**CHAPTER 4  
Using Simple Functions**

In this chapter, you’ll learn about some of the Oracle database’s built-in functions. A function accepts zero or more input parameters and returns an output parameter. There are two main types of functions you can use in an Oracle database:

Image **Single-row functions** operate on one row at a time and return one row of output for each input row. An example single-row function is CONCAT (*x*, *y*), which appends *y* to *x* and returns the resulting string.

Image **Aggregate functions** operate on multiple rows at the same time and return one row of output. An example aggregate function is AVG(*x*), which returns the average of *x* where *x* may be a column or, more generally, any expression.

You’ll learn about single-row functions first, followed by aggregate functions. You’ll see more advanced functions as you progress through this book.

**USING SINGLE-ROW FUNCTIONS**

A single-row function operates on one row at a time and returns one row of output for each row. There are five main types of single-row functions:

Image **Character functions** manipulate strings of characters.

Image **Numeric functions** perform calculations.

Image **Conversion functions** convert a value from one database type to another.

Image **Date functions** process dates and times.

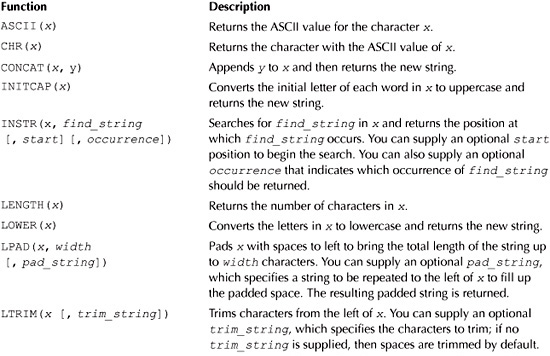
Image **Regular expression functions** use regular expressions to search data. These functions were introduced in Oracle Database 10*g* and are expanded in 11*g*.

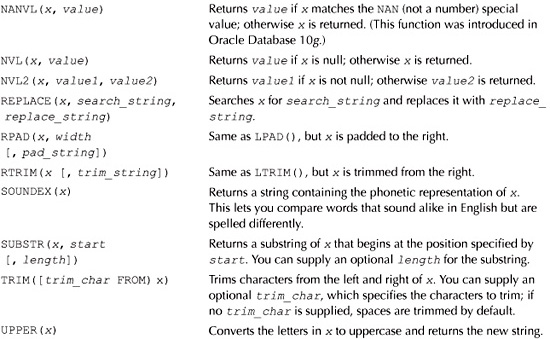
You’ll learn about character functions first, followed by numeric functions, conversion functions, and regular expression functions. You’ll learn about date functions in the next chapter.

**Character Functions**

Character functions accept character input, which may come from a column in a table or, more generally, from any expression. This input is processed and a result returned. An example character function is UPPER(), which converts the letters in an input string to uppercase and returns the new string. Another example is NVL(), which converts a null value to another value. In [Table 4-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-1), which shows some of the character functions, and in all the syntax definitions that follow, *x* and *y* may represent columns from a table or, more generally, any valid expressions.

You’ll learn more about some of the functions shown in [Table 4-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-1) in the following sections.





**TABLE 4-1** *Character Functions*

**ASCII** () **and CHR**()

You use ASCII(*x*) to get the ASCII value for the character *x*. You use CHR(*x*) to get the character with the ASCII value of *x*.

The following query gets the ASCII value of a, A, z, Z, 0, and 9 using ASCII():

**SELECT ASCII('a'), ASCII('A'), ASCII('z'), ASCII('Z'), ASCII(0), ASCII(9)**

**FROM dual;**

ASCII('A') ASCII('A') ASCII('Z') ASCII('Z') ASCII(0) ASCII(9)

---------- ---------- ---------- ---------- ---------- ----------

97 65 122 90 48 57

Image

**NOTE**  
*The* dual *table is used in this query. As you saw in*[*Chapter 2*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02)*, the* dual *table contains a single row through which you may perform queries that don’t go against a particular table*.

The following query gets the characters with the ASCII values of 97, 65, 122, 90, 48, and 57 using CHR():

**SELECT CHR(97), CHR(65), CHR(122), CHR(90), CHR(48), CHR(57)**

**FROM dual;**

C C C C C C

- - - - - -

a A z Z 0 9

Notice the characters returned from CHR() in this query are the same as those passed to ASCII() in the previous query. This shows that CHR() and ASCII() have the opposite effect.

**CONCAT()**

You use CONCAT(*x*, *y*) to append *y* to *x* and then return the new string.

The following query appends last\_name to first\_name using CONCAT():

**SELECT CONCAT(first\_name, last\_name)**

**FROM customers;**

CONCAT(FIRST\_NAME, LA

--------------------

JohnBrown

CynthiaGreen

SteveWhite

GailBlack

DoreenBlue

Image

**NOTE**  
CONCAT() *is the same as the* || *operator you saw in*[*Chapter 2*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02).

**INITCAP()**

You use INITCAP(*x*) to convert the initial letter of each word in *x* to uppercase.

The following query retrieves the product\_id and description columns from the products table, then uses INITCAP() to convert the first letter of each word in description to uppercase:

**SELECT product\_id, INITCAP(description)**

**FROM products**

**WHERE product\_id < 4;**

PRODUCT\_ID INITCAP(DESCRIPTION)

---------- -------------------------------

1 A Description Of Modern Science

2 Introduction To Chemistry

3 A Star Explodes

**INSTR()**

You use INSTR(*x*, *find\_string* [, *start*] [, *occurrence*]) to search for *find\_string* in *x*. INSTR() returns the position at which *find\_string* occurs. You can supply an optional *start* position to begin the search. You can also supply an optional *occurrence* that indicates which occurrence of *find\_string* should be returned.

The following query gets the position where the string Science occurs in the name column for product #1:

**SELECT name, INSTR(name, ’Science')**

**FROM products**

**WHERE product\_id = 1;**

NAME INSTR(NAME, ’SCIENCE')

------------------------------ ---------------------

Modern Science 8

The next query displays the position where the second occurrence of the e character occurs, starting from the beginning of the product name:

**SELECT name, INSTR(name, 'e', 1, 2)**

**FROM products**

**WHERE product\_id = 1;**

NAME INSTR(NAME,'E',1,2)

------------------------------ -------------------

Modern Science 11

Notice the second e in Modern Science is the eleventh character.

You can also use dates with character functions. The following query gets the position where the string JANoccurs in the dob column for customer #1:

**SELECT customer\_id, dob, INSTR(dob, 'JAN')**

**FROM customers**

**WHERE customer\_id = 1;**

CUSTOMER\_ID DOB INSTR(DOB,'JAN')

----------- --------- ----------------

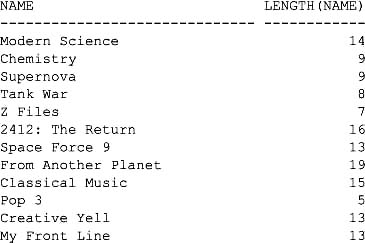
1 01-JAN-65 4

**LENGTH()**

You use LENGTH(*x*) to get the number of characters in *x*. The following query gets the length of the strings in the name column of the products table using LENGTH():

**SELECT name, LENGTH(name)**

**FROM products;**



The next query gets the total number of characters that make up the product price; notice that the decimal point (.) is counted in the number of price characters:

**SELECT price, LENGTH(price)**

**FROM products**

**WHERE product\_id < 3;**

PRICE LENGTH(PRICE)

---------- -------------

19.95 5

30 2

**LOWER() and UPPER()**

You use LOWER(*x*) to convert the letters in *x* to lowercase. Similarly, you use UPPER(*x*) to convert the letters in *x* to uppercase.

The following query converts the strings in the first\_name column to uppercase using the UPPER() function and the strings in the last\_name column to lowercase using the LOWER() function:

**SELECT UPPER(first\_name), LOWER(last\_name)**

**FROM customers;**

UPPER(FIRS LOWER(LAST

---------- ----------

JOHN brown

CYNTHIA green

STEVE white

GAIL black

DOREEN blue

**LPAD() and RPAD()**

You use LPAD (*x*, *width* [, *pad\_string*]) to pad *x* with spaces to the left to bring the total length of the string up to *width* characters. You can supply an optional *pad\_string*, which specifies a string to be repeated to the left of *x* to fill up the padded space. The resulting padded string is then returned. Similarly, you use RPAD(*x*, *width* [, *pad\_string*]) to pad *x* with strings to the right.

The following query retrieves the name and price columns from the products table. The name column is right-padded using RPAD() to a length of 30 characters, with periods filling up the padded space. The pricecolumn is left-padded using LPAD() to a length of 8, with the string \*+ filling up the padded space.

**SELECT RPAD(name, 30, '.'), LPAD(price, 8, '\*+')**

**FROM products**

**WHERE product\_id < 4;**

RPAD(NAME,30,'.') LPAD(PRI

------------------------------ --------

Modern Science……………. \*+\*19.95

Chemistry………………… \*+\*+\*+30

Supernova………………… \*+\*25.99

Image

**NOTE**  
*This example shows that character functions can use numbers. Specifically, the* price *column in the example contains a number that was left-padded by* LPAD().

**LTRIM(), RTRIM(), and TRIM()**

You use LTRIM(*x* [, *trim\_string*]) to trim characters from the left of *x*. You can supply an optional *trim\_string*, which specifies the characters to trim; if no *trim\_string* is supplied; spaces are trimmed by default. Similarly, you use RTRIM() to trim characters from the right of *x*. You use TRIM() to trim characters from the left and right of *x*. The following query uses these three functions:

**SELECT**

**LTRIM(' Hello Gail Seymour!')**,

**RTRIM('Hi Doreen Oakley!abcabc', 'abc')**,

**TRIM('0' FROM '000Hey Steve Button!00000')**

**FROM dual;**

LTRIM('HELLOGAILSEY RTRIM('HIDOREENOA TRIM('0'FROM'000H

------------------- ----------------- -----------------

Hello Gail Seymour! Hi Doreen Oakley! Hey Steve Button!

**NVL()**

You use NVL() to convert a null value to another value. NVL(*x, value*) returns *value* if *x* is null; otherwise *x* is returned.

The following query retrieves the customer\_id and phone columns from the customers table. Null values in the phone column are converted to the string Unknown Phone Number by NVL():

**SELECT customer\_id, NVL(phone, 'Unknown Phone Number')**

**FROM customers;**

CUSTOMER\_ID NVL(PHONE,'UNKNOWNPH

----------- --------------------

1 800-555-1211

2 800-555-1212

3 800-555-1213

4 800-555-1214

5 Unknown Phone Number

The phone column for customer #5 is converted to Unknown Phone Number because the phone column is null for that row.

**NVL2()**

NVL2(*x*, *value1*, *value2*) returns *value1* if *x* is not null; otherwise *value2* is returned.

The following query retrieves the customer\_id and phone columns from the customers table. Non-null values in the phone column are converted to the string Known, and null values are converted to Unknown:

**SELECT customer\_id, NVL2(phone, 'Known', 'Unknown')**

**FROM customers;**

CUSTOMER\_ID NVL2(PH

----------- -------

1 Known

2 Known

3 Known

4 Known

5 Unknown

Notice that the phone column values are converted to Known for customers #1 through #4 because the phonecolumn values for those rows are not null. For customer #5 the phone column value is converted to Unknownbecause the phone column is null for that row.

**REPLACE()**

You use REPLACE(*x*, *search\_string*, *replace\_string*) to search *x* for *search\_string* and replace it with *replace\_string*.

The following example retrieves the name column from the products table for product #1 (whose name is Modern Science) and replaces the string Science with Physics using REPLACE():

**SELECT REPLACE(name, ’Science', 'Physics')**

**FROM products**

**WHERE product\_id = 1;**

REPLACE(NAME,’SCIENCE','PHYSICS')

---------------------------------

Modern Physics

Image

**NOTE**  
REPLACE() *doesn’t modify the actual row in the database; only the row returned by the function is modified*.

**SOUNDEX()**

You use SOUNDEX(*x*) to get a string containing the phonetic representation of *x*. This lets you compare words that sound alike in English but are spelled differently.

The following query retrieves the last\_name column from the customers table where last\_name sounds like "whyte":

**SELECT last\_name**

**FROM customers**

**WHERE SOUNDEX(last\_name) = SOUNDEX('whyte');**

LAST\_NAME

----------

White

The next query gets last names that sound like "bloo":

**SELECT last\_name**

**FROM customers**

**WHERE SOUNDEX(last\_name) = SOUNDEX('bloo');**

LAST\_NAME

----------

Blue

**SUBSTR()**

You use SUBSTR(*x*, *start* [, *length*]) to return a substring of *x* that begins at the position specified by *start*. You can also provide an optional *length* for the substring.

The following query uses SUBSTR() to get the 7-character substring starting at position 2 from the namecolumn of the products table:

**SELECT SUBSTR(name, 2, 7)**

**FROM products**

**WHERE product\_id < 4;**

SUBSTR(

-------

odern S

hemistr

upernov

**Using Expressions with Functions**

You’re not limited to using columns in functions: you can supply any valid expression that evaluates to a string. The following query uses the SUBSTR() function to get the substring little from the string Mary had a little lamb:

**SELECT SUBSTR('Mary had a little lamb', 12, 6)**

**FROM dual;**

SUBSTR

------

little

**Combining Functions**

You can use any valid combination of functions in an SQL statement. The following query combines the UPPER() and SUBSTR() functions; notice that the output from SUBSTR() is passed to UPPER():

**SELECT name, UPPER(SUBSTR(name, 2, 8))**

**FROM products**

**WHERE product\_id < 4;**

NAME UPPER(SU

------------------------------ --------

Modern Science ODERN SC

Chemistry HEMISTRY

Supernova UPERNOVA

Image

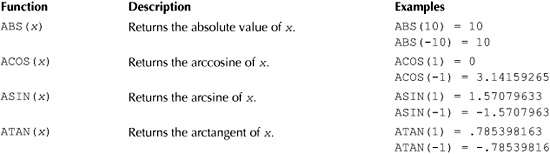
**NOTE**  
*This ability to combine functions is not limited to character functions. Any valid combination of functions will work*.

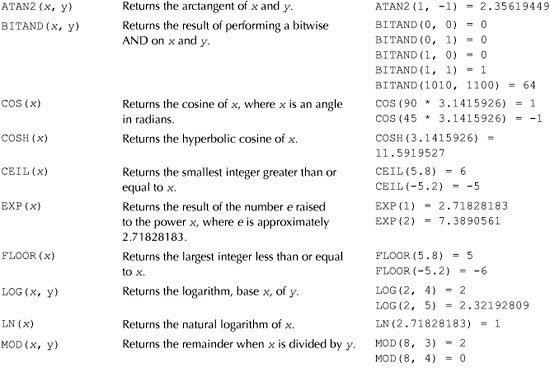
**Numeric Functions**

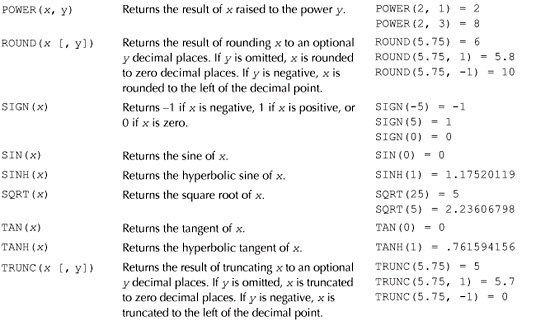
You use the numeric functions to perform calculations. These functions accept an input number, which may come from a numeric column or any expression that evaluates to a number. A calculation is then performed and a number returned. An example of a numeric function is SQRT(*x*), which returns the square root of *x*.

[Table 4-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-2) shows some of the numeric functions.

You’ll learn more about some of the functions shown in [Table 4-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-2) in the following sections.







**TABLE 4-2** *Numeric Functions*

**ABS()**

You use ABS(*x*) to get the absolute value of *x*. The absolute value of a number is that number without any positive or negative sign. The following query gets the absolute value of 10 and −10:

**SELECT ABS(10), ABS(-10)**

**FROM dual;**

BS(10) ABS(−10)

---------- ----------

10 10

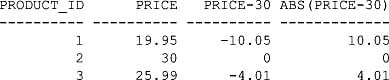
The absolute value of 10 is 10. The absolute value of −10 is 10.

Of course, the parameters that are input to any of the number functions don’t have to be literal numbers. The input may also be a numeric column from a table or, more generally, any valid expression. The following query gets the absolute value of subtracting 30 from the price column from the products table for the first three products:

**SELECT product\_id, price, price - 30, ABS(price - 30)**

**FROM products**

**WHERE product\_id < 4 ;**



**CEIL()**

You use CEIL(*x*) to get the smallest integer greater than or equal to *x*. The following query uses CEIL() to get the absolute values of 5.8 and −5.2:

**SELECT CEIL(5.8), CEIL(-5.2)**

**FROM dual;**

CEIL(5.8) CEIL(-5.2)

---------- ----------

6 −5

The ceiling for 5.8 is 6, because 6 is the smallest integer greater than 5.8. The ceiling for −5.2 is −5, because −5.2 is negative, and the smallest integer greater than this is −5.

**FLOOR()**

You use FLOOR(*x*) to get the largest integer less than or equal to *x*. The following query uses FLOOR() to get the absolute value of 5.8 and −5.2:

**SELECT FLOOR(5.8), FLOOR(−5.2)**

**FROM dual;**

FLOOR(5.8) FLOOR(-5.2)

---------- -----------

5 −6

The floor for 5.8 is 5; because 5 is the largest integer less than 5.8. The floor for −5.2 is −6, because −5.2 is negative, and the largest integer less than this is −6.

**MOD()**

You use MOD(*x*, *y*) to get the remainder when *x* is divided by *y*. The following query uses MOD() to get the remainder when 8 is divided by 3 and 4:

**SELECT MOD(8, 3), MOD(8, 4)**

**FROM dual;**

MOD(8,3) MOD(8,4)

---------- ----------

2 0

The remainder when 8 is divided by 3 is 2: 3 goes into 8 twice, leaving 2 left over—the remainder. The remainder when 8 is divided by 4 is 0: 4 goes into 8 twice, leaving nothing left over.

**POWER()**

You use POWER(*x*, *y*) to get the result of *x* raised to the power *y*. The following query uses POWER() to get 2 raised to the power 1 and 3:

**SELECT POWER(2, 1), POWER(2, 3)**

**FROM dual;**

POWER(2,1) POWER(2,3)

---------- ----------

2 8

When 2 is raised to the power 1, which is equivalent to 2\*1, the result is 2; 2 raised to the power 3 is equivalent to 2\*2\*2, the result of which is 8.

**ROUND()**

You use ROUND(*x*, [*y*]) to get the result of rounding *x* to an optional *y* decimal places. If *y* is omitted, *x* is rounded to zero decimal places. If *y* is negative, *x* is rounded to the left of the decimal point.

The following query uses ROUND() to get the result of rounding 5.75 to zero, 1, and −1 decimal places:

**SELECT ROUND(5.75), ROUND(5.75, 1), ROUND(5.75, −1)**

**FROM dual;**

ROUND(5.75) ROUND(5.75,1) ROUND(5.75,−1)

----------- ------------- --------------

6 5.8 10

5.75 rounded to zero decimal places is 6; 5.75 rounded to one decimal place (to the right of the decimal point) is 5.8; and 5.75 rounded to one decimal place to the left of the decimal point (as indicated using a negative sign) is 10.

**SIGN()**

You use SIGN(*x*) to get the sign of *x*. SIGN() returns −1 if *x* is negative, 1 if *x* is positive, or 0 if *x* is zero. The following query gets the sign of −5, 5, and 0:

**SELECT SIGN(−5), SIGN(5), SIGN(0)**

**FROM dual;**

SIGN(−5) SIGN(5) SIGN(0)

---------- ---------- ----------

−1 1 0

The sign of −5 is −1; the sign of 5 is 1; the sign of 0 is 0.

**SQRT()**

You use SQRT(*x*) to get the square root of *x*. The following query gets the square root of 25 and 5:

**SELECT SQRT(25), SQRT(5)**

**FROM dual;**

SQRT(25) SQRT(5)

---------- ----------

5 2.23606798

The square root of 25 is 5; the square root of 5 is approximately 2.236.

**TRUNC()**

You use TRUNC(*x*, [*y*]) to get the result of truncating the number *x* to an optional *y* decimal places. If *y* is omitted, *x* is truncated to zero decimal places. If *y* is negative, *x* is truncated to the left of the decimal point. The following query truncates 5.75 to zero, 1, and −1 decimal places:

**SELECT TRUNC(5.75), TRUNC(5.75, 1), TRUNC(5.75, −1)**

**FROM dual;**

TRUNC(5.75) TRUNC(5.75,1) TRUNC(5.75,−1)

----------- ------------- --------------

5 5.7 0

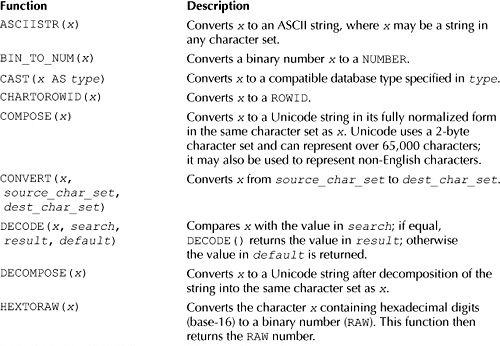
In the above, 5.75 truncated to zero decimal places is 5; 5.75 truncated to one decimal place (to the right of the decimal point) is 5.7; and 5.75 truncated to one decimal place to the left of the decimal point (as indicated using a negative sign) is 0.

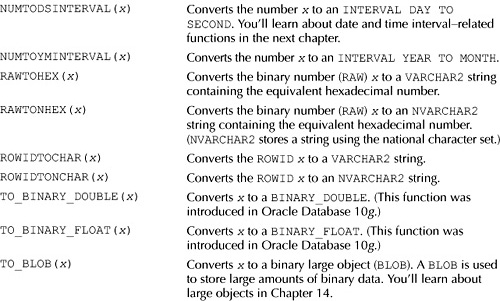
**Conversion Functions**

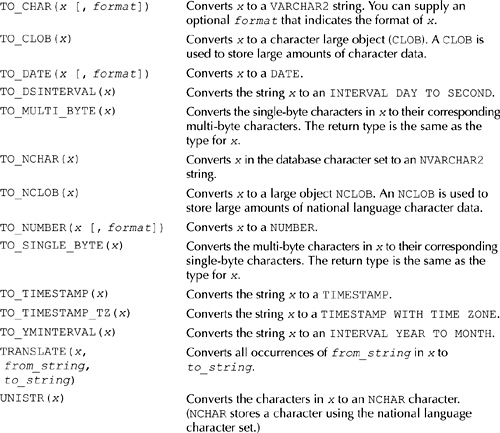
Sometimes you need to convert a value from one data type to another. For example, you might want to reformat the price of a product that is stored as a number (e.g., 1346.95) to a string containing dollar signs and commas (e.g., $1,346.95). For this purpose, you use a conversion function to convert a value from one data type to another.

[Table 4-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-3) shows some of the conversion functions.

You’ll learn more about the TO\_CHAR() and TO\_NUMBER() functions in the following sections. You’ll learn about some of the other functions in [Table 4-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-3) as you progress through this book. You can find out more about national language character sets and Unicode in the *Oracle Database Globalization Support Guide* from Oracle Corporation.







**TABLE 4-3** *Conversion Functions*

**TO\_CHAR()**

You use TO\_CHAR(*x* [, *format*]) to convert *x* to a string. You can also provide an optional *format* that indicates the format of *x*. The structure *format* depends on whether *x* is a number or date. You’ll learn how to use TO\_CHAR() to convert a number to a string in this section, and you’ll see how to convert a date to a string in the next chapter.

Let’s take a look at a couple of simple queries that use TO\_CHAR() to convert a number to a string. The following query converts 12345.67 to a string:

**SELECT TO\_CHAR(12345.67)**

**FROM dual;**

TO\_CHAR(1

---------

12345.67

The next query uses TO\_CHAR() to convert 12345678.90 to a string and specifies this number is to be converted using the format 99,999.99. This results in the string returned by TO\_CHAR() having a comma to delimit the thousands:

**SELECT TO\_CHAR(12345.67, '99,999.99')**

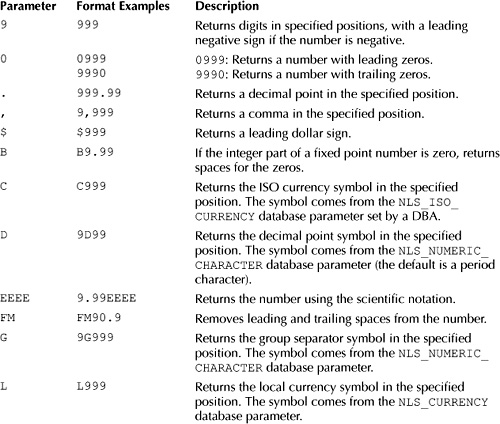
**FROM dual;**

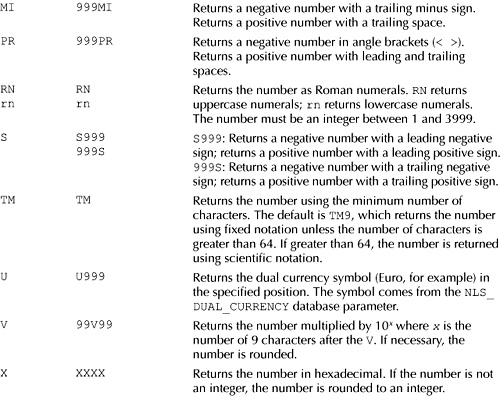
TO\_CHAR(12

----------

12,345.67

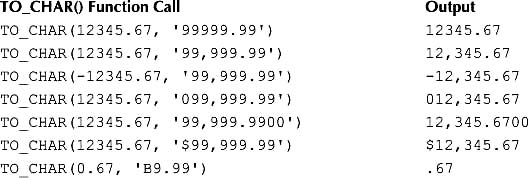
The optional *format* string you may pass to TO\_CHAR() has a number of parameters that affect the string returned by TO\_CHAR(). Some of these parameters are listed in [Table 4-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-4).

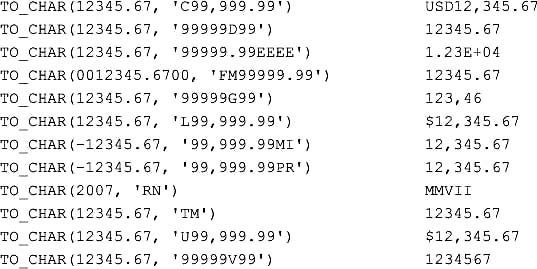




**TABLE 4-4** *Numeric Formatting Parameters*

Let’s look at some more examples that convert numbers to strings using TO\_CHAR(). The following table shows examples of calling TO\_CHAR(), along with the output returned.





TO\_CHAR() will return a string of pound characters (#) if you try to format a number that contains too many digits for the format. For example:

**SELECT TO\_CHAR(12345678.90, '99,999.99')**

**FROM dual;**

TO\_CHAR(12

----------

##########

Pound characters are returned by TO\_CHAR() because the number 12345678.90 has more digits than those allowed in the format 99,999.99.

You can also use TO\_CHAR() to convert columns containing numbers to strings. For example, the following query uses TO\_CHAR() to convert the price column of the products table to a string:

**SELECT product\_id, 'The price of the product is' || TO\_CHAR(price, '$99.99')**

**FROM products**

**WHERE product\_id < 5;**

PRODUCT\_ID 'THEPRICEOFTHEPRODUCTIS'||TO\_CHAR(

---------- ----------------------------------

1 The price of the product is $19.95

2 The price of the product is $30.00

3 The price of the product is $25.99

4 The price of the product is $13.95

**TO\_NUMBER()**

You use TO\_NUMBER(*x* [, *format*]) to convert *x* to a number. You can provide an optional *format* string to indicate the format of *x*. Your *format* string may use the same parameters as those listed earlier in [Table 4-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-4).

The following query converts the string 970.13 to a number using TO\_NUMBER():

**SELECT TO\_NUMBER('970.13')**

**FROM dual;**

TO\_NUMBER('970.13')

-------------------

970.13

The next query converts the string 970.13 to a number using TO\_NUMBER() and then adds 25.5 to that number:

**SELECT TO\_NUMBER('970.13') + 25.5**

**FROM dual;**

TO\_NUMBER('970.13')+25.5

------------------------

995.63

The next query converts the string −$12,345.67 to a number, passing the format string $99,999.99 to TO\_NUMBER():

**SELECT TO\_NUMBER('-$12,345.67', '$99,999.99')**

**FROM dual;**

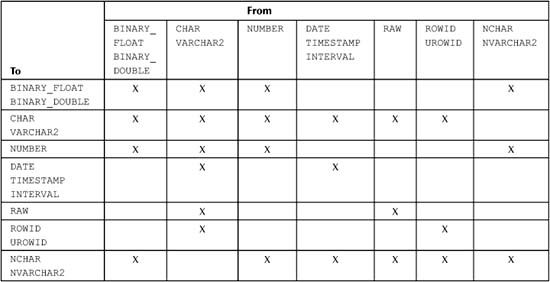
TO\_NUMBER('-$12,345.67','$99,999.99')

-------------------------------------

-12345.67

**CAST()**

You use CAST(*x* AS *type*) to convert *x* to a compatible database type specified by *type*. The following table shows the valid type conversions (valid conversions are marked with an X):



The following query shows the use of CAST() to convert literal values to specific types:

**SELECT**

**CAST(12345.67 AS VARCHAR2(10))**,

**CAST('9A4F' AS RAW(2))**,

**CAST('05-JUL-07' AS DATE)**,

**CAST(12345.678 AS NUMBER(10,2))**

**FROM dual;**

CAST(12345 CAST CAST('05- CAST(12345.678ASNUMBER(10,2))

---------- ---- --------- -----------------------------

12345.67 9A4F 05-JUL-07 12345.68

You can also convert column values from one type to another, as shown in the following query:

**SELECT**

**CAST(price AS VARCHAR2(10))**,

**CAST(price + 2 AS NUMBER(7,2))**,

**CAST(price AS BINARY\_DOUBLE)**

**FROM products**

**WHERE product\_id = 1;**

CAST(PRICE CAST(PRICE+2ASNUMBER(7,2)) CAST(PRICEASBINARY\_DOUBLE)

---------- -------------------------- --------------------------

19.95 21.95 1.995E+001

You’ll see additional examples in [Chapter 5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#ch05) that show how to use CAST() to convert dates, times, and intervals. Also, [Chapter 13](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch13.html#ch13) shows how you use CAST() to convert collections.

**Regular Expression Functions**

In this section, you’ll learn about regular expressions and their associated Oracle database functions. These functions allow you to search for a pattern of characters in a string. For example, let’s say you have the following list of years,

1965

1968

1971

1970

and want to get the years 1965 through to 1968. You can do this using the following regular expression:

^196[5-8]$

The regular expression contains a number of *metacharacters*. In this example, ^, [5-8], and $ are the metacharacters; ^ matches the beginning position of a string; [5-8] matches characters between 5 and 8; $matches the end position of a string. Therefore ^196 matches a string that begins with 196, and [5-8]$matches a string that ends with 5, 6, 7, or 8. So ^196[5-8]$ matches 1965, 1966, 1967, and 1968, which are the years you wanted to get from the list.

The next example uses the following string, which contains a quote from Shakespeare’s *Romeo and Juliet*:

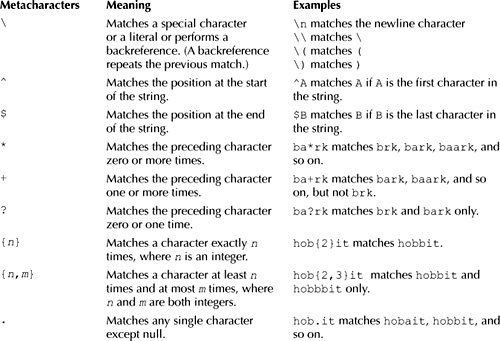
But, soft! What light through yonder window breaks?

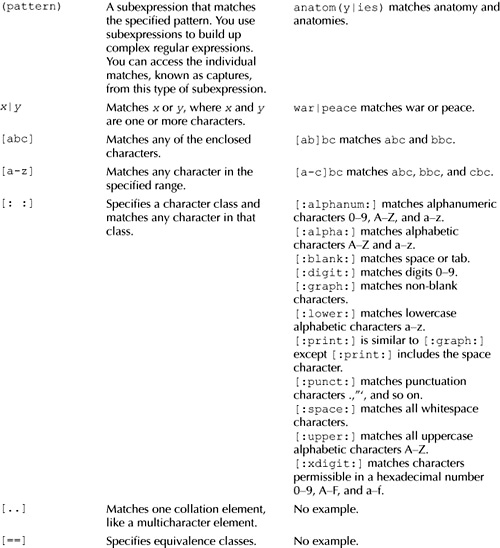
Let’s say you want to get the substring light. You do this using the following regular expression:

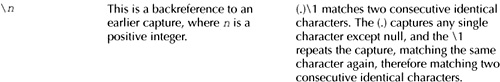
l[[:alpha:]]{4}

In this regular expression, [[:alpha:]] and {4} are the metacharacters. [[:alpha:]] matches an alphanumeric character A-Z and a-z; {4} repeats the previous match four times. When l, [[:alpha:]], and {4} are combined, they match a sequence of five letters starting with l. Therefore, the regular expression l[[:alpha:]]{4} matches light in the string.

[Table 4-5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-5) lists some the metacharacters you can use in a regular expression, along with their meaning and an example of their use.



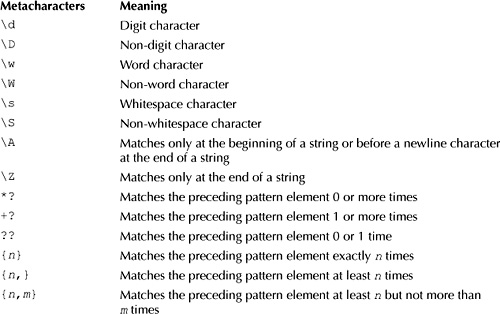




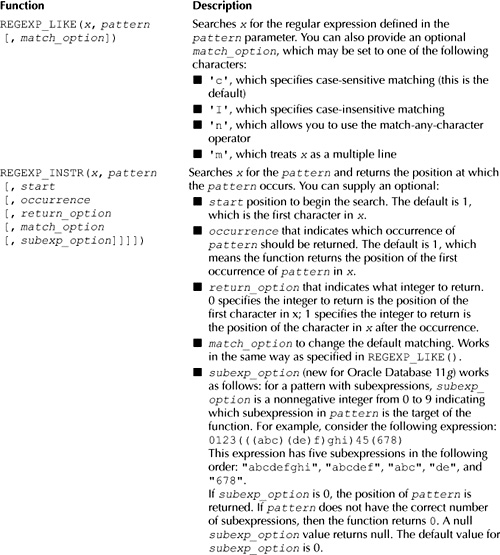
**TABLE 4-5** *Regular Expression Metacharacters*

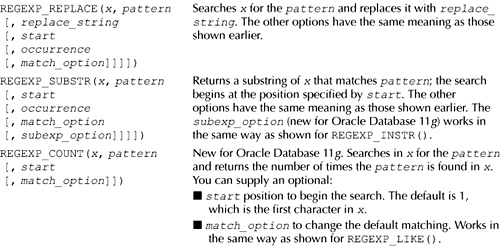
Oracle Database 10*g* Release 2 introduced a number of Perl-influenced metacharacters, which are shown in [Table 4-6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-6).

[Table 4-7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-7) shows the regular expression functions. Regular expression functions were introduced in Oracle Database 10*g*, and additional items have been added to 11*g*, as shown in the table.



**TABLE 4-6** *Perl-Influenced Metacharacters*





**TABLE 4-7** *Regular Expression Functions*

You’ll learn more about the regular expression functions in the following sections.

**REGEXP\_LIKE()**

You use REGEXP\_LIKE(*x*, *pattern* [, *match\_option*]) to search *x* for the regular expression defined in the *pattern* parameter. You can also provide an optional *match\_option*, which may be set to one of the following characters:

Image 'c', which specifies case-sensitive matching (this is the default)

Image 'I', which specifies case-insensitive matching

Image 'n', which allows you to use the match-any-character operator

Image 'm', which treats *x* as a multiple line

The following query retrieves customers whose date of birth is between 1965 and 1968 using REGEXP\_LIKE():

**SELECT customer\_id, first\_name, last\_name, dob**

**FROM customers**

**WHERE REGEXP\_LIKE(TO\_CHAR(dob, 'YYYY'), '^196[5-8]$');**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB

----------- ---------- ---------- ---------

1 John Brown 01-JAN-65

2 Cynthia Green 05-FEB-68

The next query retrieves customers whose first name starts with J or j. Notice the regular expression passed to REGEXP\_LIKE() is ^j and the match option is i (i indicates case-insensitive matching and so in this example ^j matches J or j).

**SELECT customer\_id, first\_name, last\_name, dob**

**FROM customers**

**WHERE REGEXP\_LIKE(first\_name, '^j', 'i');**

CUSTOMER\_ID FIRST\_NAME LAST\_NAME DOB

----------- ---------- ---------- ---------

1 John Brown 01-JAN-65

**REGEXP\_INSTR()**

You use REGEXP\_INSTR(*x*, *pattern* [, *start*[, *occurrence* [, *return\_option* [,*match\_option*]]]]) to search *x* for the *pattern*;. This function returns the position at which *pattern*occurs (positions start at number 1).

The following query returns the position that matches the regular expression l[[:alpha:]]{4} using REGEXP\_INSTR():

**SELECT**

**REGEXP\_INSTR('But, soft! What light through yonder window breaks?'**,

**'l[[:alpha:]]{4}') AS result**

**FROM dual;**

RESULT

----------

17

Notice that 17 is returned, which is the position of the l in light.

The next query returns the position of the second occurrence that matches the regular expression s[[:alpha:]]{3} starting at position 1:

**SELECT**

**REGEXP\_INSTR('But, soft! What light through yonder window softly breaks?'**,

**’s[[:alpha:]]{3}', 1, 2) AS result**

**FROM dual;**

RESULT

----------

45

The next query returns the position of the second occurrence that matches the letter o starting the search at position 10:

**SELECT**

**REGEXP\_INSTR('But, soft! What light through yonder window breaks?'**,

**'o', 10, 2) AS result**

**FROM dual;**

RESULT

----------

32

**REGEXP\_REPLACE()**

You use REGEXP\_REPLACE(*x*, *pattern* [, *replace\_string* [, *start* [, *occurrence* [,*match\_option*]]]]) to search *x* for the *pattern* and replace it with *replace\_string*.

The following query replaces the substring that matches the regular expression l[[:alpha:]]{4} with the string ’sound' using REGEXP\_REPLACE():

**SELECT**

**REGEXP\_REPLACE('But, soft! What light through yonder window breaks?'**,

**'l[[:alpha:]]{4}', ’sound') AS result**

**FROM dual;**

RESULT

---------------------------------------------------

But, soft! What sound through yonder window breaks?

Notice that light has been replaced by sound.

**REGEXP\_SUBSTR()**

You use REGEXP\_SUBSTR(*x*, *pattern* [, *start* [, *occurrence* [, *match\_option*]]]) to get a substring of *x* that matches *pattern*; the search begins at the position specified by *start*.

The following query returns the substring that matches the regular expression l[[:alpha:]]{4} using REGEXP\_SUBSTR():

**SELECT**

**REGEXP\_SUBSTR('But, soft! What light through yonder window breaks?'**,

**'l[[:alpha:]]{4}') AS result**

**FROM dual;**

RESUL

-----

light

**REGEXP\_COUNT()**

REGEXP\_COUNT() is new for Oracle Database 11*g*. You use REGEXP\_COUNT(*x*, *pattern* [, *start*[, *match\_option*]]) to search in *x* for the *pattern* and get the number of times *pattern* is found in *x*. You can provide an optional *start* number to indicate the character in *x* to begin searching for *pattern* and an optional *match\_option* string to indicate the match option.

The following query returns the number of times the regular expression s[[:alpha:]]{3} occurs in a string using REGEXP\_COUNT():

**SELECT**

**REGEXP\_COUNT('But, soft! What light through yonder window softly breaks?'**,

**’s[[:alpha:]]{3}') AS result**

**FROM dual;**

RESULT

----------

2

Notice that 2 is returned, which means the regular expression has two matches in the supplied string.

**USING AGGREGATE FUNCTIONS**

The functions you’ve seen so far operate on a single row at a time and return one row of output for each input row. In this section, you’ll learn about aggregate functions, which operate on a group of rows and return one row of output.

Image

**NOTE**  
*Aggregate functions are also known as group functions because they operate on groups of rows*.

[Table 4-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-8) lists some of the aggregate functions, all of which return a NUMBER.

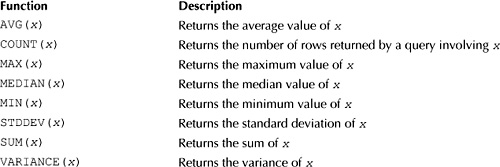
Here are some points to remember when using aggregate functions:

Image You can use the aggregate functions with any valid expression. For example, you can use the COUNT(), MAX(), and MIN() functions with numbers, strings, and datetimes.

Image Null values are ignored by aggregate functions, because a null value indicates the value is unknown and therefore cannot be used in the aggregate function’s calculation.

Image You can use the DISTINCT keyword with an aggregate function to exclude duplicate entries from the aggregate function’s calculation.

You’ll learn more about some of the aggregate functions shown in [Table 4-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-8) in the following sections. In [Chapters 7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#ch07) and [8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch08.html#ch08), you’ll see how to use these functions in conjunction with the SELECT statement’s ROLLUPand RETURNING clauses. As you’ll see, ROLLUP allows you to get a subtotal for a group of rows, where the subtotal is calculated using one of the aggregate functions; RETURNING allows you to store the value returned by an aggregate function in a variable.



**TABLE 4-8** *Aggregate Functions*

**AVG()**

You use AVG(*x*) to get the average value of *x*. The following query gets the average price of the products; notice that the price column from the products table is passed to the AVG() function:

**SELECT AVG(price)**

**FROM products;**

AVG(PRICE)

----------

19.7308333

You can use the aggregate functions with any valid expression. For example, the following query passes the expression price + 2 to AVG(); this adds 2 to each row’s price and then returns the average of those values.

**SELECT AVG(price + 2)**

**FROM products;**

AVG(PRICE)

----------

21.7308333

You can use the DISTINCT keyword to exclude identical values from a computation. For example, the following query uses the DISTINCT keyword to exclude identical values in the price column when computing the average using AVG():

**SELECT AVG(DISTINCT price)**

**FROM products;**

AVG(DISTINCTPRICE)

------------------

20.2981818

Notice that the average in this example is slightly higher than the average returned by the first query in this section. This is because the value for product #12 (13.49) in the price column is the same as the value for product #7; it is considered a duplicate and excluded from the computation performed by AVG(). Therefore, the average is slightly higher in this example.

**COUNT()**

You use COUNT(*x*) to get the number of rows returned by a query. The following query gets the number of rows in the products table using COUNT():

**SELECT COUNT(product\_id)**

**FROM products;**

COUNT(PRODUCT\_ID)

-----------------

12

Image

**TIP**  
*You should avoid using the asterisk (*\* *) with the* COUNT() *function, as it may take longer for*COUNT() *to return the result. Instead, you should use a column in the table or use the* ROWID *pseudo column. (As you saw in*[*Chapter 2*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02)*, the* ROWID *column contains the internal location of the row in the Oracle database.)*

The following example passes ROWID to COUNT() and gets the number of rows in the products table:

**SELECT COUNT(ROWID)**

**FROM products;**

COUNT(ROWID)

------------

12

**MAX() and MIN()**

You use MAX(*x*) and MIN(*x*) to get the maximum and minimum values for *x*. The following query gets the maximum and minimum values of the price column from the products table using MAX() and MIN():

**SELECT MAX(price), MIN(price)**

**FROM products;**

MAX(PRICE) MIN(PRICE)

---------- ----------

49.99 10.99

You may use MAX() and MIN() with any type, including strings and dates. When you use MAX() with strings, the strings are ordered alphabetically with the "maximum" string being at the bottom of a list and the "minimum" string being at the top of the list. For example, the string Albert would appear before Zeb in such a list. The following example gets the maximum and minimum name strings from the products table using MAX() and MIN():

**SELECT MAX(name), MIN(name)**

**FROM products;**

MAX(NAME) MIN(NAME)

------------------------------ ------------------------------

Z Files 2412: The Return

In the case of dates, the "maximum" date occurs at the latest point in time, and the "minimum" date at the earliest point in time. The following query gets the maximum and minimum dob from the customers table using MAX() and MIN():

**SELECT MAX(dob), MIN(dob)**

**FROM customers;**

MAX(DOB) MIN(DOB)

--------- ---------

16-MAR-71 01-JAN-65

**STDDEV()**

You use STDDEV(*x*) to get the standard deviation of *x*. Standard deviation is a statistical function and is defined as the square root of the variance (you’ll learn about variance shortly).

The following query gets the standard deviation of the price column values from the products table using STDDEV():

**SELECT STDDEV(price)**

**FROM products;**

STDDEV(PRICE)

-------------

11.0896303

**SUM()**

SUM(*x*) adds all the values in *x* and returns the total. The following query gets the sum of the price column from the products table using SUM():

**SELECT SUM(price)**

**FROM products;**

SUM(PRICE)

----------

236.77

**VARIANCE()**

You use VARIANCE(*x*) to get the variance of *x*. Variance is a statistical function and is defined as the spread or variation of a group of numbers in a sample. Variance is equal to the square of the standard deviation.

The following example gets the variance of the price column values from the products table using VARIANCE():

**SELECT VARIANCE(price)**

**FROM products;**

VARIANCE(PRICE)

---------------

122.979899

**GROUPING ROWS**

Sometimes you need to group blocks of rows in a table and get some information on those groups of rows. For example, you might want to get the average price for the different types of products in the products table. You’ll see how to do this the hard way first, then you’ll see the easy way, which involves using the GROUP BYclause to group similar rows together.

To do it the hard way, you limit the rows passed to the AVG() function using a WHERE clause. For example, the following query gets the average price for books from the products table (books have a product\_type\_idof 1):

**SELECT AVG(price)**

**FROM products**

**WHERE product\_type\_id = 1;**

AVG(PRICE)

----------

24.975

To get the average price for the other types of products, you would need to perform additional queries with different values for the product\_type\_id in the WHERE clause. As you can imagine, this is very labor intensive. You’ll be glad to know there’s an easier way to do this through the use of the GROUP BY clause.

**Using the GROUP BY Clause to Group Rows**

You use the GROUP BY clause to group rows into blocks with a common column value. For example, the following query groups the rows from the products table into blocks with the same product\_type\_id:

**SELECT product\_type\_id**

**FROM products**

**GROUP BY product\_type\_id;**

PRODUCT\_TYPE\_ID

---------------

1

2

3

4

Notice that there’s one row in the result set for each block of rows with the same product\_type\_id and that there’s a gap between 1 and 2 (you’ll see why this gap occurs shortly). In the result set, there’s one row for products with a product\_type\_id of 1, another for products with a product\_type\_id of 2, and so on. There are actually two rows in the products table with a product\_type\_id of 1, four rows with a product\_type\_id of 2, and so on for the other rows in the table. These rows are grouped together into separate blocks by the GROUP BY clause, one block for each product\_type\_id. The first block contains two rows, the second contains four rows, and so on.

The gap between 1 and 2 is caused by a row whose product\_type\_id is null. This row is shown in the following example:

**SELECT product\_id, name, price**

**FROM products**

**WHERE product\_type\_id IS NULL;**

PRODUCT\_ID NAME PRICE

---------- ------------------------------ ----------

12 My Front Line 13.49

Because this row’s product\_type\_id is null, the GROUP BY clause in the earlier query groups this row into a single block. The row in the result set is blank because the product\_type\_id is null for the block, so there’s a gap between 1 and 2.

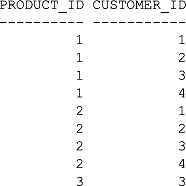
**Using Multiple Columns in a Group**

You can specify multiple columns in a GROUP BY clause. For example, the following query includes the product\_id and customer\_id columns from the purchases table in a GROUP BY clause:

**SELECT product\_id, customer\_id**

**FROM purchases**

**GROUP BY product\_id, customer\_id;**



**Using Groups of Rows with Aggregate Functions**

You can pass blocks of rows to an aggregate function. The aggregate function performs its computation on the group of rows in each block and returns one value per block. For example, to get the number of rows with the same product\_type\_id from the products table, you do the following:

image Use the GROUP BY clause to group rows into blocks with the same product\_type\_id.

image Use COUNT(ROWID) to get the number of rows in each block.

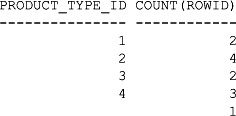
The following query shows this:

**SELECT product\_type\_id, COUNT(ROWID)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



Notice that there are five rows in the result set, with each row corresponding to one or more rows in the products table grouped together with the same product\_type\_id. From the result set, you can see there are two rows with a product\_type\_id of 1, four rows with a product\_type\_id of 2, and so on. The last line in the result set shows there is one row with a null product\_type\_id (this is caused by the "My Front Line" product mentioned earlier).

Let’s take a look at another example. To get the average price for the different types of products in the products table, you do the following:

image Use the GROUP BY clause to group rows into blocks with the same product\_type\_id.

image Use AVG(price) to get the average price for each block of rows.

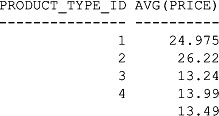
The following query shows this:

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



Each group of rows with the same product\_type\_id is passed to the AVG() function. AVG() then computes the average price for each group. As you can see from the result set, the average price for the group of products with a product\_type\_id of 1 is 24.975. Similarly, the average price of the products with a product\_type\_id of 2 is 26.22. Notice that the last row in the result set shows an average price of 13.49; this is simply the price of the "My Front Line" product, the only row with a null product\_type\_id.

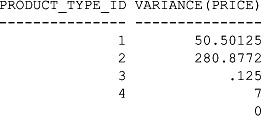
You can use any of the aggregate functions with the GROUP BY clause. For example, the next query gets the variance of product prices for each product\_type\_id:

**SELECT product\_type\_id, VARIANCE(price)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



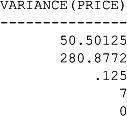
One point to remember is that you don’t have to include the columns used in the GROUP BY in the list of columns immediately after the SELECT. For example, the following query is the same as the previous one except product\_type\_id is omitted from the SELECT clause:

**SELECT VARIANCE(price)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



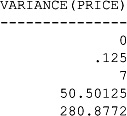
You can also include an aggregate function call in the ORDER BY, as shown in the following query:

**SELECT VARIANCE(price)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY VARIANCE(price);**



**Incorrect Usage of Aggregate Function Calls**

When your query contains an aggregate function—and retrieves columns not placed within an aggregate function—those columns must be placed in a GROUP BY clause. If you forget to do this, you’ll get the following error: ORA-00937: not a single-group group function. For example, the following query attempts to retrieve the product\_type\_id column and AVG(price) but omits a GROUP BY clause for product\_type\_id:

SQL> **SELECT product\_type\_id, AVG(price)**

2 **FROM products;**

SELECT product\_type\_id, AVG(price)

\*

ERROR at line 1:

ORA-00937: not a single-group group function

The error occurs because the database doesn’t know what to do with the product\_type\_id column. Think about it: the query attempts to use the AVG() aggregate function, which operates on multiple rows, but also attempts to get the product\_type\_id column values for each individual row. You can’t do both at the same time. You must provide a GROUP BY clause to tell the database to group multiple rows with the same product\_type\_id together; then the database passes those groups of rows to the AVG() function.

image

**CAUTION**  
*When a query contains an aggregate function—and retrieves columns not placed within an aggregate function—then those columns must be placed in a* GROUP BY *clause*.

Also, you cannot use an aggregate function to limit rows in a WHERE clause. If you try to do so, you will get the following error: ORA-00934: group function is not allowed here. For example:

SQL> **SELECT product\_type\_id, AVG(price)**

2 **FROM products**

3 **WHERE AVG(price) > 20**

4 **GROUP BY product\_type\_id;**

WHERE AVG(price) > 20

\*

ERROR at line 3:

ORA-00934: group function is not allowed here

The error occurs because you may only use the WHERE clause to filter *individual* rows, not *groups* of rows. To filter groups of rows, you use the HAVING clause, which you’ll learn about next.

**Using the HAVING Clause to Filter Groups of Rows**

You use the HAVING clause to filter groups of rows. You place the HAVING clause after the GROUP BY clause:

SELECT …

FROM …

WHERE

GROUP BY …

HAVING …

ORDER BY …;

image

**NOTE**  
GROUP BY *can be used without* HAVING, *but* HAVING *must be used in conjunction with* GROUP BY.

Let’s take a look at an example. Say you want to view the types of products that have an average price greater than $20. To do this, you do the following:

image Use the GROUP BY clause to group rows into blocks with the same product\_type\_id.

image Use the HAVING clause to limit the returned results to those groups that have an average price greater than $20.

The following query shows this:

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

**HAVING AVG(price) > 20;**

image

As you can see, only the groups of rows having an average price greater than $20 are displayed.

**Using the WHERE and GROUP BY Clauses Together**

You can use the WHERE and GROUP BY clauses together in the same query. When you do this, first the WHEREclause filters the rows returned, then the GROUP BY clause groups the remaining rows into blocks. For example, the following query uses

image A WHERE clause to filter the rows from the products table to select those whose price is less than $15.

image A GROUP BY clause to group the remaining rows by the product\_type\_id column.

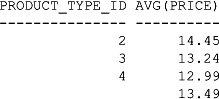
**SELECT product\_type\_id, AVG(price)**

**FROM products**

**WHERE price < 15**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



**Using the WHERE, GROUP BY, and HAVING Clauses Together**

You can use the WHERE, GROUP BY, and HAVING clauses together in the same query. When you do this, the WHERE clause first filters the rows, the GROUP BY clause then groups the remaining rows into blocks, and finally the HAVING clause filters the row groups. For example, the following query uses

image A WHERE clause to filter the rows from the products table to select those whose price is less than $15.

image A GROUP BY clause to group the remaining rows by the product\_type\_id column.

image A HAVING clause to filter the row groups to select those whose average price is greater than $13.

**SELECT product\_type\_id, AVG(price)**

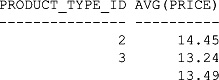
**FROM products**

**WHERE price < 15**

**GROUP BY product\_type\_id**

**HAVING AVG(price) > 13**

**ORDER BY product\_type\_id;**



Compare these results with the previous example. Notice that the group of rows with the product\_type\_idof 4 is filtered out. That’s because the group of rows has an average price less than $13.

The final query uses ORDER BY AVG(price) to re-order the results by the average price:

**SELECT product\_type\_id, AVG(price)**

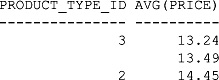
**FROM products**

**WHERE price < 15**

**GROUP BY product\_type\_id**

**HAVING AVG(price) > 13**

**ORDER BY AVG(price);**



**SUMMARY**

In this chapter, you have learned the following:

image The Oracle database has two main groups of functions: single-row functions and aggregate functions.

image Single-row functions operate on one row at a time and return one row of output for each input row. There are five main types of single-row functions: character functions, numeric functions, conversion functions, date functions, and regular expression functions.

image Aggregate functions operate on multiple rows and return one row of output.

image Blocks of rows may be grouped together using the GROUP BY clause.

image Groups of rows may be filtered using the HAVING clause.

**CHAPTER 5  
Storing and Processing Dates and Times**

In this chapter, you will see how to

Image Process and store a specific date and time, collectively known as a datetime. An example of a datetime is 7:15:30 p.m. on October 10, 2007. You store a datetime using the DATE type. The DATE type stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second.

Image Use *timestamps* to store a specific date and time. A timestamp stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. The advantages of a timestamp over a DATE are that a timestamp can store a fractional second and a time zone.

Image Use time *intervals* to store a length of time. An example of a time interval is 1 year 3 months.

Let’s plunge in and see some simple examples of storing and retrieving dates.

**SIMPLE EXAMPLES OF STORING AND RETRIEVING DATES**

By default the database uses the format DD-MON-YYYY to represent a date, where

Image DD is a two-digit day, e.g., 05

Image MON is the first three letters of the month, e.g., FEB

Image YYYY is a four-digit year, e.g., 1968

Let’s take a look at an example of adding a row to the customers table, which contains a DATE column named dob. The following INSERT adds a row to the customers table, setting the dob column to 05-FEB-1968:

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name, dob, phone**

**) VALUES (**

**6, 'Fred', 'Brown', '05-FEB-1968', '800-555-1215'**

**);**

You can also use the DATE keyword to supply a date literal to the database. The date must use the ANSI standard date format YYYY-MM-DD, where

Image YYYY is a four-digit year.

Image MM is a two-digit month from 1 to 12.

Image DD is a two-digit day.

Image

**TIP**  
*Using ANSI standard dates in SQL statements has the advantage that those statements could potentially run against non-Oracle databases*.

For example, to specify a date of October 25, 1972, you use DATE '1972-10-25'. The following INSERTadds a row to the customers table, specifying DATE '1972-10-25' for the dob column:

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name, dob, phone**

**) VALUES (**

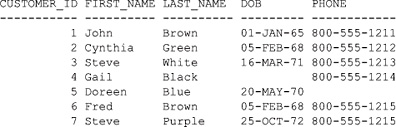
**7, ’Steve', 'Purple', DATE '1972-10-25', '800-555-1215'**

**);**

By default, the database returns dates in the format DD-MON-YY, where YY are the last two digits of the year. For example, the following example retrieves rows from the customers table and then performs a ROLLBACKto undo the results of the previous two INSERT statements; notice the two-digit years in the dob column returned by the query:

**SELECT \***

**FROM customers;**



**ROLLBACK;**

Customer #4’s dob is null and is therefore blank in the previous result set.

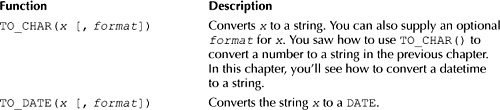
Image

**NOTE**  
*If you actually ran the two* INSERT *statements, make sure you undo the changes by executing the*ROLLBACK. *That way, you’ll keep the database in its initial state, and the results from your queries will match those in this chapter*.

In this section, you saw some simple examples of using dates that use default formats. You’ll learn how to provide your own date formats in the following section and see how to convert a datetime to another database type.

**CONVERTING DATETIMES USING TO\_CHAR() AND TO\_DATE()**

The Oracle database has functions that enable you to convert a value in one data type to another. You saw some of these functions in the previous chapter. In this section, you’ll see how to use the TO\_CHAR() and TO\_DATE() functions to convert a datetime to a string and vice versa. [Table 5-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-1) summarizes the TO\_CHAR()and TO\_DATE() functions.



**TABLE 5-1** *TO\_CHAR() and TO\_DATE() Conversion Functions*

Let’s start off by examining how you use TO\_CHAR() to convert a datetime to a string. Later, you’ll see how to use TO\_DATE() to convert a string to a DATE.

**Using TO\_CHAR() to Convert a Datetime to a String**

You can use TO\_CHAR(*x* [, *format*]) to convert the datetime *x* to a string. You can also provide an optional *format* for *x*. An example format is MONTH DD, YYYY, where

Image MONTH is the full name of the month in uppercase, e.g., JANUARY.

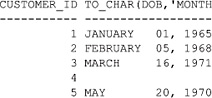
Image DD is the two-digit day.

Image YYYY is the four-digit year.

The following query uses TO\_CHAR() to convert the dob column from the customers table to a string with the format MONTH DD, YYYY:

**SELECT customer\_id, TO\_CHAR(dob, 'MONTH DD, YYYY')**

**FROM customers;**



The next query gets the current date and time from the database using the SYSDATE function, then converts the date and time to a string using TO\_CHAR() with the format MONTH DD, YYYY, HH24:MI:SS. The time portion of this format indicates that the hours are in 24-hour format and that the minutes and seconds are also to be included in the string.

**SELECT TO\_CHAR(SYSDATE, 'MONTH DD, YYYY, HH24:MI:SS')**

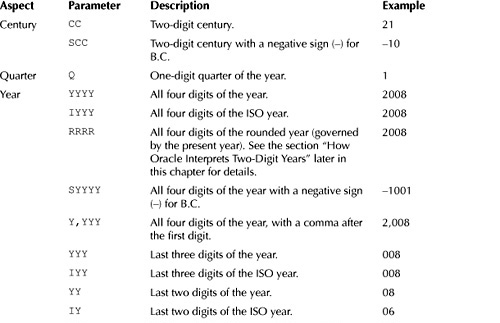
**FROM dual;**

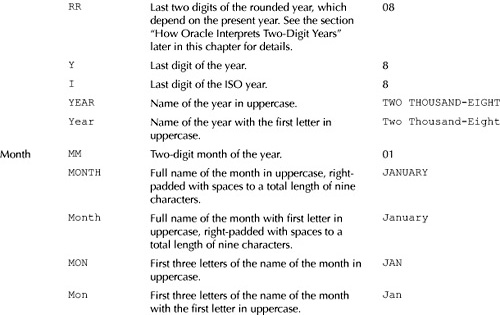
TO\_CHAR(SYSDATE,'MONTHDD, YYY

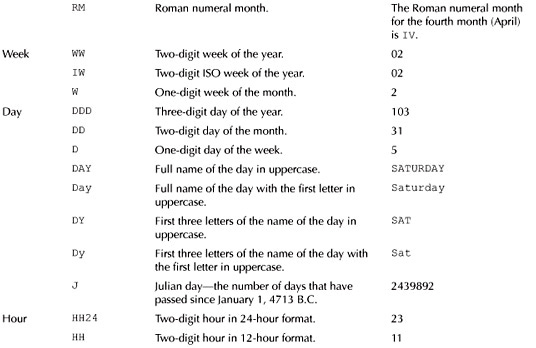
----------------------------

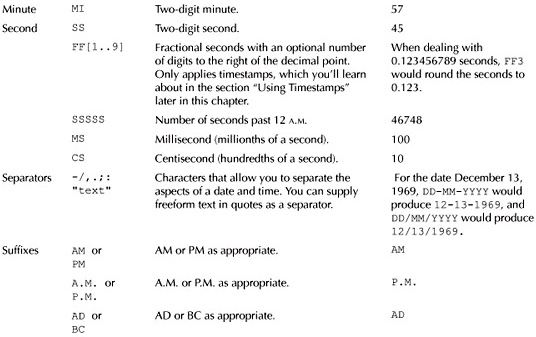
NOVEMBER 05, 2007, 12:34:36

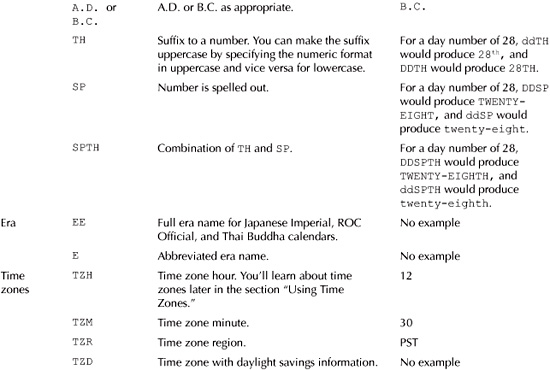
When you use TO\_CHAR() to convert a datetime to a string, the format has a number of parameters that affect the returned string. Some of these parameters are listed in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2).





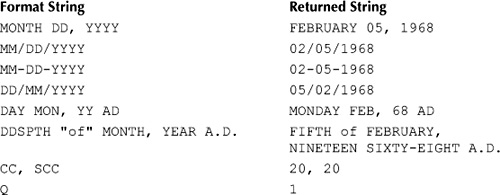


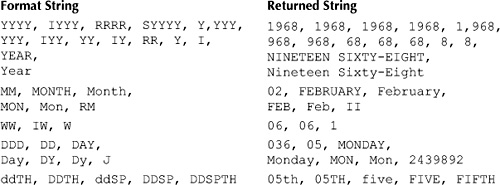




**TABLE 5-2** *Datetime Formatting Parameters*

The following table shows examples of strings to format the date February 5, 1968, along with the string returned from a call to TO\_CHAR().





You can see the results shown in this table by calling TO\_CHAR() in a query. For example, the following query converts February 5, 1968, to a string with the format MONTH DD, YYYY:

**SELECT TO\_CHAR(TO\_DATE('05-FEB-1968'), 'MONTH DD, YYYY')**

**FROM dual;**

TO\_CHAR(TO\_DATE('0

------------------

FEBRUARY 05, 1968

Image

**NOTE**  
*The* TO\_DATE() *function converts a string to a datetime. You’ll learn more about the* TO\_DATE()*function shortly*.

The following table shows examples of strings to format the time 19:32:36 (32 minutes and 36 seconds past 7 P.M.)—along with the output that would be returned from a call to TO\_CHAR() with that time and format string.

Image

**Using TO\_DATE() to Convert a String to a Datetime**

You use TO\_DATE(*x* [, *format*]) to convert the *x* string to a datetime. You can provide an optional *format* string to indicate the format of *x*. If you omit *format*, the date must be in the default database format (usually DD-MON-YYYY or DD-MON-YY).

Image

**NOTE**  
*The* NLS\_DATE\_FORMAT *database parameter specifies the default date format for the database. As you’ll learn later in the section "Setting the Default Date Format," you can change the setting of*NLS\_DATE\_FORMAT.

The following query uses TO\_DATE() to convert the strings 04-JUL-2007 and 04-JUL-07 to the date July 4, 2007; notice that the final date is displayed in the default format of DD-MON-YY:

**SELECT TO\_DATE('04-JUL-2007'), TO\_DATE('04-JUL-07')**

**FROM dual;**

TO\_DATE(' TO\_DATE('

--------- ---------

04-JUL-07 04-JUL-07

**Specifying a Datetime Format**

As mentioned earlier, you can supply an optional format for a datetime to TO\_DATE(). You use the same format parameters as those defined previously in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2). The following query uses TO\_DATE() to convert the string July 4, 2007 to a date, passing the format string MONTH DD, YYYY to TO\_DATE():

**SELECT TO\_DATE('July 4, 2007', 'MONTH DD, YYYY')**

**FROM dual;**

TO\_DATE('

---------

04-JUL-07

The next query passes the format string MM.DD.YY to TO\_DATE() and converts the string 7.4.07 to the date July 4, 2007; again, the final date is displayed in the default format DD-MON-YY:

**SELECT TO\_DATE('7.4.07', 'MM.DD.YY')**

**FROM dual;**

TO\_DATE('

---------

04-JUL-07

**Specifying Times**

You can also specify a time with a datetime. If you don’t supply a time with a datetime, the time part of your datetime defaults to 12:00:00 A.M. You can supply the format for a time using the various formats shown earlier in [Table 5-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-3). One example time format is HH24:MI:SS, where

Image HH24 is a two-digit hour in 24-hour format from 00 to 23.

Image MI is a two-digit minute from 00 to 59.

Image SS is a two-digit second from 00 to 59.

An example of a time that uses the HH24:MI:SS format is 19:32:36. A full example of a datetime that uses this time is

05-FEB-1968 19:32:36

with the format for this datetime being

DD-MON-YYYY HH24:MI:SS

The following TO\_DATE() call shows the use of this datetime format and value:

TO\_DATE('05-FEB-1968 19:32:36', ’DD-MON-YYYY HH24:MI:SS')

The datetime returned by TO\_DATE() in the previous example is used in the following INSERT that adds a row to the customers table; notice that the dob column for the new row is set to the datetime returned by TO\_DATE():

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name,**

**dob,**

**phone**

**) VALUES (**

**6, 'Fred', 'Brown'**

**TO\_DATE('05-FEB-1968 19:32:36', ’DD-MON-YYYY HH24:MI:SS')**

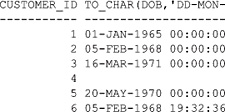
**'800-555-1215'**

**);**

You use TO\_CHAR() to view the time part of a datetime. For example, the following query retrieves the rows from the customers table and uses TO\_CHAR() to convert the dob column values; notice that customer #6 has the time previously set in the INSERT:

**SELECT customer\_id, TO\_CHAR(dob, ’DD-MON-YYYY HH24:MI:SS')**

**FROM customers;**



Notice the time for the dob column for customers #1, #2, #3, and #5 is set to 00:00:00 (12 a.m.). This is the default time substituted by the database when you don’t provide a time in a datetime.

The next statement rolls back the addition of the new row:

**ROLLBACK;**

Image

**NOTE**  
*If you actually ran the earlier* INSERT *statement, make sure you undo the change using* ROLLBACK.

**Combining TO\_CHAR() and TO\_DATE() Calls**

You can combine TO\_CHAR() and TO\_DATE() calls; doing this allows you to use datetimes in different formats. For example, the following query combines TO\_CHAR() and TO\_DATE() in order to view just the time part of a datetime; notice that the output from TO\_DATE() is passed to TO\_CHAR():

**SELECT TO\_CHAR(TO\_DATE('05-FEB-1968 19:32:36',**

**’DD-MON-YYYY HH24:MI:SS'), 'HH24:MI:SS')**

**FROM dual;**

TO\_CHAR(

--------

19:32:36

**SETTING THE DEFAULT DATE FORMAT**

The default date format is specified in the NLS\_DATE\_FORMAT database parameter. A DBA can change the setting of NLS\_DATE\_FORMAT by setting this parameter’s value in the database’s init.ora or spfile.orafile, both of which are read when the database is started. A DBA can also set NLS\_DATE\_FORMAT using the ALTER SYSTEM command. You can also set the NLS\_DATE\_FORMAT parameter for your own session using SQL\*Plus, which you do by using the ALTER SESSION command. For example, the following ALTER SESSION statement sets NLS\_DATE\_FORMAT to MONTH-DD-YYYY:

**ALTER SESSION SET NLS\_DATE\_FORMAT = 'MONTH-DD-YYYY';**

Session altered

Image

**NOTE**  
*A session is started when you connect to a database and is ended when you disconnect*.

You can see the use of this new date format in the results from the following query that retrieves the dob column for customer #1:

**SELECT dob**

**FROM customers**

**WHERE customer\_id = 1;**

DOB

-----------------

JANUARY -01-1965

You may also use the new date format when inserting a row in the database. For example, the following INSERTadds a new row to the customers table; notice the use of the format MONTH-DD-YYYY when supplying the dob column’s value:

**INSERT INTO customers (**

**customer\_id, first\_name, last\_name, dob, phone**

**) VALUES (**

**6, 'Fred', 'Brown', 'MARCH-15-1970', '800-555-1215'**

**);**

Go ahead and disconnect from the database and connect again as the store user; you’ll find that the date format is back to the default. That’s because any changes you make using the ALTER SESSION statement last only for that particular session—when you disconnect, you lose the change.

Image

**NOTE**  
*If you ran the previous* INSERT *statement, go ahead and delete the row using* DELETE FROM customers WHERE customer\_id = 6.

**HOW ORACLE INTERPRETS TWO-DIGIT YEARS**

The Oracle database stores all four digits of the year, but if you supply only two digits, the database will interpret the century according to whether the YY or RR format is used.

Image

**TIP**  
*You should always specify all four digits of the year. That way*, you *won’t get confused as to which year you mean*.

Let’s take a look at the YY format first, followed by the RR format.

**Using the YY Format**

If your date format uses YY for the year and you supply only two digits of a year, then the century for your year is assumed to be the same as the present century currently set on the database server. Therefore, *the first two digits of your supplied year are set to the first two digits of the present year*. For example, if your supplied year is 15 and the present year is 2007, your supplied year is set to 2015; similarly, a supplied year of 75 is set to 2075.

Image

**NOTE**  
*If you use the* YYYY *format but only supply a two-digit year, then your year is interpreted using the* YY*format*.

Let’s take a look at a query that uses the YY format for interpreting the years 15 and 75. In the following query, notice that the input dates 15 and 75 are passed to TO\_DATE(), whose output is passed to TO\_CHAR(), which converts the dates to a string with the format DD-MON-YYYY. (The YYYY format is used here, so you can see that all four digits of the year returned by TO\_DATE().)

**SELECT**

**TO\_CHAR(TO\_DATE('04-JUL-15', ’DD-MON-YY'), ’DD-MON-YYYY'),**

**TO\_CHAR(TO\_DATE('04-JUL-75', ’DD-MON-YY'), ’DD-MON-YYYY')**

**FROM dual;**

TO\_CHAR(TO\_ TO\_CHAR(TO\_

----------- -----------

04-JUL-2015 04-JUL-2075

As expected, the years 15 and 75 are interpreted as 2015 and 2075.

**Using the RR Format**

If your date format is RR and you supply the last two digits of a year, the first two digits of your year are determined using the two-digit year you supply (your *supplied year*) and the last two digits of the present date on the database server (the *present year*). The rules used to determine the century of your supplied year are as follows:

Image **Rule 1** If your supplied year is between 00 and 49 and the present year is between 00 and 49, the century is the same as the present century. Therefore, *the first two digits of your supplied year are set to the first two digits of the present year*. For example, if your supplied year is 15 and the present year is 2007, your supplied year is set to 2015.

Image **Rule 2** If your supplied year is between 50 and 99 and the present year is between 00 and 49, the century is the present century minus 1. Therefore, *the first two digits of your supplied year are set to the present year’s first two digits minus 1*. For example, if your supplied year is 75 and the present year is 2007, your supplied year is set to 1975.

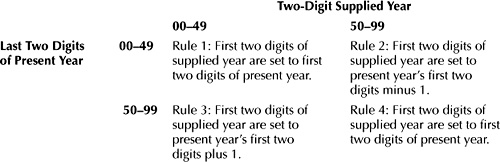
Image **Rule 3** If your supplied year is between 00 and 49 and the present year is between 50 and 99, the century is the present century plus 1. Therefore, *the first two digits of your supplied year are set to the present year’s first two digits plus 1*. For example, if your supplied year is 15 and the present year is 2075, your supplied year is set to 2115.

Image **Rule 4** If your supplied year is between 50 and 99 and the present year is between 50 and 99, the century is the same as the present century. Therefore, *the first two digits of your supplied year are set to the first two digits of the present year*. For example, if your supplied year is 55 and the present year is 2075, your supplied year is set to 2055.

[Table 5-3](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-3) summarizes these results.

Image

**NOTE**  
*If you use the* RRRR *format but supply only a two-digit year, then your year is interpreted using the* RR*format*.



**TABLE 5-3** *How Two-Digit Years Are Interpreted*

Let’s take a look at a query that uses the RR format for interpreting the years 15 and 75. (In the following query, you should assume the present year is 2007.)

**SELECT**

**TO\_CHAR(TO\_DATE('04-JUL-15', ’DD-MON-RR'), ’DD-MON-YYYY'),**

**TO\_CHAR(TO\_DATE('04-JUL-75', ’DD-MON-RR'), ’DD-MON-YYYY')**

**FROM dual;**

TO\_CHAR(TO\_ TO\_CHAR(TO\_

----------- -----------

04-JUL-2015 04-JUL-1975

As expected from rules 1 and 2, the years 15 and 75 are interpreted as 2015 and 1975.

In the next query, you should assume the present year is 2075.

**SELECT**

**TO\_CHAR(TO\_DATE('04-JUL-15', ’DD-MON-RR'), ’DD-MON-YYYY'),**

**TO\_CHAR(TO\_DATE('04-JUL-55', ’DD-MON-RR'), ’DD-MON-YYYY')**

**FROM dual;**

TO\_CHAR(TO\_ TO\_CHAR(TO\_

----------- -----------

04-JUL-2115 04-JUL-2055

As expected from rules 3 and 4, the years 15 and 75 are interpreted as 2115 and 2055.

**USING DATETIME FUNCTIONS**

You use the datetime functions to get or process datetimes and timestamps (you’ll learn about timestamps later in this chapter). [Table 5-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-4) shows some of the datetime functions. In this table, *x* represents a datetime or a timestamp.

You’ll learn more about the functions shown in [Table 5-4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-4) in the following sections.

**ADD\_MONTHS()**

ADD\_MONTHS(*x*, *y*) returns the result of adding *y* months to *x*. If *y* is negative, then *y* months are subtracted from *x*. The following example adds 13 months to January 1, 2007:

**SELECT ADD\_MONTHS('01-JAN-2007', 13)**

**FROM dual;**

ADD\_MONTH

---------

01-FEB-08

The next example subtracts 13 months from the January 1, 2008; notice that −13 months are "added" to this date using ADD\_MONTHS():

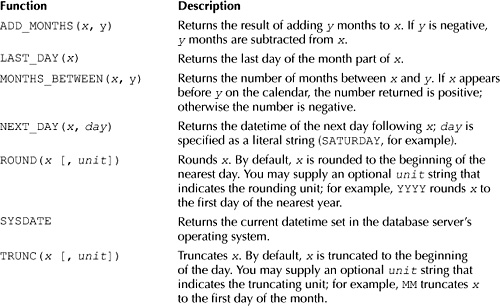
**SELECT ADD\_MONTHS('01-JAN-2008', -13)**

**FROM dual;**

ADD\_MONTH

---------

01-DEC-06



**TABLE 5-4** *Datetime Functions*

You can provide a time and date to the ADD\_MONTHS() function. For example, the following query adds two months to the datetime 7:15:26 P.M. on January 1, 2007:

**SELECT ADD\_MONTHS(TO\_DATE('01-JAN-2007 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS'), 2)**

**FROM dual;**

ADD\_MONTH

---------

01-MAR-07

The next query rewrites the previous example to convert the returned datetime from ADD\_MONTHS() to a string using TO\_CHAR() with the format DD-MON-YYYY HH24:MI:SS:

**SELECT TO\_CHAR(ADD\_MONTHS(TO\_DATE('01-JAN-2007 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS'), 2), ’DD-MON-YYYY HH24:MI:SS')**

**FROM dual;**

TO\_CHAR(ADD\_MONTHS(T

--------------------

01-MAR-2007 19:15:26

Image

**NOTE**  
*You can provide a date and time to any of the functions shown earlier in*[*Table 5-4*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-4).

**LAST\_DAY()**

LAST\_DAY(*x*) returns the date of the last day of the month part of *x*. The following example displays the last date in January 2008:

**SELECT LAST\_DAY('01-JAN-2008')**

**FROM dual;**

LAST\_DAY(

---------

31-JAN-08

**MONTHS\_BETWEEN()**

MONTHS\_BETWEEN(*x*, *y*) returns the number of months between *x* and *y*. If *x* occurs before *y* in the calendar, then the number returned by MONTHS\_BETWEEN() is negative.

Image

**NOTE**  
*The ordering of the dates in your call to the* MONTHS\_BETWEEN() *function is important: the* later date *must appear* first if *you want the result as a positive number*.

The following example displays the number of months between May 25, 2008, and January 15, 2008. Notice that since the later date (May 25, 2008) appears first, the result returned is a positive number:

**SELECT MONTHS\_BETWEEN('25-MAY-2008', '15-JAN-2008')**

**FROM dual;**

MONTHS\_BETWEEN('25-MAY-2008','15-JAN-2008')

-------------------------------------------

4.32258065

The next example reverses the same dates in the call to the MONTHS\_BETWEEN() function, and therefore the returned result is a negative number of months:

**SELECT MONTHS\_BETWEEN('15-JAN-2008', '25-MAY-2008')**

**FROM dual;**

MONTHS\_BETWEEN('15-JAN-2008','25-MAY-2008')

--------------------------------------------

−4.3225806

**NEXT\_DAY()**

NEXT\_DAY(*x*, *day*) returns the date of the next day following *x*; you specify *day* as a literal string (SATURDAY, for example).

The following example displays the date of the next Saturday after January 1, 2008:

**SELECT NEXT\_DAY('01-JAN-2008', ’SATURDAY')**

**FROM dual;**

NEXT\_DAY(

---------

05-JAN-08

**ROUND()**

ROUND(*x* [, *unit*]) rounds *x*, by default, to the beginning of the nearest day. If you supply an optional *unit* string, *x* is rounded to that unit; for example, YYYY rounds *x* to the first day of the nearest year. You can use many of the parameters shown earlier in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2) to round a datetime.

The following example uses ROUND() to round October 25, 2008, up to the first day in the nearest year, which is January 1, 2009. Notice that the date is specified as 25-OCT-2008 and is contained within a call to the TO\_DATE() function:

**SELECT ROUND(TO\_DATE('25-OCT-2008'), 'YYYY')**

**FROM dual;**

ROUND(TO\_

---------

01-JAN-09

The next example rounds May 25, 2008, to the first day in the nearest month, which is June 1, 2008, because May 25 is closer to the beginning of June than it is to the beginning of May:

**SELECT ROUND(TO\_DATE('25-MAY-2008'), 'MM')**

**FROM dual;**

ROUND(TO\_

---------

01-JUN-08

The next example rounds 7:45:26 P.M. on May 25, 2008, to the nearest hour, which is 8:00 P.M.:

**SELECT TO\_CHAR(ROUND(TO\_DATE('25-MAY-2008 19:45:26',**

**’DD-MON-YYYY HH24:MI:SS'), 'HH24'), ’DD-MON-YYYY HH24:MI:SS')**

**FROM dual;**

TO\_CHAR(ROUND(TO\_DAT

--------------------

25-MAY-2008 20:00:00

**SYSDATE**

SYSDATE returns the current datetime set in the database server’s operating system. The following example gets the current date:

**SELECT SYSDATE**

**FROM dual;**

SYSDATE

---------

05-NOV-07

**TRUNC()**

TRUNC(*x* [, *unit*]) truncates *x*. By default, *x* is truncated to the beginning of the day. If you supply an optional *unit* string, *x* is truncated to that unit; for example, MM truncates *x* to the first day in the month. You can use many of the parameters shown earlier in [Table 5-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2) to truncate a datetime.

The following example uses TRUNC() to truncate May 25, 2008, to the first day in the year, which is January 1, 2008:

**SELECT TRUNC(TO\_DATE('25-MAY-2008'), 'YYYY')**

**FROM dual;**

TRUNC(TO\_

---------

01-JAN-08

The next example truncates May 25, 2008, to the first day in the month, which is May 1, 2008:

**SELECT TRUNC(TO\_DATE('25-MAY-2008'), 'MM')**

**FROM dual;**

TRUNC(TO\_

---------

01-MAY-08

The next example truncates 7:45:26 P.M. on May 25, 2008, to the hour, which is 7:00 P.M.:

**SELECT TO\_CHAR(TRUNC(TO\_DATE('25-MAY-2008 19:45:26',**

**’DD-MON-YYYY HH24:MI:SS'), 'HH24'), ’DD-MON-YYYY HH24:MI:SS')**

**FROM dual;**

TO\_CHAR(TRUNC(TO\_DAT

--------------------

25-MAY-2008 19:00:00

**USING TIME ZONES**

Oracle Database 9*i* introduced the ability to use different time zones. A time zone is an offset from the time in Greenwich, England. The time in Greenwich was once known as Greenwich Mean Time (GMT), but is now known as Coordinated Universal Time (UTC, which comes from the initials of the French wording).

You specify a time zone using either an offset from UTC or a geographic region (e.g., PST). When you specify an offset, you use HH:MI prefixed with a plus or minus sign:

+|-HH:MI

where

Image + or–indicates an increase or decrease for the offset from UTC.

Image HH:MI specifies the offset in hours and minutes for the time zone.

Image

**NOTE**  
*The time zone hour and minute use the format parameters* TZH *and* TZR, *shown earlier in*[*Table 5-2*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2).

The following example shows offsets of 8 hours behind UTC and 2 hours 15 minutes ahead of UTC:

-08:00

+02:15

You may also specify a time zone using the geographical region. For example, PST indicates Pacific Standard Time, which is 8 hours behind UTC. EST indicates Eastern Standard Time, which is 5 hours behind UTC.

Image

**NOTE**  
*The time zone region uses the format parameter* TZR, *shown earlier in*[*Table 5-2*](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-2).

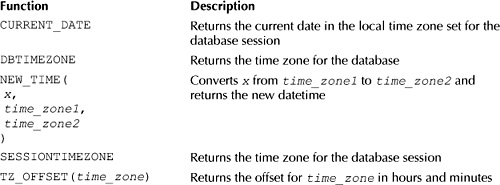
**Time Zone Functions**

There are a number of functions that are related to time zones; these functions are shown in [Table 5-5](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-5).

You’ll learn more about these functions in the following sections.

**The Database Time Zone and Session Time Zone**

If you’re working for a large worldwide organization, the database you access may be located in a different time zone than your local time zone. The time zone for the database is known as the *database time zone*, and the time zone set for your database session is known as the *session time zone*. You’ll learn about the database and session time zones in the following sections.



**TABLE 5-5** *Time Zone Functions*

**The Database Time Zone**

The database time zone is controlled using the TIME\_ZONE database parameter. Your database administrator can change the setting of the TIME\_ZONE parameter in the database’s init.ora or spfile.ora file, or by using ALTER DATABASE SET TIME\_ZONE = *offset* | *region* (e.g., ALTER DATABASE SET TIME\_ZONE = '-8:00' or ALTER DATABASE SET TIME\_ZONE = 'PST').

You can get the database time zone using the DBTIMEZONE function. For example, the following query gets the time zone for my database:

**SELECT DBTIMEZONE**

**FROM dual;**

DBTIME

------

+00:00

As you can see, +00:00 is returned. This means my database uses the time zone set in the operating system, which is set to PST on my computer.

Image

**NOTE**  
*The Windows operating system is typically set up to adjust the clock for daylight savings. For California, this means that in the summer the clock is only 7 hours behind UTC, rather than 8 hours. When I wrote this chapter, I set the date to November 5, 2007, which means my clock is 8 hours behind UTC (I'm located in California).*

**The Session Time Zone**

The session time zone is the time zone for a particular session. By default, the session time zone is the same as the operating system time zone. You can change your session time zone using the ALTER SESSION statement to set the session TIME\_ZONE parameter (e.g., ALTER SESSION SET TIME\_ZONE = 'PST' sets the local time zone to Pacific Standard Time). You can also set the session TIME\_ZONE to LOCAL, which sets the time zone to the one used by the operating system of the computer on which the ALTER SESSION statement was run. You can also set the session TIME\_ZONE to DBTIMEZONE, which sets the time zone to the one used by the database.

You can get the session time zone using the SESSIONTIMEZONE function. For example, the following query gets the time zone for my session:

**SELECT SESSIONTIMEZONE**

**FROM dual;**

SESSIONTIMEZONE

----------------

−08:00

As you can see, my session time zone is 8 hours behind UTC.

**Getting the Current Date in the Session Time Zone**

The SYSDATE function gets the date from the database. This gives you the date in the database time zone. You can get the date in your session time zone using the CURRENT\_DATE function. For example:

**SELECT CURRENT\_DATE**

**FROM dual;**

CURRENT\_D

---------

05-NOV-07

**Obtaining Time Zone Offsets**

You can get the time zone offset hours using the TZ\_OFFSET() function, passing the time zone region name to TZ\_OFFSET(). For example, the following query uses TZ\_OFFSET() to get the time zone offset hours for PST, which is 8 hours behind UTC:

**SELECT TZ\_OFFSET('PST')**

**FROM dual;**

TZ\_OFFS

--------

−08:00

Image

**NOTE**  
*In the summer, this will be −7:00 when using Windows, which sets the clock automatically to adjust for daylight savings*.

**Obtaining Time Zone Names**

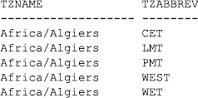
You can obtain all the time zone names by selecting all the rows from v$timezone\_names. To query v$timezone\_names, you should first connect to the database as the system user. The following query shows the first five rows from v$timezone\_names:

**SELECT \***

**FROM v$timezone\_names**

**WHERE ROWNUM <= 5**

**ORDER BY tzabbrev;**



You may use any of the tzname or tzabbrev values for your time zone setting.

Image

**NOTE**  
*The* ROWNUM *pseudo column contains the row number. For example, the first row returned by a query has a row number of 1, the second has a row number of 2, and so on. Therefore, the* WHERE *clause in the previous query causes the query to return the first five rows*.

**Converting a Datetime from One Time Zone to Another**

You use the NEW\_TIME() function to convert a datetime from one time zone to another. For example, the following query uses NEW\_TIME() to convert 7:45 P.M. on May 13, 2008, from PST to EST:

**SELECT TO\_CHAR(NEW\_TIME(TO\_DATE('25-MAY-2008 19:45',**

**’DD-MON-YYYY HH24:MI'), 'PST', 'EST'), ’DD-MON-YYYY HH24:MI')**

**FROM dual;**

TO\_CHAR(NEW\_TIME(

-----------------

25-MAY-2008 22:45

EST is 3 hours ahead of PST: therefore, 3 hours are added to 7:45 P.M. to give 10:45 P.M. (or 22:45 in 24-hour format).

**USING TIMESTAMPS**

Oracle Database 9*i* introduced the ability to store timestamps. A timestamp stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. The advantages of a timestamp over a DATE are

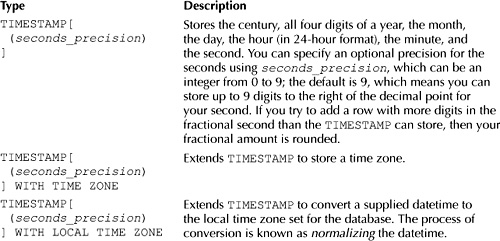
Image A timestamp can store a fractional second.

Image A timestamp can store a time zone.

Let’s examine the timestamp types.

**Using the Timestamp Types**

There are three timestamp types, which are shown in [Table 5-6](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-6).



**TABLE 5-6** *Timestamp Types*

You’ll learn how to use these timestamp types in the following sections.

**Using the TIMESTAMP Type**

As with the other types, you can use the TIMESTAMP type to define a column in a table. The following statement creates a table named purchases\_with\_timestamp that stores customer purchases. This table contains a TIMESTAMP column named made\_on to record when a purchase was made; notice a precision of 4 is set for the TIMESTAMP (this means up to four digits may be stored to the right of the decimal point for the second):

CREATE TABLE purchases\_with\_timestamp (

product\_id INTEGER REFERENCES products(product\_id),

customer\_id INTEGER REFERENCES customers(customer\_id),

made\_on TIMESTAMP(4)

);

Image

**NOTE**  
*The* purchases\_with\_timestamp *table is created and populated with rows by the*store\_schema.sql *script. You’ll see other tables in the rest of this chapter that are also created by the script, so you don’t need to type in the* CREATE TABLE *or any of the* INSERT *statements shown in this chapter*.

To supply a TIMESTAMP literal value to the database, you use the TIMESTAMP keyword along with a datetime in the following format:

TIMESTAMP 'YYYY-MM-DD HH24:MI:SS.SSSSSSSSS'

Notice there are nine S characters after the decimal point, which means you can supply up to nine digits for the fractional second in your literal string. The number of digits you can actually store in a TIMESTAMP column depends on how many digits were set for storage of fractional seconds when the column was defined. For example, up to four digits can be stored in the made\_ on column of the purchases\_with\_timestamptable. If you try to add a row with more than four fractional second digits, your fractional amount is rounded. For example,

2005-05-13 07:15:31.123456789

would be rounded to

2005-05-13 07:15:31.1235

The following INSERT statement adds a row to the purchases\_with\_timestamp table; notice the use of the TIMESTAMP keyword to supply a datetime literal:

INSERT INTO purchases\_with\_timestamp (

product\_id, customer\_id, made\_on

) VALUES (

1, 1, TIMESTAMP '2005-05-13 07:15:31.1234'

);

The following query retrieves the row:

**SELECT \***

**FROM purchases\_with\_timestamp;**

Image

**Using the TIMESTAMP WITH TIME ZONE Type**

The TIMESTAMP WITH TIME ZONE type extends TIMESTAMP to store a time zone. The following statement creates a table named purchases\_timestamp\_with\_tz that stores customer purchases; this table contains a TIMESTAMP WITH TIME ZONE column named made\_on to record when a purchase was made:

CREATE TABLE purchases\_timestamp\_with\_tz (

product\_id INTEGER REFERENCES products(product\_id),

customer\_id INTEGER REFERENCES customers(customer\_id),

made\_on TIMESTAMP(4) WITH TIME ZONE

);

To supply a timestamp literal with a time zone to the database, you simply add the time zone to your TIMESTAMP. For example, the following TIMESTAMP includes a time zone offset of −07:00:

TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'

You may also supply a time zone region, as shown in the following example that specifies PST as the time zone:

TIMESTAMP '2005-05-13 07:15:31.1234 PST'

The following example adds two rows to the purchases\_timestamp\_with\_tz table:

INSERT INTO purchases\_timestamp\_with\_tz (

product\_id, customer\_id, made\_on

) VALUES (

1, 1, TIMESTAMP '2005-05-13 07:15:31.1234 -07:00'

);

INSERT INTO purchases\_timestamp\_with\_tz (

product\_id, customer\_id, made\_on

) VALUES (

1, 2, TIMESTAMP '2005-05-13 07:15:31.1234 PST'

);

The following query retrieves the rows:

**SELECT \***

**FROM purchases\_timestamp\_with\_tz;**

Image

**Using the TIMESTAMP WITH LOCAL TIME ZONE Type**

The TIMESTAMP WITH LOCAL TIME ZONE type extends TIMESTAMP to store a timestamp with the local time zone set for your database. When you supply a timestamp for storage in a TIMESTAMP WITH LOCAL TIME ZONE column, your timestamp is converted—or *normalized*—to the time zone set for the database. When you retrieve the timestamp, it is normalized to the time zone set for your session.

Image

**TIP**  
*You should use* TIMESTAMP WITH LOCAL TIME ZONE *to store timestamps when your organization has a global system accessed throughout the world. This is because* TIMESTAMP WITH LOCAL TIME ZONE *stores a timestamp using the local time where the database is located, but users see the timestamp normalized to their own time zone*.

My database time zone is PST (PST is 8 hours behind UTC), and I want to store the following timestamp in my database:

2005-05-13 07:15:30 EST

EST is 5 hours behind UTC, and the difference between PST and EST is 3 hours (8 −5 = 3). Therefore, to normalize the previous timestamp for PST, 3 hours must be subtracted from it to give the following normalized timestamp:

2005-05-13 04:15:30

This is the timestamp that would be stored in a TIMESTAMP WITH LOCAL TIME ZONE column in my database.

The following statement creates a table named purchases\_with\_local\_tz that stores customer purchases; this table contains a TIMESTAMP WITH LOCAL TIME ZONE column named made\_on to record when a purchase was made:

CREATE TABLE purchases\_with\_local\_tz (

product\_id INTEGER REFERENCES products(product\_id),

customer\_id INTEGER REFERENCES customers(customer\_id),

made\_on TIMESTAMP(4) WITH LOCAL TIME ZONE

);

The following INSERT adds a row to the purchases\_with\_local\_tz table with the made\_on column set to 2005-05-13 07:15:30 EST:

INSERT INTO purchases\_with\_local\_tz (

product\_id, customer\_id, made\_on

) VALUES (

1, 1, TIMESTAMP '2005-05-13 07:15:30 EST'

);

Although the timestamp for the made\_on column is set to 2005-05-13 07:15:30 EST, the actual timestamp stored in my database is 2005-05-13 04:15:30 (the timestamp normalized for PST).

The following query retrieves the row:

**SELECT \***

**FROM purchases\_with\_local\_tz;**

Image

Because my database time zone and session time zone are both PST, the timestamp returned by the query is for PST.

Image

**CAUTION**  
*The timestamp returned by the previous query is normalized for PST. If your database time zone or session time zone are not PST, the timestamp returned when you run the query will be different (it will be normalized for your time zone)*.

If I set the local time zone for my session to EST and repeat the previous query, I get the timestamp normalized for EST:

**ALTER SESSION SET TIME\_ZONE = 'EST';**

Session altered.

**SELECT \***

**FROM purchases\_with\_local\_tz;**

Image

As you can see, the timestamp returned by the query is 13-MAY-05 07.15.30.0000 AM, which is the timestamp normalized for the session time zone of EST. Because EST is three hours ahead of PST, three hours must be added to 13-MAY-05 04:15:30 (the timestamp stored in the database) to give 13-MAY-05 07.15.30 AM (the timestamp returned by the query).

The following statement sets my session time zone back to PST:

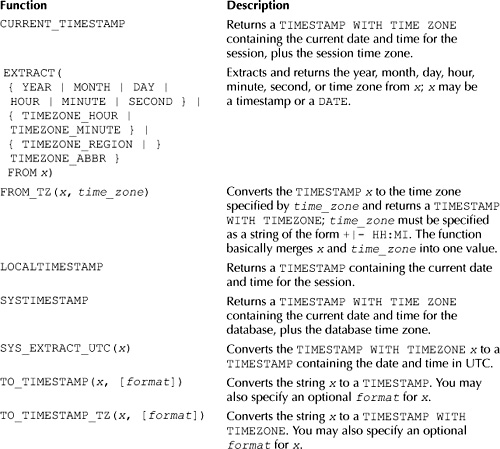
**ALTER SESSION SET TIME\_ZONE = 'PST';**

Session altered.

**Timestamp Functions**

There are a number of functions that allow you to get and process timestamps. These functions are shown in [Table 5-7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-7).

You’ll learn more about the functions shown in [Table 5-7](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-7) in the following sections.



**TABLE 5-7** *Timestamp Functions*

**CURRENT\_TIMESTAMP, LOCALTIMESTAMP, and SYSTIMESTAMP**

The following query calls the CURRENT\_TIMESTAMP, LOCALTIMESTAMP, and SYSTIMESTAMP functions (my session time zone and database time zone are both PST, which is 8 hours behind UTC):

**SELECT CURRENT\_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP**

**FROM dual;**

CURRENT\_TIMESTAMP

-----------------------------------

LOCALTIMESTAMP

-----------------------------------

SYSTIMESTAMP

-----------------------------------

05-NOV-07 12.15.32.734000 PM PST

05-NOV-07 12.15.32.734000 PM

05-NOV-07 12.15.32.734000 PM -08:00

If I change my session TIME\_ZONE to EST and repeat the previous query, I get the following results:

**ALTER SESSION SET TIME\_ZONE = 'EST';**

Session altered.

**SELECT CURRENT\_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP**

**FROM dual;**

CURRENT\_TIMESTAMP

-----------------------------------------------------------

LOCALTIMESTAMP

-----------------------------------------------------------

SYSTIMESTAMP

-----------------------------------------------------------

05-NOV-07 03.19.57.562000 PM EST

05-NOV-07 03.19.57.562000 PM

05-NOV-07 12.19.57.562000 PM -08:00

The following statement sets my session time zone back to PST:

**ALTER SESSION SET TIME\_ZONE = 'PST';**

Session altered.

**EXTRACT()**

EXTRACT() extracts and returns the year, month, day, hour, minute, second, or time zone from *x*; *x* may be a timestamp or a DATE. The following query uses EXTRACT() to get the year, month, and day from a DATEreturned by TO\_DATE():

**SELECT**

**EXTRACT(YEAR FROM TO\_DATE('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS YEAR,**

**EXTRACT(MONTH FROM TO\_DATE('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS MONTH,**

**EXTRACT(DAY FROM TO\_DATE('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS DAY**

**FROM dual;**

Image

The next query uses EXTRACT() to get the hour, minute, and second from a TIMESTAMP returned by TO\_TIMESTAMP():

**SELECT**

**EXTRACT(HOUR FROM TO\_TIMESTAMP('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS HOUR,**

**EXTRACT(MINUTE FROM TO\_TIMESTAMP('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS MINUTE,**

**EXTRACT(SECOND FROM TO\_TIMESTAMP('01-JAN-2008 19:15:26',**

**’DD-MON-YYYY HH24:MI:SS')) AS SECOND**

**FROM dual;**

Image

The following query uses EXTRACT() to get the time zone hour, minute, second, region, and region abbreviation from a TIMESTAMP WITH TIMEZONE returned by TO\_TIMESTAMP\_TZ():

**SELECT**

**EXTRACT(TIMEZONE\_HOUR FROM TO\_TIMESTAMP\_TZ(**

**'01-JAN-2008 19:15:26 -7:15', ’DD-MON-YYYY HH24:MI:SS TZH:TZM'))**

**AS TZH,**

**EXTRACT(TIMEZONE\_MINUTE FROM TO\_TIMESTAMP\_TZ(**

**'01-JAN-2008 19:15:26 -7:15', ’DD-MON-YYYY HH24:MI:SS TZH:TZM'))**

**AS TZM,**

**EXTRACT(TIMEZONE\_REGION FROM TO\_TIMESTAMP\_TZ(**

**'01-JAN-2008 19:15:26 PST', ’DD-MON-YYYY HH24:MI:SS TZR'))**

**AS TZR,**

**EXTRACT(TIMEZONE\_ABBR FROM TO\_TIMESTAMP\_TZ(**

**'01-JAN-2008 19:15:26 PST', ’DD-MON-YYYY HH24:MI:SS TZR'))**

**AS TZA**

**FROM dual;**

Image

**FROM\_TZ()**

FROM\_TZ(*x*, *time\_zone*) converts the TIMESTAMP *x* to the time zone specified by *time\_zone* and returns a TIMESTAMP WITH TIMEZONE; *time\_zone* must be specified as a string of the form +|- HH:MI. The function basically merges *x* and *time\_zone* into one value.

For example, the following query merges the timestamp 2008-05-13 07:15:31.1234 and the time zone offset of −7:00 from UTC:

**SELECT CURRENT\_TIMESTAMP, LOCALTIMESTAMP, SYSTIMESTAMP**

**FROM dual;**

FROM\_TZ(TIMESTAMP'2008-05-1307:15:31.1234','-7:00')

--------------------------------------------------------------------------

13-MAY-08 07.15.31.123400000 AM -07:00

**SYS\_EXTRACT\_UTC()**

SYS\_EXTRACT\_UTC *(x)* converts the TIMESTAMP WITH TIMEZONE *x* to a TIMESTAMP containing the date and time in UTC.

The following query converts 2008-11-17 19:15:26 PST to UTC:

**SELECT SYS\_EXTRACT\_UTC(TIMESTAMP '2008-11-17 19:15:26 PST')**

**FROM dual;**

SYS\_EXTRACT\_UTC(TIMESTAMP'2008-11-1719:15:26PST')

-------------------------------------------------

18-NOV-08 02.15.26.000000000 AM

Because PST is 8 hours behind UTC in the winter, the query returns a TIMESTAMP 8 hours ahead of 2008-11-17 19:15:26 PST, which is 18-NOV-08 03.15.26 AM.

For a date in the summer, the returned TIMESTAMP is only 7 hours ahead of UTC:

**SELECT SYS\_EXTRACT\_UTC(TIMESTAMP '2008-05-17 19:15:26 PST')**

**FROM dual;**

SYS\_EXTRACT\_UTC(TIMESTAMP'2008-05-1719:15:26PST')

-------------------------------------------------

18-MAY-08 02.15.26.000000000 AM

**TO\_TIMESTAMP()**

TO\_TIMESTAMP(*x*, *format*) converts the string *x* (which may be a CHAR, VARCHAR2, NCHAR, or NVARCHAR2) to a TIMESTAMP. You may also specify an optional *format* for *x*.

The following query converts the string 2005-05-13 07:15:31.1234 with the format YYYY-MM-DD HH24:MI:SS.FF to a TIMESTAMP:

**SELECT TO\_TIMESTAMP('2008-05-13 07:15:31.1234', 'YYYY-MM-DD HH24:MI:SS.FF')**

**FROM dual;**

TO\_TIMESTAMP('2008-05-1307:15:31.1234','YYYY-MM-DDHH24:MI:SS.FF')

-----------------------------------------------------------------

13-MAY-08 07.15.31.123400000 AM

**TO\_TIMESTAMP\_TZ()**

TO\_TIMESTAMP\_TZ(*x*, [*format*]) converts *x* to a TIMESTAMP WITH TIMEZONE. You may specify an optional *format* for *x*.

The following query passes the PST time zone (identified using TZR in the format string) to TO\_TIMESTAMP\_TZ():

**SELECT TO\_TIMESTAMP\_TZ('2008-05-13 07:15:31.1234 PST',**

**'YYYY-MM-DD HH24:MI:SS.FF TZR')**

**FROM dual;**

TO\_TIMESTAMP\_TZ('2008-05-1307:15:31.1234PST','YYYY-MM-DDHH24:MI:SS.FFTZR')

--------------------------------------------------------------------------

13-MAY-08 07.15.31.123400000 AM PST

The next query uses a time zone offset of −7:00 from UTC (−7:00 is identified using TZR and TZM in the format string):

**SELECT TO\_TIMESTAMP\_TZ('2008-05-13 07:15:31.1234 -7:00',**

**'YYYY-MM-DD HH24:MI:SS.FF TZH:TZM')**

**FROM dual;**

TO\_TIMESTAMP\_TZ('2008-05-1307:15:31.1234-7:00','YYYY-MM-DDHH24:MI:SS.FFTZH

---------------------------------------------

13-MAY-08 07.15.31.123400000 AM -07:00

**Converting a String to a TIMESTAMP WITH LOCAL TIME ZONE**

You can use the CAST() function to convert a string to a TIMESTAMP WITH LOCAL TIME ZONE. You were introduced to CAST() in the previous chapter. As a reminder, CAST(*x* AS *type*) converts *x* to a compatible database type specified by *type*.

The following query uses CAST() to convert the string 13-JUN-08 to a TIMESTAMP WITH LOCAL TIME ZONE:

**SELECT CAST('13-JUN-08' AS TIMESTAMP WITH LOCAL TIME ZONE)**

**FROM dual;**

CAST('13-JUN-08'ASTIMESTAMPWITHLOCALTIMEZONE)

---------------------------------------------

13-JUN-08 12.00.00.000000 AM

The timestamp returned by this query contains the date June 13, 2008 and the time of 12 A.M.

The next query uses CAST() to convert a more complex string to a TIMESTAMP WITH LOCAL TIME ZONE:

**SELECT CAST(TO\_TIMESTAMP\_TZ('2008-05-13 07:15:31.1234 PST',**

**'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)**

**FROM dual;**

CAST(TO\_TIMESTAMP\_TZ('2008-05-1307:15:31.1234PST','YYYY-MM-DDHH24:MI:SS.FF

--------------------------------------------------------------------------

13-MAY-08 06.15.31.123400 AM

The timestamp returned by this query contains the date May 13, 2008 and the time of 6:15:31.1234 AM PST (PST is the time zone for both my database and session).

The following query does the same thing as the previous one, except this time EST is the time zone:

**SELECT CAST(TO\_TIMESTAMP\_TZ('2008-05-13 07:15:31.1234 EST',**

**'YYYY-MM-DD HH24:MI:SS.FF TZR') AS TIMESTAMP WITH LOCAL TIME ZONE)**

**FROM dual;**

CAST(TO\_TIMESTAMP\_TZ('2008-05-1307:15:31.1234EST','YYYY-MM-DDHH24:MI:SS.FF

--------------------------------------------------------------------------

13-MAY-08 04.15.31.123400 AM

The timestamp returned by this query contains the date May 13, 2008 and the time of 4:15:31.1234 AM PST (because PST is 3 hours behind EST, the time returned in the timestamp is 3 hours earlier than the time in the actual query).

**USING TIME INTERVALS**

Oracle Database 9*i* introduced data types that allow you to store time *intervals*. Examples of time intervals include

Image 1 year 3 months

Image 25 months

Image −3 days 5 hours 16 minutes

Image 1 day 7 hours

Image −56 hours

Image

**NOTE**  
*Do not confuse time intervals with datetimes or timestamps. A time interval records a length of time (e.g., 1 year 3 months), whereas a datetime or timestamp records a specific date and time (e.g., 7:32:16 P.M. on October 28, 2006)*.

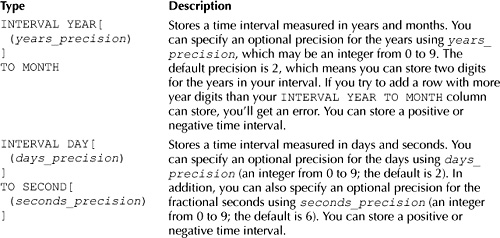
In our imaginary online store, you might want to offer limited time discounts on products. For example, you could give customers a coupon that is valid for a few months, or you could run a special promotion for a few days. You’ll see examples that feature coupons and promotions later in this section.

[Table 5-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-8) shows the interval types.

You’ll learn how to use the time interval types in the following sections.

**Using the INTERVAL YEAR TO MONTH Type**

INTERVAL YEAR TO MONTH stores a time interval measured in years and months. The following statement creates a table named coupons that stores coupon information. The coupons table contains an INTERVAL YEAR TO MONTH column named duration to record the interval of



**TABLE 5-8** *Time Interval Types*

time for which the coupon is valid; notice that I’ve provided a precision of 3 for the duration column, which means that up to three digits may be stored for the year:

CREATE TABLE coupons (

coupon\_id INTEGER CONSTRAINT coupons\_pk PRIMARY KEY,

name VARCHAR2(30) NOT NULL,

duration INTERVAL YEAR(3) TO MONTH

);

To supply an INTERVAL YEAR TO MONTH literal value to the database, you use the following simplified syntax:

INTERVAL '[+|-][y][-m]' [YEAR[(*years\_precision*)])] [TO MONTH]

where

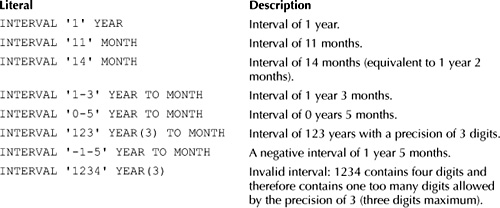
Image + or - is an optional indicator that specifies whether the time interval is positive or negative (the default is positive).

Image *y* is the optional number of years for the interval.

Image *m* is the optional number of months for the interval. If you supply years *and* months, you must include TO MONTH in your literal.

Image *years\_precision* is the optional precision for the years (the default is 2).

The following table shows some examples of year-to-month interval literals.



The following INSERT statements add rows to the coupons table with the duration column set to some of the intervals shown in the previous table:

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (1, '$1 off Z Files', INTERVAL '1' YEAR);

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (2, '$2 off Pop 3', INTERVAL '11' MONTH);

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (3, '$3 off Modern Science', INTERVAL '14' MONTH);

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (4, '$2 off Tank War', INTERVAL '1-3' YEAR TO MONTH);

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (5, '$1 off Chemistry', INTERVAL '0-5' YEAR TO MONTH);

INSERT INTO coupons (coupon\_id, name, duration)

VALUES (6, '$2 off Creative Yell', INTERVAL '123' YEAR(3));

If you try to add a row with the duration column set to the invalid interval of INTERVAL '1234' YEAR(3), you’ll get an error because the precision of the duration column is 3 and is therefore too small to accommodate the number 1234. The following INSERT shows the error:

SQL> **INSERT INTO coupons (coupon\_id, name, duration)**

**2 VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3));**

VALUES (7, '$1 off Z Files', INTERVAL '1234' YEAR(3))

\*

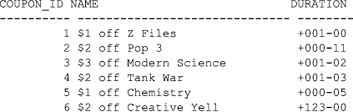
ERROR at line 2:

ORA-01873: the leading precision of the interval is too small

The following query retrieves the rows from the coupons table; notice the formatting of the duration values:

**SELECT \***

**FROM coupons;**



**Using the INTERVAL DAY TO SECOND Type**

INTERVAL DAY TO SECOND stores time intervals measured in days and seconds. The following statement creates a table named promotions that stores promotion information. The promotions table contains an INTERVAL DAY TO SECOND column named duration to record the interval of time for which the promotion is valid:

CREATE TABLE promotions (

promotion\_id INTEGER CONSTRAINT promotions\_pk PRIMARY KEY,

name VARCHAR2(30) NOT NULL,

duration INTERVAL DAY(3) TO SECOND (4)

);

Notice I’ve provided a precision of 3 for the day and a precision of 4 for the fractional seconds of the durationcolumn. This means that up to three digits may be stored for the day of the interval and up to four digits to the right of the decimal point for the fractional seconds.

To supply an INTERVAL DAY TO SECOND literal value to the database, you use the following simplified syntax:

INTERVAL '[+|-][*d*] [*h*[:*m*[:*s*]]]' [DAY[(*days\_precision*)]])

[TO HOUR | MINUTE | SECOND[(*seconds\_precision*)]]

where

Image + or - is an optional indicator that specifies whether the time interval is positive or negative (the default is positive).

Image *d* is the number of days for the interval.

Image *h* is the optional number of hours for the interval; if you supply days and hours, you must include TO HOUR in your literal.

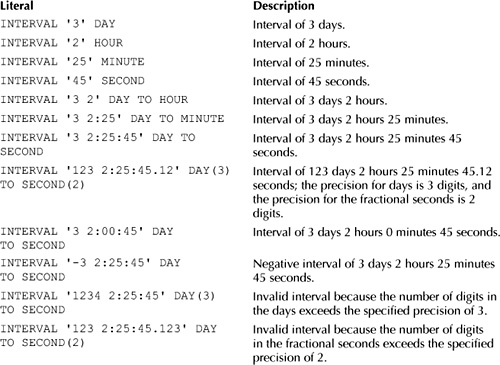
Image *m* is the optional number of minutes for the interval; if you supply days and minutes, you must include TO MINUTES in your literal.

Image *s* is the optional number of seconds for the interval; if you supply days and seconds, you must include TO SECOND in your literal.

Image *days\_precision* is the optional precision for the days (the default is 2).

Image *seconds\_precision* is the optional precision for the fractional seconds (the default is 6).

The following table shows some examples of day-to-second interval literals.



The following INSERT statements add rows to the promotions table:

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (1, '10%$ off Z Files', INTERVAL '3' DAY);

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (2, '20%$ off Pop 3', INTERVAL '2' HOUR);

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (3, '30%$ off Modern Science', INTERVAL '25' MINUTE);

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (4, '20%$ off Tank War', INTERVAL '45' SECOND);

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (5, '10%$ off Chemistry', INTERVAL '3 2:25' DAY TO MINUTE);

INSERT INTO promotions (promotion\_id, name, duration)

VALUES (6, '20%$ off Creative Yell', INTERVAL '3 2:25:45' DAY TO SECOND);

INSERT INTO promotions (promotion\_id, name, duration)

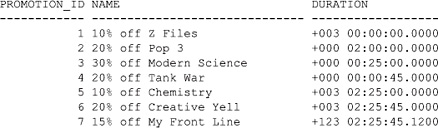
VALUES (7, '15%$ off My Front Line',

INTERVAL '123 2:25:45.12' DAY(3) TO SECOND(2));

The following query retrieves the rows from the promotions table; notice the formatting of the durationvalues:

**SELECT \***

**FROM promotions;**



**Time Interval Functions**

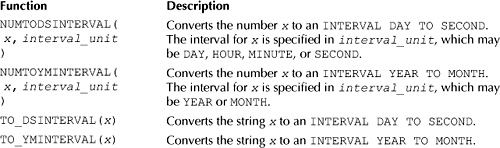
There are a number of functions that allow you to get and process time intervals; these functions are shown in [Table 5-9](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-9).

You’ll learn more about the functions shown in [Table 5-9](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch05.html#table_5-9) in the following sections.

**NUMTODSINTERVAL()**

NUMTODSINTERVAL(*x*, *interval\_unit*) converts the number *x* to an INTERVAL DAY TO SECOND. The interval for *x* is specified in *interval\_unit*, which may be DAY, HOUR, MINUTE, or SECOND.

For example, the following query converts several numbers to time intervals using NUMTODSINTERVAL():



**TABLE 5-9** *Time Interval Functions*

**SELECT**

**NUMTODSINTERVAL(1.5, ’DAY'),**

**NUMTODSINTERVAL(3.25, 'HOUR'),**

**NUMTODSINTERVAL(5, 'MINUTE'),**

**NUMTODSINTERVAL(10.123456789, ’SECOND')**

**FROM dual;**

NUMTODSINTERVAL(1.5,’DAY')

----------------------------------------------

NUMTODSINTERVAL(3.25,'HOUR')

----------------------------------------------

NUMTODSINTERVAL(5,'MINUTE')

----------------------------------------------

NUMTODSINTERVAL(10.123456789,’SECOND')

----------------------------------------------

+000000001 12:00:00.000000000

+000000000 03:15:00.000000000

+000000000 00:05:00.000000000

+000000000 00:00:10.123456789

**NUMTOYMINTERVAL()**

NUMTOYMINTERVAL(*x*, *interval\_unit*) converts the number *x* to an INTERVAL YEAR TO MONTH. The interval for *x* is specified in *interval\_unit*, which may be YEAR or MONTH.

For example, the following query converts two numbers to time intervals using NUMTOYMINTERVAL():

**SELECT**

**NUMTOYMINTERVAL(1.5, 'YEAR'),**

**NUMTOYMINTERVAL(3.25, 'MONTH')**

**FROM dual;**

NUMTOYMINTERVAL(1.5,'YEAR')

---------------------------

NUMTOYMINTERVAL(3.25,'MONTH')

-----------------------------

+000000001-06

+000000000-03

**SUMMARY**

In this chapter, you learned the following:

Image You may store a datetime using the DATE type. The DATE type stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second.

Image You may use TO\_CHAR() and TO\_DATE() to convert between strings and dates and times.

Image The Oracle database always stores all four digits of a year and interprets two-digit years using a set of rules. You should always specify all four digits of the year.

Image There are a number of functions that process dates and times. An example is ADD\_MONTHS(*x*, *y*), which returns the result of adding *y* months to *x*.

Image Oracle Database 9*i* introduced the ability to use different time zones. A time zone is an offset from the time in Greenwich, England. The time in Greenwich was once known as Greenwich Mean Time (GMT), but it is now known as Coordinated Universal Time (UTC). You specify a time zone using either an offset from UTC or the name of the region (e.g., PST).

Image Oracle Database 9*i* introduced the ability to store timestamps. A timestamp stores the century, all four digits of a year, the month, the day, the hour (in 24-hour format), the minute, and the second. The advantages of a timestamp over a DATE are a timestamp can store a fractional second and a time zone.

Image Oracle Database 9*i* introduced the ability to handle time intervals, which allow you to store a length of time. An example time interval is 1 year 3 months.