

Expressions

GitHub Docs

In this article

About expressions

Literals

Operators

Functions

Status check functions

Object filters

You can evaluate expressions in workflows and actions.

Note: GitHub-hosted runners are not currently supported on GitHub Enterprise Server. You can see more information about planned future support on the <u>GitHub public roadmap</u>.

About expressions *∂*

You can use expressions to programmatically set environment variables in workflow files and access contexts. An expression can be any combination of literal values, references to a context, or functions. You can combine literals, context references, and functions using operators. For more information about contexts, see "Contexts."

Expressions are commonly used with the conditional if keyword in a workflow file to determine whether a step should run. When an if conditional is true, the step will run.

You need to use specific syntax to tell GitHub to evaluate an expression rather than treat it as a string.

```
${{ <expression> }}
```

Note: The exception to this rule is when you are using expressions in an <code>if</code> clause, where, optionally, you can usually omit \${{ and }} . For more information about <code>if</code> conditionals, see "Workflow syntax for GitHub Actions."

Warning: When creating workflows and actions, you should always consider whether your code might execute untrusted input from possible attackers. Certain contexts should be treated as untrusted input, as an attacker could insert their own malicious content. For more information, see "Security hardening for GitHub Actions."

Example setting an environment variable $\mathscr {P}$

```
env:
MY_ENV_VAR: ${{ <expression> }}
```

Literals 🔗

As part of an expression, you can use boolean, null, number, or string data types.

Data type	Literal value
boolean	true Or false
null	null
number	Any number format supported by JSON.
string	You don't need to enclose strings in \${{ and }}}. However, if you do, you must use single quotes (') around the string. To use a literal single quote, escape the literal single quote using an additional single quote (' '). Wrapping with double quotes (") will throw an error.

Example of literals $\mathscr O$

```
env:
  myNull: ${{ null }}
  myBoolean: ${{ false }}
  myIntegerNumber: ${{ 711 }}
  myFloatNumber: ${{ -9.2 }}
  myHexNumber: ${{ 0xff }}
  myExponentialNumber: ${{ -2.99e-2 }}
  myString: Mona the Octocat
  myStringInBraces: ${{ 'It''s open source!' }}
```

Operators @

Operator	Description
()	Logical grouping
	Index
	Property de-reference
	Not
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
=	Equal
!=	Not equal
&&	And
II	Or

Notes:

- GitHub ignores case when comparing strings.
- steps.<step_id>.outputs.<output_name> evaluates as a string. You need to use specific syntax to tell GitHub to evaluate an expression rather than treat it as a string. For more information, see "Contexts."
- For numerical comparison, the fromJSON() function can be used to convert a string to a number. For more information on the fromJSON() function, see "fromJSON."

GitHub performs loose equality comparisons.

• If the types do not match, GitHub coerces the type to a number. GitHub casts data types to a number using these conversions:

Туре	Result
Null	0
Boolean	true returns 1 false returns 0
String	Parsed from any legal JSON number format, otherwise NaN . Note: empty string returns 0.
Array	NaN
Object	NaN

- A comparison of one NaN to another NaN does not result in true. For more information, see the "NaN Mozilla docs."
- GitHub ignores case when comparing strings.
- Objects and arrays are only considered equal when they are the same instance.

GitHub offers ternary operator like behaviour that you can use in expressions. By using a ternary operator in this way, you can dynamically set the value of an environment variable based on a condition, without having to write separate if-else blocks for each possible option.

Example @

```
env:
   MY_ENV_VAR: ${{ github.ref == 'refs/heads/main' && 'value_for_main_branch' ||
'value_for_other_branches' }}
```

In this example, we're using a ternary operator to set the value of the MY_ENV_VAR environment variable based on whether the GitHub reference is set to refs/heads/main or not. If it is, the variable is set to value_for_main_branch . Otherwise, it is set to value_for_other_branches . It is important to note that the first value after the && condition must be truthy otherwise the value after the || will always be returned.

Functions &

GitHub offers a set of built-in functions that you can use in expressions. Some functions cast values to a string to perform comparisons. GitHub casts data types to a string using these conversions:

Туре	Result
Null	
Boolean	'true' or 'false'
Number	Decimal format, exponential for large numbers
Array	Arrays are not converted to a string
Object	Objects are not converted to a string

contains &

```
contains( search, item )
```

Returns true if search contains item. If search is an array, this function returns true if the item is an element in the array. If search is a string, this function returns true if the item is a substring of search. This function is not case sensitive. Casts values to a string.

Example using a string 🔗

```
contains('Hello world', 'llo') returns true.
```

Example using an object filter &

contains(github.event.issue.labels.*.name, 'bug') returns true if the issue related to the event has a label "bug".

For more information, see "Object filters."

Example matching an array of strings &

Instead of writing github.event_name == "push" || github.event_name == "pull_request" ,
you can use contains() with fromJSON() to check if an array of strings contains an
item .

For example, contains(fromJSON('["push", "pull_request"]'), github.event_name) returns true if github.event_name is "push" or "pull_request".

startsWith @

```
startsWith( searchString, searchValue )
```

Returns true when searchString starts with searchValue. This function is not case sensitive. Casts values to a string.

Example of startsWith @

```
startsWith('Hello world', 'He') returns true.
```

endsWith @

```
endsWith( searchString, searchValue )
```

Returns true if searchString ends with searchValue. This function is not case sensitive. Casts values to a string.

Example of endsWith @

endsWith('Hello world', 'ld') returns true.

format @

```
format( string, replaceValue0, replaceValue1, ..., replaceValueN)
```

Replaces values in the string, with the variable replaceValueN. Variables in the string are specified using the {N} syntax, where N is an integer. You must specify at least one replaceValue and string. There is no maximum for the number of variables (replaceValueN) you can use. Escape curly braces using double braces.

Example of format @

```
format('Hello {0} {1} {2}', 'Mona', 'the', 'Octocat')
```

Returns 'Hello Mona the Octocat'.

Example escaping braces $\mathscr O$

```
format('{{Hello {0} {1} {2}!}}', 'Mona', 'the', 'Octocat')
```

Returns '{Hello Mona the Octocat!}'.

join 🔗

```
join( array, optionalSeparator )
```

The value for array can be an array or a string. All values in array are concatenated into a string. If you provide optionalSeparator, it is inserted between the concatenated values. Otherwise, the default separator, is used. Casts values to a string.

Example of join &

```
join(github.event.issue.labels.*.name, ', ') may return 'bug, help wanted'
```

toJSON 🔗

toJSON(value)

Returns a pretty-print JSON representation of value . You can use this function to debug the information provided in contexts.

Example of toJSON &

```
toJSON(job) might return { "status": "success" }
```

fromJSON @

```
fromJSON(value)
```

Returns a JSON object or JSON data type for value . You can use this function to provide a JSON object as an evaluated expression or to convert environment variables from a string.

Example returning a JSON object &

This workflow sets a JSON matrix in one job, and passes it to the next job using an output and from JSON.

```
name: build
on: push
jobs:
 job1:
    runs-on: ubuntu-latest
    outputs:
     matrix: ${{ steps.set-matrix.outputs.matrix }}
      - id: set-matrix
       run: echo "matrix={\"include\":
[{\"project\":\"foo\",\"config\":\"Debug\"},
{\"project\":\"bar\",\"config\":\"Release\"}]}" >> $GITHUB OUTPUT
 job2:
   needs: job1
    runs-on: ubuntu-latest
   strategy:
     matrix: ${{ fromJSON(needs.job1.outputs.matrix) }}
     - run: build
```

Example returning a JSON data type 🔗

This workflow uses from JSON to convert environment variables from a string to a Boolean or integer.

```
name: print
on: push
env:
    continue: true
    time: 3
jobs:
    jobl:
       runs-on: ubuntu-latest
       steps:
       - continue-on-error: ${{ fromJSON(env.continue) }}
       timeout-minutes: ${{ fromJSON(env.time) }}
       run: echo ...
```

hashFiles @

hashFiles(path)

Returns a single hash for the set of files that matches the path pattern. You can provide a single path pattern or multiple path patterns separated by commas. The path is relative to the <code>GITHUB_WORKSPACE</code> directory and can only include files inside of the <code>GITHUB_WORKSPACE</code>. This function calculates an individual SHA-256 hash for each matched file, and then uses those hashes to calculate a final SHA-256 hash for the set of files. If the path pattern does not match any files, this returns an empty string. For more information about SHA-256, see "SHA-2."

You can use pattern matching characters to match file names. Pattern matching for hashFiles follows glob pattern matching and is case-insensitive on Windows. For more information about supported pattern matching characters, see the Patterns section in the @actions/glob documentation.

Matches any package-lock.json file in the repository.

```
hashFiles('**/package-lock.json')
```

Example with multiple patterns

Creates a hash for any package-lock.json and Gemfile.lock files in the repository.

```
hashFiles('**/package-lock.json', '**/Gemfile.lock')
```

Status check functions &

You can use the following status check functions as expressions in if conditionals. A default status check of success() is applied unless you include one of these functions. For more information about if conditionals, see "Workflow syntax for GitHub Actions" and "Metadata syntax for GitHub Actions".

success ∂

Returns true when all previous steps have succeeded.

Example of success @

```
steps:
...
- name: The job has succeeded
if: ${{ success() }}
```

always 🔗

Causes the step to always execute, and returns true, even when canceled. The always expression is best used at the step level or on tasks that you expect to run even when a job is canceled. For example, you can use always to send logs even when a job is canceled.

Warning: Avoid using always for any task that could suffer from a critical failure, for example: getting sources, otherwise the workflow may hang until it times out. If you want to run a job or step regardless of its success or failure, use the recommended alternative: if: \${{ !cancelled()}}}

Example of always 🔗

```
if: ${{ always() }}
```

cancelled @

Returns true if the workflow was canceled.

Example of cancelled &

```
if: ${{ cancelled() }}
```

failure 🔗

Returns true when any previous step of a job fails. If you have a chain of dependent jobs, failure() returns true if any ancestor job fails.

Example of failure \mathscr{O}

```
steps:
...
- name: The job has failed
  if: ${{ failure() }}
```

failure with conditions &

You can include extra conditions for a step to run after a failure, but you must still include failure() to override the default status check of success() that is automatically applied to if conditions that don't contain a status check function.

Example of failure with conditions &

```
steps:
...
- name: Failing step
  id: demo
  run: exit 1
- name: The demo step has failed
  if: ${{ failure() && steps.demo.conclusion == 'failure' }}
```

Object filters @

You can use the * syntax to apply a filter and select matching items in a collection.

For example, consider an array of objects named fruits.

```
[
    { "name": "apple", "quantity": 1 },
    { "name": "orange", "quantity": 2 },
    { "name": "pear", "quantity": 1 }
]
```

The filter fruits.*.name returns the array ["apple", "orange", "pear"].

You may also use the * syntax on an object. For example, suppose you have an object named vegetables .

```
{
    "scallions":
    {
        "colors": ["green", "white", "red"],
        "ediblePortions": ["roots", "stalks"],
},
    "beets":
    {
        "colors": ["purple", "red", "gold", "white", "pink"],
        "ediblePortions": ["roots", "stems", "leaves"],
},
    "artichokes":
    {
```

```
"colors": ["green", "purple", "red", "black"],
    "ediblePortions": ["hearts", "stems", "leaves"],
},
}
```

The filter vegetables.*.ediblePortions could evaluate to:

```
[
    ["roots", "stalks"],
    ["hearts", "stems", "leaves"],
    ["roots", "stems", "leaves"],
]
```

Since objects don't preserve order, the order of the output cannot be guaranteed.

Legal

```
© 2023 GitHub, Inc. <u>Terms Privacy Status Pricing Expert services Blog</u>
```