Table of Contents

1. Numeric functions

We will use the following numeric functions:

Abs	Power	Round	Ceiling	Mod
Sign	Sqrt	Truncate	Floor	Rand

MySQL also includes functions to do trig and log calculations; we won't be using those

These demos are all run with literals- such as shown here. Since MySQL uses the expression for the column headers I am not including the sql statements.

```
Select ABS(12), ABS(-12), ABS(0);
```

One thing to watch for with MySQL 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.

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

```
+---------+----------+-----------+-----------+
| SIGN(0) | SIGN(12) | SIGN(-12) | SIGN(8.9) |
+---------+----------+-----------+-----------+
|       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 |
+----------+-----------------+-----------+------------------+
```

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

Demo 05: **ROUND** returns the number rounded at the specified precision.

The precision defaults to 0. You can have a negative precision. Round rounds up at .5 for exact value.

```
+-------------------+----------------+-----------------+-------------------+
| 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 round 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 |
+--------------+--------------+----------------+---------------+
1 row in set (0.00 sec)
```

Demo 06: **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 |
+----------------------+----------------------+-------------------------+
```

Demo 07: **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 08: **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 |
+-----------+-------------+-------------+--------------+
```

Demo 09: **MOD** takes two numbers and returns the remainder after division; you can also use the % operator**.**

```
+------------+------------------+
| mod( 25,7) | mod( 12.45, 2.3) |
+------------+------------------+
|          4 |             0.95 |
+------------+------------------+

+-----------+-----------+-----------+-------------+
| mod(10,3) | mod(-10,3) | mod(10,-3) | mod(-10,-3) |
+-----------+-----------+-----------+-------------+
|         1 |         -1 |          1 |          -1 |
+-----------+-----------+-----------+-------------+


+----------+----------+----------+
| mod(0,3) | mod(10,0) | mod(0,0) |
+----------+----------+----------+
|        0 |      NULL |     NULL |
+----------+----------+----------+
```

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_numerics( id int, val_1 float);
insert into a_testbed.z_numerics 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_numerics
;
```

```
+------+----------+--------+----------+------------+------------+
| 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; Convert(int truncates.

Demo 12: Using a negative precision of -1 with round gives 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 a_oe.order_details
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 |
+--------+----------+-----------+
```

2. 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 data can be handled with random functions.

Demo 13: **RAND** returns a pseudo-random value between 0 and 1 exclusive

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

This is a run with a seed and the same seed is used for each call to Rand. The seed determines the starting value and each call to random starts with that seed. With a different seed you get a different starting value.

```
Select rand(5) as col_1, rand(5) as col_2, rand(5)as col_3;
+-------------------+-------------------+-------------------+
| col_1             | col_2             | col_3             |
+-------------------+-------------------+-------------------+
| 0.406135974830143 | 0.406135974830143 | 0.406135974830143 |
+-------------------+-------------------+-------------------+
```

This time the first call to rand gets a seed and the second two calls use the next two values generated. This query will always return the first value and the other two will vary.

```
Select rand(5)  as col_1, rand() as col_2, rand()as col_3;
+--------------------+--------------------+--------------------+
| col_1              | col_2              | col_3              |
+--------------------+--------------------+--------------------+
| 0.40613597483014313 | 0.5978257279822843 | 0.5091723227027546 |
+--------------------+--------------------+--------------------+

Select rand(5)  as col_1, rand() as col_2, rand()as col_3;
+--------------------+--------------------+--------------------+
| col_1              | col_2              | col_3              |
+--------------------+--------------------+--------------------+
| 0.40613597483014313 | 0.7523844603005652 | 0.2344053641282072 |
+--------------------+--------------------+--------------------+

Select rand(5)  as col_1, rand() as col_2, rand()as col_3;
+--------------------+--------------------+--------------------+
| col_1              | col_2              | col_3              |
+--------------------+--------------------+--------------------+
| 0.40613597483014313 | 0.9148734704729854 | 0.8711512907139504 |
+--------------------+--------------------+--------------------+
```

Demo 14: You can manipulate the value returned by the function to get results such as a random integer between 1 and 20.

```
Select Floor(rand() * 20 + 1 ) as col_1;
```

For a random number in a range you can use

```
rand()*(value1_max-value1_min)+value1_min
```

To obtain a random positive integer R in the range `low <= R < high`, use the expression

```
FLOOR(low + RAND() * (high – low))
```

Suppose you wanted random values from 0.3 to 0.9 with 1 digit after the decimal

```
Select (floor(rand() * (10-3)) + 3 )/10;
```