<https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm>

[Skip Headers](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#BEGIN)

4 Using Regulaar Expressions in Oracle Database

This chapter introduces regular expression support for Oracle Database. This chapter covers the following topics:

* [Using Regular Expressions with Oracle Database: Overview](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#i1011021)
* [Regular Expression Metacharacters in Oracle Database](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#i1007670)
* [Using Regular Expressions in SQL Statements: Scenarios](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDBCCDJ)

|  |
| --- |
| **See Also:**   * [*Oracle Database SQL Reference*](https://docs.oracle.com/cd/B19306_01/server.102/b14200/toc.htm) for information about Oracle Database SQL functions for regular expressions * [*Oracle Database Globalization Support Guide*](http://www.oracle.com/pls/topic/lookup?ctx=db102&id=NLSPG041) for details on using SQL regular expression functions in a multilingual environment * *Oracle Regular Expressions Pocket Reference* by Jonathan Gennick, O'Reilly & Associates * *Mastering Regular Expressions* by Jeffrey E. F. Friedl, O'Reilly & Associates |

Using Regular Expressions with Oracle Database: Overview

This section contains the following topics:

* [What Are Regular Expressions?](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDDFAFA)
* [How Are Oracle Database Regular Expressions Useful?](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDEDGJD)
* [Oracle Database Implementation of Regular Expressions](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#i1007663)
* [Oracle Database Support for the POSIX Regular Expression Standard](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDJGBGG)

What Are Regular Expressions?

Regular expressions enable you to search for patterns in string data by using standardized syntax conventions. You specify a regular expression by means of the following types of characters:

* Metacharacters, which are operators that specify search algorithms
* Literals, which are the characters for which you are searching

A regular expression can specify complex patterns of character sequences. For example, the following regular expression searches for the literals **f** or **ht**, the **t** literal, the **p** literal optionally followed by the **s** literal, and finally the colon (**:**) literal:

(f|ht)tps?:

The parentheses are metacharacters that group a series of pattern elements to a single element; the pipe symbol (**|**) matches one of the alternatives in the group. The question mark (**?**) is a metacharacter indicating that the preceding pattern, in this case the **s** character, is optional. Thus, the preceding regular expression matches the **http:**, **https:**, **ftp:**, and **ftps:** strings.

How Are Oracle Database Regular Expressions Useful?

Regular expressions are a powerful text processing component of programming languages such as Perl and Java. For example, a Perl script can process each HTML file in a directory, read its contents into a scalar variable as a single string, and then use regular expressions to search for URLs in the string. One reason that many developers write in Perl is for its robust pattern matching functionality.

Oracle's support of regular expressions enables developers to implement complex match logic in the database. This technique is useful for the following reasons:

* By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications. For example, life science customers often rely on Perl to do pattern analysis on bioinformatics data stored in huge databases of DNAs and proteins. Previously, finding a match for a protein sequence such as **[AG].{4}GK[ST]** would be handled in the middle tier. The SQL regular expression functions move the processing logic closer to the data, thereby providing a more efficient solution.
* Prior to Oracle Database 10*g*, developers often coded data validation logic on the client, requiring the same validation logic to be duplicated for multiple clients. Using server-side regular expressions to enforce constraints solves this problem.
* The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and less cumbersome than in previous releases of Oracle Database.

Oracle Database Implementation of Regular Expressions

Oracle Database implements regular expression support with a set of Oracle Database SQL functions and conditions that enable you to search and manipulate string data. You can use these functions in any environment that supports Oracle Database SQL. You can use these functions on a text literal, bind variable, or any column that holds character data such as **CHAR**, **NCHAR**, **CLOB**, **NCLOB**, **NVARCHAR2**, and **VARCHAR2** (but not **LONG**).

[Table 4-1](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDIBCGD) gives a brief description of the regular expression functions and conditions.

***Table 4-1 SQL Regular Expression Functions and Conditions***

| **SQL Element** | **Category** | **Description** |
| --- | --- | --- |
| REGEXP\_LIKE | Condition | Searches a character column for a pattern. Use this function in the **WHERE** clause of a query to return rows matching a regular expression. The condition is also valid in a constraint or as a PL/SQL function returning a boolean. The following **WHERE** clause filters employees with a first name of Steven or Stephen:  WHERE REGEXP\_LIKE(first\_name, '^Ste(v|ph)en$') |
| REGEXP\_REPLACE | Function | Searches for a pattern in a character column and replaces each occurrence of that pattern with the specified string. The following function puts a space after each character in the **country\_name** column:  REGEXP\_REPLACE(country\_name, '(.)', '\1 ') |
| REGEXP\_INSTR | Function | Searches a string for a given occurrence of a regular expression pattern and returns an integer indicating the position in the string where the match is found. You specify which occurrence you want to find and the start position. For example, the following performs a boolean test for a valid email address in the **email** column:  REGEXP\_INSTR(email, '\w+@\w+(\.\w+)+') > 0 |
| REGEXP\_SUBSTR | Function | Returns the substring matching the regular expression pattern that you specify. The following function uses the **x** flag to match the first string by ignoring spaces in the regular expression:  REGEXP\_SUBSTR('oracle', 'o r a c l e', 1, 1, 'x') |

A string literal in a **REGEXP** function or condition conforms to the rules of SQL text literals. By default, regular expressions must be enclosed in single quotes. If your regular expression includes the single quote character, then enter two single quotation marks to represent one single quotation mark within the expression. This technique ensures that the entire expression is interpreted by the SQL function and improves the readability of your code. You can also use the q-quote syntax to define your own character to terminate a text literal. For example, you could delimit your regular expression with the pound sign (**#**) and then use a single quote within the expression.

|  |
| --- |
| **Note:**  If your expression comes from a column or a bind variable, then the same rules for quoting do not apply. |

|  |
| --- |
| **See Also:**   * [*Oracle Database SQL Reference*](https://docs.oracle.com/cd/B19306_01/server.102/b14200/ap_posix.htm#SQLRF020) for syntax, descriptions, and examples of the **REGEXP** functions and conditions * [*Oracle Database SQL Reference*](https://docs.oracle.com/cd/B19306_01/server.102/b14200/sql_elements003.htm#SQLRF00218) for a discussion of character literals |

Oracle Database Support for the POSIX Regular Expression Standard

Oracle's implementation of regular expressions conforms to the following standards:

* IEEE Portable Operating System Interface (POSIX) standard draft 1003.2/D11.2
* Unicode Regular Expression Guidelines of the Unicode Consortium

Oracle Database follows the exact syntax and matching semantics for these operators as defined in the POSIX standard for matching ASCII (English language) data. You can find the POSIX standard draft at the following URL:

[**http://www.opengroup.org/onlinepubs/007908799/xbd/re.html**](http://www.opengroup.org/onlinepubs/007908799/xbd/re.html)

Oracle Database enhances regular expression support in the following ways:

* Extends the matching capabilities for multilingual data beyond what is specified in the POSIX standard.
* Adds support for the common Perl regular expression extensions that are not included in the POSIX standard but do not conflict with it. Oracle Database provides built-in support for some of the most heavily used Perl regular expression operators, for example, character class shortcuts, the non-greedy modifier, and so on.

Oracle Database supports a set of common metacharacters used in regular expressions. The behavior of supported metacharacters and related features is described in ["Regular Expression Metacharacters in Oracle Database"](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#i1007670).

|  |
| --- |
| **Note:**  The interpretation of metacharacters differs between tools that support regular expressions. If you are porting regular expressions from another environment to Oracle Database, ensure that the regular expression syntax is supported and the behavior is what you expect. |

Regular Expression Metacharacters in Oracle Database

This section contains the following topics:

* [POSIX Metacharacters in Oracle Database Regular Expressions](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDIDJJC)
* [Regular Expression Operator Multilingual Enhancements](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#i690823)
* [Perl-Influenced Extensions in Oracle Regular Expressions](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDGHBHF)

POSIX Metacharacters in Oracle Database Regular Expressions

[Table 4-2](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDIEGEI) lists the list of metacharacters supported for use in regular expressions passed to SQL regular expression functions and conditions. These metacharacters conform to the POSIX standard; any differences in behavior from the standard are noted in the "Description" column.

***Table 4-2 POSIX Metacharacters in Oracle Database Regular Expressions***

| **Syntax** | **Operator Name** | **Description** | **Example** |
| --- | --- | --- | --- |
| **.** | Any Character — Dot | Matches any character in the database character set. If the **n** flag is set, it matches the newline character. The newline is recognized as the linefeed character (**\x0a**) on UNIX and Windows or the carriage return character (**\x0d**) on Macintosh platforms.  **Note:** In the POSIX standard, this operator matches any English character except NULL and the newline character. | The expression **a.b** matches the strings **abb**, **acb**, and **adb**, but does not match **acc**. |
| **+** | One or More — Plus Quantifier | Matches one or more occurrences of the preceding subexpression. | The expression **a+** matches the strings **a**, **aa**, and aaa, but does not match **bbb**. |
| **?** | Zero or One — Question Mark Quantifier | Matches zero or one occurrence of the preceding subexpression. | The expression **ab?c** matches the strings **abc** and **ac**, but does not match **abbc**. |
| **\*** | Zero or More — Star Quantifier | Matches zero or more occurrences of the preceding subexpression. By default, a quantifier match is greedy because it matches as many times as possible while still allowing the rest of the match to succeed. | The expression **ab\*c** matches the strings **ac**, **abc**, and **abbc**, but does not match **abb**. |
| **{*m*}** | Interval—Exact Count | Matches exactly***m*** occurrences of the preceding subexpression. | The expression **a{3}** matches the strings **aaa**, but does not match **aa**. |
| **{*m*,}** | Interval—At Least Count | Matches at least ***m*** occurrences of the preceding subexpression. | The expression **a{3,}** matches the strings **aaa** and **aaaa**, but does not match **aa**. |
| **{*m*,*n*}** | Interval—Between Count | Matches at least ***m***, but not more than ***n*** occurrences of the preceding subexpression. | The expression **a{3,5}** matches the strings **aaa**, **aaaa**, and **aaaaa**, but does not match **aa**. |
| **[ ... ]** | Matching Character List | Matches any single character in the list within the brackets. The following operators are allowed within the list, but other metacharacters included are treated as literals:   * Range operator: **-** * POSIX character class: **[: :]** * POSIX collation element:**[. .]** * POSIX character equivalence class: **[= =]**   A dash (**-**) is a literal when it occurs first or last in the list, or as an ending range point in a range expression, as in **[#--]**. A right bracket (**]**) is treated as a literal if it occurs first in the list.  **Note:** In the POSIX standard, a range includes all collation elements between the start and end of the range in the linguistic definition of the current locale. Thus, ranges are linguistic rather than byte values ranges; the semantics of the range expression are independent of character set. In Oracle Database, the linguistic range is determined by the **NLS\_SORT** initialization parameter. | The expression **[abc]** matches the first character in the strings **all**, **bill**, and **cold**, but does not match any characters in **doll**. |
| **[^ ... ]** | Non-Matching Character List | Matches any single character not in the list within the brackets. Characters not in the non-matching character list are returned as a match. Refer to the description of the Matching Character List operator for an account of metacharacters allowed in the character list. | The expression **[^abc]** matches the character **d** in the string **abcdef**, but not the character **a**, **b**, or **c**. The expression **[^abc]+** matches the sequence **def** in the string **abcdef**, but not **a**, **b**, or **c**.  The expression **[^a-i]** excludes any character between **a** and **i** from the search result. This expression matches the character**j** in the string **hij**, but does not match any characters in the string **abcdefghi**. |
| **|** | Or | Matches one of the alternatives. | The expression **a|b** matches character **a** or character **b**. |
| **( ... )** | Subexpression or Grouping | Treats the expression within parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators. | The expression **(abc)?def** matches the optional string **abc**, followed by **def**. Thus, the expression matches **abcdefghi** and **def**, but does not match **ghi**. |
| **\*n*** | Backreference | Matches the nth preceding subexpression, that is, whatever is grouped within parentheses, where ***n*** is an integer from 1 to 9. The parentheses cause an expression to be remembered; a backreference refers to it. A backreference counts subexpressions from left to right, starting with the opening parenthesis of each preceding subexpression. The expression is invalid if the source string contains fewer than *n* subexpressions preceding the **\n**.  Oracle supports the backreference expression in the regular expression pattern and the replacement string of the **REGEXP\_REPLACE** function. | The expression **(abc|def)xy\1** matches the strings **abcxyabc** and **defxydef**, but does not match **abcxydef** or **abcxy**.  A backreference enables you to search for a repeated string without knowing the actual string ahead of time. For example, the expression **^(.\*)\1$** matches a line consisting of two adjacent instances of the same string. |
| **\** | Escape Character | Treats the subsequent metacharacter in the expression as a literal. Use a backslash (\) to search for a character that is normally treated as a metacharacter. Use consecutive backslashes (**\\**) to match the backslash literal itself. | The expression **\+** searches for the plus character (**+**). It matches the plus character in the string **abc+def**, but does not match **abcdef**. |
| **^** | Beginning of Line Anchor | Matches the beginning of a string (default). In multiline mode, it matches the beginning of any line within the source string. | The expression **^def** matches **def** in the string **defghi** but does not match **def** in **abcdef**. |
| **$** | End of Line Anchor | Matches the end of a string (default). In multiline mode, it matches the beginning of any line within the source string. | The expression **def$** matches **def** in the string **abcdef** but does not match **def** in the string **defghi**. |
| **[:*class*:]** | POSIX Character Class | Matches any character belonging to the specified POSIX character ***class***. You can use this operator to search for characters with specific formatting such as uppercase characters, or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported.  **Note:** In English regular expressions, range expressions often indicate a character class. For example, **[a-z]** indicates any lowercase character. This convention is not useful in multilingual environments, where the first and last character of a given character class may not be the same in all languages. Oracle supports the character classes in [Table 4-3](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#g692318) based on character class definitions in Globalization classification data. | The expression **[[:upper:]]+** searches for one or more consecutive uppercase characters. This expression matches **DEF** in the string **abcDEFghi** but does not match the string **abcdefghi**. |
| **[.*element*.]** | POSIX Collating Element Operator | Specifies a collating element to use in the regular expression. The ***element*** must be a defined collating element in the current locale. Use any collating element defined in the locale, including single-character and multicharacter elements. The **NLS\_SORT** initialization parameter determines supported collation elements.This operator lets you use a multicharacter collating element in cases where only one character would otherwise be allowed. For example, you can ensure that the collating element **ch**, when defined in a locale such as Traditional Spanish, is treated as one character in operations that depend on the ordering of characters. | The expression **[[.ch.]]** searches for the collating element **ch** and matches **ch** in string **chabc**, but does not match **cdefg**. The expression **[a-[.ch.]]** specifies the range **a** to **ch**. |
| **[=*character*=]** | POSIX Character Equivalence Class | Matches all characters that are members of the same character equivalence class in the current locale as the specified ***character***.  The character equivalence class must occur within a character list, so the character equivalence class is always nested within the brackets for the character list in the regular expression.  Usage of character equivalents depends on how canonical rules are defined for your database locale. Refer to the[*Oracle Database Globalization Support Guide*](http://www.oracle.com/pls/topic/lookup?ctx=db102&id=NLSPG041) for more information on linguistic sorting and string searching. | The expression **[[=n=]]** searches for characters equivalent to **n** in a Spanish locale. It matches both **N** and **ñ** in the string **El Niño**. |

|  |
| --- |
| **See Also:**  [*Oracle Database SQL Reference*](https://docs.oracle.com/cd/B19306_01/server.102/b14200/ap_posix.htm#SQLRF020) for syntax, descriptions, and examples of the **REGEXP** functions and conditions |

Regular Expression Operator Multilingual Enhancements

When applied to multilingual data, Oracle's implementation of the POSIX operators extends beyond the matching capabilities specified in the POSIX standard. [Table 4-3](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#g692318) shows the relationship of the operators in the context of the POSIX standard.

* The first column lists the supported operators.
* The second column indicates whether the POSIX standard for Basic Regular Expression (BRE) defines the operator.
* The third column indicates whether the POSIX standard for Extended Regular Expression (ERE) defines the operator.
* The fourth column indicates whether the Oracle Database implementation extends the operator's semantics for handling multilingual data.

Oracle Database lets you enter multibyte characters directly, if you have a direct input method, or use functions to compose the multibyte characters. You cannot use the Unicode hexadecimal encoding value of the form **\xxxx**. Oracle evaluates the characters based on the byte values used to encode the character, not the graphical representation of the character.

***Table 4-3 POSIX and Multilingual Operator Relationships***

| **Operator** | **POSIX BRE syntax** | **POSIX ERE Syntax** | **Multilingual Enhancement** |
| --- | --- | --- | --- |
| **\** | Yes | Yes | -- |
| **\*** | Yes | Yes | -- |
| **+** | -- | Yes | -- |
| **?** | -- | Yes | -- |
| **|** | -- | Yes | -- |
| **^** | Yes | Yes | Yes |
| **$** | Yes | Yes | Yes |
| **.** | Yes | Yes | Yes |
| **[ ]** | Yes | Yes | Yes |
| **( )** | Yes | Yes | -- |
| **{m}** | Yes | Yes | -- |
| **{m,}** | Yes | Yes | -- |
| **{m,n}** | Yes | Yes | -- |
| **\n** | Yes | Yes | Yes |
| **[..]** | Yes | Yes | Yes |
| **[::]** | Yes | Yes | Yes |
| **[==]** | Yes | Yes | Yes |

Perl-Influenced Extensions in Oracle Regular Expressions

[Table 4-4](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDDGICJ) describes Perl-influenced metacharacters supported in Oracle Database regular expression functions and conditions. These metacharacters are not in the POSIX standard, but are common at least partly due to the popularity of Perl. Note that Perl character class matching is based on the locale model of the operating system, whereas Oracle Database regular expressions are based on the language-specific data of the database. In general, a regular expression involving locale data cannot be expected to produce the same results between Perl and Oracle Database.

***Table 4-4 Perl-Influenced Extensions in Oracle Regular Expressions***

| **Reg. Exp.** | **Matches . . .** | **Example** |
| --- | --- | --- |
| **\d** | A digit character. It is equivalent to the POSIX class **[[:digit:]]**. | The expression **^\(\d{3}\) \d{3}-\d{4}$** matches **(650) 555-1212** but does not match **650-555-1212**. |
| **\D** | A non-digit character. It is equivalent to the POSIX class **[^[:digit:]]**. | The expression **\w\d\D** matches **b2b** and **b2\_** but does not match **b22**. |
| **\w** | A word character, which is defined as an alphanumeric or underscore (**\_**) character. It is equivalent to the POSIX class **[[:alnum:]\_]**. Note that if you do not want to include the underscore character, you can use the POSIX class **[[:alnum:]]**. | The expression **\w+@\w+(\.\w+)+** matches the string **jdoe@company.co.uk** but not the string **jdoe@company**. |
| **\W** | A non-word character. It is equivalent to the POSIX class **[^[:alnum:]\_]**. | The expression **\w+\W\s\w+** matches the string **to: bill** but not the string **to bill**. |
| **\s** | A whitespace character. It is equivalent to the POSIX class **[[:space:]]**. | The expression **\(\w\s\w\s\)** matches the string **(a b )** but not the string **(ab)**. |
| **\S** | A non-whitespace character. It is equivalent to the POSIX class **[^[:space:]]**. | The expression **\(\w\S\w\S\)** matches the string**(abde)** but not the string **(a b d e)**. |
| **\A** | Only at the beginning of a string. In multi-line mode, that is, when embedded newline characters in a string are considered the termination of a line, **\A** does not match the beginning of each line. | The expression **\AL** matches only the first **L** character in the string **Line1\nLine2\n**, regardless of whether the search is in single-line or multi-line mode. |
| **\Z** | Only at the end of string or before a newline ending a string. In multi-line mode, that is, when embedded newline characters in a string are considered the termination of a line, **\Z** does not match the end of each line. | In the expression **\s\Z,**the **\s** matches the last space in the string **L i n e \n**, regardless of whether the search is in single-line or multi-line mode. |
| **\z** | Only at the end of a string. | In the expression **\s\z**, the **\s** matches the newline in the string **L i n e \n**, regardless of whether the search is in single-line or multi-line mode. |
| **\*?** | The preceding pattern element 0 or more times (non-greedy). Note that this quantifier matches the empty string whenever possible. | The expression **\w\*?x\w** is "non-greedy" and so matches **abxc** in the string **abxcxd**. The expression **\w\*x\w** is "greedy" and so matches **abxcxd in the string abxcxd**. The expression **\w\*?x\w** also matches the string **xa**. |
| **+?** | The preceding pattern element 1 or more times (non-greedy). | The expression **\w+?x\w** is "non-greedy" and so matches **abxc** in the string **abxcxd**. The expression **\w+x\w** is "greedy" and so matches **abxcxd in the string abxcxd**. The expression **\w+?x\w** does not match the string **xa**, but does match the string **axa**. |
| **??** | The preceding pattern element 0 or 1 time (non-greedy). Note that this quantifier matches the empty string whenever possible. | The expression **a??aa** is "non-greedy" and matches **aa** in the string **aaaa**. The expression **a?aa** is "greedy" and so matches **aaa** in the string **aaaa**. |
| **{n}?** | The preceding pattern element exactly **n** times (non-greedy). In this case **{n}?** is equivalent to **{n}**. | The expression **(a|aa){2}?** matches **aa** in the string **aaaa**. |
| **{n,}?** | The preceding pattern element at least **n** times (non-greedy). | The expression **a{2,}?** is "non-greedy" and matches **aa** in the string **aaaaa**. The expression **a{2,}** is "greedy" and so matches **aaaa**a. |
| **{n,m}?** | At least **n** but not more than **m** times (non-greedy). Note that **{0,m}?** matches the empty string whenever possible. | The expression **a{2,4}?** is "non-greedy" and matches **aa** in the string **aaaaa**. The expression **a{2,4}** is "greedy" and so matches **aaaa**. |

The Oracle Database regular expression functions and conditions support the pattern matching modifiers described in [Table 4-5](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHFCFC).

***Table 4-5 Pattern Matching Modifiers***

| **Mod.** | **Description** | **Example** |
| --- | --- | --- |
| **i** | Specifies case-insensitive matching. | The following regular expression returns **AbCd**:  REGEXP\_SUBSTR('AbCd', 'abcd', 1, 1, 'i') |
| **c** | Specifies case-sensitive matching. | The following regular expression fails to match:  REGEXP\_SUBSTR('AbCd', 'abcd', 1, 1, 'c') |
| n | Allows the period (.), which by default does not match newlines, to match the newline character. | The following regular expression matches the string, but would not match if the **n** flag were not specified:  REGEXP\_SUBSTR('a'||CHR(10)||'d', 'a.d', 1, 1, 'n') |
| m | Performs the search in multi-line mode. The metacharacter **^** and **$** signify the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. | The following regular expression returns **ac**:  REGEXP\_SUBSTR('ab'||CHR(10)||'ac', '^a.', 1, 2, 'm') |
| **x** | Ignores whitespace characters in the regular expression. By default, whitespace characters match themselves. | The following regular expression returns **abcd**:  REGEXP\_SUBSTR('abcd', 'a b c d', 1, 1, 'x') |

Using Regular Expressions in SQL Statements: Scenarios

This section contains the following scenarios:

* [Using an Integrity Constraint to Enforce a Phone Number Format](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDJAGEF)
* [Using Back References to Reposition Characters](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHCIGH)

Using an Integrity Constraint to Enforce a Phone Number Format

Regular expressions are a useful way to enforce integrity constraints. For example, suppose that you want to ensure that phone numbers are entered into the database in a standard format. [Example 4-1](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHIHII) creates a **contacts** table and adds a check constraint to the **p\_number** column to enforce the following format mask:

(XXX) XXX-XXXX

***Example 4-1 Enforcing a Phone Number Format with Regular Expressions***

CREATE TABLE contacts

(

l\_name VARCHAR2(30),

p\_number VARCHAR2(30)

CONSTRAINT p\_number\_format

CHECK ( REGEXP\_LIKE ( p\_number, '^\(\d{3}\) \d{3}-\d{4}$' ) )

);

[Table 4-6](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDEIEIC) explains the elements of the regular expression.

***Table 4-6 Explanation of the Regular Expression Elements in***[***Example 4-1***](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHIHII)

| **Regular Expression Element** | **Matches . . .** |
| --- | --- |
| **^** | The beginning of the string. |
| **\(** | A left parenthesis. The backward slash (**\**) is an escape character that indicates that the left parenthesis following it is a literal rather than a grouping expression. |
| **\d{3}** | Exactly three digits. |
| **\)** | A right parenthesis. The backward slash (**\**) is an escape character that indicates that the right parenthesis following it is a literal rather than a grouping expression. |
| (space character) | A space character. |
| **\d{3}** | Exactly three digits. |
| **-** | A hyphen. |
| **\d{4}** | Exactly four digits. |
| **$** | The end of the string. |

[Example 4-2](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDBDIFJ) shows a SQL script that attempts to insert seven phone numbers into the **contacts** table. Only the first two **INSERT** statements use a format that conforms to the **p\_number\_format** constraint; the remaining statements generate check constraint errors.

***Example 4-2 insert\_contacts.sql***

-- first two statements use valid phone number format

INSERT INTO contacts (p\_number)

VALUES( '(650) 555-5555' );

INSERT INTO contacts (p\_number)

VALUES( '(215) 555-3427' );

-- remaining statements generate check contraint errors

INSERT INTO contacts (p\_number)

VALUES( '650 555-5555' );

INSERT INTO contacts (p\_number)

VALUES( '650 555 5555' );

INSERT INTO contacts (p\_number)

VALUES( '650-555-5555' );

INSERT INTO contacts (p\_number)

VALUES( '(650)555-5555' );

INSERT INTO contacts (p\_number)

VALUES( ' (650) 555-5555' );

/

Using Back References to Reposition Characters

As explained in [Table 4-2](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDIEGEI), back references store matched subexpressions in a temporary buffer, thereby enabling you to reposition characters. You access buffers with the **\n** notation, where **\n** is a number between 1 and 9. Each subexpression is contained in parentheses and is numbered from left to right.

[Example 4-3](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDICBBC) creates a **famous\_people** table and populates the **famous\_people.names** column with names in different formats.

***Example 4-3 Using Back References to Reposition Characters***

CREATE TABLE famous\_people

( names VARCHAR2(30) );

-- populate table with data

INSERT INTO famous\_people

VALUES ('John Quincy Adams');

INSERT INTO famous\_people

VALUES ('Harry S. Truman');

INSERT INTO famous\_people

VALUES ('John Adams');

INSERT INTO famous\_people

VALUES (' John Quincy Adams');

INSERT INTO famous\_people

VALUES ('John\_Quincy\_Adams');

COMMIT;

[Example 4-4](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHCFCD) shows a query that repositions names in the format "first middle last" to the format "last, first middle". It ignores names not in the format "first middle last".

***Example 4-4 Using Back References to Reposition Characters***

SELECT names "names",

REGEXP\_REPLACE(names,

'^(\S+)\s(\S+)\s(\S+)$',

'\3, \1 \2')

AS "names after regexp"

FROM famous\_people;

[Table 4-7](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDBIJBI) explains the elements of the regular expression.

***Table 4-7 Explanation of the Regular Expression Elements in***[***Example 4-4***](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHCFCD)

| **Regular Expression Element** | **Description** |
| --- | --- |
| **^** | Matches the beginning of the string. |
| **$** | Matches the end of the string. |
| **(\S+)** | Matches one or more non-space characters. The parentheses are not escaped so they function as a grouping expression. |
| **\s** | Matches a whitespace character. |
| **\1** | Substitutes the first subexpression, that is, the first group of parentheses in the matching pattern. |
| **\2** | Substitutes the second subexpression, that is, the second group of parentheses in the matching pattern. |
| **\3** | Substitutes the third subexpression, that is, the third group of parentheses in the matching pattern. |
| **,** | Inserts a comma character. |

[Example 4-5](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDJGEDH) shows the result set of the query in [Example 4-4](https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_regexp.htm#CHDHCFCD). The regular expression matched only the first two rows.

***Example 4-5 Result Set of Regular Expression Query***

names

------------------------------

names after regexp

------------------------------

John Quincy Adams

Adams, John Quincy

Harry S. Truman

Truman, Harry S.

John Adams

John Adams

John Quincy Adams

John Quincy Adams

John\_Quincy\_Adams

John\_Quincy\_Adams

Name

[: :] (Character Class) — Specifies a character class

Synopsis

Use [: and :] to enclose a character class name, for example: [:alpha:]. Character classes must be specified within bracket expressions, as in [[:alpha:]].

The following example uses the character class [:digit:] to match the digits in a ZIP code:

SELECT REGEXP\_SUBSTR(

'Munising MI 49862',

'[[:digit:]]{5}') zip\_code

FROM dual;

49862

In this example, we could just as well have used the pattern [0-9]{5}. However, in multilingual environments digits are not always the characters 0-9. The character class [:digit:] matches the English 0-9, the Arabic-Indic ٠–٩, the Tibetan ༠–༩, and so forth.

Table 1-5 describes the character class names recognized by Oracle. All names are case-sensitive.

Table 1-5. Supported character classes

Class Description

[:alnum:] Alphanumeric characters (same as [:alpha:] + [:digit:])

[:alpha:] Alphabetic characters only

[:blank:] Blank space characters, such as space and tab

[:cntrl:] Nonprinting or control characters

[:digit:] Numeric digits

[:graph:] Graphical characters (same as [:punct:] + [:upper:] + [:lower:] + [:digit:])

[:lower:] Lowercase letters

[:print:] Printable characters

[:punct:] Punctuation characters

[:space:] Whitespace characters such as space, form-feed, newline,carriage return, horizontal tab, and vertical tab

[:upper:] Uppercase letters

[:xdigit:] Hexadecimal characters

SQL REGEXP\_LIKE() Function

SQL REGEXP\_LIKE() function same as like condition but matching regular expression pattern to perform like condition.

SQL REGEXP\_LIKE() function supported Oracle SQL version

Syntax

REGEXP\_LIKE(original\_string, pattern [ , match\_param ] )

Parameters

*original\_string* is a string which we want to represent in regular expression pattern.

*pattern* is a regular expression pattern.

*match\_param* is a expression flag.

***i* - ignore case**

***c* - case sensitive**

***n* - match any character as well as match newline character**

***m* - multi line**

***x* - ignore whitespace in match-patterns**

Example

Consider following example is REGEXP\_LIKE function fetching 'Opa?l' regular expression pattern from the name.

SQL> SELECT employee\_name

FROM emp\_info

WHERE REGEXP\_LIKE (name, 'Opa?l', 'im');

employee\_name

-------------

Opal

Opl

Regular Expression References

Following are regular expressions operator that are create patterns for letter use either string replacing or getting sub string from the string using regular expression pattern.

Flags Reference

You can also specify optional regular expression flags. Flags that allow for global searching, case insensitive searching. These flags you can define either separately or together.

|  |  |
| --- | --- |
| Flags | Description |
| i | ignore case |
| c | case sensitive |
| n | match any character as well as match newline character |
| m | multi line |
| x | ignore whitespace |

Quantifiers/Alternative Classes

|  |  |
| --- | --- |
| Character | Description |
| . | Any character except newline Example. . - Matches any character |
| \* | Matches O or more preceding character Example. b\* - bbbeee |
| + | Matches 1 or more preceding character Example. b+ - bbbeee, beee |
| ? | Matches either 0 or 1 preceding character, effectively matches is optional Example. Goog?le - Goole , Google |
| | | Represent like a boolean OR for alternative matches Example. AB|CD - match ab or cd |

Grouping Classes

|  |  |
| --- | --- |
| Character | Description |
| [ ] | Matches any character in the set Example. [ABC] - matches any of a, b, or c |
| ( ) | Capture groups of sequence character together Example. (name) - matches sequence of group character |

Ranging Classes

|  |  |
| --- | --- |
| Character | Description |
| {a} | matches exactly m time Example. b{1} - match exactly 1 time |
| {a,} | matches exactly m or more time Example. b{1,} - match exactly 1 or more time |
| {a, z} | matches m to n times Example. b{3,5} - match between 3 & 5 |

Escape Character Classes

|  |  |
| --- | --- |
| Character | Description |
| \ | specified the next special character Example. \\ - Matches a "\" character. |
| \n | Matches a n number (digit 1 to 10) LINE FEED character enclosed between ( and ). |

Anchors Classes

|  |  |
| --- | --- |
| Character | Description |
| ^ | Beginning of the string. If more then one line matches any beginning line. Example. ^ABC - starting character A then match ABC |
| $ | Ending of the string. If more then one line matches any ending line. Example. ABC$ - ending character C then match ABC |
| \A | Matches only at the beginning of the string. Example. h\A - hello Opal! (matches only 'hello') |
| \Z | Matches only at the ending of the string. Example. o\A - hello Opal! (matches only 'hello') |

Character Classes

|  |  |
| --- | --- |
| Character | Description |
| \d | Matches digit character Example. \d - Hello123 (matches only '123') |
| \D | Matches non digit character Example. \d - Hello123 (matches only 'Hello') |
| \w | Matches word character Example. \w - Hello123###/\* (matches only 'Hello123') |
| \W | Matches non word character Example. \W - Hello123###/\* (matches only '###/\*') |
| \s | Matches whitespace Example. \s - Hello 123 ### (matches only whitespace) |
| \S | Matches non whitespace Example. \S - Hello 123 ### (matches non whitespace 'Hello' and '123' and '###') |

**https://www.salvis.com/blog/2018/09/28/regular-expressions-sql-examples/**

Regular Expressions in SQL by Examples

 2018-09-28



Are you reluctant to use regular expressions in SQL? Then continue reading. Examples helped me to understand regular expressions years ago. Thus I hope this collection of simple examples and the tooling tips will encourage you to use regular expressions. It’s not as complicated as it looks at first glance. Once you get used to the syntax, it’s fun to figure out the right match pattern.

Use Cases in SQL

The Oracle Database supports regular expression since version 10g Release 1. You may use it to:

1. Validate an input using [regexp\_like](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/Pattern-matching-Conditions.html#GUID-D2124F3A-C6E4-4CCA-A40E-2FFCABFD8E19);
2. Find patterns in text using [regexp\_count](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/REGEXP_COUNT.html#GUID-5148AF2E-9CED-497D-A78D-3A7847A45276), [regexp\_instr](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/REGEXP_INSTR.html#GUID-D21B53A1-83E2-4722-9BBB-638470715DD6) and [regexp\_substr](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/REGEXP_SUBSTR.html#GUID-2903904D-455F-4839-A8B2-1731EF4BD099);
3. Find and replace patterns in text using [regexp\_replace](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/REGEXP_REPLACE.html#GUID-EA80A33C-441A-4692-A959-273B5A224490).

Finding text using regular expressions is known as pattern matching. Those who understand regular expressions will quickly find their way around [row pattern matching](https://docs.oracle.com/en/database/oracle/oracle-database/18/dwhsg/sql-pattern-matching-data-warehouses.html#GUID-136DAC89-DA17-45C6-9E37-C9892723AC79), since the pattern syntax is very similar.

The Text

All examples use this famous quote from Henry Ford:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

All matches are highlighted.

Single Character

The simplest match pattern (regular expression without match parameters) is a single character. There are some characters with a special meaning such as ., \, ?, \*, +, {, }, [, ], ^, $, |, (, ). We deal with these characters later. However, as long as you do not use one of these characters, the match pattern behaves like the substring parameter in the well-known [instr](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/INSTR.html#GUID-47E3A7C4-ED72-458D-A1FA-25A9AD3BE113) function.

Match pattern: t returns 5 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

The following query produces a row per match. It can be used in the subsequent examples as well (with an adapted match pattern).

1) One row per match in SQL

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | WITH     base AS (        SELECT '"Whether you think you can or think you can''t - you are right."'               || chr(10) || '-- Henry Ford (1863 - 1947)' AS text,               't' AS pattern          FROM dual     )  -- main  SELECT regexp\_substr(text, pattern, 1, level) AS matched\_text,          regexp\_instr(text, pattern, 1, level) AS at\_pos     FROM base  CONNECT BY level <= regexp\_count(text, pattern);    MATCHED\_TEXT             AT\_POS  -------------------- ----------  t                             5  t                            14  t                            31  t                            45  t                            61 |

The named subquery base provides the text and the match pattern. This way the expressions do not have to be repeated. The regexp\_count function on line 12 limits the result to 5 rows. The regexp\_substr function call on line 9 returns the matched text and the regexp\_instr function call on line 10 the position.

Multiple Characters

A string is just a series of characters.

The match pattern thin returns 2 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Any Character Wildcard .

A dot . matches per default any character except newline chr(10).

The match pattern c.n returns 2 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Escape Character \

If we want to match special characters such as a dot . than we have to escape it with a \.

The match pattern \. returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

0..1 Matches (Optionality) ?

We use a ? to express that a character (or a group of characters) is optional.

The match pattern c?.n returns 5 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

You see that the c is part of a match in can, but h before in is not.

0..n Matches \*

We use a \* to express that a character (or a group of characters) can appear between 0 and n times. n is not defined and is in fact unbounded.

The match pattern you.\*n returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Please note that the first match was not you thin. Rather it was extended to the last n in the first line. This behavior is called **greedy**.

Nongreedy Matches ?

We use a ? at the end of quantifier (?, \*, +, {}) to match as few as possible.

The match pattern you.\*?n returns 3 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Please note that we now have three matches. This behavior is called **nongreedy** or **reluctant** or **lazy**.

1..n Matches +

We use a + to express that a character (or a group of characters) can appear between 1 and n times. n is not defined and is in fact unbounded.

The match pattern -+ returns 3 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Exact Match {n}

We use {n} to express that a character (or a group of characters) must appear exactly n times.

The match pattern -{2} returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Match Ranges {m,n}

We use {m,n} to express that a character (or a group of characters) must appear between m and n times. You may skip the definition for n to express an unbounded value.

The match pattern -{1,3} returns 3 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Alphanumeric Wildcard \w

A \w matches any alphanumeric character.

The match pattern \w+ returns 17 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Non-alphanumeric Wildcard \W

A \W matches any non-alphanumeric character. Please note that the match pattern is case-sensitive. The upper case letter W leads to the opposite result than the lower case letter w. This is an essential principle for match patterns.

The match pattern \W+ returns 18 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that the newline chr(10) is part of match 14.

Digit Wildcard \d

A \d matches any digit (0 to 9).

The match pattern \d+ returns 2 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Non-digit Wildcard \D

A \D matches any non-digit character. Please note that the match pattern is case-sensitive. The upper case letter D leads to the opposite result than the lower case letter d. This is an essential principle for match patterns.

The match pattern \D+ returns 3 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that the newline chr(10) is part of the first match.

Whitespace Wildcard \s

A \s matches any whitespace character. **Whitespaces are:**

* *spaces chr(32)*
* *horizontal tabs chr(9)*
* *carriage returns chr(13)*
* *line feeds/newlines chr(10)*
* *form feeds chr(12)*
* *vertical tabs chr(11)*

The match pattern \s+ returns 18 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that the match 13 is a newline chr(10).

Non-whitespace Wildcard \S

A \S matches any non-whitespace character. Please note that the match pattern is case-sensitive. The upper case letter S leads to the opposite result than the lower case letter s. This is an essential principle for match patterns.

The match pattern \S+ returns 19 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Character Class [xyz]

A character class is a list of characters defined within brackets. You can also use a hyphen - to specify a range of characters. For example [0-9] which is equivalent to \d. You can combine ranges and single characters.

The match pattern [a-zA-Z']+ returns 14 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Negated Character Class [^xyz]

A negated character class matches all characters that are not defined within brackets. A ^ at the first position within the brackets defines a negated character class.

The match pattern [^a-zA-Z']+ returns 15 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that the newline chr(10) is part of match 13.

Beginning of Line or String ^

A ^ matches the position before the first character within a line or string. By default a text is treated as a string.

The match pattern ^- returns 0 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that by default the whole text is treated as a single line. Hence ^ means beginning of string. And the string starts with a " and not with a -. Therefore no matches.

Multiline Mode m

A regular expressions has two parts. The first part is the match pattern. The second part are match parameters. Until now we have not defined match parameters, hence the default has been used. The match parameter m will logically change the text from a single line to an array of lines.

The match pattern ^- with the match parameter m returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

The next query produces a row per match as the query above, but applies the match parameter m.

2) One row per match in SQL with match parameter

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | WITH     base AS (        SELECT '"Whether you think you can or think you can''t - you are right."'               || chr(10) || '-- Henry Ford (1863 - 1947)' AS text,               '^-' AS pattern,               'm' AS param          FROM dual     )  -- main  SELECT regexp\_substr(text, pattern, 1, level, param) AS matched\_text,          regexp\_instr(text, pattern, 1, level, 0, param) AS at\_pos     FROM base  CONNECT BY level <= regexp\_count(text, pattern, 1, param);    MATCHED\_TEXT             AT\_POS  -------------------- ----------  -                            65 |

The match parameter is defined on line 6. The regex\_substr function call on line 10 and the regex\_instr function call on line 11 get this match parameter as an additional input.

You may use this query with adapted match pattern and match parameters to reproduce the results of the subsequent examples.

End of Line or String $

A $ matches the position after the last character within a line or string. By default a text is treated as a string.

The match pattern "$ with the match parameter m returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Ignore Case Mode i

Use the match parameter i for case-insensitive matches.

The match pattern he with the match parameter i returns 3 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Case-sensitive Mode c

Use the match parameter c for case-sensitive matches. This is the default. However, when NLS\_SORT is set to a case-insensitive sort order – e.g. BINARY\_CI, GENERIC\_M\_CI, FRENCH\_M\_CI, etc. – then the default changes to case-insensitive matches.

The match pattern he with the match parameter c returns 2 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Period Matches Newline Mode n

Use the match parameter n to change the behavior of the any character wildcard . to match newlines chr(10) as well.

The match pattern .+ with the match parameter n returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Ignore Whitespace in Pattern Mode x

Use the match parameter x to ignore whitespaces in match patterns. For long match patterns it might be helpful to add spaces, tabs and newlines to make the regular expressions more readable. By default these whitespaces are considered to be part of the match pattern. To ignore them you have to use the x mode. However, whitespaces in brackets are always considered, e.g. [ ].

The match pattern h  e nr y with the match parameters ix returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

Please note that multiple match parameters (i and x) are used.

Alternatives |

Use a | to express alternative options. The number of options is not limited. The order of the options corresponds to the priority.

The match pattern  think|can't|can returns 4 matches:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

It’s important to note that the order of the options matter in this case. The match pattern think|can|can't would never match can't. Furthermore, to avoid redundancies in match patterns you would use groups. For example think|can('t)?.

Numbered Groups (xyz)

Use parenthesis – ( and ) – to define groups. You may nest groups as well. The complete match pattern is group 0. All other (sub-)groups are numbered from left to right. You may simply count the number of open parenthesis in a match pattern up to the cursor position of a group to determine the group number.

The match pattern ^("|')(.+)(\1)\s+--\s+(\w+)\s+(\w+)\s+(\((\d+)\s\*-\s\*(\d+)\))$ returns 1 match:

*"Whether you think you can or think you can't - you are right."  
-- Henry Ford (1863 - 1947)*

The matches for the groups are:

* 0=full match (as shown above)
* 1="
* 2=Whether … right.
* 3="
* 4=Henry
* 5=Ford
* 6=(1863 - 1947)
* 7=1863
* 8=1947

Please note that the group 3 in the match pattern is referencing the result of the group 1 ("). This means a quote starting with ' must end on ' and a quote starting with " must end on ".

The next query produces a row per group.

3) One row per group in SQL

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | WITH     base AS (        SELECT '"Whether you think you can or think you can''t - you are right."'               || chr(10) || '-- Henry Ford (1863 - 1947)' AS text,               '^("|'')(.+)(\1)\s+--\s+(\w+)\s+(\w+)\s+(\((\d+)\s\*-\s\*(\d+)\))$' AS pattern          FROM dual     )  -- main  SELECT level-1 AS group\_no,          regexp\_substr(text, pattern, 1, 1, null, level-1) AS matched\_group\_text     FROM base  CONNECT BY level <= regexp\_count(pattern, '[^\\]?\(') + 1;    GROUP\_NO MATCHED\_GROUP\_TEXT  -------- ----------------------------------------------------------------         0 "Whether you think you can or think you can't - you are right."           -- Henry Ford (1863 - 1947)           1 "         2 Whether you think you can or think you can't - you are right.         3 "         4 Henry         5 Ford         6 (1863 - 1947)         7 1863         8 1947    9 rows selected. |

The regex\_substr function call on line 10 gets the group number as last input parameter.

Tooling

The match pattern used in the previous example is not that easy to read. Hence I recommend to use some tools to build regular expressions. These tools provide quick references and libraries for common regular expressions. And of course they provide features to test regular expressions and show matches. But they also can explain a regular expression in detail. Here are three of them:

* [Expresso](http://www.ultrapico.com/expresso.htm) is a longtime, reliable companion of mine. This tool has helped me to build and understand many regular expressions. It runs under Windows, is free, but requires a registration.
* [regular expressions 101](https://regex101.com/) is popular online regular expressions tester and debugger.
* [RegExr](https://regexr.com/) is another popular online tool to learn, test and build regular expressions.

Here’s a screenshot of Expresso showing the match results and some explanation of the regular expression.

[Afbeelding met tekst

Automatisch gegenereerde beschrijving](https://www.salvis.com/blog/wp-content/uploads/2018/09/expresso3.png)

It’s important to note that the [regular expressions in the Oracle Database](https://docs.oracle.com/en/database/oracle/oracle-database/18/sqlrf/Oracle-Regular-Expression-Support.html#GUID-969230D6-FC1A-4C75-BF2A-6B1BE909DED6) conforms to [POSIX](http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html) with a few extensions influenced by [PCRE](http://www.pcre.org/current/doc/html/pcre2syntax.html). So these tools support regular expression features which are not available in Oracle SQL. I miss for example non-capturing groups, lookaheads and some escaped characters (\r, \n, \t, etc.).

Summary

Regular expressions are not self-explanatory. In this post I covered most of the regular expressions grammar that is applicable in SQL functions of an Oracle Database.

* Strings: t, thin
* Greedy quantifiers: ?, \*, +, {2}, {1, 3}
* Nongreedy quantifiers: ??, \*?, +?, {2}?, {1, 3}?
* Character classes: ., \., \w, \W, \d, \D, \s, \S, [a-z], [^a-z]
* Positions: ^, $
* Alternatives: |
* Numbered groups: (abc), \1, \2, …, \9
* Match parameters: m, i, c, n, x

With a basic knowledge of regular expressions the available tooling make building, testing and understanding regular expressions quite easy.