9.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types `character`, `character varying`, and `text`. Except where noted, these functions and operators are declared to accept and return type `text`. They will interchangeably accept `character varying` arguments. Values of type `character` will be converted to `text` before the function or operator is applied, resulting in stripping any trailing spaces in the `character` value.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9.9. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9.10).

> **Note**
>
> The string concatenation operator (`||`) will accept non-string input, so long as at least one input is of string type, as shown in Table 9.9. For other cases, inserting an explicit coercion to `text` can be used to have non-string input accepted.

Table 9.9. SQL String Functions and Operators

<table>
<tr><th>Function/Operator</th><th>Description</th><th>Example(s)</th></tr>
<tr><td>text || text → text</td><td>Concatenates the two strings.</td><td>'Post' || 'greSQL' → PostgreSQL</td></tr>
<tr><td>text || anynonarray → text
anynonarray || text → text</td><td>Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array || operators. If you want to concatenate an array's text equivalent, cast it to text explicitly.)</td><td>'Value: ' || 42 → Value: 42</td></tr>
<tr><td>text IS [NOT] [form] NORMALIZED → boolean</td><td>Checks whether the string is in the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This expression can only be used when the server encoding is UTF8. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings.</td><td>U&'\0061\0308bc' IS NFD NORMALIZED → t</td></tr>
<tr><td>bit_length (text) → integer</td><td>Returns number of bits in the string (8 times the octet_length).</td><td>bit_length('jose') → 32</td></tr>
<tr><td>char_length (text) → integer
character_length (text) → integer</td><td>Returns number of characters in the string.</td><td>char_length('josé') → 4</td></tr>
<tr><td>lower (text) → text</td><td>Converts the string to all lower case, according to the rules of the database's locale.</td><td>lower('TOM') → tom</td></tr>
<tr><td>normalize (text [, form]) → text</td><td>Converts the string to the specified Unicode normalization form. The optional form key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This function can only be used when the server encoding is UTF8.</td><td>normalize(U&'\0061\0308bc', NFC) → U&'\00E4bc'</td></tr>
<tr><td>octet_length (text) → integer</td><td>Returns number of bytes in the string.</td><td>octet_length('josé') → 5 (if server encoding is UTF8)</td></tr>
<tr><td>octet_length (character) → integer</td><td>Returns number of bytes in the string. Since this version of the function accepts type character directly, it will not strip trailing spaces.</td><td>octet_length('abc '::character(4)) → 4</td></tr>
<tr><td>overlay (string text PLACING newsubstring text FROM start integer [FOR count integer]) → text</td><td>Replaces the substring of string that starts at the start'th character and extends for count characters with newsubstring. If count is omitted, it defaults to the length of newsubstring.</td><td>overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas</td></tr>
<tr><td>position (substring text IN string text) → integer</td><td>Returns first starting index of the specified substring within string, or zero if it's not present.</td><td>position('om' in 'Thomas') → 3</td></tr>
<tr><td>substring (string text [FROM start integer] [FOR count integer]) → text</td><td>Extracts the substring of string starting at the start'th character if that is specified, and stopping after count characters if that is specified. Provide at least one of start and count.</td><td>substring('Thomas' from 2 for 3) → hom
substring('Thomas' from 3) → omas
substring('Thomas' for 2) → Th</td></tr>
<tr><td>substring (string text FROM pattern text) → text</td><td>Extracts the first substring matching POSIX regular expression; see Section 9.7.3.</td><td>substring('Thomas' from '...$') → mas</td></tr>
<tr><td>substring (string text SIMILAR pattern text ESCAPE escape text) → text
substring (string text FROM pattern text FOR escape text) → text</td><td>Extracts the first substring matching SQL regular expression; see Section 9.7.2. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete.</td><td>substring('Thomas' similar '%#"o_a#"_' escape '#') → oma</td></tr>
<tr><td>trim ([LEADING | TRAILING | BOTH] [characters text] FROM string text) → text</td><td>Removes the longest string containing only characters in characters (a space by default) from the start, end, or both ends (BOTH is the default) of string.</td><td>trim(both 'xyz' from 'yxTomxx') → Tom</td></tr>
<tr><td>trim ([LEADING | TRAILING | BOTH] [FROM] string text [, characters text]) → text</td><td>This is a non-standard syntax for trim().</td><td>trim(both from 'yxTomxx', 'xyz') → Tom</td></tr>
<tr><td>upper (text) → text</td><td>Converts the string to all upper case, according to the rules of the database's locale.</td><td>upper('tom') → TOM</td></tr>
</table>

Additional string manipulation functions are available and are listed in Table 9.10. Some of them are used internally to implement the SQL-standard string functions listed in Table 9.9.

Table 9.10. Other String Functions

<table>
<tr><th>Function</th><th>Description</th><th>Example(s)</th></tr>
<tr><td>ascii (text) → integer</td><td>Returns the numeric code of the first character of the argument. In UTF8 encoding, returns the Unicode code point of the character. In other multibyte encodings, the argument must be an ASCII character.</td><td>ascii('x') → 120</td></tr>
<tr><td>btrim (string text [, characters text]) → text</td><td>Removes the longest string containing only characters in characters (a space by default) from the start and end of string.</td><td>btrim('xyxtrimyyx', 'xyz') → trim</td></tr>
<tr><td>chr (integer) → text</td><td>Returns the character with the given code. In UTF8 encoding the argument is treated as a Unicode code point. In other multibyte encodings the argument must designate an ASCII character. chr(0) is disallowed because text data types cannot store that character.</td><td>chr(65) → A</td></tr>
<tr><td>concat (val1 "any" [, val2 "any" [, ...]]) → text</td><td>Concatenates the text representations of all the arguments. NULL arguments are ignored.</td><td>concat('abcde', 2, NULL, 22) → abcde222</td></tr>
<tr><td>concat_ws (sep text, val1 "any" [, val2 "any" [, ...]]) → text</td><td>Concatenates all but the first argument, with separators. The first argument is used as the separator string, and should not be NULL. Other NULL arguments are ignored.</td><td>concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22</td></tr>
<tr><td>format (formatstr text [, formatarg "any" [, ...]]) → text</td><td>Formats arguments according to a format string; see Section 9.4.1. This function is similar to the C function sprintf.</td><td>format('Hello %s, %1$s', 'World') → Hello World, World</td></tr>
<tr><td>initcap (text) → text</td><td>Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.</td><td>initcap('hi THOMAS') → Hi Thomas</td></tr>
<tr><td>left (string text, n integer) → text</td><td>Returns first n characters in the string, or when n is negative, returns all but last |n| characters.</td><td>left('abcde', 2) → ab</td></tr>
<tr><td>length (text) → integer</td><td>Returns the number of characters in the string.</td><td>length('jose') → 4</td></tr>
<tr><td>lpad (string text, length integer [, fill text]) → text</td><td>Extends the string to length length by prepending the characters fill (a space by default). If the string is already longer than length then it is truncated (on the right).</td><td>lpad('hi', 5, 'xy') → xyxhi</td></tr>
<tr><td>ltrim (string text [, characters text]) → text</td><td>Removes the longest string containing only characters in characters (a space by default) from the start of string.</td><td>ltrim('zzzytest', 'xyz') → test</td></tr>
<tr><td>md5 (text) → text</td><td>Computes the MD5 hash of the argument, with the result written in hexadecimal.</td><td>md5('abc') → 900150983cd24fb0d6963f7d28e17f72</td></tr>
<tr><td>parse_ident (qualified_identifier text [, strict_mode boolean DEFAULT true]) → text[]</td><td>Splits qualified_identifier into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is false, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions.) Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to name[].</td><td>parse_ident('"SomeSchema".someTable') → {SomeSchema,sometable}</td></tr>
<tr><td>pg_client_encoding ( ) → name</td><td>Returns current client encoding name.</td><td>pg_client_encoding() → UTF8</td></tr>
<tr><td>quote_ident (text) → text</td><td>Returns the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also Example 43.1.</td><td>quote_ident('Foo bar') → "Foo bar"</td></tr>
<tr><td>quote_literal (text) → text</td><td>Returns the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that quote_literal returns null on null input; if the argument might be null, quote_nullable is often more suitable. See also Example 43.1.</td><td>quote_literal(E'O\'Reilly') → 'O''Reilly'</td></tr>
<tr><td>quote_literal (anyelement) → text</td><td>Converts the given value to text and then quotes it as a literal. Embedded single-quotes and backslashes are properly doubled.</td><td>quote_literal(42.5) → '42.5'</td></tr>
<tr><td>quote_nullable (text) → text</td><td>Returns the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. See also Example 43.1.</td><td>quote_nullable(NULL) → NULL</td></tr>
<tr><td>quote_nullable (anyelement) → text</td><td>Converts the given value to text and then quotes it as a literal; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled.</td><td>quote_nullable(42.5) → '42.5'</td></tr>
<tr><td>regexp_match (string text, pattern text [, flags text]) → text[]</td><td>Returns captured substrings resulting from the first match of a POSIX regular expression to the string; see Section 9.7.3.</td><td>regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}</td></tr>
<tr><td>regexp_matches (string text, pattern text [, flags text]) → setof text[]</td><td>Returns captured substrings resulting from the first match of a POSIX regular expression to the string, or multiple matches if the g flag is used; see Section 9.7.3.</td><td>regexp_matches('foobarbequebaz', 'ba.', 'g') →
{bar}
{baz}</td></tr>
<tr><td>regexp_replace (string text, pattern text, replacement text [, flags text]) → text</td><td>Replaces substrings resulting from the first match of a POSIX regular expression, or multiple substring matches if the g flag is used; see Section 9.7.3.</td><td>regexp_replace('Thomas', '.[mN]a.', 'M') → ThM</td></tr>
<tr><td>regexp_split_to_array (string text, pattern text [, flags text]) → text[]</td><td>Splits string using a POSIX regular expression as the delimiter, producing an array of results; see Section 9.7.3.</td><td>regexp_split_to_array('hello world', '\s+') → {hello,world}</td></tr>
<tr><td>regexp_split_to_table (string text, pattern text [, flags text]) → setof text</td><td>Splits string using a POSIX regular expression as the delimiter, producing a set of results; see Section 9.7.3.</td><td>regexp_split_to_table('hello world', '\s+') →
hello
world</td></tr>
<tr><td>repeat (string text, number integer) → text</td><td>Repeats string the specified number of times.</td><td>repeat('Pg', 4) → PgPgPgPg</td></tr>
<tr><td>replace (string text, from text, to text) → text</td><td>Replaces all occurrences in string of substring from with substring to.</td><td>replace('abcdefabcdef', 'cd', 'XX') → abXXefabXXef</td></tr>
<tr><td>reverse (text) → text</td><td>Reverses the order of the characters in the string.</td><td>reverse('abcde') → edcba</td></tr>
<tr><td>right (string text, n integer) → text</td><td>Returns last n characters in the string, or when n is negative, returns all but first |n| characters.</td><td>right('abcde', 2) → de</td></tr>
<tr><td>rpad (string text, length integer [, fill text]) → text</td><td>Extends the string to length length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.</td><td>rpad('hi', 5, 'xy') → hixyx</td></tr>
<tr><td>rtrim (string text [, characters text]) → text</td><td>Removes the longest string containing only characters in characters (a space by default) from the end of string.</td><td>rtrim('testxxzx', 'xyz') → test</td></tr>
<tr><td>split_part (string text, delimiter text, n integer) → text</td><td>Splits string at occurrences of delimiter and returns the n'th field (counting from one), or when n is negative, returns the |n|'th-from-last field.</td><td>split_part('abc~@~def~@~ghi', '~@~', 2) → def
split_part('abc,def,ghi,jkl', ',', -2) → ghi</td></tr>
<tr><td>strpos (string text, substring text) → integer</td><td>Returns first starting index of the specified substring within string, or zero if it's not present. (Same as position(substring in string), but note the reversed argument order.)</td><td>strpos('high', 'ig') → 2</td></tr>
<tr><td>substr (string text, start integer [, count integer]) → text</td><td>Extracts the substring of string starting at the start'th character, and extending for count characters if that is specified. (Same as substring(string from start for count).)</td><td>substr('alphabet', 3) → phabet
substr('alphabet', 3, 2) → ph</td></tr>
<tr><td>starts_with (string text, prefix text) → boolean</td><td>Returns true if string starts with prefix.</td><td>starts_with('alphabet', 'alph') → t</td></tr>
<tr><td>string_to_array (string text, delimiter text [, null_string text]) → text[]</td><td>Splits the string at occurrences of delimiter and forms the resulting fields into a text array. If delimiter is NULL, each character in the string will become a separate element in the array. If delimiter is an empty string, then the string is treated as a single field. If null_string is supplied and is not NULL, fields matching that string are replaced by NULL.</td><td>string_to_array('xx~~yy~~zz', '~~', 'yy') → {xx,NULL,zz}</td></tr>
<tr><td>string_to_table (string text, delimiter text [, null_string text]) → setof text</td><td>Splits the string at occurrences of delimiter and returns the resulting fields as a set of text rows. If delimiter is NULL, each character in the string will become a separate row of the result. If delimiter is an empty string, then the string is treated as a single field. If null_string is supplied and is not NULL, fields matching that string are replaced by NULL.</td><td>string_to_table('xx~^~yy~^~zz', '~^~', 'yy') →
xx
NULL
zz</td></tr>
<tr><td>to_ascii (string text) → text
to_ascii (string text, encoding name) → text
to_ascii (string text, encoding integer) → text</td><td>Converts string to ASCII from another encoding, which may be identified by name or number. If encoding is omitted the database encoding is assumed (which in practice is the only useful case). The conversion consists primarily of dropping accents. Conversion is only supported from LATIN1, LATIN2, LATIN9, and WIN1250 encodings. (See the unaccent module for another, more flexible solution.)</td><td>to_ascii('Karél') → Karel</td></tr>
<tr><td>to_hex (integer) → text
to_hex (bigint) → text</td><td>Converts the number to its equivalent hexadecimal representation.</td><td>to_hex(2147483647) → 7fffffff</td></tr>
<tr><td>translate (string text, from text, to text) → text</td><td>Replaces each character in string that matches a character in the from set with the corresponding character in the to set. If from is longer than to, occurrences of the extra characters in from are deleted.</td><td>translate('12345', '143', 'ax') → a2x5</td></tr>
<tr><td>unistr (text) → text</td><td>Evaluate escaped Unicode characters in the argument. Unicode characters can be specified as \XXXX (4 hexadecimal digits), \+XXXXXX (6 hexadecimal digits), \uXXXX (4 hexadecimal digits), or \UXXXXXXXX (8 hexadecimal digits). To specify a backslash, write two backslashes. All other characters are taken literally.
If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.
This function provides a (non-standard) alternative to string constants with Unicode escapes (see Section 4.1.2.3).</td><td>unistr('d\0061t\+000061') → data
unistr('d\u0061t\U00000061') → data</td></tr>
</table>

The `concat`, `concat_ws` and `format` functions are variadic, so it is possible to pass the values to be concatenated or formatted as an array marked with the `VARIADIC` keyword (see Section 38.5.6). The array's elements are treated as if they were separate ordinary arguments to the function. If the variadic array argument is NULL, `concat` and `concat_ws` return NULL, but `format` treats a NULL as a zero-element array.

See also the aggregate function `string_agg` in Section 9.21, and the functions for converting between strings and the `bytea` type in Table 9.13.

9.4.1. format

The function `format` produces output formatted according to a format string, in a style similar to the C function `sprintf`.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

formatstr is a format string that specifies how the result should be formatted. Text in the format string is copied directly to the result, except where *format specifiers* are used. Format specifiers act as placeholders in the string, defining how subsequent function arguments should be formatted and inserted into the result. Each *formatarg* argument is converted to text according to the usual output rules for its data type, and then formatted and inserted into the result string according to the format specifier(s).

Format specifiers are introduced by a `%` character and have the form

```
%[position][flags][width]type
```

where the component fields are:

position (optional)

A string of the form *n*$ where *n* is the index of the argument to print. Index 1 means the first argument after *formatstr*. If the *position* is omitted, the default is to use the next argument in sequence.

flags (optional)

Additional options controlling how the format specifier's output is formatted. Currently the only supported flag is a minus sign (`-`) which will cause the format specifier's output to be left-justified. This has no effect unless the *width* field is also specified.

width (optional)

Specifies the *minimum* number of characters to use to display the format specifier's output. The output is padded on the left or right (depending on the `-` flag) with spaces as needed to fill the width. A too-small width does not cause truncation of the output, but is simply ignored. The width may be specified using any of the following: a positive integer; an asterisk (`*`) to use the next function argument as the width; or a string of the form `*n$` to use the *n*th function argument as the width.

If the width comes from a function argument, that argument is consumed before the argument that is used for the format specifier's value. If the width argument is negative, the result is left aligned (as if the `-` flag had been specified) within a field of length `abs`(*width*).

type (required)

The type of format conversion to use to produce the format specifier's output. The following types are supported:

- `s` formats the argument value as a simple string. A null value is treated as an empty string.
- `I` treats the argument value as an SQL identifier, double-quoting it if necessary. It is an error for the value to be null (equivalent to `quote_ident`).
- `L` quotes the argument value as an SQL literal. A null value is displayed as the string `NULL`, without quotes (equivalent to `quote_nullable`).

In addition to the format specifiers described above, the special sequence `%%` may be used to output a literal `%` character.

Here are some examples of the basic format conversions:

```
SELECT format('Hello %s', 'World');
Result: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Result: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Result: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
Result: INSERT INTO locations VALUES('C:\Program Files')
```

Here are examples using *width* fields and the `-` flag:

```
SELECT format('|%10s|', 'foo');
Result: |       foo|

SELECT format('|%-10s|', 'foo');
Result: |foo       |

SELECT format('|%*s|', 10, 'foo');
Result: |       foo|

SELECT format('|%*s|', -10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', -10, 'foo');
Result: |foo       |
```

These examples show use of *position* fields:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Result: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Result: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Result: |       foo|
```

Unlike the standard C function `sprintf`, PostgreSQL's `format` function allows format specifiers with and without *position* fields to be mixed in the same format string. A format specifier without a *position* field always uses the next argument after the last argument consumed. In addition, the `format` function does not require all function arguments to be used in the format string. For example:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Result: Testing three, two, three
```

The `%I` and `%L` format specifiers are particularly useful for safely constructing dynamic SQL statements. See Example 43.1.

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.