



Creating Functions

Objectives

After completing this lesson, you should be able to do the following:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function

Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value

Creating Functions

The PL/SQL block must have at least one `RETURN` statement.

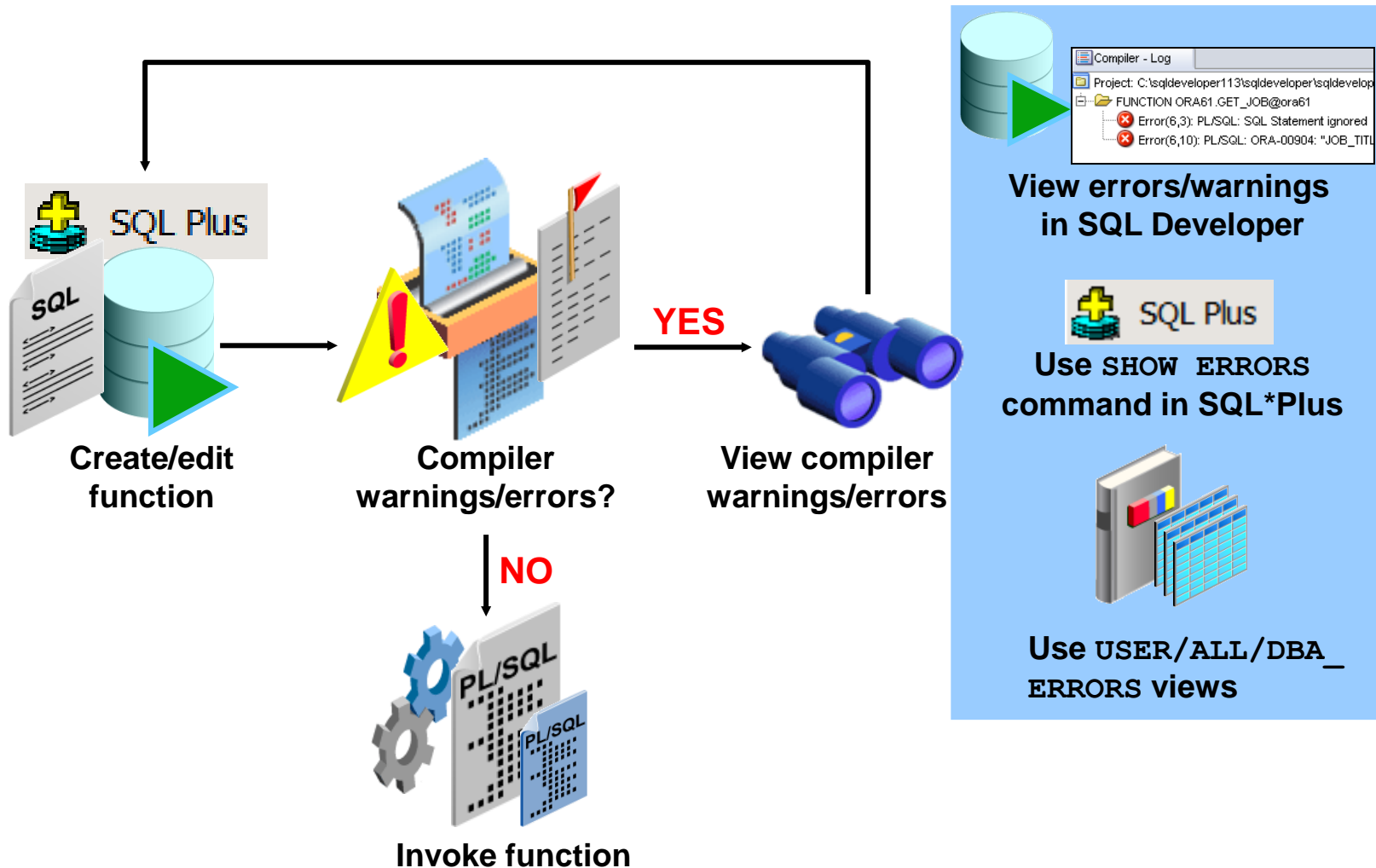
```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter1 [mode1] datatype1, . . .)]
RETURN datatype IS|AS
  [local_variable_declarations;
   . . .]
BEGIN
  -- actions;
  RETURN expression;
END [function_name];
```

PL/SQL Block

The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain <code>RETURN</code> clause in the header	Must contain a <code>RETURN</code> clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a <code>RETURN</code> statement without a value	Must contain at least one <code>RETURN</code> statement

Creating and Running Functions: Overview



Creating and Invoking a Stored Function Using the CREATE FUNCTION Statement: Example

```
CREATE OR REPLACE FUNCTION get_sal
(p_id employees.employee_id%TYPE) RETURN NUMBER IS
  v_sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO    v_sal
  FROM    employees
  WHERE   employee_id = p_id;
  RETURN v_sal;
END get_sal; /
```

```
FUNCTION get_sal Compiled.
```

```
-- Invoke the function as an expression or as
-- a parameter value.
```

```
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed
24000
```

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables
```

```
VARIABLE b_salary NUMBER  
EXECUTE :b_salary := get_sal(100)
```

```
anonymous block completed  
b_salary  
-----  
24000
```

```
-- As a PL/SQL expression, get the results using a local  
-- variable
```

```
DECLARE  
    sal employees.salary%type;  
BEGIN  
    sal := get_sal(100);  
    DBMS_OUTPUT.PUT_LINE('The salary is: ' || sal);  
END;/
```

```
anonymous block completed  
The salary is: 24000
```


Using Different Methods for Executing Functions

```
-- Use as a parameter to another subprogram  
  
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed  
24000
```

```
-- Use in a SQL statement (subject to restrictions)  
  
SELECT job_id, get_sal(employee_id) FROM employees;
```

JOB_ID	GET_SAL(EMPLOYEE_ID)
SH_CLERK	2600
SH_CLERK	2600
AD_ASST	4400
MK_MAN	13000

...

SH_CLERK	3100
SH_CLERK	3000

107 rows selected

Creating and Compiling Functions Using SQL Developer

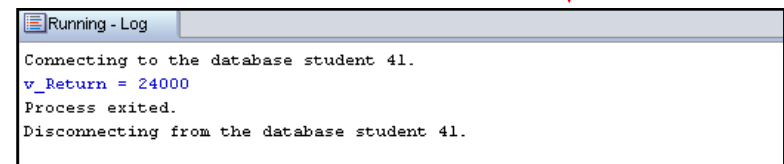
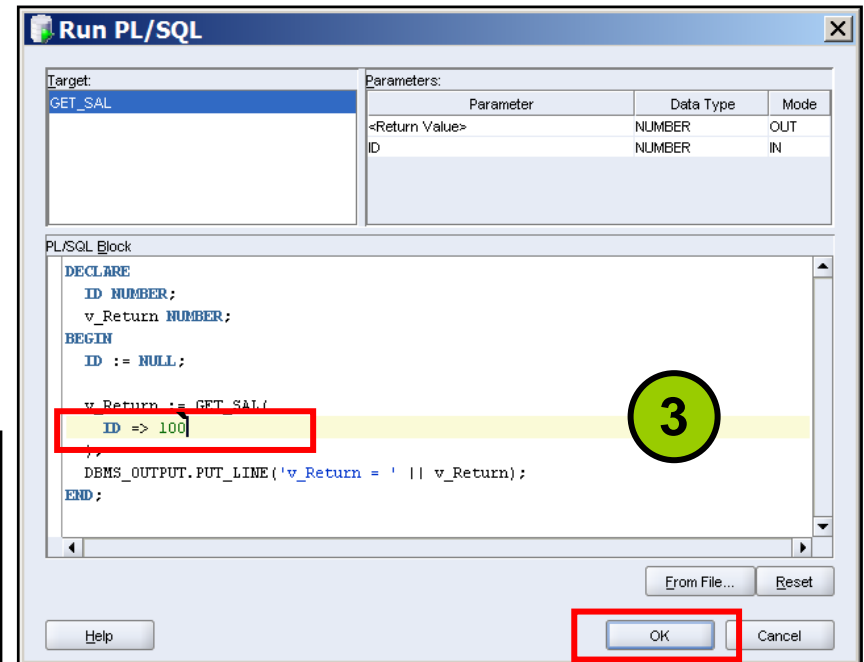
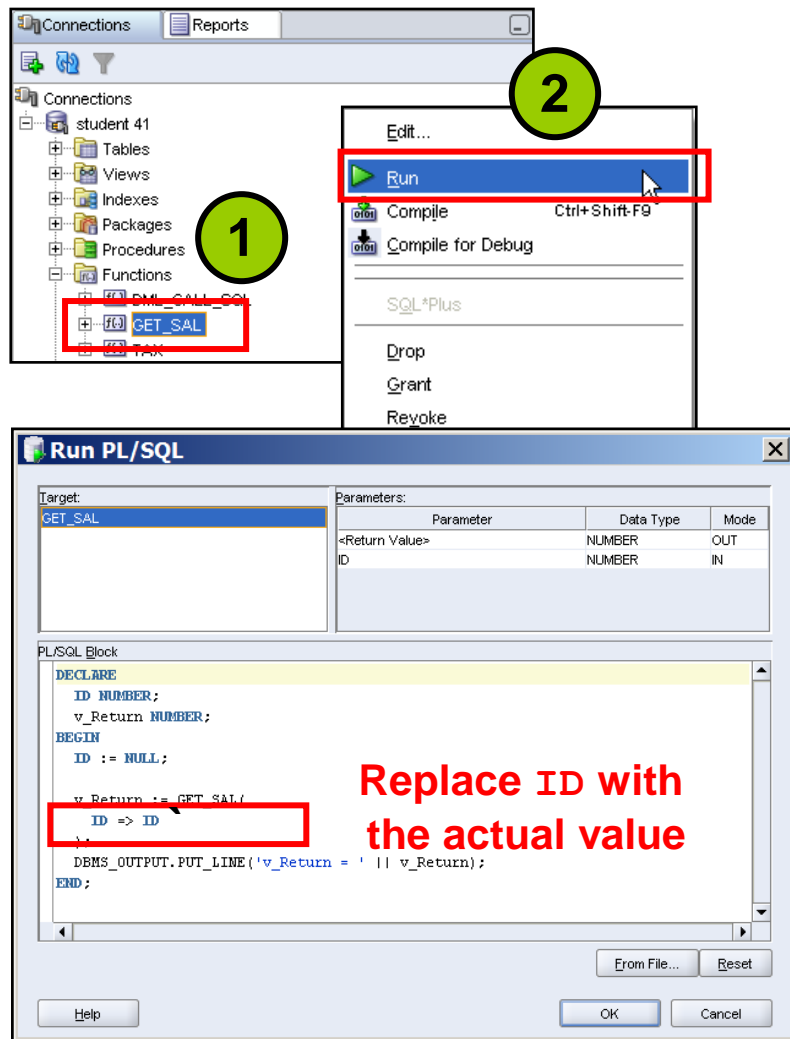
The diagram illustrates the process of creating and compiling a PL/SQL function in SQL Developer, consisting of four numbered steps:

- Step 1:** In the **Connections** pane, the **Functions** folder is selected.
- Step 2:** The **New Function...** option is chosen from the context menu.
- Step 3:** The **Create PL/SQL Function** dialog box is shown. The **Schema** is set to **ORA41** and the **Name** is **get_sal**. The **Parameters** tab is active, showing a return type of **VARCHAR2**. The **OK** button is highlighted.
- Step 4:** The **SQL Editor** window displays the SQL code for the function. The **Run** button (a green play icon) is highlighted to compile the function.

The SQL code shown in the editor is:

```
CREATE OR REPLACE FUNCTION get_sal(id employees.employee_id%TYPE) RETURN NUMBER
IS
sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO sal
  FROM employees
  WHERE employee_id = id;
  RETURN sal;
END get_sal;
```

Executing Functions Using SQL Developer

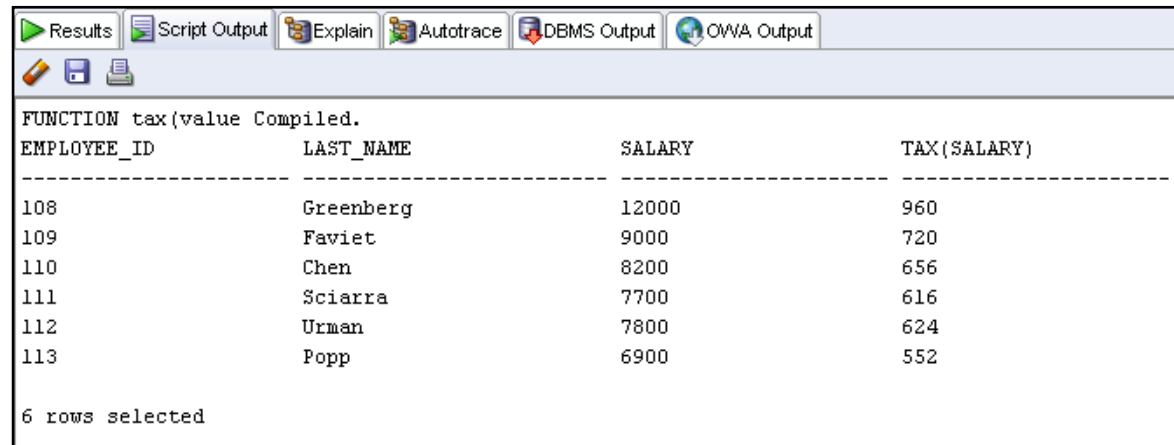


Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the `WHERE` clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```



EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected

Calling User-Defined Functions in SQL Statements

User-defined functions act like built-in single-row functions and can be used in:

- The `SELECT` list or clause of a query
- Conditional expressions of the `WHERE` and `HAVING` clauses
- The `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses of a query
- The `VALUES` clause of the `INSERT` statement
- The `SET` clause of the `UPDATE` statement

Restrictions When Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
 - Be stored in the database
 - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
 - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
 - You must own the function or have the `EXECUTE` privilege

Controlling Side Effects When Calling Functions from SQL Expressions

Functions called from:

- A `SELECT` statement cannot contain DML statements
- An `UPDATE` or `DELETE` statement on a table `T` cannot query or contain DML on the same table `T`
- SQL statements cannot end transactions (that is, cannot execute `COMMIT` or `ROLLBACK` operations)

Note: Calls to subprograms that break these restrictions are also not allowed in the function.

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_sal NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                        email, hire_date, job_id, salary)
VALUES(1, 'Frost', 'jfrost@company.com',
      SYSDATE, 'SA_MAN', p_sal);
  RETURN (p_sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170;
```

```
FUNCTION dml_call_sql(p_sal Compiled.

Error starting at line 1 in command:
UPDATE employees
  SET salary = dml_call_sql(2000)
WHERE employee_id = 170
Error report:
SQL Error: ORA-04091: table ORA62.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA62.DML_CALL_SQL", line 4
04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"
*Cause:      A trigger (or a user defined plsql function that is referenced in
              this statement) attempted to look at (or modify) a table that was
              in the middle of being modified by the statement which fired it.
*Action:     Rewrite the trigger (or function) so it does not read that table.
```

Named and Mixed Notation from SQL

- PL/SQL allows arguments in a subroutine call to be specified using positional, named, or mixed notation
- Prior to Oracle Database 11g, only the positional notation is supported in calls from SQL
- Starting in Oracle Database 11g, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements
- For long parameter lists, with most having default values, you can omit values from the optional parameters
- You can avoid duplicating the default value of the optional parameter at each call site

Named and Mixed Notation from SQL: Example

```
CREATE OR REPLACE FUNCTION f(  
  p_parameter_1 IN NUMBER DEFAULT 1,  
  p_parameter_5 IN NUMBER DEFAULT 5)  
RETURN NUMBER  
IS  
  v_var number;  
BEGIN  
  v_var := p_parameter_1 + (p_parameter_5 * 2);  
  RETURN v_var;  
END f;  
/
```

```
FUNCTION f( Compiled.
```

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

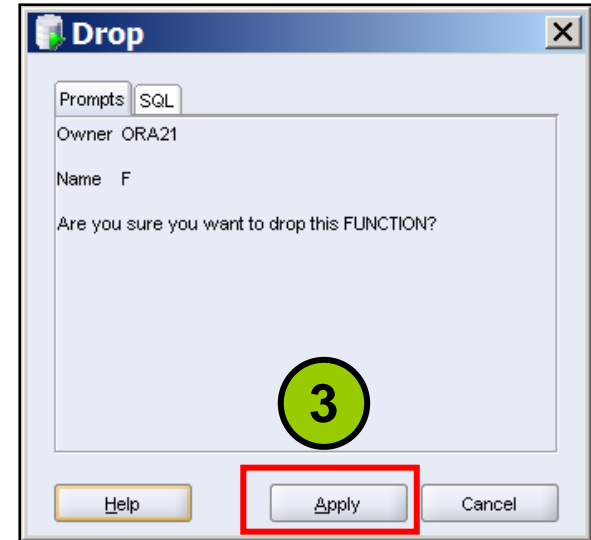
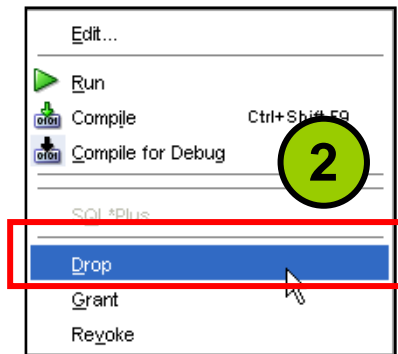
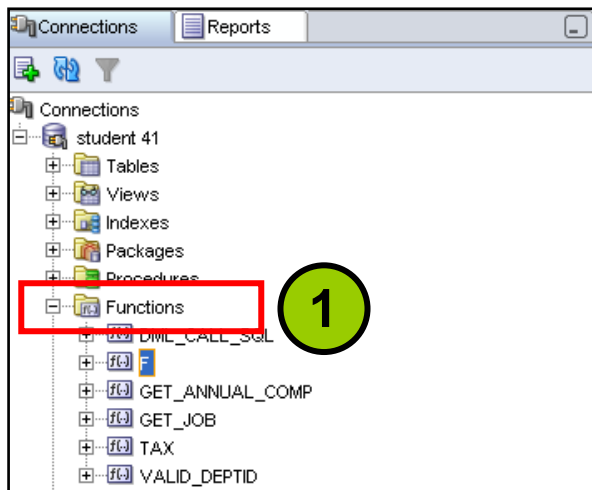
```
F(P_PARAMETER_5=>10)  
-----  
21  
  
1 rows selected
```

Removing Functions: Using the DROP SQL Statement or SQL Developer

- Using the DROP statement:

```
DROP FUNCTION f;
```

- Using SQL Developer:



Viewing Functions

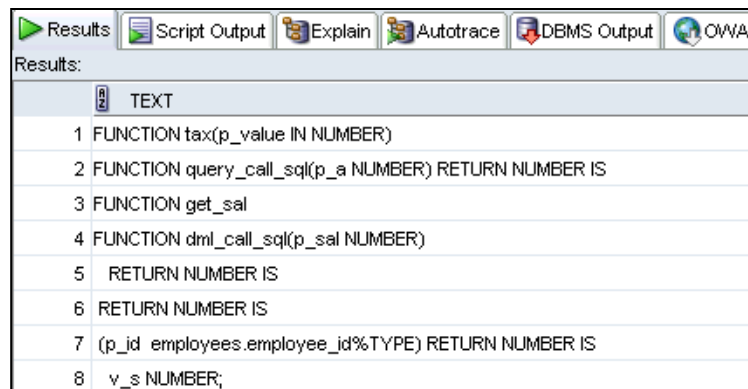
Using Data Dictionary Views

```
DESCRIBE USER_SOURCE
```

```
DESCRIBE user_source
Name                Null    Type
-----
NAME                VARCHAR2(30)
TYPE                VARCHAR2(12)
LINE                NUMBER
TEXT                VARCHAR2(4000)

4 rows selected
```

```
SELECT  text
FROM    user_source
WHERE   type = 'FUNCTION'
ORDER  BY line;
```



	TEXT
1	FUNCTION tax(p_value IN NUMBER)
2	FUNCTION query_call_sql(p_a NUMBER) RETURN NUMBER IS
3	FUNCTION get_sal
4	FUNCTION dml_call_sql(p_sal NUMBER)
5	RETURN NUMBER IS
6	RETURN NUMBER IS
7	(p_id employees.employee_id%TYPE) RETURN NUMBER IS
8	v_s NUMBER;

Quiz

A PL/SQL function:

1. Can be invoked as part of an expression
2. Must contain a `RETURN` clause in the header
3. Must return a single value
4. Must contain at least one `RETURN` statement
5. Does not contain a `RETURN` clause in the header

Summary

In this lesson, you should have learned how to:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function

Practice 3: Overview

This practice covers the following topics:

- Creating stored functions:
 - To query a database table and return specific values
 - To be used in a SQL statement
 - To insert a new row, with specified parameter values, into a database table
 - Using default parameter values
- Invoking a stored function from a SQL statement
- Invoking a stored function from a stored procedure