Copy code snippet

```
SQL> DECLARE
  2     l_company_name   VARCHAR2;
  3  BEGIN
  4     l_company_name :=
'Oracle Corporation';
  5  END;
  6  /
l_company_name   VARCHAR2;
                    *
ERROR at line 2:
ORA-06550: line 2, column 21:
PLS-00215: String length constraints
must be in range (1 .. 32767)
```

To declare a fixed-length string, use the CHAR data type:

Copy code snippet

```
DECLARE
   l_yes_or_no CHAR(1) := 'Y';
```

With CHAR (unlike with VARCHAR2) 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 of 1. In other words, the two declarations below are identical:

Copy code snippet

```
DECLARE
   l_yes_or_no1 CHAR(1) := 'Y';
   l_yes_or_no2 CHAR := 'Y';
```

If you declare a CHAR variable with a length greater than 1, Oracle Database automatically pads whatever value you assign to that variable with spaces to the maximum length specified.

Finally, to declare a character large object, use the CLOB data type. You do not specify a maximum length; the length is determined automatically by Oracle Database and is based on the database block size. Here is an example:

Copy code snippet

```
DECLARE
   l_lots_of_text CLOB;
```

So, how do you determine which data type to use in your programs? Here are some guidelines:

So, how do you determine which data type to use in your programs? Here are some guidelines:

- If your string might contain more than 32,767 characters, use the CLOB (or NCLOB) data type.
- If the value assigned to a string *always* has a fixed length (such as a US Social Security number, which always has the same format and length, *NNN-NN-NNNN*), use CHAR (or NCHAR).
- Otherwise (and, therefore, most of the time), use the VARCHAR2 data type (or NVARCHAR2, when working with Unicode data).

Using the CHAR data type for anything but strings that always have a fixed number of characters can lead to unexpected and undesirable results. Consider the following block, which mixes variable and fixed-length strings:

Copy code snippet

```
DECLARE
  l_variable VARCHAR2 (10) := 'Logic';
  l_fixed    CHAR (10) := 'Logic';
BEGIN
  IF l_variable = l_fixed
  THEN
   DBMS_OUTPUT.put_line ('Equal');
  ELSE
   DBMS_OUTPUT.put_line ('Not Equal');
  END IF;
END;
```

At first glance, you would expect that the word “Equal” would be displayed after execution. That is not the case. Instead, “Not Equal” is displayed, because the value of l_fixed has been padded to a length of 10 with spaces. Consider the padding demonstrated in the following block; you would expect the block to display “Not Equal”:

Copy code snippet

```
BEGIN
  IF 'Logic' = 'Logic     '
  THEN
    DBMS_OUTPUT.put_line ('Equal');
  ELSE
    DBMS_OUTPUT.put_line ('Not Equal');
  END IF;
END;
```

You should, as a result, be very careful about the use of the CHAR data type, whether as the type of a variable, database column, or parameter.

Once you have declared a variable, you can assign it a value, change its value, and perform operations on the string contained in that variable using string functions and operators.

For the rest of this article, I will focus on the VARCHAR2 data type.

Using built-in functions with strings

Once you assign a string to a variable, you most likely need to analyze the contents of that string, change its value in some way, or combine it with other strings. Oracle Database offers a wide array of built-in functions to help you with all such requirements. Here are some of the most commonly used functions:

Concatenate multiple strings. One of the most basic and frequently needed operations on strings is to combine or concatenate them together. PL/SQL offers two ways to do this:

The CONCAT built-in function

The || (concatenation) operator

The CONCAT function accepts two strings as its arguments and returns those two strings "stuck together." The concatenation operator also concatenates together two strings, but it is easier to use when combining more than two strings, as you can see in this example:

Copy code snippet

```
DECLARE
  l_first  VARCHAR2 (10) := 'Steven';
  l_middle VARCHAR2 (5) := 'Eric';
  l_last   VARCHAR2 (20)
              := 'Feuerstein';
BEGIN
  /* Use the CONCAT function */
  DBMS_OUTPUT.put_line (
     CONCAT ('Steven', 'Feuerstein'));
  /* Use the || operator */
  DBMS_OUTPUT.put_line (
      l_first
      || ' '
      || l_middle
      || ' '
      || l_last);
END;
/
```

The output from this block is the following:

Copy code snippet

```
StevenFeuerstein
Steven Eric Feuerstein
```

In my experience, you rarely encounter the CONCAT function. Instead, the || operator is almost universally used by PL/SQL developers.

universally used by PL/SQL developers.

If either of the strings passed to CONCAT or || is NULL or '' (a zero-length string), both the function and the operator simply return the non-NULL string. If both strings are NULL, NULL is returned.

Change the case of a string. Three built-in functions change the case of characters in a string:

UPPER changes all characters to uppercase.

LOWER changes all characters to lowercase.

INITCAP changes the first character of each word to uppercase (characters are delimited by a white space or nonalphanumeric character).

Listing 1 shows some examples that use these case-changing functions.

Code listing 1: Examples of case-changing functions

Copy code snippet

```
SQL> DECLARE
  2     l_company_name   VARCHAR2 (25) := 'oraCLE corporatION';
  3  BEGIN
  4     DBMS_OUTPUT.put_line (UPPER (l_company_name));
  5     DBMS_OUTPUT.put_line (LOWER (l_company_name));
  6     DBMS_OUTPUT.put_line (INITCAP (l_company_name));
  7  END;
  8  /
ORACLE CORPORATION
oracle corporation
Oracle Corporation
```

Extract part of a string. One of the most commonly utilized built-in functions for strings is SUBSTR, which is used to extract a substring from a string. When calling SUBSTR, you provide the string, the position at which the desired substring starts, and the number of characters in the substring.

Listing 2 shows some examples that use the SUBSTR function.

Code listing 2: Examples of the SUBSTR function

Copy code snippet

```
DECLARE
   l_company_name   VARCHAR2 (6) := 'Oracle';
BEGIN
   /* Retrieve the first character in the string */
   DBMS_OUTPUT.put_line (
             SUBSTR (l_company_name, 1, 1));
   /* Retrieve the last character in the string */
   DBMS_OUTPUT.put_line (
             SUBSTR (l_company_name, -1, 1));
   /* Retrieve three characters,
```

```
            starting from the second position. */
   DBMS_OUTPUT.put_line (
             SUBSTR (l_company_name, 2, 3));
   /* Retrieve the remainder of the string,
            starting from the second position. */
   DBMS_OUTPUT.put_line (
             SUBSTR (l_company_name, 2));
END;
/
```

The output from this block is the following:

Copy code snippet

```
O
e
rac
racle
```

As you can see, with the SUBSTR function you can specify a negative starting position for the substring, in which case Oracle Database counts backward from the end of the string. If you do not provide a third argument—the number of characters in the substring—Oracle Database automatically returns the remainder of the string from the specified position.

Find a string within another string. Use the INSTR function to determine where (and if) a string appears within another string. INSTR accepts as many as four arguments:

- The string to be searched (required).
- The substring of interest (required).
- The starting position of the search (optional). If the value is negative, count from the end of the string. If no value is provided, Oracle Database starts at the beginning of the string; that is, the starting position is 1.
- The Nth occurrence of the substring (optional). If no value is provided, Oracle Database looks for the first occurrence.

Listing 3 shows some examples that use the INSTR function.

Code listing 3: Examples of the INSTR function

Copy code snippet

```
BEGIN
   /* Find the location of the first "e" */
   DBMS_OUTPUT.put_line (
      INSTR ('steven feuerstein', 'e'));
   /* Find the location of the first "e" starting from position 6 */
   DBMS_OUTPUT.put_line (
```

```
      INSTR ('steven feuerstein'
           , 'e'
           , 6));
   /* Find the location of the first "e" starting from the 6th position from
      the end of string and counting to the left. */
   DBMS_OUTPUT.put_line (
      INSTR ('steven feuerstein'
           , 'e'
           , -6));
   /* Find the location of the 3rd "e" starting from the 6th position from
      the end of string. */
   DBMS_OUTPUT.put_line (
      INSTR ('steven feuerstein'
           , 'e'
           , -6
           , 3));
END;
/
```

The output from this block is the following:

Copy code snippet

```
3
9
11
5
```

INSTR is a very flexible and handy utility. It can easily be used to determine whether or not a substring appears *at all* in a string. Here is a Boolean function that does just that:

Copy code snippet

```
CREATE OR REPLACE FUNCTION
is_in_string (
  string_in      IN VARCHAR2
 ,substring_in   IN VARCHAR2)
  RETURN BOOLEAN
IS
BEGIN
  RETURN INSTR (string_in
              , substring_in) > 0;
END is_in_string;
/
```

Pad a string with spaces (or other characters). I warned earlier about using the CHAR data type, because Oracle Database pads your string value with spaces to the maximum length specified in the

declaration.

However, there are times, primarily when generating reports, when you want to put spaces (or other characters) in front of or after the end of your string. For these situations, Oracle Database offers LPAD and RPAD.

When you call these functions, you specify the length to which you want your string padded and with what character or characters. If you do not specify any pad characters, Oracle Database defaults to padding with spaces.

Listing 4 shows some examples that use these LPAD and RPAD padding functions.

Code listing 4: Examples of padding functions

Copy code snippet

```
DECLARE
   l_first   VARCHAR2 (10) := 'Steven';
   l_last    VARCHAR2 (20) := 'Feuerstein';
   l_phone   VARCHAR2 (20) := '773-426-9093';
BEGIN
   /* Indent the subheader by 3 characters */
   DBMS_OUTPUT.put_line ('Header');
   DBMS_OUTPUT.put_line (
      LPAD ('Sub-header', 13, '.'));

   /* Add "123" to the end of the string, until the 20 character is reached.*
   DBMS_OUTPUT.put_line (
      RPAD ('abc', 20, '123'));

   /* Display headers and then values to fit within the columns. */
   DBMS_OUTPUT.put_line (
     /*1234567890x12345678901234567890x*/
      'First Name Last Name            Phone');
   DBMS_OUTPUT.put_line (
         RPAD (l_first, 10)
      || ' '
      || RPAD (l_last, 20)
      || ' '
      || l_phone);
END;
/
```

The output from this block is this:

Copy code snippet

```
Header
...Sub-header
```

```
abc12312312312312312
First Name Last Name            Phone
Steven     Feuerstein           773-426-9093
```

Replace characters in a string. Oracle Database provides a number of functions that allow you to selectively change one or more characters in a string. You might need, for example, to replace all spaces in a string with the HTML equivalent (“”) so the text is displayed properly in a browser. Two functions take care of such needs for you:

REPLACE replaces a set or pattern of characters with another set.

TRANSLATE translates or replaces individual characters.

Listing 5 shows some examples of these two character-replacement built-in functions. Notice that when you are replacing a single character, the effect of REPLACE and TRANSLATE is the same. When replacing multiple characters, REPLACE and TRANSLATE act differently. The call to REPLACE asked that appearances of “abc” be replaced with “123.” If, however, any of the individual characters (a, b, or c) appeared in the string *outside* of this pattern (“abc”), they would not be replaced.

Code listing 5: Examples of character replacement functions

Copy code snippet

```
DECLARE
   l_name   VARCHAR2 (50) := 'Steven Feuerstein';
BEGIN
   /* Replace all e's with the number 2. Since you are replacing a single
      character, you can use either REPLACE or TRANSLATE. */
   DBMS_OUTPUT.put_line (
      REPLACE (l_name, 'e', '2'));
   DBMS_OUTPUT.put_line (
      TRANSLATE (l_name, 'e', '2'));

   /* Replace all instances of "abc" with "123" */
   DBMS_OUTPUT.put_line (
      REPLACE ('abc-a-b-c-abc'
             , 'abc'
             , '123'));
   /* Replace "a" with "1", "b" with "2", "c" with "3". */
   DBMS_OUTPUT.put_line (
      TRANSLATE ('abc-a-b-c-abc'
               , 'abc'
               , '123'));
END;
/
```

The output from this block is the following:

Copy code snippet

```
St2v2n F2u2rst2in
St2v2n F2u2rst2in
123-a-b-c-123
123-1-2-3-123
```

The call to TRANSLATE, however, specified that any occurrence of each of the individual characters be replaced with the character in the third argument *in the same position*.

Generally, you should use REPLACE whenever you need to replace a pattern of characters, while TRANSLATE is best applied to situations in which you need to replace or substitute individual characters in the string.

Remove characters from a string. What LPAD and RPAD giveth, TRIM, LTRIM, and RTRIM taketh away. Use these trim functions to remove characters from either the beginning (left) or end (right) of the string. **Listing 6** shows an example of both RTRIM and LTRIM.

Code listing 6: Examples of LTRIM and RTRIM functions

Copy code snippet

```
DECLARE
   a  VARCHAR2 (40)
      := 'This sentence has too many periods....';
   b  VARCHAR2 (40) := 'The number 1';
BEGIN
   DBMS_OUTPUT.put_line (
      RTRIM (a, '.'));
   DBMS_OUTPUT.put_line (
      LTRIM (
        b
      , 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
        || 'abcdefghijklmnopqrstuvwxyz'));
END;
```

The output from this block is this:

```
This sentence has too many periods
1
```

RTRIM removed all the periods, because the second argument specifies the character (or characters) to trim, in this case, a period. The call to LTRIM demonstrates that you can specify multiple characters to trim. In this case, I asked that all letters and spaces be trimmed from the beginning of string b, and I got what I asked for.

The default behavior of both RTRIM and LTRIM is to trim spaces from the beginning or end of the string. Specifying RTRIM(a) is the same as asking for RTRIM(a,' '). The same goes for LTRIM(a) and LTRIM(a,' ').

The other trimming function is just plain TRIM. TRIM works a bit differently from LTRIM and RTRIM, as

The other trimming function is just plain TRIM. TRIM works a bit differently from LTRIM and RTRIM, as you can see in this block:

Copy code snippet

```
DECLARE
   x   VARCHAR2 (30)
          := '.....Hi there!.....';
BEGIN
   DBMS_OUTPUT.put_line (
      TRIM (LEADING '.' FROM x));
   DBMS_OUTPUT.put_line (
      TRIM (TRAILING '.' FROM x));
   DBMS_OUTPUT.put_line (
      TRIM (BOTH '.' FROM x));

   --The default is to trim
   --from both sides
   DBMS_OUTPUT.put_line (
      TRIM ('.' FROM x));

   --The default trim character
   --is the space:
   DBMS_OUTPUT.put_line (TRIM (x));
END;
```

The output from this block is the following:

Copy code snippet

```
Hi there!.....
     .....Hi there!
    Hi there!
    Hi there!
.....Hi there!.....
```

With TRIM, you can trim from either side or from both sides. However, you can specify only a single character to remove. You cannot, for example, write the following:

```
TRIM(BOTH ',.;' FROM x)
```

If you need to remove more than one character from the front and back of a string, you need to use RTRIM *and* LTRIM:

```
RTRIM(LTRIM(x,',.;'),',.;')
```

You can also use TRANSLATE to remove characters from a string by replacing them with (or "translating" them *into*) NULL. You must, however, take care with how you specify this replacement. Suppose I want to remove all digits (0 through 9) from a string. My first attempt yields the following block:

Copy code snippet

```
BEGIN
   /* Remove all digits (0-9)
      from the string. */
   DBMS_OUTPUT.put_line (
      TRANSLATE ('S1t2e3v4e56n'
                 , '1234567890'
                 , ''));
END;
/
```

When I execute this block, however, nothing (well, a NULL string) is displayed. This happens because if any of the arguments passed to TRANSLATE are NULL (or a zero-length string), the function returns a NULL value.

So all three arguments must be non-NULL, which means that you need to put at the start of the second and third arguments a character that will simply be replaced with itself, as in the following:

Copy code snippet

```
BEGIN
   /* Remove all digits (0-9)
      from the string. */
   DBMS_OUTPUT.put_line (
      TRANSLATE ('S1t2e3v4e56n'
                 , 'A1234567890'
                 , 'A'));
END;
/
```

Now, “A” is replaced with “A” and the remaining characters in the string are replaced with NULL, so the string “Steven” is then displayed.

String information that’s good to know

Beyond awareness of the basic properties of strings in PL/SQL and built-in functions, you can benefit by keeping the following points about long strings and maximum string sizes in mind.

When the string is too long. You must specify a maximum length when you declare a variable based on the VARCHAR2 type. What happens, then, when you try to assign a value to that variable whose length is greater than the maximum? Oracle Database raises the ORA-06502 error, which is also defined in PL/SQL as the VALUE_ERROR exception.

Here is an example of the exception being raised and propagated out of the block unhandled:

Copy code snippet

```
SQL> DECLARE
  2    l_name VARCHAR2(3);
  3  BEGIN
  4    l_name := 'Steven';
  5  END;
  6  /
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value
error: character string buffer too small
ORA-06512: at line 4
```

Here is a rewrite of the same block that traps the VALUE_ERROR exception:

Copy code snippet

```
SQL> DECLARE
  2    l_name   VARCHAR2 (3);
  3  BEGIN
  4    l_name := 'Steven';
  5  EXCEPTION
  6    WHEN VALUE_ERROR
  7    THEN
  8      DBMS_OUTPUT.put_line (
  9         'Value too large!');
 10  END;
 11  /
Value too large!
```

Interestingly, if you try to insert or update a value in a VARCHAR2 column of a database table, Oracle Database raises a *different* error, which you can see below:

Copy code snippet

```
SQL> CREATE TABLE small_varchar2
  2  (
  3     string_value   VARCHAR2 (2)
  4  )
  5  /
Table created.

SQL> BEGIN
  2     INSERT INTO small_varchar2
  3          VALUES ('abc');
  4  END;
  5  /
```

```
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column
"HR"."SMALL_VARCHAR2"."STRING_VALUE"
(actual: 3, maximum: 2)
ORA-06512: at line 2
```

Different maximum sizes. There are a number of differences between SQL and PL/SQL for the maximum sizes for string data types. In PL/SQL, the maximum size for VARCHAR2 is 32,767 bytes, while in SQL the maximum is 4,000 bytes. In PL/SQL, the maximum size for CHAR is 32,767 bytes, while in SQL the maximum is 2,000 bytes.

Therefore, if you need to save a value from a VARCHAR2 variable in the column of a table, you might encounter the ORA-12899 error. If this happens, you have two choices:

- Use SUBSTR to extract no more than 4,000 bytes from the larger string, and save that substring to the table. This option clearly has a drawback: You lose some of your data.
- Change the data type of the column from VARCHAR2 to CLOB. This way, you can save all your data.

In PL/SQL, the maximum size for CLOB is 128 terabytes, while in SQL the maximum is just (4 GB - 1) * DB_BLOCK_SIZE.

There's much more to data than strings

Character data plays a very large role in PL/SQL applications, but those same applications undoubtedly also rely on data of other types, especially numbers and dates. I will cover these data types in the next PL/SQL 101 article.

Dig deeper

See the rest of the series.

Steven Feuerstein
Developer Advocate for PL/SQL

Steven Feuerstein was Oracle Corporation's Developer Advocate for PL/SQL between 2014 and 2021. He is an expert on the Oracle PL/SQL language, having

Show more

Previous Post

Controlling the flow of execution in PL/SQL

Steven Feuerstein | 18 min read

Next Post

Working with Numbers in PL/SQL

Steven Feuerstein | 12 min read