Table of Contents

I am starting with the numeric functions. the function we use the most is the Round function, following by Floor and Ceiling. We also use the Rand to generate random numbers.

These demos are all run with literals- such as shown here. Instead of including these selects, I have used the expression as the column header for the result. I have adjusted column widths to fit the page.

One thing to watch for with some versions of MySQL and functions is to avoid spaces between the name of the function: use `ABS(12)` and not `ABS (12)`. There is a switch to allow the inclusion of a space but it causes other problems in writing SQL.

1. Math functions: Abs, Power, Sign, Sqrt

I am discussing Abs, Sign, Power, Mod, and Sqrt here because they are pretty simple. T-SQL also includes functions to do trig calculations; we won't be using those.

Demo 01: **ABS** returns the absolute value of the argument

```
+---------+----------+--------+------------+
| ABS(12) | ABS(-12) | ABS(0) | ABS(45.34) |
+---------+----------+--------+------------+
|      12 |       12 |      0 |      45.34 |
+---------+----------+--------+------------+
```

Demo 02: **SIGN** returns only 0, 1, or –1;

if the argument is 0, sign returns 0;
if the argument is positive, sign returns +1;
if the argument is negative, sign returns -1.

```
+---------+----------+-----------+-------------+
| SIGN(0) | SIGN(12) | SIGN(-12) | SIGN(856.9) |
+---------+----------+-----------+-------------+
|       0 |        1 |        -1 |           1 |
+---------+----------+-----------+-------------+
```

Demo 03: **Power**(a, b), is used to calculate a raised to the b power ; POW is an alias for Power

```
+-------------+-------------+------------------+---------------+
| POWER(3, 2) | POWER(10, 3) | POWER(4.5, 3.2)  | POWER(10, -3) |
+-------------+-------------+------------------+---------------+
|           9 |        1000 | 123.106233510191 |         0.001 |
+-------------+-------------+------------------+---------------+
+----------------+--------------------+---------------------+
|  POWER(10, -3) |  POWER(10.0000, -3) |  POWER(4.5000, 3.2) |
+----------------+--------------------+---------------------+
|          0.001 |              0.001 |  123.10623351019093 |
+----------------+--------------------+---------------------+
```

Demo 04: **SQRT** returns the square root of its argument; the argument must be a non- negative number. Note that MySQL does not report an error with a negative argument, instead it returns a null. You can nest the sqrt and the abs function

```
+----------+------------------+-----------+-------------------+
| SQRT(64) | SQRT(68.56)      | SQRT(-45) | SQRT(ABS(-45))    |
+----------+------------------+-----------+-------------------+
|        8 | 8.28009661779378 |      NULL | 6.708203932499369 |
+----------+------------------+-----------+-------------------+
```

Demo 05: **MOD** takes two numbers and returns the remainder after division; you can also use the % operator**.** Mod makes the most sense when used with integers. If mod (a, 1) = 0 then a is an integer

If mod(a, 2) = 0 then a is even;
If mod(a, 2) = 1 or -1 then a is odd
Note the result when the second argument is 0.

```
+-------------+-----------+-----------+
| MOD(18, 12) | MOD(18, 6) | MOD(6, 18) |
+-------------+-----------+-----------+
|           6 |          0 |          6 |
+-------------+-----------+-----------+
+---------------+--------------+----------------+
| MOD(18.6, 12) | MOD(18, 6.4) | MOD(18.6, 6.4) |
+---------------+--------------+----------------+
|           6.6 |          5.2 |            5.8 |
+---------------+--------------+----------------+
+-----------+-----------+-----------+-------------+
| MOD(10,3) | MOD(-10,3) | MOD(10,-3) | MOD(-10,-3) |
+-----------+-----------+-----------+-------------+
|         1 |         -1 |          1 |          -1 |
+-----------+-----------+-----------+-------------+
+------------+----------------+----------------+-----------+------------+
| MOD(128,1) | MOD(128.002,1) | MOD(128.00, 1) |  MOD(35,2) | MOD(368,2) |
+------------+----------------+----------------+-----------+------------+
|          0 |          0.002 |           0.00 |          1 |          0 |
+------------+----------------+----------------+-----------+------------+
+----------+-----------+----------+
| MOD(0,3) | MOD(10,0) | MOD(0,0) |
+----------+-----------+----------+
|        0 |      NULL |     NULL |
+----------+-----------+----------+
```

2. Rounding functions

The next set of numeric functions returns a value close to the argument.

Demo 06: **ROUND** returns the number rounded at the specified precision.

The precision defaults to 0. You can have a negative precision.

```
+-------------------+-----------------+-----------------+--------------------+
| ROUND( 45.678, 0) | ROUND( 45.2, 0) | ROUND( 46.5, 0) | ROUND( 45.678, 2) |
+-------------------+-----------------+-----------------+--------------------+
|                46 |              45 |              47 |              45.68 |
+-------------------+-----------------+-----------------+--------------------+

+-----------------+--------------------+-----------------+-----------------+
| ROUND(-46.5, 0) | ROUND( 345.67, -2) | ROUND( 45, -1 ) | ROUND( 45, -2 ) |
+-----------------+--------------------+-----------------+-----------------+
|             -47 |                300 |              50 |               0 |
+-----------------+--------------------+-----------------+-----------------+
```

If you use Round with an exact-value number (a decimal) then round uses the round away from zero at the half way mark rule.) If you use round with a floating point number (such as a number in E notation) then it rounds to the next even value.

These are all exact-value literals and round away from zero at the .5 mark.

```
Select round(2.5), round(3.5), round(-2.5), round(-3.5);
+------------+------------+-------------+-------------+
| round(2.5) | round(3.5) |  round(-2.5) | round(-3.5) |
+------------+------------+-------------+-------------+
|          3 |          4 |           -3 |          -4 |
+------------+------------+-------------+-------------+
```

By using a literal expresses in E notation, I have a floating point number and these round to the nearest even value. You may see this called Banker's rounding

```
Select round(2.5E0), round(3.5E0), round(-2.5E0), round(-3.5E0);
+--------------+--------------+---------------+---------------+
| round(2.5E0) | round(3.5E0) |  round(-2.5E0) | round(-3.5E0) |
+--------------+--------------+---------------+---------------+
|            2 |            4 |             -2 |            -4 |
+--------------+--------------+---------------+---------------+
```

Demo 07: **Truncate** is similar to round except that it drops digits after the number indicated in the second argument. If the second argument is negative, it truncates to the left of the decimal returning zeros in those places in the value.

```
+----------------------+----------------------+------------------------+
| Truncate( 45.678, 0) | Truncate( 45.678, 2) | Truncate( 453446.5, -2) |
+----------------------+----------------------+------------------------+
|                   45 |                45.67 |                  453400 |
+----------------------+----------------------+------------------------+

+-----------------------+-----------------------+-----------------------+
| TRUNCATE(345.67, - 2) | TRUNCATE(399.99, - 2) | TRUNCATE(399.99, - 5) |
+-----------------------+-----------------------+-----------------------+
|                   300 |                   300 |                     0 |
+-----------------------+-----------------------+-----------------------+
```

Demo 08: **CEILING** returns the smallest integer greater than or equal to the argument. CEIL is an alias for CEILING,

```
+-------------+--------------+---------------+----------------+
| CEILING(10) | CEILING(10.2) | CEILING(10.8) | CEILING(-10.5) |
+-------------+--------------+---------------+----------------+
|          10 |            11 |            11 |            -10 |
+-------------+--------------+---------------+----------------+
```

Demo 09: **FLOOR** returns the largest integer less than or equal to the argument

```
+-----------+-------------+-------------+--------------+
| FLOOR(10) | FLOOR(10.2) | FLOOR(10.8) | FLOOR(-10.5) |
+-----------+-------------+-------------+--------------+
|        10 |          10 |          10 |          -11 |
+-----------+-------------+-------------+--------------+
```

Comparison of Round, Ceiling, Floor, and Truncate. Suppose you had calculated a numeric value and you needed to display it with 2 digits after the decimal. How would you want to display the value 25.0279? This often comes up in issues such as calculating sales tax- do we round up or round down? The decision about which one is correct is a business decision- not a programming decision. But we can use the following small table to see our choices in terms of SQL.

Demo 10: Create and populate table. Note that I am using a two-part name for the table to put it into the a_testbed database.

```
Create table a_testbed.z_tst_floats( id int, val_1 float);
insert into a_testbed. z_tst_floats values
  (1,  25.0034) , (2,  25.0079)
, (3,  25.0279) , (4,  25.4239)
, (5, -25.0279) , (6, -25.4239)
;
```

Demo 11: Comparison query. Before you look at the results, think how you would round each number in the table.

```
select id, val_1
, ROUND(val_1,2)                    as "round"
, TRUNCATE(val_1,2)                 as "truncate"
, CEILING(val_1 * 100.00)/100.00    as "ceil"
, FLOOR(val_1* 100.00)/100.00       as "floor"
from a_testbed. z_tst_floats;
+------+----------+--------+----------+------------+------------+
| id   | val_1    | round  | truncate | ceil       | floor      |
+------+----------+--------+----------+------------+------------+
|    1 |  25.0034 |  25.00 |    25.00 |  25.010000 |  25.000000 |
|    2 |  25.0079 |  25.01 |    25.00 |  25.010000 |  25.000000 |
|    3 |  25.0279 |  25.03 |    25.02 |  25.030000 |  25.020000 |
|    4 |  25.4239 |  25.42 |    25.42 |  25.430000 |  25.420000 |
|    5 | -25.0279 | -25.03 |   -25.02 | -25.020000 | -25.030000 |
|    6 | -25.4239 | -25.42 |   -25.42 | -25.420000 | -25.430000 |
+------+----------+--------+----------+------------+------------+
```

The calculation multiplies the value by 100 before doing ceiling or floor since they return integers. For example

```
val_1           => 25.0034
val_1 * 100     => 2500.34
ceiling(val_1 * 100.00          => 2501
ceiling(val_1 * 100.00)/100.00  => 25.01
```

Speaking loosely the Ceiling function always goes up to the next value, Floor always goes down and Round goes to the nearest value.

Demo 12: Using a negative precision of -1 with round gives our price values rounded off to the nearest 10 dollars.

```
Select  quoted_price as price
, round(quoted_price, -1) as Price_10
, round(quoted_price, -2) as Price_100
From orderEntry.OrderDetails
limit 10;
+--------+----------+-----------+
| price  | Price_10 | Price_100 |
+--------+----------+-----------+
|  25.00 |       30 |         0 |
|  12.95 |       10 |         0 |
| 150.00 |      150 |       200 |
| 255.95 |      260 |       300 |
|  49.99 |       50 |         0 |
|  22.50 |       20 |         0 |
| 149.99 |      150 |       100 |
| 149.99 |      150 |       100 |
| 149.99 |      150 |       100 |
|   4.99 |        0 |         0 |
+--------+----------+-----------+
```

3. Random values- pseudo-random values

We use the term random numbers to refer to a series of numbers that are produced by some process where the next number to be generated cannot be predicted. We also assume with random numbers that if we generate a set of 10,000 random integers between 1 and 10, then each of the values 1 through 10 should occur approximately the same number of times. Of course, we are not quite saying what "approximately" the same number of times really means.

There are discussions about what "truly" random means and for some applications there are random numbers that are based on using atmosphere noise or the degradation of atomic particles.

For many programming purposes, pseudo-random numbers are good enough. Pseudo-random numbers are generated by an algorithm which is provided with a seed value to get the series of values started. If you give the generator the same seed it will produce the same series of numbers. This is very good for test purposes, so that you can run your program repeatedly against the same series of test data. The pseudo-random number generators also work without a seed, in which case the seed is generally based on the computer time- that means each time you run the generator you get a different set of numbers. This is also good for testing programs.

Many pseudo-random number generators allow you to also specify the range of values you want as output- for example integers between 1 and 100, or floating point numbers between -10 and +10.

One use for random numbers is to create test values to insert into a table. Generating 100 or 100,000 rows of random test data can be handled with random functions.

Demo 13: **RAND** returns a pseudo-random value **between** 0 and 1. (0 <= value < 1.0 ; it should generate a 0 value sometime and never a value of 1).

You can supply an integer seed value. This is a sample run with no seed. If you run this, you would expect to get different value each time.

```
Select rand() as col_1, rand() as col_2, rand()as col_3;
+------------------+------------------+------------------+
| col_1            | col_2            | col_3            |
+------------------+------------------+------------------+
| 0.837175076675764 | 0.374826394370595 | 0.362606772559329 |
+------------------+------------------+------------------+
```

Suppose we show each row from a small table- I will use the table vt_animals types and also display the values calculated by rand(). If we do this twice, we get different random values

```
select an_type, rand() from  vt_animal_types limit 5;
+------------+--------------------+
| an_type    | rand()             |
+------------+--------------------+
| bird       |  0.9885779628423769 |
| capybara   | 0.37835228478382554 |
| cat        |  0.9260275661256421 |
| chelonian  |  0.4950810070103789 |
| crocodilian |  0.6973223674086131 |
+------------+--------------------+

select an_type, rand() from  vt_animal_types limit 5;
+------------+----------------------+
| an_type    | rand()               |
+------------+----------------------+
| bird       | 0.0013688467455737728 |
| capybara   |    0.9148771622356746 |
| cat        |    0.5702793016752967 |
| chelonian  |   0.10676505240310454 |
| crocodilian |    0.8229873877232777 |
+------------+----------------------+
```

But if I do this with a seed, then the series of random values is consistent between runs.

```
select an_type, rand(852) from  vt_animal_types limit 5;
+------------+--------------------+
| an_type    | rand(852)          |
+------------+--------------------+
| bird       | 0.31122946209388735 |
| capybara   | 0.24491867352735128 |
| cat        |  0.2909050120887393 |
| chelonian  |  0.7197690705952878 |
| crocodilian |   0.726130347443866 |
+------------+--------------------+
```

```
select an_type, rand(852) from  vt_animal_types limit 5;
+-------------+---------------------+
| an_type     | rand(852)           |
+-------------+---------------------+
| bird        | 0.31122946209388735 |
| capybara    | 0.24491867352735128 |
| cat         |  0.2909050120887393 |
| chelonian   |  0.7197690705952878 |
| crocodilian |   0.726130347443866 |
+-------------+---------------------+
```

With a different seed:

```
 select an_type, rand(999) from  vt_animal_types limit 5;
+-------------+---------------------+
| an_type     | rand(999)           |
+-------------+---------------------+
| bird        |  0.0881465143413716 |
| capybara    | 0.10258702012019885 |
| cat         | 0.24849558831052443 |
| chelonian   |  0.9347169119256706 |
| crocodilian |  0.9280978254304247 |
+-------------+---------------------+

 select an_type, rand(999) from  vt_animal_types limit 5;
+-------------+---------------------+
| an_type     | rand(999)           |
+-------------+---------------------+
| bird        |  0.0881465143413716 |
| capybara    | 0.10258702012019885 |
| cat         | 0.24849558831052443 |
| chelonian   |  0.9347169119256706 |
| crocodilian |  0.9280978254304247 |
+-------------+---------------------+
```

Demo 14: You can manipulate the value returned by the function to get results such as a random integer between 10 and 25.

```
Select Floor(rand() * 16)  + 10  as col_1;
```

There are 16 values between 10 and 25 (including the endpoints). If I use the expression in the previous demo to generate 1600 values, I would expect to get each values approximate 100 times. Using a loop technique I generated those numbers and counted them. The results were are shown here. Note that the number 26 did not occur at all. And that the number of occurrences of 10 and 25 is pretty much the same.

number	10	11	12	13	14	15	16	17
count	106	96	94	99	90	103	92	106
number	18	19	20	21	22	23	24	25
count	113	104	98	90	92	110	101	106

Don't use round- that causes the endpoints of the range to occur half as often as the other values.

If I use the expression round(10 + rand() * 15), 0) generate 1600 values, I get a count of 45 for value 10 and 47 for value 25 and the other values were all between 96 and 122. This certainly does not give the numbers at the end points the same chance as the others.