MySQL\_for\_Python\_Albert\_c09

e and Time Values

When beginning to program database-oriented applications, it is not uncommon to

focus on processing data and to forget about matters of time and timing. If a program

is simply storing addresses or otherwise dealing with data that is not naturally

time-sensitive, it is easy to overlook matters of date and time. However, storing date

and time for data and activity on the server allows for several desiderata, some of

which are:

•Rolling back documents to an earlier draft version without affecting the

current draft

•Figuring out why a process or subprocess jammed up or failed

•Forensic analysis of a compromised server

In this chapter, we will see:

•What data types MySQL supports for date and time

•When to use which data type and in what format and range

•What functions MySQL supports for managing temporal values

•The most frequently used functions for managing date and time

As mentioned in the previous chapter, the project for this chapter will build on the

web application we began earlier. After looking at MySQL's date and time support

and how to use it in Python, we will add similar functionality to our web-based

administration program.

Date and time data types in MySQL

MySQL supports five data types for temporal matters: DATETIME, DATE, TIMESTAMP,

TIME, and YEAR. These serve as types to be used when architecting a table like INT

and VARCHAR that were employed in earlier chapters.Date and Time Values

DATETIME

The DATETIME data type is used to specify a value that includes both the date and

the time. It is important to realize that DATETIME accepts its values from the user like

any other data type. It does not automatically generate values. For that purpose, one

should use the TIMESTAMP type (seen later in this chapter).

Output format

The DATETIME type receives data in several formats, but returns it in only one.

YYYY-MM-DD HH:MM:SS

For example, the BBC's evening report of the Berlin Wall being torn down began

at nine o'clock in the evening on November 10, 1989 and ran for 37 minutes and 9

seconds. A DATETIME value depicting the moment of its conclusion would read:

1989-10-10 21:37:09

Input formats

Where MySQL always returns DATETIME values in the same format, it will accept

values in several formats:

•

Four- or two-digit years

Either YYYY-MM-DD HH:MM:SS or YY-MM-DD HH:MM:SS is acceptable

•

Any delimiter, as long as it is consistently applied

The above sample date could be input as 1989.10.10 21@37@09 or 89/10/10

21\*37\*09.

•

No delimiters

So we could just input 19891010213709, and MySQL would understand it

appropriately.

•

No delimiters with a two-digit year

So 891010213709 is fair game.

DATETIME values may also be passed from DATE and TIME functions like NOW(). See

the section on DATE and TIME functions, below.

[ 248 ]Chapter 9

Input range

MySQL insists that all dates be after the turn of the first millennium CE—1000-01-

01. The turn of the millennium here is literal. The exact beginning of acceptable dates

and times is 1000-01-01 00:00:00.

On the far terminus of the range, MySQL cannot handle dates that are seen post the

end of New Year's Eve 9999, or 9999-12-31 23:59:59.

Where MySQL can handle the sundry formats mentioned above, these must always

reconcile to a date-time combination within this range. If a value does not, MySQL

will record a zero value. So a value of either 0999-13-45 25:73:62 or 10000-14-35

25:61:61 would become 0000-00-00 00:00:00.

This is something to watch out for. When MySQL renders your data with this series

of zeroes, it offers no signifier of the correction. Therefore, you will not be able to tell

a bad value from a good one as both will look like 0000-00-00 00:00:00.

Using DATETIME in a CREATE statement

As mentioned previously, DATETIME is a data type. It is therefore used in table

creation to specify the data type of a column. If, for example, we were keeping a

record of dates and times of birth, a table creation statement might read as follows:

CREATE TABLE birthtimes(birthid INT NOT NULL AUTO\_INCREMENT PRIMARY

KEY, babyname VARCHAR(30) DEFAULT '', birthtime DATETIME);

We would then populate that table with INSERT statements like:

INSERT INTO birthtimes(babyname, birthtime) VALUES('Johnny', '2005-12-

02 03:15:46');

DATE

The DATE type is a shorthand means of using only the first half of the DATETIME

value format.

Output and Input formats

The format of output for such types follows that of DATETIME:

•

YYYY-MM-DD

For input, DATE accepts any of the following formats:

•

Four- or two-digit years

[ 249 ]Date and Time Values

Either YYYY-MM-DD or YY-MM-DD

•

Any delimiter, as long as it is consistently applied

So YYYY/MM/DD or YY/MM/DD or YYYY\*MM\*DD. Again, consistency is the key.

•

No delimiters

So YYYYMMDD is allowed, and MySQL massages it to suit its needs.

•

No delimiters with a two-digit year

So we can use YYMMDD.

Input range

DATE also has a similar range to DATETIME—from 1000-01-01 through 9999-12-31.

As with the first data type that we discussed in this chapter, values outside this range

will be zeroed out to 0000-00-00 with no warning being given.

TIMESTAMP

The TIMESTAMP data type differs from the previous two in several important ways.

TIMESTAMP values follow the same format as DATETIME's long form:

YYYY-MM-DD HH:MM:SS

This is a fixed width and cannot be changed, even by setting a default (see the

following section for more on this).

Input of values

TIMESTAMP is an auto-generated data type. Therefore, it's value is not specified in

an INSERT statement but is culled from the server's local time. For the local time,

MySQL uses the clock of the hardware on which it is running by accessing the

operating system's clock.

Internally, MySQL always deals in UTC (Universal Time, Coordinated—a so-called

backronym for Coordinated Universal Time). In common use, UTC is the same as

Greenwich Mean Time (GMT), though the two may differ by up to 0.9 seconds.

If the server is not on UTC, the server's time is converted to UTC for purposes of data

storage and then converted back when that value is requested. Therefore, if a record

was saved under one time zone and retrieved after the time zone value was changed

(for example, Daylight Saving Time, British Summer Time, or similar), the value that

is returned will be different than the value that was given by the user.

[ 250 ]Chapter 9

Range

The TIMESTAMP type differs from DATETIME, wherein its range is much more

restricted. MySQL uses the Unix epoch as the beginning of its range: 1970-01-01

00:00:01. The terminus for valid timestamps is 2038-01-19 03:14:07. All times

are in UTC.

Why January 19, 2038 at 3:14:07? Initially, MySQL was developed

for Unix-like systems. All Unix systems currently share a bug called

variously the year 2038 bug, the Y2K28 bug, or just the Unix Millenium

Bug. The issue results from the problem of 32-bit signed integers being

used to save dates and how those values are processed internally, in

binary. The timestamp for the MySQL terminus is as follows:

2038-01-19 03:14:07 in binary is 11111111 11111111 11111111

11111111

As all addition inside a computer occurs in binary, the computer will

simply add one to this value which will then bring the binary value of the

timestamp to the following:

00000000 00000000 00000000 00000000 or 1901-12-13 20:45:52

The one at the far left of the number drops off the display and disappears

from the system. It is not clear how many systems will be impacted

by this bug. It is known that Unix-like systems are not alone in being

impacted. For more on the year 2038 bug, including AOL's database crash

of May 2006, see http://en.wikipedia.org/wiki/Year\_2038\_

problem

Invalid TIMESTAMP values are converted to 0. Therefore, if a date of 1963-11-22

12:30:00 is used, it cannot be a TIMESTAMP type, but must be a DATETIME.

Defaults, initialization, and updating

MySQL allows timestamps to be set manually or automatically. If set manually, you

need to give a default value. If set automatically, MySQL will use the system time to

initialize the table. Updating can occur under either circumstance.

To affect a default TIMESTAMP value that is not coordinated with the system time, one

would use the following in a table creation statement:

TIMESTAMP DEFAULT 0

[ 251 ]Date and Time Values

If one wanted to coordinate the timestamp to the system time only at startup, set it to

default to the current time:

TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

This sets the timestamp at initialization.

If the table is created with the argument for TIMESTAMP as ON UPDATE CURRENT\_

TIMESTAMP, the TIMESTAMP value will automatically be updated to the current

timestamp. One can set the default to 0 and still allow updates as appropriate with

the following clause:

TIMESTAMP DEFAULT 0 ON UPDATE CURRENT\_TIMESTAMP

This sets an initial timestamp of 0, but requests updating to the current timestamp.

To set the timestamp to the system time at initialization and then update it, we

combine the two statements:

TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

YEAR

As the name implies, the YEAR data type stores only year values. Whether the year is

treated as a two-digit or four-digit value depends on how the type is set.

Two-digit YEAR values

The two-digit YEAR value is set with the following syntax:

YEAR(2)

A two-digit YEAR type is obviously only valid for up to 99 years. So YEAR(2) is used

to specify years from 1970, the start of the aforementioned epoch, through 2069.

Four-digit YEAR values

Four-digit YEAR values are specified with the following:

YEAR(4)

These values naturally have a wider range than the two-digit form. Four-digit years

may range from 1901 through 2155.

[ 252 ]Chapter 9

Valid input

Obviously, two-digit YEAR values require two-digit input. Four-digit YEAR values

will accept two-digit input, as well. However, the consequent value may not be what

is intended. For example, let's assume we create a dummy table called yeareg using

the following statement:

CREATE TABLE yeareg(ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY, year

YEAR(4));

We can populate that table with four-digit values without a problem:

INSERT INTO yeareg(year) VALUES('1923');

Query OK, 1 row affected (0.00 sec)

But if we use only two digits, the result is quite different:

mysql> INSERT INTO yeareg(year) VALUES('23');

Query OK, 1 row affected (0.00 sec)

mysql> SELECT \* FROM yeareg;

+----+------+

| ID | year |

+----+------+

|1 | 1923 |

|2 | 2023 |

+----+------+

2 rows in set (0.00 sec)

Two-digit values are assumed to belong in the 21st century. The only exceptions are

values from 70 through 99 and the value 00. These are rendered automatically as the

years between 1970 and 1999 and, in the case of the last, the year 2000.

As with the other data types discussed in this chapter, MySQL does not guess if it is

given invalid data. Data outside the above ranges will be zeroed out as 00 or 0000,

depending on which size of YEAR value is expected.

TIME

The TIME data type is used for one of three purposes. It can represent:

•A time of day

•An elapsed time

•An interval of time

[ 253 ]Date and Time Values

Because of its multiple applications, TIME has a slightly different format and set of

constraints that might appear intuitive.

Format

The TIME type accepts input in multiple formats, but outputs data in only one. The

output format for TIME is:

HH:MM:SS

For input, however, one can input hours, minutes, and seconds in several formats:

•As a string in the form HHMMSS: So '234545' is valid.

•As an integer in the same form HHMMSS: So a numeric value 123456 is

also valid.

•As a string using colons for delimiters: This follows the output format

HH:MM:SS

TIME also accepts shorthand for extremely large or small quantities in terms of days

as well as fractions of a second. To specify a day, simply place the day value in front

of the regular time.

D HH:MM:SS

For large values, it converts the number of days into the respective number of hours

and adds it to the hour value. So:

1 23:59:59

is the same as:

47:49:49

and that is how MySQL records it in the database. Note, however, that using the

day value requires the use of the colon as a delimiter. Therefore, the following is

INVALID:

1235959

[ 254 ]Chapter 9

Fractions of a second, on the other hand, are dropped. The value is always, in effect,

rounded down. Therefore, the following are both equal in the eyes of MySQL:

1 12:01:01.1

1 12:01.01.9

Either way, the value that is stored is:

36:01:01

Invalid values

Invalid TIME values are zeroed out to 00:00:00. While it shows that a transaction

has occurred, zero values cannot be distinguished from midnight.

Valid values for each of the columns are as follows:

•Days: A whole number from 1 to 34

•Hours: A whole number between -838 and 838

•Minutes: A whole number less than or equal to 59

•Seconds: A number less than or equal to 59

•Fractions: Positive decimal value less than 1

For reasons of storage, the limits of MySQL's TIME type range from -838:59:59 to

838:59:59. Any value that does not make sense as a time within these constraints

will result in an erroneous value being stored. Precisely which value is stored will

depend on the error. Exceeding the limits of minutes and seconds results in a zeroed

value. Examples include:

5 23:01:61

20:61:23

18:75:75

However, if the hours and minutes are valid, but the number of hours represented

by the day and hour values exceeds 838, either negatively or positively, MySQL will

reduce that number to the respective terminus of either -838:59:59 or 838:59:59.

[ 255 ]Date and Time Values

If you want to ensure that MySQL handles your TIME values securely and

outputs any errors, use MySQL's STRICT\_ALL\_TABLES mode. STRICT\_

ALL\_TABLES returns an error and aborts any operation in which a

problem with the data arises. Note, however, that this can result in the

partial completion of multi-line operations and so can cause trouble.

Using STRICT\_ALL\_TABLES in development is a good way of checking

your codebase. Just remember to switch it off in production.

An alternative to STRICT\_ALL\_TABLES is STRICT\_TRANS\_TABLES.

This continues to process the data, but outputs a warning when it has

massaged the data to fit the column.

To use either mode, start MySQL using the executable, which is

appropriate for your platform and equate one of the above modes to the

flag --sql-mode. For example, one would use mysqld as follows:

mysqld --sql-mode="STRICT\_TRANS\_TABLES"

One can also specify the mode in the MySQL configuration file by setting

the value sql-mode (my.cnf for Unix-like systems and my.ini for

Windows). For more on MySQL modes, see the MySQL manual at:

http://dev.mysql.com/doc/refman/5.5/en/server-sql-

mode.html

If TIME is used to reference a time of day, possibly to receive the value of a function

like CURRENT\_TIME, the value must be within the 24 hour period.

Date and time types in Python

The date and time interfaces to MySQL for Python are very robust. As a

consequence, as long as the values you pass make sense within the above parameters

for formatting, neither MySQL nor Python will throw an error about aspects such as

the Python data type. For example, if we go back to the yeareg table, we can pass

integers and strings to it through Python without issue:

import MySQLdb

mydb = MySQLdb.connect('localhost', 'skipper', 'secret', 'datetime')

cursor = mydb.cursor()

x = 2012

statement = "INSERT INTO yeareg(year) VALUES('%s')" %(x)

results = cursor.execute(statement)

x = str(x)

statement = "INSERT INTO yeareg(year) VALUES('%s')" %(x)

results = cursor.execute(statement)

[ 256 ]Chapter 9

In the first statement, we pass a string. In the second, an integer. Yet both statements

will be executed by MySQL without issue.

mysql> SELECT \* FROM yeareg;

+----+------+

| ID | year |

+----+------+

|1 | 1923 |

|2 | 2023 |

| 3 | 2013 |

| 3 | 2013 |

+----+------+

4 rows in set (0.00 sec)

However, passing invalid values will result in an OperationalError being thrown.

Further, while this is possible, it is not advisable. Best practice holds that your

program should submit data in a format suited to its target type, so strings

should be formatted as strings and integers as integers.

Date and time functions

MySQL's time-related functions are used to manipulate date and time data types. As

such, they are very helpful tools for accessing data with optimum speed and minimal

overhead.

The MySQL list of date and time-related functions are legion. The full list is

contained in the table below (Source: MySQL 5.5 Reference Manual):

NameDescription

ADDDATE()Add time values (intervals) to a date value

ADDTIME()Add time

CONVERT\_TZ()Convert from one time zone to another

CURDATE()

CURRENT\_

DATE(),CURRENT\_DATE

CURRENT\_

TIME(),CURRENT\_TIME

CURRENT\_

TIMESTAMP(),CURRENT\_

TIMESTAMP

CURTIME()Return the current date

Synonyms for CURDATE()

Synonyms for CURTIME()

Synonyms for NOW()

Return the current time

[ 257 ]Date and Time Values

NameDescription

DATE\_ADD()Add time values (intervals) to a date value

DATE\_FORMAT()Format date as specified

DATE\_SUB()Subtract two dates

DATE()Extract the date part of a date or datetime expression

DATEDIFF()Subtract two dates

DAY()Synonym for DAYOFMONTH()

DAYNAME()Return the name of the weekday

DAYOFMONTH()Return the day of the month (0-31)

DAYOFWEEK()Return the weekday index of the argument

DAYOFYEAR()Return the day of the year (1-366)

EXTRACTExtract part of a date

FROM\_DAYS()Convert a day number to a date

FROM\_UNIXTIME()Format Unix timestamp as a date

GET\_FORMAT()Return a date format string

HOUR()Extract the hour

LAST\_DAYReturn the last day of the month for the argument

LOCALTIME(), LOCALTIME

LOCALTIMESTAMP,LOCALT

IMESTAMP()

MAKEDATE()Synonym for NOW()

MAKETIMEMAKETIME()

MICROSECOND()Return the microseconds from argument

MINUTE()Return the minute from the argument

MONTH()Return the month from the date passed

MONTHNAME()Return the name of the month

NOW()Return the current date and time

PERIOD\_ADD()Add a period to a year-month

PERIOD\_DIFF()Return the number of months between periods

QUARTER()Return the quarter from a date argument

SEC\_TO\_TIME()Converts seconds to HH:MM:SS format

SECOND()Return the second (0-59)

Synonym for NOW()

Create a date from the year and day of year

[ 258 ]Chapter 9

NameDescription

STR\_TO\_DATE()Convert a string to a date

SUBDATE()A synonym for DATE\_SUB() when invoked with three

arguments

SUBTIME()Subtract times

SYSDATE()Return the time at which the function executes

TIME\_FORMAT()Format as time

TIME\_TO\_SEC()Return the argument converted to seconds

TIME()Extract the time portion of the expression passed

TIMEDIFF()Subtract time

TIMESTAMP()With a single argument, this function returns the date or

datetime expression; with two arguments, the sum of the

arguments

TIMESTAMPADD()Add an interval to a datetime expression

TIMESTAMPDIFF()Subtract an interval from a datetime expression

TO\_DAYS()Return the date argument converted to days

TO\_SECONDS()Return the date or datetime argument converted to

seconds since Year 0

UNIX\_TIMESTAMP()Return a Unix timestamp

UTC\_DATE()Return the current UTC date

UTC\_TIME()Return the current UTC time

UTC\_TIMESTAMP()Return the current UTC date and time

WEEK()Return the week number

WEEKDAY()Return the weekday index

WEEKOFYEAR()Return the calendar week of the date (0-53)

YEAR()Return the year

YEARWEEK()Return the year and week

Obviously, it is not feasible to keep all of these in mind without constant use. This

table helps to give a primer on each, but all are discussed in great detail in the

MySQL manual. For this chapter, we will look at the most frequently used date

and time-related functions.

[ 259 ]Date and Time Values

For more on any of of the date and time functions, see the MySQL

manual: http://dev.mysql.com/doc/refman/5.5/en/date-and-

time-functions.html

NOW()

The NOW() function returns the current date and time from the system. Which

parts of the timestamp are used is determined by the data type of the column. For

example, assume three tables as follows:

dteg with a column dati of type DATETIME

dateeg with a column date of type DATE

yeareg with a column year of type YEAR(4)

timeeg with a column time of type TIME

We can use NOW() as a value in our INSERT statements. For each of the following

examples, the data like the respective values would be stored:

INSERT INTO dteg(dati) VALUES(NOW()); → 2010-01-14 11:33:51

INSERT INTO dateeg(date) VALUES(NOW()); → 2010-01-14

INSERT INTO yeareg(year) VALUES(NOW()); → 2010

INSERT INTO timeeg(time) VALUES(NOW()); → 11:33:51

CURDATE()

The current date is returned by CURDATE(). As the name implies, this function

returns the current date. If we assume the tables from the NOW() function (seen

previously), the following examples will store the relevant part of the current date in

the given column:

INSERT INTO dateeg(date) VALUES(CURDATE()); →

INSERT INTO yeareg(year) VALUES(CURDATE()); →

2010-01-14

2010

Depending on your MySQL configuration, you can use this function in statements

relative to TIME type values, but the stored value will be zeroed out to 00:00:00.

INSERT INTO timeeg(time) VALUES(CURDATE()); →

[ 260 ]

00:00:00Chapter 9

CURTIME()

The function CURTIME() returns the current time in the format HH:MM:SS. Usage of it

is as follows:

INSERT INTO timeeg(time) VALUES(CURTIME()); →

11:55:23

Obviously, the date is not included in the returned value, making this function

unsuitable for use where DATE, DATETIME, or YEAR values are expected. If a

CURTIME() returned value should be inserted where one of these three are expected,

the results are predictable. The returned value is never in the appropriate format for

DATE or DATETIME types, therefore MySQL stores a zeroed value.

INSERT INTO dteg(dati) VALUES(CURTIME()); → 0000-00-00 00:00:00

INSERT INTO dateeg(date) VALUES(CURTIME()); → 0000-00-00

However, in the case of YEAR, a more insidious error arises.

INSERT INTO yeareg(year) VALUES(CURTIME()); → 2011

The column is defined as YEAR(4). But MySQL will accept two digits for the value

and will convert it to four digits. Therefore, when it receives for YEAR(4) a value

that does not make sense, it accepts what data it can. In this case, the hour value of

11:33:51 is read simply as 11 and converted to 2011.

DATE()

The function DATE() strips the date out of a time-related string. The string is

expected to be in the format of DATE or DATETIME, but need not be from columns of

either of these types. One can simply feed a string to DATE() and receive the answer

back as a table:

mysql> select date('2009-12-02 12:14');

+-----------------------+

| date('2009-12-02 12') |

+-----------------------+

| 2009-12-02

|

+-----------------------+

1 row in set (0.00 sec)

[ 261 ]Date and Time Values

For DATE(), the only part that matters is the date aspect of the string. One can

include just about anything after the date, including text.

mysql> select date('2009-12-02 Wednesday');

+------------------------------+

| date('2009-12-02 Wednesday') |

+------------------------------+

| 2009-12-02

|

+------------------------------+

1 row in set, 1 warning (0.00 sec)

If there is a time or time-like datum after the date, it does not need to be

properly formatted.

mysql> select date('2009-12-02 11am');

+-------------------------+

| date('2009-12-02 11am') |

+-------------------------+

| 2009-12-02

|

+-------------------------+

1 row in set, 1 warning (0.00 sec)

The date, however, must come first in the string passed to DATE(). Otherwise,

MySQL returns NULL results.

DATE\_SUB() and DATE\_ADD()

The DATE\_ADD() and DATE\_SUB() functions are used to modify date values

according to specific intervals of time. DATE\_ADD() adds an interval of time.

DATE\_SUB() subtracts an interval of time. Their syntax is essentially the same:

DATE\_ADD(date, INTERVAL amount unit)

DA TE\_SUB(date, INTERVAL amount unit)

In each case, date is the value to be modified. The keyword INTERVAL is required to

introduce the amount and unit of time to be used.

[ 262 ]Chapter 9

In practice, these functions can be used as:

mysql> SELECT DATE\_ADD(now(), INTERVAL 20 minute);

+-------------------------------------+

| DATE\_ADD(now(), INTERVAL 20 minute) |

+-------------------------------------+

| 2010-01-14 13:04:48

|

+-------------------------------------+

1 row in set (0.00 sec)

And it can also be used as:

mysql> SELECT DATE\_SUB(now(), INTERVAL 20 minute);

+-------------------------------------+

| DATE\_SUB(now(), INTERVAL 20 minute) |

+-------------------------------------+

| 2010-01-14 12:25:22

|

+-------------------------------------+

1 row in set (0.00 sec)

The format and type of the returned value depends on the argument passed to

the function. If the date is a DATETIME or TIMESTAMP value or it is a DATE and the

INTERVAL value using hours, minutes, or seconds, then the returned value will be a

DATETIME value. Otherwise, these functions return a string.

An alternative to DATE\_SUB() is to increment the date value by a negative number.

So instead of the last example, we could have simply modified the first one

as follows:

mysql> SELECT DATE\_ADD(now(), INTERVAL -20 minute);

+--------------------------------------+

| DATE\_ADD(now(), INTERVAL -20 minute) |

+--------------------------------------+

| 2010-01-14 12:24:51

|

+--------------------------------------+

1 row in set (0.00 sec)

[ 263 ]Date and Time Values

Just as with DATE(), we can substitute a DATE or DATETIME formatted string for

the date.

mysql> SELECT '2009-12-24' + INTERVAL 1 DAY;

+-------------------------------+

| '2009-12-24' + INTERVAL 1 DAY |

+-------------------------------+

| 2009-12-25

|

+-------------------------------+

1 row in set (0.00 sec)

Additionally, both support shorthand ways of affecting the same end. Instead of

using DATE\_ADD() or DATE\_SUB(), we can simply use the plus sign + or the minus

sign - respectively. The syntax in these cases is:

date + INTERVAL amount unit

date - INTERVAL amount unit

For example, to add 20 days to today, we use:

mysql> SELECT NOW() + INTERVAL 20 DAY;

+-------------------------+

| NOW() + INTERVAL 20 DAY |

+-------------------------+

| 2010-02-03 12:50:13

|

+-------------------------+

1 row in set (0.00 sec)

and

mysql> SELECT INTERVAL 20 DAY + '2009-12-24';

+--------------------------------+

| INTERVAL 20 DAY + '2009-12-24' |

+--------------------------------+

| 2010-01-13

|

+--------------------------------+

1 row in set (0.01 sec)

[ 264 ]Chapter 9

Similarly, to subtract is as follows:

mysql> SELECT NOW() - INTERVAL 20 DAY;

+-------------------------+

| NOW() - INTERVAL 20 DAY |

+-------------------------+

| 2009-12-25 12:48:47

|

+-------------------------+

1 row in set (0.00 sec)

However, basic logic applies to the syntax here. When adding, we can juggle the two

addends without impacting on the sum. With subtraction, however, the INTERVAL

clause must come second because it makes no sense to subtract a date from an

interval of time.

Both functions accept several different units of time. However, unlike with the TIME

data type, mentioned previously, all amounts must relate precisely to the unit used.

One cannot, for example, pass day values with an hour unit. A complete listing of the

options and the expected quantity type for each is as follows:

For a unit of typeA valid Amount must be in

MICROSECONDMicroseconds

SECONDSeconds

MINUTEMinutes

HOURHours

DAYDays

WEEKWeeks

MONTHMonths

QUARTERQuarters

YEARYears

SECOND\_

MICROSECONDSeconds and microseconds with format 'seconds.

microseconds'

MINUTE\_

MICROSECONDMinutes, seconds, and microseconds with format 'MINUTES:

SECONDS.MICROSECONDS'

MINUTE\_SECONDMinutes and seconds with format 'MINUTES:SECONDS'

HOUR\_MICROSECONDThe variables of time, including microseconds, with format

'HOURS:MINUTES:SECONDS.MICROSECONDS'

HOUR\_SECONDThe variable of time with format 'HOURS:MINUTES:

SECONDS'

HOUR\_MINUTEThe basic time with format 'HOURS:MINUTES'

[ 265 ]Date and Time Values

For a unit of typeA valid Amount must be in

DAY\_MICROSECONDThe basic variables of time as well as number of days and

microseconds with format 'DAYS HOURS:MINUTES:

SECONDS.MICROSECONDS'

DAY\_SECONDThe same as day\_microsecond excluding microseconds with

format 'DAYS HOURS:MINUTES:SECONDS'

DAY\_MINUTEDays, hours, and minutes with format 'DAYS HOURS:

MINUTES'

DAY\_HOURDays and hours with format 'DAYS HOURS'

YEAR\_MONTHYears and months with format 'YEARS-MONTHS'

DATEDIFF()

The DATEDIFF() function is used to calculate the number of days between dates. It

requires two arguments, both in the DATE format:

DATEDIFF(first\_date, second\_date);

In practice, it can calculate just about any number of days.

mysql> SELECT DATEDIFF('2009-12-25', '2010-01-14');

+--------------------------------------+

| DATEDIFF('2009-12-25', '2010-01-14') |

+--------------------------------------+

|

-20 |

+--------------------------------------+

1 row in set (0.00 sec)

It also does not require explicit dates, but can accept output from other time-related

functions.

mysql> SELECT DATEDIFF('2001-09-11', DATE(NOW()));

+-------------------------------------+

| DATEDIFF('2001-09-11', NOW()) |

+-------------------------------------+

|

-3047 |

+-------------------------------------+

1 row in set (0.00 sec)

[ 266 ]Chapter 9

Any results passed to DATEDIFF() must be as DATE formatted values. If it is

unclear whether a value will be in the appropriate format, you can pass it

through DATE() first.

mysql> SELECT DATEDIFF('2001-09-11', DATE(NOW()));

+-------------------------------------+

| DATEDIFF('2001-09-11', DATE(NOW())) |

+-------------------------------------+

|

-3047 |

+-------------------------------------+

1 row in set (0.00 sec)

However, bad data will not be made good by DATE():

mysql> SELECT DATEDIFF('2001-09-11', DATE(CURTIME()));

+-----------------------------------------+

| DATEDIFF('2001-09-11', DATE(CURTIME())) |

+-----------------------------------------+

|

NULL |

+-----------------------------------------+

1 row in set, 1 warning (0.00 sec)

DATE\_FORMAT()

The DATE\_FORMAT() function is very helpful for creating human-friendly date and

time strings automatically. It requires two arguments—a date and the formatting

structure for the output. For example, the expression %W denotes the day of the week

for DATE\_FORMAT(). We can therefore learn what day of the week a particular date is

by using:

mysql> SELECT DATE\_FORMAT('2001-09-11', '%W');

+---------------------------------+

| DATE\_FORMAT('2001-09-11', '%W') |

+---------------------------------+

| Tuesday

|

+---------------------------------+

1 row in set (0.07 sec)

[ 267 ]Date and Time Values

The supported formats and their abbreviations are listed below. Note that they bear

many similarities to the Unix command date, but there are many differences.

SpecifierDescription

%aAbbreviated weekday name (Sun, Mon,and so on)

%b%DAbbreviated month name (Jan, Feb, and so on)

Numeric month value without a leading 0 in single-digit values (0, 1, 2, and

so on)

Day of the month with English suffix (0th, 1st, 2nd, 3rd, …)

%dNumeric day of the month with a leading 0 for single-digit values (00-31)

%eNumeric day of the month with no leading 0 for single-digit values (0-31)

%fMicroseconds (000000..999999)

%HHour based on a 24-hour day and with a leading 0 (00 through 23)

%hHour based on two 12-hour segments with a leading 0 (01, 02, 03, and so on.)

%ITwo-digit hour based on two 12-hour segments (01, 02, 03, and so on)

%iNumeric value of minutes with leading 0 (00, 01, 02, and so on)

%jThree-digit day of the year (001, 002, through to 366)

%kHour based on 24-hour day and with no leading 0 (0, 1, 2, and so on)

%lHour based on two 12-hour segments and with no leading 0 (1, 2, and so on)

%MFull name of the m (January, February, and so on)

%mNumeric value of the month with leading 0 (00, 01, 02, and so on)

%pAM or PM

%rTime based on two 12-hour segments (hh:mm:ss followed by AM or PM)

%SSeconds with leading 0 (00, 01, 02, and so on)

%sSeconds with leading 0 (00, 01, 02, and so on)

%TTime based on 24-hour clock (hh:mm:ss)

%UWeek (00 to 53), where Sunday is the first day of the week

%uWeek (00 to 53), where Monday is the first day of the week

%VWeek (01 to 53), where Sunday is the first day of the week; used with %X

%vWeek (01 to 53), where Monday is the first day of the week; used with %x

%WFull name of day (Sunday through Saturday)

%wNumeric value of the day of the week (0 [Sunday] through 6 [Saturday])

%XNumeric, four-digit value of year for the week where Sunday is the first day

of the week; used with %V

%c

[ 268 ]Chapter 9

SpecifierDescription

%xNumeric, four-digit value of year for the week where Monday is the first day

of the week; used with %v

%YNumeric, four-digit year

%yNumeric, two-digit year

%%The '%' character

%xx for any "x" not listed above

These sundry formatting specifiers can then be combined for customized date

formats. For example, a more human-friendly format of the aforementioned BBC

report on the day the Berlin Wall fell (November 10, 1989 at 9 o'clock in the evening)

might be formatted as follows:

mysql> SELECT DATE\_FORMAT('1989-10-11 21:37:09:04', '%W %d %M %Y at %h:%m

%p and %s seconds');

+------------------------------------------------------------------------

---------+

| DATE\_FORMAT('1989-10-11 21:37:09:04', '%W %d %M %Y at %h:%m %p and %s

seconds') |

+------------------------------------------------------------------------

---------+

| Wednesday 11 October 1989 at 09:10 PM and 09 seconds

|

+------------------------------------------------------------------------

---------+

1 row in set, 1 warning (0.00 sec)

EXTRACT()

The EXTRACT() function returns a designated part from a date string. It requires only

the unit needed and the date from which to draw the data.

mysql> SELECT EXTRACT(SECOND FROM '1989-10-11 21:00:09');

+--------------------------------------------+

| EXTRACT(SECOND FROM '1989-10-11 21:00:09') |

+--------------------------------------------+

|

9 |

+--------------------------------------------+

1 row in set (0.00 sec)

It works with the same units of time as DATE\_ADD() and DATE\_SUB().

[ 269 ]Date and Time Values

TIME()

Like DATE(), the TIME() function is used to extract the time (as opposed to

date) value from a DATE or DATETIME string. Where DATE() expects the relevant

information to be first in the string, TIME() looks to the next set of data in the string.

It accepts one argument—the string to be evaluated.

mysql> SELECT TIME('1989-10-11 21:00:09');

+-----------------------------+

| TIME('1989-10-11 21:00:09') |

+-----------------------------+

| 21:00:09

|

+-----------------------------+

1 row in set (0.00 sec)

It is further worth noting that the format of the time value is expected to be

HH:MM:SS. If more information follows the time, even without a space delimiter,

it is ignored.

mysql> SELECT TIME('1989-10-11 21:00:09abcdef');

+-----------------------------------+

| TIME('1989-10-11 21:00:09abcdef') |

+-----------------------------------+

| 21:00:09

|

+-----------------------------------+

1 row in set, 1 warning (0.00 sec)

Project: Logging user activity

Between July 2005 and January 2007, a major international clothing retailer suffered a

data breach that amounted to the theft of 45.6 million credit card numbers. TJX is the

parent company of stores such as TJMaxx, Bob's Stores, Marshalls, HomeGoods, and

AJ Wright stores in the United States, as well as Winners and HomeSense stores in

Canada, and TKMaxx stores across Europe. The credit card information from most,

if not all, of these stores was electronically burgled over an undetermined amount

of time. After the breach was discovered, TJX spokeswoman Sherry Lang went on

record as saying: "There is a lot of information we don't know, and may never be able to know,

which is why this investigation has been so laborious" (Sources: AP, CNET).

[ 270 ]Chapter 9

Computer forensic examinations rely extensively on computer logs. Without

sufficient logging, as seen in the case of TJX, one cannot tell where things went

wrong if catastrophe strikes. While MySQL keeps its own logs, the fact that these are

known makes them a target for cleansing or deletion by anyone who would break

into the system. In security terms, homogeneity makes breaking into a system much

easier. Therefore, it is worth it to make one's system sufficiently different from out-

of-the-box solutions, in order to render it difficult to break into.

Note that while heterogeneity goes a long way to assuring the security

of a system, a balance must be struck. One must be careful not to

make the layout so complex or wildly unique such that it is difficult to

administer either by the system administrator or by the other support

staff, including new-hires.

Aside from such extreme circumstances, logging is also very helpful for debugging

and for learning about your users' needs. Studying web logs indicates the browsing

habits of users from entry to exit. Therefore, database logging can ensure that you

know which data is being accessed most frequently and in what way.

By default, MySQL keeps its logging feature switched off. To turn it on, edit

the my.cnf file (or my.ini on Windows). Under the section entitled Logging and

Replication, you will find a number of logs that can be switched on or off at runtime.

By default, these logs record a bevy of information that can be quite useful.

Particularly detailed is the general query log. However, as the comment in the

MySQL 5 default configuration reads:

# Be aware that this log type is a performance killer.

Information on each log type can be found in the MySQL manual at:

http://dev.mysql.com/doc/refman/5.5/en/server-logs.html

Suffice it to say, however, that MySQL logs can be helpful in their detail. However,

that detail can be costly in server performance. For this reason, many MySQL

installations do not use the general log, but only the error log. This records

debugging information for the performance of MySQL—when it was started, when

it stopped, and what, if any, errors caused the server to die. It is not oriented toward

security or usage patterns per se.

[ 271 ]Date and Time Values

It is therefore helpful to maintain one's own customized log. In this project, we

will revise the project from the previous chapter to introduce such logging. The

emendations will follow this specification:

•Log all MySQL activity sent through MySQLdb except its own

•The log will include the username and statement submitted for processing

•If the expected database and table are not available, they will be created

•All logging will occur before any MySQL statement is submitted

•If the logging is unsuccessful, the MySQL statement will not be processed

•The log infrastructure should be created by the program, not by a MySQL

script or command-line

The log framework

For reasons of simplicity, the log for this project will be another MySQL database. In

real-world deployment, you would want to store your log files off the local server

for security reasons. Using the flexible connect() method of MySQL for Python, one

can affect such a change of host with relative ease.

The database that we use is called logdb. Obviously, you can name yours as you like,

as long as you are consistent in using that name.

Within logdb, the table for logging activity will be called entry\_log. Its architecture

is as follows:

mysql> describe entry\_log;

+-------------+--------------+------+-----+-------------------+----------

------+

| Field

|

| Type

| Null | Key | Default

| Extra

+-------------+--------------+------+-----+-------------------+----------

------+

| transaction | int(11)

increment || NO| PRI | NULL| auto\_

| username

|| varchar(30)| NO|| NULL|

| query

|| varchar(256) | NO|| NULL|

| qtime

|| timestamp|| CURRENT\_TIMESTAMP |

| NO

+-------------+--------------+------+-----+-------------------+----------

------+

[ 272 ]Chapter 9

As the table shows, every transaction has its own identifier. In addition to the

username and query, every entry will also be given a timestamp of when it

was performed.

To affect this table, we will use the following CREATE statement as a basis:

CREATE TABLE entry\_log(transaction INT NOT NULL AUTO\_INCREMENT PRIMARY

KEY, username VARCHAR(30) NOT NULL, query VARCHAR(256) NOT NULL, qtime

TIMESTAMP);

However, we will not do this from within MySQL. The specification reads that we

should not have to set up tables and databases in MySQL itself. Rather we will let

Python and MySQLdb do the dirty work for us.

The logger() function

To affect the new logging functionality, we will create a separate function called

logger(). It will accept as an argument the statement to be passed to MySQL and

will also use the username submitted with the program.

For the sake of security, we will not allow any of the login details to be changed.

They will be hardcoded into the connect() dialogue. So the first lines of the function

will look like this:

def logger(statement):

"""Logs each transaction in a MySQL database."""

mydb = MySQLdb.connect('localhost', 'loguser', 'secretphrase')

cursor = mydb.cursor()

Note that we do not indicate the database we will use. This is because the login could

fail if we specify a database that does not exist. Therefore, we need to ensure that the

database exists before we try to use it.

Creating the database

Having created the cursor, we now need to ensure the database is available. For this,

we will rely on the IF NOT EXISTS clause:

createdb = "CREATE DATABASE IF NOT EXISTS logdb"

As discussed in earlier chapters, this ensures that the command will not fail. For

reasons of debugging, we then pass the results of cursor.execute() to a variable.

resdb = cursor.execute(createdb)

[ 273 ]Date and Time Values

This ensures that we can learn the results of this execution if we need to. If there is

any question of whether the database will be created, we can test the value of resdb

and respond accordingly.

Using the database

Before we can issue any statements pertaining to the database or table, we need to

tell MySQL that we will use the logdb database.

usedb = "USE logdb"

resuse = cursor.execute(usedb)

Again, by storing the results of each execute() call, we are able to test the success of

the process.

Creating the table

With the database created, we next need to ensure that the table is available. We will

use the preceding statement, but again include the IF NOT EXISTS clause:

createtb = "CREATE TABLE IF NOT EXISTS entry\_log(transaction INT

UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY, username VARCHAR(30) NOT

NULL, query VARCHAR(256) NOT NULL, qtime TIMESTAMP)"

restb = cursor.execute(createtb)

The result is a flexible way of ensuring that the database logdb and the table

entry\_log are available to us.

As this function will log all activity, we can test the values one-by-one if something

is awry. For example, if no queries are being processed and no log entries made, we

are not guaranteed that Python or MySQL will throw an error. However, we can test

each of the resdb, resuse, and restb values and print out messages upon success.

That will allow us to see how far the program goes before exiting, thus indicating the

point of error.

Forming the INSERT statement

Having the database infrastructure in place, we can then form and execute an

INSERT statement to log the activity. As previously indicated, logger() will take the

statement value and submit it. We will also use the global value of the username. In

a CGI implementation, this latter value may be user. In PHP, it will be referenced as

opt.user as part of the options parsed by optparse.

[ 274 ]Chapter 9

An important issue when passing a MySQL statement as part of another MySQL

statement is the issue of quotation marks—whether single or double. Therefore, we

need to message the value of statement before we submit it as part of the INSERT

statement for logdb.

statement = statement.replace("'", "\\'")

statement = statement.replace('"', '\\"')

statement = "'" + statement + "'"

user = "'" + opt.user + "'"

Here we use Python's ability to handle arguments in either single or double quotes to

replace—first the single quotes and then the double in the value statement. In order

to ensure the quotes are properly escaped in the ensuing MySQL statement, we must

escape the escape character (\) by reduplicating it.

Then we sandwich each of the statement and opt.user (or user) values between

quotes and assign them to statement and user, respectively. This is in preparation

for the VALUES field in the INSERT statement.

Finally, we form the statement to be executed:

entry = "INSERT INTO entry\_log(username, query) VALUES(%s, %s)"

%(user, statement)

And later we pass it to execute()

reslog = cursor.execute(entry)

As we will see in the next section, the results saved to reslog are important for the

successful deployment of the log functionality. For now, however, we will simply

return that value as the end product of the function.

return reslog

Ensure logging occurs

As mentioned in the specification for this project, the log must record all activity

passed to MySQL. The natural implication of this requirement is to modify the

execute() function to call logger() and not to process any statement until

logger() records it successfully.

First we pass the statement received by execute() to logger() as the first thing to

be done in execute():

def execute(statement, cursor, type):

"""Attempts execution of the statement."""

reslog = logger(statement)

[ 275 ]Date and Time Values

Next, we test the value of reslog. Recall that every successful INSERT statement

returns a value of 1 if successful. So we can affect the test with a switch

called success.

if reslog == 1:

success = 1

else:

success = 0

This if...else structure determines whether to turn success on or off. As usual,

True and False can be used in lieu of 1 and 0, respectively.

For the effects of the switch to be implemented, we then need to change the while

statement on which execute() operates. Currently, it reads as:

while True:

We then need to change it to read:

while success == 1:

Alternatively, we could just use one of the following conditionals:

while success

Or, we can use:

while reslog > 0

From this point, execute() will only process statements after logger() has

recorded them. As we hardcoded everything in logger() that we could, this

renders it much more secure than a dynamic log function.

Room to grow

While this program works, there are more than a few ways to improve it. Obviously,

before you use it in the wild, you should change the host used by connect() in

logger(). Logs should never be in the same place as the activity they record.

Other points for improvement that you might consider trying out are:

•Instead of writing to a database, write the log to a file and post it to a

remote machine

•Record the user's IP address by searching the web server's logs

•Keep separate logs for each user

•Record whether a query successfully returned any hits or was erroneous

[ 276 ]Chapter 9

•

Record unsuccessful logins, sending an email to the database administrator if

too many are received in the name of the same user within a certain amount

of time

A lot of information will be logged by the likes of the web server, MySQL itself,

and other logging applications in the stack. However, this program allows you to

customize what information you log, combining it with other information that the

others may not record.

Summary

In this chapter, we have covered how to log user activity with MySQL for Python.

We have seen:

•The several data types for date and time that MySQL supports

•When different data types should be used and with what format and range

•The date and time functions that can be accessed through MySQL for Python

•Frequently used functions for handling matters of date and time

In the next chapter, we will look at using aggregate functions in MySQL for Python.

[ 277 ]Aggreg