9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without standard mathematical conventions (e.g., date/time types) we describe the actual behavior in subsequent sections.

Table 9.4 shows the mathematical operators that are available for the standard numeric types. Unless otherwise noted, operators shown as accepting `numeric_type` are available for all the types `smallint`, `integer`, `bigint`, `numeric`, `real`, and `double precision`. Operators shown as accepting `integral_type` are available for the types `smallint`, `integer`, and `bigint`. Except where noted, each form of an operator returns the same data type as its argument(s). Calls involving multiple argument data types, such as `integer + numeric`, are resolved by using the type appearing later in these lists.

Table 9.4. Mathematical Operators

Operator	Description	Example(s)
numeric_type + *numeric_type* → *numeric_type*	Addition	`2 + 3` → `5`
+ *numeric_type* → *numeric_type*	Unary plus (no operation)	`+ 3.5` → `3.5`
numeric_type - *numeric_type* → *numeric_type*	Subtraction	`2 - 3` → `-1`
- *numeric_type* → *numeric_type*	Negation	`- (-4)` → `4`
numeric_type * *numeric_type* → *numeric_type*	Multiplication	`2 * 3` → `6`

numeric_type / *numeric_type* → *numeric_type*	Division (for integral types, division truncates the result towards zero)	`5.0 / 2` → `2.5000000000000000`
		`5 / 2` → `2`
		`(-5) / 2` → `-2`

numeric_type % *numeric_type* → *numeric_type*	Modulo (remainder); available for `smallint`, `integer`, `bigint`, and `numeric`	`5 % 4` → `1`

`numeric` ^ `numeric` → `numeric`
`double precision` ^ `double precision` → `double precision`
	Exponentiation
	Unlike typical mathematical practice, multiple uses of ^ will associate left to right by default:
		`2 ^ 3` → `8`
		`2 ^ 3 ^ 3` → `512`
		`2 ^ (3 ^ 3)` → `134217728`

|/ `double precision` → `double precision`	Square root	`|/ 25.0` → `5`
||/ `double precision` → `double precision`	Cube root	`||/ 64.0` → `4`
@ *numeric_type* → *numeric_type*	Absolute value	`@ -5.0` → `5.0`
integral_type & *integral_type* → *integral_type*	Bitwise AND	`91 & 15` → `11`
integral_type | *integral_type* → *integral_type*	Bitwise OR	`32 | 3` → `35`
integral_type # *integral_type* → *integral_type*	Bitwise exclusive OR	`17 # 5` → `20`
~ *integral_type* → *integral_type*	Bitwise NOT	`~1` → `-2`
integral_type << `integer` → *integral_type*	Bitwise shift left	`1 << 4` → `16`
integral_type >> `integer` → *integral_type*	Bitwise shift right	`8 >> 2` → `2`

Table 9.5 shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument(s); cross-type cases are resolved in the same way as explained above for operators. The functions working with `double precision` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases can therefore vary depending on the host system.

Table 9.5. Mathematical Functions

Function	Description	Example(s)
`abs` (*numeric_type*) → *numeric_type*	Absolute value	`abs(-17.4)` → `17.4`
`cbrt` (`double precision`) → `double precision`	Cube root	`cbrt(64.0)` → `4`
`ceil` (`numeric`) → `numeric` `ceil` (`double precision`) → `double precision`	Nearest integer greater than or equal to argument	`ceil(42.2)` → `43` `ceil(-42.8)` → `-42`
`ceiling` (`numeric`) → `numeric` `ceiling` (`double precision`) → `double precision`	Nearest integer greater than or equal to argument (same as `ceil`)	`ceiling(95.3)` → `96`
`degrees` (`double precision`) → `double precision`	Converts radians to degrees	`degrees(0.5)` → `28.64788975654116`
`div` (*y* `numeric`, *x* `numeric`) → `numeric`	Integer quotient of *y*/*x* (truncates towards zero)	`div(9, 4)` → `2`
`exp` (`numeric`) → `numeric` `exp` (`double precision`) → `double precision`	Exponential (e raised to the given power)	`exp(1.0)` → `2.7182818284590452`
`factorial` (`bigint`) → `numeric`	Factorial	`factorial(5)` → `120`
`floor` (`numeric`) → `numeric` `floor` (`double precision`) → `double precision`	Nearest integer less than or equal to argument	`floor(42.8)` → `42` `floor(-42.8)` → `-43`
`gcd` (*numeric_type*, *numeric_type*) → *numeric_type*	Greatest common divisor (the largest positive number that divides both inputs with no remainder); returns `0` if both inputs are zero; available for `integer`, `bigint`, and `numeric`	`gcd(1071, 462)` → `21`
`lcm` (*numeric_type*, *numeric_type*) → *numeric_type*	Least common multiple (the smallest strictly positive number that is an integral multiple of both inputs); returns `0` if either input is zero; available for `integer`, `bigint`, and `numeric`	`lcm(1071, 462)` → `23562`
`ln` (`numeric`) → `numeric` `ln` (`double precision`) → `double precision`	Natural logarithm	`ln(2.0)` → `0.6931471805599453`
`log` (`numeric`) → `numeric` `log` (`double precision`) → `double precision`	Base 10 logarithm	`log(100)` → `2`
`log10` (`numeric`) → `numeric` `log10` (`double precision`) → `double precision`	Base 10 logarithm (same as `log`)	`log10(1000)` → `3`
`log` (*b* `numeric`, *x* `numeric`) → `numeric`	Logarithm of *x* to base *b*	`log(2.0, 64.0)` → `6.0000000000000000`
`min_scale` (`numeric`) → `integer`	Minimum scale (number of fractional decimal digits) needed to represent the supplied value precisely	`min_scale(8.4100)` → `2`
`mod` (*y* *numeric_type*, *x* *numeric_type*) → *numeric_type*	Remainder of *y*/*x*; available for `smallint`, `integer`, `bigint`, and `numeric`	`mod(9, 4)` → `1`
`pi` ( ) → `double precision`	Approximate value of π	`pi()` → `3.141592653589793`
`power` (*a* `numeric`, *b* `numeric`) → `numeric` `power` (*a* `double precision`, *b* `double precision`) → `double precision`	*a* raised to the power of *b*	`power(9, 3)` → `729`
`radians` (`double precision`) → `double precision`	Converts degrees to radians	`radians(45.0)` → `0.7853981633974483`
`round` (`numeric`) → `numeric` `round` (`double precision`) → `double precision`	Rounds to nearest integer. For `numeric`, ties are broken by rounding away from zero. For `double precision`, the tie-breaking behavior is platform dependent, but "round to nearest even" is the most common rule.	`round(42.4)` → `42`
`round` (*v* `numeric`, *s* `integer`) → `numeric`	Rounds *v* to *s* decimal places. Ties are broken by rounding away from zero.	`round(42.4382, 2)` → `42.44`
`scale` (`numeric`) → `integer`	Scale of the argument (the number of decimal digits in the fractional part)	`scale(8.4100)` → `4`
`sign` (`numeric`) → `numeric` `sign` (`double precision`) → `double precision`	Sign of the argument (-1, 0, or +1)	`sign(-8.4)` → `-1`
`sqrt` (`numeric`) → `numeric` `sqrt` (`double precision`) → `double precision`	Square root	`sqrt(2)` → `1.4142135623730951`
`trim_scale` (`numeric`) → `numeric`	Reduces the value's scale (number of fractional decimal digits) by removing trailing zeroes	`trim_scale(8.4100)` → `8.41`
`trunc` (`numeric`) → `numeric` `trunc` (`double precision`) → `double precision`	Truncates to integer (towards zero)	`trunc(42.8)` → `42` `trunc(-42.8)` → `-42`
`trunc` (*v* `numeric`, *s* `integer`) → `numeric`	Truncates *v* to *s* decimal places	`trunc(42.4382, 2)` → `42.43`
`width_bucket` (*operand* `numeric`, *low* `numeric`, *high* `numeric`, *count* `integer`) → `integer` `width_bucket` (*operand* `double precision`, *low* `double precision`, *high* `double precision`, *count* `integer`) → `integer`	Returns the number of the bucket in which *operand* falls in a histogram having *count* equal-width buckets spanning the range *low* to *high*. Returns `0` or *count*`+1` for an input outside that range.	`width_bucket(5.35, 0.024, 10.06, 5)` → `3`
`width_bucket` (*operand* `anycompatible`, *thresholds* `anycompatiblearray`) → `integer`	Returns the number of the bucket in which *operand* falls given an array listing the lower bounds of the buckets. Returns `0` for an input less than the first lower bound. *operand* and the array elements can be of any type having standard comparison operators. The *thresholds* array *must be sorted*, smallest first, or unexpected results will be obtained.	`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])` → `2`

Table 9.6 shows functions for generating random numbers.

Table 9.6. Random Functions

Function	Description	Example(s)
`random` ( ) → `double precision`	Returns a random value in the range 0.0 <= x < 1.0	`random()` → `0.897124072839091`
`setseed` (`double precision`) → `void`	Sets the seed for subsequent `random()` calls; argument must be between -1.0 and 1.0, inclusive	`setseed(0.12345)`

The `random()` function uses a simple linear congruential algorithm. It is fast but not suitable for cryptographic applications; see the pgcrypto module for a more secure alternative. If `setseed()` is called, the series of results of subsequent `random()` calls in the current session can be repeated by re-issuing `setseed()` with the same argument. Without any prior `setseed()` call in the same session, the first `random()` call obtains a seed from a platform-dependent source of random bits.

Table 9.7 shows the available trigonometric functions. Each of these functions comes in two variants, one that measures angles in radians and one that measures angles in degrees.

Table 9.7. Trigonometric Functions

Function	Description	Example(s)
`acos` (`double precision`) → `double precision`	Inverse cosine, result in radians	`acos(1)` → `0`
`acosd` (`double precision`) → `double precision`	Inverse cosine, result in degrees	`acosd(0.5)` → `60`
`asin` (`double precision`) → `double precision`	Inverse sine, result in radians	`asin(1)` → `1.5707963267948966`
`asind` (`double precision`) → `double precision`	Inverse sine, result in degrees	`asind(0.5)` → `30`
`atan` (`double precision`) → `double precision`	Inverse tangent, result in radians	`atan(1)` → `0.7853981633974483`
`atand` (`double precision`) → `double precision`	Inverse tangent, result in degrees	`atand(1)` → `45`
`atan2` (*y* `double precision`, *x* `double precision`) → `double precision`	Inverse tangent of *y*/*x*, result in radians	`atan2(1, 0)` → `1.5707963267948966`
`atan2d` (*y* `double precision`, *x* `double precision`) → `double precision`	Inverse tangent of *y*/*x*, result in degrees	`atan2d(1, 0)` → `90`
`cos` (`double precision`) → `double precision`	Cosine, argument in radians	`cos(0)` → `1`
`cosd` (`double precision`) → `double precision`	Cosine, argument in degrees	`cosd(60)` → `0.5`
`cot` (`double precision`) → `double precision`	Cotangent, argument in radians	`cot(0.5)` → `1.830487721712452`
`cotd` (`double precision`) → `double precision`	Cotangent, argument in degrees	`cotd(45)` → `1`
`sin` (`double precision`) → `double precision`	Sine, argument in radians	`sin(1)` → `0.8414709848078965`
`sind` (`double precision`) → `double precision`	Sine, argument in degrees	`sind(30)` → `0.5`
`tan` (`double precision`) → `double precision`	Tangent, argument in radians	`tan(1)` → `1.5574077246549023`
`tand` (`double precision`) → `double precision`	Tangent, argument in degrees	`tand(45)` → `1`

Note

Another way to work with angles measured in degrees is to use the unit transformation functions `radians()` and `degrees()` shown earlier. However, using the degree-based trigonometric functions is preferred, as that way avoids round-off error for special cases such as `sind(30)`.

Table 9.8 shows the available hyperbolic functions.

Table 9.8. Hyperbolic Functions

Function	Description	Example(s)
`sinh` (`double precision`) → `double precision`	Hyperbolic sine	`sinh(1)` → `1.1752011936438014`
`cosh` (`double precision`) → `double precision`	Hyperbolic cosine	`cosh(0)` → `1`
`tanh` (`double precision`) → `double precision`	Hyperbolic tangent	`tanh(1)` → `0.7615941559557649`
`asinh` (`double precision`) → `double precision`	Inverse hyperbolic sine	`asinh(1)` → `0.881373587019543`
`acosh` (`double precision`) → `double precision`	Inverse hyperbolic cosine	`acosh(1)` → `0`
`atanh` (`double precision`) → `double precision`	Inverse hyperbolic tangent	`atanh(0.5)` → `0.5493061443340548`

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.