

PL/SQL

Autor:

Kevin Cárdenas.

2023

Índice

| | |
|--|----------|
| 1. Introducción | 2 |
| 1.1. Bloques de PL/SQL | 2 |
| 1.2. Variables y constantes | 4 |
| 1.3. Strings | 7 |
| 1.4. Condicionales y bucles | 10 |
| 1.5. Matrices | 15 |
| 1.5.1. Ejemplo de declaración y uso de matrices: | 15 |
| 1.6. Funciones | 16 |

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';
```

```

5      edad NUMBER := 30;
6 BEGIN
7     -- Logic of program
8     IF edad >= 18 THEN
9         DBMS_OUTPUT.PUT_LINE(nombre || ' es mayor de edad');
10    ELSE
11        DBMS_OUTPUT.PUT_LINE(nombre || ' es menor de edad');
12    END IF;
13 END;
14
15 -- Sentencias SQL
16 INSERT INTO empleados (nombre, salario)
17     VALUES ('Ana', 5000);
18
19 COMMIT;
20 END;
21 /

```

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:

```

1 CREATE OR REPLACE PROCEDURE calcular_salario(p_empleado_id NUMBER) AS
2     v_salario NUMBER;
3 BEGIN
4     -- Logica para calcular el salario del empleado
5     SELECT salario INTO v_salario
6     FROM empleados
7     WHERE empleado_id = p_empleado_id;
8
9     DBMS_OUTPUT.PUT_LINE('El salario del empleado ' || p_empleado_id || ' es: ' || v_salario);
10 END;
11 /

```

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”:

```
1 CREATE OR REPLACE TRIGGER insertar_empleado_trigger
2 AFTER INSERT ON empleados
3 FOR EACH ROW
4 BEGIN
5     DBMS_OUTPUT.PUT_LINE('Nuevo empleado insertado: ' || :NEW.nombre);
6 END;
7 /
```

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.

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

```

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

```

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

```

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

```

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

```

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

```

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

```

1 DECLARE
2     es_mayor BOOLEAN := TRUE;
3 BEGIN
4     IF es_mayor THEN
5         DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
6     ELSE
7         DBMS_OUTPUT.PUT_LINE('Es menor de edad');
8     END IF;
9 END;

```

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

```

1 DECLARE
2     inicial CHAR(1) := 'A';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Inicial: ' || inicial);
5 END;

```

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

```

1 DECLARE
2     descripcion LONG := 'Esta es una descripcion larga';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
5 END;

```

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

```

1 DECLARE
2     datos RAW(100) := UTL_RAW.CAST_TO_RAW('010101');
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Datos: ' || UTL_RAW.CAST_TO_VARCHAR2(datos));
5 END;

```

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

```

1 DECLARE
2     es_mayor BOOLEAN := FALSE;
3     edad INTEGER := 18;
4 BEGIN
5     es_mayor := (edad >= 18);
6     IF es_mayor THEN
7         DBMS_OUTPUT.PUT_LINE('Es mayor de edad');
8     ELSE
9         DBMS_OUTPUT.PUT_LINE('Es menor de edad');
10    END IF;
11 END;

```

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

```

1 DECLARE
2     descripcion CLOB := 'Esta es una descripcion larga';
3 BEGIN
4     DBMS_OUTPUT.PUT_LINE('Descripcion: ' || descripcion);
5 END;

```

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

```

1 DECLARE
2     fecha_hora TIMESTAMP := SYSTIMESTAMP;
3 BEGIN
4     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.

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

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

```
1 DECLARE
2     TYPE tipo_empleado IS RECORD (
3         nombre VARCHAR2(50),
4         edad  NUMBER,
5         sueldo NUMBER(10, 2)
6     );
7
8     empleado tipo_empleado;
9 BEGIN
10    empleado.nombre := 'Juan';
11    empleado.edad := 30;
12    empleado.sueldo := 5000.50;
13
14    DBMS_OUTPUT.PUT_LINE('Nombre: ' || empleado.nombre);
15    DBMS_OUTPUT.PUT_LINE('Edad: ' || empleado.edad);
16    DBMS_OUTPUT.PUT_LINE('Sueldo: ' || empleado.sueldo);
17 END;
```

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

```
1 DECLARE
2     TYPE tipo_empleados IS TABLE OF VARCHAR2(50);
3     empleados tipo_empleados := tipo_empleados('Juan', 'Maria', 'Pedro');
4 BEGIN
5     FOR i IN 1..empleados.COUNT LOOP
6         DBMS_OUTPUT.PUT_LINE('Empleado ' || i || ': ' || empleados(i));
7     END LOOP;
8 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:

```
1 DECLARE
2     nombre VARCHAR2(50);
3     direccion VARCHAR2(100);
4 BEGIN
5     nombre := 'Juan';
6     direccion := 'Calle Principal';
7     -- Resto del codigo...
8 END;
```

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

```
1 DECLARE
2     nombre VARCHAR2(50) := 'Juan';
3     apellido VARCHAR2(50) := 'Perez';
4     nombre_completo VARCHAR2(100);
5 BEGIN
6     nombre_completo := nombre || ' ' || apellido;
7     dbms_output.put_line('Nombre completo: ' || nombre_completo);
8     -- Resto del codigo...
9 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:

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

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

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     longitud NUMBER;
4 BEGIN
5     longitud := LENGTH(cadena);
6     dbms_output.put_line('Longitud: ' || longitud);
7     -- Resto del codigo...
8 END;
```

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

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     cadena_mayusculas VARCHAR2(50);
4 BEGIN
5     cadena_mayusculas := UPPER(cadena);
6     dbms_output.put_line('Cadena en mayusculas: ' || cadena_mayusculas);
7     -- Resto del codigo...
8 END;
```

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

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     cadena_minusculas VARCHAR2(50);
4 BEGIN
5     cadena_minusculas := LOWER(cadena);
6     dbms_output.put_line('Cadena en minusculas: ' || cadena_minusculas);
7     -- Resto del codigo...
8 END;
```

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

```
1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     posicion NUMBER;
4 BEGIN
5     posicion := INSTR(cadena, 'World');
6     dbms_output.put_line('Posicion de "World": ' || posicion);
7     -- Resto del codigo...
8 END;
```

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

```

1 DECLARE
2     cadena VARCHAR2(50) := 'Hello, World!';
3     nueva_cadena VARCHAR2(50);
4 BEGIN
5     nueva_cadena := REPLACE(cadena, 'World', 'Mundo');
6     dbms_output.put_line('Cadena modificada: ' || nueva_cadena);
7     -- Resto del codigo...
8 END;
```

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

```

1 DECLARE
2     cadena VARCHAR2(50) := '  Hola, Mundo!  ';
3     cadena_sin_espacios VARCHAR2(50);
4 BEGIN
5     cadena_sin_espacios := TRIM(cadena);
6     dbms_output.put_line('Cadena sin espacios: ' || cadena_sin_espacios);
7     -- Resto del codigo...
8 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:

```

1 DECLARE
2     edad NUMBER := 18;
3 BEGIN
4     IF edad >= 18 THEN
5         dbms_output.put_line('Eres mayor de edad');
6     ELSE
7         dbms_output.put_line('Eres menor de edad');
```

```

8  END IF;
9  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:

```

1  CASE expression
2      WHEN valor1 THEN
3          -- Codigo a ejecutar cuando la expresion es igual a valor1
4      WHEN valor2 THEN
5          -- Codigo a ejecutar cuando la expresion es igual a valor2
6          ...
7      ELSE
8          -- Codigo a ejecutar cuando la expresion no coincide con ninguno de los valores anteriores
9  END CASE;

```

En este caso, `expression` 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:

```

1  DECLARE
2      nota NUMBER := 80;
3  BEGIN
4      CASE nota
5          WHEN 90 THEN
6              DBMS_OUTPUT.PUT_LINE('Excelente');
7          WHEN 80 THEN
8              DBMS_OUTPUT.PUT_LINE('Bueno');
9          WHEN 70 THEN
10                 DBMS_OUTPUT.PUT_LINE('Aceptable');
11         ELSE
12             DBMS_OUTPUT.PUT_LINE('Reprobado');
13     END CASE;
14 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`:

```
1 DECLARE
2     nota NUMBER := 75;
3 BEGIN
4     CASE
5         WHEN nota >= 90 THEN
6             DBMS_OUTPUT.PUT_LINE('Excelente');
7         WHEN nota >= 80 THEN
8             DBMS_OUTPUT.PUT_LINE('Bueno');
9         WHEN nota >= 70 THEN
10            DBMS_OUTPUT.PUT_LINE('Aceptable');
11        ELSE
12            DBMS_OUTPUT.PUT_LINE('Reprobado');
13    END CASE;
14 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:

```
1 DECLARE
2     contador NUMBER := 1;
3 BEGIN
4     LOOP
5         dbms_output.put_line('Contador: ' || contador);
6         contador := contador + 1;
7
8         IF contador > 5 THEN
```

```

9      EXIT; -- Sale del bucle
10
11     END IF;
12 END LOOP;
13 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:

```

1 DECLARE
2     contador NUMBER := 1;
3 BEGIN
4     WHILE contador <= 5 LOOP
5         dbms_output.put_line('Contador: ' || contador);
6         contador := contador + 1;
7     END LOOP;
8 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:

```

1 DECLARE
2     total NUMBER := 0;
3 BEGIN
4     FOR i IN 1..5 LOOP
5         total := total + i;
6     END LOOP;
7
8     dbms_output.put_line('Total: ' || total);
9 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:

```
1 DECLARE
2     nota NUMBER := 85;
3 BEGIN
4     IF nota >= 90 THEN
5         dbms_output.put_line('Calificacion: A');
6     ELSIF nota >= 80 THEN
7         dbms_output.put_line('Calificacion: B');
8     ELSIF nota >= 70 THEN
9         dbms_output.put_line('Calificacion: C');
10    ELSE
11        dbms_output.put_line('Calificacion: D');
12    END IF;
13    -- Resto del codigo...
14 END;
15 /
```

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:

```
1 DECLARE
2     limite_filas NUMBER := 5;
3     limite_columnas NUMBER := 5;
4 BEGIN
5     FOR i IN 1..limite_filas LOOP
6         FOR j IN 1..limite_columnas LOOP
7             dbms_output.put(i*j || ' ');
8         END LOOP;
9         dbms_output.new_line;
10    END LOOP;
11    -- Resto del codigo...
12 END;
13 /
```

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

Las matrices o arrays son estructuras de datos utilizadas para almacenar múltiples valores del mismo tipo en una sola variable. En PL/SQL, se pueden declarar y manipular matrices para procesar conjuntos de datos de manera eficiente. Las matrices se acceden mediante índices y se pueden utilizar en bucles y operaciones aritméticas.

1.5.1. Ejemplo de declaración y uso de matrices:


```

1 DECLARE
2     TYPE numeros_array IS VARRAY(5) OF NUMBER;
3     numeros numeros_array := numeros_array(1, 2, 3, 4, 5);
4 BEGIN
5     FOR i IN 1..numeros.COUNT LOOP
6         DBMS_OUTPUT.PUT_LINE('Numero en la posicion ' || i || ': ' || numeros(i));
7     END LOOP;
8 END;

```

1.6. Funciones