

If *expr1* is not `NULL`, `IFNULL()` returns *expr1*; otherwise it returns *expr2*. `IFNULL()` returns a numeric or string value, depending on the context in which it is used.

```
mysql> SELECT IFNULL(1,0);
-> 1
mysql> SELECT IFNULL(NULL,10);
-> 10
mysql> SELECT IFNULL(1/0,10);
-> 10
mysql> SELECT IFNULL(1/0,'yes');
-> 'yes'
```

The default result value of `IFNULL(expr1,expr2)` is the more “general” of the two expressions, in the order `STRING`, `REAL`, or `INTEGER`. Consider the case of a table based on expressions or where MySQL must internally store a value returned by `IFNULL()` in a temporary table:

```
mysql> CREATE TABLE tmp SELECT IFNULL(1,'test') AS test;
mysql> DESCRIBE tmp;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| test  | varbinary(4)  | NO   |     |         |       |
+-----+-----+-----+-----+-----+-----+
```

In this example, the type of the `test` column is `VARBINARY(4)`.

- `NULLIF(expr1,expr2)`

Returns `NULL` if *expr1* = *expr2* is true, otherwise returns *expr1*. This is the same as `CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END`.

```
mysql> SELECT NULLIF(1,1);
-> NULL
mysql> SELECT NULLIF(1,2);
-> 1
```

Note that MySQL evaluates *expr1* twice if the arguments are not equal.

12.5 String Functions

Table 12.7 String Operators

Name	Description
<code>ASCII()</code>	Return numeric value of left-most character
<code>BIN()</code>	Return a string containing binary representation of a number
<code>BIT_LENGTH()</code>	Return length of argument in bits
<code>CHAR()</code>	Return the character for each integer passed
<code>CHAR_LENGTH()</code>	Return number of characters in argument
<code>CHARACTER_LENGTH()</code>	Synonym for <code>CHAR_LENGTH()</code>
<code>CONCAT()</code>	Return concatenated string
<code>CONCAT_WS()</code>	Return concatenate with separator
<code>ELT()</code>	Return string at index number
<code>EXPORT_SET()</code>	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string

Name	Description
<code>FIELD()</code>	Return the index (position) of the first argument in the subsequent arguments
<code>FIND_IN_SET()</code>	Return the index position of the first argument within the second argument
<code>FORMAT()</code>	Return a number formatted to specified number of decimal places
<code>HEX()</code>	Return a hexadecimal representation of a decimal or string value
<code>INSERT()</code>	Insert a substring at the specified position up to the specified number of characters
<code>INSTR()</code>	Return the index of the first occurrence of substring
<code>LCASE()</code>	Synonym for LOWER()
<code>LEFT()</code>	Return the leftmost number of characters as specified
<code>LENGTH()</code>	Return the length of a string in bytes
<code>LIKE</code>	Simple pattern matching
<code>LOAD_FILE()</code>	Load the named file
<code>LOCATE()</code>	Return the position of the first occurrence of substring
<code>LOWER()</code>	Return the argument in lowercase
<code>LPAD()</code>	Return the string argument, left-padded with the specified string
<code>LTRIM()</code>	Remove leading spaces
<code>MAKE_SET()</code>	Return a set of comma-separated strings that have the corresponding bit in bits set
<code>MATCH</code>	Perform full-text search
<code>MID()</code>	Return a substring starting from the specified position
<code>NOT LIKE</code>	Negation of simple pattern matching
<code>NOT REGEXP</code>	Negation of REGEXP
<code>OCT()</code>	Return a string containing octal representation of a number
<code>OCTET_LENGTH()</code>	Synonym for LENGTH()
<code>ORD()</code>	Return character code for leftmost character of the argument
<code>POSITION()</code>	Synonym for LOCATE()
<code>QUOTE()</code>	Escape the argument for use in an SQL statement
<code>REGEXP</code>	Pattern matching using regular expressions
<code>REPEAT()</code>	Repeat a string the specified number of times
<code>REPLACE()</code>	Replace occurrences of a specified string
<code>REVERSE()</code>	Reverse the characters in a string
<code>RIGHT()</code>	Return the specified rightmost number of characters
<code>RLIKE</code>	Synonym for REGEXP
<code>RPAD()</code>	Append string the specified number of times
<code>RTRIM()</code>	Remove trailing spaces
<code>SOUNDEX()</code>	Return a soundex string
<code>SOUNDS LIKE</code>	Compare sounds
<code>SPACE()</code>	Return a string of the specified number of spaces

Name	Description
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TRIM()	Remove leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase

String-valued functions return [NULL](#) if the length of the result would be greater than the value of the `max_allowed_packet` system variable. See [Section 5.1.1, “Configuring the Server”](#).

For functions that operate on string positions, the first position is numbered 1.

For functions that take length arguments, noninteger arguments are rounded to the nearest integer.

- [ASCII\(*str*\)](#)

Returns the numeric value of the leftmost character of the string *str*. Returns 0 if *str* is the empty string. Returns [NULL](#) if *str* is [NULL](#). [ASCII\(\)](#) works for 8-bit characters.

```
mysql> SELECT ASCII('2');
-> 50
mysql> SELECT ASCII(2);
-> 50
mysql> SELECT ASCII('dx');
-> 100
```

See also the [ORD\(\)](#) function.

- [BIN\(*N*\)](#)

Returns a string representation of the binary value of *N*, where *N* is a longlong ([BIGINT](#)) number. This is equivalent to [CONV\(N,10,2\)](#). Returns [NULL](#) if *N* is [NULL](#).

```
mysql> SELECT BIN(12);
-> '1100'
```

- [BIT_LENGTH\(*str*\)](#)

Returns the length of the string *str* in bits.

```
mysql> SELECT BIT_LENGTH('text');
-> 32
```

- [CHAR\(*N*,... \[USING *charset_name*\]\)](#)

[CHAR\(\)](#) interprets each argument *N* as an integer and returns a string consisting of the characters given by the code values of those integers. [NULL](#) values are skipped.

```
mysql> SELECT CHAR(77,121,83,81,'76');
-> 'MySQL'
mysql> SELECT CHAR(77,77.3,'77.3');
-> 'MMM'
```

`CHAR()` arguments larger than 255 are converted into multiple result bytes. For example, `CHAR(256)` is equivalent to `CHAR(1,0)`, and `CHAR(256*256)` is equivalent to `CHAR(1,0,0)`:

```
mysql> SELECT HEX(CHAR(1,0)), HEX(CHAR(256));
+-----+-----+
| HEX(CHAR(1,0)) | HEX(CHAR(256)) |
+-----+-----+
| 0100          | 0100          |
+-----+-----+
mysql> SELECT HEX(CHAR(1,0,0)), HEX(CHAR(256*256));
+-----+-----+
| HEX(CHAR(1,0,0)) | HEX(CHAR(256*256)) |
+-----+-----+
| 010000          | 010000          |
+-----+-----+
```

By default, `CHAR()` returns a binary string. To produce a string in a given character set, use the optional `USING` clause:

```
mysql> SELECT CHARSET(CHAR(X'65')), CHARSET(CHAR(X'65' USING utf8));
+-----+-----+
| CHARSET(CHAR(X'65')) | CHARSET(CHAR(X'65' USING utf8)) |
+-----+-----+
| binary              | utf8                          |
+-----+-----+
```

If `USING` is given and the result string is illegal for the given character set, a warning is issued. Also, if strict SQL mode is enabled, the result from `CHAR()` becomes `NULL`.

- `CHAR_LENGTH(str)`

Returns the length of the string `str`, measured in characters. A multibyte character counts as a single character. This means that for a string containing five 2-byte characters, `LENGTH()` returns 10, whereas `CHAR_LENGTH()` returns 5.

- `CHARACTER_LENGTH(str)`

`CHARACTER_LENGTH()` is a synonym for `CHAR_LENGTH()`.

- `CONCAT(str1, str2, ...)`

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent string form. This is a nonbinary string as of MySQL 5.5.3. Before 5.5.3, it is a binary string; to avoid that and produce a nonbinary string, you can use an explicit type cast, as in this example:

```
SELECT CONCAT(CAST(int_col AS CHAR), char_col);
```

`CONCAT()` returns `NULL` if any argument is `NULL`.

```
mysql> SELECT CONCAT('My', 'S', 'QL');
-> 'MySQL'
mysql> SELECT CONCAT('My', NULL, 'QL');
-> NULL
mysql> SELECT CONCAT(14.3);
-> '14.3'
```

For quoted strings, concatenation can be performed by placing the strings next to each other:

```
mysql> SELECT 'My' 'S' 'QL';
```

```
-> 'MySQL'
```

- `CONCAT_WS(separator, str1, str2, ...)`

`CONCAT_WS()` stands for Concatenate With Separator and is a special form of `CONCAT()`. The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is `NULL`, the result is `NULL`.

```
mysql> SELECT CONCAT_WS(',', 'First name', 'Second name', 'Last Name');
-> 'First name,Second name,Last Name'
mysql> SELECT CONCAT_WS(',', 'First name', NULL, 'Last Name');
-> 'First name,Last Name'
```

`CONCAT_WS()` does not skip empty strings. However, it does skip any `NULL` values after the separator argument.

- `ELT(N, str1, str2, str3, ...)`

`ELT()` returns the *N*th element of the list of strings: *str1* if *N* = 1, *str2* if *N* = 2, and so on. Returns `NULL` if *N* is less than 1 or greater than the number of arguments. `ELT()` is the complement of `FIELD()`.

```
mysql> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');
-> 'ej'
mysql> SELECT ELT(4, 'ej', 'Heja', 'hej', 'foo');
-> 'foo'
```

- `EXPORT_SET(bits, on, off[, separator[, number_of_bits]])`

Returns a string such that for every bit set in the value *bits*, you get an *on* string and for every bit not set in the value, you get an *off* string. Bits in *bits* are examined from right to left (from low-order to high-order bits). Strings are added to the result from left to right, separated by the *separator* string (the default being the comma character ,). The number of bits examined is given by *number_of_bits*, which has a default of 64 if not specified. *number_of_bits* is silently clipped to 64 if larger than 64. It is treated as an unsigned integer, so a value of -1 is effectively the same as 64.

```
mysql> SELECT EXPORT_SET(5, 'Y', 'N', ',', 4);
-> 'Y,N,Y,N'
mysql> SELECT EXPORT_SET(6, '1', '0', ',', 10);
-> '0,1,1,0,0,0,0,0,0,0'
```

- `FIELD(str, str1, str2, str3, ...)`

Returns the index (position) of *str* in the *str1, str2, str3, ...* list. Returns 0 if *str* is not found.

If all arguments to `FIELD()` are strings, all arguments are compared as strings. If all arguments are numbers, they are compared as numbers. Otherwise, the arguments are compared as double.

If *str* is `NULL`, the return value is 0 because `NULL` fails equality comparison with any value. `FIELD()` is the complement of `ELT()`.

```
mysql> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo');
-> 2
mysql> SELECT FIELD('fo', 'Hej', 'ej', 'Heja', 'hej', 'foo');
-> 0
```

- `FIND_IN_SET(str, strlist)`

Returns a value in the range of 1 to *N* if the string *str* is in the string list *strlist* consisting of *N* substrings. A string list is a string composed of substrings separated by , characters. If the first argument is a constant string and the second is a column of type SET, the `FIND_IN_SET()` function is optimized to use bit arithmetic. Returns 0 if *str* is not in *strlist* or if *strlist* is the empty string. Returns NULL if either argument is NULL. This function does not work properly if the first argument contains a comma (,) character.

```
mysql> SELECT FIND_IN_SET('b','a,b,c,d');
      -> 2
```

- `FORMAT(X,D[,locale])`

Formats the number *X* to a format like '#,###,###.##', rounded to *D* decimal places, and returns the result as a string. If *D* is 0, the result has no decimal point or fractional part.

The optional third parameter enables a locale to be specified to be used for the result number's decimal point, thousands separator, and grouping between separators. Permissible locale values are the same as the legal values for the `lc_time_names` system variable (see [Section 10.7, "MySQL Server Locale Support"](#)). If no locale is specified, the default is 'en_US'.

```
mysql> SELECT FORMAT(12332.123456, 4);
      -> '12,332.1235'
mysql> SELECT FORMAT(12332.1,4);
      -> '12,332.1000'
mysql> SELECT FORMAT(12332.2,0);
      -> '12,332'
mysql> SELECT FORMAT(12332.2,2,'de_DE');
      -> '12.332,20'
```

- `HEX(str), HEX(N)`

For a string argument *str*, `HEX()` returns a hexadecimal string representation of *str* where each byte of each character in *str* is converted to two hexadecimal digits. (Multibyte characters therefore become more than two digits.) The inverse of this operation is performed by the `UNHEX()` function.

For a numeric argument *N*, `HEX()` returns a hexadecimal string representation of the value of *N* treated as a longlong (BIGINT) number. This is equivalent to `CONV(N,10,16)`. The inverse of this operation is performed by `CONV(HEX(N),16,10)`.

```
mysql> SELECT X'616263', HEX('abc'), UNHEX(HEX('abc'));
      -> 'abc', 616263, 'abc'
mysql> SELECT HEX(255), CONV(HEX(255),16,10);
      -> 'FF', 255
```

- `INSERT(str,pos,len,newstr)`

Returns the string *str*, with the substring beginning at position *pos* and *len* characters long replaced by the string *newstr*. Returns the original string if *pos* is not within the length of the string. Replaces the rest of the string from position *pos* if *len* is not within the length of the rest of the string. Returns NULL if any argument is NULL.

```
mysql> SELECT INSERT('Quadratic', 3, 4, 'What');
      -> 'QuWhattic'
mysql> SELECT INSERT('Quadratic', -1, 4, 'What');
      -> 'Quadratic'
mysql> SELECT INSERT('Quadratic', 3, 100, 'What');
      -> 'QuWhat'
```

This function is multibyte safe.

- `INSTR(str,substr)`

Returns the position of the first occurrence of substring *substr* in string *str*. This is the same as the two-argument form of `LOCATE()`, except that the order of the arguments is reversed.

```
mysql> SELECT INSTR('foobarbar', 'bar');
-> 4
mysql> SELECT INSTR('xbar', 'foobar');
-> 0
```

This function is multibyte safe, and is case sensitive only if at least one argument is a binary string.

- `LCASE(str)`

`LCASE()` is a synonym for `LOWER()`.

- `LEFT(str, len)`

Returns the leftmost *len* characters from the string *str*, or `NULL` if any argument is `NULL`.

```
mysql> SELECT LEFT('foobarbar', 5);
-> 'fooba'
```

This function is multibyte safe.

- `LENGTH(str)`

Returns the length of the string *str*, measured in bytes. A multibyte character counts as multiple bytes. This means that for a string containing five 2-byte characters, `LENGTH()` returns 10, whereas `CHAR_LENGTH()` returns 5.

```
mysql> SELECT LENGTH('text');
-> 4
```



Note

The `Length()` OpenGIS spatial function is named `GLength()` in MySQL.

- `LOAD_FILE(file_name)`

Reads the file and returns the file contents as a string. To use this function, the file must be located on the server host, you must specify the full path name to the file, and you must have the `FILE` privilege. The file must be readable by all and its size less than `max_allowed_packet` bytes. If the `secure_file_priv` system variable is set to a nonempty directory name, the file to be loaded must be located in that directory.

If the file does not exist or cannot be read because one of the preceding conditions is not satisfied, the function returns `NULL`.

The `character_set_filesystem` system variable controls interpretation of file names that are given as literal strings.

```
mysql> UPDATE t
      SET blob_col=LOAD_FILE('/tmp/picture')
      WHERE id=1;
```

- `LOCATE(substr, str)`, `LOCATE(substr, str, pos)`

The first syntax returns the position of the first occurrence of substring *substr* in string *str*. The second syntax returns the position of the first occurrence of substring *substr* in string *str*, starting at position *pos*. Returns 0 if *substr* is not in *str*. Returns `NULL` if *substr* or *str* is `NULL`.

```
mysql> SELECT LOCATE('bar', 'foobarbar');
-> 4
mysql> SELECT LOCATE('xbar', 'foobar');
-> 0
mysql> SELECT LOCATE('bar', 'foobarbar', 5);
-> 7
```

This function is multibyte safe, and is case-sensitive only if at least one argument is a binary string.

- `LOWER(str)`

Returns the string `str` with all characters changed to lowercase according to the current character set mapping. The default is `latin1` (cp1252 West European).

```
mysql> SELECT LOWER('QUADRATICALLY');
-> 'quadratically'
```

`LOWER()` (and `UPPER()`) are ineffective when applied to binary strings (`BINARY`, `VARBINARY`, `BLOB`). To perform lettercase conversion, convert the string to a nonbinary string:

```
mysql> SET @str = BINARY 'New York';
mysql> SELECT LOWER(@str), LOWER(CONVERT(@str USING latin1));
+-----+-----+
| LOWER(@str) | LOWER(CONVERT(@str USING latin1)) |
+-----+-----+
| New York   | new york                         |
+-----+-----+
```

This function is multibyte safe.

- `LPAD(str, len, padstr)`

Returns the string `str`, left-padded with the string `padstr` to a length of `len` characters. If `str` is longer than `len`, the return value is shortened to `len` characters.

```
mysql> SELECT LPAD('hi', 4, '??');
-> '??hi'
mysql> SELECT LPAD('hi', 1, '??');
-> 'h'
```

- `LTRIM(str)`

Returns the string `str` with leading space characters removed.

```
mysql> SELECT LTRIM('  barbar');
-> 'barbar'
```

This function is multibyte safe.

- `MAKE_SET(bits, str1, str2, ...)`

Returns a set value (a string containing substrings separated by `,` characters) consisting of the strings that have the corresponding bit in `bits` set. `str1` corresponds to bit 0, `str2` to bit 1, and so on. `NULL` values in `str1`, `str2`, ... are not appended to the result.

```
mysql> SELECT MAKE_SET(1, 'a', 'b', 'c');
-> 'a'
mysql> SELECT MAKE_SET(1 | 4, 'hello', 'nice', 'world');
-> 'hello,world'
mysql> SELECT MAKE_SET(1 | 4, 'hello', 'nice', NULL, 'world');
-> 'hello'
```



```
mysql> SELECT MAKE_SET(0,'a','b','c');  
-> ''
```

- `MID(str,pos,len)`

`MID(str,pos,len)` is a synonym for `SUBSTRING(str,pos,len)`.

- `OCT(N)`

Returns a string representation of the octal value of *N*, where *N* is a longlong (`BIGINT`) number. This is equivalent to `CONV(N,10,8)`. Returns `NULL` if *N* is `NULL`.

```
mysql> SELECT OCT(12);  
-> '14'
```

- `OCTET_LENGTH(str)`

`OCTET_LENGTH()` is a synonym for `LENGTH()`.

- `ORD(str)`

If the leftmost character of the string *str* is a multibyte character, returns the code for that character, calculated from the numeric values of its constituent bytes using this formula:

```
(1st byte code)  
+ (2nd byte code * 256)  
+ (3rd byte code * 2562) ...
```

If the leftmost character is not a multibyte character, `ORD()` returns the same value as the `ASCII()` function.

```
mysql> SELECT ORD('2');  
-> 50
```

- `POSITION(substr IN str)`

`POSITION(substr IN str)` is a synonym for `LOCATE(substr,str)`.

- `QUOTE(str)`

Quotes a string to produce a result that can be used as a properly escaped data value in an SQL statement. The string is returned enclosed by single quotation marks and with each instance of backslash (`\`), single quote (`'`), ASCII `NUL`, and Control+Z preceded by a backslash. If the argument is `NULL`, the return value is the word “NULL” without enclosing single quotation marks.

```
mysql> SELECT QUOTE('Don\'t!');  
-> 'Don\'t!'  
mysql> SELECT QUOTE(NULL);  
-> NULL
```

For comparison, see the quoting rules for literal strings and within the C API in [Section 9.1.1, “String Literals”](#), and [Section 23.8.7.53, “mysql_real_escape_string\(\)”](#).

- `REPEAT(str,count)`

Returns a string consisting of the string *str* repeated *count* times. If *count* is less than 1, returns an empty string. Returns `NULL` if *str* or *count* are `NULL`.

```
mysql> SELECT REPEAT('MySQL', 3);  
-> 'MySQLMySQLMySQL'
```

- `REPLACE(str, from_str, to_str)`

Returns the string `str` with all occurrences of the string `from_str` replaced by the string `to_str`. `REPLACE()` performs a case-sensitive match when searching for `from_str`.

```
mysql> SELECT REPLACE('www.mysql.com', 'w', 'Ww');
      -> 'WwWwWw.mysql.com'
```

This function is multibyte safe.

- `REVERSE(str)`

Returns the string `str` with the order of the characters reversed.

```
mysql> SELECT REVERSE('abc');
      -> 'cba'
```

This function is multibyte safe.

- `RIGHT(str, len)`

Returns the rightmost `len` characters from the string `str`, or `NULL` if any argument is `NULL`.

```
mysql> SELECT RIGHT('foobarbar', 4);
      -> 'rbar'
```

This function is multibyte safe.

- `RPAD(str, len, padstr)`

Returns the string `str`, right-padded with the string `padstr` to a length of `len` characters. If `str` is longer than `len`, the return value is shortened to `len` characters.

```
mysql> SELECT RPAD('hi', 5, '?');
      -> 'hi???'
mysql> SELECT RPAD('hi', 1, '?');
      -> 'h'
```

This function is multibyte safe.

- `RTRIM(str)`

Returns the string `str` with trailing space characters removed.

```
mysql> SELECT RTRIM('barbar ');
      -> 'barbar'
```

This function is multibyte safe.

- `SOUNDEX(str)`

Returns a soundex string from `str`. Two strings that sound almost the same should have identical soundex strings. A standard soundex string is four characters long, but the `SOUNDEX()` function returns an arbitrarily long string. You can use `SUBSTRING()` on the result to get a standard soundex string. All nonalphabetic characters in `str` are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.



Important

When using `SOUNDEX()`, you should be aware of the following limitations:

- This function, as currently implemented, is intended to work well with strings that are in the English language only. Strings in other languages may not produce reliable results.
- This function is not guaranteed to provide consistent results with strings that use multibyte character sets, including `utf-8`.

We hope to remove these limitations in a future release. See Bug #22638 for more information.

```
mysql> SELECT SOUNDEX('Hello');
-> 'H400'
mysql> SELECT SOUNDEX('Quadratically');
-> 'Q36324'
```



Note

This function implements the original Soundex algorithm, not the more popular enhanced version (also described by D. Knuth). The difference is that original version discards vowels first and duplicates second, whereas the enhanced version discards duplicates first and vowels second.

- `expr1 SOUNDS LIKE expr2`

This is the same as `SOUNDEX(expr1) = SOUNDEX(expr2)`.

- `SPACE(N)`

Returns a string consisting of *N* space characters.

```
mysql> SELECT SPACE(6);
-> '      '
```

- `SUBSTR(str,pos)`, `SUBSTR(str FROM pos)`, `SUBSTR(str,pos,len)`, `SUBSTR(str FROM pos FOR len)`

`SUBSTR()` is a synonym for `SUBSTRING()`.

- `SUBSTRING(str,pos)`, `SUBSTRING(str FROM pos)`, `SUBSTRING(str,pos,len)`, `SUBSTRING(str FROM pos FOR len)`

The forms without a *len* argument return a substring from string *str* starting at position *pos*. The forms with a *len* argument return a substring *len* characters long from string *str*, starting at position *pos*. The forms that use `FROM` are standard SQL syntax. It is also possible to use a negative value for *pos*. In this case, the beginning of the substring is *pos* characters from the end of the string, rather than the beginning. A negative value may be used for *pos* in any of the forms of this function.

For all forms of `SUBSTRING()`, the position of the first character in the string from which the substring is to be extracted is reckoned as 1.

```
mysql> SELECT SUBSTRING('Quadratically',5);
-> 'ratically'
mysql> SELECT SUBSTRING('foobarbar' FROM 4);
-> 'barbar'
mysql> SELECT SUBSTRING('Quadratically',5,6);
-> 'ratica'
mysql> SELECT SUBSTRING('Sakila', -3);
-> 'ila'
mysql> SELECT SUBSTRING('Sakila', -5, 3);
-> 'aki'
mysql> SELECT SUBSTRING('Sakila' FROM -4 FOR 2);
-> 'ki'
```

This function is multibyte safe.

If *len* is less than 1, the result is the empty string.

- `SUBSTRING_INDEX(str,delim,count)`

Returns the substring from string *str* before *count* occurrences of the delimiter *delim*. If *count* is positive, everything to the left of the final delimiter (counting from the left) is returned. If *count* is negative, everything to the right of the final delimiter (counting from the right) is returned. `SUBSTRING_INDEX()` performs a case-sensitive match when searching for *delim*.

```
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
-> 'www.mysql'
mysql> SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
-> 'mysql.com'
```

This function is multibyte safe.

- `TRIM([BOTH | LEADING | TRAILING] [remstr] FROM] str), TRIM([remstr FROM] str)`

Returns the string *str* with all *remstr* prefixes or suffixes removed. If none of the specifiers *BOTH*, *LEADING*, or *TRAILING* is given, *BOTH* is assumed. *remstr* is optional and, if not specified, spaces are removed.

```
mysql> SELECT TRIM(' bar ');
-> 'bar'
mysql> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
-> 'barxxx'
mysql> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
-> 'bar'
mysql> SELECT TRIM(TRAILING 'xyz' FROM 'barxyz');
-> 'barx'
```

This function is multibyte safe.

- `UCASE(str)`

`UCASE()` is a synonym for `UPPER()`.

- `UNHEX(str)`

For a string argument *str*, `UNHEX(str)` interprets each pair of characters in the argument as a hexadecimal number and converts it to the byte represented by the number. The return value is a binary string.

```
mysql> SELECT UNHEX('4D7953514C');
-> 'MySQL'
mysql> SELECT X'4D7953514C';
-> 'MySQL'
mysql> SELECT UNHEX(HEX('string'));
-> 'string'
mysql> SELECT HEX(UNHEX('1267'));
-> '1267'
```

The characters in the argument string must be legal hexadecimal digits: '0' .. '9', 'A' .. 'F', 'a' .. 'f'. If the argument contains any nonhexadecimal digits, the result is `NULL`:

```
mysql> SELECT UNHEX('GG');
+-----+
| UNHEX('GG') |
+-----+
```

```
| NULL |
+-----+
```

A `NULL` result can occur if the argument to `UNHEX()` is a `BINARY` column, because values are padded with 0x00 bytes when stored but those bytes are not stripped on retrieval. For example, '41' is stored into a `CHAR(3)` column as '41 ' and retrieved as '41' (with the trailing pad space stripped), so `UNHEX()` for the column value returns 'A'. By contrast '41' is stored into a `BINARY(3)` column as '41\0' and retrieved as '41\0' (with the trailing pad 0x00 byte not stripped). '\0' is not a legal hexadecimal digit, so `UNHEX()` for the column value returns `NULL`.

For a numeric argument `N`, the inverse of `HEX(N)` is not performed by `UNHEX()`. Use `CONV(HEX(N), 16, 10)` instead. See the description of `HEX()`.

- `UPPER(str)`

Returns the string `str` with all characters changed to uppercase according to the current character set mapping. The default is `latin1` (cp1252 West European).

```
mysql> SELECT UPPER('Hej');
-> 'HEJ'
```

See the description of `LOWER()` for information that also applies to `UPPER()`, such as information about how to perform lettercase conversion of binary strings (`BINARY`, `VARBINARY`, `BLOB`) for which these functions are ineffective.

This function is multibyte safe.

12.5.1 String Comparison Functions

Table 12.8 String Comparison Operators

Name	Description
<code>LIKE</code>	Simple pattern matching
<code>NOT LIKE</code>	Negation of simple pattern matching
<code>STRCMP()</code>	Compare two strings

If a string function is given a binary string as an argument, the resulting string is also a binary string. A number converted to a string is treated as a binary string. This affects only comparisons.

Normally, if any expression in a string comparison is case sensitive, the comparison is performed in case-sensitive fashion.

- `expr LIKE pat [ESCAPE 'escape_char']`

Pattern matching using an SQL pattern. Returns 1 (`TRUE`) or 0 (`FALSE`). If either `expr` or `pat` is `NULL`, the result is `NULL`.

The pattern need not be a literal string. For example, it can be specified as a string expression or table column.

Per the SQL standard, `LIKE` performs matching on a per-character basis, thus it can produce results different from the `=` comparison operator:

```
mysql> SELECT 'ä' LIKE 'ae' COLLATE latin1_german2_ci;
+-----+
| 'ä' LIKE 'ae' COLLATE latin1_german2_ci |
+-----+
| 0 |
+-----+
mysql> SELECT 'ä' = 'ae' COLLATE latin1_german2_ci;
+-----+
```

```
| 'ä' = 'ae' COLLATE latin1_german2_ci |
+-----+
|                                     1 |
+-----+
```

In particular, trailing spaces are significant, which is not true for [CHAR](#) or [VARCHAR](#) comparisons performed with the `=` operator:

```
mysql> SELECT 'a' = 'a ', 'a' LIKE 'a ';
+-----+-----+
| 'a' = 'a ' | 'a' LIKE 'a ' |
+-----+-----+
|          1 |             0 |
+-----+-----+
1 row in set (0.00 sec)
```

With [LIKE](#) you can use the following two wildcard characters in the pattern:

- `%` matches any number of characters, even zero characters.
- `_` matches exactly one character.

```
mysql> SELECT 'David!' LIKE 'David_';
-> 1
mysql> SELECT 'David!' LIKE '%D%v%';
-> 1
```

To test for literal instances of a wildcard character, precede it by the escape character. If you do not specify the [ESCAPE](#) character, `\` is assumed.

- `\%` matches one `%` character.
- `_` matches one `_` character.

```
mysql> SELECT 'David!' LIKE 'David\%';
-> 0
mysql> SELECT 'David_' LIKE 'David\_';
-> 1
```

To specify a different escape character, use the [ESCAPE](#) clause:

```
mysql> SELECT 'David_' LIKE 'David|_' ESCAPE '|';
-> 1
```

The escape sequence should be empty or one character long. The expression must evaluate as a constant at execution time. If the [NO_BACKSLASH_ESCAPES](#) SQL mode is enabled, the sequence cannot be empty.

The following two statements illustrate that string comparisons are not case sensitive unless one of the operands is a case sensitive (uses a case-sensitive collation or is a binary string):

```
mysql> SELECT 'abc' LIKE 'ABC';
-> 1
mysql> SELECT 'abc' LIKE _latin1 'ABC' COLLATE latin1_general_cs;
-> 0
mysql> SELECT 'abc' LIKE _latin1 'ABC' COLLATE latin1_bin;
-> 0
mysql> SELECT 'abc' LIKE BINARY 'ABC';
-> 0
```

As an extension to standard SQL, MySQL permits [LIKE](#) on numeric expressions.

```
mysql> SELECT 10 LIKE '1%';
-> 1
```



Note

Because MySQL uses C escape syntax in strings (for example, `\n` to represent a newline character), you must double any `\` that you use in `LIKE` strings. For example, to search for `\n`, specify it as `\\n`. To search for `\`, specify it as `\\\\`; this is because the backslashes are stripped once by the parser and again when the pattern match is made, leaving a single backslash to be matched against.

Exception: At the end of the pattern string, backslash can be specified as `\\`. At the end of the string, backslash stands for itself because there is nothing following to escape. Suppose that a table contains the following values:

```
mysql> SELECT filename FROM t1;
+-----+
| filename |
+-----+
| C:       |
| C:\      |
| C:\Programs |
| C:\Programs\ |
+-----+
```

To test for values that end with backslash, you can match the values using either of the following patterns:

```
mysql> SELECT filename, filename LIKE '%\\' FROM t1;
+-----+-----+
| filename | filename LIKE '%\\' |
+-----+-----+
| C:       | 0 |
| C:\      | 1 |
| C:\Programs | 0 |
| C:\Programs\ | 1 |
+-----+-----+

mysql> SELECT filename, filename LIKE '%\\\\' FROM t1;
+-----+-----+
| filename | filename LIKE '%\\\\' |
+-----+-----+
| C:       | 0 |
| C:\      | 1 |
| C:\Programs | 0 |
| C:\Programs\ | 1 |
+-----+-----+
```

- `expr NOT LIKE pat [ESCAPE 'escape_char']`

This is the same as `NOT (expr LIKE pat [ESCAPE 'escape_char'])`.



Note

Aggregate queries involving `NOT LIKE` comparisons with columns containing `NULL` may yield unexpected results. For example, consider the following table and data:

```
CREATE TABLE foo (bar VARCHAR(10));

INSERT INTO foo VALUES (NULL), (NULL);
```

The query `SELECT COUNT(*) FROM foo WHERE bar LIKE '%baz%';` returns 0. You might assume that `SELECT COUNT(*) FROM foo WHERE bar NOT LIKE '%baz%';` would return 2. However, this is not the case: The second query returns 0. This is because `NULL NOT LIKE expr` always returns `NULL`, regardless of the value of `expr`. The same is true for aggregate queries involving `NULL` and comparisons using `NOT RLIKE` or `NOT REGEXP`. In such cases, you must test explicitly for `NOT NULL` using `OR` (and not `AND`), as shown here:

```
SELECT COUNT(*) FROM foo WHERE bar NOT LIKE '%baz%' OR bar IS NULL;
```

- `STRCMP(expr1,expr2)`

`STRCMP()` returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise.

```
mysql> SELECT STRCMP('text', 'text2');
      -> -1
mysql> SELECT STRCMP('text2', 'text');
      -> 1
mysql> SELECT STRCMP('text', 'text');
      -> 0
```

`STRCMP()` performs the comparison using the collation of the arguments.

```
mysql> SET @s1 = _latin1 'x' COLLATE latin1_general_ci;
mysql> SET @s2 = _latin1 'X' COLLATE latin1_general_ci;
mysql> SET @s3 = _latin1 'x' COLLATE latin1_general_cs;
mysql> SET @s4 = _latin1 'X' COLLATE latin1_general_cs;
mysql> SELECT STRCMP(@s1, @s2), STRCMP(@s3, @s4);
+-----+-----+
| STRCMP(@s1, @s2) | STRCMP(@s3, @s4) |
+-----+-----+
| 0 | 1 |
+-----+-----+
```

If the collations are incompatible, one of the arguments must be converted to be compatible with the other. See [Section 10.1.8.4, “Collation of Expressions”](#).

```
mysql> SELECT STRCMP(@s1, @s3);
ERROR 1267 (HY000): Illegal mix of collations (latin1_general_ci,IMPLICIT)
and (latin1_general_cs,IMPLICIT) for operation 'strcmp'
mysql> SELECT STRCMP(@s1, @s3 COLLATE latin1_general_ci);
+-----+
| STRCMP(@s1, @s3 COLLATE latin1_general_ci) |
+-----+
| 0 |
+-----+
```

12.5.2 Regular Expressions

Table 12.9 String Regular Expression Operators

Name	Description
<code>NOT REGEXP</code>	Negation of <code>REGEXP</code>
<code>REGEXP</code>	Pattern matching using regular expressions
<code>RLIKE</code>	Synonym for <code>REGEXP</code>

A regular expression is a powerful way of specifying a pattern for a complex search.

MySQL uses Henry Spencer's implementation of regular expressions, which is aimed at conformance with POSIX 1003.2. MySQL uses the extended version to support pattern-matching operations performed with the `REGEXP` operator in SQL statements.

This section summarizes, with examples, the special characters and constructs that can be used in MySQL for `REGEXP` operations. It does not contain all the details that can be found in Henry Spencer's `regex(7)` manual page. That manual page is included in MySQL source distributions, in the `regex.7` file under the `regex` directory. See also [Section 3.3.4.7, “Pattern Matching”](#).

Regular Expression Operators

- `expr NOT REGEXP pat, expr NOT RLIKE pat`

This is the same as `NOT (expr REGEXP pat)`.

- `expr REGEXP pat, expr RLIKE pat`

Performs a pattern match of a string expression `expr` against a pattern `pat`. The pattern can be an extended regular expression, the syntax for which is discussed later in this section. Returns `1` if `expr` matches `pat`; otherwise it returns `0`. If either `expr` or `pat` is `NULL`, the result is `NULL`. `RLIKE` is a synonym for `REGEXP`, provided for `mSQL` compatibility.

The pattern need not be a literal string. For example, it can be specified as a string expression or table column.



Note

Because MySQL uses the C escape syntax in strings (for example, `\n` to represent the newline character), you must double any `\` that you use in your `REGEXP` strings.

`REGEXP` is not case sensitive, except when used with binary strings.

```
mysql> SELECT 'Monty!' REGEXP '.*';
-> 1
mysql> SELECT 'new*\n*line' REGEXP 'new\\*\\.\\*line';
-> 1
mysql> SELECT 'a' REGEXP 'A', 'a' REGEXP BINARY 'A';
-> 1 0
mysql> SELECT 'a' REGEXP '^[a-d]';
-> 1
```

`REGEXP` and `RLIKE` use the character set and collations of the arguments when deciding the type of a character and performing the comparison. If the arguments have different character sets or collations, coercibility rules apply as described in [Section 10.1.8.4, “Collation of Expressions”](#).



Warning

The `REGEXP` and `RLIKE` operators work in byte-wise fashion, so they are not multibyte safe and may produce unexpected results with multibyte character sets. In addition, these operators compare characters by their byte values and accented characters may not compare as equal even if a given collation treats them as equal.

Syntax of Regular Expressions

A regular expression describes a set of strings. The simplest regular expression is one that has no special characters in it. For example, the regular expression `hello` matches `hello` and nothing else.

Nontrivial regular expressions use certain special constructs so that they can match more than one string. For example, the regular expression `hello|word` matches either the string `hello` or the string `word`.

As a more complex example, the regular expression `B[an]*s` matches any of the strings `Bananas`, `Baaaaas`, `Bs`, and any other string starting with a `B`, ending with an `s`, and containing any number of `a` or `n` characters in between.

A regular expression for the `REGEXP` operator may use any of the following special characters and constructs:

- `^`

Match the beginning of a string.

```
mysql> SELECT 'fo\info' REGEXP '^fo$';          -> 0
mysql> SELECT 'fofo' REGEXP '^fo$';            -> 1
```

- `$`

Match the end of a string.

```
mysql> SELECT 'fo\no' REGEXP '^fo\no$';         -> 1
mysql> SELECT 'fo\no' REGEXP '^fo$';            -> 0
```

- `.`

Match any character (including carriage return and newline).

```
mysql> SELECT 'fofo' REGEXP '^f.*$';            -> 1
mysql> SELECT 'fo\r\nfo' REGEXP '^f.*$';       -> 1
```

- `a*`

Match any sequence of zero or more `a` characters.

```
mysql> SELECT 'Ban' REGEXP '^Ba*n';             -> 1
mysql> SELECT 'Baaan' REGEXP '^Ba*n';          -> 1
mysql> SELECT 'Bn' REGEXP '^Ba*n';             -> 1
```

- `a+`

Match any sequence of one or more `a` characters.

```
mysql> SELECT 'Ban' REGEXP '^Ba+n';             -> 1
mysql> SELECT 'Bn' REGEXP '^Ba+n';             -> 0
```

- `a?`

Match either zero or one `a` character.

```
mysql> SELECT 'Bn' REGEXP '^Ba?n';             -> 1
mysql> SELECT 'Ban' REGEXP '^Ba?n';            -> 1
mysql> SELECT 'Baan' REGEXP '^Ba?n';          -> 0
```

- `de|abc`

Match either of the sequences `de` or `abc`.

```
mysql> SELECT 'pi' REGEXP 'pi|apa';            -> 1
mysql> SELECT 'axe' REGEXP 'pi|apa';           -> 0
mysql> SELECT 'apa' REGEXP 'pi|apa';           -> 1
mysql> SELECT 'apa' REGEXP '^ (pi|apa)$';       -> 1
mysql> SELECT 'pi' REGEXP '^ (pi|apa)$';       -> 1
mysql> SELECT 'pix' REGEXP '^ (pi|apa)$';      -> 0
```

- `(abc)*`

Match zero or more instances of the sequence `abc`.

```
mysql> SELECT 'pi' REGEXP '^ (pi)*$';          -> 1
mysql> SELECT 'pip' REGEXP '^ (pi)*$';         -> 0
mysql> SELECT 'pypi' REGEXP '^ (pi)*$';        -> 1
```

- `{1}`, `{2,3}`

`{n}` or `{m,n}` notation provides a more general way of writing regular expressions that match many occurrences of the previous atom (or “piece”) of the pattern. `m` and `n` are integers.

- `a*`

Can be written as `a{0,}`.

- `a+`

Can be written as `a{1,}`.

- `a?`

Can be written as `a{0,1}`.

To be more precise, `a{n}` matches exactly `n` instances of `a`. `a{n,}` matches `n` or more instances of `a`. `a{m,n}` matches `m` through `n` instances of `a`, inclusive.

`m` and `n` must be in the range from 0 to `RE_DUP_MAX` (default 255), inclusive. If both `m` and `n` are given, `m` must be less than or equal to `n`.

```
mysql> SELECT 'abcde' REGEXP 'a[bcd]{2}e';      -> 0
mysql> SELECT 'abcde' REGEXP 'a[bcd]{3}e';      -> 1
mysql> SELECT 'abcde' REGEXP 'a[bcd]{1,10}e';    -> 1
```

- `[a-dX]`, `[^a-dX]`

Matches any character that is (or is not, if `^` is used) either `a`, `b`, `c`, `d` or `X`. A `-` character between two other characters forms a range that matches all characters from the first character to the second. For example, `[0-9]` matches any decimal digit. To include a literal `]` character, it must immediately follow the opening bracket `[`. To include a literal `-` character, it must be written first or last. Any character that does not have a defined special meaning inside a `[]` pair matches only itself.

```
mysql> SELECT 'aXbc' REGEXP '[a-dXYZ]';         -> 1
mysql> SELECT 'aXbc' REGEXP '^ [a-dXYZ]$';      -> 0
mysql> SELECT 'aXbc' REGEXP '^ [a-dXYZ]+$';     -> 1
mysql> SELECT 'aXbc' REGEXP '^ [^a-dXYZ]+$';    -> 0
mysql> SELECT 'gheis' REGEXP '^ [^a-dXYZ]+$';   -> 1
mysql> SELECT 'gheisa' REGEXP '^ [^a-dXYZ]+$';  -> 0
```

- `[.characters.]`

Within a bracket expression (written using `[` and `]`), matches the sequence of characters of that collating element. `characters` is either a single character or a character name like `newline`. The following table lists the permissible character names.

The following table shows the permissible character names and the characters that they match. For characters given as numeric values, the values are represented in octal.

Name	Character	Name	Character
NUL	0	SOH	001

Name	Character	Name	Character
STX	002	ETX	003
EOT	004	ENQ	005
ACK	006	BEL	007
alert	007	BS	010
backspace	'\b'	HT	011
tab	'\t'	LF	012
newline	'\n'	VT	013
vertical-tab	'\v'	FF	014
form-feed	'\f'	CR	015
carriage-return	'\r'	SO	016
SI	017	DLE	020
DC1	021	DC2	022
DC3	023	DC4	024
NAK	025	SYN	026
ETB	027	CAN	030
EM	031	SUB	032
ESC	033	IS4	034
FS	034	IS3	035
GS	035	IS2	036
RS	036	IS1	037
US	037	space	' '
exclamation-mark	'!'	quotation-mark	'\"'
number-sign	'#'	dollar-sign	'\$'
percent-sign	'%'	ampersand	'&'
apostrophe	'\''	left-parenthesis	'('
right-parenthesis	')'	asterisk	'*'
plus-sign	'+'	comma	','
hyphen	'-'	hyphen-minus	'_'
period	'.'	full-stop	'.'
slash	'/'	solidus	'/'
zero	'0'	one	'1'
two	'2'	three	'3'
four	'4'	five	'5'
six	'6'	seven	'7'
eight	'8'	nine	'9'
colon	':'	semicolon	';'
less-than-sign	'<'	equals-sign	'='
greater-than-sign	'>'	question-mark	'?'
commercial-at	'@'	left-square-bracket	'['
backslash	'\\'	reverse-solidus	'\\'

Name	Character	Name	Character
right-square-bracket	'] '	circumflex	' ^ '
circumflex-accent	' ^ '	underscore	' _ '
low-line	' _ '	grave-accent	' ` '
left-brace	' { '	left-curly-bracket	' { '
vertical-line	' '	right-brace	' } '
right-curly-bracket	' } '	tilde	' ~ '
DEL	177		

```
mysql> SELECT '~' REGEXP '[[.~.]]';          -> 1
mysql> SELECT '~' REGEXP '[[.tilde.]]';      -> 1
```

- `[=character_class=]`

Within a bracket expression (written using `[` and `]`), `[=character_class=]` represents an equivalence class. It matches all characters with the same collation value, including itself. For example, if `o` and `(+)` are the members of an equivalence class, `[[=o=]]`, `[[=(+)=]]`, and `[o(+)]` are all synonymous. An equivalence class may not be used as an endpoint of a range.

- `[:character_class:]`

Within a bracket expression (written using `[` and `]`), `[:character_class:]` represents a character class that matches all characters belonging to that class. The following table lists the standard class names. These names stand for the character classes defined in the `ctype(3)` manual page. A particular locale may provide other class names. A character class may not be used as an endpoint of a range.

Character Class Name	Meaning
<code>alnum</code>	Alphanumeric characters
<code>alpha</code>	Alphabetic characters
<code>blank</code>	Whitespace characters
<code>cntrl</code>	Control characters
<code>digit</code>	Digit characters
<code>graph</code>	Graphic characters
<code>lower</code>	Lowercase alphabetic characters
<code>print</code>	Graphic or space characters
<code>punct</code>	Punctuation characters
<code>space</code>	Space, tab, newline, and carriage return
<code>upper</code>	Uppercase alphabetic characters
<code>xdigit</code>	Hexadecimal digit characters

```
mysql> SELECT 'justalnums' REGEXP '[[[:alnum:]]]+';          -> 1
mysql> SELECT '!!!' REGEXP '[[[:alnum:]]]+';                -> 0
```

- `[[[:<:]]]`, `[[[:>:]]]`

These markers stand for word boundaries. They match the beginning and end of words, respectively. A word is a sequence of word characters that is not preceded by or followed by word characters. A word character is an alphanumeric character in the `alnum` class or an underscore (`_`).

```
mysql> SELECT 'a word a' REGEXP '[:<:]word[:>:]';    -> 1
mysql> SELECT 'a xword a' REGEXP '[:<:]word[:>:]';    -> 0
```

To use a literal instance of a special character in a regular expression, precede it by two backslash (`\`) characters. The MySQL parser interprets one of the backslashes, and the regular expression library interprets the other. For example, to match the string `1+2` that contains the special `+` character, only the last of the following regular expressions is the correct one:

```
mysql> SELECT '1+2' REGEXP '1+2';                    -> 0
mysql> SELECT '1+2' REGEXP '1\+2';                    -> 0
mysql> SELECT '1+2' REGEXP '1\\+2';                    -> 1
```

12.5.3 Character Set and Collation of Function Results

MySQL has many operators and functions that return a string. This section answers the question: What is the character set and collation of such a string?

For simple functions that take string input and return a string result as output, the output's character set and collation are the same as those of the principal input value. For example, `UPPER(X)` returns a string with the same character string and collation as `X`. The same applies for `INSTR()`, `LCASE()`, `LOWER()`, `LTRIM()`, `MID()`, `REPEAT()`, `REPLACE()`, `REVERSE()`, `RIGHT()`, `RPAD()`, `RTRIM()`, `SOUNDEX()`, `SUBSTRING()`, `TRIM()`, `UCASE()`, and `UPPER()`.



Note

The `REPLACE()` function, unlike all other functions, always ignores the collation of the string input and performs a case-sensitive comparison.

If a string input or function result is a binary string, the string has the `binary` character set and collation. This can be checked by using the `CHARSET()` and `COLLATION()` functions, both of which return `binary` for a binary string argument:

```
mysql> SELECT CHARSET(BINARY 'a'), COLLATION(BINARY 'a');
+-----+-----+
| CHARSET(BINARY 'a') | COLLATION(BINARY 'a') |
+-----+-----+
| binary              | binary                |
+-----+-----+
```

For operations that combine multiple string inputs and return a single string output, the “aggregation rules” of standard SQL apply for determining the collation of the result:

- If an explicit `COLLATE Y` occurs, use `Y`.
- If explicit `COLLATE Y` and `COLLATE Z` occur, raise an error.
- Otherwise, if all collations are `Y`, use `Y`.
- Otherwise, the result has no collation.

For example, with `CASE ... WHEN a THEN b WHEN b THEN c COLLATE X END`, the resulting collation is `X`. The same applies for `UNION`, `||`, `CONCAT()`, `ELT()`, `GREATEST()`, `IF()`, and `LEAST()`.

For operations that convert to character data, the character set and collation of the strings that result from the operations are defined by the `character_set_connection` and `collation_connection` system variables that determine the default connection character set

and collation (see [Section 10.1.4, “Connection Character Sets and Collations”](#)). This applies only to `CAST()`, `CONV()`, `FORMAT()`, `HEX()`, and `SPACE()`.

If there is any question about the character set or collation of the result returned by a string function, use the `CHARSET()` or `COLLATION()` function to find out:

```
mysql> SELECT USER(), CHARSET(USER()), COLLATION(USER());
+-----+-----+-----+
| USER()          | CHARSET(USER()) | COLLATION(USER()) |
+-----+-----+-----+
| test@localhost  | utf8            | utf8_general_ci   |
+-----+-----+-----+
mysql> SELECT CHARSET(COMPRESS('abc')), COLLATION(COMPRESS('abc'));
+-----+-----+
| CHARSET(COMPRESS('abc')) | COLLATION(COMPRESS('abc')) |
+-----+-----+
| binary                  | binary                     |
+-----+-----+
```

12.6 Numeric Functions and Operators

Table 12.10 Numeric Functions and Operators

Name	Description
<code>ABS()</code>	Return the absolute value
<code>ACOS()</code>	Return the arc cosine
<code>ASIN()</code>	Return the arc sine
<code>ATAN()</code>	Return the arc tangent
<code>ATAN2(), ATAN()</code>	Return the arc tangent of the two arguments
<code>CEIL()</code>	Return the smallest integer value not less than the argument
<code>CEILING()</code>	Return the smallest integer value not less than the argument
<code>CONV()</code>	Convert numbers between different number bases
<code>COS()</code>	Return the cosine
<code>COT()</code>	Return the cotangent
<code>CRC32()</code>	Compute a cyclic redundancy check value
<code>DEGREES()</code>	Convert radians to degrees
<code>DIV</code>	Integer division
<code>/</code>	Division operator
<code>EXP()</code>	Raise to the power of
<code>FLOOR()</code>	Return the largest integer value not greater than the argument
<code>LN()</code>	Return the natural logarithm of the argument
<code>LOG()</code>	Return the natural logarithm of the first argument
<code>LOG10()</code>	Return the base-10 logarithm of the argument
<code>LOG2()</code>	Return the base-2 logarithm of the argument
<code>-</code>	Minus operator
<code>MOD()</code>	Return the remainder
<code>%, MOD</code>	Modulo operator
<code>PI()</code>	Return the value of pi
<code>+</code>	Addition operator
<code>POW()</code>	Return the argument raised to the specified power