MySQL\_for\_Python\_Albert\_c12

ring Functions

We have already seen how we can pass the burden of certain processes onto the

database server. The MySQL aggregate functions are optimized for their tasks. This

reduces the amount of I/O passed back to Python and results in less data for our

programs to process. In addition to the aggregate functions, MySQL also provides

other functions that help us process data before it is returned to Python.

In this chapter, we will see:

•The way MySQL allows us to combine strings and return the single,

resulting value

•How to extract part of a string or the location of a part, thus saving

on processing

•How to convert cases of results

All of this can be done through MySQL for Python. After we have seen the common

ways to juggle strings in MySQL, we will then apply it in this chapter's project.

Preparing results before their return

MySQL offers a bevy of functions for preparing results before returning them. Here

we will look at the more common ones. Before we begin, it is worth noting that all

MySQL functions are limited by the size of the max\_allowed\_packet variable. If the

value(s) to be returned exceed the maximum allowed packet size, MySQL will return

a NULL value.

For information on fine-tuning server variables such as max\_allowed\_packet, see:

http://dev.mysql.com/doc/refman/5.5/en/server-parameters.htmlString Functions

CONCAT() function

The CONCAT() function allows us to concatenate, or join, two or more values. The

basic syntax is as follows:

SELECT CONCAT(value1, value2);

The values can be either string or numeric values:

SELECT CONCAT(22, '/', 7);

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

| CONCAT(22, '/', 7) |

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

| 22/7

|

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

SELECT CONCAT('pi = ', 22, '/', 7);

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

| CONCAT('pi = ', 22, '/', 7) |

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

| pi = 22/7

|

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

However, as in Python, a string value passed without quotes causes MySQL to throw

an error:

SELECT CONCAT(pi, ' = ', 3, '.', 14156);

ERROR 1054 (42S22): Unknown column 'pi' in 'field list'

It may seem strange that MySQL should complain about a 'field list', but that is

exactly what it sees when it looks at CONCAT(). Further, one is able to use field names

in the argument of CONCAT() to affect calculations or formatting. Using the sakila

database, for example, we can return the length of each title in terms of hours

as follows:

SELECT title, CONCAT(length/60, ' hours') FROM film;

If we append WHERE title LIKE 'WAR%' after film, we get the following results:

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

| title

| CONCAT(length/60, ' hours') |

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

| WAR NOTTING

| 1.3333 hours

|

| WARDROBE PHANTOM | 2.9667 hours

|

| WARLOCK WEREWOLF | 1.3833 hours

|

| WARS PLUTO

| 2.1333 hours

|

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

[ 342 ]Chapter 12

As usual, we can use AS to clean up the results if we want:

SELECT title AS Title, CONCAT(length/60, ' hours') AS

Length FROM film WHERE title LIKE 'WAR%';

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

| Title

| Length

|

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

| WAR NOTTING

| 1.3333 hours |

...

SUBSTRING() or MID()

The function calls SUBSTRING() and MID() are synonymous. Which one you choose

is a matter of style. Their purpose is to allow you to extract a substring, or the

midsection, of a value within the bounds of certain index points. The syntax is

as follows:

SELECT SUBSTRING(value, position, length);

SELECT MID(value, position, length);

The value must adhere to the usual rules—quotes for strings. The first index point is

the beginning point within the value and is a required argument. The second index

value is optional. Its absence causes the return of the string from the initial index

point to the end of the value. For a string, "I'm afraid not", we can get everything

from the fifth position onward:

SELECT SUBSTRING("I'm afraid not", 5);

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

| SUBSTRING("I'm afraid not", 5) |

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

| afraid not

|

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

While the second index point is optional, it has a very different meaning in MySQL

from Python when it is used. Contrary to its meaning when specifying a range,

say, in xrange() and similar Python functions, the second index point is not an

absolute index in the word, but it is the number of positions forward from the

first index point.

[ 343 ]String Functions

An example for extracting the middle word from the string, "I'm afraid not is":

select mid("I'm afraid not", 4,8);

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

| mid("I'm afraid not", 4,8) |

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

| afraid

|

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

Finally, if we don't want to specify a precise range within the value, we don't have to.

MySQL's SUBSTRING() and MID() functions allow for counting backward from the

end of the value. Just use a negative number for the number of places from the end,

from where you want MySQL to count.

SELECT SUBSTRING("I'm afraid not",-3);

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

| SUBSTRING("I'm afraid not",-3) |

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

| not

|

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

TRIM()

The TRIM() function performs the same job as Python's built-in strip() function: it

strips leading and trailing whitespaces from results. However, TRIM() is also more

flexible than strip() in that it can be customized to strip a specified value(s) from

the beginning of a value, the end of a value, or both.

Basic syntax

The basic syntax is:

SELECT TRIM(<some value>);

It looks like this in real life:

SELECT TRIM(' Barstow

') AS spot;

[ 344 ]Chapter 12

In order to show the stripping of the string, the results column is given a smaller

title than the string itself.

+---------+

| spot

|

+---------+

| Barstow |

+---------+

TRIM() can be applied to numeric values as well:

SELECT trim( 1234567890

) AS numbers;

This results in the following output:

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

| numbers

|

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

| 1234567890 |

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

Options

In addition to this basic syntax, TRIM() also supports the option of specifying

whether the value to be removed is at the beginning or end of the value, or both. To

do this, we specify either LEADING, TRAILING, or BOTH in the argument. We also must

specify then precisely what we want trimmed as shown in the following session:

mysql> SELECT TRIM(LEADING ' ' FROM '

Barstow') AS spot;

Barstow') AS spot;

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

| spot

|

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

| Barstow

|

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

1 row in set (0.00 sec)

mysql> SELECT TRIM(TRAILING ' ' FROM '

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

| spot

|

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

|

Barstow |

[ 345 ]String Functions

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

1 row in set (0.00 sec)

mysql> SELECT TRIM(BOTH ' ' FROM '

Barstow

') AS spot;

+---------+

| spot

|

+---------+

| Barstow |

+---------+

1 row in set (0.00 sec)

Obviously, the default for TRIM() is to strip whitespace from both sides. However, as

you might deduce from the previous examples, we can also specify precise values to

pull instead of whitespaces.

mysql> SELECT TRIM(LEADING 'B' FROM 'Barstow') AS spot;

+--------+

| spot |

+--------+

| arstow |

+--------+

1 row in set (0.00 sec)

Note that we removed the whitespace from the argument. If we hadn't, the stripping

would not have done what we wanted. For example, try the following statement on

your own:

SELECT TRIM(LEADING 'B' FROM '

Barstow

Alternatives

') as spot;

There are also two variants to TRIM() that offer a shorthand for the LEADING and

TRAILING options. Instead of passing those words with their necessary additional

syntax, we can strip whitespace from the left of the value with LTRIM(), and from

the right with RTRIM(), as follows:

mysql> SELECT LTRIM('

Barstow

') as spot;

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

| spot

|

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

| Barstow

|

[ 346 ]Chapter 12

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

1 row in set (0.00 sec)

mysql> SELECT RTRIM('

Barstow

') as spot;

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

| spot

|

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

|

Barstow |

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

1 row in set (0.00 sec)

REPLACE()

The REPLACE() substitutes one value for another within a given stream of text,

whether string or numeric. For string values, the operation is case-sensitive. Unlike

Python's substitution functions, it does not work with regular expressions

(as of MySQL 5.5). The basic syntax is:

SELECT REPLACE(base value, value to be replaced, replacement value);

A string-based example is:

mysql> SELECT REPLACE("I'm afraid not", 'fraid ', ' frayed k') as spot;

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

| spot

|

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

| I'm a frayed knot |

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

1 row in set (0.00 sec)

But it also works with numeric values because the result is first converted

to a string:

mysql> SELECT REPLACE(22/7, 3, 2) as num;

+--------+

| num |

+--------+

| 2.1429 |

+--------+

1 row in set (0.00 sec)

[ 347 ]String Functions

In this case, quotes are optional:

mysql> SELECT REPLACE(10/3, '3', '2') as spot;

+--------+

| spot |

+--------+

| 2.2222 |

+--------+

1 row in set (0.00 sec)

Note that the replacement is indiscriminate—all instances of the first value are

changed to the second. One must consequently be careful in how REPLACE() is

applied. REPLACE() is multibyte safe.

INSERT()

INSERT() functions similarly to REPLACE() in that it injects a substring into a value.

Whether it overwrites part or all of that value is determined by the arguments you

pass to it. The basic syntax is as follows:

SELECT INSERT(base value, position, length, string to be inserted);

An example is:

mysql> SELECT INSERT("I'm afraid", 4, 0, ' not ') as clause;

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

| clause

|

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

| I'm not afraid |

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

1 row in set (0.00 sec)

This inserts the word not surrounded by two blank spaces just after the "m" in

I'm. As the length of the insertion is 0 characters, no part of the original value is

overwritten. However, because of that, the resulting value has two spaces between

"not" and "afraid" instead of just one. We could either truncate the value to be

inserted by one space or we can overwrite the space before "afraid".

[ 348 ]Chapter 12

To truncate "not", we would use TRIM:

mysql> SELECT INSERT("I'm afraid", 4, 0, RTRIM(' not ')) as clause;

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

| clause

|

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

| I'm not afraid |

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

1 row in set (0.00 sec)

But a less obtuse way of affecting the same results is to increase the length by one:

mysql> SELECT INSERT("I'm afraid", 4, 1, ' not ') as clause;

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

| clause

|

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

| I'm not afraid |

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

1 row in set (0.00 sec)

INSERT() works well with multi-byte characters as either the base value or the

insertion string. If the position exceeds the length of the base value, the base value

itself is returned:

mysql> SELECT INSERT("I'm afraid", 23, 1, ' not ') as clause;

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

| clause

|

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

| I'm afraid |

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

1 row in set (0.00 sec)

[ 349 ]String Functions

If the length of the insertion exceeds the length of the original base value but the

position of insertion is valid, the rest of the base value is overwritten:

mysql> SELECT INSERT("I'm afraid", 4, 23, ' not ') as clause;

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

| clause |

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

| I'm not |

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

1 row in set (0.00 sec)

All four of the arguments for INSERT() are required, and an error is thrown if any of

them is missing.

REGEXP

Technically speaking, REGEXP does not appear as a function. However, it is included

here because it operates on strings, such as SUBSTRING() and because it is listed

among the string functions in the MySQL documentation. The syntax for the REGEXP

phrase is:

value REGEXP pattern

REGEXP is more a keyword token to distinguish a part of the SELECT statement as

being special, as being a regular expression. MySQL evaluates regular expressions

on the fly, compiling them as needed, instead of compiling them as one usually does

in Python.

If the pattern match is a success, REGEXP causes MySQL to return 1; otherwise, it

returns 0. Here a carat (^) is used to test the beginning of the value.

mysql> SELECT 'Barstow' REGEXP '^B';

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

| 'Barstow' REGEXP '^B' |

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

|

1 |

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

1 row in set (0.00 sec)

mysql> SELECT 'Barstow' REGEXP '^C';

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

| 'Barstow' REGEXP '^C' |

[ 350 ]Chapter 12

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

|

0 |

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

1 row in set (0.00 sec)

The nature of the value against which the pattern is checked can be either a precise

value or a representation of search results (for example, column name). Therefore,

REGEXP can appear in the column specification or in other clauses like WHERE.

Consider the following example using sakila:

mysql> SELECT title FROM film WHERE title REGEXP '^TA.{3}K';

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

| title

|

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

| TAXI KICK |

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

1 row in set (0.00 sec)

The carat matches the beginning of the value to be assessed similar to the way it

matches the beginning of a line in Python. The dot after TA matches any single

character. The {3} is a complex way of rendering ...—that is, three characters in

succession. As it turns out, there is only one title in film that begins with TA and that

has three characters before K.

A complete list of MySQL's regular expression meta-characters follow:

Meta-characterMeaning

.Match any character

?Match zero or one

\*Match zero or more

+Match one or more

{n}Match n times

{m,n}Match m through n times

{n,}Match n or more times

^Beginning of line

$End of line

[[:<:]]Match beginning of words

[[:>:]]Match ending of words

[ 351 ]String Functions

Meta-characterMeaning

[:class:]Match a character class

[abc]Match one of enclosed chars

[^xyz]Match any character not enclosed

|Separates alternatives

Most of these will be familiar from working with Python's regular expressions.

However, character classes are foreign to Python. They are part of MySQL's support

for the POSIX standard.

POSIX stands for Portable Operating System Interface for

Unix, a standard created by the IEEE as IEEE 1003 and adopted by the

International Standards Organisation as ISO/IEC 9945. It initially defined

a common Application Programming Interface (API) for use across

Unix-like operating systems. But, the platforms that are at least partially

compliant with it have since been extended to include Microsoft Windows

among others.

A character class is essentially shorthand for a large number of characters. Instead of

writing every letter of the alphabet like this:

mysql> SELECT 'Barstow' REGEXP '[abcdefghijklmnopqrstuvwxyz]';

We can use [:alpha:] to represent every alphabetic character and get the

same results:

mysql> SELECT 'Barstow' REGEXP '[:alpha:]';

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

| 'Barstow' REGEXP '[:alpha:]' |

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

|

1 |

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

1 row in set (0.00 sec)

[ 352 ]Chapter 12

Character classes can include every alphanumeric character, punctuation, or several

others. A complete list of POSIX character classes are listed with their definition and

ASCII regex equivalent as follows:

POSIXDescriptionASCII

[:alnum:]Alphanumeric characters[a-zA-Z0-9]

[:alpha:]Alphabetic characters[a-zA-Z]

[:ascii:]ASCII characters[\x00-\x7F]

[:blank:]Space and tab[ \t]

[:cntrl:]Control characters[\x00-\x1F\x7F]

[:digit:]Digits[0-9]

[:graph:]Visible characters[\x21-\x7E]

[:lower:]Lowercase letters[a-z]

[:print:]Visible characters and spaces[\x20-\x7E]

[:punct:]Punctuation and symbols[!"#$%&'()\*+,\-

./:;<=>?@[\\\]^\_`{|}~]

[:space:]All whitespace characters, including

line breaks[ \t\r\n\v\f]

[:upper:]Uppercase letters[A-Z]

[:word:]Word characters[A-Za-z0-9\_]

[:xdigit:]Hexadecimal digits[A-Fa-f0-9]

[ 353 ]String Functions

Accessing and using index data

Here you will see how to calcualte the length of a value with LENGTH() function.

LENGTH()

Like len() in Python, the LENGTH() function of MySQL returns the bitwise length of

the value that is passed to it. Where multi-byte characters are used in the argument,

multiple byte values are tabulated. Therefore, in sakila, we can retrieve the title

and length of the title for each film. Here we use a WHERE clause to limit the results

for the sake of space.

SELECT title as title, LENGTH(title) as title\_length FROM film WHERE

title LIKE 'TA%E';

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

| title

| title\_length |

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

| TALENTED HOMICIDE |17 |

| TARZAN VIDEOTAPE |16 |

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

The LENGTH() function can similarly be used in the argument of other functions. For

example, if we wanted to extract the approximate middle of a value, we could divide

the length by four and cull out everything from the one fourth point, the length

divided by four, to the three fourth point, the length divided by four multiplied by 2

added to the value of the one fourth point. Remember that the second index value is

not to a point in the value, but is added to the initial index value in order to calculate

the slice. Therefore, rather than multiply by 3, we multiply it by 2 and use that value

as the second index value.

SELECT SUBSTRING("I'm afraid not", LENGTH("I'm afraid not")/4,

LENGTH("I'm afraid not")/4\*2) AS length;

The length of the value is 14 characters. If we divide that by 4, we get 3.5 (4 when

rounded). This is our first index value. If we double that value, we get 7. Seven

characters from the 4 is the end of the word "afraid". Therefore, the results are

as follows:

+---------+

| length |

+---------+

| afraid |

+---------+

[ 354 ]Chapter 12

An easier way to affect this selection is to define a user-defined variable name. It is

quite complicated to read.

By using the variable name @string, we can assign the string value and

not have to retype it.

SET @string = "I'm afraid not"; SELECT SUBSTRING(@

string, LENGTH(@string)/4, LENGTH(@string)/4\*2) AS

length;

If we need to do this repeatedly, we could simply insert the value of @

string into a variable assignment for statement as follows:

statement1 = "SET @string = "%s"" %("I'm afraid

not")

We then define the SELECT statement in Python accordingly:

statement2 = "SELECT SUBSTRING(@string, LENGTH(@

string)/4, LENGTH(@string)/4\*2) AS length"

We do not need to define @string if we execute both statements through

the same connection. So after creating a connection and our cursor within

a try...except structure, we can then execute the two statements in

succession:

cursor.execute(statement1)

cursor.execute(statement2)

We can then follow the execution of statement2 with a fetchall()

call and subsequent processing:

results = cursor.fetchall()

for item in results:

print item[0]

The results:

afraid

We can retrieve the index location of given strings using the INSTR() or

LOCATE() functions.

INSTR() or LOCATE()

Both INSTR() and LOCATE() serve to return the beginning index location of a given

string. The main difference between them is the order in which the arguments are

given. However, LOCATE() also supports more options.

[ 355 ]String Functions

INSTR()

INSTR() accepts the base string first followed by the pattern to be found:

SELECT INSTR(base string, pattern);

It then returns the numerical index where the first instance of the pattern begins. To

illustrate, consider the following two examples:

mysql> SELECT INSTR('Can you find a bar in Barstow?', 'Bar') as results;

+-------+

| results |

+-------+

| 16 |

+-------+

1 row in set (0.00 sec)

mysql> SELECT INSTR('Can you find a bar in Barstow?', 'Bars') as results;

+-------+

| results |

+-------+

| 23 |

+-------+

1 row in set (0.00 sec)

In the first instance, we search for Bar and find it first at point 16 in the string.

However, in the second instance, we look for Bars, thus avoid the word bar,

and find the beginning of Barstow, instead.

You will notice that the match is not case-sensitive. Therefore, if you want

case-sensitive matching, you will need to pull the results into Python. MySQL

only performs case-sensitive matching when one of the strings is binary.

LOCATE()

As previously mentioned, LOCATE() supports a simpler syntax, similar to INSTR(),

and a more complex syntax. To function similarly to INSTR(), we simply reverse the

arguments and pass it to LOCATE():

mysql> SELECT LOCATE('bar', 'Can you find a bar in Barstow?') as results;

+---------+

| results |

+---------+

|

16 |

+---------+

1 row in set (0.00 sec)

[ 356 ]Chapter 12

This simpler syntax is also echoed in a synonymous operation called

POSITION(). That function requires syntax exemplified as follows:

mysql> SELECT POSITION('bar' in 'Can you find a bar in

Barstow?') as results;

+---------+

| results |

+---------+

|

16 |

+---------+

1 row in set (0.00 sec)

As before, MySQL gives us the initial index of the first occurrence of the string.

However, what if you want the second occurrence of the string? LOCATE() allows

you to specify the beginning point of the search by stating the index point after the

base string.

mysql> SELECT LOCATE('bar', 'Can you find a bar in Barstow?', 16) as

results;

+---------+

| results |

+---------+

|

16 |

+---------+

1 row in set (0.00 sec)

As with INSTR(), LOCATE() is safe for use with multi-byte characters and is only

case-sensitive when one of the arguments is a binary string.

Nuancing data

In addition to performing indexing and substituting on data, MySQL also allows for

several ways of massaging the data to suit your needs. Some of the more common

ones are discussed in this chapter.

ROUND()

As the name suggests, the mathematical function ROUND() serves to round decimal

values to a specified number of places. The base value comes first in the syntax:

SELECT ROUND(base value, number of positions);

[ 357 ]String Functions

So rounding an already rough approximation of Pi would look like this:

mysql> SELECT ROUND(22/7, 2) as PI;

+------+

| PI |

+------+

| 3.14 |

+------+

1 row in set (0.00 sec)

The ROUND() function will accept whatever value you give it for the number of

positions. However, if the number of places exceeds MySQL's built-in abilities to

calculate a value, the extra places will be filled with zeroes:

mysql> SELECT ROUND(22/7, 20) as PI;

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

| PI

|

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

| 3.14285714200000000000 |

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

1 row in set (0.00 sec)

ROUND() operates on numerical values only. In an effort to fail gracefully, it will

return all zero values and a warning if you pass it a string:

mysql> SELECT ROUND('cat', 2) as PI;

+------+

| PI |

+------+

| 0.00 |

+------+

1 row in set, 1 warning (0.00 sec)

This obviously cuts against the grain of the Zen of Python's "Errors should never

pass silently", so one must be wary of it.

[ 358 ]Chapter 12

FORMAT()

The "format" feature functions similarly to ROUND() in that it allows you to specify

the number of decimal places for the results. It differs in that it will make the output

of the statement more human-friendly by adding punctuation for the value. For

example, division of large numbers frequently results in four or more digits to the

left of the decimal point. ROUND() treats them as follows:

mysql> SELECT ROUND(10000/3, 5) AS result;

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

| result

|

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

| 3333.33333 |

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

1 row in set (0.00 sec)

But FORMAT() makes the results much easier to read:

mysql> SELECT FORMAT(10000/3, 2) AS result;

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

| result |

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

| 3,333.33 |

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

1 row in set (0.00 sec)

Additionally, FORMAT() supports multiple locales. If your MySQL installation

allows for more than your default locale, you can specify the format you require by

including the locale as a third argument to the function.

SELECT FORMAT(base value, number of decimal places, locale);

You can find more on FORMAT() in the MySQL documentation:

http://dev.mysql.com/doc/refman/5.5/en/string-functions.

html#function\_format

More on specifying locales can be found at:

http://dev.mysql.com/doc/refman/5.5/en/locale-support.html

[ 359 ]String Functions

UPPER()

In addition to the mathematical functions, MySQL also provides functions to

massage the format of string data. String values frequently come as normal text—

essentially a camel-backed mixture of capitals and lowercase letters. The UPPER()

function makes them all uppercase and takes only the string as an argument:

mysql> SELECT UPPER('Can you find a bar in Barstow?') as results;

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

| results

|

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

| CAN YOU FIND A BAR IN BARSTOW? |

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

1 row in set (0.00 sec)

LOWER()

The LOWER() function is similar to UPPER() in that it performs a single function and

takes only the string to be modified as an argument. As the name implies, it renders

all characters lowercase:

mysql> SELECT LOWER('Meeting at the UN HQ in NYC') as results;

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

| results

|

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

| meeting at the un hq in nyc |

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

1 row in set (0.00 sec)

Project: Creating your own functions

Comparing MySQL's string functions to Python's, you will notice that Python

supports the capitalize() and capwords() functions. These capitalize the initial

letter of the string and the first letter of each word, respectively. MySQL has no

built-in capability to do this. It either returns all uppercase, all lowercase, or the

original format of the string value. To put the onus of capitalization on the MySQL

server, we need to define our own functions.

[ 360 ]Chapter 12

Hello()

To create a function, we necessarily have to go back to the CREATE statement. As in

a Python function definition, MySQL expects us to declare the name of the function

as well as any arguments it requires. Unlike Python, MySQL also wants the type of

data that will be received by the function. The beginning of a basic MySQL function

definition looks like this:

CREATE FUNCTION hello(s CHAR(20))

MySQL then expects to know what kind of data to return. Again, we use the MySQL

data type definitions for this.

RETURNS CHAR(50)

This just tells MySQL that the function will return a character string of 50 characters

or less.

If the function will always perform the same task, it is best for the sake of

performance to include the keyword DETERMINISTIC next. If the behavior of the

function varies, use the keyword NON-DETERMINISTIC. If no keyword is set for the

characteristic of the function, MySQL defaults to NON-DETERMINISTIC.

You can learn more about the characteristic keywords used in function

definitions at:

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

procedure.html

Finally comes the meat of the function definition. Here we can set variables and

perform any calculations that we want. For our basic definition, we will simply

return a concatenated string:

RETURN CONCAT('Hello, ', s, '!');

The function obviously concatenates the word 'Hello' with whatever argument is

passed to it and appends an exclamation point at the end. To call it we use SELECT as

with the other functions:

mysql> SELECT hello('world') as Greeting;

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

| Greeting

|

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

| Hello, world! |

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

1 row in set (0.00 sec)

[ 361 ]String Functions

Capitalise()

A function to capitalize every initial letter in a string follows the same pattern. The

main point of the function is to walk through the string, character by character, and

use UPPER() on every character that does not follow a letter.

DELIMITER

Obviously, we need a way to pass the entire function to MySQL without having any

of the lines evaluated until we call it. To do this, we use the keyword DELIMITER.

DELIMITER allows users to tell MySQL to evaluate lines that end in the character(s)

we set. So the process for complex function definitions becomes:

1. Change the delimiter.

2. Pass the function with the usual semicolons to indicate the end of the line.

3. Change the delimiter back to a semicolon.

4. Call the function.

The DELIMITER keyword allows us to specify more than one character as the line

delimiter. So in order to ensure we don't need to worry about our code inadvertently

conflicting with a line delimiter, let's make the delimiter @@:

DELIMITER @@

The function definition

From here, we are free to define a function to our specification. The definition line

will read as follows:

CREATE FUNCTION `Capitalise`(instring VARCHAR(1000))

The function will return a character string of similar length and variability:

RETURNS VARCHAR(1000)

When MySQL functions extend beyond the simplest calculations, such as hello(),

MySQL requires us to specify the beginning and ending of the function. We do that

with the keywords BEGIN and END. So let's begin the function:

BEGIN

Next, we need to declare our variables and their types using the keyword DECLARE:

DECLARE i INT DEFAULT 1;

DECLARE achar, imark CHAR(1);

DECLARE outstring VARCHAR(1000) DEFAULT LOWER(instring);

[ 362 ]Chapter 12

The DEFAULT keyword allows us to specify what should happen if outstring should

fail for some reason.

Next, we define a WHILE loop:

WHILE i <= CHAR\_LENGTH(instring) DO

The WHILE loop obviously begins with a conditional statement based on the character

length of instring. The resulting action begins with the keyword DO. From here, we

set a series of variables and express what should happen where a character follows

one of the following:

blank space & '' \_ ? ; : ! , - / ( .

The operational part of the function looks like this:

SET achar = SUBSTRING(instring, i, 1);

SET imark = CASE WHEN i = 1 THEN ' '

ELSE SUBSTRING(instring, i - 1, 1) END CASE;

IF imark IN (' ', '&', '''', '\_', '?', ';', ':', '!', ',', '-

', '/', '(', '.') THEN SET outstring = INSERT(outstring, i, 1,

UPPER(achar));

END IF;

SET i = i + 1;

Much of this code is self-explanatory. It is worth noting, however, that the apodosis

of any conditional in MySQL must end with the keyword END. In the case of IF, we

use END IF.

In the second SET statement, the keyword CASE is an evaluative keyword that

functions similar to the try...except structure in Python. If the WHEN condition is

met, the empty THEN apodosis is executed. Otherwise, the ELSE exception applies

and the SUBSTRING function is run. The CASE structure ends with END CASE. MySQL

will equally recognize the use of END instead.

The subsequent IF clause evaluates whether imark, defined as the character before

achar, is one of the declared characters. If it is, then that character in instring is

replaced with its uppercase equivalent in outstring.

After the IF clause is finished, the loop is incremented by one. After the entire string

is processed, we then end the WHILE loop with:

END WHILE;

After the function's operations are completed, we return the value of outstring and

indicate the end of the function:

RETURN outstring;

END@@

[ 363 ]String Functions

Finally, we must not forget to return the delimiter to a semicolon:

DELIMITER ;

It is worth noting that, instead of defining a function in a MySQL session

we can define it in a separate file and load it on the fly with the SOURCE

command. If we save the function to a file called capfirst.sql in a

directory temp, we can source it relatively:

SOURCE capfirst.sql;

We can also use:

SOURCE /home/skipper/temp/capfirst.sql;

Calling the function

With the function loaded into memory, we can then call it:

mysql> SELECT Capitalise('we have a meeting a.s.a.p.');

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

| Capitalise('we have a meeting a.s.a.p.') |

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

| We Have A Meeting A.S.A.P.

|

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

1 row in set (0.00 sec)

Of course, we would not normally write like this. However, we can call the function

as part of a SELECT statement, just like any other MySQL function.

mysql> SELECT CONCAT(Capitalise('we '), 'have a meeting ', Capitalise('a.

s.a.p.')) as Message;

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

| Message

|

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

| We have a meeting A.S.A.P. |

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

1 row in set (0.00 sec)

[ 364 ]Chapter 12

Defining the function in Python

As you can guess by now, calling the function in Python is as simple as passing it

through cursor.execute(). If we have a cursor defined as cursor, we can pass the

last example of the previous section as follows:

statement = "SELECT CONCAT(Capitalise('we '), 'have a meeting ',

Capitalise('a.s.a.p.')) as Message"

cursor.execute(statement)

We then proceed to fetchall() the results.

Defining the function as a Python value

Defining the function is a bit different from calling it. If you try to pass the

function we previously defined through Python as a value of statement, you get a

programming error that reads something like this:

\_mysql\_exceptions.ProgrammingError: (1064, "You have an error in your SQL

syntax; check the manual that corresponds to your MySQL server version

for the right syntax to use near 'DELIMITER @@\n\nCREATE FUNCTION

`Capitalise`(instring varchar(1000))\n\tRETURNS VARC' at line 1")

The problem is the DELIMITER statement. If we pull those and define statement

as follows, we will have no problems (thanks to William Chiquito for the

following code):

statement = """

CREATE FUNCTION `Capitalise`(instring varchar(1000))

RETURNS VARCHAR(1000)

BEGIN

DECLARE i INT DEFAULT 1;

DECLARE achar, imark CHAR(1);

DECLARE outstring VARCHAR(1000) DEFAULT LOWER(instring);

WHILE i <= CHAR\_LENGTH(instring) DO

SET achar = SUBSTRING(instring, i, 1);

SET imark = CASE WHEN i = 1 THEN ' '

ELSE SUBSTRING(instring, i - 1, 1) END;

IF imark IN (' ', '&', '''', '\_', '?', ';', ':', '!', ',', '-', '/',

'(', '.') THEN SET outstring = INSERT(outstring, i, 1, UPPER(achar));

END IF;

[ 365 ]String Functions

SET i = i + 1;

END WHILE;

RETURN outstring;

END;

"""

Putting function definitions into your code increases the amount of runtime

resources needed and can make maintenance quite onerous. Instead, we can save it

in a separate file. Note that MySQL for Python does not allow the use of MySQL's

SOURCE command. So one must use alternative means to the same effect.

Sourcing the MySQL function as a Python module

We can take the preceding code and source it as a Python module. Begin it with

the following:

#!/usr/bin/env python

## This shebang is for a Linux machine. Adjust your shebang line

accordingly

def MakeStatement():

statement = """

CREATE FUNCTION `CapMe`(instring varchar(1000))

...

and end it:

...

END;

"""

return statement

If we save it as capfirst.py, we can reference it as follows:

import capfirst

...

statement = capfirst.MakeStatement()

Sourcing the function as MySQL code

It may seem a bit excessive to put Python's function trappings around a variable

assignment. A simpler way of affecting the same results is to read the MySQL file

into memory as the value of statement and then pass that value through execute():

statement = open("/home/skipper/temp/capfirst.sql").read()

runit = cursor.execute(statement)

[ 366 ]Chapter 12

The effect is the same and the function is created in the end. Similar tactics can

be used whenever you need to source a MySQL file (for example, the dump of

a database).

Room to grow

This function does essentially what we want it to do: capitalizes the first letter of

every word in the string. However, it still has a few drawbacks that should

be addressed:

•What happens when you evaluate the string "we have a meeting a.s.a.p. in

Brussels"? How would you rectify it?

•How would you handle a string such as "we were 'discussing' just that"?

•How do you write a function to load the function when you need it and get

rid of it when you don't?

•The current function emulates the capwords() function of Python. How

would you simulate Python's capitalize() function?

Summary

In this chapter, we have covered several of MySQL's string functions and ways to

use them in Python. We have seen:

•How to concatenate two or more strings

•Ways to return only part of a string, leaving the larger string behind

•How to trim whitespaces from around a value

•Mysql's functions for replacing and inserting values into others

•How to form regular expressions for MySQL

•Ways to locate strings within other strings

•How to calculate the length of a value

•Ways to work with both numbers and character values to nuance raw data

•How to write functions for MySQL and use them in Python

In the next chapter, we will look at how to access the MySQL's metadata from

within Python.

[ 367 ]Showin