Learn Regular Expression

▼ What is Regular Expression?

A
Regular Expression (RegEx) is a sequence of characters that defines a search pattern. It is primarily used for searching, matching, and manipulating text in strings. Regular expressions are powerful tools that allow for complex string pattern recognition, including matching specific sequences, searching for particular types of characters, or validating formats like email addresses, phone numbers, or dates.

▼ What is sequence of characters in Regex ?

A **regular expression** consists of a series of characters (letters, digits, symbols) and special operators. These characters can be literal, meaning they match exactly as they appear, or they can be symbolic, representing a broader set of possible characters or patterns.

- **Literal characters:** Regular letters and numbers that stand for themselves in a pattern. For instance, in the regex pattern `"abc"`, each character represents itself, so it will only match the string "abc".

- **Special (Meta) characters:** These have a special meaning in regular expressions and allow for more flexible, dynamic pattern matching. For example, a period (`.`) in a regex means "match any character."

▼ What is search pattern in Regex ?

> A **search pattern within text data** refers to a specific set of rules or criteria defined to locate and match particular sequences of characters (words, numbers, symbols, etc.) in a block of text. This pattern helps identify whether the text contains certain elements or structures, based on the defined rules, rather than looking for exact matches.

For example, a search pattern might say:

- "Find any text that starts with 'A' and ends with 'e'."
- "Find a sequence of exactly 5 digits."
- "Find all words that contain both 't' and 'e'."

▼ What is types of search pattern ?

▼ Search Patterns Can Be Specific or General:

- **Specific patterns** match very narrowly defined strings.

    - Example: The pattern `"apple"` only matches the exact word "apple".

- **General patterns** use special symbols (called **metacharacters**) to match a broader set of strings.
    - Example: The pattern `\d{3}` matches any sequence of three digits, like "123", "456", or "789".

▼ Fundamental element of regular expressions :

▼ Basic Matchers :

> **Basic matchers** are fundamental elements that allow you to search for and match specific sequences of characters in text. Understanding these basic matchers is essential for constructing effective regular expressions.

1. Literals

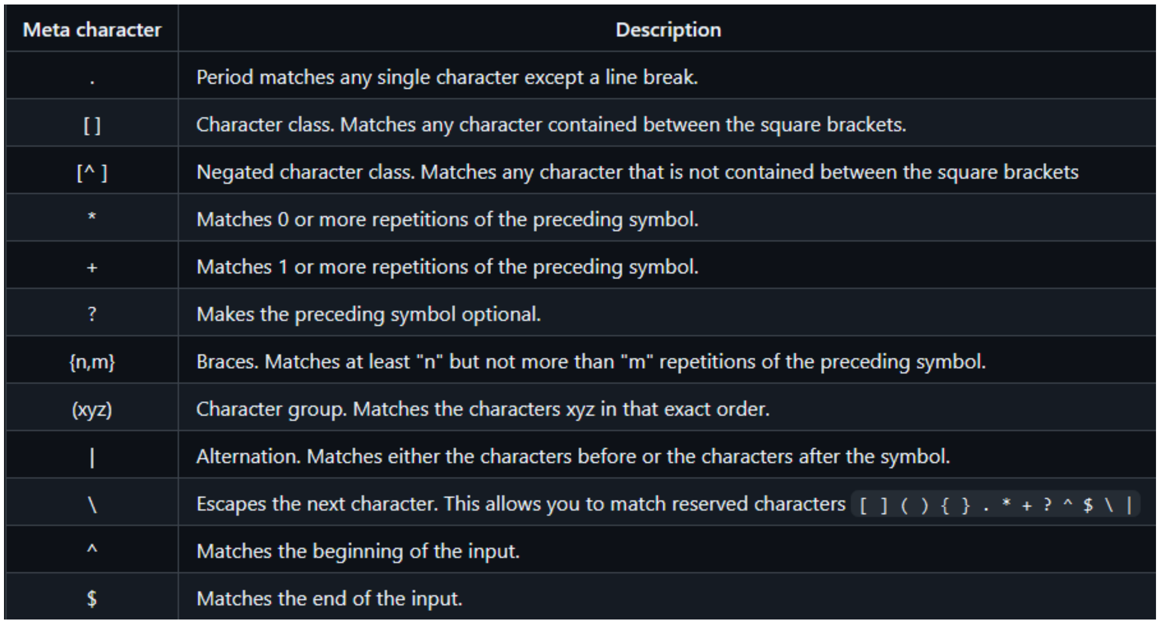
- **Definition:** Literal characters are the most straightforward type of matchers in regex. They match the exact character(s) as they appear in the text.

- **Example:**
    - Pattern: `cat`
    - Matches: The string "cat", but not "catalog" or "cater".

Try another example by your hands .

▼ Meta Characters :

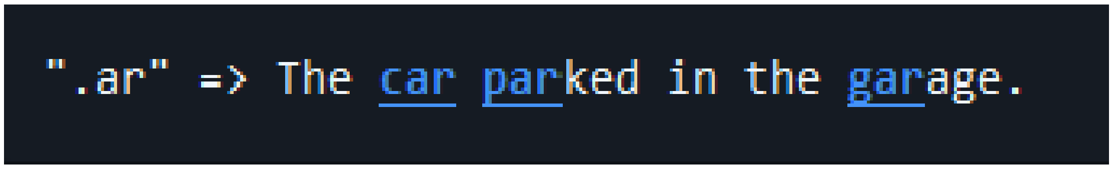
> **Metacharacters** are special characters in regular expressions (RegEx) that have specific meanings and functionalities, allowing users to construct complex search patterns. Unlike regular characters, which represent themselves, metacharacters enable operations such as matching multiple characters, specifying the position of matches, defining repetitions, and grouping elements. They play a crucial role in text processing, pattern matching, and data validation.



Meta character	Description
.	Period matches any single character except a line break.
[ ]	Character class. Matches any character contained between the square brackets.
[^]	Negated character class. Matches any character that is not contained between the square brackets
*	Matches 0 or more repetitions of the preceding symbol.
+	Matches 1 or more repetitions of the preceding symbol.
?	Makes the preceding symbol optional.
{n,m}	Braces. Matches at least "n" but not more than "m" repetitions of the preceding symbol.
(xyz)	Character group. Matches the characters xyz in that exact order.
|	Alternation. Matches either the characters before or the characters after the symbol.
\	Escapes the next character. This allows you to match reserved characters `[ ] ( ) { } . * + ? ^ $ \ |`
^	Matches the beginning of the input.
$	Matches the end of the input.

▼ The Full Stop :

The full stop
`.` is the simplest example of a meta character. The meta character `.` matches any single character. It will not match return or newline characters.
For example, the regular expression
`.ar` means: any character, followed by the letter
`a`, followed by the letter `r`.



```
".ar" => The car parked in the garage.
```

>>Test Regex Here<<

Character Sets :

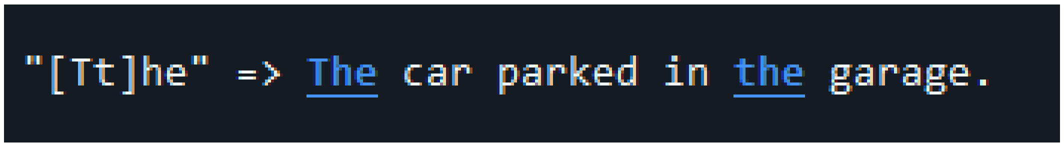
> **Character Sets** (also known as **character classes**) in regular expressions are a way to define a group of characters that can be matched at a specific position in a string. By enclosing characters in square brackets `[ ]`, you create a **character set**, allowing the regex engine to match any one of the characters inside the brackets. This provides greater flexibility for pattern matching because a single position in the pattern can match multiple characters.

Basic Character Set :

> Character sets are also called character classes. Square brackets are used to specify character sets. Use a hyphen inside a character set to specify the characters' range. The order of the character range inside the square brackets doesn't matter.

For example, the regular expression `[Tt]he` means: an uppercase

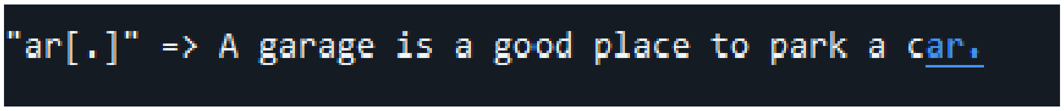
`T` or lowercase `t`, followed by the letter `h`, followed by the letter `e`.



```
"[Tt]he" => The car parked in the garage.
```

>>Test Regex Here<<

A period inside a character set, however, means a literal period. The regular
expression
`ar[.]` means: a lowercase character `a`, followed by the letter `r`, followed by a period
`.` character.



```
"ar[.]" => A garage is a good place to park a car.
```

>>Test Regex Here<<

▼ Character Ranges :

> Ranges of characters can be specified using a hyphen (`-`) inside square brackets. This allows you to match any character within a defined range.

- **Syntax:** `[a-z]` matches any lowercase letter from `a` to `z`.

▼ For Example :

Pattern:

```
[a-z]
```

Text:

```
"h3ll0 world"
```

Steps:

1. The first character in the text is "h" → it matches because "h" is a lowercase letter.
2. The next character "3" is skipped since it's not a lowercase letter.
3. "l", "l", and "o" are lowercase letters, so they match.
4. After a space, "w", "o", "r", "l", and "d" also match because they are lowercase letters.

- Syntax: `[A-Z]` m**atches:** any uppercase letter from `A` to `Z`.

▼ **For Example :**

Pattern:

```
[A-Z]
```

Text:

```
"Hello World, Welcome to 2024!"
```

Steps:

1. The text starts with "H" → it matches because "H" is an uppercase letter.
2. "e", "l", "l", and "o" are lowercase, so they are skipped.
3. "W" matches because it's uppercase.
4. "o", "r", "l", and "d" are lowercase, so they are skipped.
5. The uppercase "W" in "Welcome" matches.
6. "e", "l", "c", "o", and "m" are skipped (lowercase letters).
7. The uppercase "T" in "to" matches.
8. The numbers and special characters like space and punctuation are skipped.

- **Syntax:** `[0-9]` matches any digit from `0` to `9`.

▼ For Example :

Pattern:

```
[0-9]
```

Text:

```
"Phone number: 123-456-7890"
```

Steps:

1. The text starts with "Phone number:" which has no digits, so they are ignored.
2. The digits "1", "2", and "3" match.
3. The digits "4", "5", and "6" match after the hyphen.
4. Finally, "7", "8", "9", and "0" match after another hyphen.

▼ Combining Ranges :

- Multiple ranges can be combined within a character set.

- **Syntax:** `[a-zA-Z0-9]` matches any letter (uppercase or lowercase) or digit.

▼ For example :

Pattern:

```
[a-zA-Z0-9]
```

- **Matches:** any letter or digit.
- **Text:** "H3ll0W0rld"
- It matches any single alphabetic or numeric character like "H", "3", "l", etc.

▼ Negated Character Sets :

> Negated character sets, also called **negative character classes**, allow you to match any character **except** those specified inside the square brackets. To define a negated character set, you use the caret (`^`) symbol as the **first character** inside the square brackets.

This feature is powerful when you want to exclude certain characters from a match, providing fine control over what gets matched.

Basic Syntax: [^characters]

- The `^` symbol **negates** the character set.
- **Example:** `[^abc]` matches any character **except** `a`, `b`, or `c`.

▼ For Example :

- **Text:** `"Hello, world! How are you?"`
- **Pattern Explanation:** The regular expression `[^.,!?]` will match:
  - All alphabetic characters (`H`, `e`, `l`, `o`, `w`, `r`, `d`, `H`, `o`, `w`, etc.)

    - Spaces are also matched.
    - Punctuation marks `,`, `!`, and `?` are **skipped**.

▼ Special Characters in Character Sets :

> Some characters have special meanings in regex (e.g., `.`, `*`, `+`). Inside character sets, these characters are often treated as literals and lose their special meaning, except for a few exceptions like `-` (range indicator) and `^` (negation if placed at the beginning).

- **Syntax:** `[.*+]` matches any of the characters `.`, ` `, or `+` literally.
- **Example:**
    - **Pattern:** `[.*+]`
        - **Matches:** a literal period (`.`), asterisks (` `), or plus (`+`).
        - **Text:** "a.b*c+"
        - The regex will match ".", "*", and "+".

▼ Predefined Character Classes :

Instead of defining a character set manually, regex provides predefined character classes to simplify matching common types of characters.

- `\d`: Matches any digit (equivalent to `[0-9]`).
- `\D`: Matches any non-digit character (equivalent to `[^0-9]`).

- `\w`: Matches any "word" character: alphanumeric and underscore (equivalent to `[a-zA-Z0-9_]`).
- `\W`: Matches any non-word character (equivalent to `[^a-zA-Z0-9_]`).
- `\s`: Matches any whitespace character (spaces, tabs, line breaks).
- `\s`: Matches any non-whitespace character.

▼ Magic of Escape Sequences in Character Sets :

- Certain characters have special meanings inside a character set, such as the caret `^`, hyphen , and closing bracket `]`. If you want to match these characters literally, they need to be **escaped** using a backslash `\`.

- **Example:**
  - **Pattern:** `[a\-z]`
    - **Matches:** a literal hyphen , or any lowercase letter between `a` and `z`.
    - **Text:** "a-z"
    - The regex will match "a", "-", and "z".

▼ Repetitions Characters :

> **Repetition characters** (also called quantifiers) define **how many times** a character, group, or pattern should appear in the text. They allow you to search for repeating occurrences of a pattern, making them a powerful tool for matching varying lengths of text.

▼ Types of Repetition Characters (Quantifiers) :

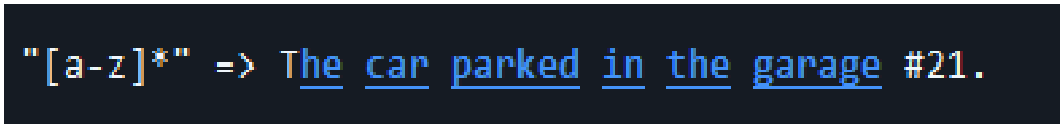
- The Asterisk (`*`)
- **The Plus (`+`)**
- **The Question Mark (`?`)**
- **The Curly Braces (`{}`)**

▼ The Asterisk (`*`): Zero or More Repetitions :

> The `*` symbol matches zero or more repetitions of the preceding matcher. The
> regular expression
> `a*` means: zero or more repetitions of the preceding lowercase
> character
> `a`.

For example, the regular expression

`[a-z]*` means: any number of lowercase letters in a row.

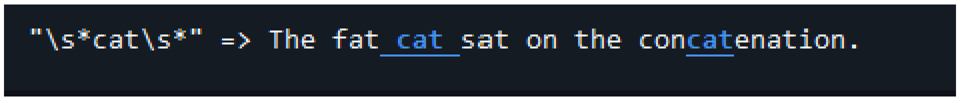


```
"[a-z]*" => The car parked in the garage #21.
```

>>Test Regex Here<<

For Example , The `*` symbol can be used with the meta character `.` to match any string of characters `.*`. The `*` symbol can be used with the whitespace character `\s` to match a string of whitespace characters. For example, the expression

`\s*cat\s*` means: zero or more spaces, followed by a lowercase `c`, followed by a lowercase
`a`, followed by a lowercase `t`,
followed by zero or more spaces.



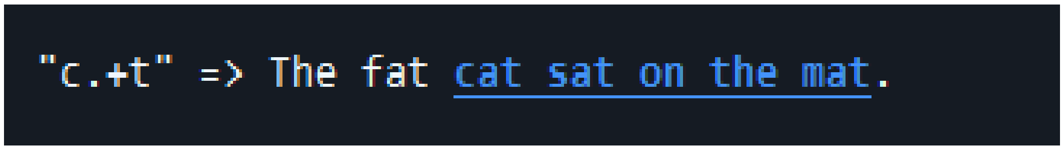
```
"\s*cat\s*" => The fat cat sat on the concatenation.
```

>>Test Regex Here<<

▼ The Plus (`+`): One or More Repetitions :

> The `+` symbol matches one or more repetitions of the preceding character. For
> example, the regular expression

For example, the regular expression
`c.+t` means: a lowercase `c`, followed by
at least one character, followed by a lowercase
`t`. It needs to be
clarified that
`t` is the last `t` in the sentence.



```
"c.+t" => The fat cat sat on the mat.
```

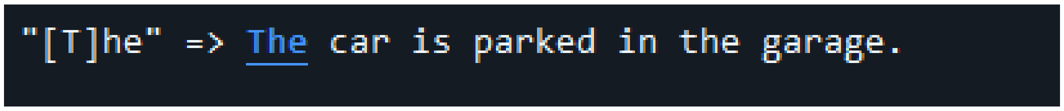
>>[Test Regex Here](#)<<

▼ The Question Mark (`?`): Zero or One Repetition :

> The meta character `?` makes the preceding
> character
> optional. This symbol matches zero or one instance
> of the preceding character, That mean Match
> **zero or one** occurrence of the preceding element.

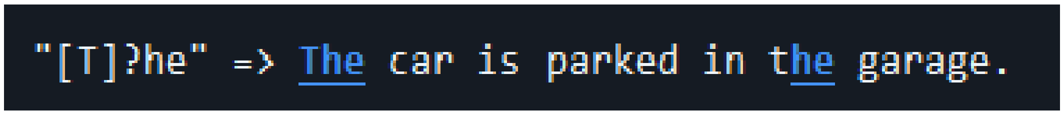
For example, the regular expression
`[T]?he` means: Optional uppercase

`T`, followed by a lowercase `h`, followed by a lowercase `e`.



```
"[T]he" => The car is parked in the garage.
```

>>[Test Regex Here](#)<<



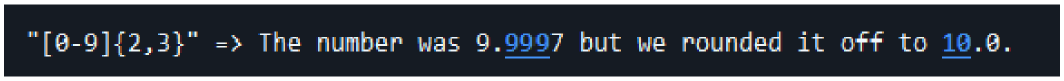
```
"[T]?he" => The car is parked in the garage.
```

>>[Test Regex Here](#)<<

▼ The Curly Braces (`{}`): Specific Number of Repetitions:

> Curly braces allow you to specify **exactly how many times** you want to match the character. You can specify an exact number or a range.

For example, the regular expression `[0-9]{2,3}` means: Match at least 2 digits, but not more than 3, ranging from 0 to 9.

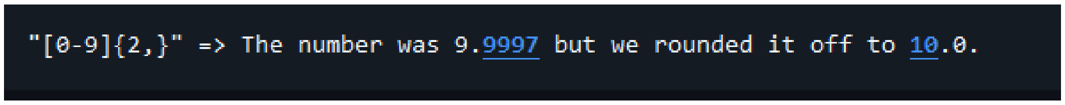


```
"[0-9]{2,3}" => The number was 9.9997 but we rounded it off to 10.0.
```

>>Test Regex Here<<

We can leave out the second number. For example, the regular expression

`[0-9]{2,}` means: Match 2 or more digits. If we also remove the comma,



```
"[0-9]{2,}" => The number was 9.9997 but we rounded it off to 10.0.
```

>>Test Regex Here<<

The regular expression `[0-9]{3}` means: Match exactly 3 digits.

```
"[0-9]{3}" => The number was 9.9997 but we rounded it off to 10.0.
```

>>[Test Regex Here](#)<<

▼ Capturing Groups :