

Strings and Regular Expressions

This page covers string literals, string methods, regular expressions, and the encoding and hash helpers of DG Script.

Topics on this page:

- String literals and escapes
- Joining strings and converting values to strings
- String methods
- Regular expressions
- Encoding and hash helpers
- `parseUrl`

See also: [Numbers and Math](#), [Lists and Maps](#), [JSON and XML](#).

String Literals and Escapes

You can quote a string with single or double quotes. A string must end on the same line it starts on.

```
str1 = 'Single quotes';
str2 = "Double quotes";
str3 = 'Double quotes in "single" quotes';
str4 = "Single quotes in 'double' quotes";
str5 = "escape quote \"\"";
```

A backslash (\) starts an escape sequence:

Sequence	Result
\n	New line
\r	Carriage return
\t	Tab
\b	Backspace
\f	Form feed
\uXXXX	The Unicode character with hex code XXXX (exactly four hex digits)
\ \" \\	A single quote, double quote, or backslash
backslash at the end of a line	The line continues; the line break is removed
\ followed by any other character	That character. For example \x41 gives x41 and \0 gives 0

For example, the following code sets the string block's value to C:\:

```
@parent.string.value = "C:\\\";
```

Template literals (backticks) are not supported. Build strings with + instead.

Joining Strings and Converting Values

When the **left** side of + is a string, the right side is converted to a string and appended. null is appended as an empty string, and lists and maps are appended as JSON.

When the left side is a number, the right side is converted to a **number**. So 1 + "2" is 3 and 1 + "a" is NaN. To build a string that starts with a number, start with an empty string: "" + 1 + "2".

```
print("a" + 1, "\n");  
print("a" + null, "\n");  
print("list: " + [1, 2], "\n");  
print(1 + "2", "\n");  
return "" + 1 + "2";
```

Print output:

```
a1  
a  
list: [1,2]  
3
```

Output: 12

String(value)

String(value) converts a value to a string. String(null) is an empty string. Lists and maps become JSON, and a DateTime becomes an ISO 8601 string.

String(number, radix) converts a number to a string in the given base (radix must be an integer from 2 to 36).

```
print(String(123), "\n");  
print(String({"a": 1}), "\n");  
return String(27, 16);
```

Print output:

```
123
```

```
{"a":1}
```

Output: 1b

String Methods

Call a method on any string value, for example `"abc".toUpperCase()`. Positions start at 0.

- Strings cannot be indexed with brackets. `"abc"[1]` does not return `b`; use `charAt(1)`.
- A position outside the string causes an error (for example `"abc".charAt(5)` or `"abc".substring(2, 9)`).
- These JavaScript methods do **not** exist: `startsWith`, `endsWith`, `includes`, `padStart`, `padEnd`, `repeat`, `search`, `concat`, `localeCompare`. Use `indexOf` or a regular expression instead.

Method	Returns
<code>str.length</code>	The number of characters (a property, not a method)
<code>str.charAt(pos)</code>	The character at <code>pos</code>
<code>str.charCodeAt(pos)</code>	The UTF-16 code of the character at <code>pos</code>
<code>str.indexOf(search)</code>	Position of the first occurrence of <code>search</code> , or <code>-1</code> . There is no <code>start-position</code> argument
<code>str.lastIndexOf(search)</code>	Position of the last occurrence of <code>search</code> , or <code>-1</code>
<code>str.substring(start, end)</code>	Characters from <code>start</code> up to (not including) <code>end</code> . Negative values count from the end. <code>end</code> is optional. Same as <code>subString</code> and <code>slice</code>
<code>str.substr(start, length)</code>	<code>length</code> characters from <code>start</code> . <code>length</code> is optional. Same as <code>subStr</code> . Negative values are not supported
<code>str.split(separator, removeEmpty)</code>	A list of parts. <code>separator</code> is a string or regular expression. When <code>removeEmpty</code> is <code>true</code> , empty parts are removed
<code>str.replace(find, replacement)</code>	Replaces the first match of <code>find</code> (string or regular expression). A regular expression with the <code>g</code> flag replaces all matches
<code>str.replaceAll(find, replacement)</code>	Replaces all matches of <code>find</code> (string or regular expression)
<code>str.replaceAllMapped(regex, function)</code>	Replaces all matches with the return value of <code>function</code> . See Regular expressions
<code>str.match(regex)</code>	See Regular expressions
<code>str.matchAll(regex)</code>	See Regular expressions
<code>str.toLowerCase()</code> , <code>str.toUpperCase()</code>	The string in lower or upper case

Method	Returns
<code>str.trim()</code> , <code>str.trimLeft()</code> , <code>str.trimRight()</code>	The string without whitespace at both ends, the start, or the end
<code>str.encodeUriComponent()</code> , <code>str.decodeUriComponent()</code>	See Encoding
<code>str.encodeBase64()</code> , <code>str.decodeBase64()</code>	See Encoding
<code>str.encodeCamelCase()</code> , <code>str.decodeCamelCase()</code>	See Encoding
<code>str.splitQuery()</code>	See Encoding
<code>str.md5()</code> , <code>str.sha1()</code> , <code>str.sha256()</code>	See Hashes

The replacement text in `replace` and `replaceAll` is used literally. `$1` and `$&` are **not** replaced by captured groups; use `replaceAllMapped` for that.

Examples

```
s = "  Hello, World  ".trim();
print(s.length, "\n");
print(s.charAt(1), "\n");
print(s.charCodeAt(0), "\n");
print(s.indexOf("o"), " ", s.lastIndexOf("o"), "\n");
print(s.substring(7), "\n");
print(s.substring(0, -7), "\n");
print(s.substr(7, 3), "\n");
print(s.toUpperCase(), "\n");
return s.replace("l", "L") + " / " + s.replaceAll("l", "L");
```

Print output:

```
12
e
72
4 8
World
Hello
Wor
HELLO, WORLD
```

Output: HeLlo, World / HeLlo, WorLd

Splitting:

```
print("1,2,3".split(","), "\n");
print("a,,b".split(",").length, "\n");
print("a,,b".split(",", true), "\n");
return "a1b22c".split(/\d+/);
```

Print output:

```
[1, 2, 3]
4
[a, b]
```

Output: [a, b, c]

Regular Expressions

Write a regular expression as a literal: `/pattern/flags`. The pattern uses the browser's regular expression syntax. Supported flags:

- `i` — ignore case
- `m` — multi-line: `^` and `$` match at line breaks
- `g` — global: affects `replace` and `match`

There is no new `RegExp(...)` constructor, so a pattern cannot be built from a string at run time. Regular expressions have no `lastIndex` state.

Method	Returns
<code>regex.test(str)</code>	true if <code>str</code> contains a match
<code>regex.exec(str)</code>	A list with the whole first match followed by the captured groups, or null if nothing matches
<code>str.match(regex)</code>	Without <code>g</code> : the first matched string (not a list as in JavaScript), or null. With <code>g</code> : a list of all matched strings, or null
<code>str.matchAll(regex)</code>	A list of all matched strings (empty list if none). Groups are not included
<code>str.replace(regex, text)</code>	Replaces the first match, or all matches with <code>g</code>
<code>str.replaceAll(regex, text)</code>	Replaces all matches
<code>str.replaceAllMapped(regex, function)</code>	Calls <code>function(m)</code> for each match, where <code>m</code> is a list [<code>wholeMatch</code> , <code>group1</code> , <code>group2</code> , ...], and replaces the match with the returned string
<code>str.split(regex)</code>	Splits at every match

```
print(/^ab/i.test("ABC"), "\n");
print(/((\w+)\s+(\w+)\s+).com/.exec("mail joe@site.com now"), "\n");
print("a1b22c333".match(/\d+/), "\n");
print("a1b22c333".match(/\d+/g), "\n");
print("abc".match(/\d+/g), "\n");
print("DabcG DdefG".replace(/D(\w+)/g, "d"), "\n");
return "2024-01-05".replaceAllMapped(/((\d+)-(\d+)-(\d+))/, function (m) {
```

```
return m[3] + "/" + m[2] + "/" + m[1];
});
```

Print output:

```
true
[joe@site.com, joe, site]
1
[1, 22, 333]
null
d d
```

Output: 05/01/2024

Encoding and Hash Helpers

Function	Returns
<code>str.encodeURIComponent()</code>	The string with URL-unsafe characters percent-encoded (space becomes %20)
<code>str.decodeURIComponent()</code>	The decoded string
<code>str.splitQuery()</code>	A map of the decoded name/value pairs of a URL query string such as <code>a=1&b=2</code>
<code>str.encodeBase64()</code>	The UTF-8 bytes of the string, encoded as Base64
<code>str.decodeBase64()</code>	The decoded UTF-8 string
<code>str.decodeBase64(true)</code>	The decoded bytes as a list of numbers
<code>list.encodeBase64()</code>	A list of byte values (0-255) encoded as Base64
<code>str.encodeCamelCase()</code>	A camel-case name: characters other than letters, digits, <code>_</code> , <code>-</code> and space are removed, leading digits and underscores are removed, the text is lower-cased, and each letter after a space or hyphen (or a letter after a digit) is upper-cased
<code>str.decodeCamelCase()</code>	Each upper-case letter is replaced with a space and the lower-case letter
<code>str.md5()</code>	The MD5 hash of the UTF-8 bytes, as 32 hex characters
<code>str.sha1()</code>	The SHA-1 hash, as 40 hex characters
<code>str.sha256()</code>	The SHA-256 hash, as 64 hex characters
<code>sha1Hmac(key, message)</code>	The HMAC-SHA1 of message with key, encoded as Base64
<code>sha256Hmac(key, message)</code>	The HMAC-SHA256 of message with key, encoded as Base64

```
print("a b&c".encodeURIComponent(), "\n");
print("a%20b".decodeURIComponent(), "\n");
q = "a%20b=1&c=d".splitQuery();
print(q["a b"], " ", q.c, "\n");
```

```
print("abc123".encodeBase64(), "\n");
print("YWJjMTIz".decodeBase64(), "\n");
print("YWJj".decodeBase64(true), "\n");
print([104, 105].encodeBase64(), "\n");
print("Hello World 2nd".encodeCamelCase(), "\n");
print("helloWorld".decodeCamelCase(), "\n");
print("abc123".md5(), "\n");
print("abc123".sha1(), "\n");
print(shalHmac("key", "message"), "\n");
return "abc123".sha256();
```

Print output:

```
a%20b%26c
a b
1 d
YWJjMTIz
abc123
[97, 98, 99]
aGk=
helloWorld2Nd
hello world
e99a18c428cb38d5f260853678922e03
6367c48dd193d56ea7b0baad25b19455e529f5ee
IIjfdNXyFGtIFGyvSWU3fp0L46Q=
```

Output: 6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a84118090

parseUrl(url)

`parseUrl(url)` splits a URL into its parts. It returns a map with the keys `scheme`, `host`, `path`, `port`, `fragment`, `query` and `queryParameters` (a map of decoded query values), or `null` if the URL cannot be parsed. When the URL has no port, `port` is the default port of the scheme (80 for http, 443 for https).

```
u = parseUrl("https://example.com:8080/a/b?x=1&y=two#top");
print(u.scheme, " ", u.host, " ", u.port, "\n");
print(u.path, " ", u.query, " ", u.fragment, "\n");
return u.queryParameters.y;
```

Print output:

```
https example.com 8080
```

```
/a/b x=1&y=two top
```

Output: two

From:

<http://wiki.dglogik.com/> - **DGLogik**

Permanent link:

http://wiki.dglogik.com/dglux5_wiki:dgscrip:strings

Last update: **2026/10/22 17:28**

