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**

- **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

- 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*
    ```sql
    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 altered for unintended matches
The Period Wildcard

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

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d.n</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ dan</td>
<td>✓ d6n</td>
<td>✓ Sudan</td>
</tr>
<tr>
<td>✗ Dan</td>
<td>✗ d66n</td>
<td>✗ Dance</td>
</tr>
<tr>
<td>✓ don</td>
<td>✓ Ddan</td>
<td>✓ d%n</td>
</tr>
</tbody>
</table>
Explicit Choice

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

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d[aeo]n</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔ dan</td>
<td>✔ verdant</td>
</tr>
<tr>
<td>✗ din</td>
<td>✗ duncecap</td>
</tr>
<tr>
<td>✔ don</td>
<td>✗ Dennys</td>
</tr>
</tbody>
</table>
Not / Except

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

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d[^auz]n</th>
</tr>
</thead>
<tbody>
<tr>
<td>dan</td>
<td>✔️ d6n</td>
</tr>
<tr>
<td>✔️ din</td>
<td>✗ d66n</td>
</tr>
<tr>
<td>✔️ don</td>
<td>✔️ trident</td>
</tr>
</tbody>
</table>
Character Class with Range

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

<table>
<thead>
<tr>
<th></th>
<th>All capital letters from A to Z</th>
<th>All numerals</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A-Z]</td>
<td>[ABCDEFGHIJKLMNOPQRSTUVWXYZ]</td>
<td>[0123456789]</td>
</tr>
<tr>
<td>[0-9]</td>
<td>[0123456789]</td>
<td></td>
</tr>
</tbody>
</table>

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

### TEST PATTERNS:

<table>
<thead>
<tr>
<th>d[a-m]n</th>
<th>d[0-9]n</th>
<th>d[A-Za-z]n</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔ dan</td>
<td>❌ dan</td>
<td>✔ dan</td>
</tr>
<tr>
<td>❌ don</td>
<td>✔ d6n</td>
<td>✔ dAn</td>
</tr>
<tr>
<td>✔ Eden</td>
<td>❌ d22n</td>
<td>❌ d6n</td>
</tr>
<tr>
<td>❌ damnation</td>
<td>✔ k9d4n7g0</td>
<td>❌ DAan</td>
</tr>
<tr>
<td>❌ dEn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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][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:

<table>
<thead>
<tr>
<th>Single Number</th>
<th><code>{3}</code></th>
<th>Exact Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two Numbers (With Comma)</td>
<td><code>{1, 4}</code></td>
<td>Minimum and maximum</td>
</tr>
<tr>
<td>One Number (with comma)</td>
<td><code>{2,}</code></td>
<td>Minimum, but no maximum</td>
</tr>
</tbody>
</table>
## Quantifier { }

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d.{2}n</th>
<th>d.{1,3}n</th>
<th>d.{0,}n</th>
</tr>
</thead>
<tbody>
<tr>
<td>dawn</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>drawn</td>
<td>✗</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>predawn</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>donneybrook</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>✗</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
<th>Quantifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question Mark</td>
<td>Preceding character is optional but can appear only once</td>
<td>{0,1}</td>
</tr>
<tr>
<td>Plus Sign</td>
<td>Preceding character is mandatory but can appear multiple times</td>
<td>{1,}</td>
</tr>
<tr>
<td>Asterisk</td>
<td>Preceding character is optional but can appear multiple times</td>
<td>{0,}</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d+a*n?</th>
</tr>
</thead>
<tbody>
<tr>
<td>💚 dan</td>
<td>💚 dddddd</td>
</tr>
<tr>
<td>💚 ddan</td>
<td>💚 ddddd</td>
</tr>
<tr>
<td>💚 dddddd</td>
<td>💚 dddaa</td>
</tr>
</tbody>
</table>

?  ⇒ {0,1}
*  ⇒ {0,}
+  ⇒ {1,}
Quantifying Groups of Characters

- Use parentheses
- Quantifier acts on entire sequence
- Same logic applies
  - \((ab)^2\) \Rightarrow abab
  - \((ab)^3c\) \Rightarrow abababc
## Grouper ( )

### TEST PATTERN:

<table>
<thead>
<tr>
<th></th>
<th>d(.n){2}</th>
<th>(dan)+</th>
<th>(d[0-9])?n</th>
</tr>
</thead>
<tbody>
<tr>
<td>dan</td>
<td>❌</td>
<td>✔️</td>
<td>✔️ d7n</td>
</tr>
<tr>
<td>danon</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️ d7d8n</td>
</tr>
<tr>
<td>dannon</td>
<td>❌</td>
<td>❌</td>
<td>✔️ n</td>
</tr>
<tr>
<td>dDnan</td>
<td>✔️</td>
<td>❌</td>
<td>✔️ d77n</td>
</tr>
</tbody>
</table>

- `?` $\Rightarrow \{0,1\}$
- `*` $\Rightarrow \{0,\}$
- `+` $\Rightarrow \{1,\}$
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):

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:digit:]</td>
<td>Any numeral</td>
<td>[0-9]</td>
</tr>
<tr>
<td>[:alpha:]</td>
<td>Any upper case or lower case letter</td>
<td>[A-Za-z]</td>
</tr>
<tr>
<td>[:upper:]</td>
<td>Upper case letter only</td>
<td>[A-Z]</td>
</tr>
<tr>
<td>[:lower:]</td>
<td>Lower case letter only</td>
<td>[a-z]</td>
</tr>
<tr>
<td>[:space:]</td>
<td>Whitespace character (space, tab)</td>
<td></td>
</tr>
<tr>
<td>[:alnum:]</td>
<td>Any letter (either case) or number</td>
<td>[A-Za-z0-9]</td>
</tr>
</tbody>
</table>
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>^dan</td>
</tr>
<tr>
<td>$</td>
<td>dan$</td>
</tr>
<tr>
<td>^$</td>
<td>^dan$</td>
</tr>
</tbody>
</table>
# Two more metacharacters

| Pipe | Indicates an explicit choice. Used with groups. I.e:  (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. I.e: \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?
- Could we write a better one?

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

Pattern matches these:
- 801.442.3470
- 801-442-3470
- (801) 442-3470
- 801 442 3470
- 801 442-3470
- 801 442 3470

But it also permits these:
- (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

<table>
<thead>
<tr>
<th>TEST PATTERN:</th>
<th>d(.n){2}</th>
<th>d(.n)\1</th>
</tr>
</thead>
<tbody>
<tr>
<td>dan</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>danon</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>dDnan</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>danan</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gardening</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>dining</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
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**

```sql
SELECT * 
FROM regexp_examples 
WHERE REGEXP\_LIKE (testval, 'd.n')
```

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

<table>
<thead>
<tr>
<th>TEST PATTERN</th>
<th>d.n</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ dan</td>
<td>✓ d6n</td>
</tr>
<tr>
<td>✗ Dan</td>
<td>✗ d66n</td>
</tr>
<tr>
<td>✓ don</td>
<td>✓ Ddan</td>
</tr>
<tr>
<td></td>
<td>✓ d&amp;n</td>
</tr>
<tr>
<td></td>
<td>✓ Sudan</td>
</tr>
<tr>
<td></td>
<td>✓ Dance</td>
</tr>
</tbody>
</table>
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?

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

<table>
<thead>
<tr>
<th>PATTERN</th>
<th>TEST_STR</th>
<th>MATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>d+a*n?</td>
<td>dan</td>
<td>dan</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>ddan</td>
<td>ddan</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>dddddd</td>
<td>dddddd</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>ddddddnn</td>
<td>dddddd</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>ddddddnn</td>
<td>dddddd</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>ddddaaa</td>
<td>ddaaa</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>dada</td>
<td>da</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>din</td>
<td>d</td>
</tr>
<tr>
<td>d+a*n?</td>
<td>Dda</td>
<td>da</td>
</tr>
</tbody>
</table>

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?
```

<table>
<thead>
<tr>
<th>START</th>
<th>POS</th>
<th>OCCURRENCE</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>ddddn</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>1</td>
<td>dn</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1</td>
<td>daan</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>daan</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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;

0 0 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 INsensitive
- Other options to ignore line breaks
Subexpression

- 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

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

```sql
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:
  
  \[ \text{REGEXP_INSTR (test_string, pattern)} \]

  - 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 nth character after the end of the greediest match

| TEST STR: | dddddnnndaandonan |
| PATTERN:  | d+a*n?             |

<table>
<thead>
<tr>
<th>START POS</th>
<th>OCCURRENCE</th>
<th>RETURN OPTION</th>
<th>MATCH RESULT</th>
<th>FUNCTION</th>
<th>RETURN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ddddn</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ddddn</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>dddn</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>daan</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>dan</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>NULL</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Note that this value is beyond the length of the string.
REGEXP_REPLACE

- **Syntax:**
  ```sql
  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

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test String</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>2</td>
<td>Pattern</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>3</td>
<td>Replacement Str</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>4</td>
<td>Starting position</td>
<td>NUMBER</td>
</tr>
<tr>
<td>5</td>
<td>Occurrence</td>
<td>NUMBER</td>
</tr>
<tr>
<td>6</td>
<td>Match Option</td>
<td>VARCHAR2</td>
</tr>
</tbody>
</table>
The image provides an example of using the `REGEXP_REPLACE` function in Oracle SQL. The query selects a string and replaces occurrences of a pattern with a replacement string. Here is the SQL query and the resulting table:

```sql
SELECT REGEXP_REPLACE ('1dan2den3don4', 'd.n', '', 1, 2) FROM DUAL
```

The test string is `1dan2den3don4`. The pattern to match is `d.n`, and the replacement string is ''. The query replaces each occurrence of `d.n` in the test string with nothing. The table shows the start position, occurrence, and return value for each replacement:

<table>
<thead>
<tr>
<th>Start Pos</th>
<th>Occurrence</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>1234</td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td>1dan234</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1dan23don4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1dan2den34</td>
</tr>
</tbody>
</table>
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

```sql
SELECT REGEXP_REPLACE ('Paris in the the spring.' , 
   '([[:alpha:]]+)\1', ' ' ) r 
FROM DUAL 
/
R------------------
Paris in spring.
1 row selected.
```

```sql
SELECT REGEXP_REPLACE ('Paris in the the spring.' , 
   '([[:alpha:]]+)\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

```sql
SELECT REGEXP_REPLACE ('1dan2den3don4' , '(d.n)' , '<b>' || '<\b>' , 1, 3) r
FROM DUAL /
R
---------------------
1dan2den3<b>don</b>4
1 row selected.
```

```sql
SELECT REGEXP_REPLACE ('1dan2den3don4' , '(d.n)' , '<b>' || '<\b>' , 1, 3) r
FROM DUAL /
R
---------------------
1dan2den3<b>don</b>4
1 row selected.
```

On one occurrence . . .

. . . or on All Occurrences
REGEXP_COUNT

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

- Syntax:
  ```sql
  REGEXP_COUNT (test_string, pattern)
  ```
  - Additional parameters optional
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