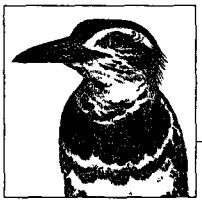

# 11

# String Functions

MySQL has several built-in functions for formatting, manipulating, and analyzing strings, both user-specified and within columns of data. This chapter lists these string functions, provides the syntax of each, and gives examples of their use. The examples in this chapter use a fictitious database for a college.

String functions do not change their inputs; the functions' return values contain the changes.

## String Functions Grouped by Type

The list of string functions is quite long, but many perform similar roles. The following list groups the functions by these roles.

### Character Sets and Collation

`CHARSET( )`, `COALESCE( )`, `COERCIBILITY( )`, `COLLATION( )`.

### Converting

`ASCII( )`, `BIN( )`, `BINARY`, `CAST( )`, `CHAR( )`, `COMPRESS( )`, `CONVERT( )`, `EXPORT_SET( )`, `HEX( )`, `MAKE_SET( )`, `ORD( )`, `SOUNDEX( )`, `UNCOMPRESS( )`, `UNHEX( )`.

### Formatting

`CONCAT( )`, `CONCAT_WS( )`, `LCASE( )`, `LENGTH( )`, `LOWER( )`, `LPAD( )`, `LTRIM( )`, `OCTET_LENGTH( )`, `QUOTE( )`, `RPAD( )`, `RTRIM( )`, `SPACE( )`, `TRIM( )`, `UCASE( )`, `UPPER( )`.

### Expressions

BIT_LENGTH( ), CRC32( ), CHAR_LENGTH( ), CHARACTER_LENGTH( ), ELT( ), FIELD( ), FIND_IN_SET( ), INSTR( ), INTERVAL( ), LOCATE( ), MATCH( ) AGAINST( ), POSITION( ), STRCMP( ), UNCOMPRESSED_LENGTH( ).

### Extracting

LEFT( ), LOAD_FILE( ), MID( ), RIGHT( ), SUBSTR( ), SUBSTRING( ), SUBSTRING_INDEX( ).

### Manipulating

INSERT( ), REPEAT( ), REPLACE( ), REVERSE( ).

## String Functions in Alphabetical Order

The rest of this chapter lists the string functions in alphabetical order.

---

### ASCII( )

ASCII(*string*)

This function returns the numeric code corresponding to the first character of a given string. If the given string is empty, 0 is returned. Despite the function's name, it works for characters outside the ASCII set (that is, characters that correspond to values above 127) and is probably most useful for such characters.

As an example of this function's use, suppose that for a college we had a table listing the names of fraternities with their Greek letters. For easier manipulation of the data contained in a column, we might want to convert the Greek letters to a numeric code with this function:

```
SELECT greek_id,
CONCAT_WS('-',
   ASCII( SUBSTR(greek_id, 1, 1) ),
   ASCII( SUBSTR(greek_id, 2, 1) ),
   ASCII( SUBSTR(greek_id, 3, 1) )
) AS 'ASCII Values'
FROM fraternities WHERE frat_id = 101;

+----------+--------------+
| greek_id | ASCII Values |
+----------+--------------+
| ΔΣΠ      | 196-211-208  |
+----------+--------------+
```

In this example, we use the SUBSTR( ) function to extract each letter so we can then convert each one individually to its numeric equivalent with the ASCII( ) function. Then, using CONCAT_WS( ), we insert hyphens between each number returned. We can use this number to more easily manage the data related to this fraternity. See the descriptions of CHAR( )

and CONVERT( ) later in this chapter for more information on this function and for more details related to this example.

## BIN( )

BIN(*number*)

This function returns a binary number for a given integer. It returns NULL if the input is NULL:

```
SELECT BIN(1), BIN(2), BIN(3);

+--------+--------+--------+
| BIN(1) | BIN(2) | BIN(3) |
+--------+--------+--------+
| 1      | 10     | 11     |
+--------+--------+--------+
```

For the number 1 in a base 10 system, the first position in a binary system is *on*, or 1. For the number 2, the first position from the right is *off* and the second is *on*. For 3, the first and the second positions are *on*.

## BINARY

BINARY *string*

Use this function to treat strings in their binary state. This function is useful for making SQL statements case-sensitive. Notice that the syntax does not call for parentheses:

```
SELECT student_id, name_last
FROM students
WHERE BINARY LEFT(UCASE(name_last), 1) <>
   LEFT(name_last, 1);

+------------+-----------+
| student_id | name_last |
+------------+-----------+
|  433302000 | dyer      |
|  434016005 | de Vitto  |
+------------+-----------+
```

This statement checks for any student whose last name starts with a lowercase letter. Each student's last name is converted to uppercase letters, and then the first letter starting from the left is extracted to be compared with the first letter of the last name without case conversion. The results show one record that is probably a typing error and a second that is probably correct. Notice that the BINARY keyword is specified before the comparison is made between the strings, and is applied to both strings.

## BIT_LENGTH( )

BIT_LENGTH(*string*)

This function returns the number of bits in a given string. The following example uses the default character set, where one character requires 8 bits:

CAST( )

```
SELECT BIT_LENGTH('a') AS 'One Character',
BIT_LENGTH('ab') AS 'Two Characters';

+---------------+----------------+
| One Character | Two Characters |
+---------------+----------------+
|             8 |             16 |
+---------------+----------------+
```

## CAST( )

CAST(*expression* AS *type* [CHARACTER SET *character_set*])

Use this function to convert a value from one data type to another. This function is available as of version 4.0.2 of MySQL. The data type given as the second argument can be BINARY, CHAR, DATE, DATETIME, SIGNED [INTEGER], TIME, or UNSIGNED [INTEGER]. BINARY converts a string to a binary string.

CHAR conversion is available as of version 4.0.6 of MySQL. This function is similar to CONVERT( ). Optionally, you can add CHARACTER SET to use a different character set from the default for the value given. The default is drawn from the system variables character_set_connection and collation_connection.

As an example, suppose we want to retrieve a list of courses for the current semester (Spring) and their locations, sorting them alphabetically by their building name. Unfortunately, the building names are in an ENUM( ) column because we're at a small college. Since they're not in alphabetical order in the column definition, they won't be sorted the way we want. Instead, they will be sorted in the lexical order of the column definition, that is, the order they are listed in the ENUM( ) column of the table definition. Using CAST( ) in the WHERE clause can resolve this:

```
SELECT course_id, course_name,
CONCAT(building, '-', room_num) AS location
FROM courses
WHERE year = YEAR(CURDATE())
AND semester = 'spring'
ORDER BY CAST(building AS CHAR);
```

By using the CAST( ) function to treat the values of building as a CHAR data type, we make sure the results will be ordered alphabetically.

## CHAR( )

CHAR(*ascii*[, ...] [USING *character_set*])

This function returns a string corresponding to the numeric code passed as the argument. This is the reverse of ASCII( ), described earlier in this chapter. You can optionally give the USING parameter to specify a different character set to use in relation to the string given. If you give it a value greater than 255, it assumes the amount over 255 is another character. So, CHAR(256) is equivalent to CHAR(1,0).

As an example of this function's use, suppose that a college database has a table for fraternities on campus and that the table has a column to contain the Greek letters for

each fraternity's name. To create a table with such a column, we would at a minimum enter something like the following:

```
CREATE TABLE fraternities (
frat_id INT(11),
greek_id CHAR(10) CHARACTER SET greek);
```

Notice that for the column greek_id we're specifying a special character set to be used. This can be different from the character set for other columns and for the table. With this minimal table, we enter the following INSERT statement to add one fraternity and then follow that with a SELECT statement to see the results:



```
INSERT INTO fraternities
VALUES(101,
   CONCAT(CHAR(196 USING greek),
   CHAR(211 USING greek),
   CHAR(208 USING greek)));

SELECT greek_id
FROM fraternities
WHERE frat_id = 101;

+----------+
| greek_id |
+----------+
| ΔΣΠ      |
+----------+
```

Using the CHAR( ) function and looking at a chart showing the Greek alphabet, we figure out the ASCII number for each of the three Greek letters for the fraternity Delta Sigma Pi. If we had a Greek keyboard, we could just type them. If we used a chart available online in a graphical browser, we could just copy and paste them into our *mysql* client. Using the CONCAT( ) function, we put the results of each together to insert the data into the column in the table.

## CHAR_LENGTH( )

CHAR_LENGTH(*string*)

This function returns the number of characters in a given string. This is synonymous with CHARACTER_LENGTH( ). A multiple-byte character is treated as one character. Use LENGTH( ) if you want each byte to be counted. Here is an example:

```
SELECT course_id,
   CASE
   WHEN CHAR_LENGTH(course_desc) > 30
   THEN CONCAT(SUBSTRING(course_desc, 1, 27), '...')
   ELSE course_desc
   END AS Description
FROM courses;
```

In this example, a CASE control statement is used to specify different display results based on a condition. Using the CHAR_LENGTH( ) function, MySQL determines whether the content of course_desc is longer than 30 characters. If it is, the SUBSTRING( ) function extracts the first 27 characters and the CONCAT( ) function adds ellipsis points to the end of the truncated data to indicate that there is more text. Otherwise, the full contents of

course_desc are displayed. See the CHARACTER_LENGTH( ) description next for another example of how CHAR_LENGTH( ) may be used.

## CHARACTER_LENGTH( )

CHARACTER_LENGTH(*string*)

This function returns the number of characters of a given string. A multiple-byte character is treated as one character. It's synonymous with CHAR_LENGTH( ).

As another example of how this function or CHAR_LENGTH( ) might be used, suppose that in a college's table containing students names we notice that some of the names appear garbled. We realize this is happening because we weren't prepared for non-Latin characters. We could enter an SQL statement like the following to find students with the names containing multibyte characters:

```
SELECT student_id,
CONCAT(name_first, SPACE(1), name_last) AS Name
FROM students
WHERE CHARACTER_LENGTH(name_first) != LENGTH(name_first)
OR CHARACTER_LENGTH(name_last) != LENGTH(name_last);
```

In this example, in the WHERE clause we're using CHARACTER_LENGTH( ) to get the number of bytes and LENGTH( ) to get the number of characters for each name, and then we're comparing them with the != operator to return only rows where the two methods of evaluation don't equal.

## CHARSET( )

CHARSET(*string*)

This function returns the character set used by a given string. It's available as of version 4.1.0 of MySQL. Here is an example:

```
SELECT CHARSET('Rosá')
AS 'Set for My Name';

+-----------------+
| Set for My Name |
+-----------------+
| utf8            |
+-----------------+
```

## COALESCE( )

COALESCE(*column*[, ...])

This function returns the leftmost non-NULL string or column in a comma-separated list. If all elements are NULL, the function returns NULL. Here is an example:

```
SELECT CONCAT(name_first, ' ', name_last)
  AS Student,
COALESCE(phone_dorm, phone_home, 'No Telephone Number')
  AS Telephone
FROM students;
```

In this example, the results will show the student's dormitory telephone number if there is one (i.e., if the student lives in the dormitory). If not, it will show the student's home telephone number (i.e., maybe his parent's house). Otherwise, it will return the string given, indicating that there is no telephone number for the student.

## COERCIBILITY( )



`COERCIBILITY(`*`string`*`)`

This function returns an arbitrary value known as the *coercibility* of a given string or other item, showing how likely that item is to determine the collation used in an expression. MySQL sometimes needs to choose which collation to use when results of an SQL statement involve different types of data. Here are possible return values from this function:

*0*
: Collation has been explicitly specified (e.g., a statement using `COLLATE`).

*1*
: The argument merges values of different collations.

*2*
: The argument has an implicit collation (e.g., a column is given).

*3*
: The argument is a system constant, such as a system variable or a function that returns something similar.

*4*
: The argument is a literal string.

*5*
: The argument is NULL or an expression derived from a NULL value.

Lower coercibility levels take precedence over higher ones when the server is determining which collation to use. This function is available as of version 4.1.1 of MySQL. Here is an example:

```
SELECT COERCIBILITY('Russell')
AS 'My State';

+----------+
| My State |
+----------+
|        4 |
+----------+
```

## COLLATION( )

`COLLATION(`*`string`*`)`

This function returns the collation for the character set of a given string. This function is available as of version 4.1.0 of MySQL. Here is an example:

```
SELECT COLLATION('Rosá');
```

```
+--------------------+
| COLLATION('Rosá')  |
+--------------------+
| utf8_general_ci    |
+--------------------+
```

## COMPRESS( )

COMPRESS(*string*)

This function returns a given string after compressing it. It requires MySQL to have been compiled with a compression library (e.g., `zlib`). If it wasn't, a NULL value will be returned. This statement is available as of version 4.1 of MySQL. Here is an example:

```
UPDATE students_records
SET personal_essay =
(SELECT COMPRESS(essay)
 FROM student_applications
 WHERE applicant_id = '7382') AS derived1
WHERE student_id = '433302000';
```

If you want to store a value that was compressed with this function, it's best to store it in a `BLOB` column, since the results are binary. Use `UNCOMPRESS( )` to uncompress a string that was compressed with this function.

## CONCAT( )

CONCAT(*string, ...*)

With this function, strings or columns can be concatenated or pasted together into one resulting field. Any number of strings may be specified, with each argument separated by a comma. If any of the values given are NULL, a NULL value is returned. Here is an example:

```
SELECT CONCAT(name_first, ' ', name_last) AS Student
FROM students WHERE name_last = 'Dyer';
```

```
+--------------------+
| Student            |
+--------------------+
| Kenneth Dyer       |
| Geoffrey Dyer      |
| Marie Dyer         |
| NULL               |
+--------------------+
```

In this example, the database contained four students with the last name *Dyer*, but one of them had a NULL value in the `name_first` column. Within the parentheses of the function, notice that a space is given within quotes as the second element so that the results show a space between each student's first and last name.

Another use for `CONCAT( )` is to convert numeric values of a given column to strings. This may be useful when working with an API such as Perl and when using `UNION` to mix data from two different data types.

Here is an example:

```
SELECT CONCAT(type_id) AS id, type AS title
  FROM types
UNION
SELECT topic_id AS id, topic AS title
  FROM topics;
```

In this example, the column type_id is an INT, whereas the column topic_id is a CHAR column. In MySQL, the results can be mixed. However, if this SQL statement is used to create a hash of data in Perl or another API language, you may encounter problems retrieving data. In order that the data in the columns agree, the CONCAT( ) function is used to convert the numeric values to their string equivalents.



## CONCAT_WS( )

CONCAT_WS(*separator, string, ...*)

This function combines strings of text and columns, separated by the string specified in the first argument. Any number of strings may be specified after the first argument, with each argument separated by a comma. Null values are ignored. Here is an example:

```
SELECT CONCAT_WS('|', student_id, name_last, name_first)
AS 'Dyer Students'
FROM students
WHERE name_last='Dyer';

+-----------------------+
| Dyer Students         |
+-----------------------+
| 433342000|Dyer|Russell |
| 434892001|Dyer|Marie   |
+-----------------------+
```

Here, the vertical bar is used to separate the columns. This function can be useful for exporting data to formats acceptable to other software. You could incorporate something like this into an API program, or just execute it from the command line using the *mysql* client like this:

```
mysql -u root -p \
-e "SELECT CONCAT_WS('|', student_id, name_last, name_first)
AS '# Dyer Students #' FROM testing.students
WHERE name_last='Dyer';" > dyer_students.txt

cat dyer_students.txt

# Dyer Students #
433342000|Dyer|Russell
434892001|Dyer|Marie
```

The -e option in the *mysql* client instructs it to execute what is contained in quotes. The entire mysql statement is followed by a > sign to redirect output to a text file. Afterward, the cat command shows the contents of that file. Notice that the usual ASCII table format is not included. This makes the file easy to import into other applications.

## CONVERT( )

CONVERT([*_character_set*]*string* USING *character_set*)

Use this function to convert the character set of a given string to another character set specified with the USING keyword. This function is available as of version 4.0.2 of MySQL. The function has some similarities to CAST( ). If the character set for the given string is not the same as the default, you can specify its character set by listing it immediately before the string and preceded by an underscore:

```
UPDATE students SET name_first =
CONVERT(_latin1'Rosá' USING utf8)
WHERE student_id = 433342000;
```

In this example, we're converting the student's first name with the accented character into a format usable by the column that uses UTF-8. Notice that the character set given for the string is preceded by an underscore and there are no spaces before the quotation mark for the string.

## CRC32( )

CRC32(*string*)

This function returns the given string's cyclic redundancy check (CRC) value as a 32-bit unsigned value. It's available as of version 4.1 of MySQL. It returns NULL if given a NULL value. Even if a numeric value is given, it treats the value as a string:

```
SELECT CRC32('test');

+---------------+
| CRC32('test') |
+---------------+
|    3632233996 |
+---------------+
```

## ELT( )

ELT(*index, string, ...*)

This function returns the index element from the list of strings given, where the list is numbered starting with 1. If the number given is less than 1 or if the number of elements is less than the number given, this statement returns NULL:

```
SELECT student_id,
CONCAT(name_first, SPACE(1), name_last)
   AS Name,
ELT(primary_phone, phone_dorm, phone_home, phone_work)
   AS Telephone
FROM students;
```

In this SQL statement, we're using the value of the primary_phone column to provide the index for ELT( ). This column is an ENUM column that records which of the three telephone columns is the student's primary telephone number. The function will return the value for the column selected based on the index. As a result, the SQL statement will give a list of students and their primary telephone numbers.

## EXPORT_SET( )

EXPORT_SET(*number, on, off*[*, separator*,[*count*]])

This function returns a series of strings in order that represent each bit of a given *number*. The second argument specifies a string to represent bits that are 1 (an *on* bit), and the third argument specifies a string to represent bits that are 0 (an *off* bit). The fourth argument may specify a separator, and the last argument may specify a number of bit equivalents to display. The default separator is a comma. Here is an example:

```
SELECT BIN(4) AS 'Binary Number',
EXPORT_SET(4, 'on', 'off', '-', 8)
AS 'Verbal Equivalent';

+--------------+-------------------------------+
| Binary Number | Verbal Equivalent             |
+--------------+-------------------------------+
| 100          | off-off-on-off-off-off-off-off |
+--------------+-------------------------------+
```

Notice that the lowest-order bit is displayed first, so the conversion of the binary equivalent of 4 is displayed by EXPORT_SET( ) in what one might consider reverse order, from right to left: not 100, but 001 (or, as part of 8 bits, 00100000).

## FIELD( )

FIELD(*string, string*[*, ...*])

This function searches for the first string given in the following list of strings, and returns the numeric position of the first string in the list that matches. The first element is 1 among the arguments being searched. If the search string is not found or is NULL, 0 is returned.

As an example of this function, suppose that in a table containing telephone numbers of students at a college, there are three columns for telephone numbers (dormitory, home, and work numbers). Suppose further that another column is used to indicate which column contains the primary telephone number of the student. However, we realize that for many rows this primary_phone column is NULL. So, we decide to make a guess as to which is the primary telephone number by using the FIELD( ) function along with a subquery:

```
UPDATE students
JOIN
  (SELECT student_id,
   FIELD(1, phone_dorm IS TRUE,
            phone_home IS TRUE,
            phone_work IS TRUE)
   AS first_phone_found
   FROM students
   WHERE primary_phone IS NULL) AS sub_table
   USING (student_id)
SET primary_phone = first_phone_found;
```

Notice that in the subquery, within the FIELD( ) function, we're looking for a value of 1 (the first parameter of the function). For the other parameters given, each telephone

column will be examined using the IS TRUE operator: it will return true (or rather 1) if the column is not NULL. The FIELD( ) function will return the number of the element in the list that returns 1 (meaning it exists). So if phone_dorm is NULL but phone_home has a telephone number in it, the subquery will return a value of 2—even if phone_work also contains a number. The JOIN uses the results to update each student record that has a NULL value for primary_phone with the value of the first_phone_found field in the results of the subquery.

## FIND_IN_SET( )

FIND_IN_SET(*string, string_list*)

This function returns the location of the first argument within a comma-separated list that is passed as a single string in the second argument. The first element of the list is 1. A 0 is returned if the string is not found in the set or if the string list is empty. It returns NULL if either argument is NULL.

As an example of how this function might be used, suppose that a table in our college application contains the results of a survey that students took on the college's web site. One of the columns, favorite_activities, contains a list of activities each student said is her favorite in the order that she likes them, her favorite being first. The text of the column comes from a web form on which students entered a number to rank each activity they like; they left blank the ones they don't take part in. So, each column has text separated by commas and spaces (e.g., *bike riding, reading, swimming*). Here's how this function could be used to order a list of students who said that *reading* is one of their favorite activities:

```
SELECT student_id,
FIND_IN_SET('reading',
   REPLACE(favorite_activities, SPACE(1), '') )
   AS reading_rank
FROM student_surveys
WHERE survey_id = 127
AND favorite_activities LIKE '%reading%'
ORDER BY reading_rank;
```

We use the WHERE clause to choose the correct survey and the LIKE operator to select only rows where the column favorite_activities contains the value *reading*. This will eliminate those students who didn't rank reading as a favorite activity from the results. FIND_IN_SET( ) won't allow spaces because they confuse the function, so we need to remove spaces from the text in the favorite_activities column. Thus, we slip in a call to REPLACE( ) to replace any space found with an empty string. With that done, FIND_IN_SET( ) will return the ranking each student gave for *reading*. The ORDER BY clause orders those results by reading_rank—the alias given for the second field with the AS clause.

## HEX( )

HEX(*string*)

The first version of this function accepts a string and returns its numerical value, in hexadecimal, as it is represented in the underlying character set. The second version

accepts a decimal integer and returns the hexadecimal equivalent. The function returns NULL if given a NULL value.

For an example, suppose that a college has conducted a student survey through an application that has somehow saved a number of formatting characters as strings containing their hexadecimal equivalents. For instance, a tab appears as 09, and we want to replace each instance with an actual tab. Although we could do this with a straight replacement function, we'd like to use a slightly more abstract solution that can be used with many different characters that suffer from this problem in a particular column.



One solution, changing all instances in the column student_surveys, is as follows:

```
UPDATE student_surveys
SET opinion = REPLACE(opinion, HEX('\t'), UNHEX(HEX('\t')))
WHERE survey_id = 127;
```

In this SQL statement, HEX( ) is used to return the hexadecimal value of tab, represented by \t. That value is given to REPLACE( ) as the string for which it is to replace. Then, using HEX( ) again but wrapped in UNHEX( ) to return the binary character for tab, we're providing REPLACE( ) with the replacement value.

## INSERT( )

INSERT(*string, position, length, new_string*)

This function inserts the string from the final argument into the string specified by the first argument, at the specified position. If *length* is greater than 0, the function overwrites that number of characters, so the new string replaces part of the original. The function returns NULL if any of the arguments are NULL. The first position is 1. Don't confuse this function with the SQL INSERT statement. Here is an example of this function:

```
UPDATE courses
SET course_name =
INSERT(course_name, INSTR(course_name, 'Eng.'), 4, 'English')
WHERE course_name LIKE "%Eng.%";
```

In this example, some course names have the word *English* abbreviated as *Eng*. This SQL statement overwrites any such occurrences with the word *English*. It uses the INSTR( ) function to find the starting point of the abbreviation. The number value it returns is used as the position argument for the INSERT( ) function. If it's not found, the course name will not be changed because a value of 0 will be returned by INSTR( ), and the INSERT( ) function ignores any request in which *position* lies outside the length of the original string.

## INSTR( )

INSTR(*string, substring*)

This function returns the starting position of the first occurrence of the substring in the string given as the first argument. The index of the first position is 1. This function is case-insensitive unless one of the arguments given is a binary string. For an example of this function, see the description of INSERT( ) previously in this chapter. INSTR( ) is similar to one of the syntaxes of LOCATE( ), but the parameters are given in a different order.

## INTERVAL( )

INTERVAL(*search_value*, *ordered_value*, ...)

This function returns the position in which *search_value* would be located in a comma-separated list of *ordered_value* arguments. In other words, the function returns the first *ordered_value* that is less than or equal to *search_value*. All arguments are treated as integers, and the caller must list the *ordered_value* arguments in increasing order. If *search_value* would be located before the first ordered value, 0 is returned. If *search_value* would be located after the last ordered value, the position of that value is returned.

For example, suppose that a professor at our fictitious college has given the same few exams every semester for the last four semesters. Suppose that he has a table containing a row for each semester, and a column for each exam that contains the average of student grades for the semester. Now the professor wants to know how the average score for the same exam for the current semester compares against the previous semesters: he wants to know how the students on average rank by comparison. We could find this answer by running the following SQL statement:

```
SELECT INTERVAL(
   (SELECT AVG(exam1) FROM student_exams),
   S1,S2,S3,S4) AS Ranking
FROM
  (SELECT
    (SELECT exam1_avg FROM student_exams_past
       ORDER BY exam1_avg LIMIT 0,1) AS S1,
    (SELECT exam1_avg FROM student_exams_past
       ORDER BY exam1_avg LIMIT 1,1) AS S2,
    (SELECT exam1_avg FROM student_exams_past
       ORDER BY exam1_avg LIMIT 2,1) AS S3,
    (SELECT exam1_avg FROM student_exams_past
       ORDER BY exam1_avg LIMIT 3,1) AS S4) AS exam1_stats;
```

In this complex example, we're running four subqueries to get the average exam score stored (*S1*, *S2*, *S3*, and *S4*) in the same column for the four semesters for which we have data. Then we're putting each of these values into one row of a derived table (`exam1_stats`). We will then select each column of that limited derived table for the strings to compare against in the `INTERVAL( )` function. For the first parameter of that function, though, we're running yet another subquery to determine the average grades of students for the same exam for the current semester. The results will be a number from 0 to 4, depending on how this semester's average compares.

## LCASE( )

LCASE(*string*)

This function converts a string given to all lowercase letters. It's an alias of `LOWER( )`. Here is an example:

```
SELECT teacher_id,
CONCAT(LEFT(UCASE(name_last), 1),
SUBSTRING(LCASE(name_last), 2))
AS Teacher
FROM teachers;
```

In this example, we're using a combination of LEFT( ) paired with UCASE( ) and SUBSTRING( ) paired with LCASE( ) to ensure that the first letter of the teacher's name is displayed in uppercase and the rest of the name is in lowercase letters.

## LEFT( )

LEFT(*string, length*)

This function returns the first *length* characters from a string. If you want to extract the end of the string instead of the beginning, use the RIGHT( ) function. Both are multibyte-safe. Here is an example:

```
SELECT LEFT(phone_home, 3) AS 'Area Code',
COUNT(*)
FROM students
GROUP BY LEFT(phone_home, 3);
```

Using the LEFT( ) function, this statement extracts the first three digits of phone_home for each row, which is the telephone area code (i.e., city code). It then groups the results, using the same function in the WHERE clause. This returns a count of the number of students living in each telephone area code.

## LENGTH( )

LENGTH(*string*)

This function returns the number of bytes contained in a given string. It is not aware of multibyte characters, so it assumes there are eight bits to a byte and one byte to a character. OCTET_LENGTH( ) is an alias. If you want to get the length of characters regardless of whether a character is multibyte or not, use CHARACTER_LENGTH( ).

As an example, suppose that we notice in an online survey that some odd binary characters have been entered into the data through the web interface—probably from a spam program. To narrow the list of rows, we can enter the following statement to find the rows that have binary characters in three columns that have the bad data:

```
SELECT respondent_id
FROM survey
WHERE CHARACTER_LENGTH(answer1) != LENGTH(answer1)
OR CHARACTER_LENGTH(answer2) != LENGTH(answer2)
OR CHARACTER_LENGTH(answer3) != LENGTH(answer3)
survey_id = 127;
```

In this example, the WHERE clause invokes CHARACTER_LENGTH( ) to get the number of bytes, and LENGTH( ) to get the number of characters for each column containing a respondent's answers to the survey questions. We then compare them with the != operator to return only rows in which the two methods of evaluation are not equal. The LENGTH( ) will return a greater value for multibyte characters, whereas CHARACTER_LENGTH( ) will return 1 for each character, regardless of whether it's a multibyte character.

## LOAD_FILE( )

LOAD_FILE(*filename*)

This function reads the contents of a file and returns it as a string that may be used in MySQL statements and functions. The user must have FILE privileges in MySQL, and the file must be readable by all users on the filesystem. It returns NULL if the file doesn't exist, if the user doesn't have proper permissions, or if the file is otherwise unreadable. The file size in bytes must be less than the amount specified in the system variable max_allowed_packet. Starting with version 5.1.6 of MySQL, the system variable character_set_filesystem is used to provide filenames in the character set recognized by the underlying filesystem. Here is an example:

```
UPDATE applications
SET essay = LOAD_FILE('/tmp/applicant_7382.txt'),
student_photo = LOAD_FILE('/tmp/applicant_7382.jpeg')
WHERE applicant_id = '7382';
```

In this example, an essay written by someone who is applying for admission to the college is loaded into the *essay* column (which is a TEXT data type) of the row for the applicant in the applications table. The entire contents of the file, including any binary data (e.g., hard returns and tabs), are loaded from the file into the table. Additionally, an image file containing the student's photograph is loaded into another column of the same table, but in a BLOB column.

## LOCATE( )

LOCATE(*substring, string[, start_position]*)

This function returns the numeric starting point of the first occurrence of a substring in the string supplied as a second argument. A starting position for searching may be specified as a third argument. It's not case-sensitive unless one of the strings given is a binary string. The function is multibyte-safe.

As an example of this function's potential, suppose that a table for a college contains a list of courses and one of the columns (course_desc) contains the description of the courses. A typical column starts like this:

```
Victorian Literature [19th Cent. Engl. Lit.]: This course covers Engl.
   novels and Engl. short-stories...
```

We want to replace all occurrences of the abbreviation Engl. with English except in the beginning of the strings where the abbreviation is contained in square brackets, as shown here. To do this, we could enter an SQL statement like this:

```
UPDATE courses
SET course_desc =
INSERT(course_desc, LOCATE('Engl.', course_desc, LOCATE(']', course_desc)),
   5, 'English')
WHERE course_desc LIKE '%Engl.%';
```

In this statement, we use the LOCATE( ) function to locate the first occurrence of the closing square bracket. From there, we use LOCATE( ) again to find the first occurrence of Engl.. With the INSERT( ) function (not the INSERT statement), we remove the five characters starting from that point located after the closing square bracket and inserting

the text English. This is a bit complex, but it generally works. However, it replaces only one occurrence of the text we're trying to replace, whereas in the sample text shown there are at least two occurrences of Engl. after the brackets. We could keep running that SQL statement until we replace each one. A better method would be to run this SQL statement instead:

```
UPDATE courses
SET course_desc =
CONCAT(
   SUBSTRING_INDEX(course_desc, ']', 1),
   REPLACE( SUBSTR(course_desc, LOCATE(']', course_desc)),
   'Engl.', 'English')
)
WHERE course_desc LIKE '%Engl.%';
```

In this statement, we use SUBSTRING_INDEX( ) to extract the opening text until the first closing bracket. We then use LOCATE( ) to locate the closing bracket, SUBSTR( ) to extract the text from that point forward, and then REPLACE( ) to replace all occurrences of Engl. in that substring. Finally, CONCAT( ) pastes the opening text that we preserved and excluded from the replacement component together with the cleaned text.

## LOWER( )

LOWER(*string*)

This function converts a given string to all lowercase letters. It is an alias of LCASE( ):

```
SELECT course_id AS 'Course ID',
LOWER(course_name) AS Course
FROM courses;
```

This statement displays the name of each course in all lowercase letters.

## LPAD( )

LPAD(*string, length, padding*)

This function adds *padding* to the left end of *string*, stopping if the combination of *string* and the added padding reach *length* characters. If *length* is shorter than the length of the string, the string will be shortened starting from the left to comply with the length constraint. The padding can be any character. Here is an example:

```
SELECT LPAD(course_name, 25, '.') AS Courses
FROM courses
WHERE course_code LIKE 'ENGL%'
LIMIT 3;

+--------------------------+
| Courses                  |
+--------------------------+
| .........Creative Writing |
| .....Professional Writing |
| ......American Literature |
+--------------------------+
```

In this example, a list of three courses is retrieved and the results are padded with dots to the left of the course names.

## LTRIM( )

LTRIM(*string*)

This function returns the given string with any leading spaces removed. When used with an SQL statement such as UPDATE, rows that do not contain leading spaces will not be changed. This function is multibyte-safe. To trim trailing spaces, use RTRIM( ). To trim both leading and trailing spaces, use TRIM( ). Here is an example:

```
UPDATE students
SET name_last = LTRIM(name_last);
```

In this example, the last names of several students have been entered inadvertently with a space in front of the names. This SQL statement removes any leading spaces from each name retrieved that contains leading spaces and then writes the trimmed text over the existing data.

## MAKE_SET( )

MAKE_SET(*bits, string1, string2, ...*)

This function converts the decimal number in *bits* to binary and returns a comma-separated list of values for all the bits that are set in that number, using *string1* for the low-order bit, *string2* for the next lowest bit, etc. Here is an example:

```
SELECT BIN(9) AS 'Binary 9',
MAKE_SET(100, 'A','B','C','D')
AS Set;

+----------+------+
| Binary 9 | Set  |
+----------+------+
| 1001     | A,D  |
+----------+------+
```

The binary equivalent of 9 is 1001. The first bit starting from the right of the binary number shown is 1 (or *on*), so the first string in the list is put into the results. The second and third bits of the binary number are 0, so the second and third strings ('B' and 'C') are left out of the results. The fourth bit counting from the right is 1, so the fourth string of the list is added to the results.

## MATCH( ) AGAINST( )

MATCH(*column[, ...]*) AGAINST (*string*)

This function is used only for columns indexed by a FULLTEXT index, and only in WHERE clauses. In these clauses, it can be a condition used to search columns for a given string. Text in the string containing spaces is parsed into separate words, so a column matches if it contains at least one word. Small words (three characters or less) are ignored. Here is an example:

```
SELECT applicant_id
FROM applications
WHERE MATCH (essay) AGAINST ('English');
```

This SQL statement searches the table containing data on people applying for admission to the college. The essay column contains a copy of the applicant's admission essay. The column is searched for applicants who mention the word English, so that a list of applicants who have voiced an interest in the English program will be displayed.



## MID( )

MID(*string*, *position*[, *length*])

This function returns the characters of a given string, starting from the position specified in the second argument. The first character is numbered 1. You can limit the length of the string retrieved by specifying a limit in the third argument. This function is similar to SUBSTRING( ).

As an example of this function, suppose that a table of information about teachers contains a column listing their home telephone numbers. This column's entries are in a format showing only numbers, no hyphens or other separators (e.g., 50412345678). Suppose further that we decide to add the country code and hyphens in a typical U.S. format (e.g., +1-504-123-45678) because although all our teachers live in the U.S., we're about to acquire a small school in a different country. We could make these changes like so:

```
UPDATE teachers
SET phone_home =
CONCAT_WS('-', '+1',
   LEFT(phone_home, 3),
   MID(phone_home, 4, 3),
   MID(phone_home, 7) );
```

This convoluted SQL statement extracts each component of the telephone number with the LEFT( ) and MID( ) functions. Using CONCAT_WS( ), the data is merged back together along with the country code at the beginning. Components in the return value are separated with a hyphen, which is given as its first parameter.

## OCTET_LENGTH( )

OCTET_LENGTH(*string*)

This function returns the number of bytes contained in the given string. It does not recognize multibyte characters, so it assumes there are eight bits to a byte and one byte to a character. An *octet* is synonymous with *byte* in most contexts nowadays, so this function is an alias of LENGTH( ). See the description of that function earlier in this chapter for examples of its use.

## ORD( )

ORD(*string*)

This function returns an ordinal value, the position of a character in the ASCII character set of the leftmost character in a given string. For multibyte characters, it follows a formula to determine the results: byte1 + (byte2 * 256) + (byte3 *$256^2$)....

POSITION( )

Here is an example:

```
SELECT ORD('A'), ORD('a');

+----------+----------+
| ORD('A') | ORD('a') |
+----------+----------+
|       65 |       97 |
+----------+----------+
```

## POSITION( )

POSITION(*substring* IN *string*)

This function returns an index of the character in *string* where *substring* first appears. The first character of *string* is numbered 1. This function is like LOCATE( ), except that the keyword IN is used instead of a comma to separate the substring and the containing string. Also, this function does not provide a starting point to begin the search; it must begin from the leftmost character. Here is an example:

```
UPDATE courses
SET course_name =
INSERT(course_name, POSITION('Eng.' IN course_name), 4, 'English')
WHERE course_name LIKE "%Eng.%";
```

In this example, some course names have the word English abbreviated as Eng. This SQL statement overwrites any such occurrences with the word English. It uses the POSITION( ) function to find the starting point of the abbreviation. The numerical value it returns is then used as the position argument for the INSERT( ) function (not the INSERT statement). If it's not found, the course name will not be changed, because a value of 0 will be returned by POSITION( ), and the INSERT( ) function ignores any request in which *position* lies outside the length of the original string.

## QUOTE( )

QUOTE(*string*)

This function accepts a string enclosed in single quotes and returns a string that is safe to manipulate with SQL statements. Single quotes, backslashes, ASCII NULLs, and Ctrl-Zs contained in the string are escaped with a backslash. This is a useful security measure when accepting values from a public web interface. Here is an example:

```
SELECT QUOTE(course_name) AS Courses
FROM courses
WHERE course_code = 'ENGL-405';

+---------------------+
| Courses             |
+---------------------+
| 'Works of O\'Henry' |
+---------------------+
```

Notice in the results that because of the QUOTE( ) function, the string returned is enclosed in single quotes, and the single quote within the data returned is escaped with a backslash.

## REPEAT( )

REPEAT(*string, count*)

This function returns the string given in the first argument of the function as many times as specified in the second argument. It returns an empty string if *count* is less than 1. It returns NULL if either argument is NULL. Here is an example:

```
SELECT REPEAT('Urgent! ', 3)
AS 'Warning Message';
```



## REPLACE( )

REPLACE(*string, old_element, new_element*)

This function goes through the first argument and returns a string in which every occurrence of the second argument is replaced with the third argument. Here is an example:

```
UPDATE students,
REPLACE(title, 'Mrs.', 'Ms.');
```

This SQL statement will retrieve each student's title and replace any occurrences of "Mrs." with "Ms." UPDATE will change only the rows where the replacement was made.

## REVERSE( )

REVERSE(*string*)

This function returns the characters of *string* in reverse order. It's multibyte-safe. Here is an example:

```
SELECT REVERSE('MUD');

+----------------+
| REVERSE('MUD') |
+----------------+
| DUM            |
+----------------+
```

## RIGHT( )

RIGHT(*string, length*)

This function returns the final *length* characters from a string. If you want to extract the beginning of the string instead of the end, use the LEFT( ) function. Both are multibyte-safe. Here is an example:

```
SELECT RIGHT(soc_sec, 4)
FROM students
WHERE student_id = '43325146122';
```

This statement retrieves the last four digits of the student's Social Security number as an identity verification.

## RPAD( )

RPAD(*string*, *length*, *padding*)

This function adds *padding* to the right end of *string*, stopping if the combination of *string* and the added padding reach *length* characters. If the length given is shorter than the length of the string, the string will be shortened to comply with the length constraint. The padding can be any character. Here is an example:

```
SELECT RPAD(course_name, 25, '.') AS Courses
FROM courses
WHERE course_code LIKE 'ENGL%'
LIMIT 3;

+---------------------------+
| Courses                   |
+---------------------------+
| Creative Writing......... |
| Professional Writing..... |
| American Literature...... |
+---------------------------+
```

This statement presents a list of three course names that are retrieved. Each row of the results is padded with dots to the right.

## RTRIM( )

RTRIM(*string*)

This function returns the given string with any trailing spaces removed. When used with an SQL statement such as UPDATE, rows that do not contain trailing spaces will not be changed. This function is multibyte-safe. To trim leading spaces, use LTRIM( ). To trim both leading and trailing spaces, use TRIM( ). Here is an example:

```
UPDATE students
SET name_last = RTRIM(name_last);
```

In this example, the last names of several students have been entered inadvertently with a space at the end of the names. This SQL statement removes any trailing spaces from each name retrieved that contains trailing spaces and then writes the trimmed text over the existing data.

## SOUNDEX( )

SOUNDEX(*string*)

This function returns the results of a classic algorithm that can be used to compare two similar strings. Here is an example:

```
SELECT IF(SOUNDEX('him') = SOUNDEX('hymm'),
'Sounds Alike', 'Does not sound alike')
AS 'Sound Comparison';

+------------------+
| Sound Comparison |
+------------------+
```

```
| Sounds Alike     |
+------------------+
```

SOUNDEX( ) was designed to allow comparisons between fuzzy inputs, but it's rarely used.

## SPACE( )

SPACE(*count*)

This function returns a string of spaces. The number of spaces returned is set by the argument. Here is an example:

```
SELECT CONCAT(name_first, SPACE(1), name_last)
AS Name
FROM students LIMIT 1;

+------------------+
| Name             |
+------------------+
| Richard Stringer |
+------------------+
```

Although this example requires a lot more typing than just placing a space within quotes, it's more apparent when glancing at it that a space is to be inserted. For multiple or variable spaces, you could substitute the count with another function to determine the number of spaces needed based on data from a table, the length of other inputs, or some other factor.

## STRCMP( )

STRCMP(*string, string*)

This function compares two strings to determine whether the first string is before or after the second string in ASCII sequence. If the first string precedes the second string, –1 is returned. If the first follows the second, 1 is returned. If they are equal, 0 is returned. This function is often used for alphanumeric comparisons, but it is case-insensitive unless at least one of the strings given is binary. Here is an example:

```
SELECT * FROM
(SELECT STRCMP(
   SUBSTR(pre_req, 1, 8),
   SUBSTR(pre_req, 10, 8))
AS Comparison
FROM courses) AS derived1
WHERE Comparison = 1;
```

In this example, because course codes are all eight characters long, we use SUBSTR( ) to extract the first two course code numbers. Using STRCMP( ), we compare the two course codes to see if they're in sequence. To see only the results where the courses are out of sequence, we use a subquery with a WHERE clause to return only rows for which the STRCMP( ) returns a –1 value, indicating the two strings are not in sequence.

The problem with this statement is that some courses have more than two prerequisites. We would have to expand this statement to encompass them. However, that doesn't resolve the problem either; it provides only more indications of what we know. To reorder

the data, it would be easier to create a simple script using one of the APIs to extract, reorder, and then replace the column values.

## SUBSTR( )

```
SUBSTRING(string, position[, length])
SUBSTRING(string FROM position FOR length)
```

This function is an alias of SUBSTRING( ). See its description next for details and an example of its use.

## SUBSTRING( )

```
SUBSTRING(string, position[, length])
SUBSTRING(string FROM position[ FOR length])
```

This function returns the characters of a given string, starting from the position given. The first character is numbered 1. You can restrict the length of the string retrieved by specifying a limit. The function is similar to MID( ). Here is an example:

```
SELECT CONCAT_WS('-',
   SUBSTRING(soc_sec, 1, 3),
   SUBSTRING(soc_sec FROM 4 FOR 2),
   SUBSTRING(soc_sec FROM 6)
)
AS 'Social Security Nbr.'
FROM students LIMIT 1;

+----------------------+
| Social Security Nbr. |
+----------------------+
| 433-12-3456          |
+----------------------+
```

This example shows the two syntaxes of SUBSTRING( ) for reformatting a Social Security number (the U.S. federal tax identification number) stored without dashes. It uses CONCAT_WS( ) to put the three pieces of data together, separated by the hyphen given.

## SUBSTRING_INDEX( )

```
SUBSTRING_INDEX(string, delimiter, count)
```

This function returns a substring of *string*, using *delimiter* to separate substrings and *count* to determine which of the substrings to return. Thus, a *count* of 1 returns the first substring, 2 returns the second, and so on. A negative number instructs the function to count from the right end. Here is an example:

```
SELECT SUBSTRING_INDEX(pre_req, '|', -1)
AS 'Last Prerequisite',
pre_req AS 'All Prerequisites'
FROM courses WHERE course_id = '1245';

+-------------------+--------------------------+
| Last Prerequisite | All Prerequisites        |
+-------------------+--------------------------+
```

```
| ENGL-202          | ENGL-101|ENGL-201|ENGL-202 |
+-------------------+--------------------------+
```

In this example, the pre_req column for each course contains prerequisite courses separated by vertical bars. The statement displays the last prerequisite, because –1 was entered for the count.



## TRIM( )

`TRIM([[BOTH|LEADING|TRAILING] [padding] FROM] string)`

This function returns the given string with any trailing or leading padding removed, depending on which is specified. If neither is specified, BOTH is the default, causing both leading and trailing padding to be removed. The default padding is a space if none is specified. The function is multibyte-safe.

As an example, in a table containing the results of a student survey we notice that one of the columns that lists each student's favorite activities contains extra commas at the end of the comma-separated list of activities. This may have been caused by a problem in the web interface, which treated any activities that a student didn't select as blank values separated by commas at the end (e.g., biking,reading,,,,):

```
UPDATE student_surveys
SET favorite_activities =
TRIM(LEADING SPACE(1) FROM TRIM(TRAILING ',' FROM favorite_activities));
```

In this example, we're using TRIM( ) twice: once to remove the trailing commas from the column favorite_activities and then again on those results to remove leading spaces. Since the functions are part of an UPDATE statement, the double-trimmed results are saved back to the table for the row for which the data was read. This is more verbose than it needs to be, though. Because a space is the default padding, we don't have to specify it. Also, because we want to remove both leading and trailing spaces and commas from the data, we don't have to specify LEADING or TRAILING and can allow the default of BOTH to be used. Making these adjustments, we get this tighter SQL statement:

```
UPDATE student_surveys
SET favorite_activities =
TRIM(TRIM(',' FROM favorite_activities));
```

If we suspected that the faulty web form also added extra commas between the text (not just at the end), we could wrap these concentric uses of TRIM( ) within REPLACE( ) to replace any occurrences of consecutive commas with a single comma:

```
UPDATE student_surveys
SET favorite_activities =
REPLACE(TRIM(TRIM(',' FROM favorite_activities)), ',,', ',');
```

## UCASE( )

`UCASE(string)`

This function converts a given string to all uppercase letters. It's an alias of UPPER( ). Here is an example:

```
SELECT course_id AS 'Course ID',
UCASE(course_name) AS Course
FROM courses LIMIT 3;
```