Regexp Expression: META Characters, POSIX Characters, Perl backslash-sequences.

What are Regular Expressions?

- Methods of describing both simple and complex patterns for searching and manipulating
- Uses META characters `_ $ , * @ ?` for pattern matching
- It is Case Sensitive.

Perl backslash-sequences

- `\d`: decimal digit (0-9).
- `\D`: not a decimal digit.
- `\s`: whitespace character.
- `\S`: not a whitespace character.
- `\w`: “word” character (a-z, A-Z, underscore (“_”), or decimal digit).
- `\W`: not a word character.
- `\b`: word boundary.
- `\B`: not a word boundary.

ESCAPE SEQUENCES

Escape Sequence	Meaning	Representation
`\N`	Null character	`NULL`
`\B`	Backspace character	`BACKSPACE`
`\T`	Horizontal Tab character	`TAB`

`\S`	Space character	SPACE

EXAMPLE - `san\\b` this pattern will match san as one word like san dieago, it will not match character it with like santa, sansu etc.

Here we use `\\` because first `\` is to escape the character then `\b` which act as backspace. We need first `\` so that second `\` can be read with `b` and represent as backspace.

META CHARACTERS

Symbol	Description
^	Marks the start of a line
$	Marks the end of a line
[ ]	Matching list
|	Operator for specifying alternative matches (logical OR)
?	Matches zero or one occurrence
.	Matches any character except NULL
{m}	Matches exactly *m* times
{m,n}	Matches at least *m* times but no more than *n* times
[: :]	Specifies a character class and matches any character in the class
\	Escape character
+	Matches one or more occurrences
*	Matches zero or more occurrences
( )	Grouping for expression
\n	Back-reference expression

`()` - Grouping For Expression

- The characters `()` are used to define a capturing group. They are used to group together a sequence of characters and capture the matched text for further processing.
- Example - ^(Oracle), (ab|cd) etc.

`[]` - Matching List

- The characters `[]` are used to define a character class in a regular expression. A character class represents a set of characters from which one character can match.
- Example - The pattern `[abc]` will match any single occurrence of the characters 'a', 'b', or 'c'.
- Character classes can also include ranges of characters by using a hyphen. For instance, `[a-z]` will match any lowercase letter from 'a' to 'z'. They can also include predefined character sets such as `digits ("\d")`, `letters ("\w")`, or `whitespace ("\s")`.
- It's worth mentioning that within a character class, certain characters may have special meaning, such as the caret `^` when used as the first character, which indicates `negation` (e.g., "[^0-9]" matches any character that is not a digit).

SIMPLE WAY TO UNDERSTAND `[]` & `()`

Square brackets (`[]`):

- Character Class: Square brackets are used to define a character class, which represents a set of characters to match against. For example, `[abc]` matches any single character that is either "a", "b", or "c".

Parentheses (`()`):

- Grouping: Parentheses are used for grouping parts of a regular expression together. They establish a subexpression or capture group, allowing you to apply quantifiers or refer to the captured content. For example, `(abc){2}` matches the sequence "abcabc" by applying the `{2}` quantifier to the grouped subexpression "abc".

`^` - Beginning of a line or you can say starting from this.

- Word inside it is case sensitive - the way you write Oracle, it will check in that format. If you write oRacLe then it will check in this format.

Example: ^(Oracle)	**Output**
Oracle Open World	√ This is correct as Oracle comes in Starting.

Example: ^(Oracle)	Output
The Oracle at Delphi	**X** This is wrong as Oracle comes in Second.
Oracle	√ This is correct as Oracle comes in Starting

`$` - End of a Line

Example: (Oracle)$	Output
Welcome to Oracle	√
The Oracle at Delphi	**X**
Oracle	√

`|` - Logical OR - either this or this.

Example: Ste(v|ph)en	Output
Stephen	√
Stefan	**X**
Steven	√

`.` - Single character match

Example: re.d	Output
read	√
rear	**X**
reed	√

- Here `r e . d` in place of `.` any single character can come. From a-z.
- And Output will only match when it contain all these 3 character `r` `e` & `d` with the any single character replacing `.`

Breakdown (re.d)

Character	Check	Success
r	**r** e a d **r** e a r **r** e e d	√ √ √
e	r **e** a d r **e** a r r **e** e d	√ √ √
. (A-Z,a-z)	r e **a** d r e **a** r r e **e** d	√ √ √

Character	Check	Success
d	r e a **d** r e a **r** r e e **d**	√ **X** √

`{m}` - Matches exactly m times.

Example: s{2}	Output
password	√
sister	**X**
essential	√

- The pattern "s{2}" in a regex will match only those 's' characters that appear consecutively, without any other character in between. It will count occurrences of the letter 's' that are continuous and not separated by any other character.

Breakdown (s{2}) - It is checking for s character coming 2 times continuously.

password	Check	sister	Check
p	X	s	X
a	X	i	X
s	√	s	X
s	√	t	X
w	X	e	X
o	X	r	X
r	X		
d	X		

`*` - Matches zero or more occurrences
`+` - Matches One or more occurences - means atleast 1 or more.

Example: ab*c	Match
abc	√
acc	**X**
ac	√

Example 1

```
Pattern: a*

This regex will match any string that starts with zero or more occurrences
of the letter "a". So it would match strings like "a", "aa", "aaa", and so on.
```

Example 2

```
Pattern: .*regex

".*" matches zero or more occurrences of any character, and "regex" matches
the literal string "regex". So, this regex will match any string that contains
the word "regex" preceded by any number of characters. For example, it would match
"this is a regex", "123regex", "regex is awesome", and so on.
```

Example 3

```
Pattern: a.*

This regex will match any string that starts with the letter "a" followed by
zero or more occurrences of any character. It would match
strings like "a", "apple", "a123", "abcd", and so on.
```

`\` - ESCAPE CHARACTER

- In Regex, meta-characters (i.e. characters that have a special meaning) are "escaped" using a backslash. This means that Regex will treat that character as its literal version rather than a meta-character ().
- Example: `\.` , `\+`
- If you saw `\\` it means space just like `\\s` in snowflake.

POSIX Character

POSIX character classes are a set of predefined character classes commonly used in regular expressions

1. `[:alnum:]`: Matches any alphanumeric character. It is equivalent to [A-Za-z0-9].
2. `[:alpha:]`: Matches any alphabetic character. It is equivalent to [A-Za-z].

3. `[:digit:]`: Matches any digit character. It is equivalent to [0-9].
4. `[:lower:]`: Matches any lowercase alphabetic character. It is equivalent to [a-z].
5. `[:upper:]`: Matches any uppercase alphabetic character. It is equivalent to [A-Z].
6. `[:space:]`: Matches any whitespace character, including spaces, tabs, and line breaks.
7. `[:punct:]`:` Matches any punctuation character.
8. `[:graph:]`: Matches any visible character (excluding whitespace).
9. `[:print:]`: Matches any printable character (including whitespace).
10. `[:cntrl:]`: Matches any control character (non-printable characters).

Replacement of POSIX Characters

1. `[:alnum:]` can be replaced with [A-Za-z0-9] to match alphanumeric characters.
2. `[:alpha:]` can be replaced with [A-Za-z] to match alphabetic characters.
3. `[:digit:]` can be replaced with [0-9] to match digit characters.
4. `[:lower:]` can be replaced with [a-z] to match lowercase alphabetic characters.
5. `[:upper:]` can be replaced with [A-Z] to match uppercase alphabetic characters.
6. `[:space:]` can be replaced with [\t\n\r\f\v] to match whitespace characters, including spaces, tabs, line breaks, and other whitespace characters.
7. `[:punct:]` typically does not have a direct replacement, but you can create a custom character class to match specific punctuation characters as needed.
8. `[:graph:]` can be replaced with [\x21-\x7E] to match visible characters (excluding whitespace).
9. `[:print:]` can be replaced with [\x20-\x7E] to match printable characters (including whitespace).
10. `[:cntrl:]` can be replaced with [\x00-\x1F\x7F] to match control characters (non-printable characters).

Example

Q. Reformat phone number from ###.###.#### to 1 (###)-###-####

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

OUTPUT
404.777.9311 ---> 1 (404)-777-9311
```

Meta Character	Description
[[:digit:]]{3}	Three digits (group 1)
\.	Then a '.' (Since the '.' is a META Character, we have to use the \ to 'escape' it)
[[:digit:]]{3}	Three digits (group 2)
\.	Then a '.'
[[:digit:]]{4}	Four digits (group 3)

Replacements	Description (sample 404.777.9311)
1	Start with a 1
(\1)	Enclose group 1 in () -> (404)
-	Add a '-'
\2	Group 2 -> 777
-	Add a '-'
\3	Group 3 -> 9311
RESULT	1 (404)-777-9311

Reason that we use `[[]]` double square bracket, because `[:digit:]` contain no. from 0-9 if we don't use another bracket then it will take only 0 as digit and do its operation. So we use another bracket to make these no. 0-9 into list, so that any number can be counted.

WORKING OF BACKREFERENCES

1. Replacing Repeated Characters:
   Text: "Helloo, Worldd!"
   Pattern: "(.)\1+"
   Replacement: "\1"
   Output: "Helo, World!"

In this example, the pattern `(.)\1+` matches any character followed by one or more repetitions of the same character. The backreference `\1` refers to the captured character, and the replacement string `\1` replaces the repeated characters with a single occurrence.

2. Swapping Word Order:
   Text: "John Smith"
   Pattern: "(.?)\s(.?)"
   Replacement: "\2 \1"
   Output: "Smith John"

   In this example, the pattern `(.*?)\s(.*?)` captures the first and last names as separate groups. The backreferences `\1` and `\2` are used in the replacement string to swap the order of the names.

3. Extracting Date Components:
   Text: "2023-07-04"
   Pattern: "(\d{4})-(\d{2})-(\d{2})"
   Replacement: "Year: \1, Month: \2, Day: \3"
   Output: "Year: 2023, Month: 07, Day: 04"

   In this example, the pattern `(\d{4})-(\d{2})-(\d{2})` captures the year, month, and day components of a date. The backreferences `\1`, `\2`, and `\3` are used in the replacement string to format the extracted date components

How to search for a blank followed by a backslash

```
select v, v regexp '.*\\s\\\\.*'AS MATCHES from strings;

-- It gives true
-- but when in this format .*\\s\\.* then answer is flase
```

```
+-------------------------------------+---------+
| V                                   | MATCHES |
|-------------------------------------+---------|
```

```
| Contains embedded single \backslash | True    |
| Sacramento                          | False   |
| San Francisco                       | False   |
| San Jose                            | False   |
| Santa Clara                         | False   |
+-------------------------------------+---------+
```