 1;
    END give_raise;
    FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER
    IS
        v_salary NUMBER;
    BEGIN
        SELECT salary INTO v_salary FROM employees
        WHERE employee_id = p_emp_id;
        RETURN v_salary;
    END get_salary;
BEGIN
    -- Initialization section: runs ONCE, the first time this
    -- session references emp_pkg
    v_call_count := 0;
END emp_pkg;
/
```

Explanation

`v_call_count` is declared at package level, outside any single procedure, so it keeps its value across separate calls within the same session — instantiation allocates its storage, and the trailing `BEGIN...END` block initializes it to 0 exactly once. Every later call to `give_raise` in that session increments the same in-memory counter.

Note: Each database session gets its own independent copy of a package's state; one session's calls to `give_raise` do not affect another session's `v_call_count`.

Key Takeaways

- ✓ A function always returns exactly one value via a mandatory RETURN statement and can be used inside expressions and SQL.
- ✓ A function is created with CREATE [OR REPLACE] FUNCTION ... RETURN datatype, distinguishing it from a procedure.
- ✓ DETERMINISTIC, PRAGMA UDF, and minimizing context switches all help optimize function execution, especially when called from SQL.
- ✓ A package bundles related types, variables, procedures, and functions into one named unit with two parts: spec and body.
- ✓ The package specification is the public interface; the package body holds the full implementation and any private elements.
- ✓ A package can be extended with CREATE OR REPLACE by adding new public or private elements without breaking existing callers.
- ✓ A package is instantiated on first reference in a session; an optional trailing BEGIN...END block initializes it exactly once.

— End of Unit 7 —