PL/SQL

Autor:

Kevin Cárdenas.

2023

Índice

1. Introducción

PL/SQL (Procedural Language/Structured Query Language) es un lenguaje de programación procedural desarrollado por Oracle Corporation para su uso con el sistema de gestión de bases de datos Oracle. Fue creado en la década de 1990 como una extensión del lenguaje SQL estándar, con el objetivo de proporcionar capacidades avanzadas de programación y lógica empresarial en el contexto de una base de datos relacional.

PL/SQL combina elementos de lenguajes de programación procedurales tradicionales, como variables, estructuras de control de flujo y subrutinas, con la potencia del lenguaje SQL para interactuar con la base de datos. Esto permite a los desarrolladores crear aplicaciones más complejas y sofisticadas que pueden aprovechar todas las capacidades de una base de datos relacional.

Una de las principales ventajas de PL/SQL es su estrecha integración con Oracle Database. Los programas PL/SQL se ejecutan directamente en el servidor de la base de datos, lo que reduce la necesidad de enviar múltiples consultas desde una aplicación cliente y minimiza la cantidad de datos transferidos a través de la red. Esto mejora significativamente el rendimiento y la eficiencia de las aplicaciones, especialmente en entornos empresariales donde el acceso a la base de datos es fundamental.

PL/SQL se utiliza ampliamente para desarrollar funciones, procedimientos almacenados, desencadenadores (triggers) y paquetes, que encapsulan la lógica de negocio y proporcionan una capa de abstracción adicional sobre los datos almacenados en la base de datos. Estas construcciones permiten una mejor organización y modularidad del código, promoviendo la reutilización y el mantenimiento eficiente de la lógica empresarial.

1.1. Bloques de PL/SQL

Un bloque de PL/SQL es una unidad básica de código en PL/SQL. Puede contener declaraciones, sentencias SQL y lógica de programación. Los bloques de PL/SQL se utilizan para encapsular la lógica de negocio y pueden ser anónimos o nombrados.

Los bloques anónimos se ejecutan de forma inmediata, mientras que los bloques nombrados se almacenan en la base de datos y se pueden invocar desde otras partes del sistema.

A continuación, se muestra un ejemplo de un bloque de PL/SQL anónimo:

```
1 SET SERVEROUTPUT ON --Imprimir mensajes en consola
2
3 DECLARE
4     nombre VARCHAR2(50) := 'Juan';
```

```
    edad NUMBER := 30;
BEGIN
    -- Logic of program
    IF edad >= 18 THEN
        DBMS_OUTPUT.PUT_LINE(nombre || ' es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE(nombre || ' es menor de edad');
    END IF;
END;

-- Sentencias SQL
INSERT INTO empleados (nombre, salario)
    VALUES ('Ana', 5000);

COMMIT;
END;
/
```

En este ejemplo, el bloque de PL/SQL anónimo comienza con la palabra clave "BEGIN" y finaliza con "/" para indicar el final del bloque. Dentro del bloque, se pueden realizar declaraciones como la declaración de variables y constantes. Además, se puede incluir lógica de programación como condicionales y sentencias SQL para manipular la base de datos.

Por otro lado, los bloques de PL/SQL nombrados se almacenan en la base de datos y se pueden invocar desde otras partes del sistema. A continuación se muestra un ejemplo de un procedimiento almacenado, que es un tipo de bloque de PL/SQL nombrado:

```
CREATE OR REPLACE PROCEDURE calcular_salario(p_empleado_id NUMBER) AS
v_salario NUMBER;
BEGIN
    -- Logica para calcular el salario del empleado
    SELECT salario INTO v_salario
        FROM empleados
        WHERE empleado_id = p_empleado_id;

    DBMS_OUTPUT.PUT_LINE('El salario del empleado ' || p_empleado_id || ' es: ' || v_salario);
END;
/
```

En este ejemplo, se crea un procedimiento almacenado llamado "calcular_salario" que acepta un parámetro de empleado_id. Dentro del procedimiento, se realiza una consulta para obtener el salario del

empleado correspondiente al empleado_id proporcionado. Luego, se muestra el salario utilizando la función DBMS_OUTPUT.PUT_LINE.

Además de los bloques anónimos y los procedimientos almacenados, PL/SQL también ofrece la posibilidad de crear disparadores (triggers). Los disparadores son bloques de PL/SQL que se ejecutan automáticamente en respuesta a eventos específicos que ocurren en la base de datos, como la inserción, actualización o eliminación de datos en una tabla.

A continuación se muestra un ejemplo de un disparador (trigger) que se activa después de insertar una nueva fila en la tabla "empleados":

```
CREATE OR REPLACE TRIGGER insertar_empleado_trigger
AFTER INSERT ON empleados
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Nuevo empleado insertado: ' || :NEW.nombre);
END;
/
```

En este ejemplo, el disparador "insertar_empleado_trigger" se ejecuta después de cada inserción en la tabla "empleados". El bloque de PL/SQL dentro del disparador muestra un mensaje que indica el nombre del nuevo empleado que se ha insertado.

Los disparadores son una poderosa herramienta en PL/SQL que permiten automatizar acciones y aplicar lógica adicional en la base de datos en respuesta a eventos específicos.

1.2. Variables y constantes

En PL/SQL, se pueden declarar variables y constantes para almacenar y manipular datos. Las variables se utilizan para almacenar valores temporales y pueden cambiar durante la ejecución del programa. Las constantes, por otro lado, son valores fijos que no pueden modificarse una vez que se les ha asignado un valor. Tanto las variables como las constantes pueden tener diferentes tipos de datos, como enteros, caracteres, fechas, etc.

Algunos de los tipos de datos más comunes en PL/SQL. Cada tipo de dato tiene características específicas y se debe elegir según el tipo de datos que se desea almacenar y manipular en la base de datos.

- **VARCHAR2**: se utiliza para almacenar cadenas de caracteres.

```
DECLARE
    nombre VARCHAR2(50) := 'Juan';
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Nombre: ' || nombre);
END;
```

- **NUMBER**: se utiliza para almacenar números enteros o decimales.

```
DECLARE
    sueldo NUMBER(10, 2) := 5000.50;
    extras NUMBER(10, 2) := 200.50;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Saldo: ' || TO_CHAR(sueldo + extras));
END;
```

- **DATE**: se utiliza para almacenar fechas y horas.

```
DECLARE
    fecha_nacimiento DATE := TO_DATE('1990/01/01', 'YYYY/MM/DD');
BEGIN
    DBMS_OUTPUT.PUT_LINE('Fecha de nacimiento: ' || TO_CHAR(fecha_nacimiento, 'DD/MM/YYYY'));
END;
```

- **BOOLEAN**: se utiliza para almacenar valores de verdadero o falso.

```
DECLARE
    es_mayor BOOLEAN := TRUE;
BEGIN
    IF es_mayor THEN
        DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Es menor de edad');
    END IF;
END;
```

- **CHAR**: se utiliza para almacenar caracteres de longitud fija.

```
DECLARE
    inicial CHAR(1) := 'A';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inicial: ' || inicial);
END;
```

- **LONG**: se utiliza para almacenar cadenas de longitud variable.

```
DECLARE
    descripcion LONG := 'Esta es una descripcion larga';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
END;
```

- **RAW**: se utiliza para almacenar datos binarios.

```
DECLARE
    datos RAW(100) := UTL_RAW.CAST_TO_RAW('010101');
BEGIN
    DBMS_OUTPUT.PUT_LINE('Datos: ' || UTL_RAW.CAST_TO_VARCHAR2(datos));
END;
```

- **BLOB**: se utiliza para almacenar datos binarios grandes.

```
DECLARE
    es_mayor BOOLEAN := FALSE;
    edad INTEGER := 18;
BEGIN
    es_mayor := (edad >= 18);
    IF es_mayor THEN
        DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Es menor de edad');
    END IF;
END;
```

- **CLOB**: se utiliza para almacenar cadenas de caracteres grandes.

```
DECLARE
    descripcion CLOB := 'Esta es una descripcion larga';
BEGIN
    DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
END;
```

- **TIMESTAMP**: se utiliza para almacenar fechas y horas con precisión de fracciones de segundo.

```
DECLARE
    fecha_hora TIMESTAMP := SYSTIMESTAMP;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Fecha y hora: ' || TO_CHAR(fecha_hora, 'DD/MM/YYYY HH24:MI:SS.FF'));
```

- **INTERVAL**: se utiliza para almacenar intervalos de tiempo.

```
DECLARE
    duracion INTERVAL DAY TO SECOND := INTERVAL '3' HOUR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Duracion: ' || duracion);
END;
```

- **RECORD**: se utiliza para almacenar un conjunto de valores relacionados.

```
DECLARE
    TYPE tipo_empleado IS RECORD (
        nombre VARCHAR2(50),
        edad NUMBER,
        sueldo NUMBER(10, 2)
    );

    empleado tipo_empleado;
BEGIN
    empleado.nombre := 'Juan';
    empleado.edad := 30;
    empleado.sueldo := 5000.50;

    DBMS_OUTPUT.PUT_LINE('Nombre: ' || empleado.nombre);
    DBMS_OUTPUT.PUT_LINE('Edad: ' || empleado.edad);
    DBMS_OUTPUT.PUT_LINE('Sueldo: ' || empleado.sueldo);
END;
```

- **TABLE**: se utiliza para almacenar conjuntos de datos en forma de tabla.

```
DECLARE
    TYPE tipo_empleados IS TABLE OF VARCHAR2(50);
    empleados tipo_empleados := tipo_empleados('Juan', 'Maria', 'Pedro');
BEGIN
    FOR i IN 1..empleados.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Empleado ' || i || ': ' || empleados(i));
    END LOOP;
END;
```

1.3. Strings

En PL/SQL, los strings se representan utilizando el tipo de dato `VARCHAR2` o `CHAR`. Los strings son utilizados para almacenar y manipular datos textuales. A continuación se presentan algunos métodos y

operaciones comunes para trabajar con strings en PL/SQL:

Declaración y asignación de strings Se pueden declarar variables de tipo string utilizando el tipo de dato `VARCHAR2`. Luego, se pueden asignar valores utilizando el operador de asignación `:=`. Por ejemplo:

```
DECLARE
   nombre VARCHAR2(50);
   direccion VARCHAR2(100);
BEGIN
   nombre := 'Juan';
   direccion := 'Calle Principal';
   -- Resto del codigo...
END;
```

Concatenación de strings La concatenación de strings se puede realizar utilizando el operador de concatenación `||`. Por ejemplo:

```
DECLARE
   nombre VARCHAR2(50) := 'Juan';
   apellido VARCHAR2(50) := 'Perez';
   nombre_completo VARCHAR2(100);
BEGIN
   nombre_completo := nombre || ' ' || apellido;
   dbms_output.put_line('Nombre completo: ' || nombre_completo);
   -- Resto del codigo...
END;
```

Funciones y operadores de manipulación de strings PL/SQL proporciona varias funciones y operadores para manipular strings. Aquí se presentan algunos ejemplos:

- `SUBSTR(cadena, inicio, longitud)`: Retorna una subcadena de una cadena dada. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    subcadena VARCHAR2(20);
BEGIN
    subcadena := SUBSTR(cadena, 1, 5);
    dbms_output.put_line('Subcadena: ' || subcadena);
    -- Resto del codigo...
END;
```

- `LENGTH(cadena)`: Retorna la longitud de una cadena. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    longitud NUMBER;
BEGIN
    longitud := LENGTH(cadena);
    dbms_output.put_line('Longitud: ' || longitud);
    -- Resto del codigo...
END;
```

- `UPPER(cadena)`: Convierte una cadena a mayúsculas. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    cadena_mayusculas VARCHAR2(50);
BEGIN
    cadena_mayusculas := UPPER(cadena);
    dbms_output.put_line('Cadena en mayusculas: ' || cadena_mayusculas);
    -- Resto del codigo...
END;
```

- `LOWER(cadena)`: Convierte una cadena a minúsculas. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    cadena_minusculas VARCHAR2(50);
BEGIN
    cadena_minusculas := LOWER(cadena);
    dbms_output.put_line('Cadena en minusculas: ' || cadena_minusculas);
    -- Resto del codigo...
END;
```

- `INSTR(cadena, subcadena)`: Encuentra la posición de una subcadena dentro de una cadena. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    posicion NUMBER;
BEGIN
    posicion := INSTR(cadena, 'World');
    dbms_output.put_line('Posicion de "World": ' || posicion);
    -- Resto del codigo...
END;
```

- `REPLACE(cadena, subcadena, nueva_subcadena)`: Reemplaza todas las ocurrencias de una subcadena por otra en una cadena. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := 'Hello, World!';
    nueva_cadena VARCHAR2(50);
BEGIN
    nueva_cadena := REPLACE(cadena, 'World', 'Mundo');
    dbms_output.put_line('Cadena modificada: ' || nueva_cadena);
    -- Resto del codigo...
END;
```

- `TRIM(cadena)`, `LTRIM(cadena)`, `RTRIM(cadena)`: Estas funciones se utilizan para eliminar caracteres de una cadena, a la iquierda o derecha respectivamente. Ejemplo:

```
DECLARE
    cadena VARCHAR2(50) := '   Hola, Mundo!   ';
    cadena_sin_espacios VARCHAR2(50);
BEGIN
    cadena_sin_espacios := TRIM(cadena);
    dbms_output.put_line('Cadena sin espacios: ' || cadena_sin_espacios);
    -- Resto del codigo...
END;
```

1.4. Condicionales y bucles

En PL/SQL, los condicionales y los bucles son fundamentales para controlar el flujo de ejecución de un programa. Permiten tomar decisiones y repetir tareas de manera eficiente.

Condicionales

En PL/SQL, se utiliza la estructura `IF-THEN-ELSE` para evaluar una condición y ejecutar diferentes bloques de código según el resultado. A continuación se muestra un ejemplo:

```
DECLARE
   edad NUMBER := 18;
BEGIN
   IF edad >= 18 THEN
      dbms_output.put_line('Eres mayor de edad');
   ELSE
      dbms_output.put_line('Eres menor de edad');
```

```
    END IF;
END;
```

En este ejemplo, se evalúa la variable `edad` y se muestra un mensaje según el resultado.

Condicional CASE

El condicional `CASE` en PL/SQL es una estructura de control que permite evaluar una expresión y tomar decisiones basadas en su valor. Proporciona una alternativa a la estructura `IF-THEN-ELSE` y es especialmente útil cuando se tienen múltiples condiciones a evaluar.

La sintaxis básica del condicional `CASE` es la siguiente:

```
CASE expresion
    WHEN valor1 THEN
        -- Codigo a ejecutar cuando la expresion es igual a valor1
    WHEN valor2 THEN
        -- Codigo a ejecutar cuando la expresion es igual a valor2
    ...
    ELSE
        -- Codigo a ejecutar cuando la expresion no coincide con ninguno de los valores anteriores
END CASE;
```

En este caso, `expresion` es la expresión que se evalúa y `valor1`, `valor2`, etc., son los valores posibles que se comparan con la expresión. El bloque de código correspondiente se ejecutará cuando la expresión coincida con uno de los valores especificados.

A continuación, se muestra un ejemplo de uso del condicional `CASE` en PL/SQL:

```
DECLARE
    nota NUMBER := 80;
BEGIN
    CASE nota
        WHEN 90 THEN
            DBMS_OUTPUT.PUT_LINE('Excelente');
        WHEN 80 THEN
            DBMS_OUTPUT.PUT_LINE('Bueno');
        WHEN 70 THEN
            DBMS_OUTPUT.PUT_LINE('Aceptable');
        ELSE
            DBMS_OUTPUT.PUT_LINE('Reprobado');
    END CASE;
END;
```

En este ejemplo, la variable `nota` se evalúa y se ejecuta el código correspondiente según el valor de la nota. Si la nota es 90, se mostrará .Excelente"; si es 80, se mostrará "Bueno"; si es 70, se mostrará .Aceptable"; y en cualquier otro caso, se mostrará Reprobado".

El condicional `CASE` también permite el uso de condiciones adicionales utilizando la cláusula `WHEN-THEN`:

```
DECLARE
   nota NUMBER := 75;
BEGIN
   CASE
      WHEN nota >= 90 THEN
         DBMS_OUTPUT.PUT_LINE('Excelente');
      WHEN nota >= 80 THEN
         DBMS_OUTPUT.PUT_LINE('Bueno');
      WHEN nota >= 70 THEN
         DBMS_OUTPUT.PUT_LINE('Aceptable');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Reprobado');
   END CASE;
END;
```

En este caso, no se especifica una expresión en el `CASE`, pero se utilizan condiciones para evaluar la variable `nota` y ejecutar el código correspondiente.

Bucles

En PL/SQL, los bucles permiten repetir tareas y controlar el flujo de ejecución. Veamos algunos tipos de bucles comunes:

Bucle LOOP

El bucle `LOOP` es un bucle infinito que se repite hasta que se encuentra una instrucción de salida. Puedes utilizar la instrucción `EXIT` para salir del bucle. Aquí tienes un ejemplo:

```
DECLARE
   contador NUMBER := 1;
BEGIN
   LOOP
      dbms_output.put_line('Contador: ' || contador);
      contador := contador + 1;

      IF contador > 5 THEN
```

```
        EXIT; -- Sale del bucle
      END IF;
   END LOOP;
END;
```

En este ejemplo, se utiliza un bucle `LOOP` para mostrar el valor del contador y se sale del bucle cuando el contador supera 5.

Bucle WHILE

El bucle `WHILE` se repite mientras se cumpla una condición especificada. El bucle sigue ejecutándose siempre que la condición sea verdadera. Aquí tienes un ejemplo:

```
DECLARE
   contador NUMBER := 1;
BEGIN
   WHILE contador <= 5 LOOP
      dbms_output.put_line('Contador: ' || contador);
      contador := contador + 1;
   END LOOP;
END;
```

En este ejemplo, se utiliza un bucle `WHILE` para mostrar el valor del contador y se repite mientras el contador sea menor o igual a 5.

Bucle FOR

El bucle `FOR` se utiliza para recorrer un conjunto de valores en un rango o una lista. Puedes especificar un rango de valores utilizando `INICIO..FIN` o proporcionar una lista separada por comas de valores. Aquí tienes un ejemplo:

```
DECLARE
   total NUMBER := 0;
BEGIN
   FOR i IN 1..5 LOOP
      total := total + i;
   END LOOP;

   dbms_output.put_line('Total: ' || total);
END;
```

En este ejemplo, se utiliza un bucle `FOR` para sumar los números del 1 al 5 y se muestra el resultado final.

La principal diferencia entre los bucles LOOP y WHILE radica en cómo se controla la condición de finalización: en el caso del bucle LOOP, se controla internamente dentro del bucle, mientras que en el bucle WHILE, se evalúa antes de cada iteración. El bucle FOR se utiliza específicamente para iterar sobre colecciones de elementos, como cursores o conjuntos de registros.

Es importante tener cuidado con los bucles que no paran. Si un bucle no tiene una instrucción de salida o una condición de finalización que se cumpla, se producirá lo que se conoce como un "bucle infinito". En ese caso, el bucle se ejecutará continuamente sin detenerse, lo que puede llevar a problemas como un uso excesivo de recursos del sistema y un bloqueo del programa. Los bucles infinitos son un error común en la programación y deben evitarse. Si accidentalmente creas un bucle infinito, es posible que debas detener la ejecución del programa manualmente o reiniciar el entorno de ejecución

En PL/SQL, es posible anidar condicionales y bucles, lo que permite realizar estructuras de control más complejas y flexibles. A continuación, se presentan ejemplos prácticos de condicionales y bucles anidados:

Condicionales anidados

Los condicionales anidados permiten evaluar múltiples condiciones y ejecutar diferentes bloques de código en función de los resultados. Aquí tienes un ejemplo que determina la calificación de un estudiante en base a su nota:

```
DECLARE
   nota NUMBER := 85;
BEGIN
   IF nota >= 90 THEN
      dbms_output.put_line('Calificacion: A');
   ELSIF nota >= 80 THEN
      dbms_output.put_line('Calificacion: B');
   ELSIF nota >= 70 THEN
      dbms_output.put_line('Calificacion: C');
   ELSE
      dbms_output.put_line('Calificacion: D');
   END IF;
   -- Resto del codigo...
END;
/
```

En este ejemplo, se evalúa la nota del estudiante y se imprime la calificación correspondiente utilizando condicionales anidados.

Bucles anidados

Los bucles anidados permiten iterar sobre múltiples conjuntos de datos. Aquí tienes un ejemplo de un bucle `FOR` anidado que genera una tabla de multiplicación:

```
DECLARE
   limite_filas NUMBER := 5;
   limite_columnas NUMBER := 5;
BEGIN
   FOR i IN 1..limite_filas LOOP
      FOR j IN 1..limite_columnas LOOP
         dbms_output.put(i*j || ' ');
      END LOOP;
      dbms_output.new_line;
   END LOOP;
   -- Resto del codigo...
END;
/
```

En este ejemplo, se utiliza un bucle `FOR` anidado para generar una tabla de multiplicación con el límite de filas y columnas especificado.

Los condicionales y bucles anidados brindan flexibilidad para controlar el flujo de ejecución en situaciones más complejas. Puedes combinarlos y ajustarlos según tus necesidades para lograr la lógica deseada en tu programa.

Recuerda que estos son solo ejemplos básicos, y puedes adaptarlos y expandirlos según tus requerimientos específicos.

1.5. Matrices (Arrays)

En PL/SQL, las matrices se pueden representar utilizando tipos de datos de matriz, como `VARRAY` (Variable Array). Una matriz, también conocida como vector, es una estructura de datos unidimensional que puede contener un conjunto de elementos del mismo tipo.

Introducción a las matrices

Para declarar una matriz en PL/SQL, se utiliza la siguiente sintaxis:

```
DECLARE
   TYPE matriz_t IS VARRAY(n) OF tipo_dato;
   matriz matriz_t := matriz_t(elemento1, elemento2, ..., elementon);
BEGIN
   -- Resto del codigo
END;
```

Donde:

- `matriz_t`: es el tipo de dato de la matriz.
- `n`: es el tamaño máximo de la matriz (número de elementos).
- `tipo_dato`: es el tipo de dato de los elementos de la matriz.
- `matriz`: es el nombre de la matriz.
- `elemento1, elemento2, ..., elementon`: son los elementos que se asignarán a la matriz durante la inicialización.

A continuación, se muestra un ejemplo de declaración e inicialización de una matriz de números en PL/SQL:

```
DECLARE
   TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
   numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
BEGIN
   -- Resto del codigo
END;
```

Acceso a una matriz

Los elementos de una matriz se pueden acceder utilizando el nombre de la matriz seguido de un índice entre paréntesis. El índice indica la posición del elemento en la matriz, comenzando desde 1.

A continuación, se muestra un ejemplo de acceso a los elementos de una matriz en PL/SQL:

```
DECLARE
   TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
   numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
   elemento NUMBER;
BEGIN
   elemento := numeros(3); -- Acceso al elemento en la posicion 3
   DBMS_OUTPUT.PUT_LINE('Elemento: ' || elemento);
END;
```

Operaciones con matrices

En PL/SQL, se pueden realizar diversas operaciones con matrices, como actualización de elementos, obtención del tamaño de la matriz, iteración sobre los elementos y concatenación de matrices. A continuación, se presentan algunos ejemplos de estas operaciones:

- **Actualización de elementos de una matriz**:

```
DECLARE
    TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
    numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
BEGIN
    numeros(3) := 10; -- Actualizacion del elemento en la posicion 3
    -- Resto del codigo
END;
```

- **Obtención del tamaño de una matriz**:

```
DECLARE
    TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
    numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
    total_elementos INTEGER;
BEGIN
    total_elementos := numeros.COUNT; -- Obtener la cantidad de elementos en la matriz
    DBMS_OUTPUT.PUT_LINE('Total de elementos: ' || total_elementos);
END;
```

- **Iteración sobre los elementos de una matriz**:

```
DECLARE
    TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
    numeros matriz_numeros := matriz_numeros(1, 2, 3, 4, 5);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Elementos de la matriz:');
    FOR i IN 1..numeros.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Elemento ' || i || ': ' || numeros(i));
    END LOOP;
END;
```

- **Concatenación de matrices**:

```
DECLARE
    TYPE matriz_numeros IS VARRAY(5) OF NUMBER;
```

```
    numeros1 matriz_numeros := matriz_numeros(1, 2, 3);
    numeros2 matriz_numeros := matriz_numeros(4, 5);
    numeros_concat matriz_numeros;
BEGIN
    numeros_concat := numeros1 || numeros2; -- Concatenacion de las matrices numeros1 y numeros2
    -- Resto del codigo
END;
```

Estos ejemplos muestran algunas de las operaciones más comunes que se pueden realizar con matrices en PL/SQL. Puedes adaptar y combinar estas operaciones según tus necesidades específicas.

Recuerda que la sintaxis y las funciones específicas pueden variar dependiendo de la versión de Oracle y de las características soportadas.

1.6. Funciones

Las funciones en PL/SQL son subprogramas que realizan un cálculo o una operación y devuelven un valor. Pueden tener parámetros de entrada y/o de salida, lo que les permite aceptar valores proporcionados al llamar a la función y retornar un resultado al finalizar su ejecución.

Parámetros de entrada

Los parámetros de entrada de una función son utilizados para pasar valores desde el código que llama a la función hacia el interior de la función. Estos valores son utilizados dentro de la función para realizar cálculos u operaciones. Los parámetros de entrada se definen en la declaración de la función y deben especificar su tipo de datos.

A continuación, se muestra un ejemplo de una función con parámetros de entrada:

```
CREATE OR REPLACE FUNCTION calcular_area_circulo(radio IN NUMBER) RETURN NUMBER IS
   area NUMBER;
BEGIN
   area := 3.14159 * radio * radio;
   RETURN area;
END;
/
```

En este ejemplo, la función `calcular_area_circulo` acepta un parámetro de entrada llamado `radio` de tipo `NUMBER`. Dentro de la función, se utiliza este parámetro para calcular el área de un círculo y se devuelve el resultado.

Parámetros de salida

Los parámetros de salida de una función son utilizados para devolver valores desde la función hacia el código que la llama. Estos valores son definidos y asignados dentro de la función y luego son retornados al finalizar su ejecución. Los parámetros de salida se definen en la declaración de la función utilizando la cláusula `RETURN`.

A continuación, se muestra un ejemplo de una función con parámetros de salida:

```
CREATE OR REPLACE FUNCTION obtener_saludo(nombre IN VARCHAR2) RETURN VARCHAR2 IS
   saludo VARCHAR2(50);
BEGIN
   saludo := 'Hola, ' || nombre || '!';
   RETURN saludo;
END;
/
```

En este ejemplo, la función `obtener_saludo` acepta un parámetro de entrada llamado `nombre` de tipo `VARCHAR2` y devuelve un saludo personalizado utilizando ese nombre.

Ejemplos

A continuación, se presentan ejemplos de funciones que combinan parámetros de entrada y salida:

```
CREATE OR REPLACE FUNCTION calcular_promedio(numeros IN matriz_numeros) RETURN NUMBER IS
   suma NUMBER := 0;
   promedio NUMBER;
BEGIN
   FOR i IN 1..numeros.COUNT LOOP
      suma := suma + numeros(i);
   END LOOP;

   promedio := suma / numeros.COUNT;
   RETURN promedio;
END;

CREATE OR REPLACE FUNCTION obtener_persona(id IN NUMBER) RETURN persona IS
   p persona;
BEGIN
   SELECT * INTO p FROM personas WHERE persona_id = id;
   RETURN p;
END;
```

En el primer ejemplo, la función `calcular_promedio` recibe una matriz de números como parámetro de entrada y calcula el promedio de esos números. Devuelve el promedio como resultado.

En el segundo ejemplo, la función `obtener_persona` recibe un ID como parámetro de entrada y busca en una tabla de personas el registro correspondiente a ese ID. Retorna el registro de persona encontrado.