Regular Expressions for Regular People

Dan Stober
Intermountain Healthcare
Enterprise Data Warehouse Team
March xx, 2012

Objectives

- Know what a regular expression is and where and how to use it
- Make regular expressions less intimidating
- Understand the characters that make up a regular expression
- Be able to write your own regular expression
- Know how to use Oracle's 11g regular expression functions

- **Leave here with confidence that you can apply regular expressions to solve an issue in your own work!**

Agenda

- Regular expressions
  - Pattern matching
  - Metacharacters
  - Interactive examples
- Oracle's 11g functions
  - Five SQL Functions
  - Matches vs Occurrences
  - Back References
  - Regular Expressions in APEX and PLSQL ????

Who am I?

- Dan Stober
  - Intermountain Healthcare
    - Data Architect, EDW
  - UTOUG Board of Trustees
  - Working in Oracle Databases since 2001
  - California State Univ., Fresno
  - Past presenter at Oracle Conferences
    - Oracle Open World, San Francisco 2006, 2007
    - IOUG / OAUG Collaborate 2004, 2006, 2008, 2010, 2012
    - UTOUG Training Days 2007, 2008, 2009, 2010, 2011

  - Session Norms
    - Questions? Interrupt me!
      - I learn from sessions, too
    - Cell phones – Leave 'em on
    - All examples are on the slides

Intermountain Healthcare

- 25 Hospitals in two states
- 35,000 Employees
- Not for Profit
- Two Words – Not three

Intermountain Healthcare

- Enterprise Data Warehouse
  - Established 1998
- 35 Architects and Analysts
- 35,000,000,000 records in 8900 queriable tables
- Average 100,000,000 queries per month
- Supporting Clinical and Business Users
  - Care Protocols
  - Regulatory Compliance
  - Financial Stability

What are Regular Expressions?

- Text string search that facilitates complex pattern matching
  - We've all used simple pattern matching in SQL

```
WHERE last_nm LIKE 'Sto%'
```

- Terminology:
  - Pattern
  - Test String
- Boolean test
  - Does the pattern exist in the test string?
  - New SQL functions allow more than Boolean tests

Where can you use Regular Expressions?

- UNIX grep and sed
- Perl

- Text Editors
  - TextPad
  - Multi-Edit
- APEX Validation Logic

- Cheating at Scrabble
  - Thank you Brian Rawlings!

- SQL New in Oracle 10g
  - Significant enhancements on 11g
  - PLSQL, too!

Some basics

- Patterns consist of characters and metacharacters
  - Characters: letters and numerals represent themselves (generally)
  - Metacharacters: characters that have a special meaning in a pattern. ie: wildcards

```
WHERE last_nm LIKE 'Sto%'
```

- Regular Expressions are case-sensitive
  - “a” does not match “A”
- Multiple ways to express the same pattern
- Can be very difficult to read and debug
  - Designed for machine reading
  - Be alter for unintended matches

The Period Wildcard

- Matches exactly one character
  (letter, numeral, whitespace, crtl char)

TEST PATTERN:	d.n	
☑ dan	☑ d6n	☑ Sudan
☒ Dan	☒ d66n	☒ Dance
☑ don	☑ Ddan	☑ d%n

Explicit Choice

- Square Brackets
- "Character Class"
- Any one of the characters matches to exactly one character in the test string

TEST PATTERN:	d[aeo]n	
☑ dan	☑ verdant	☒ dean
☒ din	☒ duncecap	☑ Eden
☑ don	☒ Dennys	☒ damnation

Not / Except

- Character Class with negation
- Caret ⇨ shift + 6
- Matches any character, except those listed
- Remember: A character class still represents only a single character position in the string

TEST PATTERN:	d[^auz]n	
✗ dan	✓ d6n	✓ danden
✓ din	✗ d66n	✗ dandeen
✓ don	✓ trident	✓ dendan

Character Class with Range

- Square Brackets
- Still represents a single character
- List low and high ends of range
  - Separated by dash
  - Examples:

[A-Z]	All capital letters from A to Z	**[ABCDEFGHIJKLMNOPQRSTUVWXYZ]**
[0-9]	All numerals	**[0123456789]**

- More than one range is permitted
  - Do not separate with space
    - [A-Za-z] or [a-z0-9]

Character Class with Range

TEST PATTERNS:		
d[a-m]n	d[0-9]n	d[A-Za-z]n
☑ dan	☒ dan	☑ dan
☒ don	☑ d6n	☑ dAn
☑ Eden	☒ d22n	☒ d6n
☒ damnation	☑ k9d4n7g0	☒ DAan
☒ dEn		

Using What We Know So Far

- We now know enough to write a pattern
- Validate the format of a Social Security Number

```
[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]
```

- It's lengthy, but it will work!
- Notice that dash serves two roles in this pattern
  - BOTH character and metacharacter

Repeating & Counting

- Use Curly Braces {}
- Indicates number of times that preceding character should appear
- Usage:

Single Number	`{3}`	Exact Count
Two Numbers (With Comma)	`{1,4}`	Minimum and maximum
One Number (with comma)	`{2,}`	Minimum, but no maximum

Quantifier { }

TEST PATTERN:		
d.{2}n	d.{1,3}n	d.{0,}n
☑ dawn	☑ dan	☑ kidney
☒ drawn	☒ dn	☑ dogfighting
☑ predawn	☑ Dresden	☑ Could not
☑ donneybrook	☒ draping	☒ Dick Nourse

Improving our Example...

- Simplifying the pattern from earlier
- Using quantifier to validate the format of a SSN

```
[0-9]{3}-[0-9]{2}-[0-9]{4}
```

Special Quantifiers

- Indicate number of times that preceding character should appear

Question Mark	?	Preceding character is optional but can appear only once	{0,1}
Plus Sign	+	Preceding character is mandatory but can appear multiple times	{1,}
Asterisk	*	Preceding character is optional but can appear multiple times	{0,}

- These shortcut metacharacters are much more common than the {} quantifier metacharacters

Special Quantifiers

?	⇨ {0,1}
*	⇨ {0,}
+	⇨ {1,}

TEST PATTERN:	d+a*n?	
☑ dan	☑ ddddddn	☑ dada
☑ ddan	☑ ddddddnn	☑ din
☑ dddddd	☑ dddaaa	☑ Dda

- Notice that a “d” by itself matches with this pattern

Quantifying Groups of Characters

- Use parentheses
- Quantifier acts on entire sequence
- Same logic applies
  - (ab){2} ⇨ abab
  - (ab){3}c ⇨ abababc

Grouper ( )

? ⇨ {0,1}
* ⇨ {0,}
+ ⇨ {1,}

TEST PATTERN:		
d(.n){2}	(dan)+	(d[0-9])?n
✗ dan	✓ dan	✓ d7n
✓ danon	✓ dandandan	✓ d7d8n
✗ dannon	✗ ddnadn	✓ n
✓ dDnan	✗	✓ d77n

Class Operators

- Provide a name for a range of characters
- Enclosed by colons with double square brackets
  - It's a range
- Also called POSIX Class operators
  - Easier to read than perl's class operators (ie: \d)
- Here are a few (there are more):

`[[:digit:]]`	Any numeral	`[0-9]`
`[[:alpha:]]`	Any upper case or lower case letter	`[A-Za-z]`
`[[:upper:]]`	Upper case letter only	`[A-Z]`
`[[:lower:]]`	Lower case letter only	`[a-z]`
`[[:space:]]`	Whitespace character (space, tab)	
`[[:alnum:]]`	Any letter (either case) or number	`[A-Za-z0-9]`

Improving our Example again...

- Simplifying the pattern from earlier
- Using quantifier to validate the format of a SSN

```
[[:digit:]]{3}-[[:digit:]]{2}-[[:digit:]]{4}
```

Anchor Characters

^	^dan	Match MUST occur at the beginning of the string
$	dan$	Match MUST occur at the end of the string
^$	^dan$	Match is the ONLY thing permitted in the string

Two more metacharacters

Pipe	|	Indicates an explicit choice. Used with groups. Ie: (black)|(gr[ae]y) car matches “black car”, “gray car”, and “grey car”
Backslash	\	With a metacharacter: Literal escape. Ignore the special meaning of the metacharacter which follows. Instead, the character itself must be in the string
		With regular characters: when preceded with a backslash, it gives the regular a special meaning. Ie: \d does the same thing as [0-9] or [[:digit:]] . With a number, it makes it a backreference.

Two more examples

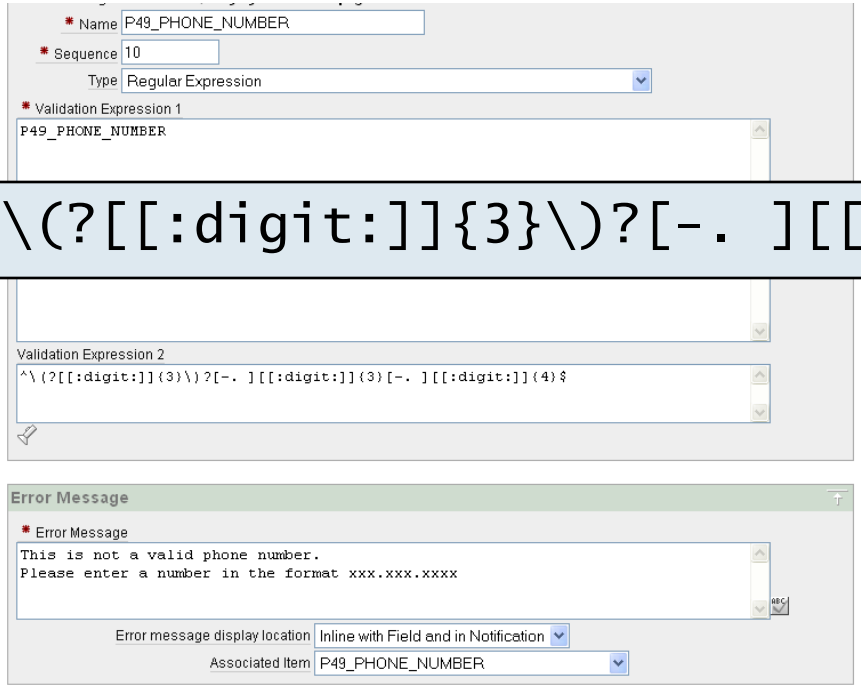
- Canadian Postal Code
  - Six characters:
    - Winnipeg MB B9G 6T3
    - letter number letter space number letter number

```
[[:alpha:]][[:digit:]][[:alpha:]]
[[:space:]][[:digit:]][[:alpha:]][[:digit:]]
```

- US Zip Code:
  - Five digits
    - West Jordan UT 84084
  - Or Nine digits (five digits dash four digits)
    - West Jordan UT 84084-1053

```
[[:digit:]]{5}(-[[:digit:]]{4})?
```

Back to that APEX Example



- Is this the best pattern?

```
^\(?[[:digit:]]{3}\)?[-. ][[:digit:]]{3}[-. ][[:digit:]]{4}$
```

- Could we write a better one?

Pattern matches these:	But it also permits these:
801.442.3470	(801)-442 3470
801-442-3470	(801).442-3470
(801) 442-3470	801) 442-3470
801 442 3470	(801.442-3470
801 442-3470	801)-442.3470

Back Reference

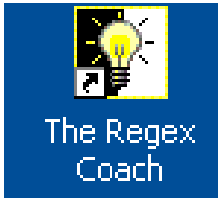
- Find the same string that was matched earlier
- Use backslash, followed by number (1-9)

```
\1
```

- Matches by ordinal position of parentheses in the pattern
- Can be used in any regular expression

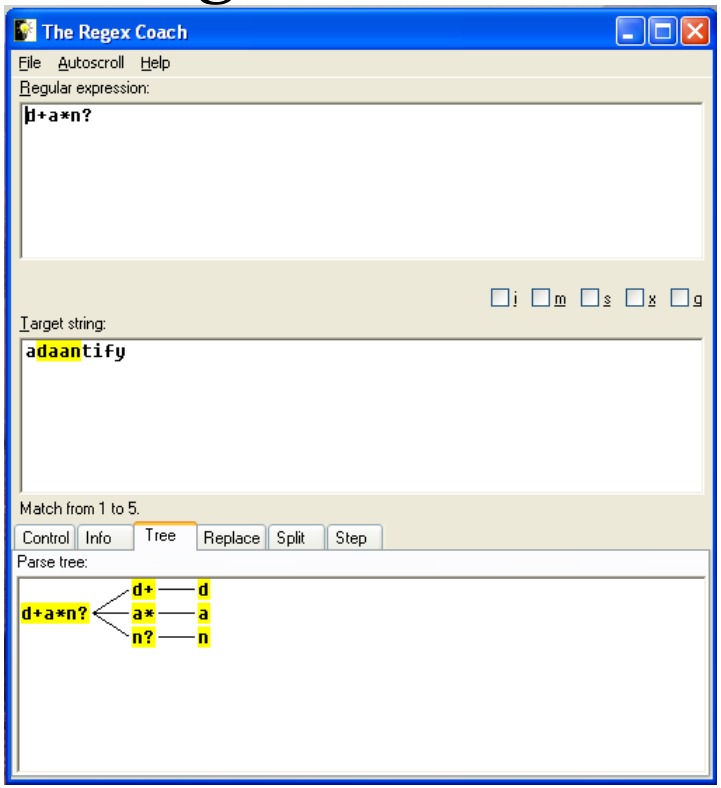
TEST PATTERN:	d(.n){2}	d(.n)\1
dan	✗	✗
danon	✓	✗
dDnan	✓	✗
danan	✓	✓
gardening	✓	✗
dining	✓	✓

RegEx Coach



- Freeware tool
- Evaluates patterns against test strings

- Download from CNET
  - www.download.com
  - Search for “Regex Coach”



Oracle SQL Functions

- In 10g, four functions
  - REGEXP_LIKE ⇨ returns BOOLEAN
  - REGEXP_SUBSTR ⇨ returns VARCHAR2
  - REGEXP_INSTR ⇨ returns NUMBER
  - REGEXP_REPLACE ⇨ returns VARCHAR2
- In 11g, one new function
  - REGEXP_COUNT ⇨ returns NUMBER
- Note the similarity to like-named functions
  - But there are some nuances

REGEXP_LIKE

- Returns Boolean
  - Does the pattern exist in test string?
- Syntax:

```
REGEXP_LIKE (test_string, pattern)
```

  - One additional parameter optional
- Differs from SQL "LIKE"
  - Do not use %
  - It's a function, not an operator – 11gR2 documentation calls it a "condition"
- Use in:
  - WHERE clause
  - a check constraint
  - PLSQL – IF and CASE statements
- Both params are VARCHAR
  - Literal strings in single quotes
  - Single quotes in the pattern or test string must be escaped with two single quotes

```
ORA-01756: quoted string not properly terminated
```

```
REGEXP_LIKE ('O''Hare Airport', '''')
```

REGEXP_LIKE

```
SELECT *
FROM regexp_examples
WHERE REGEXP_LIKE (testval, 'd.n')
```

Test String | Pattern

- This query returns all records from the table where the value in testval field matches the pattern 'd.n'

TEST PATTERN:	d.n	
☑ dan	☑ d6n	☑ Sudan
☒ Dan	☒ d66n	☒ Dance
☑ don	☑ Ddan	☑ d&n

regexp_like.sql

REGEXP_SUBSTR

- Does not just let you know that the pattern matched, tells you *what* matched
- Syntax:

```
REGEXP_SUBSTR (test_string, pattern)
```

  - Other parameters optional
- Returns String
  - Portion that matched the pattern
  - Longest match possible from starting char
  - First occurrence, by default
- If no match
  - returns NULL

Greedy Metacharacters

Do you recall this example from earlier?

TEST PATTERN:	d+a*n?	
☑ dan	☑ ddddddn	☑ dada
☑ ddan	☑ ddddddnn	☑ din
☑ dddddd	☑ dddaaa	☑ Dda

- When using * + {}
  - If a pattern matches more than one string in the test string, regular expression functions evaluate to longest possible match

```
PATTERN  TEST_STR  MATCH
-------  --------  --------
d+a*n?   dan       dan
d+a*n?   ddan      ddan
d+a*n?   dddddd    dddddd
d+a*n?   ddddddn   ddddddn
d+a*n?   dddddnn   ddddddn
d+a*n?   dddaaa    dddaaa
d+a*n?   dada      da
d+a*n?   din       d
d+a*n?   Dda       da
```

Remember: The letter "d" by itself matches the pattern

This illustrates the principle of "greediness." The function will return the longest possible match

Avoiding Greediness

```
In his inaugural address, President Kennedy said, "Ask
not what your country can do for you, ask what you can
do for your country." This speech marked the beginning
of an era known affectionately as "Camelot".
```

- Try to extract the quotation from the larger text

TEST PATTERN: `".+"`

```
REGEXP_SUBSTR (x, '".+"')
```

☒ NO!

TEST PATTERN: `"[^"]+"`

```
REGEXP_SUBSTR (x, '"[^"]+"')
```

☑ YES!

REGEXP_SUBSTR

Complete parameter list

```
1: Test String          (VARCHAR2)
2: Pattern              (VARCHAR2)
3: Starting position    (NUMBER)
4: Occurrence           (NUMBER)
5: Match Option         (VARCHAR2)
6: Subexpression        (NUMBER)
```

TEST STR: ddddnnndaandan

PATTERN: d+a*n?

START POS	OCCURRENCE	RESULT
1	1	ddddn
4	1	dn
5	1	daan
1	2	daan
1	4	*NULL*

Occurrences

TEST PATTERN: `d.+n`

Dan dances dangerously in the darned garden.

- How many occurrences do you see?

Only ONE!

```
SELECT
  REGEXP_SUBSTR('Dan dances dangerously in the darned garden','d.+n',1,2) occ2
, REGEXP_SUBSTR('Dan dances dangerously in the darned garden','d.+n',1,4) occ4
, REGEXP_SUBSTR('Dan dances dangerously in the darned garden','d.+n',1,1) occ1
FROM DUAL;

O O OCC1
- - ------------------------------------
    dances dangerously in the darned garden

1 row selected.
```

Occurrence

- Counting matches is different than counting occurrences
- Multiple matches within the same occurrence are caused by greediness
- Any substring match which is part of another, greedier substring match is part of the same occurrence

Match Option Parameter

- Fifth parameter
- VARCHAR2
- Most common:
  - 'c' Case sensitive – default
  - 'i' Case INsenstive
- Other options to ignore line breaks

Subexpression

NEW 11g

- New param for 11g
- Once a match is identified, extract only a portion of it
- Use backreferences by number

Please call (385) 555-1212 today.

Requirement:
1. Find a phone number
2. Get the area code and phone number in two separate fields

Without this parameter, the way to solve it is with nesting

```
REGEXP_SUBSTR(  REGEXP_SUBSTR(mytext
                          , '\([[:digit:]]{3}\) [[:digit:]]{3}-[[:digit:]]{4}')
             , '[[:digit:]]{3}' , 1, 1)
```

```
REGEXP_SUBSTR(  REGEXP_SUBSTR(mytext
                          , '\([[:digit:]]{3}\) [[:digit:]]{3}-[[:digit:]]{4}')
             , '[[:digit:]]{3}-[[:digit:]]{4}$')
```

Subexpression solution

- Once a match is located, extract a portion of it
- Designated by parentheses

```
REGEXP_SUBSTR(mytext
            , '\(([[:digit:]]{3})\) ([[:digit:]]{3}-[[:digit:]]{4})'
            , 1, 1, 'c'
            , 1 )
```

Return the first parenthetical subexpression

```
REGEXP_SUBSTR(mytext
            , '\(([[:digit:]]{3})\) ([[:digit:]]{3}-[[:digit:]]{4})'
            , 1, 1, 'c'
            , 2 )
```

Return the first parenthetical subexpression

- Dddaaa
- D(.)\1
- Two different matches

REGEXP_INSTR

- Returns Number
  - Starting position of match for pattern
    - This is the default Return Option
  - Vis-à-vis the greediest pattern
- Syntax:

```
REGEXP_INSTR (test_string, pattern)
```

  - Additional parameters optional
- If no match
  - Returns 0 (zero)

REGEXP_INSTR

- Complete Parameter List:

```
1: Test String          (VARCHAR2)
2: Pattern              (VARCHAR2)
3: Starting position    (NUMBER)
4: Occurrence           (NUMBER)
5: Return Option        (NUMBER)
6: Match Option         (VARCHAR2)
```

REGEXP_INSTR

- Return Option Parameter
  - 0 ⇨ Return the position of the start of the match
  - >0 ⇨ Return the position of the *n*th character after the end of the greediest match

TEST STR: ddddnnndaandan

PATTERN: d+a*n?

START POS	OCCURRENCE	RETURN OPTION	MATCH RESULT	FUNCTION RETURN
1	1	0	ddddn	1
1	1	1	ddddn	6
2	1	0	dddn	2
1	2	0	daan	8
1	3	1	dan	15
1	4	0	*NULL*	0

Note that this value is beyond the length of the string

REGEXP_REPLACE

- Syntax:

```
REGEXP_REPLACE (test_string
                , pattern
                , replacement)
```

  - Additional parameters optional
- Returns Varchar
  - Returns original "test_string" with pattern match(es) replaced by "replacement"
  - Replaces the greediest pattern match
- Default: Replaces ALL Occurrences
- If replacement parameter is null
  - Returns original string stripped of match
- If no match
  - Returns original string

REGEXP_REPLACE

- Additional Parameters

```
1: Test String        (VARCHAR2)
2: Pattern            (VARCHAR2)
3: Replacement Str    (VARCHAR2)
4: Starting position (NUMBER)
5: Occurrence         (NUMBER)
6: Match Option       (VARCHAR2)
```

REGEXP_REPLACE

TEST STR:	1dan2den3don4
PATTERN:	d.n
REPLACEMENT:	''

START POS	OCCURRENCE	RETURN VALUE
NULL	*NULL*	1234
3	*NULL*	1dan234
1	2	1dan23don4
3	2	1dan2den34

Parsing and Scrubbing Text

- To get the matching pattern only

```
REGEXP_SUBSTR (linetext, pattern)
```

- To get everything *except* the matching pattern

```
REGEXP_REPLACE (linetext, pattern)
```

Back Reference examples

```
SELECT REGEXP_REPLACE ('Paris in the the spring.'
                  ,'([[:alpha:]]+)[[:space:]]\1',  ''  ) r
FROM DUAL
/

R
-----------------
Paris in  spring.

1 row selected.
```

```
SELECT REGEXP_REPLACE ('Paris in the the spring.'
,'([[:alpha:]]+)[[:space:]]\1',  '\1'  ) r
FROM DUAL
/

R
--------------------
Paris in the spring.

1 row selected.
```

Use a Backreference in the replacement value param The reference points to the parentheses in the test string param.

REGEXP_REPLACE

- Use with a backreference, to replace pieces of the match

```
SELECT REGEXP_REPLACE
    (   'Dick VanDyke'
    ,   '([[:lower:]])([[:upper:]])'
    ,   '\1 \2')                        name_fixed
FROM DUAL
/

NAME_FIXED
--------------
Dick Van Dyke

1 row selected.
```

REGEXP_REPLACE use case

- Insert Bold HTML tags

```
SELECT REGEXP_REPLACE
    ('1dan2den3don4'
    ,'(d.n)'
    , '<b>' ||'\1' || '<\b>'
    ,  1, 3) r
FROM DUAL
/

R
--------------------
1dan2den3<b>don<\b>4

1 row selected.
```

On one occurrence . . .

```
SELECT REGEXP_REPLACE
    ('1dan2den3don4'
    ,'(d.n)'
    , '<b>' ||'\1' || '<\b>') r
FROM DUAL
/

R
---------------------------------
1<b>dan<\b>2<b>den<\b>3<b>don<\b>4

1 row selected.
```

. . . or on All Occurrences

REGEXP_COUNT

```
1: Test String        (VARCHAR2)
2: Pattern            (VARCHAR2)
3: Starting position  (NUMBER)
4: Match Option       (VARCHAR2)
```

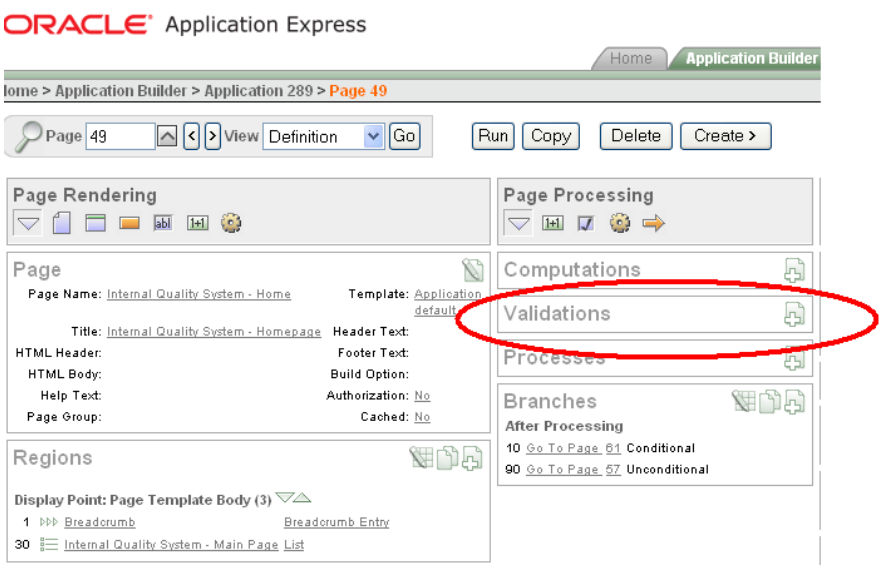
- Returns Number
  - How many times does the pattern appear in the test string?
  - Greedy matches count only once
- Syntax:

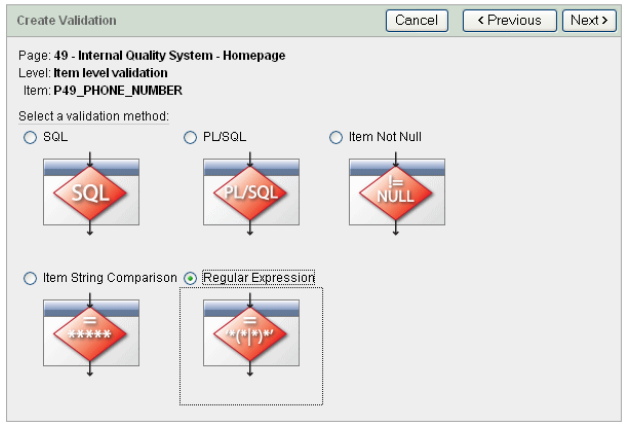
```
REGEXP_COUNT (test_string, pattern)
```

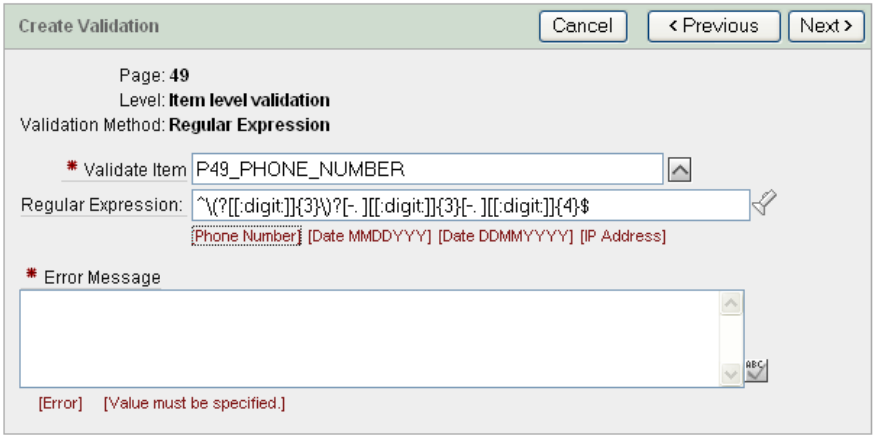
  - Additional parameters optional

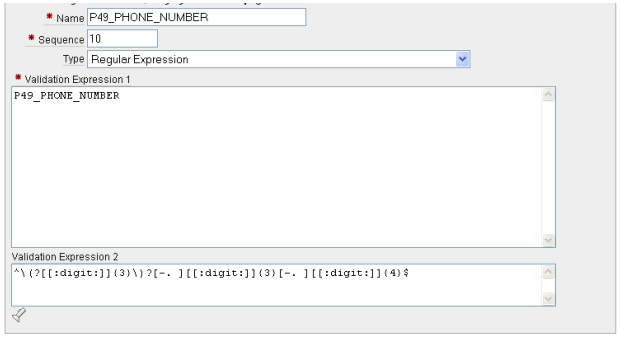
utoug
utah oracle users group

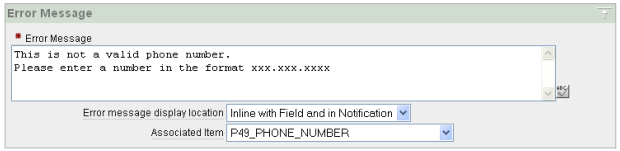
Regular Expression Validation in APEX











RegExp in PLSQL

- The same four or five functions are available
  - Same parameters
- REGEXP_LIKE returns boolean
  - Use in IF and CASE statements

RegExp SQL Errors

```
ORA-12725: unmatched parentheses in regular expression
```

- Parentheses are not balanced.
  - Sometimes caused when an escaped literal parenthesis tricks you with the appearance that the parentheses are evenly matched.
  - There is a similar error for unmatched brackets: ORA-12726

```
ORA-12729: invalid character class in regular expression
```

- Use an unknown POSIX: `[[:digits:]]`

```
ORA-12728: invalid range in regular expression
```

- For example: `[Z-A]`

Recap

- Regular expressions
  - Pattern matching
  - Metacharacters
  - Interactive examples
- Oracle's functions
  - Four SQL Functions in 10g
  - Fifth one in 11g
  - Matches vs Occurrences
  - Back References
  - Regular Expressions in APEX and PLSQL

Objectives in Review

- Know what a regular expression is and where and how to use it
- Make regular expressions less intimidating
- Understand the characters that make up a regular expression
- Be able to write your own regular expression
- Know how to use Oracle's regular expression functions

- **Leave here with confidence that you can apply regular expressions to solve an issue in your own work!**

Thank-you!

- Dan Stober
- Questions / comments:
  - dan.stober@imail.org


utoug
utah oracle users group