**PL/SQL Notes**

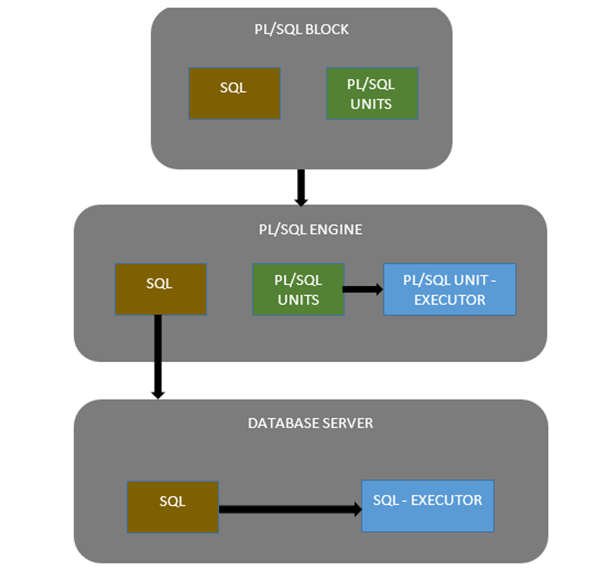
**What is PL/SQL?**

**Oracle PL/SQL** is an extension of SQL language that combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries. PL/SQL ensures seamless processing of SQL statements by enhancing the security, portability, and robustness of the Database.

PL/SQL means instructing the compiler ‘what to do’ through SQL and ‘how to do’ through its procedural way. Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object-oriented concepts. The PL/SQL Full form is “Procedural Language extensions to SQL”.

## Architecture of PL/SQL

The Below PL/SQL Example is a pictorial representation of PL/SQL Architecture.



PL/SQL Architecture Diagram

The PL/SQL architecture mainly consists of following three components:

1. PL/SQL Block
2. PL/SQL Engine
3. Database Server

### PL/SQL block:

* This is the component which has the actual PL/SQL code.
* This consists of different sections to divide the code logically (declarative section for declaring purpose, execution section for processing statements, exception handling section for handling errors)
* It also contains the SQL instruction that used to interact with the database server.
* All the PL/SQL units are treated as [PL/SQL blocks](https://www.guru99.com/blocks-pl-sql.html), and this is the starting stage of the architecture which serves as the primary input.

Following are the different type of PL/SQL units.

12.8MSeven Testing Principles Software Testing

* Anonymous Block
* Function
* Library
* Procedure
* Package Body
* Package Specification
* Trigger
* Type
* Type Body

### PL/SQL Engine

* PL/SQL engine is the component where the actual processing of the codes takes place.
* PL/SQL engine separates PL/SQL units and SQL part in the input (as shown in the image below).
* The separated PL/SQL units will be handled by the PL/SQL engine itself.
* The SQL part will be sent to database server where the actual interaction with database takes place.
* It can be installed in both database server and in the application server.

### Database Server:

* This is the most important component of Pl/SQL unit which stores the data.
* The PL/SQL engine uses the SQL from PL/SQL units to interact with the database server.
* It consists of SQL executor which parses the input SQL statements and execute the same.

## Features & Advantages of PL/SQL

1. Better performance, as SQL is executed in bulk rather than a single statement
2. High Productivity
3. Tight integration with SQL
4. Full Portability
5. Tight Security
6. Supports Object Oriented Programming concepts.
7. Scalability and Manageability
8. Supports Web Application Development
9. Supports Server Page Development

## Disadvantages of PL/SQL

1. Stored Procedures in PL/SQL uses high memory
2. Lacks functionality debugging in stored procedures
3. Any change in underlying database requires change in the presentation layer also
4. Does not completely separate roles of back-end developer and fron-end developer
5. Difficult to separate HTML development with PL/SQL development

# SQL Vs PL/SQL Vs T-SQL: Key Differences

SQL is the standard language to query a database.

PL SQL basically stands for “Procedural Language extensions to SQL.” This is the extension of Structured Query Language (SQL) that is used in Oracle.

T-SQL basically stands for ” Transact-SQL.” This is the extension of Structured Query Language (SQL) that is used in Microsoft.

**Difference between SQL and PL/SQL**

|  |  |
| --- | --- |
| **SQL** | **PL/SQL** |
| * SQL is a single query that is used to perform DML and DDL operations. | * PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc. |
| * It is declarative, that defines what need to be done, rather than how things need to be done. | * PL/SQL is procedural that defines how the things needs to be done. |
| * Execute as a single statement. | * Execute as a whole block. |
| * Mainly used to manipulate data. | * Mainly used to create an application. |
| * Interaction with a Database server. | * No interaction with the database server. |
| * Cannot contain PL/SQL code in it. | * It is an extension of SQL, so that it can contain SQL inside it. |

**Difference Between T-SQL and PL-SQL**

|  |  |
| --- | --- |
| **T-SQL** | **PL-SQL** |
| * T-SQL is a Microsoft product. | * PL-SQL is developed by Oracle. |
| * Full Form of TL SQL is Transact Structure Query language. | * Full Form of PL SQL is Procedural Language Structural Query Language. |
| * T-SQL gives a high degree of control to programmers. | * It is a natural programming language that blends easily with the SQL |
| * T-SQL performs best with Microsoft SQL server | * PL-SQL performs best with Oracle database server. |
| * It is easy and simple to understand. | * PL-SQL is complex to understand. |
| * T-SQL allows inserting multiples rows into a table using the BULK INSERT statement. | * PL/SQL supports oops concepts like data encapsulation, function overloading, and information hiding. |
| * SELECT INTO statement used in T-SQL | * INSERT INTO statement must be used in PL/SQL |
| * In T-SQL NOT EXISTS clause used along with SELECT statements. | * In PL/SQL, there is a MINUS operator, which could be used with SELECT statements |

**Difference between SQL and T-SQL**

|  |  |
| --- | --- |
| **SQL** | **T-SQL** |
| * SQL is a programming language which focuses on managing relational databases. | * T-SQL is a procedural extension used by SQL Server. |
| * This is used for controlling and manipulating data where large amounts of information are stored about products, clients, etc. | * T-SQL has some features that are not available in SQL. Like procedural programming elements and a local variable to provide more flexible control of how the application flows. |
| * SQL queries submitted individually to the database server. | * T-SQL writes a program in such a way that all commands are submitted to the server in a single go |
| * The syntax was formalized for many commands; some of these are SELECT, INSERT, UPDATE, DELETE, CREATE, and DROP. | * It also includes special functions like the converted date () and some other functions which are not part of the regular SQL. |

# PL/ SQL Block: STRUCTURE, Syntax, ANONYMOUS Example

## What is PL/SQL block?

In PL/SQL, the code is not executed in single line format, but it is always executed by grouping the code into a single element called Blocks. In this tutorial, you are going to learn about these blocks.

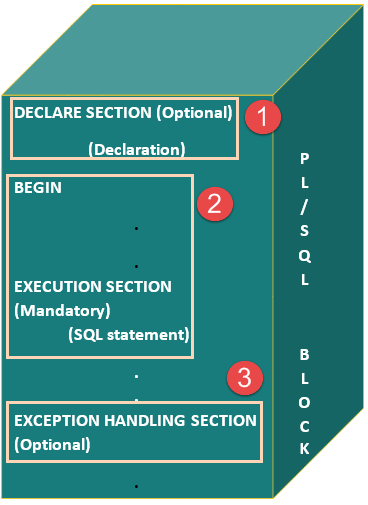
Blocks contain both PL/SQL as well as SQL instruction. All these instruction will be executed as a whole rather than executing a single instruction at a time.

## Block Structure

PL/SQL blocks have a pre-defined structure in which the code is to be grouped. Below are different sections of PL/SQL blocks.

1. Declaration section
2. Execution section
3. Exception-Handling section

The below picture illustrates the different PL/SQL block and their section order.



### Declaration Section

This is the first section of the PL/SQL blocks. This section is an optional part. This is the section in which the declaration of variables, cursors, exceptions, subprograms, pragma instructions and collections that are needed in the block will be declared. Below are few more characteristics of this part.

Important Linux Commands for Beginners Linux Tutorial

* This particular section is optional and can be skipped if no declarations are needed.
* This should be the first section in a PL/SQL block, if present.
* This section starts with the keyword ‘DECLARE’ for triggers and anonymous block. For other subprograms, this keyword will not be present. Instead, the part after the subprogram name definition marks the declaration section.
* This section should always be followed by execution section.

### Execution Section

Execution part is the main and mandatory part which actually executes the code that is written inside it. Since the PL/SQL expects the executable statements from this block this cannot be an empty block, i.e., it should have at least one valid executable code line in it. Below are few more characteristics of this part.

* This can contain both PL/SQL code and SQL code.
* This can contain one or many blocks inside it as a nested block.
* This section starts with the keyword ‘BEGIN’.
* This section should be followed either by ‘END’ or Exception-Handling section (if present)

### Exception-Handling Section:

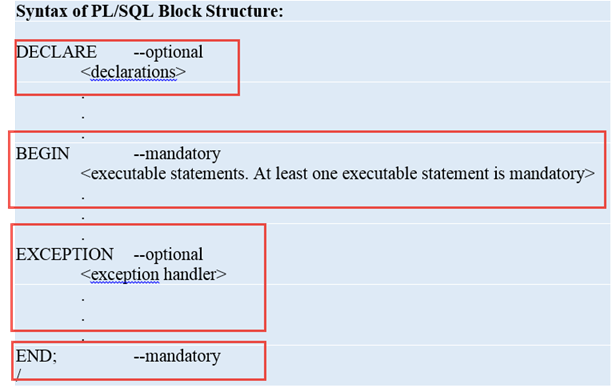
The exception is unavoidable in the program which occurs at run-time and to handle this Oracle has provided an Exception-handling section in blocks. This section can also contain PL/SQL statements. This is an optional section of the PL/SQL blocks.

* This is the section where the exception raised in the execution block is handled.
* This section is the last part of the PL/SQL block.
* Control from this section can never return to the execution block.
* This section starts with the keyword ‘EXCEPTION’.
* This section should always be followed by the keyword ‘END’.

The Keyword ‘END’ marks the end of PL/SQL block.

## PL/SQL Block Syntax

Below is the syntax of the PL/SQL block structure.



DECLARE --optional

<declarations>

BEGIN --mandatory

<executable statements. At least one executable statement is mandatory>

EXCEPTION --optional

<exception handles>

END; --mandatory

/

**Note:** A block should always be followed by ‘/’ which sends the information to the compiler about the end of the block.

## Types of PL/SQL block

PL/SQL blocks are of mainly two types.

1. Anonymous blocks
2. Named Blocks

### Anonymous blocks:

Anonymous blocks are PL/SQL blocks which do not have any names assigned to them. They need to be created and used in the same session because they will not be stored in the server as database objects.

Since they need not store in the database, they need no compilation steps. They are written and executed directly, and compilation and execution happen in a single process.

Below are few more characteristics of Anonymous blocks.

* These blocks don’t have any reference name specified for them.
* These blocks start with the keyword ‘DECLARE’ or ‘BEGIN’.
* Since these blocks do not have any reference name, these cannot be stored for later purpose. They shall be created and executed in the same session.
* They can call the other named blocks, but call to anonymous block is not possible as it is not having any reference.
* It can have nested block in it which can be named or anonymous. It can also be nested in any blocks.
* These blocks can have all three sections of the block, in which execution section is mandatory, the other two sections are optional.

### Named blocks:

Named blocks have a specific and unique name for them. They are stored as the database objects in the server. Since they are available as database objects, they can be referred to or used as long as it is present on the server. The compilation process for named blocks happens separately while creating them as a database objects.

Below are few more characteristics of Named blocks.

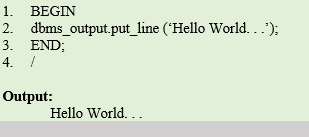
* These blocks can be called from other blocks.
* The block structure is same as an anonymous block, except it will never start with the keyword ‘DECLARE’. Instead, it will start with the keyword ‘CREATE’ which instruct the compiler to create it as a database object.
* These blocks can be nested within other blocks. It can also contain nested blocks.
* Named blocks are basically of two types:

1. Procedure
2. Function

We will learn more about these named blocks in “Procedure” and “Function” topics in later tutorial.

**How to write a simple program using PL/SQL**

In this section, we are going to write a simple program for printing “Hello World” using “Anonymous block”.



BEGIN

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

END;

/

**Output:**

Hello World...

**Code Explanation:**

* **Code line 2**: Prints the message “Hello World. . .”
* The below screenshot explains how to enter the code in SQL\* Plus.

**Note:** A block should be always followed by ‘/’ which sends the information to the compiler about the end of the block. Till the compiler encounters ‘/’, it will not consider the block is completed, and it will not execute it.



**Declaring and usage of variables in the program**

Here we are going to print the “Hello World” using the variables.

DECLARE

text VARCHAR2(25);

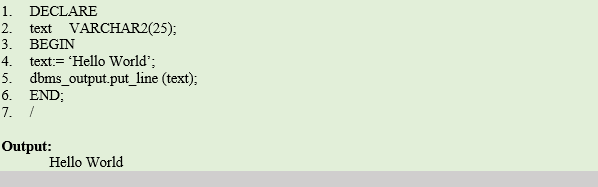
BEGIN

text:= ‘Hello World’;

dbms\_output.put\_line (text);

END:

/

**Output:** 

Hello World

**Code Explanation:**

* **Code line 2**: Declaring a variable “text” of a VARCHAR2 type with size 25
* **Code line 4**: Assigning the value “Hello World” to the variable “text”.
* **Code line 5**: Printing the value of the variable “text”.

**Comments in PL/SQL**

Commenting code simply instructs the compiler to ignore that particular code from executing.

Comment can be used in the program to increase the readability of the program. In PL/SQL codes can be commented in two ways.

* Using ‘–‘ in the beginning of the line to comment that particular line.
* Using ‘/\*…….\*/’ we can use multiple lines. The symbol ‘/\*’ marks the starting of the comment and the symbol ‘\*/’ marks the end of the comment. The code between these two symbols will be treated as comments by the compiler.

**Example**: In this example, we are going to print ‘Hello World’ and we are also going to see how the commented lines behave in the code



BEGIN

--single line comment

dbms output.put line (' Hello World ’);

/\*Multi line commenting begins

Multi line commenting ends \*/

END;

/

**Output:**

Hello World

**Code Explanation:**

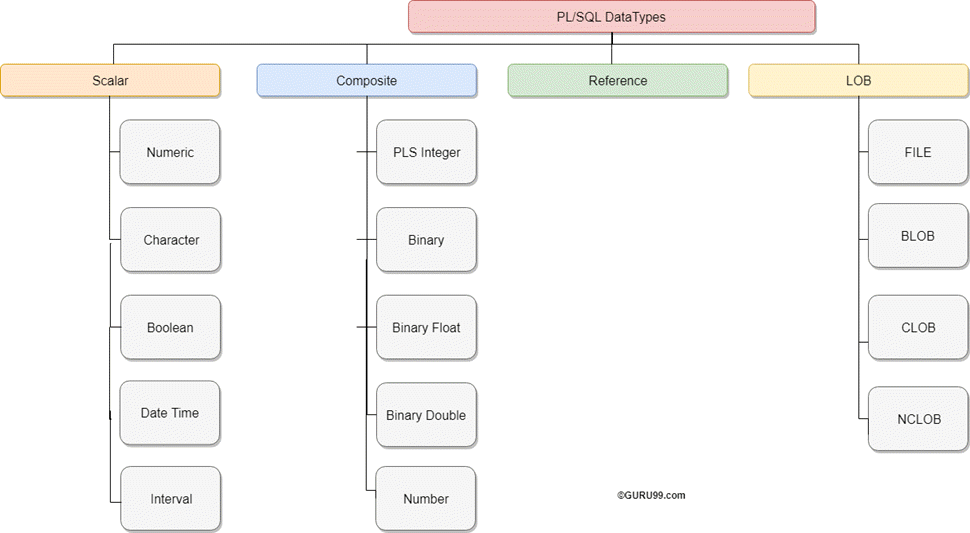
* **Code line 2**: Single line comment and compiler ignored this line from execution.
* **Code line 3**: Printing the value “Hello World.”
* **Code line 4**: Multiline commenting starts with ‘/\*’
* **Code line 5**: Multiline commenting ends with ‘\*/’

**What is PL/SQL Datatypes?**

**Data Types** in PL/SQL are used to define how the data will be stored, handled, and treated by Oracle during the data storage and processing. Data types are associated with the specific storage format and range constraints. In Oracle, each value or constant is assigned with a data type.

The main difference between PL/SQL and[SQL](https://www.guru99.com/sql.html)data types is, SQL data type are limited to table column while the PL/SQL data types are used in the [PL/SQL blocks](https://www.guru99.com/blocks-pl-sql.html). More on this later in the tutorial.

Following is the diagram of different Oracle PL/SQL Data Types:



Different Data Types in PL/SQL

In this PL/SQL Data Types Tutorial , you will learn-

* [PL/SQL CHARACTER Data Type](https://www.guru99.com/pl-sql-data-types.html#1)
* [PL/SQL NUMBER Data Type](https://www.guru99.com/pl-sql-data-types.html#2)
* [PL/SQL BOOLEAN Data Type](https://www.guru99.com/pl-sql-data-types.html#3)
* [PL/SQL DATE Data Type](https://www.guru99.com/pl-sql-data-types.html#4)
* [PL/SQL LOB Data Type](https://www.guru99.com/pl-sql-data-types.html#5)

**PL/SQL CHARACTER Data Type**

This data type basically stores alphanumeric characters in string format.

The literal values should always be enclosed in single quotes while assigning them to CHARACTER data type.

**This character data type is further classified as follows:**

* CHAR Data type (fixed string size)
* VARCHAR2 Data type (variable string size)
* VARCHAR Data type
* NCHAR (native fixed string size)
* NVARCHAR2 (native variable string size)
* LONG and LONG RAW

| **­** | **Description** | **Syntax** |
| --- | --- | --- |
|  |  |  |
| **CHAR** | This data type stores the string value, and the size of the string is fixed at the time of declaring the variable.   * Oracle would be blank-padded the variable if the variable didn’t occupy the entire size that has been declared for it, Hence Oracle will allocate the memory for declared size even if the variable didn’t occupy it fully. * The size restriction for this data type is 1-2000 bytes. * CHAR data type is more appropriate to use where ever fixed the size of data will be handled. | grade CHAR;  manager CHAR (10):= 'guru99';  **Syntax Explanation:**   * The first declaration statement declared the variable ‘grade’ of CHAR data type with the maximum size of 1 byte (default value). * The second declaration statement declared the variable ‘manager’ of CHAR data type with the maximum size of 10 and assigned the value ‘guru99’ which is of 6 bytes. Oracle will allocate the memory of 10 bytes rather than 6 bytes in this case. |
| **VARCHAR2** | This data type stores the string, but the length of the string is not fixed.   * The size restriction for this data type is 1-4000 bytes for table column size and 1-32767 bytes for variables. * The size is defined for each variable at the time of variable declaration. * But Oracle will allocate memory only after the variable is defined, i.e., Oracle will consider only the actual length of the string that is stored in a variable for memory allocation rather than the size that has been given for a variable in the declaration part. * It is always good to use VARCHAR2 instead of CHAR data type to optimize the memory usage. | manager VARCHAR2(10) := ‘guru99';  **Syntax Explanation:**   * The above declaration statement declared the variable ‘manager’ of VARCHAR2 data type with the maximum size of 10 and assigned the value ‘guru99’ which is of 6 bytes. Oracle will allocate memory of only 6 bytes in this case. |
| **VARCHAR** | This is synonymous with the VARCHAR2 data type.   * It is always a good practice to use VARCHAR2 instead of VARCHAR to avoid behavioral changes. | manager VARCHAR(10) := ‘guru99';  **Syntax Explanation:**   * The above declaration statement declared the variable ‘manager’ of VARCHAR data type with the maximum size of 10 and assigned the value ‘guru99’ which is of 6 bytes. Oracle will allocate memory of only 6 bytes in this case. (Similar to VARCHAR2) |
| **NCHAR** | This data type is same as CHAR data type, but the character set will of the national character set.   * This character set can be defined for the session using NLS\_PARAMETERS. * The character set can be either UTF16 or UTF8. * The size restriction is 1-2000 bytes. | native NCHAR(10);  **Syntax Explanation:**   * The above declaration statement declares the variable ‘native’ of NCHAR data type with the maximum size of 10. * The length of this variable depends upon the (number of lengths) per byte as defined in the character set. |
| **NVARCHAR2** | This data type is same as VARCHAR2 data type, but the character set will be of the national character set.   * This character set can be defined for the session using NLS\_PARAMETERS. * The character set can be either UTF16 or UTF8. * The size restriction is 1-4000 bytes. | Native var NVARCHAR2(10):='guru99';  **Syntax Explanation:**   * The above declaration statement declares the variable ‘Native\_var’ of NVARCHAR2 data type with the maximum size of 10. |
| **LONG and LONGRAW** | This data type is used to store large text or raw data up to the maximum size of 2GB.   * These are mainly used in the data dictionary. * LONG data type is used to store character set data, while LONG RAW is used to store data in binary format. * LONG RAW data type accepts media objects, images, etc. whereas LONG works only on data that can be stored using character set. | Large\_text LONG;  Large\_raw LONG RAW;  **Syntax Explanation:**   * The above declaration statement declares the variable ‘Large\_text’ of LONG data type and ‘Large\_raw’ of LONG RAW data type.   **Note:** Using LONG data type is not recommended by Oracle. Instead, LOB data type should be preferred. |

**PL/SQL NUMBER Data Type**

This data type stores fixed or floating point numbers up to 38 digits of precision. This data type is used to work with fields which will contain only number data. The variable can be declared either with precision and decimal digit details or without this information. Values need not enclose within quotes while assigning for this data type.

A NUMBER(8,2);

B NUMBER(8);

C NUMBER;

**Syntax Explanation:**

* In the above, the first declaration declares the variable ‘A’ is of number data type with total precision 8 and decimal digits 2.
* The second declaration declares the variable ‘B’ is of number data type with total precision 8 and no decimal digits.
* The third declaration is the most generic, declares variable ‘C’ is of number data type with no restriction in precision or decimal places. It can take up to a maximum of 38 digits.

**PL/SQL BOOLEAN Data Type**

This data type stores the logical values. Oracle Boolean Data Type represents either TRUE or FALSE and mainly used in conditional statements. Values need not enclose within quotes while assigning for this data type.

Var1 BOOLEAN;

**Syntax Explanation:**

* In the above, variable ‘Var1’ is declared as BOOLEAN data type. The output of the code will be either true or false based on the condition set.

**PL/SQL DATE Data Type**

This data type stores the values in date format, as date, month, and year. Whenever a variable is defined with DATE data type along with the date it can hold time information and by default time information is set to 12:00:00 if not specified. Values need to enclose within quotes while assigning for this data type.

The standard Oracle time format for input and output is ‘DD-MON-YY’ and it is again set at NLS\_PARAMETERS (NLS\_DATE\_FORMAT) at the session level.

newyear DATE:='01-JAN-2015';

current\_date DATE:=SYSDATE;

**Syntax Explanation:**

* In the above, variable ‘newyear’ is declared as DATE data type and assigned the value of Jan 1st, 2015 date.
* The second declaration declares the variable current\_date as DATE data type and assigned the value with current system date.
* Both these variable holds the time information.

**PL/SQL LOB Data Type**

This data type is mainly used to store and manipulate large blocks of unstructured data’s like images, multimedia files, etc. Oracle prefers LOB instead of the a LONG data type as it is more flexible than the LONG data type. The below are the few main advantage of LOB over LONG data type.

* The number of column in a table with LONG data type is limited to 1, whereas a table has no restriction on a number of columns with LOB data type.
* The data interface tool accepts LOB data type of the table during data replication, but it omits LONG column of the table. These LONG columns need to be replicated manually.
* The size of the LONG column is 2GB, whereas LOB can store up to 128 TB.
* Oracle is constantly improving the LOB data type in each of their releases according to the modern requirement, whereas LONG data type is constant and not getting many updates.

So, it is always good to use LOB data type instead of the LONG data type. Following are the different LOB data types. They can store up to the size of 128 terabytes.

1. BLOB
2. CLOB and NCLOB
3. BFILE

| **Data Type** | **Description** | **Syntax** |
| --- | --- | --- |
| **BLOB** | This data type stores the LOB data in the binary file format up to the maximum size of 128 TB. This doesn’t store data based on the character set details, so that it can store the unstructured data such as multimedia objects, images, etc. | Binary\_data BLOB;  **Syntax Explanation:**   * In the above, variable ‘Binary\_data’ is declared as a BLOB. |
| **CLOB and NCLOB** | CLOB data type stores the LOB data into the character set, whereas NCLOB stores the data in the native character set. Since these data types use character set based storage, these cannot store the data like multimedia, images, etc. that cannot be put into a character string. The maximum size of these data types is 128 TB. | Charac\_data CLOB;  **Syntax Explanation:**   * In the above, variable ‘Charac\_data’ is declared as CLOB data type. |
| **BFILE** | * BFILE are the data types that stored the unstructured binary format data outside the database as an operating-system file. * The size of BFILE is to a limited operating system, and they are read-only files and can’t be modified. |  |

**What are PL/SQL Identifiers?**

**Identifiers** in PL/SQL are nothing but names given to a PL/SQL object. The object could be constant, variable, exception, cursor, procedure, function, package, trigger, object type, reserved word, or label. These identifiers contain letters, numerical, signs, underscores, etc. They are case-insensitive and limited to 30 characters size.

In this tutorial, you will learn-

**Properties of PL/SQL Identifiers**

Here are the main properties of PL/SQL identifiers:

* Must start with a letter
* Maximum size is limited to 30 letters
* Cannot contain whitespace characters
* Can contain dollar sign (‘$’), underscore (‘\_’) and hash sign (‘#’)
* Is case-insensitive

**PL/SQL Variables**

**Variables** in PL/SQL are basic identifiers assigned to a storage area that a program can manipulate. Variables are nothing but placeholders where the user can store values. These variables need to be associated with some valid PL/SQL datatypes before using them. Datatypes define the storage and processing methods for these variables.

Linux Distributions & Installation Methods - Linux Tutorial 2

**PL/SQL Variable Declaration**

Variables are mainly used to store data during the data manipulation or data processing. They need to be declared before using them inside the program. This declaration needs to be done in the declarative section of the [PL/SQL Blocks](https://www.guru99.com/blocks-pl-sql.html).

Declaration of variables is a process of assigning the name to the placeholder and associate the same with a valid datatype.

**Syntax:**

<variable name> <datatvpe>;

The above syntax shows how to declare the variable in the declarative section.

**Data Storing in PL/SQL Variables**

Once the variable is declared, they are ready to hold the data of defined type. The values of these variables can be assigned either in execution section or at the time of declaring itself. The value can be either a literal or another variable’s value. Once a particular value has been assigned, it will be stored in the allocated memory space for that variable.

**Syntax:**

<variable\_name> <datatype> := <default\_value>;

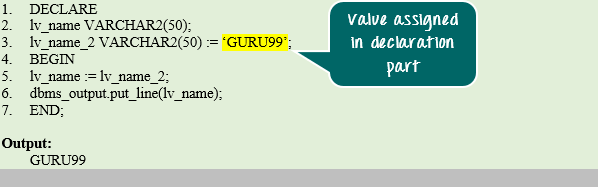
The above syntax shows how to declare the variable and assign value in the declarative section.

<variable\_name> <datatype>;

<variable name> := <value>;

The above syntax shows how to assign the value to an already declared variable.

**Example1:** In this example, we are going to learn how to declare the variable and how to assign the value to them. We are going to print ‘GURU99’ in the following program by using the variables.



DECLARE

lv\_name VARCHAR2(50);

lv\_name\_2 VARCHAR2(50) := ‘GURU99';

BEGIN

lv\_name := lv\_name\_2;

dbms\_output .put\_line(lv\_name);

END:

**Code Explanation:**

* **Code line 2**: Declaring the variable ‘lv\_name’ of VARCHAR2 with size 50.
* **Code line 3**: Declaring the variable ‘lv\_name\_2’ of VARCHAR2 with size 50 and assigned the default value using literal ‘GURU99’.
* **Code line 5**: Value for variable ‘lv\_name’ has been assigned from the variable ‘lv\_name\_2’.
* **Code line 6**: Printing the stored value of variable ‘lv\_name’.

When the above code is executed, you will get the following output.

**Output:**

GURU99

## 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.

Example of initilizing variable

Let's take a simple example to explain it well:

**DECLARE**

   a **integer** := 30;

   b **integer** := 40;

   c **integer**;

   f **real**;

**BEGIN**

   c := a + b;

   dbms\_output.put\_line('Value of c: ' || c);

   f := 100.0/3.0;

   dbms\_output.put\_line('Value of f: ' || f);

**END**;

Variable Scope in PL/SQL:

PL/SQL allows nesting of blocks. A program block can contain another inner block. If you declare a variable within an inner block, it is not accessible to an outer block. There are two types of variable scope:

* Local Variable: Local variables are the inner block variables which are not accessible to outer blocks.
* Global Variable: Global variables are declared in outermost block.

Example of Local and Global variables

Let's take an example to show 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**;

/

After the execution, this will produce the following result:

Outer Variable num1: 95

Outer Variable num2: 85

**Inner** Variable num1: 195

**Inner** Variable num2: 185

PL/SQL **procedure** successfully completed.

# **PL/SQL Constants**

A constant is a value used in a PL/SQL block that remains unchanged throughout the program. It is a user-defined literal value. It can be declared and used instead of actual values.

Let's take an example to explain it well:

Suppose, you have to write a program which will increase the salary of the employees upto 30%, you can declare a constant and use it throughout the program. Next time if you want to increase the salary again you can change the value of constant than the actual value throughout the program.

**Syntax to declare a constant:**

1. constant\_name CONSTANT datatype := VALUE;

* **Constant\_name:**it is the name of constant just like variable name. The constant word is a reserved word and its value does not change.
* **VALUE:**it is a value which is assigned to a constant when it is declared. It can not be assigned later.

Example of PL/SQL constant

Let's take an example to explain it well:

**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**;

/

After the execution of the above code at SQL prompt, it will produce the following result:.

1. Radius: 9.5
2. Diameter: 19
3. Circumference: 59.69
4. Area: 283.53
6. Pl/SQL **procedure** successfully completed.

PL/SQL Literals

Literals are the explicit numeric, character, string or boolean values which are not represented by an identifier. For example: TRUE, NULL, etc. are all literals of type boolean. PL/SQL literals are case-sensitive. There are following kinds of literals in PL/SQL:

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

Example of these different types of Literals:

|  |  |
| --- | --- |
| **Literals** | **Examples** |
| Numeric | 75125, 3568, 33.3333333 etc. |
| Character | 'A' '%' '9' ' ' 'z' '(' |
| String | Hello JavaTpoint! |
| Boolean | TRUE, FALSE, NULL etc. |
| Date and Time | '26-11-2002' , '2012-10-29 12:01:01' |

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program −

DECLARE

message varchar2(30):= 'That''s tutorialspoint.com!';

BEGIN

dbms\_output.put\_line(message);

END;

/

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

That's tutorialspoint.com!

PL/SQL procedure successfully completed.

# **PL/SQL - Operators**

In this chapter, we will discuss operators in PL/SQL. An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators −

* Arithmetic operators
* Relational operators
* Comparison operators
* Logical operators
* String operators

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter − **PL/SQL - Strings**.

## 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 −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_arithmetic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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**.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_comparison_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Opertor** | **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 >= a and x <= 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 −

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **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 Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 \* 2**; here, **x** is assigned **13**, not 20 because operator \* has higher precedence than +, so it first gets multiplied with **3\*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

[Show Examples](https://www.tutorialspoint.com/plsql/plsql_operators_precedence.htm)

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| \*\* | Exponentiation |
| +, - | identity, negation |
| \*, / | multiplication, division |
| +, -, || | addition, subtraction, concatenation |
| Comparison |  |
| NOT | logical negation |
| AND | Conjunction |
| OR | Inclusion |

# **PL/SQL - Conditions**

In this chapter, we will discuss conditions in PL/SQL. Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages −

PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **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. |
| 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. |
| 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. |
| 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. |
| 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). |

# Example: IF-THEN-ELSE Statement

There are 3 different syntax for this statement:

# Syntax 1:

**If condition THEN**

**….**

**…..**

**End if;**

declare

a int :=10; begin

if a<10 then dbms\_output.put\_line('a<10');

else

dbms\_output.put\_line('a<10');

end if; end;

**Syntax 2:**

**If condition THEN**

**….**

**Else**

**…… End if;**

**Syntax 3:**

**If condition THEN**

**….**

**Elsif condition THEN**

**….**

**Else**

**…..**

**End if;**

declare

a int :=10; begin

if a>10 then dbms\_output.put\_line('a>10');

elsif a=10 then

dbms\_output.put\_line('a=10');

else

dbms\_output.put\_line('a<10');

end if; end;

# CASE Statement:

Starting in Oracle 9i, you can use the **case** statement within an SQL statement. It has the functionality of an IF-THEN-ELSE statement.

# Syntax:

**CASE <expression>**

**WHEN Condition\_1 THEN**

**……**

**WHEN Condition\_2 THEN**

**. …… ELSE result;**

**END CASE;**

**// Program To perform case operation using local variable**

declare

a int :=55; begin

CASE

when (a <40) then dbms\_output.put\_line('Grade is fail');

when (a>=40 and a<70) then dbms\_output.put\_line('Grade is C');

when (a>=70 and a<80) then dbms\_output.put\_line('Grade is B');

Else

dbms\_output.put\_line('Grade is A');

end case; end;

# Loop Statement

**Syntax:**

**Loop**

**……..**

**……..**

**End Loop;**

**Note: This statement is mainly used when we are unaware, how many times we need to execute the loop.**

**This statement terminates when it finds EXIT or EXIT WHEN.**

declare

cnt int :=0; begin

end;

# FOR Loop:

**Syntax 1:**

**FOR loop\_counter IN Min\_Count …. Max\_Count LOOP**

**..………**

**………..**

**END LOOP;**

**Note: The counter will start at “ Min “ & ends at “ Max “. Syntax 2:**

**FOR loop\_counter IN REVERSE Min\_Count …. Max\_Count LOOP**

**..………**

**………..**

**END LOOP;**

**Note: The counter will start at “ Max “ & ends at “ Min “.**

Syntax 1:

Declare

i int;

begin

for i IN 1..10 loop

dbms\_output.put\_line(' i = '|| i);

end loop;

end;

Syntax 2:

Declare

i int;

begin

for i IN reverse 1..10 loop

dbms\_output.put\_line(' i = '|| i);

end loop;

end;

# WHILE Loop:

**Syntax:**

**WHILE Condition Loop**

**……..**

**……..**

**End Loop;**

This loop will be used when we are not sure about how many times to execute the body.

declare

cnt int :=0;

begin

while cnt<=10 loop

dbms\_output.put\_line('count = '|| cnt); cnt:=cnt+1;

end loop;

end;

# **PL/SQL GOTO Statement**

## Introduction to PL/SQL GOTO statement

The GOTO statement allows you to transfer control to a labeled block or statement. The following illustrates the syntax of the GOTO statement:

GOTO label\_name;

Code language: SQL (Structured Query Language) (sql)

The label\_name is the name of a label that identifies the target statement. In the program, you surround the label name with double enclosing angle brackets as shown below:

<<label\_name>>;

Code language: SQL (Structured Query Language) (sql)

When PL/SQL encounters a GOTO statement, it transfers control to the first executable statement after the label.

## PL/SQL GOTO statement example

The following shows an example of using the GOTO statements.

BEGIN

GOTO second\_message;

<<first\_message>>

DBMS\_OUTPUT.PUT\_LINE( 'Hello' );

GOTO the\_end;

<<second\_message>>

DBMS\_OUTPUT.PUT\_LINE( 'PL/SQL GOTO Demo' );

GOTO first\_message;

<<the\_end>>

DBMS\_OUTPUT.PUT\_LINE( 'and good bye...' );

END;

Code language: SQL (Structured Query Language) (sql)

The output is:

PL/SQL GOTO Demo

Hello

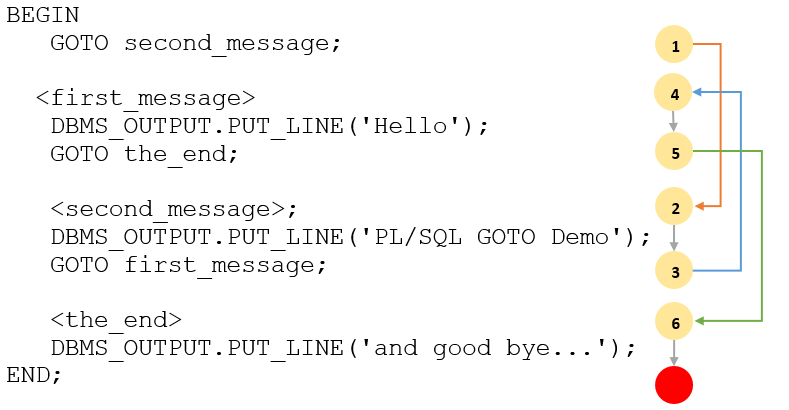
and good Bye...

Code language: SQL (Structured Query Language) (sql)

The following explains the sequence of the [block](https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/) in detail:

* First, the GOTO second\_message statement is encountered, therefore, the control is passed to the statement after the second\_message label.
* Second, the GOTO first\_message is encountered, so the control is transferred to the statement after the first\_message label.
* Third, the GOTO the\_end is reached, hence the control is passed to the statement after the the\_end label.

The picture below illustrates the sequence:



## PL/SQL GOTO statement restrictions

The GOTO statement is subject to the following restrictions.

First, you cannot use a GOTO statement to transfer control into an [IF](https://www.oracletutorial.com/plsql-tutorial/plsql-if/), [CASE](https://www.oracletutorial.com/plsql-tutorial/plsql-case-statement/) or [LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-loop/) statement, the same for sub-block.

The following example attempts to transfer control into an IF statement using a GOTO statement:

DECLARE

n\_sales NUMBER;

n\_tax NUMBER;

BEGIN

GOTO inside\_if\_statement;

IF n\_sales > 0 THEN

<<inside\_if\_statement>>

n\_tax := n\_sales \* 0.1;

END IF;

END;

Code language: SQL (Structured Query Language) (sql)

Oracle issued the following error:

PLS-00375: illegal GOTO statement; this GOTO cannot branch to label 'INSIDE\_IF\_STATEMENT'

Code language: SQL (Structured Query Language) (sql)

Second, you cannot use a GOTO statement to transfer control from one clause to another in the IF statement e.g., from IF clause to ELSIF or ELSE clause, or from one WHEN clause to another in the CASE statement.

The following example attempts to transfer control to a clause in the IF statement:

DECLARE

n\_sales NUMBER;

n\_commission NUMBER;

BEGIN

n\_sales := 120000;

IF n\_sales > 100000 THEN

n\_commission := 0.2;

GOTO zero\_commission;

elsif n\_sales > 50000 THEN

n\_commission := 0.15;

elsif n\_sales > 20000 THEN

n\_commission := 0.1;

ELSE

<<zero\_commission>>

n\_commission := 0;

END IF;

END;

Code language: SQL (Structured Query Language) (sql)

Oracle issued the following error.

PLS-00375: illegal GOTO statement; this GOTO cannot branch to label 'ZERO\_COMMISSION'

Code language: SQL (Structured Query Language) (sql)

Third, you cannot use a GOTO statement to transfer control out of a subprogram or into an [exception handler](https://www.oracletutorial.com/plsql-tutorial/plsql-exception/).

Fourth, you cannot use a GOTO statement to transfer control from an exception handler back into the current block.

In this tutorial, you have learned how to use the PL/SQL GOTO statement to transfer control to a labeled block or statement

## PL/SQL CONTINUE statement

The CONTINUE statement allows you to exit the current loop iteration and immediately continue on to the next iteration of that loop.

The CONTINUE statement has a simple syntax:

CONTINUE;

Code language: SQL (Structured Query Language) (sql)

Typically, the CONTINUE statement is used within an [IF THEN](https://www.oracletutorial.com/plsql-tutorial/plsql-if/) statement to exit the current loop iteration based on a specified condition as shown below:

IF condition THEN

CONTINUE;

END IF;

Code language: SQL (Structured Query Language) (sql)

The CONTINUE can be used in all loop constructs including [LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-loop/), [FOR LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-for-loop/) and [WHILE LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-while-loop/).

### **PL/SQL CONTINUE statement example**

The following is a simple example of using the CONTINUE statement to skip over loop body execution for odd numbers:

BEGIN

FOR n\_index IN 1 .. 10

LOOP

-- skip odd numbers

IF MOD( n\_index, 2 ) = 1 THEN

CONTINUE;

END IF;

DBMS\_OUTPUT.PUT\_LINE( n\_index );

END LOOP;

END;

Code language: SQL (Structured Query Language) (sql)

The output is:

2

4

6

8

10

Code language: SQL (Structured Query Language) (sql)

## PL/SQL CONTINUE WHEN statement

The CONTINUE WHEN statement exits the current loop iteration based on a condition and immediately continue to the next iteration of that loop.

The syntax of CONTINUE WHEN statement is:

CONTINUE WHEN condition;

Code language: SQL (Structured Query Language) (sql)

The condition in the WHEN clause is evaluated each time the CONTINUE WHEN statement is reached. If the condition is TRUE, the current loop is skipped, and control is transferred to the next iteration of the loop. If the condition is not TRUE, either FALSE or NULL, the CONTINUE WHEN statement does nothing.

Essentially, the CONTINUE WHEN statement is the combination of  an [IF THEN](https://www.oracletutorial.com/plsql-tutorial/plsql-if/) statement and CONTINUE statement:

IF condition THEN

CONTINUE;

END IF;

Code language: SQL (Structured Query Language) (sql)

Similar to the CONTINUE statement, you can use the CONTINUE WHEN statement in [LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-loop/), [FOR LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-for-loop/) and [WHILE LOOP](https://www.oracletutorial.com/plsql-tutorial/plsql-while-loop/).

### **PL/SQL CONTINUE statement example**

The following example illustrates how to use the CONTINUE WHEN statement to skip over loop body execution for even numbers:

BEGIN

FOR n\_index IN 1 .. 10

LOOP

-- skip even numbers

CONTINUE

WHEN MOD( n\_index, 2 ) = 0;

DBMS\_OUTPUT.PUT\_LINE( n\_index );

END LOOP;

END;

Code language: SQL (Structured Query Language) (sql)

Here is the output:

1

3

5

7

9

Code language: SQL (Structured Query Language) (sql)

In this tutorial, you have learned how to use exit the current loop iteration and continue to the next one using the PL/SQL CONTINUE or CONTINUE WHEN statement.

**PL/SQL String**

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings −

* **Fixed-length strings** − In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
* **Variable-length strings** − In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
* **Character large objects (CLOBs)** − These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another. For example,

'this isn''t what it looks like'

## 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.

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);

BEGIN

name := 'John Smith';

company := 'Infotech';

introduction := ' Hello! I''m John Smith from Infotech.';

choice := 'y';

IF choice = 'y' THEN

dbms\_output.put\_line(name);

dbms\_output.put\_line(company);

dbms\_output.put\_line(introduction);

END IF;

END;

/

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

John Smith

Infotech

Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. The following two declarations are identical −

red\_flag CHAR(1) := 'Y';

red\_flag CHAR := 'Y';

## PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator **(||)** for joining two strings. The following table provides the string functions provided by PL/SQL −

|  |  |
| --- | --- |
| **S.No** | **Function & Purpose** |
| 1 | **ASCII(x);**  Returns the ASCII value of the character x. |
| 2 | **CHR(x);**  Returns the character with the ASCII value of x. |
| 3 | **CONCAT(x, y);**  Concatenates the strings x and y and returns the appended string. |
| 4 | **INITCAP(x);**  Converts the initial letter of each word in x to uppercase and returns that string. |
| 5 | **INSTR(x, find\_string [, start] [, occurrence]);**  Searches for **find\_string** in x and returns the position at which it occurs. |
| 6 | **INSTRB(x);**  Returns the location of a string within another string, but returns the value in bytes. |
| 7 | **LENGTH(x);**  Returns the number of characters in x. |
| 8 | **LENGTHB(x);**  Returns the length of a character string in bytes for single byte character set. |
| 9 | **LOWER(x);**  Converts the letters in x to lowercase and returns that string. |
| 10 | **LPAD(x, width [, pad\_string]) ;**  Pads **x** with spaces to the left, to bring the total length of the string up to width characters. |
| 11 | **LTRIM(x [, trim\_string]);**  Trims characters from the left of **x**. |
| 12 | **NANVL(x, value);**  Returns value if x matches the NaN special value (not a number), otherwise **x** is returned. |
| 13 | **NLS\_INITCAP(x);**  Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT. |
| 14 | **NLS\_LOWER(x) ;**  Same as the LOWER function except that it can use a different sort method as specified by NLSSORT. |
| 15 | **NLS\_UPPER(x);**  Same as the UPPER function except that it can use a different sort method as specified by NLSSORT. |
| 16 | **NLSSORT(x);**  Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used. |
| 17 | **NVL(x, value);**  Returns value if **x** is null; otherwise, x is returned. |
| 18 | **NVL2(x, value1, value2);**  Returns value1 if x is not null; if x is null, value2 is returned. |
| 19 | **REPLACE(x, search\_string, replace\_string);**  Searches **x** for search\_string and replaces it with replace\_string. |
| 20 | **RPAD(x, width [, pad\_string]);**  Pads **x** to the right. |
| 21 | **RTRIM(x [, trim\_string]);**  Trims **x** from the right. |
| 22 | **SOUNDEX(x) ;**  Returns a string containing the phonetic representation of **x**. |
| 23 | **SUBSTR(x, start [, length]);**  Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied. |
| 24 | **SUBSTRB(x);**  Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems. |
| 25 | **TRIM([trim\_char FROM) x);**  Trims characters from the left and right of **x**. |
| 26 | **UPPER(x);**  Converts the letters in x to uppercase and returns that string. |

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

### **Example 1**

DECLARE

greetings varchar2(11) := 'hello world';

BEGIN

dbms\_output.put\_line(UPPER(greetings));

dbms\_output.put\_line(LOWER(greetings));

dbms\_output.put\_line(INITCAP(greetings));

/\* retrieve the first character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings, 1, 1));

/\* retrieve the last character in the string \*/

dbms\_output.put\_line ( SUBSTR (greetings, -1, 1));

/\* retrieve five characters,

starting from the seventh position. \*/

dbms\_output.put\_line ( SUBSTR (greetings, 7, 5));

/\* retrieve the remainder of the string,

starting from the second position. \*/

dbms\_output.put\_line ( SUBSTR (greetings, 2));

/\* find the location of the first "e" \*/

dbms\_output.put\_line ( INSTR (greetings, 'e'));

END;

/

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

HELLO WORLD

hello world

Hello World

h

d

World

ello World

2

PL/SQL procedure successfully completed.

### **Example 2**

DECLARE

greetings varchar2(30) := '......Hello World.....';

BEGIN

dbms\_output.put\_line(RTRIM(greetings,'.'));

dbms\_output.put\_line(LTRIM(greetings, '.'));

dbms\_output.put\_line(TRIM( '.' from greetings));

END;

/

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

......Hello World

Hello World.....

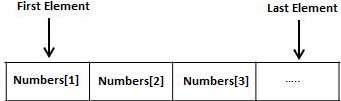
Hello World

PL/SQL procedure successfully completed.

# **PL/SQL - Arrays**

In this chapter, we will discuss arrays in PL/SQL. 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.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter **'PL/SQL Collections'**.

Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

## Creating a Varray Type

A varray type is created with the **CREATE TYPE** statement. You must specify the maximum size and the type of elements stored in the varray.

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

CREATE OR REPLACE TYPE varray\_type\_name IS VARRAY(n) of <element\_type>

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.

Maximum size of a varray can be changed using the **ALTER TYPE** statement.

For example,

CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);

/

Type created.

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 −

### **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

PL/SQL procedure successfully completed.

**Please note** −

* In Oracle environment, the starting index for varrays is always 1.
* You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
* Varrays are one-dimensional arrays.
* A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

### **Example 2**

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept.

We will use the CUSTOMERS table stored in our database as −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

# **PL/SQL anonymous blocks using tables**

%Type:to find exact value.

%rowtype :to find entire row;

**Summary**: in this tutorial, you will learn how to use the PL/SQL SELECT INTO statement to fetch data of a single row from a table into variables.

PL/SQL SELECT INTO statement is the simplest and fastest way to fetch a single row from a table into [variables](https://www.oracletutorial.com/plsql-tutorial/plsql-variables/). The following illustrates the syntax of the PL/SQL SELECT INTO statement:

SELECT

select\_list

INTO

variable\_list

FROM

table\_name

WHERE

condition;

Code language: SQL (Structured Query Language) (sql)

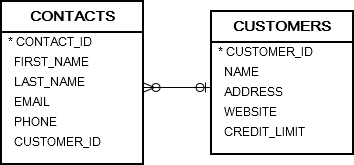
In this syntax, the number of columns in the variable\_list must be the same as the number of variables (or the number of components of a record) in the select\_list.  In addition, their corresponding data type must be compatible.

Besides the [WHERE](https://www.oracletutorial.com/oracle-basics/oracle-where/) clause, you can use other clauses in the [SELECT](https://www.oracletutorial.com/oracle-basics/oracle-select/) statement such as [INNER JOIN](https://www.oracletutorial.com/oracle-basics/oracle-inner-join/), [GROUP BY](https://www.oracletutorial.com/oracle-basics/oracle-group-by/), [HAVING](https://www.oracletutorial.com/oracle-basics/oracle-having/), and [UNION](https://www.oracletutorial.com/oracle-basics/oracle-union/).

If the SELECT statement returns more than one row, Oracle will raise the TOO\_MANY\_ROWS [exception](https://www.oracletutorial.com/plsql-tutorial/plsql-exception/). If the SELECT statement does not return any row, Oracle will raise the NO\_DATA\_FOUND exception.

## PL/SQL SELECT INTO examples

Let’s use the customers and contacts tables in the [sample database](https://www.oracletutorial.com/getting-started/oracle-sample-database/) for demonstration.



### **A) PL/SQL  SELECT INTO – selecting one column example**

The following example uses a SELECT INTO statement to get the name of a customer based on the customer id, which is the primary key of the customers table.

DECLARE

l\_customer\_name customers.name%TYPE;

BEGIN

-- get name of the customer 100 and assign it to l\_customer\_name

SELECT name INTO l\_customer\_name

FROM customers

WHERE customer\_id = 100;

-- show the customer name

dbms\_output.put\_line( l\_customer\_name );

END;

Code language: SQL (Structured Query Language) (sql)

In this example:

* First, declare a variable l\_customer\_namewhose data type anchors to the name columns of the customers table. This variable will hold the customer name.
* Second, use the SELECT INTO statement to select value from the name column and assign it to the l\_customer\_name variable.
* Third, show the customer name using the dbms\_output.put\_line procedure.

Because the customers table has only one row with customer ID 100, the code block displayed the customer name.

Verizon

Code language: SQL (Structured Query Language) (sql)

If there were no such row, the code block would fail with an unhandled NO\_DATA\_FOUND exception.

### **B) PL/SQL SELECT INTO – selecting a complete row example**

The following example fetches the entire row from the customers table for a specific customer ID:

DECLARE

r\_customer customers%ROWTYPE;

BEGIN

-- get the information of the customer 100

SELECT \* INTO r\_customer

FROM customers

WHERE customer\_id = 100;

-- show the customer info

dbms\_output.put\_line( r\_customer.name || ', website: ' || r\_customer.website );

END;

Code language: SQL (Structured Query Language) (sql)

Here is the output:

Verizon, website: http://www.verizon.com

Code language: JavaScript (javascript)

In this example:

* First, declare a [record](https://www.oracletutorial.com/plsql-tutorial/plsql-record/) based on the row of the customers table. This record will hold the entire row of the customers table.
* Second, select the customer whose id is 100 into the r\_customer record.

# Oracle PL/SQL Stored Procedure

you are going to see the detailed description on how to create and execute the named blocks (procedures and functions).

Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Apart from this, we will cover the major differences between these two subprograms. Also, we are going to discuss the Oracle built-in functions.

In this Oracle Stored Procedure tutorial, you will learn-

## Terminologies in PL/SQL Subprograms

Before we learn about PL/SQL subprograms, we will discuss the various terminologies that are the part of these subprograms. Below are the terminologies that we are going to discuss.

### Parameter:

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

* These parameters should be defined along with the subprograms at the time of creation.
* These parameters are included n the calling statement of these subprograms to interact the values with the subprograms.
* The datatype of the parameter in the subprogram and the calling statement should be same.
* The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as

1. IN Parameter
2. OUT Parameter
3. IN OUT Parameter

### IN Parameter:

* This parameter is used for giving input to the subprograms.
* It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprogram.
* In the calling statement, these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like ‘5\*8’ or ‘a/b’ where ‘a’ and ‘b’ are variables.
* By default, the parameters are of IN type.

### OUT Parameter:

* This parameter is used for getting output from the subprograms.
* It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
* In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

### IN OUT Parameter:

* This parameter is used for both giving input and for getting output from the subprograms.
* It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
* In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter type should be mentioned at the time of creating the subprograms.

### RETURN

RETURN is the keyword that instructs the compiler to switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keyword in the subprogram, the code after this will be skipped.

Normally, parent or main block will call the subprograms, and then the control will shift from those parent block to the called subprograms. RETURN in the subprogram will return the control back to their parent block. In the case of functions RETURN statement also returns the value. The datatype of this value is always mentioned at the time of function declaration. The datatype can be of any valid PL/SQL data type.

## What is Procedure in PL/SQL?

A **Procedure** in PL/SQL is a subprogram unit that consists of a group of PL/SQL statements that can be called by name. Each procedure in PL/SQL has its own unique name by which it can be referred to and called. This subprogram unit in the Oracle database is stored as a database object.

**Note:** Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

Below are the characteristics of Procedure subprogram unit in PL/SQL:

* Procedures are standalone blocks of a program that can be stored in the [database](https://www.guru99.com/introduction-to-database-sql.html).
* Call to these PLSQL procedures can be made by referring to their name, to execute the PL/SQL statements.
* It is mainly used to execute a process in PL/SQL.
* It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
* It contains declaration part (optional), execution part, exception handling part (optional).
* The values can be passed into Oracle procedure or fetched from the procedure through parameters.
* These parameters should be included in the calling statement.
* A Procedure in SQL can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.
* Procedures cannot be called directly from SELECT statements. They can be called from another block or through EXEC keyword.

#### Syntax:

CREATE OR REPLACE PROCEDURE

<procedure\_name>

(

<parameterl IN/OUT <datatype>

..

.

)

[ IS | AS ]

<declaration\_part>

BEGIN

<execution part>

EXCEPTION

<exception handling part>

END;

* CREATE PROCEDURE instructs the compiler to create new procedure in Oracle. Keyword ‘OR REPLACE’ instructs the compile to replace the existing 0procedure (if any) with the current one.
* Procedure name should be unique.
* Keyword ‘IS’ will be used, when the stored procedure in Oracle is nested into some other blocks. If the procedure is standalone then ‘AS’ will be used. Other than this coding standard, both have the same meaning.

**Example1: Creating Procedure and calling it using EXEC**

In this example, we are going to create an Oracle procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

CREATE OR REPLACE PROCEDURE welcome\_msg (

p\_name IN VARCHAR2)

IS

BEGIN

dbms\_output.put\_line (‘Welcome '|| p\_name);

END;

/

**Execution:** EXEC welcome\_msg (‘Guru99’);

**Code Explanation:**

* **Code line 1**: Creating the procedure with name ‘welcome\_msg’ and with one parameter ‘p\_name’ of ‘IN’ type.
* **Code line 4**: Printing the welcome message by concatenating the input name.
* Procedure is compiled successfully.
* **Code line 7**: Calling the procedure using EXEC command with the parameter ‘Guru99’. Procedure is executed, and the message is printed out as “Welcome Guru99”.

Example 2:

CREATE OR REPACE PROCEDURE Sum(a IN number, b IN number) IS

c number;

BEGIN

c := a+b;

dbms\_output.put\_line('Sum of two nos= '|| c);

END Sum;

Procedure created.

Way-1: Exec Sum(10,20);

Example 3:

create or replace procedure abcd(a in number,b in out number) is

2 begin

3 b:=b+a;

4 dbms\_output.put\_line(b);

5\* end abcd;

SQL> /

Procedure created.

SQL> var c number;

SQL> exec :c:=10;

PL/SQL procedure successfully completed.

SQL> exec abcd(5,:c);

15

PL/SQL procedure successfully completed.

SQL> print :c

C

----------

15

Example 4:

1 create or replace procedure abcd(a in out number,b in out number) is

2 begin

3 b:=b+a;

4 a:=a+b;

5\* end abcd;

6 /

Procedure created.

SQL> var s number

SQL> exec :s:=4;

PL/SQL procedure successfully completed.

SQL> var s1 number

SQL> exec :s1:=5;

PL/SQL procedure successfully completed.

SQL> exec abcd(:s,:s1);

PL/SQL procedure successfully completed.

SQL> print :a

SQL> print :s

S

----------

13

SQL> print :s1

S1

9

Example 5:

create or replace procedure abcd(a in out number,b in out number,

c out number ) is

begin

c:=a \*b - a;

end abcd;

SQL> /

Procedure created.

SQL> var s number

SQL> exec :s:=5;

PL/SQL procedure successfully completed.

SQL> var s1 number

SQL> exec :s1:=10;

PL/SQL procedure successfully completed.

SQL> var s2 number

SQL> exec abcd(:s,:s1,:s2);

PL/SQL procedure successfully completed.

SQL> print s2

S2

----------

45

Way-2: using un-names sql block

For calling the procedure created following code will be executed:

set serveroutput on;

DECLARE

x number;

y number;

BEGIN

x := &x;

y := &y;

Sum(x,y);

END;

Enter value for x: 10

Enter value for y: 20

Sum of two nos= 30

PL/SQL procedure successfully created.

Example 4:

1. Table: **create** **table** user(id number(10) **primary** **key**,**name** varchar2(100));

**create** or replace **procedure** "INSERTUSER"

(id IN NUMBER,

1. **name** IN VARCHAR2)
2. **is**
3. **begin**
4. **insert** **into** user **values**(id,**name**);
5. **end**;

Output:

Procedure created.

## PL/SQL program to call procedure

Let's see the code to call above created procedure.

1. **BEGIN**
2. insertuser(101,'Rahul');
3. dbms\_output.put\_line('record inserted successfully');
4. **END**;
5. /

## PL/SQL Drop Procedure

**Syntax for drop procedure**

1. **DROP** **PROCEDURE** procedure\_name;

# **PL/SQL - Functions**

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## 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.

PL/SQL Function Example

Let's see a simple example to **create a function**.

HTML Tutorial

1. **create** or replace **function** adder(n1 in number, n2 in number)
2. **return** number
3. **is**
4. n3 number(8);
5. **begin**
6. n3 :=n1+n2;
7. **return** n3;
8. **end**;
9. /

Now write another program to **call the function**.

1. **DECLARE**
2. n3 number(2);
3. **BEGIN**
4. n3 := adder(11,22);
5. dbms\_output.put\_line('Addition is: ' || n3);
6. **END**;
7. /

### **Example:2**

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

DECLARE

a number;

b number;

c number;

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;

Execution

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

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

END;

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

## Example: 3 PL/SQL function example using table

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 Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

+----+----------+-----+-----------+----------+

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

/

When the above code is executed using the SQL prompt, it will produce the following result −

Function created.

## 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;

/

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

Total no. of Customers: 6

PL/SQL procedure successfully completed.

### **Example**

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

DECLARE

a number;

b number;

c number;

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;

Execution

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

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

END;

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.