**Variables**

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows defining various types of variables, such as date time data types, records, collections, etc.

**Variable Declaration in PL/SQL**

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

Some valid variable declarations along with their definition are shown below −

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

address varchar2(100);

When you provide a size, scale or precision limit with the data type, it is called a constrained declaration. Constrained declarations require less memory than unconstrained declarations. For example −

sales number(10, 2);

name varchar2(25);

address varchar2(100);

## Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

* The DEFAULT keyword
* The assignment operator

For example −

counter binary\_integer := 0;

greetings varchar2(20) DEFAULT 'Have a Good Day';

You can also specify that a variable should not have a NULL value using the NOT NULL constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results.

# PL/SQL - Functions

A function is same as a procedure except that it returns a value.

## Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

CREATE [OR REPLACE] FUNCTION function\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

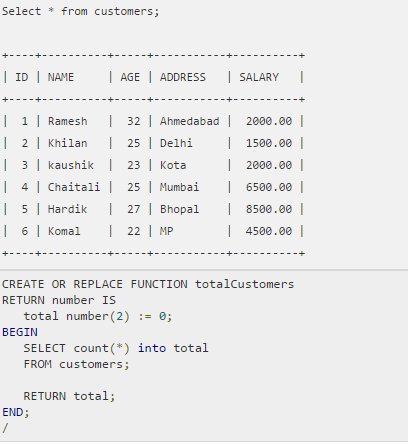
Where,

* *function-name* specifies the name of the function.
* [OR REPLACE] option allows the modification of an existing function.
* The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* The function must contain a **return** statement.
* The *RETURN* clause specifies the data type you are going to return from the function.
* *function-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone function.

**Example**

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the [PL/SQL](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm).



**Calling a Function**

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function totalCustomers from an anonymous block −

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('Total no. of Customers: ' || c);

END;

/

### Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

CREATE FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

DECLARE

a number;

b number;

c number;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

# PL/SQL - Procedures

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

* At the schema level
* Inside a package
* Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

* **Functions** − These subprograms return a single value; mainly used to compute and return a value.
* **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

## Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

Where,

* *procedure-name* specifies the name of the procedure.
* [OR REPLACE] option allows the modification of an existing procedure.
* The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
* *procedure-body* contains the executable part.
* The AS keyword is used instead of the IS keyword for creating a standalone procedure.

**Example**

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms\_output.put\_line('Hello World!');

END;

/

**Executing a Standalone Procedure**

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

# PL/SQL - Conditions

|  |  |
| --- | --- |
| **S.No** | **Statement & Description** |
| 1 | [**IF - THEN statement**](https://www.tutorialspoint.com/plsql/plsql_if_then.htm)  The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.  IF condition THEN  S;  END IF;  Example 1  DECLARE  a number(2) := 10;  BEGIN  a:= 10;  -- check the boolean condition using if statement  IF( a < 20 ) THEN  -- if condition is true then print the following  dbms\_output.put\_line('a is less than 20 ' );  END IF;  dbms\_output.put\_line('value of a is : ' || a);  END;  Example 2  DECLARE  a number(2) := 10;  BEGIN  a:= 10;  -- check the boolean condition using if statement  IF( a < 20 ) THEN  -- if condition is true then print the following  dbms\_output.put\_line('a is less than 20 ' );  END IF;  dbms\_output.put\_line('value of a is : ' || a);  END; |
| 2 | [**IF-THEN-ELSE statement**](https://www.tutorialspoint.com/plsql/plsql_if_then_else.htm)  **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.  IF condition THEN  S1;  ELSE  S2;  END IF;  IF color = red THEN  dbms\_output.put\_line('You have chosen a red car')  ELSE  dbms\_output.put\_line('Please choose a color for your car');  END IF; |
| 3 | [**IF-THEN-ELSIF statement**](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)  It allows you to choose between several alternatives. |
| 4 | [**Case statement**](https://www.tutorialspoint.com/plsql/plsql_case_statement.htm)  Like the IF statement, the **CASE statement** selects one sequence of statements to execute.  However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.  CASE selector  WHEN 'value1' THEN S1;  WHEN 'value2' THEN S2;  WHEN 'value3' THEN S3;  ...  ELSE Sn; -- default case  END CASE;  DECLARE  grade char(1) := 'A';  BEGIN  CASE grade  when 'A' then dbms\_output.put\_line('Excellent');  when 'B' then dbms\_output.put\_line('Very good');  when 'C' then dbms\_output.put\_line('Well done');  when 'D' then dbms\_output.put\_line('You passed');  when 'F' then dbms\_output.put\_line('Better try again');  else dbms\_output.put\_line('No such grade');  END CASE;  END; |
| 5 | [**Searched CASE statement**](https://www.tutorialspoint.com/plsql/plsql_searched_case.htm)  The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.  CASE  WHEN selector = 'value1' THEN S1;  WHEN selector = 'value2' THEN S2;  WHEN selector = 'value3' THEN S3;  ...  ELSE Sn; -- default case  END CASE;  DECLARE  grade char(1) := 'B';  BEGIN  case  when grade = 'A' then dbms\_output.put\_line('Excellent');  when grade = 'B' then dbms\_output.put\_line('Very good');  when grade = 'C' then dbms\_output.put\_line('Well done');  when grade = 'D' then dbms\_output.put\_line('You passed');  when grade = 'F' then dbms\_output.put\_line('Better try again');  else dbms\_output.put\_line('No such grade');  end case;  END; |
| 6 | [**nested IF-THEN-ELSE**](https://www.tutorialspoint.com/plsql/plsql_nested_if.htm)  You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

# PL/SQL – Loops

|  |  |
| --- | --- |
| **S.No** | **Loop Type & Description** |
| 1 | [**PL/SQL Basic LOOP**](https://www.tutorialspoint.com/plsql/plsql_basic_loop.htm)  In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.  LOOP  Sequence of statements;  END LOOP;  DECLARE  x number := 10;  BEGIN  LOOP  dbms\_output.put\_line(x);  x := x + 10;  IF x > 50 THEN  exit;  END IF;  END LOOP;  -- after exit, control resumes here  dbms\_output.put\_line('After Exit x is: ' || x);  END; |
| 2 | [**PL/SQL WHILE LOOP**](https://www.tutorialspoint.com/plsql/plsql_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.  WHILE condition LOOP  sequence\_of\_statements  END LOOP;  DECLARE  a number(2) := 10;  BEGIN  WHILE a < 20 LOOP  dbms\_output.put\_line('value of a: ' || a);  a := a + 1;  END LOOP;  END; |
| 3 | [**PL/SQL FOR LOOP**](https://www.tutorialspoint.com/plsql/plsql_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.  FOR counter IN initial\_value .. final\_value LOOP  sequence\_of\_statements;  END LOOP;  DECLARE  a number(2);  BEGIN  FOR a in 10 .. 20 LOOP  dbms\_output.put\_line('value of a: ' || a);  END LOOP;  END;  Reverse FOR LOOP Statement  DECLARE  a number(2) ;  BEGIN  FOR a IN REVERSE 10 .. 20 LOOP  dbms\_output.put\_line('value of a: ' || a);  END LOOP;  END; |
| 4 | [**Nested loops in PL/SQL**](https://www.tutorialspoint.com/plsql/plsql_nested_loops.htm)  You can use one or more loop inside any another basic loop, while, or for loop. |

**Labeling a PL/SQL Loop**

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

DECLARE

i number(1);

j number(1);

BEGIN

<< outer\_loop >>

FOR i IN 1..3 LOOP

<< inner\_loop >>

FOR j IN 1..3 LOOP

dbms\_output.put\_line('i is: '|| i || ' and j is: ' || j);

END loop inner\_loop;

END loop outer\_loop;

END;