**Introduction**

* **PL/SQL** stands for **Procedural Language** extension of SQL.
* PL/SQL is a combination of SQL along with the procedural features of programming languages.
* PL/SQL is a block structured language. The programs of PL/SQL are logical blocks that can contain any number of nested sub-blocks
* It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.
* PL/SQL is a completely portable, high-performance transaction-processing language.
* PL/SQL provides a built-in, interpreted and OS independent programming environment.
* PL/SQL can also directly be called from the command-line **SQL\*Plus interface**.
* Direct call can also be made from external programming language calls to database.
* PL/SQL's general syntax is based on that of ADA and Pascal programming language.
* Apart from Oracle, PL/SQL is available in **Times Ten in-memory database** and **IBM DB2**.

**Features of PL/SQL**

PL/SQL has the following features −

* PL/SQL is tightly integrated with SQL.
* It offers extensive error checking.
* It offers numerous data types.
* It offers a variety of programming structures.
* It supports structured programming through functions and procedures.
* It supports object-oriented programming.
* It supports the development of web applications and server pages.

**Advantages of PL/SQL**

PL/SQL has the following advantages −

* SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
* PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
* PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
* PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
* Applications written in PL/SQL are fully portable.
* PL/SQL provides high security level.
* PL/SQL provides access to predefined SQL packages.
* PL/SQL provides support for Object-Oriented Programming.
* PL/SQL provides support for developing Web Applications and Server Pages.

**PL/SQL Block consists of three sections:**

* The Declaration section (optional).
* The Execution section (mandatory).
* The Exception Handling (or Error) section (optional).

|  |
| --- |
| **Declarations**  This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| **Executable Commands**  This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed. |
| **Exception Handling**  This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program. |

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block –

|  |
| --- |
| DECLARE  <declarations section>  BEGIN  <executable command(s)>  EXCEPTION  <exception handling>  END; |

**The PL/SQL Identifiers**

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

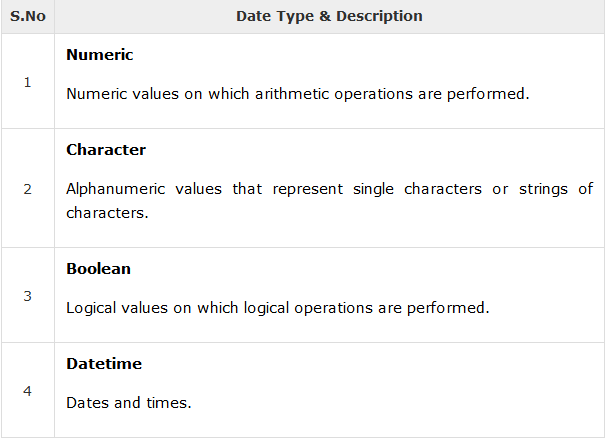
By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

# PL/SQL - Data Types

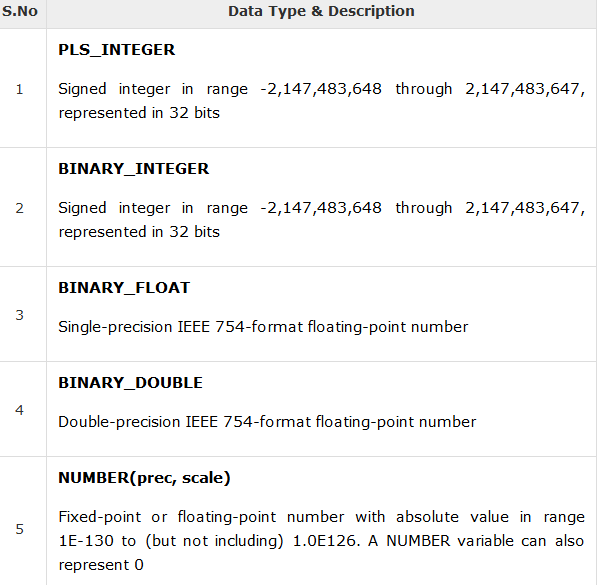
The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter.

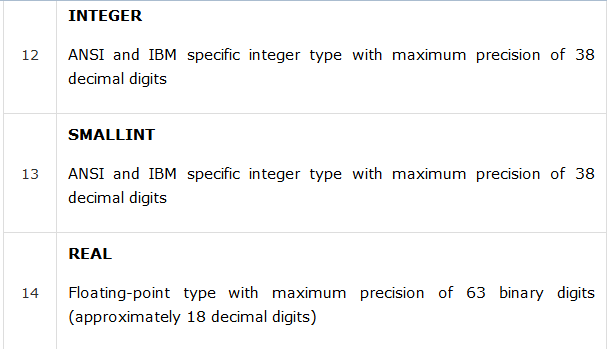
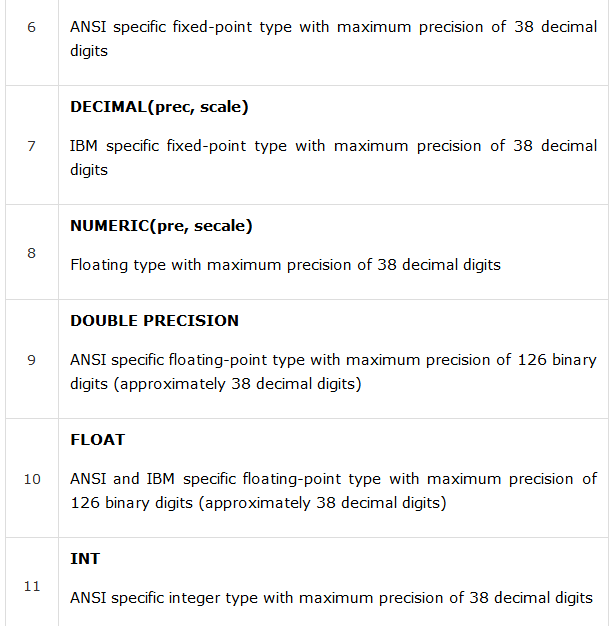
|  |
| --- |
| **Large Object (LOB)**  Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |
| **Composite**  Data items that have internal components that can be accessed individually. For example, collections and records. |
| **Reference**  Pointers to other data items. |

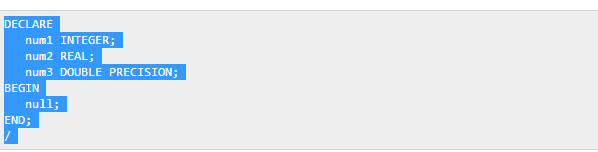
## PL/SQL Scalar Data Types and Subtypes

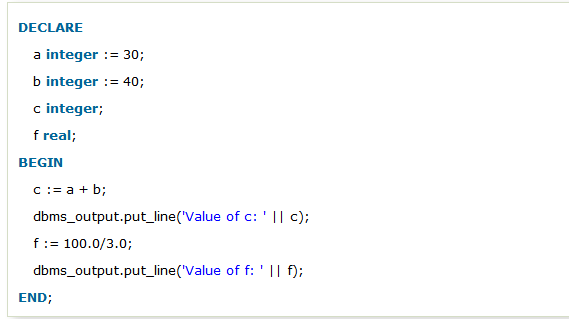


## PL/SQL Numeric Data Types and Subtypes









**Variable Scope in PL/SQL**

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

* **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
* **Global variables** − Variables declared in the outermost block or a package.

DECLARE

-- Global variables

num1 number := 95;

num2 number := 85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: ' || num1);

dbms\_output.put\_line('Outer Variable num2: ' || num2);

DECLARE

-- Local variables

num1 number := 195;

num2 number := 185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: ' || num1);

dbms\_output.put\_line('Inner Variable num2: ' || num2);

END;

END;

## Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

Example:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25),

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

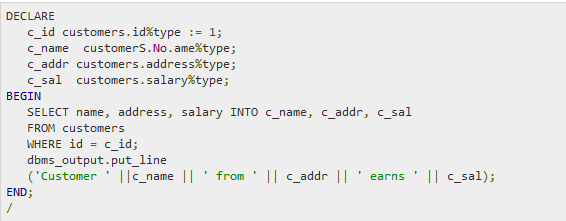
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, 'Komal', 22, 'MP', 4500.00 );

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL –



## 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

counter binary\_integer := 0;

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

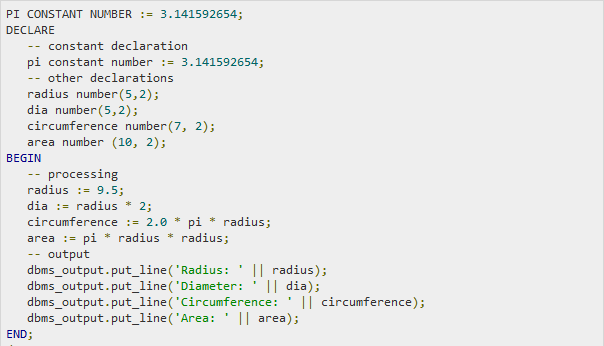
You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

# PL/SQL - Constants and Literals

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the **NOT NULL constraint**.

## Declaring a Constant

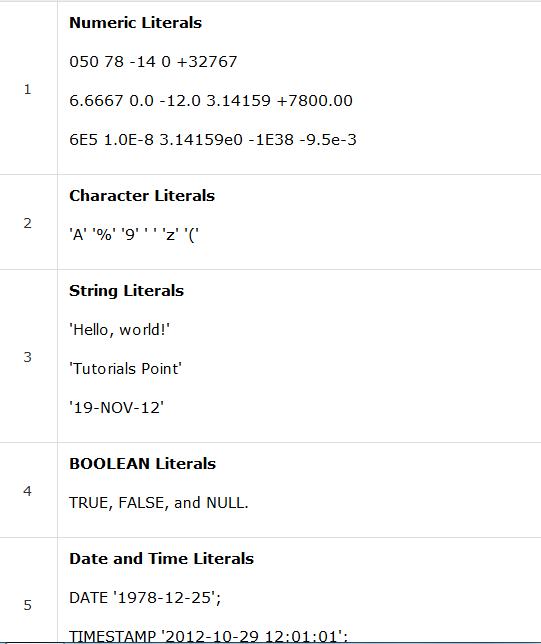
A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example –

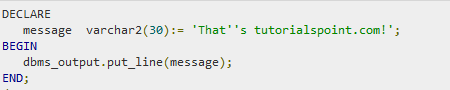


## The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals – TYPES OF LITERALS

* Numeric Literals
* Character Literals
* String Literals
* BOOLEAN Literals
* Date and Time Literals





**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 CONTRAIND 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];

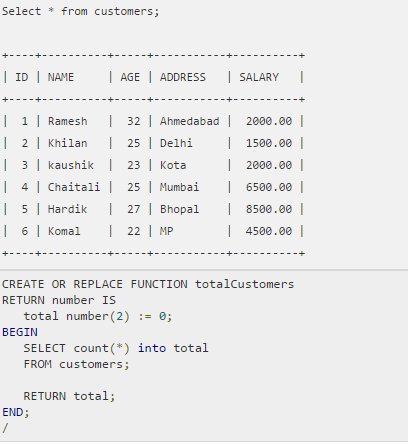
Where,

* *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 –

IMPORTANT DIAGRAM

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;