Table of Contents

1. Regular Expressions

Regular expressions allow you to do more complex pattern matching than wildcards. Wildcards are limited to matching any single character and any series of characters. With regular expressions we can build a pattern that specifies specific characters to search for, ranges of characters and repeating patterns of characters.

A regular expression is a string that expresses a pattern for text matching. The regular expression string can contain literal characters and metacharacters. For example the regular expression 'cat' matches the string 'cat and the regular expression 'p..t' matches a string that contains a 'p' followed by any two characters followed by a 't'.

One of the most important things to understand about regular expressions is that they can work very well for matching strings that have some sort of consistent pattern, but they are not a good for matching unstructured text. You may have to decide if you want to use a pattern that produces a lot of matches including undesired matches (false positives), or a pattern that produces fewer matches but misses some of the matches you want to find. You can find examples of regular expression patterns on various web sites but you need to test these- for example does the pattern for a phone number assume that you will use parentheses and a dash (415) 239-3768 or does it allow the format 415.239.3768; does it handle extensions?, country codes? Does the url pattern you found assume that the top level domain is two or three characters long? Does it assume English characters only?

If you have worked with regular expressions, you know that they can get quite complex- we will stick with some simple patterns. If you are planning to use complex regular expression, consult the Oracle manuals for more details and test carefully.

In order to use the regular expressions, you need functions to interpret the regular expression. Oracle includes the following functions for working with regular expressions.

- **REGEXP_LIKE** Returns Boolean
- **REGEXP_INSTR** Returns number
- **REGEXP_SUBSTR** Returns part of the string
- **REGEXP_REPLACE** Returns string with replacements

Each of these functions is built similar to the regular string functions/operators (Like, Instr, Substr, Replace).

They each take a source string, a regular expression pattern to match, and optional match options. The match options are

- i case insensitive
- c case sensitive
- n period matches new line
- m more than one line in source is ok

Regexp_Instr, Regexp_Substr, and Regexp_Replace take an optional start position for the matching and an optional occurrences parameter to indicate which occurrence should be returned.

Regexp_Replace has a parameter for the replacement string

Regexp_Instr has a parameter for a return option

- `rtrn_option = 0` → `return first character of the occurrence`
- `rtrn_option = 1` → `return first character following the occurrence`

That was probably rather confusing and intimidating. If this you first experience with regular expressions just work though the section on Metacharacters and the demos for RegExp_Like.

2. Metacharacters

The Like operator recognized the metacharacters % and _. Regular expressions have a much richer set of metacharacters. You can find a list of them in the Price book

What the regex expressions provide is the ability to search for ranges/lists of characters. Instead of looking for any single character as with a wildcard, you could look for any of the listed characters [aeiouy]. You could also specify that you want to match from 3 to 6 of those characters in a row [aeiouy]{3-6}

You could match a pattern of as many 0s and 1s as occurs by using [01]* and if you really want at least one of these, use [01]+

There are also predefined patterns such as `[:lower:]` which stands for any lower case letter (bracket expressions, character classes)

In this first set of demos we use the metacharacters

^	start of the string
$	end of the string
.	any single character
{n}	repetition; want exactly n of the previous character g{3} matches ggg
{n, m}	repetition; want between n and m of the previous character g{3,5} matches ggg, gggg, ggggg
*****	repetition; any number, including 0 of the preceding character
+	repetition; any number, but at least one, of the preceding character
?	repetition; zero or one of the preceding character
[abc…]	list - matches any one of the included characters
[a-p]	range - matches any one of the characters in the indicated range

3. Demos

3.1. Regexp_Like

RegExp_Like is used in a Where clause and returns True if that pattern occurs within the search string.

For these demos we will use a table set up for this purpose. The table has an ID column and a name column; we will be testing against the name column. The SQL to create the table is in the demo.

These are the rows in the table.

```
        ID NAME
---------- -------------------------
         1 Fluffy
         2 goofy
         3 ursula
         4 greg
         5 pout
         6 Sam 415
         7 pretty bird
         8 pat
         9 peat
        10 Patricia
```

```
        11 Impromptu
        12 Pete
        13 pat the cat
        14 C3PO
        15 Mary Proud
        16 ptt
        17 pita
```

As you work with these, try to do the same task using string functions and wildcards,

Demo 01: The ^ **metacharacter** matches the start of the string.

```
select * from z_tst_reg
where regexp_Like (name,  '^g')
;
```

```
        ID NAME
---------- -------------------------
         2 goofy
         4 greg
```

Note that if we do not include the character for matching the start of the string, we can match anywhere inside the string. For example, Regexp_Like (name, 'p') matches Impromptu as well as matching pita

Demo 02: The **$ metacharacter** matches the end of the string.

```
select * from z_tst_reg
where regexp_Like (name,  'g$')
;
```

```
        ID NAME
---------- -------------------------
         4 greg
```

Demo 03: The **. (dot) metacharacter** matches any single character. This pattern matches any string that starts with a p and ends with a t and has exactly one character between.

```
select * from z_tst_reg
where regexp_Like (name,  '^p.t$')
;
```

```
        ID NAME
---------- -------------------------
         8 pat
        16 ptt
```

Demo 04: This pattern matches any string that starts with a p and ends with a t and has exactly two characters between.

```
select * from z_tst_reg
where regexp_Like (name,  '^p..t$')
;
```

```
        ID NAME
---------- -------------------------
         5 pout
         9 peat
```

Demo 05: It can help to also display rows that do not match the pattern

```
select * from z_tst_reg
where Not regexp_Like(name, '^p..t$');
```

```
        ID NAME
---------- -------------------------
         1 Fluffy
         2 goofy
         3 ursula
         4 greg
         6 Sam 415
         7 pretty bird
         8 pat
        10 Patricia
        11 Impromptu
        12 Pete
        13 pat the cat
        14 C3PO
        15 Mary Proud
        16 ptt
        17 pita
```

Demo 06: We can use **{n}** to indicate that we want exactly n of the preceding character. Note that I do not have the $ to tie this to the end of the string.

```
select * from z_tst_reg
where regexp_Like (name,  '^p.{3}t')
;
```

```
        ID NAME
---------- -------------------------
         7 pretty bird
        13 pat the cat
```

Demo 07: We can use **{n, m}** to indicate that we want between n and m of the preceding character.

```
select * from z_tst_reg
where regexp_Like (name,  '^p.{4,7}a', 'i')
;
```

```
        ID NAME
---------- -------------------------
        10 Patricia
```

Demo 08: We can use ***** to indicate that we will match any number, including 0, of the preceding character.

```
select * from z_tst_reg
where regexp_Like (name,  '^p.*t')
;
```

```
        ID NAME
---------- -------------------------
         5 pout
         7 pretty bird
         8 pat
         9 peat
        13 pat the cat
        16 ptt
        17 pita
```

Demo 09: Use + to indicate that you must match at least one. Use ? for matching either 0 or 1 of the preceding.

```
select * from z_tst_reg
where regexp_Like (name,  'pr?o');
```

```
        ID NAME
---------- -------------------------
         5 pout
        11 Impromptu
```

Demo 10: Include the **'i' option** for case insensitivity.

```
select * from z_tst_reg
where regexp_Like (name,  'pr?o', 'i')
;
```

```
        ID NAME
---------- -------------------------
         5 pout
        11 Impromptu
        14 C3PO
        15 Mary Proud
```

Demo 11: The use of **[ ]** allows for a list or a range of characters to match.

```
select * from z_tst_reg
where regexp_Like (name,  '[aeiouy]$')
;
```

```
        ID NAME
---------- -------------------------
         1 Fluffy
         2 goofy
         3 ursula
        10 Patricia
        11 Impromptu
        12 Pete
        17 pita
```

Demo 12: The [ ] allows for a list or a range of characters to match. This matches an r followed by a single vowel followed by a character in the range a-m.

```
select * from z_tst_reg
where regexp_Like (name,  'r[aeiouy][a-m]')
;
```

```
        ID NAME
---------- -------------------------
         4 greg
        10 Patricia
        11 Impromptu
```

Demo 13: Regular expression also includes character classes. This matches strings that include any whitespace character

```
select * from z_tst_reg
where regexp_Like (name,'[[:blank:]]')
;
```

```
        ID NAME
---------- -------------------------
         6 Sam 415
         7 pretty bird
        13 pat the cat
        15 Mary Proud
```

Demo 14: This matches strings that include an upper case letter followed by a lower case letter.

```
select * from z_tst_reg
where regexp_Like (name,'[[:upper:]][[:lower:]]')
;
```

```
        ID NAME
---------- -------------------------
         1 Fluffy
         6 Sam 415
        10 Patricia
        11 Impromptu
        12 Pete
        15 Mary Proud
```

Demo 15: This matches any digit

```
select * from z_tst_reg
where regexp_Like (name, '[[:digit:]]')
;
```

```
        ID NAME
---------- -------------------------
         6 Sam 415
        14 C3PO
```

3.2. Regexp_Instr

This is a model to use to experiment with regular expressions. The first two statements define a bind variable that will hold a string. The next statements set the values of those variables. You can change the literals to do other tests. Finally there is a select statement that uses the regexp_Instr function to tell you where the pattern occurs in the string.

The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the `return_option` argument. If no match is found, the function returns 0.

Demo 16: Regexp_Instr

```
variable s varchar2(250)
variable p varchar2(25)

exec :s  := 'This is my test string';
exec :p  := 'is';


select regexp_Instr(:s, :p) as Location
from dual;

  LOCATION
----------
         3
```

REGEXP_INSTR and REGEXP_SUBSTR now (Oracle 11g) have an optional subexpr parameter that lets you target a particular substring of the regular expression being evaluated. You can say where to start the search and which occurrence to match.

Demo 17: Regexp_Instr with start and count parameters

```
select
  regexp_Instr(:s, :p)         as Loc_1
, regexp_Instr(:s, :p, 5 , 1)  as Loc_2
```

```
   , regexp_Instr(:s, :p, 10 , 1) as Loc_3
   from dual;
```

```
     LOC_1         LOC_2         LOC_3
----------    ----------    ----------
         3             6             0
```

(This is easier to use in SQL Developer where you can paste all of these lines into the code window and then edit the literals and rerun all of the code.)

You can also do simple demos with literals

Demo 18: Exact match with anchors; only '^CAT$' is a match

```
   select
     regexp_Instr ('CAT', '^CAT$')   as test_01,
     regexp_Instr ('CAT', '^CAAT$')  as test_02,
     regexp_Instr ('CAT', '^CAAAAT$') as test_03
   from dual;
```

```
TEST_01               TEST_02               TEST_03
--------------------- --------------------- ---------------------
1                     0                     0
```

Demo 19: The only one that matches is Sno- with 4 o's followed by a w

```
   select
     regexp_Instr ('Snoooow', '^Sno{3}w$') as test_04,
     regexp_Instr ('Snoooow', '^Sno{4}w$') as test_05,
     regexp_Instr ('Snoooow', '^Sno{5}w$') as test_06
   from dual;
```

```
TEST_04               TEST_05               TEST_06
--------------------- --------------------- ---------------------
0                     1                     0
```

Demo 20: The first argument contains 7 o's so this matches both {7,7} and {5,9}

```
   select
      regexp_Instr ('Shmooooooo' , '^Shmo{3,4}$')  as test_7,
      regexp_Instr ('Shmooooooo' , '^Shmo{7,7}$')  as test_8,
      regexp_Instr ('Shmooooooo' , '^Shmo{5,9}$')  as test_9,
      regexp_Instr ('Shmooooooo' , '^Shmo{9,12}$') as test_10
   from dual;
```

```
TEST_7                TEST_8                TEST_9                TEST_10
--------------------- --------------------- --------------------- ---------------------
0                     1                     1                     0
```

Demo 21: This uses different patterns for the preceding character. The preceding character can be a single literal character or a [ ] list of possible characters to match

```
   select
      regexp_Instr ('XabcbcacW', '^Xa{4,8}W$')     as test_11,
      regexp_Instr ('XabcbcacW', '^X[ab]{4,8}W$')  as test_12,
      regexp_Instr ('XabcbcacW', '^X[abc]{4,8}W$') as test_13,
      regexp_Instr ('XabcbcacW', '^X.{4,8}W$')     as test_14
   from dual;
```

```
TEST_11               TEST_12               TEST_13               TEST_14
--------------------- --------------------- --------------------- ---------------------
0                     0                     1                     1
```

Demo 22: Using variable for the limits take a bit more work. You do not want the variable identifier :low inside the delimiters. The first expression does not handle the variables properly. You use || to build up the expression.

```
variable b_low number;
exec :b_low := 5;
variable b_high number;
exec :b_high := 9;


select
  :'Shmooooooo' , '^Shmo{:b_low,:b_high}$' ) as test_15,
  regexp_Instr ('Shmooooooo' , '^Shmo{'  || :b_low || ',' || :b_high || '}$') as test_16
from dual;
```

```
TEST_15               TEST_16
--------------------- ----------------------
0                     1
```

3.3. RegExp_Replace

With the following literal, there are three blanks between San and Francisco. The first version of RegExp_Replace says to replace two blanks with one- that leaves us with two blanks. The second version says to replace any 2 or more blanks with a single blank.

```
select regexp_Replace('San   Francisco', '  ', ' ')
from Dual;

select regexp_Replace('San    Francisco', '[ ]{2,}', ' ')
from Dual;
```

3.4. RegExp_Count

RegExp_Count is a new (Oracle 11g) function that counts the number of occurrences of a specified regular expression pattern in a source string.

Demo 23: RegExp_Count

```
select name, regexp_Count(name, '[aeiou]') as Vowel_count
, regexp_Count(name, 'at') as at_count
from z_tst_reg ;
```

NAME	VOWEL_COUNT	AT_COUNT
Fluffy	1	0
goofy	2	0
ursula	3	0
greg	1	0
pout	2	0
Sam 415	1	0
pretty bird	2	0
pat	1	1
peat	2	1
Patricia	4	1
Impromptu	2	0
Pete	2	0
pat the cat	3	2
C3PO	0	0
Mary Proud	3	0
ptt	0	0
pita	2	0