



# PL SQL Basics

---



PL/SQL is a combination of SQL along with the procedural features of programming languages.



It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.



PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java

# Introduction

# Features of PL/SQL PL/SQL has the following

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 page

# Why PL-SQL?

SQL is a query language, it is not a typical procedural programming language like C or C++. So, it does not have facilities like functions, loops, if-else, etc..

So, in order to have those benefits of constructs like looping, decision making, using if-else or using functions (reusable blocks of code) in SQL; a procedural language was created, and that is PL/SQL

- 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.

Basic Syntax of PL/SQL which is a **block-structured** language.

This means that the PL/SQL programs are divided and written in logical blocks of code.

Each block consists of three sub-parts –



S.No	Sections & Description
1	<b>Declarations</b> This section starts with the keyword <b>DECLARE</b> . It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	<b>Executable Commands</b> This section is enclosed between the keywords <b>BEGIN</b> and <b>END</b> 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 <b>NULL command</b> to indicate that nothing should be executed.
3	<b>Exception Handling</b> This section starts with the keyword <b>EXCEPTION</b> . This optional section contains <b>exception(s)</b> 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;
```

SET SERVEROUTPUT ON

To print the output of the query execution

## The 'Hello World' Example

```
DECLARE message varchar2(20) := 'Hello, World!';  
BEGIN dbms_output.put_line(message);  
END; /
```



```
Hello World  
PL/SQL procedure successfully completed.
```

The end; line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result – Hello World PL/SQL procedure successfully completed.

## The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

```
DECLARE
  -- variable declaration
  message varchar2(20) := 'Hello, World!';
BEGIN
  /*
   * PL/SQL executable statement(s)
   */
  dbms_output.put_line(message);
END;
/
```

Single line comment

Multi line comment

# PL/SQL – Common Data Types

PL/SQL Scalar Data Types and Subtypes

PL/SQL Numeric Data Types and Subtypes

PL/SQL Character Data Types and Subtypes

PL/SQL Boolean Data Types

PL/SQL Datetime and Interval Types

PL/SQL Large Object (LOB) Data Types

PL/SQL User-Defined Subtypes



Delimiter	Description
<b>+, -, *, /</b>	Addition, subtraction/negation, multiplication, division
<b>%</b>	Attribute indicator
<b>'</b>	Character string delimiter
<b>.</b>	Component selector
<b>(,)</b>	Expression or list delimiter
<b>:</b>	Host variable indicator
<b>,</b>	Item separator
<b>"</b>	Quoted identifier delimiter
<b>=</b>	Relational operator
<b>@</b>	Remote access indicator
<b>;</b>	Statement terminator
<b>:=</b>	Assignment operator
<b>=&gt;</b>	Association operator
<b>  </b>	Concatenation operator
<b>**</b>	Exponentiation operator
<b>&lt;&lt;, &gt;&gt;</b>	Label delimiter (begin and end)
<b>/*, */</b>	Multi-line comment delimiter (begin and end)
<b>--</b>	Single-line comment indicator
<b>..</b>	Range operator
<b>&lt;, &gt;, &lt;=, &gt;=</b>	Relational operators
<b>&lt;&gt;, !=, ~=, ^=</b>	Different versions of NOT EQUAL

## The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL –

# 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 datatype [NOT NULL] [:= initial_value];
```

- First, specify the name of the variable. The name of the variable should be as descriptive as possible, e.g.
- Second, choose an appropriate [data type](#) for the variable, depending on the kind of value which you want to store, for example, number, character, Boolean, and datetime.

By convention, local variable names should start with `l_` and global variable names should have a prefix of `g_`.

## 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. Try the following example which makes use of various types of variables –

## Default values

PL/SQL allows you to set a default value for a variable at the declaration time. To assign a default value to a variable, you use the assignment operator ( `:=` ) or the `DEFAULT` keyword.

The following example declares a variable named `l_product_name` with an initial value `'Laptop'` :

```
DECLARE
    l_product_name VARCHAR2( 100 ) := 'Laptop';
BEGIN
    NULL;
END;
```

It is equivalent to the following block:

```
DECLARE
    l_product_name VARCHAR2(100) DEFAULT 'Laptop';
BEGIN
    NULL;
END;
```

In this example, instead of using the assignment operator `:=` , we used the `DEFAULT` keyword to initialize a variable.

```
DECLARE
  a integer := 10;
  b integer := 20;
  c integer;
  f real;
BEGIN
  c := a + b;
  dbms_output.put_line('Value of c: ' || c);
  f := 70.0/3.0;
  dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result –

```
Value of c: 30
Value of f: 23.333333333333333333
```

PL/SQL procedure successfully completed.

## 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.

Following example shows the usage of **Local** and **Global** variables in its simple form –

```
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 –

(For SQL statements, please refer to the SQL tutorial)

```
CREATE TABLE CUSTOMERS(  
    ID      Number NOT NULL,  
    NAME    VARCHAR (20) NOT NULL,  
    AGE     Number NOT NULL,  
    ADDRESS CHAR (25),  
    SALARY  Number (18, 2),  
    PRIMARY KEY (ID)  
);
```

Let us now insert some values in the table –

```
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 –

```
DECLARE
  c_id customers.id%type := 1;
  c_name customers.name%type;
  c_addr customers.address%type;
  c_sal customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal
  FROM customers
  WHERE id = c_id;
  dbms_output.put_line
  ('Customer ' || c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

The **%TYPE** attribute lets use the datatype of a field, record, nested table, database column, or variable in your own declarations, rather than hardcoding the **type** names.

When the above code is executed, it produces the following result –

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully



## 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 –

```
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
  dbms_output.put_line('Diameter: ' || dia);
  dbms_output.put_line('Circumference: ' || circumference);
  dbms_output.put_line('Area: ' || area);
END;
/
```

# PL/SQL - Operators

## Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then –

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

## Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then –

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

## Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE**, **FALSE** or **NULL**.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that $x \geq a$ and $x \leq b$ .	If $x = 10$ then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If $x = 'm'$ then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If $x = 'm'$ , then 'x is null' returns Boolean false.

## Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds false, then –

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.



# PL/SQL - Conditions

## Syntax (IF-THEN-ELSE)

The syntax for IF-THEN-ELSE in Oracle/PLSQL is:

```
IF condition THEN
    {...statements to execute when condition is TRUE...}

ELSE
    {...statements to execute when condition is FALSE...}

END IF;
```

## Syntax (IF-THEN-ELSIF)

The syntax for IF-THEN-ELSIF in Oracle/PLSQL is:

```
IF condition1 THEN
    {...statements to execute when condition1 is TRUE...}

ELSIF condition2 THEN
    {...statements to execute when condition1 is FALSE and condition2 is TRUE...}

END IF;
```

```

DECLARE n_sales NUMBER := 2000000;
BEGIN
    IF n_sales > 100000 THEN
        DBMS_OUTPUT.PUT_LINE( 'Sales revenue is greater than 100K ' );
    END IF;
END;

```

```

IF condition THEN
    statements;
ELSE
    else_statements;
END IF;

```

```

DECLARE
    n_sales NUMBER := 300000;
    n_commission NUMBER( 10, 2 ) := 0;
BEGIN
    IF n_sales > 200000 THEN
        n_commission := n_sales * 0.1;
    ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
        n_commission := n_sales * 0.05;
    ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
        n_commission := n_sales * 0.03;
    ELSE
        n_commission := n_sales * 0.02;
    END IF;
END;

```

## PL/SQL - Loops

S.No	Loop Type & Description
1	<p>PL/SQL Basic LOOP <a href="#">↗</a></p> <p>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.</p>
2	<p>PL/SQL WHILE LOOP <a href="#">↗</a></p> <p>Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.</p>
3	<p>PL/SQL FOR LOOP <a href="#">↗</a></p> <p>Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.</p>
4	<p>Nested loops in PL/SQL <a href="#">↗</a></p> <p>You can use one or more loop inside any another basic loop, while, or for loop.</p>

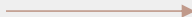


## Labeling a PL/SQL Loop

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.

The following program illustrates the concept –

```
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;
/
```



```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3
```

PL/SQL procedure successfully completed.

Let's look at a WHILE LOOP example in Oracle:

```
WHILE monthly_value <= 4000
LOOP
    monthly_value := daily_value * 31;
END LOOP;
```

In this WHILE LOOP example, the loop would terminate once the monthly\_value exceeded 4000 as specified by:

```
WHILE monthly_value <= 4000
```

Let's look at a LOOP example in Oracle:

```
LOOP
    monthly_value := daily_value * 31;
    EXIT WHEN monthly_value > 4000;
END LOOP;
```

In this LOOP example, the LOOP would terminate when the monthly\_value exceeded 4000.

- You would use a LOOP statement when you are unsure of how many times you want the loop body to execute.
- You can terminate a LOOP statement with either an EXIT statement or when it encounters an EXIT WHEN statement that evaluates to TRUE.



# Declaring String Variables

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are '**national character set**' datatypes, that store Unicode character data.

In Oracle, CLOB data type stores variable-length character data (**character large object**) in the database character set that can be single-byte or multibyte (supports more than 4 GB).

If you need to declare a variable-length string, you must provide the maximum length of that string.

For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables –

```
DECLARE name varchar2(20);  
company varchar2(30);  
introduction clob;  
choice char(1);
```

Oracle:

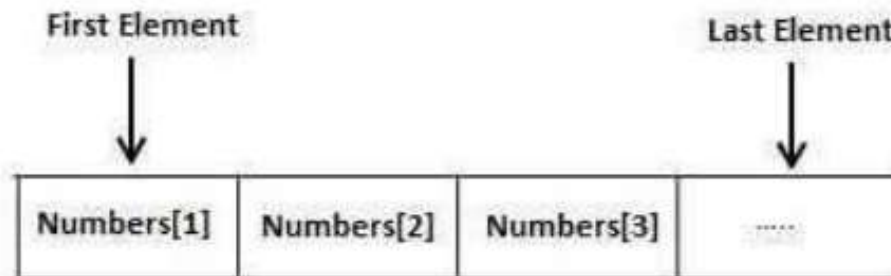
```
-- Create a table with CLOB column  
CREATE TABLE countries  
(  
    name VARCHAR2(90),  
    notes CLOB  
);  
  
-- Insert some data  
INSERT INTO countries VALUES ('Greece', 'Greece is a country in south-east Europe. Athens is the capital...');  
# 1 row created.
```

## PL/SQL - Arrays

The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type.

A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



The basic syntax for creating a **VARRAY** type at the schema level is

– `CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of` Where,

- `varray_type_name` is a valid attribute name,
- `n` is the number of elements (maximum) in the varray,
- `element_type` is the data type of the elements of the array

The basic syntax for creating a **VARRAY** type within a PL/SQL block is –

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example –

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);  
Type grades IS VARRAY(5) OF INTEGER;
```

Let us now work out on a few examples to understand the concept

In Oracle environment, the starting index for varrays is always 1.

### Example 1

The following program illustrates the use of varrays –

```
DECLARE
    type namesarray IS VARRAY(5) OF VARCHAR2(10);
    type grades IS VARRAY(5) OF INTEGER;
    names namesarray;
    marks grades;
    total integer;
BEGIN
    names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
    marks:= grades(98, 97, 78, 87, 92);
    total := names.count;
    dbms_output.put_line('Total ' || total || ' Students');
    FOR i in 1 .. total LOOP
        dbms_output.put_line('Student: ' || names(i) || '
        Marks: ' || marks(i));
    END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Total 5 Students
Student: Kavita Marks: 98
Student: Pritam Marks: 97
Student: Ayan Marks: 78
Student: Rishav Marks: 87
Student: Aziz Marks: 92
```