

Being Regular with Regular Expressions

John Garmany

Presented at IOUG 2006.

Oracle implemented the ability to use Regular Expressions in the Oracle 10g database but Regular Expressions have been around from many years. UNIX system administrators routinely use Regular Expression in everyday task. Those of us that work with Linux also use Regular Expression, often without knowing it. A Regular Expression is an expression that defines a pattern of characters. You use a Regular Expression to find strings that match a specific pattern. One common UNIX/Linux utility called *grep* uses Regular Expression to find lines that contain characters that match a pattern. In fact, GREP stands for Global Regular Expression Print. The *grep* utility looks at every line and if it contains a match to the pattern, it will display or print the line, otherwise it will skip to the next line. For example, the *rpm -qa* command in RH Linux can be used to display installed operating system packages. It executed alone, a list of all installed packages will scroll across the screen. If I am looking for one particular package, I want to see only that package or packages similar to that in name. I can pipe the list of packages to *grep* and have it filter the results, showing only the packages that match the pattern I am looking for. Lets look for the package that support Regular Expressions called *regexp*.

```
[garmanyj@svr2 Documents]$ rpm -qa | grep regexp
regexp-1.3-1jpp_4rh
```

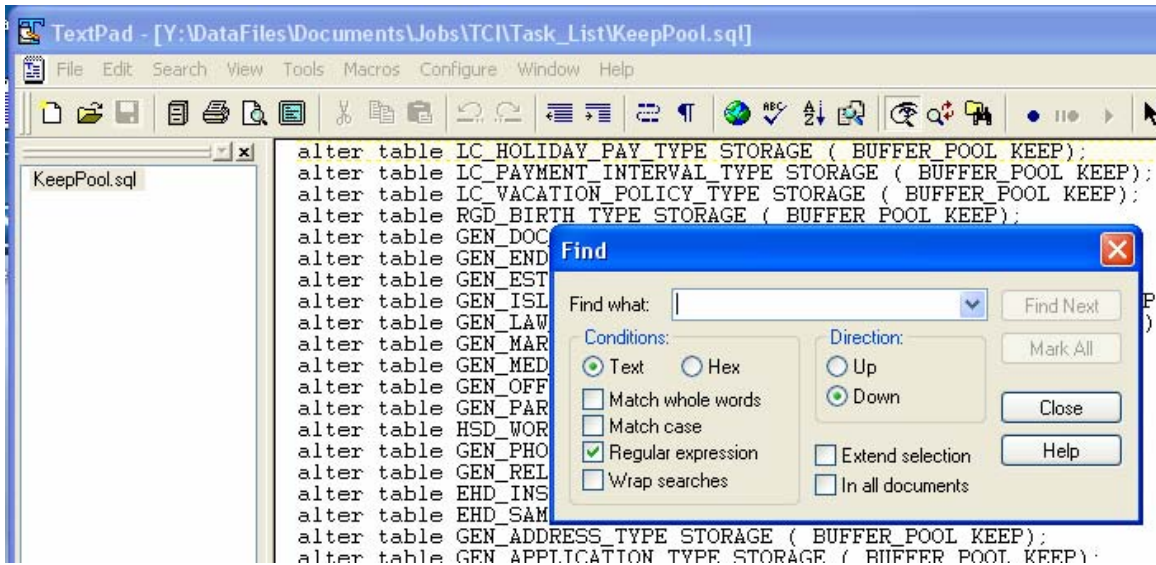
There is only one package on my system that matches the *regexp* pattern. This is similar to using the old substitution characters when looking for a file.

```
[garmanyj@svr2 Monitoring Scripts]$ ls -al clea*
-rwxr--r-- 1 garmanyj garmanyj 1072 Jan 27 19:26 cleanup.ksh
```

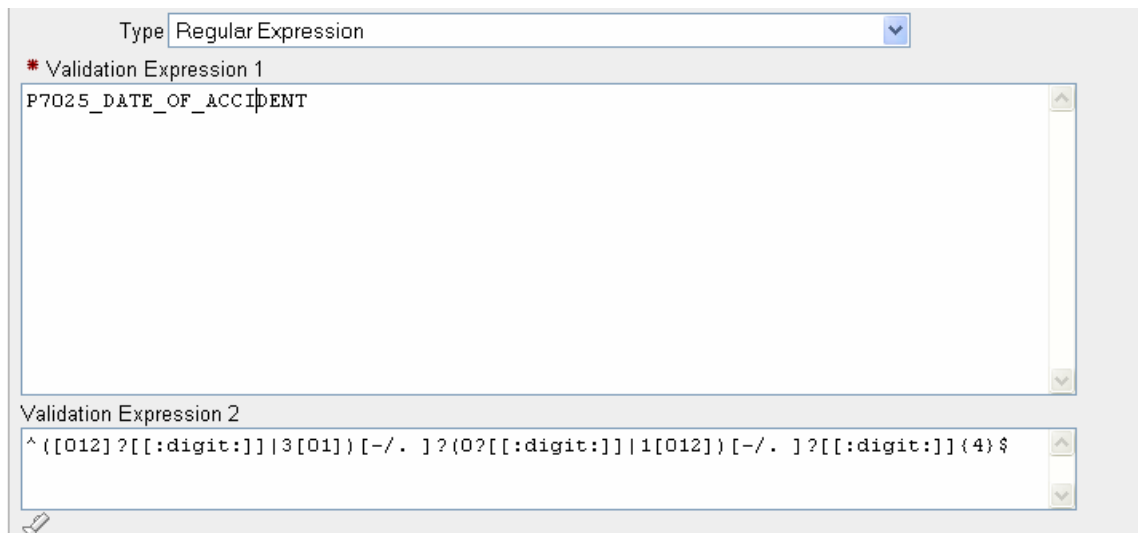
So what make Regular Expression so great? It is the ability to match complicated patters. The *grep* man page states:

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

So you use a combination of operators to define the pattern you are looking for. Regular Expressions are found in many of the tools we use every day, from text editors to compilers. Many search and replace functions are implemented using Regular Expressions. Below, a text editor uses Regular Expression to search a document.



Oracle Application Express (formerly HTMLDB) can use Regular Expressions to validate fields.



In the example above, a date field is validated using Regular Expressions. Let's look at what makes up a Regular Expression so we can understand what the above example is defining.

Regular Expression Patterns.

Regular Expressions have a formal definition as part of the POSIX standard. Different symbols define how a pattern is described. Let's start with the basic symbols.

Characters and Numbers represent themselves. If you are searching for 'abc' then the matching pattern is abc.

Period (.) represents any single character or number. The pattern 'b.e' will match bee, bye, b3e but not bei, or b55e. Likewise the pattern '..-..=...' matches any two characters, followed by a dash, followed by any two characters, followed by an equal sign, followed by any three characters.

Star (*) represent zero or more characters. The pattern 'b.*e' will match bee, bye, beee, bzyxe and be. The pattern '..-..=.*' can end with zero or more characters after the equal sign.

Plus (+) represents one or more chracters. This pattern is the same a '.*' except that there must be one character. Using the pattern 'b.+e' the string "be" will not match.

Question Mark (?) represents zero or one character. The pattern '..-..=?' can only end with one character or no character after the equal sign.

If I wanted to match a US telephone number, I could use the pattern '...-...-....'. This pattern will match any three characters, followed by a dash, followed by 3 more characters, followed by a dash and four final characters. So the string "123-456-7891" will match this pattern. Well so will "abc-def-ghij". So this simple patter will match a lot of strings that are not phone numbers. We will improve on this pattern in a moment.

Brackets are used to define numbers of characters.

{count} defines an exact number of characters. The pattern a{3} defines exactly three character 'a'. Used with the period, the {count} defines the number of characters. The phone number example could be written as the pattern '{3}-{3}-{4}'.

Note: When used with many applications, the bracket already has a meaning and to use it in a expression is must be escaped, normally with a slash. '\{3}\-\{3}\-\{4\}' In this example the slash '\' simply escapes the bracket. With Oracle, this is not necessary.

{min,max} defines a minimum and maximum number of characters. The pattern '{2,8}' will match any 2 or more characters, up to 8 characters.

{min,} defines the minimum or more number of characters. The pattern 'sto{1,}p' will match any string that has 'st' followed by one or more 'o', followed by a 'p'. This includes stop, stoop, stooooop, but not stp or stoip.

Square Brackets are used to define a subset of the expression. Any one character in the bracket will match the patter. If I want only a number character then I could use a pattern like '[123456789]'. The phone number pattern could be written as:

'[123456789]{3}-[123456789]{3}-[123456789]{4}'

With this pattern, I have excluded all the letters from matching strings. A range of characters can also be defined in square brackets. This is easier to type and read. The range is defined using the dash between the min and max. The phone example now becomes:

```
'[1-9]{3}-[1-9]{3}-[1-9]{4}'
```

Ranges of letters can also be defined. The pattern 'st[aeiou][A-Za-z]' matches any string with the characters 'st' followed by a vowel, followed by any character, upper or lower case. This pattern matches stop, stay, staY, stud. The pattern 'abc[1-9]' matches abc1, abc2, abc3,...

The caret [^] in square brackets matches any character except those following the caret. The pattern 'st[^o]p' will match step, strp, but not stop.

So far, all the patterns match is the pattern is found anywhere in the line of text. Use the caret and dollar sign to define patterns that match the start or end of a string.

^ defines that start of a string or column 1 of the string.

\$ defines the end of a string or the last column. This does not include carriage returns and line feeds.

The pattern '^St[a-z]*' matches a string that starts with 'St' followed by zero or more lower case letters. The pattern 'stop\$' only matches "stop" if it is the last word on the line.

| or vertical line defines the Boolean OR. The pattern '[1-9][a-z]' matches any number or lower case letter. The pattern 'stop|step' matches the strings stop or step.

\ or backward slash is the escape character. This is used to tell the parser that the character following it is to be taken literally. In the note earlier, it was pointed out that some characters have special meaning in some applications and must be escaped to tell the application to use that literal character. Another reason to escape a character is when you want to actually use the character in your matching pattern. For example, if you want to match a number that has two decimal places you could use the pattern:

```
'[1-9]+\.[1-9]{2}'
```

This example looks right but will not match the pattern that we are looking for. The period we use to represent the decimal place, will actually match any character. We must tell the expression parser that we want the character period and we do that by escaping the period character.

```
'[1-9]+\. [1-9]{2}'
```

Now the pattern will match one or more digits followed by a period and exactly two digits.

Class Operators

Class operators are used as an alternative way to define classes of characters. They are defined in the format `[: class :]`.

<code>[:digit:]</code>	Any digit
<code>[:alpha:]</code>	Any upper or lower case letter
<code>[:lower:]</code>	Any lower case letter
<code>[:upper:]</code>	Any upper case letter
<code>[:alnum:]</code>	Any upper or lower case letter or number
<code>[:xdigit:]</code>	Any hex digit
<code>[:blank:]</code>	Space or Tab
<code>[:space:]</code>	Space, tab, return, line feed, form feed
<code>[:cntrl:]</code>	Control Character, non printing
<code>[:print:]</code>	Printable character including a space
<code>[:graph:]</code>	Printable characters, excluding space.
<code>[:punct:]</code>	Punctuation character, not a control character or alphanumeric

Again, these class operators represent other characters. The phone number example can be rewritten using class operators.

```
'[:digit:]{3}-[:digit:]{3}-[:digit:]{4}'
```

Being Greedy

Regular expressions are greedy. By this we mean that the expression will match the largest string it can. Think of it as the expression parser takes the entire string and compares it to the pattern. The parser then gives back characters until it finds that the string has no match or if finds the match.

Lets use a string '123423434'

If my pattern is `'.*4'` (zero or more characters followed by the digit 4). The first match will be the entire string.

Expression Grouping

Expression Grouping allows part of the pattern to be grouped. This is also called tagging or referencing. You group an expression by surrounding it with parens. There can be only 9 groups in a pattern. Below is an example that contains two groups.

```
'([a-z]+) ([a-z]+)'
```

This pattern matches two lower case words. Using a string defined as 'fast stop', the first group would contain 'fast' and the second group 'stop'. The groups are referenced by a backward slash and the group number. '\1' references 'fast' while '\2' reference 'stop'. Thus \2 \1 results in 'stop fast'.

Oracle and Regular Expressions

The Oracle 10g database provides four functions to implement Regular Expressions. The Java Virtual Machine in the database also implements the Java support for Regular Expression. The four functions can be used in SQL statements or PL/SQL. They operate on the database character datatypes to include VARCHAR2, CHAR, CLOB, NVARCHAR2, NCHAR, and NCLOB. The four functions are:

REGEXP_LIKE Returns true is the pattern is matched, otherwise false.

REGEXP_INSTR Returns the position of the start or end of the matching string. Returns zero is not the pattern does not match.

REGEXP_REPLACE Returns a string where each matching string is replaced with the text specified.

REGEXP_SUBSTR Returns the matching string, or NULL if no match is found.

Let's look at each of the functions and how to put them to use.

REGEXP_LIKE

Syntax: `regexp_like(source, pattern(, options));`

The source is a text literal, variable or column. The pattern is the expression you are looking for. The options define how the matching will take place. The options are:

i = case insensitive

c = case sensitive

n = the period will match a new line character

m = allows the ^ and \$ to match the beginning and end of lines contained in the source. Normally these characters would match the beginning and end of the source. This is for multi-line sources.

This function can be used anywhere a Boolean result is acceptable.

```
begin
  n_phone_number varchar2(20);
  ...
begin
  ...
  if (regexp_like(n_phone_number, .*[567]$)) then ...
  end if;
  ...
end;
```

If the phone number ends in either 5,6 or 7, the return it true and the THEN clause is executed.

In a SQL statement, this function can only be used in the WHERE or HAVING clause.

```
select
  ...
  phone
from
  ...
where regexp_like(phone, .*[567]$);
```

The REGEXP_LIKE function can also be used in check constraints.

REGEXP_REPLACE

Syntax: `regexp_replace(source, pattern, replace string, position, occurrence, options)`

The source can be a string literal, variable, or column. The pattern is the expression to be replaced. The replace string is the text that will replace the matching patterns. The optional position defines the location to begin searching the source string. This defaults to 1. The optional occurrence defines the occurrence of the pattern that you want replaced. This defaults to 0 (all occurrences). Setting this to a positive number will result in only that occurrence being replaced. The matching options are the same.

```
select
  regexp_replace('We are driving south by south east',
                'south', 'north')
from dual;
```

We are driving north by north east

REGEXP_INSTR

Syntax: `regexp_instr(source, pattern, position, occurrence, begin_end, options)`

The source can be a string literal, variable, or column. The pattern is the expression to be replaced. The optional position is the location to begin the search and defaults to 1. The occurrence defines the occurrence you are looking for. The `begin_end` defines whether you want the position of the beginning of the occurrence or the position of the end of the occurrence. This defaults to 0 with is the beginning of the occurrence. Use 1 to get the end position. The matching options are the same.

```
select
  regexp_instr('We are driving south by south east',
              'south')
from dual;
```

```
16
select
  regexp_instr('We are driving south by south east',
              'south', 1, 2, 1)
from dual;
```

30

REGEXP_SUBSTR

Syntax: `regexp_substr(source, pattern, position, occurrence, options)`

The source can be a string literal, variable, or column. The pattern is the expression to be replaced. The optional position is the location to begin the search and defaults to 1. The optional occurrence defines the occurrence you are looking for. The matching options are the same.

```
select
  regexp_substr('We are driving south by south east',
              'south')
from dual;
```

south

Final Note

Using Regular Expressions adds a powerful pattern matching capability to your SQL and PL/SQL toolbox. However, you should not use Regular Expression when a simple LIKE clause will work. There is a lot more overhead to matching a pattern using Regular Expression as opposed to character wild cards. If you are looking for all rows where last_name starts with 'GAR', the LIKE clause (where xxx like 'GAR%') will perform better than using Regular Expression. But as your searches become more complicated, you will begin to realize just how powerful pattern matching with Regular Expression can be.