PL/SQL Built-Ins

Putting PL/SQL's Readymade Functionality to Work

By Steven Feuerstein

I believe I am a fairly good PL/SQL programmer. I can churn out hundreds of lines of complicated code that works, more or less, after just a few rounds of revisions. On the other hand, I feel I am also very much open to the possibility that others can and have done better — and that I can learn from them. None of us has all the answers — and some of us have more answers than others.

Let me give you an example. I recently published a book on PL/SQL, *ORACLE PL/SQL Programming* [O'Reilly and Associates, 1995]. It's a big book and I give every impression that I know what I am talking about. A big part of what I talk about is writing concise, high-quality code. In chapter 11 on built-in character functions, I take my readers through the exercise of building a function to count the number of times a sub-string occurs in a string (a function *not* provided by PL/SQL). I end up with two implementations, the shorter of which is shown in Figure 1.

I was quite content with this function — until I received an e-mail message from Kevin Loney, author of *ORACLE DBA Handbook* [Oracle Press, 1994]. He politely complimented me on my book and offered an alternative implementation for numinstr that is simpler, more efficient, and more elegant. As Kevin noted, he came up with this solution before the days of PL/SQL, when all he had to work with was SQL (a set-at-a-time, non-procedural language). In his approach (see Figure 2), he took advantage of the REPLACE built-in function to substitute NULL for any occurrences of the sub-string. He could then compare the size of the original string with the "replaced" string and use that as the basis for his calculation.

I felt a hot wave of embarrassment wash over me for just a moment when I first read Kevin's note. Then I regained sanity. *Of course* there are better ways of doing things. Discovering and sharing these improvements — programming altruism — is one of the finest aspects of our work. And programming humility, the willingness to accept these improvements, can make our lives much easier. If you are open to the ideas of others, then you are also open to the idea of using the work of others. (With permission!) This means that you will spend much less time coding something that is already available, i.e. you will avoid "reinventing the wheel."

In this issue of *Oracle Informant*, I offer a review of the built-in functions of PL/SQL with the goal of avoiding re-inventions of the wheel. A *built-in* is a programming element that has been defined for you within the language. A *function* accepts inputs and returns a single value. The PL/SQL built-in functions can save you an enormous amount of time and mental energy. Your challenge as a PL/SQL developer is two-fold:

1) Get familiar with the built-in functions. You can't reuse this stuff unless you know that it's there.
2) Learn how to fully leverage the capabilities of these built-ins. Many of the functions have additional parameters and usages that make them much more flexible and useful than you might at first think.

This article will help you meet both of these challenges. To become proficient with the built-ins however, you must put them to use and also study more in-depth resources on the topic.

The Families of Built-in Functions

PL/SQL offers dozens of built-in functions. They fall roughly into five families, defined by the type of data returned by the function (see Figure 3).

Within this article, it's not possible to cover all these functions in depth (my book uses four chapters and over 100 pages). Instead, I will list all current functions by family (thereby helping you surmount challenge #1), and then explore some of the more subtle nuances of a few functions (giving you a leg-up on challenge #2).

```
FUNCTION numinstr
   (string_in IN VARCHAR2,
    substring_in IN VARCHAR2,
    match_case_in IN VARCHAR2 := 'IGNORE') RETURN INTEGER
/*
|| Use the last argument to INSTR to find the nth
|| occurrence of substring, where N is incremented with
|| each spin of the loop. If INSTR returns 0 then have
|| one too many in the return_value, so when I RETURN
|| it, I subtract 1 (see code following loop).
*/
IS
  substring_loc NUMBER;
  return_value NUMBER := 1;
BEGIN
  LOOP
    IF UPPER(match_case_in) = 'IGNORE'
    THEN
        substring_loc :=
          INSTR(UPPER(string_in),
                UPPER(substring_in), 1, return_value);
      ELSE
        substring_loc :=
          INSTR(string_in substring_in, 1,
                return_value);
      END IF;

      /* Terminate loop when no more occurrences are
         found. */
      EXIT WHEN substring_loc = 0;

      /* Found match, so add to total and continue. */
      return_value := return_value + 1;
   END LOOP;
   RETURN return_value - 1;

END numinstr;
```

```
FUNCTION numinstr
  (string_in IN VARCHAR2, sub_string_in IN VARCHAR2)
RETURN INTEGER

/*
|| Function returns the number of times a sub-string
|| appears in a string. Idea for this simple, elegant
|| algorithm comes from Kevin Loney.
*/
IS
  return_value INTEGER := NULL;
  temp_string VARCHAR2(1000);
BEGIN
  temp_string := REPLACE (string_in, sub_string_in);

  /* Divide difference of two lengths by length
     of the substring. */
  return_value :=
    (LENGTH(string_in) - NVL (LENGTH(temp_string), 0))
     / LENGTH (sub_string_in);

   RETURN return_value;

END;
```

Figure 1 (Top): Counting the frequency of a sub-string within a string.
Figure 2 (Bottom): The better mousetrap for counting occurrences in a string.

Family	Description
Character Functions	Functions that operate on and return character strings. An example is the SUBSTR function, which returns a portion or sub-string of the original string.
Conversion Functions	Functions that convert data from one data type to another. For example, the TO_DATE function converts a string or number to a date.
Date Functions	Functions that operate on and return dates. An example is the ADD_MONTHS function, which returns a date N months (in the past or future) from the original date.
Numeric Functions	Functions that operate on and return numbers. PL/SQL offers a comprehensive array of standard numeric functions, as you would expect in a procedural language.
Miscellaneous Functions	Every language has got to have them! These functions don't quite fit into a specific category. One example is VSIZE, which returns the number of bytes used to store the specified data.

Figure 3: The five families of PL/SQL built-in functions.

Character Functions

A character function is one that takes one or more character values as parameters and returns either a character value or number value. The Oracle7 Server and PL/SQL provide a number of different character data types, including CHAR, VARCHAR, VARCHAR2, LONG, RAW, and LONG RAW. In PL/SQL, the three families for character data are:

- VARCHAR2 — a variable-length character data type whose data is converted by the RDBMS.
- CHAR — the fixed-length data type.
- RAW — a variable-length data type whose data is not converted by the RDBMS, but instead is left in "raw" form. Examples of RAW and LONG RAW data include binary data such as sound, graphics, etc.

When a character function returns a character value, that value is always of type VARCHAR2 (variable length), with the following two exceptions: UPPER and LOWER. These functions convert to upper- and lower-case, respectively, and return CHAR values (fixed-length) if the strings they are called on to convert are fixed-length, CHAR arguments.

PL/SQL provides a rich set of 23 character functions. These functions allow you to get information about strings and modify the contents of those strings in high-level, powerful ways. Figure 4 shows the 16 core character functions. Additional variations of these functions (such as SUBSTRB and LENGTHB) are specific to National Language Support and Trusted Oracle.

Conversion Functions

Whenever PL/SQL performs an operation involving one or more values, it must first convert the data so that it's the correct data type for the operation. There are two kinds of conversion: explicit and implicit. An *implicit* conversion is performed by PL/SQL as needed to execute a statement. An *explicit* conversion takes place when the programmer uses a built-in conversion function to force the conversion of a value from one data type to another.

If you do not use a conversion function to explicitly convert your data — or if you do use those functions and additional conversion is still needed — PL/SQL attempts to convert the data implicitly to the data types it needs to perform the operation. I recommend that you avoid allowing either the SQL or PL/SQL languages to perform implicit conversions on your behalf. Whenever possible, use a conversion function to guarantee that the appropriate conversion occurs.

Name	Description
ASCII	Returns the ASCII code of a character. Useful in identifying non-printing characters in a string.
CHR	Returns the character associated with the specified collating code. I use this function to add new-line characters — CHR(10) — to strings.
CONCAT	Concatenates two strings into one. PL/SQL also offers the double vertical bars (| |) as a concatenation operator, a more flexible and useful approach to concatenation. I do not use CONCAT.
INITCAP	Sets the first letter of each word to upper-case. All other letters are set to lower-case. Comes in handy for reports, but is not sophisticated enough to know that INITCAP ('MCDONALD') is *not* 'Mcdonald'.
INSTR	Returns the location in a string of the specified sub string. With this handy function you can specify the starting position and Nth appearance of the sub-string. You can even search backwards through the string.
LENGTH	Returns the length of a string. Just remember that LENGTH (NULL) is NULL; it's *not* zero. In fact, LENGTH will never return 0.
LOWER	Converts all letters to lower-case.
LPAD	Pads a string on the left with the specified characters. You can pad with blanks or any other string.
LTRIM	Trims the left side of a string of all specified characters. You can trim blanks (the default) or you can provide a set of characters all of which you want to make disappear.
REPLACE	Replaces a character sequence in a string with a different set of characters. REPLACE performs pattern-matching search and replaces.
RPAD	Pads a string on the right with the specified characters. You can pad with blanks or any other string.
RTRIM	Trims the right side of a string of all specified characters. You can trim blanks (the default) or you can provide a set of characters all of which you want to make disappear.
SOUNDEX	Returns the "soundex" of a string. Uses the Knuth algorithm.
SUBSTR	Returns the specified portion of a string. A very useful function, SUBSTR can also scan from the end of the string to find its starting position.
TRANSLATE	Translates single characters in a string to different characters. It does *not* perform pattern-matching like REPLACE. Instead it simply replaces the nth character in the search list with the nth character in the replace list.
UPPER	Converts all letters in the string to upper-case.

Name	Description
CHARTOROWID	Converts a string to a ROWID, which is an Oracle-specific row identifier.
CONVERT	Converts a string from one character set to another.
HEXTORAW	Converts from hexadecimal to raw format.
RAWTOHEX	Converts from raw value to hexadecimal.
ROWIDTOCHAR	Converts a binary ROWID value to a character string.
TO_CHAR	Converts a number or date to a string. You can also provide a format mask to determine the format of the data as a string.
TO_DATE	Converts a string to a date. You can also provide a format mask to inform TO_DATE of the format of the data as a string, so that it can be interpreted properly.
TO_NUMBER	Converts a string to a number. You can also provide a format mask to inform TO_NUMBER of the format of the data as a string, so it can be interpreted properly.

Figure 4 (Top): The 16 core built-in character functions.
Figure 5 (Bottom): The eight core conversion functions.

Drawbacks of Implicit Conversions

Implicit conversion has a number of drawbacks:

- Each implicit conversion PL/SQL performs represents a loss, however small, in the control you have over your program. You do not expressly perform or direct the performance of that conversion. You assume that the conversion will take place, and have the intended effect. This is always a dangerous thing to do in a program. Newer versions of Oracle may change the way and circumstances under which it performs conversions; your code could then be affected.
- The implicit conversion that PL/SQL performs depends on the context in which the code occurs, so a conversion might occur in one program and not in another even though they seem to be the same. As you'll see, the conversion PL/SQL performs is not necessarily always the one you might expect.
- Implicit conversions can actually degrade performance. A dependence on implicit conversions can result in excessive or improper conversions. For example, the conversion of a column value, instead of a constant, in a SQL statement.
- Your code is easier to read and understand if you explicitly convert data where needed. Such conversions document variances in data types between tables or between code and tables. By removing an assumption and a hidden action from your code, you remove a potential misunderstanding as well.

Explicit conversions help you to avoid unpleasant surprises, maximize performance, and make your code more self-documenting. When you perform an explicit conversion involving dates or numbers (using TO_CHAR, TO_DATE, or TO_NUMBER), you can specify a conversion format mask. PL/SQL uses this mask to either interpret the input value or format the output value. Figure 5 summarizes the PL/SQL conversion functions.

Several of the conversion functions (TO_CHAR, TO_DATE, and TO_NUMBER) use format models to determine the format of the converted data.

Date Functions

Most applications store and manipulate dates and times. Dates are quite complicated. Not only are they highly formatted, but there are myriad rules for determining valid values and valid calculations (leap days and years, national and company holidays, date ranges, etc.). Fortunately, PL/SQL and the Oracle RDBMS provide many ways to handle date information.

PL/SQL provides a true DATE data type that stores both date and time information. Each date value contains the century, year, month, day, hour, minute, and second. The DATE data type does not support the storage of fractions of time less than a second in length. The time itself is stored as the number of seconds past midnight. If you enter a date without a time (most applications do not require the tracking of time), the time portion of the database value defaults to midnight (12:00:00 a.m.).

PL/SQL validates and stores dates that fall in the range January 1, 4712 BC to December 31, 4712 AD. Yet support for a true date data type is only half the battle. You also need a language that can manipulate those dates in a natural and intelligent manner — as dates. PL/SQL offers a set of eight date functions for just this purpose (see Figure 6).

With PL/SQL you'll never have to write a program that calculates the number of days between two dates, nor will you have to write a custom utility to determine the day of the week on which a date falls. This information, and just about anything else you can think of with dates, is immediately available to you with the call of a function. The date functions in PL/SQL all take dates, and in some cases numbers, for arguments and return date values. The only exception is MONTHS_BETWEEN, which returns a number.

Name	Description
ADD_MONTHS	Adds the specified number of months to a date. If you want to shift forward by a year, simply enter ADD_MONTHS (my_date, 12).
LAST_DAY	Returns the last day in the month of the specified date. You no longer have to remember the nursery rhyme, "30 days hath September...".
MONTHS_ BETWEEN	Calculates the number of months between two dates. This function also returns a decimal value for fractional components of months.
NEW_TIME	Returns the date/time value, with the time shifted as requested by the specified time zones.
NEXT_DAY	Returns the date of the first weekday specified that is later than the date. To find the first Sunday in March, you would type NEXT_DAY ('01-MAR-96', 'SUNDAY').
ROUND	Returns the date rounded by the specified format unit. You can round to the nearest year (first day in the year), month, quarter, etc.
SYSDATE	Returns the current date and time in the Oracle Server. The precision for SYSDATE is seconds.
TRUNC	Truncates the specified date of its time portion and according to the format unit provided. Similar to TRUNC, but it simply chops off all but the desired component. For example, TRUNC(my_date) "zeroes out" (to midnight) the time stamp.

Name	Description
ABS	Returns the absolute value of the number.
CEIL	Returns the smallest integer greater than or equal to the specified number.
COS	Returns the trigonometric cosine of the specified angle.
COSH	Returns the hyperbolic cosine of the specified number.
EXP (n)	Returns *e* raised to the nth power, where e = 2.71828183...
FLOOR	Returns the largest integer equal to or less than the specified number.
LN (a)	Returns the natural logarithm of a.
LOG (a, b)	Returns the logarithm, base a, of b.
MOD (a, b)	Returns the remainder of a divided by b.
POWER (a, b)	Returns a raised to the bth power.
ROUND (a, [b])	Returns a rounded to b decimal places.
SIGN (a)	Returns 1 if a is positive, 0 if a is 0 and -1 is a is less than 0.
SIN	Returns the trigonometric sine of the specified angle.
SINH	Returns the hyperbolic sine of the specified number.
SQRT	Returns the square root of the number.
TAN	Returns the trigonometric tangent.
TANH	Returns the hyperbolic tangent of the specified number.
TRUNC (a, [b])	Returns a truncated to b decimal places.

Figure 6 (Top): The eight built-in date functions.
Figure 7 (Bottom): The numeric built-in functions.

Date Arithmetic

PL/SQL also allows you to perform arithmetic operations directly on date variables — a variation of a date "built-in" function. You may add or subtract numbers to a date. To move a date one day in the future, simply add 1 to the date as shown below:

```
hire_date + 1
```

You can even add a fractional value to a date. For example, adding 1/24 to a date adds an hour to the time component of that value. Adding 1/(24*60) adds a single minute to the time component, and so on.

Numeric Functions

Let's be honest. PL/SQL is not a numbers-crunching language. If you have to perform complex, compute-intensive operations on numbers, you might very well be better off building your program in a traditional 3rd-generation language (especially if you need to make use of arrays!). In most cases, however, your needs are not quite so intense, and you'll find that PL/SQL's collection of numeric functions will do the job just fine.

Function	Description
CEIL	Returns the smallest integer which is *greater than* the specified value. This integer is the "ceiling" over your value.
FLOOR	Returns the largest integer which is *less than* the specified value. This integer is the "floor" under your value.
ROUND	Performs rounding on a number. You can round with a positive number of decimal places (the number of digits to the *right* of the decimal point) and also with a negative number of decimal places (the number of digits to the *left* of the decimal point).
TRUNC	Truncates a number to the specified number of decimal places. TRUNC simply discards all values beyond the decimal places provided in the call.

Function	1.75	1.3	55.56	55.56	Input
	0	0	1	-1	# of decimal places
ROUND	2	1	55.6	60	
TRUNC	1	1	55.5	50	
FLOOR	1	1	55	55	
CEIL	2	2	56	56	

Figure 8 (Top): PL/SQL's rounding and summary functions.
Figure 9 (Bottom): Impact of rounding and truncating functions.

Figure 7 shows the set of supported numeric functions. There should not be any big surprises. For those of you in need of trigonometric functions, however, be happy. For many years, Oracle Corporation did not support such functions in the database. Ingres, on the other hand, coming out of an academic environment, did cosines and tangents quite gleefully, a competitive advantage highlighted *ad nauseum* by Ingres salespersons. Remember Ingres?

Rounding and Truncation with PL/SQL

There are four numeric functions that perform a variety of rounding and truncation actions: CEIL, FLOOR, ROUND, and TRUNC (see Figure 8). It's easy to get confused about which of the functions to use in a particular situation. Figure 9 illustrates the impact of the functions for different values and decimal place rounding.

Miscellaneous Functions

Some functions cut across many data types or do not apply to a data type at all. I lump these together into the "miscellaneous" category. You should not conclude, however, that a miscellaneous function is a marginal, rarely useful function. Anything and everything offered by Oracle Corporation has its use. The SQLCODE and SQLERRM functions shown in Figure 10, for example, give you information about the current error in PL/SQL processing that would otherwise be unavailable.

Leveraging the Built-ins

As I noted above, it's one thing to become familiar with what is out there and quite another to take full advantage of all the nuances of the built-in functions. I offer more detailed presentations of two of the

Name	Description
DUMP	Returns a string containing a "dump" of the specified expression. This dump includes the data type, length in bytes, and internal representation.
GREATEST	Returns the greatest of the specified list of values. One of the really nice things about GREATEST is that there is no pre-determined limitation to the number of values you can pass to it.
LEAST	Returns the least of the specified list of values. As with GREATEST, there is no pre-determined limitation to the number of values you can pass to LEAST.
NVL	Returns a substitution value if the argument is NULL. Remember, in Oracle a NULL value is not a blank or a zero or FALSE. It means "the absence of value" and must be handled differently from known values.
SQLCODE	Returns the number of the Oracle error for the most recent internal exception.
SQLERRM	Returns the error message associated with the error number returned by SQLCODE or with another, explicitly specified error number.
UID	Returns the User ID (a unique integer) of the current Oracle session.
USER	Returns the name of the current Oracle user.
USERENV	Returns a string containing information about the current session, including entry ID, language, session ID, and terminal.
VSIZE	Returns the number of bytes in the internal representation of the specified value.

Figure 10: The miscellaneous built-in functions.

character functions, INSTR and SUBSTR, to give you an idea of the kind of flexibility and power the PL/SQL built-ins can offer.

The INSTR function searches a string to find a match for the sub-string and, if found, returns the position, in the source string, of the first character of that sub-string. If there is no match, then INSTR returns 0.

The specification of the INSTR function is:

```
FUNCTION INSTR
   (string1 IN VARCHAR2,
    string2 IN VARCHAR2
   [, start_position IN NUMBER := 1
   [, nth_appearance IN NUMBER := 1]])
RETURN NUMBER
```

where `string1` is the string searched by INSTR for the position in which the `nth_appearance` of `string2` is found. The `start_position` parameter is the position in the string where the search will start. It's optional and defaults to 1 (the beginning of `string1`). The `nth_appearance` parameter is also optional and also defaults to 1.

Both the `start_position` and `nth_appearance` parameters can be literals, such as 5 or 157, variables, or complex expressions, as follows:

```
INSTR (company_name, 'INC', (last_location + 5) * 10)
```

If `start_position` is negative, then INSTR counts back `start_position` number of characters from the *end* of the string and then searches from that point towards the beginning of the string for the nth match. Figure 11 illustrates the two directions in which INSTR searches, depending on the sign of the `start_position` parameter.

I have found INSTR to be a very handy function — especially when used to the fullest extent possible. Most programmers make use of (and are even only aware of) only the first two parameters. Use

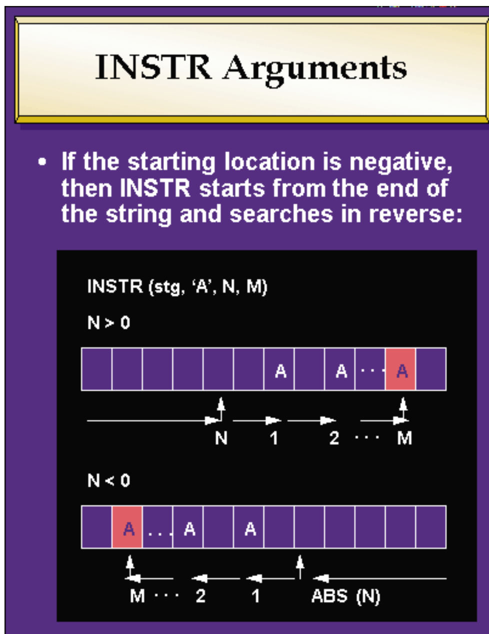


Figure 11: Forward and reverse searches with INSTR.

INSTR to search from the end of the string? Search for the nth appearance, as opposed to just the first appearance? "Wow!" many programmers would say, "I didn't know it could do that." Take the time to get familiar with INSTR and use *all* of its power.

INSTR Examples

In the examples below, you will see all four parameters used in all their permutations. As you write your programs, keep in mind the different ways in which INSTR can be used to extract information from a string. INSTR can greatly simplify the code you write to parse and analyze character data.

1) Find the first occurrence of `archie` in `bug-or-tv-character?archie`:

```
INSTR ('bug-or-tv-character?archie', 'archie') ==> 21
```

The starting position and the nth appearance both defaulted to 1.

2) Find the first occurrence of `archie` in the following string starting from position 14:

```
INSTR ('bug-or-tv-character?archie', 'archie', 14) ==> 21
```

In this example, I specified a starting position, which overrides the default of 1; the answer is still the same though. No matter where you start your search, the character position returned by INSTR is always calculated from the beginning of the string.

3) Find the second occurrence of `a` in `bug-or-tv-character?archie`:

```
INSTR ('bug-or-tv-character?archie', 'a', 1, 2) ==> 15
```

The second `a` in this string is the second `a` in `character`, which is in the 15th position in the string.

4) Find the *last* occurrence of `ar` in `bug-or-tv-character?archie`:

```
INSTR ('bug-or-tv-character?archie', 'ar', -1) ==> 21
```

Were you thinking that the answer might be 6? Remember that the character position returned by INSTR is always calculated from the leftmost character of the string being position 1.

The easiest way to find the last of anything in a string is to specify a negative number for the starting position. I did not have to specify the nth appearance (leaving me with a default value of 1), since the last occurrence is also the first when searching backwards.

5) Find the *second-to-last* occurrence of `a` in `bug-or-tv-character?archie`:

```
INSTR ('bug-or-tv-character?archie', 'a', -1, 2) ==> 15
```

No surprises here. Counting from the back of the string, INSTR passes over the `a` in `archie`, since that is the *last* occurrence, and searches for the next occurrence. Again, the character position is counted from the leftmost character, not the rightmost character, in the string.

6) Find the position of the letter `t` closest to (but not past) the question mark in the string, `bug-or-tv-character?archie tophat`:

```
INSTR ('bug-or-tv-character?archie tophat', 't', -14) ==> 17
```

I needed to find the `t` just before the question mark. The phrase "just before" indicates to me that I should search backwards from the question mark for the first occurrence.

I therefore counted through the characters and determined that the question mark appears at the 20th position. I specified -14 as the starting position so that INSTR would search backwards right from the question mark.

What? Did I hear you mutter that I cheated? That if I could count through the string to find the question mark, I could just as well count through the string to find the closest `t`? I knew that I couldn't slip something like that by my readers. So check out the "PL/SQL Challenge" on page 48. You can help me come up with a more general solution!

The SUBSTR Function

The SUBSTR (pronounced SUB-STRing) function is one of the most useful and commonly used character functions. It allows you to extract a portion or subset of contiguous (connected) characters from a string. The sub-string is specified by starting position and a length. The specification for the SUBSTR function is:

```
FUNCTION SUBSTR
   (string_in IN VARCHAR2,
    start_position_in IN NUMBER
    [, substr_length_in IN NUMBER])
RETURN VARCHAR2
```

where the arguments are used as follows:

- `string_in`: the source string
- `start_position_in`: the starting position of the sub-string in `string_in`
- `substr_length_in`: the length of the sub-string desired (the number of characters to be returned in the sub-string).

The last parameter, `substr_length_in`, is optional. If you do not specify a sub-string length, then SUBSTR returns all the characters to the end of `string_in` (from the starting position specified).

The start position cannot be zero. If the start position is less than zero, then the sub-string is retrieved from the back of the string. SUBSTR counts backwards `substr_length_in` number of characters from the *end* of `string_in`. In this case, however, the characters that are extracted are still to the *right* of the starting position. See Figure 12 for an illustration of how the different arguments are used by SUBSTR.

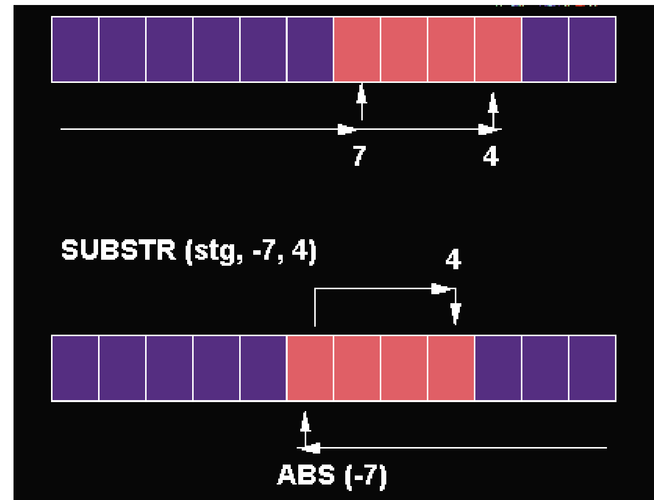


Figure 12: How arguments are used by SUBSTR.

The `substr_length_in` argument must be greater than zero or else SUBSTR returns NULL.

You will find that in practice SUBSTR is forgiving. Even if you violate the rules for the values of the starting position and the number of characters to be sub-stringed, SUBSTR will not generate errors. Instead, for the most part, it will return NULL — or the entire string — as its answer.

SUBSTR Examples

1) If the absolute value of the starting position exceeds the length of the input string, return NULL:

```
SUBSTR ('now_or_never', 200) ==> NULL
SUBSTR ('now_or_never', -200) ==> NULL
```

2) If starting position is 0, SUBSTR acts as though the starting position was actually 1.

```
SUBSTR ('now_or_never', 0, 3) ==> 'now'
SUBSTR ('now_or_never', 0) ==> 'now_or_never'
```

3) If the sub-string length is less than or equal to zero, return NULL:

```
SUBSTR ('now_or_never', 5, -2) ==> NULL
SUBSTR ('now_or_never', 1, 0) ==> NULL
```

4) Return the last character in a string:

```
SUBSTR ('Another sample string', -1) ==> 'g'
```

This is the cleanest way to get the single last character. A more "direct," but cumbersome, approach is:

```
SUBSTR
    ('Sample string', LENGTH ('Sample string'), 1) ==> 'g'
```

In other words, calculate the LENGTH of the string and the one character from the string that starts at that last position. No thanks.

5) Use SUBSTR to extract the portion of a string between the specified starting and ending points. I run into this requirement all the time. SUBSTR requires a starting position and the number of characters to pull out. Often, however, I have only the starting position and the ending position — and I then have to compute the number of characters in between. Is it just the difference between the end and start positions? Is it one more or one less than that? Invariably, I get it wrong the first time and have to scribble a little example on scrap paper to prove the formula to myself.

So to save all of you the trouble, I offer a tiny function below, called *betwnstr* (for "BETWeeN STRing"). This function encapsulates the calculation you must perform to come up with the number of characters between start and end positions, which is:

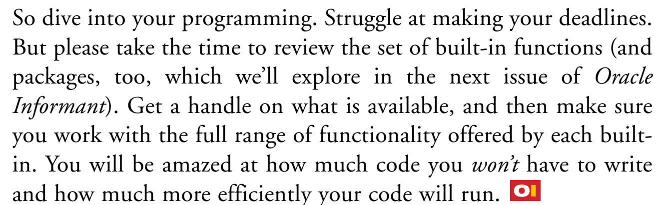
```
end_position - start_position - 1

FUNCTION betwnstr
   (string_in IN VARCHAR2,
    start_in IN INTEGER,
    end_in IN INTEGER)
   RETURN VARCHAR2
IS
BEGIN
   RETURN SUBSTR (string_in, start_in, end_in - start_in + 1);
END;
```

Hopefully these examples will inspire you to fully explore the other PL/SQL built-in functions. There's a lot to learn, but it's all there to help you get your job done!

Power Programming with PL/SQL Built-ins

To become truly proficient with PL/SQL, to become in effect a "power programmer," you must fully leverage everything that PL/SQL has to offer. This includes becoming familiar with the block structure, exception handling architecture and, of course, the wonderful packages of PL/SQL.

When it comes right down to meeting your business requirements, however, there is nothing quite like knowing your built-ins. I would be hard-pressed to complete any complex program without relying heavily on pre-defined programs in the base PL/SQL STANDARD package.



So dive into your programming. Struggle at making your deadlines. But please take the time to review the set of built-in functions (and packages, too, which we'll explore in the next issue of *Oracle Informant*). Get a handle on what is available, and then make sure you work with the full range of functionality offered by each built-in. You will be amazed at how much code you *won't* have to write and how much more efficiently your code will run. OI

Steven Feuerstein is the author of *ORACLE PL/SQL Programming* [O'Reilly and Associates, 1995], the first independent reference and user's guide for the PL/SQL language. Director of the Oracle Practice of SSC, a systems management consulting firm based in Chicago, Steven offers training and consulting for Oracle-based application development. His writings on PL/SQL are published regularly in magazines and user group newsletters. He has developed Oracle-based applications for over nine years, five of those with Oracle Corporation. He can be reached through CompuServe at 72053,441.

Available December 1996

The Must Have Reference Source For The Serious Oracle® Developer

The Entire Text of all Technical Articles Appearing in Oracle® Informant® in 1996

The Oracle® Informant® Works 1996 CD-ROM will include:

- All Technical Articles
- Text and Keyword Search Capability
- Improved Speed and Performance
- All Supporting Code and Sample Files
- 16-Page Color Booklet
- Third-Party Add-In Product Demos
- CompuServe Starter Kit with $15 Usage Credit.

A $130 Value
Available Now for only

$49.95

California residents add 7¼% Sales Tax, plus $5 shipping & handling for US orders.
(International orders add $15 shipping & handling)

Call Now Toll Free
1-800-88-INFORM

1-800-884-6367 Ask for offer # WEB
To order by mail,
send check or Money Order to:
Informant Communications Group, Inc.
ATTN: Works CD offer # WEB
10519 E. Stockton Blvd, Suite 142
Elk Grove, CA 95624-9704
or Fax your order to 916-686-8497

Get Informed!

Subscribe to Oracle Informant, The Independent Monthly Guide to Oracle Development.

Order Now and Get One Issue FREE!

For a limited time you can receive the first issue FREE plus 12 additional issues for only $49.95 That's nearly 25% off the yearly cover price!

Each big issue of *Oracle Informant* is packed with Oracle tips, techniques , news, and more!

- Client/Server Development
- Using Developer/2000™
- Tuning Oracle7
- PL/SQL Techniques
- Advanced Oracle Topics
- Distributed Managment
- Product Reviews
- Book Reviews
- News from the Oracle Community
- Oracle User Group Information

Order Now!

To order, mail or fax the adjoining form or call (916) 686-6610 Fax: (916) 686-8497

International rates

Magazine-only
$54.95/year to Canada
$74.95/year to Mexico
$79.95/year to all other countries

Magazine AND Companion Disk Subscriptions
$124.95/year to Canada
$154.95/year to Mexico
$179.95 to all other countries

California Residents add 7¼% sales tax on disk subscription

YES!, I want to sharpen my Oracle skills. I've checked the subscription plan I'm interested in below

☐ Magazine-Only Subscription Plan...
13 Issues Including One Bonus Issue at $49.95.

☐ Magazine AND Companion Disk Subscription Plan...
13 Issues and Disks Including One Bonus Issue and Disk at $119.95
The Oracle Informant Companion Disk contains source code, support files, examples, utilities, samples, and more!

☐ Oracle Informant Works 1996 CD-ROM $49.95 (Available December 1996)
US residents add $5 shipping and handling. International customers add $15 shipping and handling.

Name ______________________

Company ______________________

Address ______________________

City ____________ State ______ Zip Code ______

Country ____________ Phone ____________

FAX ____________ E-Mail ____________

Payment Method...

☐ Check (payable to Informant Communications Group) ☐ Purchase Order-- Provide Number ______

☐ Visa ☐ Mastercard ☐ American Express Card Number ______________

Expiration Date ______ Signature ______________

WEB

Oracle and its products are trademarks of Oracle Corporation