

Bash Parameter Expansion Cheatsheet

By Dejan Panovski • Updated on Oct 27, 2026

Bash parameter expansion syntax for default values, substrings, prefix and suffix removal, search and replace, case conversion, and arrays.

Parameter expansion reads, tests, and reshapes variable values inside `${...}` without running an external command. The syntax below is for Bash 5. If a script must also run on macOS, which ships Bash 3.2, check the version card first.

Basic Expansion

Braces mark where the variable name ends. Quote expansions to keep the value as one word.

<code>\$var</code>	Value of <code>var</code>
<code>\${var}_old</code>	Value followed by <code>_old</code> ; without braces, Bash reads <code>\$var_old</code>
<code>"\${var}"</code>	Value as one word, with no word splitting or globbing
<code>\${10}</code>	Tenth positional parameter; <code>\$10</code> is <code>\$1</code> followed by 0

Default Values

The colon forms treat an empty value like an unset one. Without the colon, only an unset variable triggers the operator.

<code>\${var:-word}</code>	word if <code>var</code> is unset or empty; <code>var</code> stays unchanged
<code>\${var-word}</code>	word only if <code>var</code> is unset
<code>\${var:=word}</code>	Assign word to <code>var</code> if unset or empty, then expand it
<code>\${var=word}</code>	Assign word only if <code>var</code> is unset
<code>\${var:?message}</code>	Print message and fail if <code>var</code> is unset or empty
<code>\${var?message}</code>	Fail only if <code>var</code> is unset
<code>\${var:+word}</code>	word if <code>var</code> is set and non-empty, otherwise nothing
<code>\${var+word}</code>	word if <code>var</code> is set, even to an empty string
<code>: "\${var:=word}"</code>	Assign a default on its own line

Bash expands `word` only when it is used, so `${var:-$(cmd)}` runs `cmd` only when `var` is unset or empty. In a script, a failed `:?` check exits the script.

Length and Substrings

Offsets start at 0. A negative offset needs a space or parentheses, otherwise Bash reads it as the `: -` operator.

<code>\${#var}</code>	Number of characters in the value
<code>\${var:7}</code>	Everything from offset 7 to the end
<code>\${var:0:5}</code>	Five characters starting at offset 0
<code>\${var: -5}</code>	Last five characters
<code>\${var: (-5)}</code>	Last five characters, using parentheses instead of a space
<code>\${var: -5:3}</code>	Three characters, starting five from the end
<code>\${var:1:-1}</code>	Value without its first and last character
<code>\${var:i:n}</code>	Offset and length from variables; both are arithmetic, so <code>\$</code> is optional

`${var:-5}` is a default value, not an offset. It returns the whole value when `var` is set and non-empty, or `5` when it is unset or empty. In a UTF-8 locale, lengths and offsets count characters, not bytes.

Split Paths and Filenames

These examples use

```
path=/var/www/site/index.backup.html and
file=archive.tar.gz .
```

<code>\${path##*/}</code>	<code>index.backup.html</code> , the filename
<code>\${path%/*}</code>	<code>/var/www/site</code> , the directory
<code>\${path%.*}</code>	<code>/var/www/site/index.backup</code> , without the last extension
<code>\${path%%.*}</code>	<code>/var/www/site/index</code> , without any extension
<code>\${path##*.}</code>	<code>html</code> , the last extension
<code>\${file#*.}</code>	<code>tar.gz</code> , every extension
<code>\${file%%.*}</code>	<code>archive</code> , the name before the first dot

A name without a dot comes back unchanged from `${file##*.}`, and `${file%.*}` turns `.bashrc` into an empty string. Extension patterns also match dots in directory names, so extract the filename with `file=${path##*/}` before removing extensions when directory names may contain dots. For the edge cases that the `basename` command handles, such as a trailing slash, use the command instead.

Remove a Prefix or Suffix

`#` trims from the start and `%` trims from the end. One operator removes the shortest match; two remove the longest.

<code>\${var#pattern}</code>	Remove the shortest match from the start
<code>\${var##pattern}</code>	Remove the longest match from the start
<code>\${var%pattern}</code>	Remove the shortest match from the end
<code>\${var%%pattern}</code>	Remove the longest match from the end
<code>\${var#"prefix"}</code>	Remove the exact text in <code>prefix</code> , even if it contains <code>*</code> or <code>?</code>
<code>\${dir%/}</code>	Remove one trailing slash

A pattern that does not match leaves the value unchanged.

Search and Replace

Patterns are globs, not regular expressions. Leave out the replacement to delete the match.

<code>\${var/pattern/text}</code>	Replace the first match
<code>\${var//pattern/text}</code>	Replace every match
<code>\${var/#pattern/text}</code>	Replace a match at the start only
<code>\${var/%pattern/text}</code>	Replace a match at the end only
<code>\${var/pattern}</code>	Delete every match
<code>\${var// /_}</code>	Replace each space with an underscore
<code>\${var//[0-9]/#}</code>	Replace each digit with <code>#</code>
<code>\${var/#/text}</code>	Add <code>text</code> at the start
<code>\${var/%/text}</code>	Add <code>text</code> at the end
<code>\${var//o/[&]}</code>	Wrap each <code>o</code> in brackets; <code>&</code> inserts the matched text
<code>\${var//o/\&}</code>	Replace each <code>o</code> with a literal <code>&</code>

Bash 5.2 turned on `patsub_replacement`, so an unquoted `&` in the replacement means the matched text, even when it comes from a variable. Quote the replacement, as in `${var//o/"$text"}`, to keep it literal.

Change Case

A pattern after the operator limits which characters change.

<code>\${var^}</code>	Uppercase the first character
<code>\${var^^}</code>	Uppercase every character
<code>\${var,}</code>	Lowercase the first character
<code>\${var,,}</code>	Lowercase every character
<code>\${var^[aeiou]}</code>	Uppercase only the vowels
<code>[[\${a,,} == "\${b,,}"]]</code>	Compare two strings without regard to case

Transform Values

The `${var@op}` operators format a value or describe the variable that holds it.

<code>\${var@Q}</code>	Value quoted so the shell can read it back as input
<code>\${var@E}</code>	Value with backslash escapes such as <code>\n</code> and <code>\t</code> expanded
<code>\${var@P}</code>	Value expanded like a prompt string, so <code>\u@h</code> becomes <code>user@host</code>
<code>\${var@A}</code>	Assignment or <code>declare</code> command that recreates <code>var</code>
<code>\${var@a}</code>	Attribute flags, such as <code>a</code> (array), <code>i</code> (integer), <code>r</code> (read-only), or <code>x</code> (exported)
<code>\${var@U}</code>	Uppercase the value
<code>\${var@u}</code>	Uppercase the first character
<code>\${var@L}</code>	Lowercase the value
<code>"\${arr[@]@Q}"</code>	Every element, quoted separately
<code>\${map[@]@K}</code>	Keys and quoted values of an associative array, as one string
<code>"\${map[@]@k}"</code>	Keys and values as separate words

Arrays

Quote `[@]` to keep each element as one word. The string operators above apply to every element.

<code>\${arr[0]}</code>	Element at index 0
<code>\${arr[-1]}</code>	Last element
<code>"\${arr[@]}"</code>	Every element, one word each
<code>"\${arr[*]}"</code>	Every element in one word, joined by the first character of <code>IFS</code>
<code>\${#arr[@]}</code>	Number of elements
<code>\${#arr[2]}</code>	Length of element 2
<code>\${!arr[@]}</code>	Indices, or the keys of an associative array
<code>"\${arr[@]:1:2}"</code>	Two elements, starting at index 1
<code>"\${arr[@]: -2}"</code>	Last two elements when indices are consecutive
<code>"\${arr[@]%.*}"</code>	Every element without its extension
<code>"\${arr[@]#/dir/}"</code>	Every element with <code>dir/</code> added at the start

Negative slice offsets count back from one greater than the highest index, so gaps can leave fewer elements in the result. `$arr` and `${#arr}` refer only to element 0. For declaring and changing arrays, see the Bash arrays guide.

Positional Parameters

`$@` accepts the same substring and pattern operators as an array.

<code>\${1:-.}</code>	First argument, or <code>.</code> when it is missing or empty
<code>\${1:?usage: \$0 FILE}</code>	Stop with a usage message when the first argument is missing or empty
<code>\$#</code>	Number of arguments
<code>"\${@:2}"</code>	Every argument from the second onward
<code>"\${@:2:3}"</code>	Three arguments, starting at the second
<code>\${@: -1}</code>	Last argument
<code>\${!#}</code>	Last argument, using indirection
<code>"\${@/#/--}"</code>	Every argument with <code>--</code> added at the start

See Bash positional parameters for `shift`, `$*`, and quoting rules.

Indirect Expansion

These forms expand a variable whose name is stored in another variable.

<code>\${!ref}</code>	Value of the variable named in <code>ref</code>
<code>\${!ref:-default}</code>	Same, with a default when that variable is unset or empty
<code>ref='arr[@]'; "\${!ref}"</code>	Every element of the array named in <code>ref</code>
<code>\${!prefix*}</code>	Names of all variables that start with <code>prefix</code>
<code>"\${!prefix@}"</code>	Same names, one word each
<code>declare -n ref=var</code>	Make <code>ref</code> a nameref; reading or assigning <code>ref</code> uses <code>var</code>

Do not build the name in `ref` from untrusted input. Bash evaluates array subscripts in the name, so `ref='a[${cmd}]'` runs `cmd`.

Pattern Syntax

The `#`, `%`, and `/` operators match with the same wildcards as filenames. The `extglob` forms need `shopt -s extglob`.

<code>*</code>	Any string, including an empty one
<code>?</code>	Any single character
<code>[abc]</code>	One of the listed characters
<code>[!abc]</code> or <code>[^abc]</code>	One character that is not listed
<code>[0-9]</code>	One character in the range
<code>[[:space:]]</code>	One character from a class, such as <code>[[:digit:]]</code> , <code>[[:alpha:]]</code> , or <code>[[:upper:]]</code>
<code>+(pattern)</code>	One or more matches (<code>extglob</code>)
<code>*(pattern)</code>	Zero or more matches (<code>extglob</code>)
<code>?(pattern)</code>	Zero or one match (<code>extglob</code>)
<code>@(jpg jpeg)</code>	Exactly one of the alternatives (<code>extglob</code>)

Common Recipes

<code>echo "\${PATH//:/\$'\n'}"</code>	Print each <code>PATH</code> directory on its own line
<code>n=\${var//[,];} echo "\${#n}"</code>	Count the commas in <code>var</code>
<code>(IFS=,; printf '%s\n' "\${arr[*]}")</code>	Join array elements with commas
<code>host=\${url#*://}; host=\${host%/*}</code>	Host and port from a URL such as <code>https://example.com:8080/page</code>
<code>\${ver%.*}</code>	Major version, 5 from 5.3.9
<code>\${var##+([[:space:]])}</code>	Trim leading whitespace (<code>extglob</code>)
<code>\${var%+([[:space:]])}</code>	Trim trailing whitespace (<code>extglob</code>)
<code>\${var#"\${var%[![:space:]]*}"}</code>	Trim leading whitespace without <code>extglob</code>
<code>\${var%"\${var##*[![:space:]]}"}</code>	Trim trailing whitespace without <code>extglob</code>
<code>\${file/%.@(jpg jpeg)/.png}</code>	Change a <code>.jpg</code> or <code>.jpeg</code> extension to <code>.png</code> (<code>extglob</code>)
<code>[[-z \${var+x}]]</code>	True only when <code>var</code> is unset

Common Mistakes

Most expansion bugs come from a missing space, missing quotes, or the wrong shell.

<code>\${var:-3}</code> returns the whole value	Add a space for a negative offset: <code>\${var:- 3}</code>
A value with spaces splits into several words	Quote the expansion: <code>"\$var"</code> , <code>"\${arr[@]}"</code>
A variable in the pattern acts as a wildcard	Quote it: <code>\${var#"\${prefix}"}</code>
<code>&</code> in a replacement turns into the match	Write <code>\&</code> or quote the replacement
<code>.+</code> or <code>^</code> in a pattern does not work	Patterns are globs; use <code>[[\$var =~ regex]]</code> or <code>sed</code> for regular expressions
unbound variable under <code>set -u</code>	Write <code>\${var:-}</code> for variables that may be unset
Bad substitution	The script runs under <code>sh</code> or an old Bash; run it with <code>bash</code> and check the version
<code>substring expression < 0</code>	The negative length points before the offset, or Bash is older than 4.2

See the Bash set command guide for `set -u` and the other strict mode options.

Bash Version Support

Run `echo "$BASH_VERSION"` to check your version. Bash 3.2, still `/bin/bash` on macOS, has none of the features below.

<code>\${var^}, \${var,,}, associative arrays</code>	4.0
<code>\${var:1:-1}, \${arr[-1]}</code>	4.2
<code>declare -n namerefs</code>	4.3
<code>@Q, @E, @P, @A, @a</code>	4.4
<code>@U, @u, @L, @K</code>	5.1
<code>@k, &</code> as the match in replacements	5.2

Related Guides

Use these guides for worked examples and the full syntax.

Bash Parameter Expansion	Worked examples of defaults, trimming, and replacement
Bash String Manipulation	Substrings, replacement, case, and substring checks
Bash Cheatsheet	Variables, conditionals, loops, and functions
Regex Cheatsheet	Regular expressions for <code>[[=~]]</code> , <code>grep</code> , and <code>sed</code>
GNU Bash Shell Parameter Expansion	Official definitions of every operator