



Linux Ubuntu 26.04 Manual Pages on command 'jq.1'

\$ man jq.1

JQ(1)

JQ(1)

NAME

jq - Command-line JSON processor

SYNOPSIS

jq [options...] filter [files...]

jq can transform JSON in various ways, by selecting, iterating, reducing and otherwise manipulating JSON documents. For instance, running the command `jq ?map(.price) | add?` will take an array of JSON objects as input and return the sum of their "price" fields.

jq can accept text input as well, but by default, jq reads a stream of JSON entities (including numbers and other literals) from stdin. Whitespace is only needed to separate entities such as 1 and 2, and true and false. One or more files may be specified, in which case jq will read input from those instead.

The options are described in the [INVOKING JQ] section; they mostly concern input and output formatting. The filter is written in the jq language and specifies how to transform the input file or document.

FILTERS

A jq program is a "filter": it takes an input, and produces an output. There are a lot of builtin filters for extracting a particular field of an object, or converting a number to a string, or various other standard tasks.

Filters can be combined in various ways - you can pipe the output of one filter into another filter, or collect the output of a filter into an array.

Some filters produce multiple results, for instance there's one that produces all the elements of its input array. Piping that filter into a second runs the second filter for each

element of the array. Generally, things that would be done with loops and iteration in other languages are just done by gluing filters together in jq.

It's important to remember that every filter has an input and an output. Even literals like "hello" or 42 are filters - they take an input but always produce the same literal as output. Operations that combine two filters, like addition, generally feed the same input to both and combine the results. So, you can implement an averaging filter as `add / length` - feeding the input array both to the `add` filter and the `length` filter and then performing the division.

But that's getting ahead of ourselves. :) Let's start with something simpler:

INVOKING JQ

jq filters run on a stream of JSON data. The input to jq is parsed as a sequence of white-space-separated JSON values which are passed through the provided filter one at a time. The output(s) of the filter are written to standard output, as a sequence of newline-separated JSON data.

The simplest and most common filter (or jq program) is `.`, which is the identity operator, copying the inputs of the jq processor to the output stream. Because the default behavior of the jq processor is to read JSON texts from the input stream, and to pretty-print outputs, the `.` program's main use is to validate and pretty-print the inputs. The jq programming language is quite rich and allows for much more than just validation and pretty-printing.

Note: it is important to mind the shell's quoting rules. As a general rule it's best to always quote (with single-quote characters on Unix shells) the jq program, as too many characters with special meaning to jq are also shell meta-characters. For example, `jq "foo"` will fail on most Unix shells because that will be the same as `jq foo`, which will generally fail because `foo` is not defined. When using the Windows command shell (`cmd.exe`) it's best to use double quotes around your jq program when given on the command-line (instead of the `-f program-file` option), but then double-quotes in the jq program need backslash escaping. When using the Powershell (`powershell.exe`) or the Powershell Core (`pwsh/pwsh.exe`), use single-quote characters around the jq program and backslash-escaped double-quotes (`\"`) inside the jq program.

? Unix shells: `jq ?["foo"]?`

? Powershell: `jq ?["foo\"]?`

? Windows command shell: `jq "\"foo\""`

Note: jq allows user-defined functions, but every jq program must have a top-level expres?

sion.

You can affect how jq reads and writes its input and output using some command-line options:

`--null-input / -n:`

Don't read any input at all. Instead, the filter is run once using null as the input.

This is useful when using jq as a simple calculator or to construct JSON data from scratch.

`--raw-input / -R:`

Don't parse the input as JSON. Instead, each line of text is passed to the filter as a string. If combined with `--slurp`, then the entire input is passed to the filter as a single long string.

`--slurp / -s:`

Instead of running the filter for each JSON object in the input, read the entire input stream into a large array and run the filter just once.

`--compact-output / -c:`

By default, jq pretty-prints JSON output. Using this option will result in more compact output by instead putting each JSON object on a single line.

`--raw-output / -r:`

With this option, if the filter's result is a string then it will be written directly to standard output rather than being formatted as a JSON string with quotes. This can be useful for making jq filters talk to non-JSON-based systems.

`--raw-output0:`

Like `-r` but jq will print NUL instead of newline after each output. This can be useful when the values being output can contain newlines. When the output value contains NUL, jq exits with non-zero code.

`--join-output / -j:`

Like `-r` but jq won't print a newline after each output.

`--ascii-output / -a:`

jq usually outputs non-ASCII Unicode codepoints as UTF-8, even if the input specified them as escape sequences (like `"\u03bc"`). Using this option, you can force jq to produce pure ASCII output with every non-ASCII character replaced with the equivalent escape sequence.

`--sort-keys / -S:`

Output the fields of each object with the keys in sorted order.

`--color-output / -C` and `--monochrome-output / -M`:

By default, jq outputs colored JSON if writing to a terminal. You can force it to produce color even if writing to a pipe or a file using `-C`, and disable color with `-M`. When the `NO_COLOR` environment variable is not empty, jq disables colored output by default, but you can enable it by `-C`.

Colors can be configured with the `JQ_COLORS` environment variable (see below).

`--tab`:

Use a tab for each indentation level instead of two spaces.

`--indent n`:

Use the given number of spaces (no more than 7) for indentation.

`--unbuffered`:

Flush the output after each JSON object is printed (useful if you're piping a slow data source into jq and piping jq's output elsewhere).

`--stream`:

Parse the input in streaming fashion, outputting arrays of path and leaf values (scalars and empty arrays or empty objects). For example, `"a"` becomes `[[0], "a"]`, and `[[0], "a"], ["b"]]` becomes `[[0], []], [[1], "a"]`, and `[[2, 0], "b"]`.

This is useful for processing very large inputs. Use this in conjunction with `filter?` and the `reduce` and `foreach` syntax to reduce large inputs incrementally.

`--stream-errors`:

Like `--stream`, but invalid JSON inputs yield array values where the first element is the error and the second is a path. For example, `["a", n]` produces `["Invalid literal at line 1, column 7", [1]]`.

Implies `--stream`. Invalid JSON inputs produce no error values when `--stream` without `--stream-errors`.

`--seq`:

Use the `application/json-seq` MIME type scheme for separating JSON texts in jq's input and output. This means that an ASCII RS (record separator) character is printed before each value on output and an ASCII LF (line feed) is printed after every output. Input JSON texts that fail to parse are ignored (but warned about), discarding all subsequent input until the next RS. This mode also parses the output of jq without the `--seq` option.

`-f / --from-file`:

Read the filter from a file rather than from a command line, like `awk -f option`.

This changes the filter argument to be interpreted as a filename, instead of the source of a program.

`-L directory / --library-path directory:`

Prepend `directory` to the search list for modules. If this option is used then no builtin search list is used. See the section on modules below.

`--arg name value:`

This option passes a value to the `jq` program as a predefined variable. If you run `jq` with `--arg foo bar`, then `$foo` is available in the program and has the value "bar".

Note that value will be treated as a string, so `--arg foo 123` will bind `$foo` to "123".

Named arguments are also available to the `jq` program as `$ARGS.named`. When the name is not a valid identifier, this is the only way to access it.

`--argjson name JSON-text:`

This option passes a JSON-encoded value to the `jq` program as a predefined variable.

If you run `jq` with `--argjson foo 123`, then `$foo` is available in the program and has the value 123.

`--slurpfile variable-name filename:`

This option reads all the JSON texts in the named file and binds an array of the parsed JSON values to the given global variable. If you run `jq` with `--slurpfile foo bar`, then `$foo` is available in the program and has an array whose elements correspond to the texts in the file named `bar`.

`--rawfile variable-name filename:`

This option reads in the named file and binds its contents to the given global variable. If you run `jq` with `--rawfile foo bar`, then `$foo` is available in the program and has a string whose contents are to the texts in the file named `bar`.

`--args:`

Remaining arguments are positional string arguments. These are available to the `jq` program as `$ARGS.positional[]`.

`--jsonargs:`

Remaining arguments are positional JSON text arguments. These are available to the `jq` program as `$ARGS.positional[]`.

`--exit-status / -e:`

Sets the exit status of jq to 0 if the last output value was neither false nor null, 1 if the last output value was either false or null, or 4 if no valid result was ever produced. Normally jq exits with 2 if there was any usage problem or system error, 3 if there was a jq program compile error, or 0 if the jq program ran.

Another way to set the exit status is with the `halt_error` builtin function.

`--binary / -b:`

Windows users using WSL, MSYS2, or Cygwin, should use this option when using a native `jq.exe`, otherwise jq will turn newlines (LFs) into carriage-return-then-newline (CRLF).

`--version / -V:`

Output the jq version and exit with zero.

`--build-configuration:`

Output the build configuration of jq and exit with zero. This output has no supported format or structure and may change without notice in future releases.

`--help / -h:`

Output the jq help and exit with zero.

`--:`

Terminates argument processing. Remaining arguments are not interpreted as options.

`--run-tests [filename]:`

Runs the tests in the given file or standard input. This must be the last option given and does not honor all preceding options. The input consists of comment lines, empty lines, and program lines followed by one input line, as many lines of output as are expected (one per output), and a terminating empty line. Compilation failure tests start with a line containing only `%%FAIL`, then a line containing the program to compile, then a line containing an error message to compare to the actual.

Be warned that this option can change backwards-incompatibly.

BASIC FILTERS

Identity: `.`

The absolute simplest filter is `.`. This filter takes its input and produces the same value as output. That is, this is the identity operator.

Since jq by default pretty-prints all output, a trivial program consisting of nothing but `.` can be used to format JSON output from, say, `curl`.

Although the identity filter never modifies the value of its input, jq processing can some?

times make it appear as though it does. For example, using the current implementation of jq, we would see that the expression:

```
1E1234567890 | .
```

produces 1.7976931348623157e+308 on at least one platform. This is because, in the process of parsing the number, this particular version of jq has converted it to an IEEE754 double-precision representation, losing precision.

The way in which jq handles numbers has changed over time and further changes are likely within the parameters set by the relevant JSON standards. Moreover, build configuration options can alter how jq processes numbers.

The following remarks are therefore offered with the understanding that they are intended to be descriptive of the current version of jq and should not be interpreted as being prescriptive:

- (1) Any arithmetic operation on a number that has not already been converted to an IEEE754 double precision representation will trigger a conversion to the IEEE754 representation.
- (2) jq will attempt to maintain the original decimal precision of number literals (if the `--disable-decnum` build configuration option was not used), but in expressions such as `1E1234567890`, precision will be lost if the exponent is too large.
- (3) Comparisons are carried out using the untruncated big decimal representation of numbers if available, as illustrated in one of the following examples.

The examples below use the builtin function `have_decnum` in order to demonstrate the expected effects of using / not using the `--disable-decnum` build configuration option, and also to allow automated tests derived from these examples to pass regardless of whether that option is used.

```
jq ?.?
```

```
"Hello, world!"
```

```
=> "Hello, world!"
```

```
jq ?.?
```

```
0.12345678901234567890123456789
```

```
=> 0.12345678901234567890123456789
```

```
jq ?[., tojson] == if have_decnum then [12345678909876543212345,"12345678909876543212345"] else
[12345678909876543000000,"12345678909876543000000"] end?
```

```
12345678909876543212345
```

```
=> true
```

```

jq ?[1234567890987654321,-1234567890987654321 | tojson] == if have_decnum then
["1234567890987654321",-1234567890987654321"] else ["1234567890987654400",-1234567890987654400"] end?

null

=> true

jq ?. < 0.12345678901234567890123456788?

0.12345678901234567890123456789

=> false

jq ?map([., . == 1]) | tojson == if have_decnum then "[[1,true],[1.000,true],[1.0,true],[1.00,true]]" else
"[[1,true],[1,true],[1,true],[1,true]]" end?

[1, 1.000, 1.0, 100e-2]

=> true

jq ?. as $big | [$big, $big + 1] | map(. > 1000000000000000000000000000000) | . == if have_decnum then [true,
false] else [false, false] end?

10000000000000000000000000000001

=> true

```

Object Identifier-Index: .foo, .foo.bar

The simplest useful filter has the form .foo. When given a JSON object (aka dictionary or hash) as input, .foo produces the value at the key "foo" if the key is present, or null otherwise.

A filter of the form .foo.bar is equivalent to .foo | .bar.

The .foo syntax only works for simple, identifier-like keys, that is, keys that are all made of alphanumeric characters and underscore, and which do not start with a digit.

If the key contains special characters or starts with a digit, you need to surround it with double quotes like this: ."foo\$", or else .["foo\$"].

For example .["foo::bar"] and .["foo.bar"] work while .foo::bar does not.

```

jq ?.foo?

{"foo": 42, "bar": "less interesting data"}

=> 42

jq ?.foo?

{"notfoo": true, "alsonotfoo": false}

=> null

jq ?.["foo"]?

{"foo": 42}

```



```
=> 42
```

Optional Object Identifier-Index: `.foo?`

Just like `.foo`, but does not output an error when `.` is not an object.

```
jq ?.foo??
```

```
{"foo": 42, "bar": "less interesting data"}
```

```
=> 42
```

```
jq ?.foo??
```

```
{"notfoo": true, "alsonotfoo": false}
```

```
=> null
```

```
jq ?.["foo"]??
```

```
{"foo": 42}
```

```
=> 42
```

```
jq ?[.foo]??
```

```
[1,2]
```

```
=> []
```

Object Index: `.[<string>]`

You can also look up fields of an object using syntax like `.["foo"]` (`.foo` above is a short-hand version of this, but only for identifier-like strings).

Array Index: `.[<number>]`

When the index value is an integer, `.[<number>]` can index arrays. Arrays are zero-based, so `.[2]` returns the third element.

Negative indices are allowed, with `-1` referring to the last element, `-2` referring to the next to last element, and so on.

```
jq ?.[0]?
```

```
[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]
```

```
=> {"name":"JSON", "good":true}
```

```
jq ?.[2]?
```

```
[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]
```

```
=> null
```

```
jq ?.[-2]?
```

```
[1,2,3]
```

```
=> 2
```

Array/String Slice: `.[<number>:<number>]`

The `.[<number>:<number>]` syntax can be used to return a subarray of an array or substring of a string. The array returned by `.[10:15]` will be of length 5, containing the elements from index 10 (inclusive) to index 15 (exclusive). Either index may be negative (in which case it counts backwards from the end of the array), or omitted (in which case it refers to the start or end of the array). Indices are zero-based.

```
jq ?.[2:4]?
["a","b","c","d","e"]
=> ["c", "d"]

jq ?.[2:4]?
"abcdefghi"
=> "cd"

jq ?.[:3]?
["a","b","c","d","e"]
=> ["a", "b", "c"]

jq ?.[:-2]?
["a","b","c","d","e"]
=> ["d", "e"]
```

Array/Object Value Iterator: `.[ ]`

If you use the `.[index]` syntax, but omit the index entirely, it will return all of the elements of an array. Running `.[ ]` with the input `[1,2,3]` will produce the numbers as separate results, rather than as a single array. A filter of the form `.foo[ ]` is equivalent to `.foo | .[ ]`.

You can also use this on an object, and it will return all the values of the object.

Note that the iterator operator is a generator of values.

```
jq ?.[ ]?
[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]
=> {"name":"JSON", "good":true}, {"name":"XML", "good":false}

jq ?.[ ]?
[]
=>

jq ?.foo[ ]?
{"foo":[1,2,3]}
=> 1, 2, 3
```

```
jq ?.[]?
```

```
{ "a": 1, "b": 1 }
```

```
=> 1, 1
```

.[]?

Like .[], but no errors will be output if . is not an array or object. A filter of the form

.foo[]? is equivalent to .foo | .[]?.

Comma: ,

If two filters are separated by a comma, then the same input will be fed into both and the two filters' output value streams will be concatenated in order: first, all of the outputs produced by the left expression, and then all of the outputs produced by the right. For instance, filter .foo, .bar, produces both the "foo" fields and "bar" fields as separate outputs.

The , operator is one way to construct generators.

```
jq ?.foo, .bar?
```

```
{ "foo": 42, "bar": "something else", "baz": true }
```

```
=> 42, "something else"
```

```
jq ?.user, .projects[]?
```

```
{ "user": "stedolan", "projects": [ "jq", "wikiflow" ] }
```

```
=> "stedolan", "jq", "wikiflow"
```

```
jq ?.[4,2]?
```

```
[ "a", "b", "c", "d", "e" ]
```

```
=> "e", "c"
```

Pipe: |

The | operator combines two filters by feeding the output(s) of the one on the left into the input of the one on the right. It's similar to the Unix shell's pipe, if you're used to that.

If the one on the left produces multiple results, the one on the right will be run for each of those results. So, the expression .[] | .foo retrieves the "foo" field of each element of the input array. This is a cartesian product, which can be surprising.

Note that .a.b.c is the same as .a | .b | .c.

Note too that . is the input value at the particular stage in a "pipeline", specifically: where the . expression appears. Thus .a | . | .b is the same as .a.b, as the . in the middle refers to whatever value .a produced.

```
jq ?.[] | .name?
```

```
[{"name":"JSON", "good":true}, {"name":"XML", "good":false}]
```

```
=> "JSON", "XML"
```

Parenthesis

Parenthesis work as a grouping operator just as in any typical programming language.

```
jq ?(. + 2) * 5?
```

```
1
```

```
=> 15
```

TYPES AND VALUES

jq supports the same set of datatypes as JSON - numbers, strings, booleans, arrays, objects (which in JSON-speak are hashes with only string keys), and "null".

Booleans, null, strings and numbers are written the same way as in JSON. Just like every? thing else in jq, these simple values take an input and produce an output - 42 is a valid jq expression that takes an input, ignores it, and returns 42 instead.

Numbers in jq are internally represented by their IEEE754 double precision approximation.

Any arithmetic operation with numbers, whether they are literals or results of previous fil? ters, will produce a double precision floating point result.

However, when parsing a literal jq will store the original literal string. If no mutation is applied to this value then it will make to the output in its original form, even if conver? sion to double would result in a loss.

Array construction: []

As in JSON, [] is used to construct arrays, as in [1,2,3]. The elements of the arrays can be any jq expression, including a pipeline. All of the results produced by all of the expres? sions are collected into one big array. You can use it to construct an array out of a known quantity of values (as in [.foo, .bar, .baz]) or to "collect" all the results of a filter into an array (as in [.items[].name])

Once you understand the "," operator, you can look at jq?s array syntax in a different light: the expression [1,2,3] is not using a built-in syntax for comma-separated arrays, but is instead applying the [] operator (collect results) to the expression 1,2,3 (which pro? duces three different results).

If you have a filter X that produces four results, then the expression [X] will produce a single result, an array of four elements.

```
jq ?[.user, .projects[]]?
```

```
{"user":"stedolan", "projects": ["jq", "wikiflow"]}
```

```
=> ["stedolan", "jq", "wikiflow"]
```

```
jq ?[ .[] | . * 2]?
```

```
[1, 2, 3]
```

```
=> [2, 4, 6]
```

Object Construction: {}

Like JSON, {} is for constructing objects (aka dictionaries or hashes), as in: {"a": 42, "b": 17}.

If the keys are "identifier-like", then the quotes can be left off, as in {a:42, b:17}.

Variable references as key expressions use the value of the variable as the key. Key expressions other than constant literals, identifiers, or variable references, need to be parenthesized, e.g., {"a"+"b":59}.

The value can be any expression (although you may need to wrap it in parentheses if, for example, it contains colons), which gets applied to the {} expression's input (remember, all filters have an input and an output).

```
{foo: .bar}
```

will produce the JSON object {"foo": 42} if given the JSON object {"bar":42, "baz":43} as its input. You can use this to select particular fields of an object: if the input is an object with "user", "title", "id", and "content" fields and you just want "user" and "title", you can write

```
{user: .user, title: .title}
```

Because that is so common, there's a shortcut syntax for it: {user, title}.

If one of the expressions produces multiple results, multiple dictionaries will be produced.

If the input's

```
{"user":"stedolan", "titles":["JQ Primer", "More JQ"]}
```

then the expression

```
{user, title: .titles[]}
```

will produce two outputs:

```
{"user":"stedolan", "title": "JQ Primer"}
```

```
{"user":"stedolan", "title": "More JQ"}
```

Putting parentheses around the key means it will be evaluated as an expression. With the same input as above,

```
{(.user): .titles}
```

produces

```
{"stedolan": ["JQ Primer", "More JQ"]}
```

Variable references as keys use the value of the variable as the key. Without a value then the variable's name becomes the key and its value becomes the value,

```
"foo" as $foo | "bar" as $bar | {$foo, $bar:$foo}
```

produces

```
{"foo":"foo","bar":"foo"}
```

```
jq ?{user, title: .titles[]}? 
```

```
{"user":"stedolan","titles":["JQ Primer", "More JQ"]}
```

```
=> {"user":"stedolan", "title": "JQ Primer"}, {"user":"stedolan", "title": "More JQ"}
```

```
jq ?{(.user): .titles}? 
```

```
{"user":"stedolan","titles":["JQ Primer", "More JQ"]}
```

```
=> {"stedolan": ["JQ Primer", "More JQ"]}
```

Recursive Descent: ..

Recursively descends .., producing every value. This is the same as the zero-argument recurse builtin (see below). This is intended to resemble the XPath // operator. Note that ..a does not work; use .. | .a instead. In the example below we use .. | .a? to find all the values of object keys "a" in any object found "below" ..

This is particularly useful in conjunction with path(EXP) (also see below) and the ? operator.

```
jq ?.. | .a?? 
```

```
[[{"a":1}]]
```

```
=> 1
```

BUILTIN OPERATORS AND FUNCTIONS

Some jq operators (for instance, +) do different things depending on the type of their arguments (arrays, numbers, etc.). However, jq never does implicit type conversions. If you try to add a string to an object you'll get an error message and no result.

Please note that all numbers are converted to IEEE754 double precision floating point representation. Arithmetic and logical operators are working with these converted doubles. Results of all such operations are also limited to the double precision.

The only exception to this behaviour of number is a snapshot of original number literal.

When a number which originally was provided as a literal is never mutated until the end of the program then it is printed to the output in its original literal form. This also in?

cludes cases when the original literal would be truncated when converted to the IEEE754 double precision floating point number.

Addition: +

The operator + takes two filters, applies them both to the same input, and adds the results together. What "adding" means depends on the types involved:

- ? Numbers are added by normal arithmetic.
- ? Arrays are added by being concatenated into a larger array.
- ? Strings are added by being joined into a larger string.
- ? Objects are added by merging, that is, inserting all the key-value pairs from both objects into a single combined object. If both objects contain a value for the same key, the object on the right of the + wins. (For recursive merge use the * operator.)

null can be added to any value, and returns the other value unchanged.

```
jq ?.a + 1?
```

```
{ "a": 7 }
```

```
=> 8
```

```
jq ?.a + .b?
```

```
{ "a": [1,2], "b": [3,4] }
```

```
=> [1,2,3,4]
```

```
jq ?.a + null?
```

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

```
=> 1
```

```
jq ?.a + 1?
```

```
{ }
```

```
=> 1
```

```
jq ?{a: 1} + {b: 2} + {c: 3} + {a: 42}?
```

```
null
```

```
=> { "a": 42, "b": 2, "c": 3 }
```

Subtraction: -

As well as normal arithmetic subtraction on numbers, the - operator can be used on arrays to remove all occurrences of the second array's elements from the first array.

```
jq ?4 - .a?
```

```
{ "a": 3 }
```

```
=> 1
```

```
jq ?. - ["xml", "yaml"]?
["xml", "yaml", "json"]
=> ["json"]
```

Multiplication, division, modulo: *, /, %

These infix operators behave as expected when given two numbers. Division by zero raises an error. $x \% y$ computes x modulo y .

Multiplying a string by a number produces the concatenation of that string that many times.

"x" * 0 produces "".

Dividing a string by another splits the first using the second as separators.

Multiplying two objects will merge them recursively: this works like addition but if both objects contain a value for the same key, and the values are objects, the two are merged with the same strategy.

```
jq ?10 / . * 3?
5
=> 6

jq ?. / ", "?
"a, b,c,d, e"
=> ["a", "b,c,d", "e"]

jq ?{"k": {"a": 1, "b": 2}} * {"k": {"a": 0, "c": 3}}?
null
=> {"k": {"a": 0, "b": 2, "c": 3}}

jq ?.[] | (1 / .)??
[1,0,-1]
=> 1, -1
```

abs

The builtin function `abs` is defined naively as: if $. < 0$ then $- .$ else $.$ end.

For numeric input, this is the absolute value. See the section on the `identity` filter for the implications of this definition for numeric input.

To compute the absolute value of a number as a floating point number, you may wish use `fabs`.

```
jq ?map(abs)?
[-10, -1.1, -1e-1]
=> [10,1.1,1e-1]
```

length

The builtin function `length` gets the length of various different types of value:

- ? The `length` of a string is the number of Unicode codepoints it contains (which will be the same as its JSON-encoded length in bytes if it's pure ASCII).
- ? The length of a number is its absolute value.
- ? The length of an array is the number of elements.
- ? The length of an object is the number of key-value pairs.
- ? The length of null is zero.
- ? It is an error to use `length` on a boolean.

```
jq ?.[] | length?
```

```
[[1,2], "string", {"a":2}, null, -5]
```

```
=> 2, 6, 1, 0, 5
```

`utf8bytlength`

The builtin function `utf8bytlength` outputs the number of bytes used to encode a string in UTF-8.

```
jq ?utf8bytlength?
```

```
"\u03bc"
```

```
=> 2
```

`keys, keys_unsorted`

The builtin function `keys`, when given an object, returns its keys in an array.

The keys are sorted "alphabetically", by unicode codepoint order. This is not an order that makes particular sense in any particular language, but you can count on it being the same for any two objects with the same set of keys, regardless of locale settings.

When `keys` is given an array, it returns the valid indices for that array: the integers from 0 to `length-1`.

The `keys_unsorted` function is just like `keys`, but if the input is an object then the keys will not be sorted, instead the keys will roughly be in insertion order.

```
jq ?keys?
```

```
{"abc": 1, "abcd": 2, "Foo": 3}
```

```
=> ["Foo", "abc", "abcd"]
```

```
jq ?keys?
```

```
[42,3,35]
```

```
=> [0,1,2]
```

`has(key)`

The builtin function `has` returns whether the input object has the given key, or the input array has an element at the given index.

`has($key)` has the same effect as checking whether `$key` is a member of the array returned by `keys`, although `has` will be faster.

```
jq ?map(has("foo"))?
```

```
{ "foo": 42 }, {}
```

```
=> [true, false]
```

```
jq ?map(has(2))?
```

```
[[0,1], ["a","b","c"]]
```

```
=> [false, true]
```

`in`

The builtin function `in` returns whether or not the input key is in the given object, or the input index corresponds to an element in the given array. It is, essentially, an inversed version of `has`.

```
jq ?.[] | in({ "foo": 42 })?
```

```
"foo", "bar"
```

```
=> true, false
```

```
jq ?map(in([0,1]))?
```

```
[2, 0]
```

```
=> [false, true]
```

`map(f)`, `map_values(f)`

For any filter `f`, `map(f)` and `map_values(f)` apply `f` to each of the values in the input array or object, that is, to the values of `.[]`.

In the absence of errors, `map(f)` always outputs an array whereas `map_values(f)` outputs an array if given an array, or an object if given an object.

When the input to `map_values(f)` is an object, the output object has the same keys as the input object except for those keys whose values when piped to `f` produce no values at all.

The key difference between `map(f)` and `map_values(f)` is that the former simply forms an array from all the values of `($x|f)` for each value, `$x`, in the input array or object, but `map_values(f)` only uses `first($x|f)`.

Specifically, for object inputs, `map_values(f)` constructs the output object by examining in turn the value of `first(.[ $k ]|f)` for each key, `$k`, of the input. If this expression produces no values, then the corresponding key will be dropped; otherwise, the output object will

have that value at the key, \$k.

Here are some examples to clarify the behavior of map and map_values when applied to arrays.

These examples assume the input is [1] in all cases:

```
map(.+1)      #=> [2]
map(., .)     #=> [1,1]
map(empty)    #=> []
map_values(.+1) #=> [2]
map_values(., .) #=> [1]
map_values(empty) #=> []
```

map(f) is equivalent to [., | f] and map_values(f) is equivalent to .[] |= f.

In fact, these are their implementations.

```
jq ?map(.+1)?
[1,2,3]
=> [2,3,4]

jq ?map_values(.+1)?
{"a": 1, "b": 2, "c": 3}
=> {"a": 2, "b": 3, "c": 4}

jq ?map(., .)?
[1,2]
=> [1,1,2,2]

jq ?map_values(. // empty)?
{"a": null, "b": true, "c": false}
=> {"b":true}
```

pick(pathexps)

Emit the projection of the input object or array defined by the specified sequence of path expressions, such that if p is any one of these specifications, then (. | p) will evaluate to the same value as (. | pick(pathexps) | p). For arrays, negative indices and .[m:n] specifications should not be used.

```
jq ?pick(.a, .b.c, .x)?
{"a": 1, "b": {"c": 2, "d": 3}, "e": 4}
=> {"a":1,"b":{"c":2},"x":null}

jq ?pick(.[2], .[0], .[0])?
[1,2,3,4]
```

```
=> [1,null,3]
```

`path(path_expression)`

Outputs array representations of the given path expression in `..`. The outputs are arrays of strings (object keys) and/or numbers (array indices).

Path expressions are jq expressions like `.a`, but also `.[ ]`. There are two types of path expressions: ones that can match exactly, and ones that cannot. For example, `.a.b.c` is an exact match path expression, while `.a[ ].b` is not.

`path(exact_path_expression)` will produce the array representation of the path expression even if it does not exist in `..`, if `.` is null or an array or an object.

`path(pattern)` will produce array representations of the paths matching pattern if the paths exist in `..`.

Note that the path expressions are not different from normal expressions. The expression `path(..|select(type=="boolean"))` outputs all the paths to boolean values in `..`, and only those paths.

```
jq ?path(.a[0].b)?
```

```
    null
```

```
=> ["a",0,"b"]
```

```
jq ?[path(..)]?
```

```
    {"a":[{"b":1}]}
```

```
=> [[],["a"],["a",0],["a",0,"b"]]
```

`del(path_expression)`

The builtin function `del` removes a key and its corresponding value from an object.

```
jq ?del(.foo)?
```

```
    {"foo": 42, "bar": 9001, "baz": 42}
```

```
=> {"bar": 9001, "baz": 42}
```

```
jq ?del(.[1, 2])?
```

```
    ["foo", "bar", "baz"]
```

```
=> ["foo"]
```

`getpath(PATHS)`

The builtin function `getpath` outputs the values in `.` found at each path in `PATHS`.

```
jq ?getpath(["a","b"])?
```

```
    null
```

```
=> null
```

```
jq ?[getpath(["a","b"], ["a","c"])]?
```

```
{"a":{"b":0, "c":1}}
```

```
=> [0, 1]
```

setpath(PATHS; VALUE)

The builtin function setpath sets the PATHS in . to VALUE.

```
jq ?setpath(["a","b"]; 1)?
```

```
null
```

```
=> {"a": {"b": 1}}
```

```
jq ?setpath(["a","b"]; 1)?
```

```
{"a":{"b":0}}
```

```
=> {"a": {"b": 1}}
```

```
jq ?setpath([0,"a"]; 1)?
```

```
null
```

```
=> [{"a":1}]
```

delpaths(PATHS)

The builtin function delpaths deletes the PATHS in .. PATHS must be an array of paths, where each path is an array of strings and numbers.

```
jq ?delpaths([["a","b"]])?
```

```
{"a":{"b":1}, "x":{"y":2}}
```

```
=> {"a":{}, "x":{"y":2}}
```

to_entries, from_entries, with_entries(f)

These functions convert between an object and an array of key-value pairs. If to_entries is passed an object, then for each k: v entry in the input, the output array includes {"key": k, "value": v}.

from_entries does the opposite conversion, and with_entries(f) is a shorthand for to_entries

| map(f) | from_entries, useful for doing some operation to all keys and values of an object.

from_entries accepts "key", "Key", "name", "Name", "value", and "Value" as keys.

```
jq ?to_entries?
```

```
{"a": 1, "b": 2}
```

```
=> [{"key":"a", "value":1}, {"key":"b", "value":2}]
```

```
jq ?from_entries?
```

```
[{"key":"a", "value":1}, {"key":"b", "value":2}]
```

```
=> {"a": 1, "b": 2}
```

```
jq ?with_entries(.key |= "KEY_" + .)?
```

```
{ "a": 1, "b": 2 }
```

```
=> { "KEY_a": 1, "KEY_b": 2 }
```

`select(boolean_expression)`

The function `select(f)` produces its input unchanged if `f` returns true for that input, and produces no output otherwise.

It's useful for filtering lists: `[1,2,3] | map(select(. >= 2))` will give you `[2,3]`.

```
jq ?map(select(. >= 2))?
```

```
[1,5,3,0,7]
```

```
=> [5,3,7]
```

```
jq ?.[ ] | select(.id == "second")?
```

```
[{"id": "first", "val": 1}, {"id": "second", "val": 2}]
```

```
=> {"id": "second", "val": 2}
```

arrays, objects, iterables, booleans, numbers, normals, finites, strings, nulls, values, scalars

These built-ins select only inputs that are arrays, objects, iterables (arrays or objects), booleans, numbers, normal numbers, finite numbers, strings, null, non-null values, and non-iterables, respectively.

```
jq ?.[ ]|numbers?
```

```
[[], {}, 1, "foo", null, true, false]
```

```
=> 1
```

`empty`

`empty` returns no results. None at all. Not even null.

It's useful on occasion. You'll know if you need it :)

```
jq ?1, empty, 2?
```

```
null
```

```
=> 1, 2
```

```
jq ?[1,2,empty,3]?
```

```
null
```

```
=> [1,2,3]
```

`error, error(message)`

Produces an error with the input value, or with the message given as the argument. Errors can be caught with `try/catch`; see below.

```
jq ?try error catch .?
```

```
"error message"
```

```
=> "error message"
```

```
jq ?try error("invalid value: \(.)") catch .?
```

```
42
```

```
=> "invalid value: 42"
```

halt

Stops the jq program with no further outputs. jq will exit with exit status 0.

halt_error, halt_error(exit_code)

Stops the jq program with no further outputs. The input will be printed on stderr as raw output (i.e., strings will not have double quotes) with no decoration, not even a newline.

The given exit_code (defaulting to 5) will be jq's exit status.

For example, "Error: something went wrong\n"|halt_error(1).

\$__loc__

Produces an object with a "file" key and a "line" key, with the filename and line number where \$__loc__ occurs, as values.

```
jq ?try error("\($__loc__)") catch .?
```

```
null
```

```
=> "{\"file\":\"<top-level>\",\"line\":1}"
```

paths, paths(node_filter)

paths outputs the paths to all the elements in its input (except it does not output the empty list, representing . itself).

paths(f) outputs the paths to any values for which f is true. That is, paths(type == "number") outputs the paths to all numeric values.

```
jq ?[paths]?
```

```
[1,[],{"a":2}]
```

```
=> [[0],[1],[1,0],[1,1],[1,1,"a"]]
```

```
jq ?[paths(type == "number")]?
```

```
[1,[],{"a":2}]
```

```
=> [[0],[1,1,"a"]]
```

add, add(generator)

The filter add takes as input an array, and produces as output the elements of the array added together. This might mean summed, concatenated or merged depending on the types of the elements of the input array - the rules are the same as those for the + operator (described

above).

If the input is an empty array, add returns null.

add(generator) operates on the given generator rather than the input.

```
jq ?add?
```

```
["a","b","c"]
```

```
=> "abc"
```

```
jq ?add?
```

```
[1, 2, 3]
```

```
=> 6
```

```
jq ?add?
```

```
[]
```

```
=> null
```

```
jq ?add(.[].a)?
```

```
[{"a":3}, {"a":5}, {"b":6}]
```

```
=> 8
```

any, any(condition), any(generator; condition)

The filter any takes as input an array of boolean values, and produces true as output if any of the elements of the array are true.

If the input is an empty array, any returns false.

The any(condition) form applies the given condition to the elements of the input array.

The any(generator; condition) form applies the given condition to all the outputs of the given generator.

```
jq ?any?
```

```
[true, false]
```

```
=> true
```

```
jq ?any?
```

```
[false, false]
```

```
=> false
```

```
jq ?any?
```

```
[]
```

```
=> false
```

all, all(condition), all(generator; condition)

The filter all takes as input an array of boolean values, and produces true as output if all

of the elements of the array are true.

The `all(condition)` form applies the given condition to the elements of the input array.

The `all(generator; condition)` form applies the given condition to all the outputs of the given generator.

If the input is an empty array, `all` returns true.

```
jq ?all?
```

```
[true, false]
```

```
=> false
```

```
jq ?all?
```

```
[true, true]
```

```
=> true
```

```
jq ?all?
```

```
[]
```

```
=> true
```

`flatten`, `flatten(depth)`

The filter `flatten` takes as input an array of nested arrays, and produces a flat array in which all arrays inside the original array have been recursively replaced by their values.

You can pass an argument to it to specify how many levels of nesting to flatten.

`flatten(2)` is like `flatten`, but going only up to two levels deep.

```
jq ?flatten?
```

```
[1, [2], [[3]]]
```

```
=> [1, 2, 3]
```

```
jq ?flatten(1)?
```

```
[1, [2], [[3]]]
```

```
=> [1, 2, [3]]
```

```
jq ?flatten?
```

```
[[]]
```

```
=> []
```

```
jq ?flatten?
```

```
[{"foo": "bar"}, [{"foo": "baz"}]]
```

```
=> [{"foo": "bar"}, {"foo": "baz"}]
```

`range(upto)`, `range(from; upto)`, `range(from; upto; by)`

The `range` function produces a range of numbers. `range(4; 10)` produces 6 numbers, from 4 (in?

clusive) to 10 (exclusive). The numbers are produced as separate outputs. Use [range(4; 10)] to get a range as an array.

The one argument form generates numbers from 0 to the given number, with an increment of 1.

The two argument form generates numbers from from to upto with an increment of 1.

The three argument form generates numbers from to upto with an increment of by.

```
jq ?range(2; 4)?
```

```
    null
```

```
=> 2, 3
```

```
jq ?[range(2; 4)]?
```

```
    null
```

```
=> [2,3]
```

```
jq ?[range(4)]?
```

```
    null
```

```
=> [0,1,2,3]
```

```
jq ?[range(0; 10; 3)]?
```

```
    null
```

```
=> [0,3,6,9]
```

```
jq ?[range(0; 10; -1)]?
```

```
    null
```

```
=> []
```

```
jq ?[range(0; -5; -1)]?
```

```
    null
```

```
=> [0,-1,-2,-3,-4]
```

floor

The floor function returns the floor of its numeric input.

```
jq ?floor?
```

```
    3.14159
```

```
=> 3
```

sqrt

The sqrt function returns the square root of its numeric input.

```
jq ?sqrt?
```

```
    9
```

```
=> 3
```

tonumber

The `tonumber` function parses its input as a number. It will convert correctly-formatted strings to their numeric equivalent, leave numbers alone, and give an error on all other input.

```
jq ?.[ ] | tonumber?
```

```
[1, "1"]
```

```
=> 1, 1
```

toboolean

The `toboolean` function parses its input as a boolean. It will convert correctly-formatted strings to their boolean equivalent, leave booleans alone, and give an error on all other input.

```
jq ?.[ ] | toboolean?
```

```
["true", "false", true, false]
```

```
=> true, false, true, false
```

tostring

The `tostring` function prints its input as a string. Strings are left unchanged, and all other values are JSON-encoded.

```
jq ?.[ ] | tostring?
```

```
[1, "1", [1]]
```

```
=> "1", "1", "[1]"
```

type

The `type` function returns the type of its argument as a string, which is one of `null`, `boolean`, `number`, `string`, `array` or `object`.

```
jq ?map(type)?
```

```
[0, false, [], {}, null, "hello"]
```

```
=> ["number", "boolean", "array", "object", "null", "string"]
```

infinite, nan, isinfinite, isnan, isfinite, isnormal

Some arithmetic operations can yield infinities and "not a number" (NaN) values. The `isinfinite` builtin returns true if its input is infinite. The `isnan` builtin returns true if its input is a NaN. The `infinite` builtin returns a positive infinite value. The `nan` builtin returns a NaN. The `isnormal` builtin returns true if its input is a normal number.

Note that division by zero raises an error.

Currently most arithmetic operations operating on infinities, NaNs, and sub-normals do not

raise errors.

```
jq ?.[ ] | (infinite * .) < 0?
```

```
[-1, 1]
```

```
=> true, false
```

```
jq ?infinite, nan | type?
```

```
null
```

```
=> "number", "number"
```

sort, sort_by(path_expression)

The sort functions sorts its input, which must be an array. Values are sorted in the follow?

ing order:

? null

? false

? true

? numbers

? strings, in alphabetical order (by unicode codepoint value)

? arrays, in lexical order

? objects

The ordering for objects is a little complex: first they're compared by comparing their sets of keys (as arrays in sorted order), and if their keys are equal then the values are compared key by key.

sort_by may be used to sort by a particular field of an object, or by applying any jq filter.

sort_by(f) compares two elements by comparing the result of f on each element. When f produces multiple values, it firstly compares the first values, and the second values if the first values are equal, and so on.

```
jq ?sort?
```

```
[8,3,null,6]
```

```
=> [null,3,6,8]
```

```
jq ?sort_by(.foo)?
```

```
[{"foo":4, "bar":10}, {"foo":3, "bar":10}, {"foo":2, "bar":1}]
```

```
=> [{"foo":2, "bar":1}, {"foo":3, "bar":10}, {"foo":4, "bar":10}]
```

```
jq ?sort_by(.foo, .bar)?
```

```
[{"foo":4, "bar":10}, {"foo":3, "bar":20}, {"foo":2, "bar":1}, {"foo":3, "bar":10}]
```

```
=> [{"foo":2, "bar":1}, {"foo":3, "bar":10}, {"foo":3, "bar":20}, {"foo":4, "bar":10}]
```

group_by(path_expression)

group_by(.foo) takes as input an array, groups the elements having the same .foo field into separate arrays, and produces all of these arrays as elements of a larger array, sorted by the value of the .foo field.

Any jq expression, not just a field access, may be used in place of .foo. The sorting order is the same as described in the sort function above.

```
jq ?group_by(.foo)?
```

```
[{"foo":1, "bar":10}, {"foo":3, "bar":100}, {"foo":1, "bar":1}]
```

```
=> [[{"foo":1, "bar":10}, {"foo":1, "bar":1}], [{"foo":3, "bar":100}]]
```

min, max, min_by(path_exp), max_by(path_exp)

Find the minimum or maximum element of the input array.

The min_by(path_exp) and max_by(path_exp) functions allow you to specify a particular field or property to examine, e.g. min_by(.foo) finds the object with the smallest foo field.

```
jq ?min?
```

```
[5,4,2,7]
```

```
=> 2
```

```
jq ?max_by(.foo)?
```

```
[{"foo":1, "bar":14}, {"foo":2, "bar":3}]
```

```
=> {"foo":2, "bar":3}
```

unique, unique_by(path_exp)

The unique function takes as input an array and produces an array of the same elements, in sorted order, with duplicates removed.

The unique_by(path_exp) function will keep only one element for each value obtained by applying the argument. Think of it as making an array by taking one element out of every group produced by group.

```
jq ?unique?
```

```
[1,2,5,3,5,3,1,3]
```

```
=> [1,2,3,5]
```

```
jq ?unique_by(.foo)?
```

```
[{"foo": 1, "bar": 2}, {"foo": 1, "bar": 3}, {"foo": 4, "bar": 5}]
```

```
=> [{"foo": 1, "bar": 2}, {"foo": 4, "bar": 5}]
```

```
jq ?unique_by(length)?
```

```
["chunky", "bacon", "kitten", "cicada", "asparagus"]
```

```
=> ["bacon", "chunky", "asparagus"]
```

reverse

This function reverses an array.

```
jq ?reverse?
```

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

```
=> [4,3,2,1]
```

contains(element)

The filter contains(b) will produce true if b is completely contained within the input. A string B is contained in a string A if B is a substring of A. An array B is contained in an array A if all elements in B are contained in any element in A. An object B is contained in object A if all of the values in B are contained in the value in A with the same key. All other types are assumed to be contained in each other if they are equal.

```
jq ?contains("bar")?
```

```
"foobar"
```

```
=> true
```

```
jq ?contains(["baz", "bar"])?
```

```
["foobar", "foobaz", "blarp"]
```

```
=> true
```

```
jq ?contains(["bazzzzz", "bar"])?
```

```
["foobar", "foobaz", "blarp"]
```

```
=> false
```

```
jq ?contains({foo: 12, bar: [{barp: 12}]})?
```

```
{"foo": 12, "bar": [1,2,{"barp":12, "blip":13}]}
```

```
=> true
```

```
jq ?contains({foo: 12, bar: [{barp: 15}]})?
```

```
{"foo": 12, "bar": [1,2,{"barp":12, "blip":13}]}
```

```
=> false
```

indices(s)

Outputs an array containing the indices in . where s occurs. The input may be an array, in which case if s is an array then the indices output will be those where all elements in . match those of s.

```
jq ?indices(",")?
```

```
"a,b, cd, efg, hijk"
```

```
=> [3,7,12]
```

```
jq ?indices(1)?
```

```
[0,1,2,1,3,1,4]
```

```
=> [1,3,5]
```

```
jq ?indices([1,2])?
```

```
[0,1,2,3,1,4,2,5,1,2,6,7]
```

```
=> [1,8]
```

index(s), rindex(s)

Outputs the index of the first (index) or last (rindex) occurrence of s in the input.

```
jq ?index(", ")?
```

```
"a,b, cd, efg, hijk"
```

```
=> 3
```

```
jq ?index(1)?
```

```
[0,1,2,1,3,1,4]
```

```
=> 1
```

```
jq ?index([1,2])?
```

```
[0,1,2,3,1,4,2,5,1,2,6,7]
```

```
=> 1
```

```
jq ?rindex(", ")?
```

```
"a,b, cd, efg, hijk"
```

```
=> 12
```

```
jq ?rindex(1)?
```

```
[0,1,2,1,3,1,4]
```

```
=> 5
```

```
jq ?rindex([1,2])?
```

```
[0,1,2,3,1,4,2,5,1,2,6,7]
```

```
=> 8
```

inside

The filter inside(b) will produce true if the input is completely contained within b. It is, essentially, an inversed version of contains.

```
jq ?inside("foobar")?
```

```
"bar"
```

```
=> true
```

```
jq ?inside(["foobar", "foobaz", "blarp"])?
```

```
["baz", "bar"]
```

```
=> true
```

```
jq ?inside(["foobar", "foobaz", "blarp"])?
```

```
["bazzzzz", "bar"]
```

```
=> false
```

```
jq ?inside({"foo": 12, "bar": [1,2,{"barp":12, "blip":13}]}?)
```

```
{"foo": 12, "bar": [{"barp": 12}]}
```

```
=> true
```

```
jq ?inside({"foo": 12, "bar": [1,2,{"barp":12, "blip":13}]}?)
```

```
{"foo": 12, "bar": [{"barp": 15}]}
```

```
=> false
```

startswith(str)

Outputs true if . starts with the given string argument.

```
jq ?[.[]|startswith("foo")]?
```

```
["fo", "foo", "barfoo", "foobar", "barfoob"]
```

```
=> [false, true, false, true, false]
```

endswith(str)

Outputs true if . ends with the given string argument.

```
jq ?[.[]|endswith("foo")]?
```

```
["foobar", "barfoo"]
```

```
=> [false, true]
```

combinations, combinations(n)

Outputs all combinations of the elements of the arrays in the input array. If given an argument n, it outputs all combinations of n repetitions of the input array.

```
jq ?combinations?
```

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

```
=> [1, 3], [1, 4], [2, 3], [2, 4]
```

```
jq ?combinations(2)?
```

```
[0, 1]
```

```
=> [0, 0], [0, 1], [1, 0], [1, 1]
```

ltrimstr(str)

Outputs its input with the given prefix string removed, if it starts with it.


```
jq ?[.[]|ltrimstr("foo")]?
```

```
["fo", "foo", "barfoo", "foobar", "afoo"]
```

```
=> ["fo", "", "barfoo", "bar", "afoo"]
```

rtrimstr(str)

Outputs its input with the given suffix string removed, if it ends with it.

```
jq ?[.[]|rtrimstr("foo")]?
```

```
["fo", "foo", "barfoo", "foobar", "foob"]
```

```
=> ["fo", "", "bar", "foobar", "foob"]
```

trimstr(str)

Outputs its input with the given string removed at both ends, if it starts or ends with it.

```
jq ?[.[]|trimstr("foo")]?
```

```
["fo", "foo", "barfoo", "foobarfoo", "foob"]
```

```
=> ["fo", "", "bar", "bar", "b"]
```

trim, ltrim, rtrim

trim trims both leading and trailing whitespace.

ltrim trims only leading (left side) whitespace.

rtrim trims only trailing (right side) whitespace.

Whitespace characters are the usual " ", "\n" "\t", "\r" and also all characters in the Uni?

code character database with the whitespace property. Note that what considers whitespace

might change in the future.

```
jq ?trim, ltrim, rtrim?
```

```
" abc "
```

```
=> "abc", "abc ", " abc"
```

explode

Converts an input string into an array of the string's codepoint numbers.

```
jq ?explode?
```

```
"foobar"
```

```
=> [102,111,111,98,97,114]
```

implode

The inverse of explode.

```
jq ?implode?
```

```
[65, 66, 67]
```

```
=> "ABC"
```

split(str)

Splits an input string on the separator argument.

split can also split on regex matches when called with two arguments (see the regular expressions section below).

```
jq ?split(", ")?  
"a, b,c,d, e, "  
=> ["a","b,c,d","e",""]
```

join(str)

Joins the array of elements given as input, using the argument as separator. It is the inverse of split: that is, running split("foo") | join("foo") over any input string returns said input string.

Numbers and booleans in the input are converted to strings. Null values are treated as empty strings. Arrays and objects in the input are not supported.

```
jq ?join(", ")?  
["a","b,c,d","e"]  
=> "a, b,c,d, e"  
jq ?join(" ")?  
["a",1,2.3,true,null,false]  
=> "a 1 2.3 true false"
```

ascii_downcase, ascii_upcase

Emit a copy of the input string with its alphabetic characters (a-z and A-Z) converted to the specified case.

```
jq ?ascii_upcase?  
"useful but not for ?"  
=> "USEFUL BUT NOT FOR ?"
```

while(cond; update)

The while(cond; update) function allows you to repeatedly apply an update to . until cond is false.

Note that while(cond; update) is internally defined as a recursive jq function. Recursive calls within while will not consume additional memory if update produces at most one output for each input. See advanced topics below.

```
jq ?[while(<.100; .*2)]?
```

```
=> [1,2,4,8,16,32,64]
```

`repeat(exp)`

The `repeat(exp)` function allows you to repeatedly apply expression `exp` to `.` until an error is raised.

Note that `repeat(exp)` is internally defined as a recursive `jq` function. Recursive calls within `repeat` will not consume additional memory if `exp` produces at most one output for each input. See advanced topics below.

```
jq ?[repeat(. * 2, error)]?
```

```
1
```

```
=> [2]
```

`until(cond; next)`

The `until(cond; next)` function allows you to repeatedly apply the expression `next`, initially to `.` then to its own output, until `cond` is true. For example, this can be used to implement a factorial function (see below).

Note that `until(cond; next)` is internally defined as a recursive `jq` function. Recursive calls within `until()` will not consume additional memory if `next` produces at most one output for each input. See advanced topics below.

```
jq ?[.,1]]until(.[0] < 1; [.[0] - 1, .[1] * .[0]])|. [1]?
```

```
4
```

```
=> 24
```

`recurse(f), recurse, recurse(f; condition)`

The `recurse(f)` function allows you to search through a recursive structure, and extract interesting data from all levels. Suppose your input represents a filesystem:

```
{ "name": "/", "children": [
  { "name": "/bin", "children": [
    { "name": "/bin/lis", "children": [] },
    { "name": "/bin/sh", "children": [] } ] },
  { "name": "/home", "children": [
    { "name": "/home/stephen", "children": [
      { "name": "/home/stephen/jq", "children": [] } ] } ] } ] }
```

Now suppose you want to extract all of the filenames present. You need to retrieve `.name`, `.children[].name`, `.children[].children[].name`, and so on. You can do this with:

```
recurse(.children[]) | .name
```

When called without an argument, `recurse` is equivalent to `recurse(.[]?)`.

`recurse(f)` is identical to `recurse(f; true)` and can be used without concerns about recursion depth.

`recurse(f; condition)` is a generator which begins by emitting `.` and then emits in turn `.|f`,

`.|f|f`, `.|f|f|f`, ... so long as the computed value satisfies the condition. For example, to

generate all the integers, at least in principle, one could write `recurse(.+1; true)`.

The recursive calls in `recurse` will not consume additional memory whenever `f` produces at most a single output for each input.

```
jq ?recurse(.foo[])?
```

```
  {"foo":{"foo": []}, {"foo":{"foo":{"foo":[]}}}}
```

```
=> {"foo":{"foo":[]}, {"foo":{"foo":{"foo":[]}}}, {"foo":[]}, {"foo":{"foo":[]}}, {"foo":[]}
```

```
jq ?recurse?
```

```
  {"a":0,"b":[1]}
```

```
=> {"a":0,"b":[1]}, 0, [1], 1
```

```
jq ?recurse(. * .; . < 20)?
```

```
  2
```

```
=> 2, 4, 16
```

`walk(f)`

The `walk(f)` function applies `f` recursively to every component of the input entity. When an

array is encountered, `f` is first applied to its elements and then to the array itself; when

an object is encountered, `f` is first applied to all the values and then to the object. In

practice, `f` will usually test the type of its input, as illustrated in the following exam-

ples. The first example highlights the usefulness of processing the elements of an array of

arrays before processing the array itself. The second example shows how all the keys of all

the objects within the input can be considered for alteration.

```
jq ?walk(if type == "array" then sort else . end)?
```

```
  [[4, 1, 7], [8, 5, 2], [3, 6, 9]]
```

```
=> [[1,4,7],[2,5,8],[3,6,9]]
```

```
jq ?walk( if type == "object" then with_entries( .key |= sub( "^_"+"; "" ) ) else . end )?
```

```
  [ { "_a": { "__b": 2 } } ]
```

```
=> [{"a":{"b":2}}]
```

`have_literal_numbers`

This builtin returns true if jq's build configuration includes support for preservation of

input number literals.

have_decnum

This builtin returns true if jq was built with "decnum", which is the current literal number preserving numeric backend implementation for jq.

\$JQ_BUILD_CONFIGURATION

This builtin binding shows the jq executable's build configuration. Its value has no particular format, but it can be expected to be at least the ./configure command-line arguments, and may be enriched in the future to include the version strings for the build tooling used.

Note that this can be overridden in the command-line with --arg and related options.

\$ENV, env

\$ENV is an object representing the environment variables as set when the jq program started.

env outputs an object representing jq's current environment.

At the moment there is no builtin for setting environment variables.

```
jq ?$ENV.PAGER?
```

```
    null
```

```
=> "less"
```

```
jq ?env.PAGER?
```

```
    null
```

```
=> "less"
```

transpose

Transpose a possibly jagged matrix (an array of arrays). Rows are padded with nulls so the result is always rectangular.

```
jq ?transpose?
```

```
[[1], [2,3]]
```

```
=> [[1,2],[null,3]]
```

bsearch(x)

bsearch(x) conducts a binary search for x in the input array. If the input is sorted and contains x, then bsearch(x) will return its index in the array; otherwise, if the array is sorted, it will return (-1 - ix) where ix is an insertion point such that the array would still be sorted after the insertion of x at ix. If the array is not sorted, bsearch(x) will return an integer that is probably of no interest.

```
jq ?bsearch(0)?
```

```
[0,1]
```

```
=> 0
```

```
jq ?bsearch(0)?
```

```
[1,2,3]
```

```
=> -1
```

```
jq ?bsearch(4) as $ix | if $ix < 0 then .[-(1+$ix)] = 4 else . end?
```

```
[1,2,3]
```

```
=> [1,2,3,4]
```

String interpolation: `\(exp)`

Inside a string, you can put an expression inside parens after a backslash. Whatever the expression returns will be interpolated into the string.

```
jq ?"The input was \(.), which is one less than \(.+1)"?
```

```
42
```

```
=> "The input was 42, which is one less than 43"
```

Convert to/from JSON

The `tojson` and `fromjson` builtins dump values as JSON texts or parse JSON texts into values, respectively. The `tojson` builtin differs from `tostring` in that `tostring` returns strings unmodified, while `tojson` encodes strings as JSON strings.

```
jq ?[.[]|tostring]?
```

```
[1, "foo", ["foo"]]
```

```
=> ["1","foo",["foo"]]
```

```
jq ?[.[]|tojson]?
```

```
[1, "foo", ["foo"]]
```

```
=> ["1","foo",["foo"]]
```

```
jq ?[.[]|tojson|fromjson]?
```

```
[1, "foo", ["foo"]]
```

```
=> [1,"foo",["foo"]]
```

Format strings and escaping

The `@foo` syntax is used to format and escape strings, which is useful for building URLs, documents in a language like HTML or XML, and so forth. `@foo` can be used as a filter on its own, the possible escapings are:

`@text:`

Calls `tostring`, see that function for details.

`@json:`

Serializes the input as JSON.

@html:

Applies HTML/XML escaping, by mapping the characters `<>&?"` to their entity equivalents `<`, `>`, `&`, `'`, `"`.

@uri:

Applies percent-encoding, by mapping all reserved URI characters to a `%XX` sequence.

@urid:

The inverse of `@uri`, applies percent-decoding, by mapping all `%XX` sequences to their corresponding URI characters.

@csv:

The input must be an array, and it is rendered as CSV with double quotes for strings, and quotes escaped by repetition.

@tsv:

The input must be an array, and it is rendered as TSV (tab-separated values). Each input array will be printed as a single line. Fields are separated by a single tab (ascii 0x09). Input characters line-feed (ascii 0x0a), carriage-return (ascii 0x0d), tab (ascii 0x09) and backslash (ascii 0x5c) will be output as escape sequences `\n`, `\r`, `\t`, `\\` respectively.

@sh:

The input is escaped suitable for use in a command-line for a POSIX shell. If the input is an array, the output will be a series of space-separated strings.

@base64:

The input is converted to base64 as specified by RFC 4648.

@base64d:

The inverse of `@base64`, input is decoded as specified by RFC 4648. Note: If the decoded string is not UTF-8, the results are undefined.

This syntax can be combined with string interpolation in a useful way. You can follow a `@foo` token with a string literal. The contents of the string literal will not be escaped. However, all interpolations made inside that string literal will be escaped. For instance,

```
@uri "https://www.google.com/search?q=\.search)"
```

will produce the following output for the input `{"search":"what is jq?"}`:

```
"https://www.google.com/search?q=what%20is%20jq%3F"
```

Note that the slashes, question mark, etc. in the URL are not escaped, as they were part of

the string literal.

```
jq ?@html?
```

```
"This works if x < y"
```

```
=> "This works if x &lt; y"
```

```
jq ?@sh "echo \(.)"?
```

```
"O?Hara?s Ale"
```

```
=> "echo ?O?\??Hara?\??s Ale?"
```

```
jq ?@base64?
```

```
"This is a message"
```

```
=> "VGhpcyBpcyBhIG1lc3NhZ2U="
```

```
jq ?@base64d?
```

```
"VGhpcyBpcyBhIG1lc3NhZ2U="
```

```
=> "This is a message"
```

Dates

jq provides some basic date handling functionality, with some high-level and low-level builtins. In all cases these builtins deal exclusively with time in UTC.

The `fromdateiso8601` builtin parses datetimes in the ISO 8601 format to a number of seconds since the Unix epoch (1970-01-01T00:00:00Z). The `todateiso8601` builtin does the inverse.

The `fromdate` builtin parses datetime strings. Currently `fromdate` only supports ISO 8601 datetime strings, but in the future it will attempt to parse datetime strings in more formats.

The `todate` builtin is an alias for `todateiso8601`.

The `now` builtin outputs the current time, in seconds since the Unix epoch.

Low-level jq interfaces to the C-library time functions are also provided: `strptime`, `strftime`, `time`, `strflocaltime`, `mktime`, `gmtime`, and `localtime`. Refer to your host operating system's documentation for the format strings used by `strptime` and `strftime`. Note: these are not necessarily stable interfaces in jq, particularly as to their localization functionality.

The `gmtime` builtin consumes a number of seconds since the Unix epoch and outputs a "broken down time" representation of Greenwich Mean Time as an array of numbers representing (in this order): the year, the month (zero-based), the day of the month (one-based), the hour of the day, the minute of the hour, the second of the minute, the day of the week, and the day of the year -- all one-based unless otherwise stated. The day of the week number may be wrong on some systems for dates before March 1st 1900, or after December 31 2099.

The `localtime` builtin works like the `gmtime` builtin, but using the local timezone setting.

The `mktime` builtin consumes "broken down time" representations of time output by `gmtime` and `strptime`.

The `strptime(fmt)` builtin parses input strings matching the `fmt` argument. The output is in the "broken down time" representation consumed by `mktime` and output by `gmtime`.

The `strftime(fmt)` builtin formats a time (GMT) with the given format. The `strflocaltime` does the same, but using the local timezone setting.

The format strings for `strptime` and `strftime` are described in typical C library documentation. The format string for ISO 8601 datetime is `"%Y-%m-%dT%H:%M:%SZ"`.

`jq` may not support some or all of this date functionality on some systems. In particular, the `%u` and `%j` specifiers for `strptime(fmt)` are not supported on macOS.

```
jq ?fromdate?
```

```
"2015-03-05T23:51:47Z"
```

```
=> 1425599507
```

```
jq ?strptime("%Y-%m-%dT%H:%M:%SZ")?
```

```
"2015-03-05T23:51:47Z"
```

```
=> [2015,2,5,23,51,47,4,63]
```

```
jq ?strptime("%Y-%m-%dT%H:%M:%SZ")|mktime?
```

```
"2015-03-05T23:51:47Z"
```

```
=> 1425599507
```

SQL-Style Operators

`jq` provides a few SQL-style operators.

`INDEX(stream; index_expression):`

This builtin produces an object whose keys are computed by the given index expression applied to each value from the given stream.

`JOIN($idx; stream; idx_expr; join_expr):`

This builtin joins the values from the given stream to the given index. The index's keys are computed by applying the given index expression to each value from the given stream. An array of the value in the stream and the corresponding value from the index is fed to the given join expression to produce each result.

`JOIN($idx; stream; idx_expr):`

Same as `JOIN($idx; stream; idx_expr; .)`.

`JOIN($idx; idx_expr):`

This builtin joins the input . to the given index, applying the given index expression to . to compute the index key. The join operation is as described above.

IN(s):

This builtin outputs true if . appears in the given stream, otherwise it outputs false.

IN(source; s):

This builtin outputs true if any value in the source stream appears in the second stream, otherwise it outputs false.

builtins

Returns a list of all builtin functions in the format name/arity. Since functions with the same name but different arities are considered separate functions, all/0, all/1, and all/2 would all be present in the list.

CONDITIONALS AND COMPARISONS

==, !=

The expression ?a == b? will produce ?true? if the results of evaluating a and b are equal (that is, if they represent equivalent JSON values) and ?false? otherwise. In particular, strings are never considered equal to numbers. In checking for the equality of JSON objects, the ordering of keys is irrelevant. If you're coming from JavaScript, please note that jq?s == is like JavaScript?s ===, the "strict equality" operator.

!= is "not equal", and ?a != b? returns the opposite value of ?a == b?

```
jq ?. == false?
```

```
    null
```

```
=> false
```

```
jq ?. == {"b": {"d": (4 + 1e-20), "c": 3}, "a":1}?
```

```
    {"a":1, "b": {"c": 3, "d": 4}}
```

```
=> true
```

```
jq ?.[] == 1?
```

```
    [1, 1.0, "1", "banana"]
```

```
=> true, true, false, false
```

if-then-else-end

if A then B else C end will act the same as B if A produces a value other than false or null, but act the same as C otherwise.

if A then B end is the same as if A then B else . end. That is, the else branch is op?

tional, and if absent is the same as .. This also applies to elif with absent ending else branch.

Checking for false or null is a simpler notion of "truthiness" than is found in JavaScript or Python, but it means that you'll sometimes have to be more explicit about the condition you want. You can't test whether, e.g. a string is empty using if .name then A else B end; you'll need something like if .name == "" then A else B end instead.

If the condition A produces multiple results, then B is evaluated once for each result that is not false or null, and C is evaluated once for each false or null.

More cases can be added to an if using elif A then B syntax.

```
jq ?if . == 0 then
```

```
  "zero"
```

```
elif . == 1 then
```

```
  "one"
```

```
else
```

```
  "many"
```

```
end?
```

```
2
```

```
=> "many"
```

>, >=, <=, <

The comparison operators >, >=, <=, < return whether their left argument is greater than, greater than or equal to, less than or equal to or less than their right argument (respectively).

The ordering is the same as that described for sort, above.

```
jq ?. < 5?
```

```
2
```

```
=> true
```

and, or, not

jq supports the normal Boolean operators and, or, not. They have the same standard of truth as if expressions - false and null are considered "false values", and anything else is a "true value".

If an operand of one of these operators produces multiple results, the operator itself will produce a result for each input.

not is in fact a builtin function rather than an operator, so it is called as a filter to

which things can be piped rather than with special syntax, as in `.foo` and `.bar | not`.

These three only produce the values `true` and `false`, and so are only useful for genuine Boolean operations, rather than the common Perl/Python/Ruby idiom of "value_that_may_be_null or default". If you want to use this form of "or", picking between two values rather than evaluating a condition, see the `//` operator below.

```
jq ?42 and "a string"?
```

```
    null
```

```
=> true
```

```
jq ?(true, false) or false?
```

```
    null
```

```
=> true, false
```

```
jq ?(true, true) and (true, false)?
```

```
    null
```

```
=> true, false, true, false
```

```
jq ?[true, false | not]?
```

```
    null
```

```
=> [false, true]
```

Alternative operator: `//`

The `//` operator produces all the values of its left-hand side that are neither `false` nor `null`. If the left-hand side produces no values other than `false` or `null`, then `//` produces all the values of its right-hand side.

A filter of the form `a // b` produces all the results of `a` that are not `false` or `null`. If `a` produces no results, or no results other than `false` or `null`, then `a // b` produces the results of `b`.

This is useful for providing defaults: `.foo // 1` will evaluate to `1` if there's no `.foo` element in the input. It's similar to how `or` is sometimes used in Python (jq's `or` operator is reserved for strictly Boolean operations).

Note: `some_generator // defaults_here` is not the same as `some_generator | . // defaults_here`. The latter will produce default values for all non-`false`, non-`null` values of the left-hand side, while the former will not. Precedence rules can make this confusing. For example, in `false, 1 // 2` the left-hand side of `//` is `1`, not `false, 1 -- false, 1 // 2` parses the same way as `false, (1 // 2)`. In `(false, null, 1) | . // 42` the left-hand side of `//` is `.`, which always produces just one value, while in `(false, null, 1) // 42` the left-hand

side is a generator of three values, and since it produces a value other false and null, the default 42 is not produced.

```
jq ?empty // 42?
```

```
    null
```

```
=> 42
```

```
jq ?.foo // 42?
```

```
    {"foo": 19}
```

```
=> 19
```

```
jq ?.foo // 42?
```

```
    {}
```

```
=> 42
```

```
jq ?(false, null, 1) // 42?
```

```
    null
```

```
=> 1
```

```
jq ?(false, null, 1) | . // 42?
```

```
    null
```

```
=> 42, 42, 1
```

try-catch

Errors can be caught by using `try EXP catch EXP`. The first expression is executed, and if it fails then the second is executed with the error message. The output of the handler, if any, is output as if it had been the output of the expression to try.

The `try EXP` form uses `empty` as the exception handler.

```
jq ?try .a catch ". is not an object"?
```

```
    true
```

```
=> ". is not an object"
```

```
jq ?[.[]|try .a]?
```

```
    [{}, true, {"a":1}]
```

```
=> [null, 1]
```

```
jq ?try error("some exception") catch .?
```

```
    true
```

```
=> "some exception"
```

Breaking out of control structures

A convenient use of `try/catch` is to break out of control structures like `reduce`, `foreach`,

while, and so on.

For example:

```
# Repeat an expression until it raises "break" as an
# error, then stop repeating without re-raising the error.
# But if the error caught is not "break" then re-raise it.
try repeat(exp) catch if .=="break" then empty else error
```

jq has a syntax for named lexical labels to "break" or "go (back) to":

```
label $out | ... break $out ...
```

The `break $label_name` expression will cause the program to act as though the nearest (to the left) label `$label_name` produced empty.

The relationship between the break and corresponding label is lexical: the label has to be "visible" from the break.

To break out of a reduce, for example:

```
label $out | reduce .[] as $item (null; if .==false then break $out else ... end)
```

The following jq program produces a syntax error:

```
break $out
```

because no label `$out` is visible.

Error Suppression / Optional Operator: ?

The `? operator`, used as `EXP?`, is shorthand for `try EXP`.

```
jq ?[.[] | .a?]?
[{}, true, {"a":1}]
=> [null, 1]

jq ?[.[] | tonumber?]?
["1", "invalid", "3", 4]
=> [1, 3, 4]
```

REGULAR EXPRESSIONS

jq uses the Oniguruma regular expression library, as do PHP, TextMate, Sublime Text, etc, so the description here will focus on jq specifics.

Oniguruma supports several flavors of regular expression, so it is important to know that jq uses the "Perl NG" (Perl with named groups) flavor.

The jq regex filters are defined so that they can be used using one of these patterns:

```
STRING | FILTER(REGEX)
STRING | FILTER(REGEX; FLAGS)
```

STRING | FILTER([REGEX])

STRING | FILTER([REGEX, FLAGS])

where:

? STRING, REGEX, and FLAGS are jq strings and subject to jq string interpolation;

? REGEX, after string interpolation, should be a valid regular expression;

? FILTER is one of test, match, or capture, as described below.

Since REGEX must evaluate to a JSON string, some characters that are needed to form a regular expression must be escaped. For example, the regular expression \s signifying a whitespace character would be written as "\\s".

FLAGS is a string consisting of one or more of the supported flags:

? g - Global search (find all matches, not just the first)

? i - Case insensitive search

? m - Multi line mode (. will match newlines)

? n - Ignore empty matches

? p - Both s and m modes are enabled

? s - Single line mode (^ -> \A, \$ -> \Z)

? l - Find longest possible matches

? x - Extended regex format (ignore whitespace and comments)

To match a whitespace with the x flag, use \s, e.g.

```
jq -n ?"a b" | test("a\\sb"; "x")?
```

Note that certain flags may also be specified within REGEX, e.g.

```
jq -n ?("test", "tEst", "teST", "TEST") | test("(?i)te(?-i)st")?
```

evaluates to: true, true, false, false.

test(val), test(regex; flags)

Like match, but does not return match objects, only true or false for whether or not the regex matches the input.

```
jq ?test("foo")?
```

```
"foo"
```

```
=> true
```

```
jq ?.[ ] | test("a b c # spaces are ignored"; "ix")?
```

```
["xabcd", "ABC"]
```

```
=> true, true
```

match(val), match(regex; flags)

match outputs an object for each match it finds. Matches have the following fields:

- ? offset - offset in UTF-8 codepoints from the beginning of the input
- ? length - length in UTF-8 codepoints of the match
- ? string - the string that it matched
- ? captures - an array of objects representing capturing groups.

Capturing group objects have the following fields:

- ? offset - offset in UTF-8 codepoints from the beginning of the input
- ? length - length in UTF-8 codepoints of this capturing group
- ? string - the string that was captured
- ? name - the name of the capturing group (or null if it was unnamed)

Capturing groups that did not match anything return an offset of -1

```
jq ?match("(abc)+"; "g")?
```

```
"abc abc"
```

```
=> {"offset": 0, "length": 3, "string": "abc", "captures": [{"offset": 0, "length": 3, "string": "abc", "name": null}], {"offset":
```

```
4, "length": 3, "string": "abc", "captures": [{"offset": 4, "length": 3, "string": "abc", "name": null}]}
```

```
jq ?match("foo")?
```

```
"foo bar foo"
```

```
=> {"offset": 0, "length": 3, "string": "foo", "captures": []}
```

```
jq ?match(["foo", "ig"])?
```

```
"foo bar FOO"
```

```
=> {"offset": 0, "length": 3, "string": "foo", "captures": []}, {"offset": 8, "length": 3, "string": "FOO", "captures": []}
```

```
jq ?match("foo (?<bar123>bar)? foo"; "ig")?
```

```
"foo bar foo foo foo"
```

```
=> {"offset": 0, "length": 11, "string": "foo bar foo", "captures": [{"offset": 4, "length": 3, "string": "bar", "name":  
"bar123"}]}, {"offset": 12, "length": 8, "string": "foo foo", "captures": [{"offset": -1, "length": 0, "string": null, "name": "bar123"}]}
```

```
jq ?[ match(".", "g")] | length?
```

```
"abc"
```

```
=> 3
```

capture(val), capture(regex; flags)

Collects the named captures in a JSON object, with the name of each capture as the key, and the matched string as the corresponding value.

```
jq ?capture("(?<a>[a-z]+)-(?<n>[0-9]+)")?
```

```
"xyzy-14"
```



```
=> { "a": "xyzzzy", "n": "14" }
```

`scan(regex), scan(regex; flags)`

Emit a stream of the non-overlapping substrings of the input that match the regex in accordance with the flags, if any have been specified. If there is no match, the stream is empty.

To capture all the matches for each input string, use the idiom `[ expr ]`, e.g. `[ scan(regex) ]`. If the regex contains capturing groups, the filter emits a stream of arrays, each of which contains the captured strings.

```
jq ?scan("c")?
```

```
"abcdefabc"
```

```
=> "c", "c"
```

```
jq ?scan("(a+)(b+)")?
```

```
"abaabbbaabbbb"
```

```
=> ["a","b"], ["aa","bb"], ["aaa","bbb"]
```

`split(regex; flags)`

Splits an input string on each regex match.

For backwards compatibility, when called with a single argument, `split` splits on a string, not a regex.

```
jq ?split(",", *); null)?
```

```
"ab,cd, ef"
```

```
=> ["ab","cd","ef"]
```

`splits(regex), splits(regex; flags)`

These provide the same results as their `split` counterparts, but as a stream instead of an array.

```
jq ?splits(",", *)?
```

```
"ab,cd, ef, gh"
```

```
=> "ab", "cd", "ef", "gh"
```

```
jq ?splits(",? *"; "n")?
```

```
"ab,cd ef, gh"
```

```
=> "ab", "cd", "ef", "gh"
```

`sub(regex; tostring), sub(regex; tostring; flags)`

Emit the string obtained by replacing the first match of `regex` in the input string with `tostring`, after interpolation. `tostring` should be a jq string or a stream of such strings, each of which may contain references to named captures. The named captures are, in effect,

presented as a JSON object (as constructed by capture) to `tostring`, so a reference to a captured variable named "x" would take the form: `"\(.x)"`.

```
jq ?sub("[^a-z]*(?<x>[a-z]+)"; "Z\(.x)"; "g")?  
  "123abc456def"  
=> "ZabcZdef"  
  
jq ?[sub("(?<a>.)"; "\(.a|ascii_uppercase)", "\(.a|ascii_lowercase)")]?  
  "aB"  
=> ["AB", "aB"]
```

`gsub(regex; tostring)`, `gsub(regex; tostring; flags)`

`gsub` is like `sub` but all the non-overlapping occurrences of the regex are replaced by `tostring`, after interpolation. If the second argument is a stream of jq strings, then `gsub` will produce a corresponding stream of JSON strings.

```
jq ?gsub("(?<x>.)[^a]*"; "+\(.x)-")?  
  "AbcabC"  
=> "+A-+a-"  
  
jq ?[gsub("p"; "a", "b")]?  
  "p"  
=> ["a", "b"]
```

ADVANCED FEATURES

Variables are an absolute necessity in most programming languages, but they're relegated to an "advanced feature" in jq.

In most languages, variables are the only means of passing around data. If you calculate a value, and you want to use it more than once, you'll need to store it in a variable. To pass a value to another part of the program, you'll need that part of the program to define a variable (as a function parameter, object member, or whatever) in which to place the data. It is also possible to define functions in jq, although this is a feature whose biggest use is defining jq's standard library (many jq functions such as `map` and `select` are in fact written in jq).

jq has reduction operators, which are very powerful but a bit tricky. Again, these are mostly used internally, to define some useful bits of jq's standard library.

It may not be obvious at first, but jq is all about generators (yes, as often found in other languages). Some utilities are provided to help deal with generators.

Some minimal I/O support (besides reading JSON from standard input, and writing JSON to

standard output) is available.

Finally, there is a module/library system.

Variable / Symbolic Binding Operator: ... as \$identifier | ...

In jq, all filters have an input and an output, so manual plumbing is not necessary to pass a value from one part of a program to the next. Many expressions, for instance `a + b`, pass their input to two distinct subexpressions (here `a` and `b` are both passed the same input), so variables aren't usually necessary in order to use a value twice.

For instance, calculating the average value of an array of numbers requires a few variables in most languages - at least one to hold the array, perhaps one for each element or for a loop counter. In jq, it's simply `add / length` - the `add` expression is given the array and produces its sum, and the `length` expression is given the array and produces its length.

So, there's generally a cleaner way to solve most problems in jq than defining variables.

Still, sometimes they do make things easier, so jq lets you define variables using `expression as $variable`. All variable names start with `$`. Here's a slightly uglier version of the array-averaging example:

```
length as $array_length | add / $array_length
```

We'll need a more complicated problem to find a situation where using variables actually makes our lives easier.

Suppose we have an array of blog posts, with "author" and "title" fields, and another object which is used to map author usernames to real names. Our input looks like:

```
{ "posts": [{ "title": "First post", "author": "anon"},
             { "title": "A well-written article", "author": "person1" } ],
  "realnames": { "anon": "Anonymous Coward",
                 "person1": "Person McPherson" } }
```

We want to produce the posts with the author field containing a real name, as in:

```
{ "title": "First post", "author": "Anonymous Coward" }
{ "title": "A well-written article", "author": "Person McPherson" }
```

We use a variable, `$names`, to store the `realnames` object, so that we can refer to it later when looking up author usernames:

```
.realnames as $names | .posts[] | {title, author: $names[.author]}
```

The expression `exp as $x | ...` means: for each value of expression `exp`, run the rest of the pipeline with the entire original input, and with `$x` set to that value. Thus `as` functions as something of a `foreach` loop.

Just as `{foo}` is a handy way of writing `{foo: .foo}`, so `{foo}` is a handy way of writing `{foo: $foo}`.

Multiple variables may be declared using a single `as` expression by providing a pattern that matches the structure of the input (this is known as "destructuring"):

```
. as {realnames: $names, posts: [$first, $second]} | ...
```

The variable declarations in array patterns (e.g., `. as [$first, $second]`) bind to the elements of the array in from the element at index zero on up, in order. When there is no value at the index for an array pattern element, `null` is bound to that variable.

Variables are scoped over the rest of the expression that defines them, so

```
.realnames as $names | (.posts[] | {title, author: $names[.author]})
```

will work, but

```
(.realnames as $names | .posts[]) | {title, author: $names[.author]}
```

won't.

For programming language theorists, it's more accurate to say that jq variables are lexically-scoped bindings. In particular there's no way to change the value of a binding; one can only setup a new binding with the same name, but which will not be visible where the old one was.

```
jq ?.bar as $x | .foo | . + $x?
```

```
{"foo":10, "bar":200}
```

```
=> 210
```

```
jq ?. as $i|[(.*2|. as $i| $i), $i]?
```

```
5
```

```
=> [10,5]
```

```
jq ?. as [$a, $b, {c: $c}] | $a + $b + $c?
```

```
[2, 3, {"c": 4, "d": 5}]
```

```
=> 9
```

```
jq ?.[] as [$a, $b] | {a: $a, b: $b}?
```

```
[[0], [0, 1], [2, 1, 0]]
```

```
=> {"a":0,"b":null}, {"a":0,"b":1}, {"a":2,"b":1}
```

Destructuring Alternative Operator: `?//`

The destructuring alternative operator provides a concise mechanism for destructuring an input that can take one of several forms.

Suppose we have an API that returns a list of resources and events associated with them, and

we want to get the user_id and timestamp of the first event for each resource. The API (having been clumsily converted from XML) will only wrap the events in an array if the resource has multiple events:

```
{
  "resources": [
    {
      "id": 1,
      "kind": "widget",
      "events": {
        "action": "create",
        "user_id": 1,
        "ts": 13
      }
    },
    {
      "id": 2,
      "kind": "widget",
      "events": [
        {
          "action": "create",
          "user_id": 1,
          "ts": 14
        },
        {
          "action": "destroy",
          "user_id": 1,
          "ts": 15
        }
      ]
    }
  ]
}
```

We can use the destructuring alternative operator to handle this structural change simply:

```
.resources[] as {$id, $kind, events: {$user_id, $ts}} ?// {$id, $kind, events: [{$user_id, $ts}] | {$user_id, $kind, $id, $ts}}
```

Or, if we aren't sure if the input is an array of values or an object:

```
.[] as [$id, $kind, $user_id, $ts] ?// {$id, $kind, $user_id, $ts} | ...
```

Each alternative need not define all of the same variables, but all named variables will be available to the subsequent expression. Variables not matched in the alternative that succeeded will be null:

```
.resources[] as {$id, $kind, events: {$user_id, $ts}} ?// {$id, $kind, events: [{$first_user_id, $first_ts}] | {$user_id, $first_user_id, $kind, $id, $ts, $first_ts}}
```

Additionally, if the subsequent expression returns an error, the alternative operator will attempt to try the next binding. Errors that occur during the final alternative are passed through.

```
[[3]] | .[] as [$a] ?// [$b] | if $a != null then error("err: \"($a)\") else {$a,$b} end
```

```
jq ?.[] as {$a, $b, c: {$d, $e}} ?// {$a, $b, c: [{$d, $e}]} | {$a, $b, $d, $e}?
```

```
{
  "a": 1,
  "b": 2,
  "c": {
    "d": 3,
    "e": 4
  }
}, {
  "a": 1,
  "b": 2,
  "c": {
    "d": 3,
    "e": 4
  }
}
```

```
=> {
  "a": 1, "b": 2, "d": 3, "e": 4,
  "a": 1, "b": 2, "d": 3, "e": 4
}
```

```
jq ?.[] as {$a, $b, c: {$d}} ?// {$a, $b, c: [{$e}]} | {$a, $b, $d, $e}?
```

```
{
  "a": 1,
  "b": 2,
  "c": {
    "d": 3,
    "e": 4
  }
}, {
  "a": 1,
  "b": 2,
  "c": {
    "d": 3,
    "e": 4
  }
}
```

```
=> {
  "a": 1, "b": 2, "d": 3, "e": null,
  "a": 1, "b": 2, "d": null, "e": 4
}
```

```
jq ?.[] as [$a] ?// [$b] | if $a != null then error("err: \"($a)\") else {$a,$b} end?
```

```
[[3]]
```

```
=> {
  "a": null, "b": 3
}
```

Defining Functions

You can give a filter a name using "def" syntax:

```
def increment: . + 1;
```

From then on, increment is usable as a filter just like a builtin function (in fact, this is how many of the builtins are defined). A function may take arguments:

```
def map(f): [.] | f];
```

Arguments are passed as filters (functions with no arguments), not as values. The same argument may be referenced multiple times with different inputs (here `f` is run for each element of the input array). Arguments to a function work more like callbacks than like value arguments. This is important to understand. Consider:

```
def foo(f): f|f;
5|foo(*2)
```

The result will be 20 because `f` is `*2`, and during the first invocation of `f` . will be 5, and the second time it will be 10 ($5 * 2$), so the result will be 20. Function arguments are filters, and filters expect an input when invoked.

If you want the value-argument behaviour for defining simple functions, you can just use a variable:

```
def addvalue(f): f as $f | map(. + $f);
```

Or use the short-hand:

```
def addvalue($f): ...;
```

With either definition, `addvalue(.foo)` will add the current input's `.foo` field to each element of the array. Do note that calling `addvalue([])` will cause the `map(. + $f)` part to be evaluated once per value in the value of `.` at the call site.

Multiple definitions using the same function name are allowed. Each re-definition replaces the previous one for the same number of function arguments, but only for references from functions (or main program) subsequent to the re-definition. See also the section below on scoping.

```
jq ?def addvalue(f): . + [f]; map(addvalue([0]))?
[[1,2],[10,20]]
=> [[1,2,1], [10,20,10]]

jq ?def addvalue(f): f as $x | map(. + $x); addvalue([0])?
[[1,2],[10,20]]
=> [[1,2,1,2], [10,20,1,2]]
```

Scoping

There are two types of symbols in `jq`: value bindings (a.k.a., "variables"), and functions. Both are scoped lexically, with expressions being able to refer only to symbols that have been defined "to the left" of them. The only exception to this rule is that functions can refer to themselves so as to be able to create recursive functions.

For example, in the following expression there is a binding which is visible "to the right" of it, ... | .*3 as \$times_three | [. + \$times_three] | ..., but not "to the left". Consider this expression now, ... | (.*3 as \$times_three | [. + \$times_three]) | ...: here the binding \$times_three is not visible past the closing parenthesis.

`isempty(exp)`

Returns true if exp produces no outputs, false otherwise.

```
jq ?isempty(empty)?
```

```
    null
```

```
=> true
```

```
jq ?isempty([])?
```

```
    []
```

```
=> true
```

```
jq ?isempty([1,2,3])?
```

```
    [1,2,3]
```

```
=> false
```

`limit(n; expr)`

The limit function extracts up to n outputs from expr.

```
jq ?[limit(3; .[])]?
```

```
    [0,1,2,3,4,5,6,7,8,9]
```

```
=> [0,1,2]
```

`skip(n; expr)`

The skip function skips the first n outputs from expr.

```
jq ?[skip(3; .[])]?
```

```
    [0,1,2,3,4,5,6,7,8,9]
```

```
=> [3,4,5,6,7,8,9]
```

`first(expr), last(expr), nth(n; expr)`

The first(expr) and last(expr) functions extract the first and last values from expr, respectively.

The nth(n; expr) function extracts the nth value output by expr. Note that nth(n; expr) doesn't support negative values of n.

```
jq ?[first(range(.)), last(range(.)), nth(5; range(.))]?
```

```
    10
```

```
=> [0,9,5]
```

```
jq ?[first(empty), last(empty), nth(5; empty)]?
```

```
    null
```

```
=> []
```

first, last, nth(n)

The first and last functions extract the first and last values from any array at ..

The nth(n) function extracts the nth value of any array at ..

```
jq ?[range(.)][first, last, nth(5)]?
```

```
    10
```

```
=> [0,9,5]
```

reduce

The reduce syntax allows you to combine all of the results of an expression by accumulating them into a single answer. The form is reduce EXP as \$var (INIT; UPDATE). As an example, we'll pass [1,2,3] to this expression:

```
reduce .[] as $item (0; . + $item)
```

For each result that .[] produces, . + \$item is run to accumulate a running total, starting from 0 as the input value. In this example, .[] produces the results 1, 2, and 3, so the effect is similar to running something like this:

```
0 | 1 as $item | . + $item |
```

```
  2 as $item | . + $item |
```

```
  3 as $item | . + $item
```

```
jq ?reduce .[] as $item (0; . + $item)?
```

```
    [1,2,3,4,5]
```

```
=> 15
```

```
jq ?reduce .[] as [$i,$j] (0; . + $i * $j)?
```

```
    [[1,2],[3,4],[5,6]]
```

```
=> 44
```

```
jq ?reduce .[] as {$x,$y} (null; .x += $x | .y += [$y])?
```

```
    [{"x":"a","y":1},{"x":"b","y":2},{"x":"c","y":3}]
```

```
=> {"x":"abc","y":[1,2,3]}
```

foreach

The foreach syntax is similar to reduce, but intended to allow the construction of limit and reducers that produce intermediate results.

The form is foreach EXP as \$var (INIT; UPDATE; EXTRACT). As an example, we'll pass [1,2,3]

to this expression:

```
foreach .[] as $item (0; . + $item; [$item, . * 2])
```

Like the reduce syntax, `. + $item` is run for each result that `.[]` produces, but `[$item, . * 2]` is run for each intermediate values. In this example, since the intermediate values are 1, 3, and 6, the foreach expression produces [1,2], [2,6], and [3,12]. So the effect is similar to running something like this:

```
0 | 1 as $item | . + $item | [$item, . * 2],  
2 as $item | . + $item | [$item, . * 2],  
3 as $item | . + $item | [$item, . * 2]
```

When `EXTRACT` is omitted, the identity filter is used. That is, it outputs the intermediate values as they are.

```
jq ?foreach .[] as $item (0; . + $item)?  
[1,2,3,4,5]  
=> 1, 3, 6, 10, 15  
  
jq ?foreach .[] as $item (0; . + $item; [$item, . * 2])?  
[1,2,3,4,5]  
=> [1,2], [2,6], [3,12], [4,20], [5,30]  
  
jq ?foreach .[] as $item (0; . + 1; {index: ., $item})?  
["foo", "bar", "baz"]  
=> {"index":1,"item":"foo"}, {"index":2,"item":"bar"}, {"index":3,"item":"baz"}
```

Recursion

As described above, `recurse` uses recursion, and any jq function can be recursive. The `while` builtin is also implemented in terms of recursion.

Tail calls are optimized whenever the expression to the left of the recursive call outputs its last value. In practice this means that the expression to the left of the recursive call should not produce more than one output for each input.

For example:

```
def recurse(f): def r: ., (f | select(. != null) | r); r;  
  
def while(cond; update):  
  def _while:  
    if cond then ., (update | _while) else empty end;  
  _while;  
  
def repeat(exp):
```

```
def _repeat:
  exp, _repeat;
_repeat;
```

Generators and iterators

Some jq operators and functions are actually generators in that they can produce zero, one, or more values for each input, just as one might expect in other programming languages that have generators. For example, `.[ ]` generates all the values in its input (which must be an array or an object), `range(0; 10)` generates the integers between 0 and 10, and so on.

Even the comma operator is a generator, generating first the values generated by the expression to the left of the comma, then the values generated by the expression on the right of the comma.

The empty builtin is the generator that produces zero outputs. The empty builtin backtracks to the preceding generator expression.

All jq functions can be generators just by using builtin generators. It is also possible to construct new generators using only recursion and the comma operator. If recursive calls are "in tail position" then the generator will be efficient. In the example below the recursive call by `_range` to itself is in tail position. The example shows off three advanced topics: tail recursion, generator construction, and sub-functions.

```
jq ?def range(init; upto; by): def _range: if (by > 0 and . < upto) or (by < 0 and . > upto) then ., ((.+by)|_range) else
empty end; if init == upto then empty elif by == 0 then init else init|_range end; range(0; 10; 3)?
null
=> 0, 3, 6, 9

jq ?def while(cond; update): def _while: if cond then ., (update | _while) else empty end; _while; [while(<.100; .*2)]?
1
=> [1,2,4,8,16,32,64]
```

MATH

jq currently only has IEEE754 double-precision (64-bit) floating point number support.

Besides simple arithmetic operators such as `+`, jq also has most standard math functions from the C math library. C math functions that take a single input argument (e.g., `sin()`) are available as zero-argument jq functions. C math functions that take two input arguments (e.g., `pow()`) are available as two-argument jq functions that ignore `..` C math functions that take three input arguments are available as three-argument jq functions that ignore `..`

Availability of standard math functions depends on the availability of the corresponding

math functions in your operating system and C math library. Unavailable math functions will be defined but will raise an error.

One-input C math functions: `acos` `acosh` `asin` `asinh` `atan` `atanh` `cbrt` `ceil` `cos` `cosh` `erf` `erfc` `exp` `exp10` `exp2` `expm1` `fabs` `floor` `gamma` `j0` `j1` `lgamma` `log` `log10` `log1p` `log2` `logb` `nearbyint` `rint` `round` `significand` `sin` `sinh` `sqrt` `tan` `tanh` `tgamma` `trunc` `y0` `y1`.

Two-input C math functions: `atan2` `copysign` `drem` `fdim` `fmax` `fmin` `fmod` `frexp` `hypot` `jn` `ldexp` `modf` `nextafter` `nexttoward` `pow` `remainder` `scalb` `scalbln` `yn`.

Three-input C math functions: `fma`.

See your system's manual for more information on each of these.

I/O

At this time `jq` has minimal support for I/O, mostly in the form of control over when inputs are read. Two builtin functions are provided for this, `input` and `inputs`, that read from the same sources (e.g., `stdin`, files named on the command-line) as `jq` itself. These two builtins, and `jq`'s own reading actions, can be interleaved with each other. They are commonly used in combination with the null input option `-n` to prevent one input from being read implicitly.

Two builtins provide minimal output capabilities, `debug`, and `stderr`. (Recall that a `jq` program's output values are always output as JSON texts on `stdout`.) The `debug` builtin can have application-specific behavior, such as for executables that use the `libjq` C API but aren't the `jq` executable itself. The `stderr` builtin outputs its input in raw mode to `stderr` with no additional decoration, not even a newline.

Most `jq` builtins are referentially transparent, and yield constant and repeatable value streams when applied to constant inputs. This is not true of I/O builtins.

`input`

Outputs one new input.

Note that when using `input` it is generally necessary to invoke `jq` with the `-n` command-line option, otherwise the first entity will be lost.

```
echo 1 2 3 4 | jq ?[., input]? # [1,2] [3,4]
```

`inputs`

Outputs all remaining inputs, one by one.

This is primarily useful for reductions over a program's inputs. Note that when using `inputs` it is generally necessary to invoke `jq` with the `-n` command-line option, otherwise the first entity will be lost.

```
echo 1 2 3 | jq -n 'reduce inputs as $i (0; . + $i)?' # 6
```

`debug`, `debug(msgs)`

These two filters are like `.` but have as a side-effect the production of one or more messages on `stderr`.

The message produced by the `debug` filter has the form

```
["DEBUG:",<input-value>]
```

where `<input-value>` is a compact rendition of the input value. This format may change in the future.

The `debug(msgs)` filter is defined as `(msgs | debug | empty)`, thus allowing great flexibility in the content of the message, while also allowing multi-line debugging statements to be created.

For example, the expression:

```
1 as $x | 2 | debug("Entering function foo with $x == \"($x)\", .) | (.+1)
```

would produce the value 3 but with the following two lines being written to `stderr`:

```
["DEBUG:","Entering function foo with $x == 1"]
```

```
["DEBUG:",2]
```

`stderr`

Prints its input in raw and compact mode to `stderr` with no additional decoration, not even a newline.

`input_filename`

Returns the name of the file whose input is currently being filtered. Note that this will not work well unless `jq` is running in a UTF-8 locale.

`input_line_number`

Returns the line number of the input currently being filtered.

STREAMING

With the `--stream` option `jq` can parse input texts in a streaming fashion, allowing programs to start processing large JSON texts immediately rather than after the parser completes. If you have a single JSON text that is 1GB in size, streaming it will allow you to process it much more quickly.

However, streaming isn't easy to deal with as the `jq` program will have `[<path>, <leaf-value>]` (and a few other forms) as inputs.

Several builtins are provided to make handling streams easier.

The examples below use the streamed form of `["a",["b"]]`, which is

```
[[0],"a"],[[1,0],"b"],[[1,0]],[[1]].
```

Streaming forms include [`<path>`, `<leaf-value>`] (to indicate any scalar value, empty array, or empty object), and [`<path>`] (to indicate the end of an array or object). Future versions of jq run with `--stream` and `--seq` may output additional forms such as ["error message"] when an input text fails to parse.

`truncate_stream(stream_expression)`

Consumes a number as input and truncates the corresponding number of path elements from the left of the outputs of the given streaming expression.

```
jq ?truncate_stream([[[0],"a"],[[1,0],"b"],[[1,0]],[[1]])?
```

```
1
```

```
=> [[0],"b"], [[0]]
```

`fromstream(stream_expression)`

Outputs values corresponding to the stream expression's outputs.

```
jq ?fromstream(1|truncate_stream([[[0],"a"],[[1,0],"b"],[[1,0]],[[1]])?)
```

```
null
```

```
=> ["b"]
```

`tostream`

The `tostream` builtin outputs the streamed form of its input.

```
jq ?. as $dot|fromstream($dot|tostream)|.==$dot?
```

```
[0,[1,{"a":1},{"b":2}]]
```

```
=> true
```

ASSIGNMENT

Assignment works a little differently in jq than in most programming languages. jq doesn't distinguish between references to and copies of something - two objects or arrays are either equal or not equal, without any further notion of being "the same object" or "not the same object".

If an object has two fields which are arrays, `.foo` and `.bar`, and you append something to `.foo`, then `.bar` will not get bigger, even if you've previously set `.bar = .foo`. If you're used to programming in languages like Python, Java, Ruby, JavaScript, etc. then you can think of it as though jq does a full deep copy of every object before it does the assignment (for performance it doesn't actually do that, but that's the general idea).

This means that it's impossible to build circular values in jq (such as an array whose first element is itself). This is quite intentional, and ensures that anything a jq program can

produce can be represented in JSON.

All the assignment operators in jq have path expressions on the left-hand side (LHS). The right-hand side (RHS) provides values to set to the paths named by the LHS path expressions. Values in jq are always immutable. Internally, assignment works by using a reduction to compute new, replacement values for `.` that have had all the desired assignments applied to `.`, then outputting the modified value. This might be made clear by this example: `{a:{b:{c:1}}}` | `(.a.b|=3), ..` This will output `{"a":{"b":3}}` and `{"a":{"b":{"c":1}}}` because the last sub-expression, `..`, sees the original value, not the modified value.

Most users will want to use modification assignment operators, such as `|=` or `+=`, rather than `=`.

Note that the LHS of assignment operators refers to a value in `..`. Thus `$var.foo = 1` won't work as expected (`$var.foo` is not a valid or useful path expression in `.`); use `$var | .foo = 1` instead.

Note too that `.a.b=0` does not set `.a` and `.b`, but `(.a.b)=0` sets both.

Update-assignment: `|=`

This is the "update" operator `|=`. It takes a filter on the right-hand side and works out the new value for the property of `.` being assigned to by running the old value through this expression. For instance, `(.foo, .bar) |= .+1` will build an object with the `foo` field set to the input's `foo` plus 1, and the `bar` field set to the input's `bar` plus 1.

The left-hand side can be any general path expression; see `path()`.

Note that the left-hand side of `|=` refers to a value in `..`. Thus `$var.foo |= . + 1` won't work as expected (`$var.foo` is not a valid or useful path expression in `.`); use `$var | .foo |= . + 1` instead.

If the right-hand side outputs no values (i.e., empty), then the left-hand side path will be deleted, as with `del(path)`.

If the right-hand side outputs multiple values, only the first one will be used (COMPATIBILITY NOTE: in jq 1.5 and earlier releases, it used to be that only the last one was used).

ITY NOTE: in jq 1.5 and earlier releases, it used to be that only the last one was used).

```
jq ?(..|select(type=="boolean")) |= if . then 1 else 0 end?  
[true,false,[5,true,[true,[false]],false]]  
=> [1,0,[5,1,[1,[0]],0]]
```

Arithmetic update-assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `//=`

jq has a few operators of the form `a op= b`, which are all equivalent to `a |= . op b`. So, `+=` can be used to increment values, being the same as `|= . + 1`.

```
jq ?.foo += 1?
```

```
{"foo": 42}
```

```
=> {"foo": 43}
```

Plain assignment: =

This is the plain assignment operator. Unlike the others, the input to the right-hand side (RHS) is the same as the input to the left-hand side (LHS) rather than the value at the LHS path, and all values output by the RHS will be used (as shown below).

If the RHS of = produces multiple values, then for each such value jq will set the paths on the left-hand side to the value and then it will output the modified .. For example, (.a,.b) = range(2) outputs {"a":0,"b":0}, then {"a":1,"b":1}. The "update" assignment forms (see above) do not do this.

This example should show the difference between = and |=:

Provide input {"a": {"b": 10}, "b": 20} to the programs

```
.a = .b
```

and

```
.a |= .b
```

The former will set the a field of the input to the b field of the input, and produce the output {"a": 20, "b": 20}. The latter will set the a field of the input to the a field's b field, producing {"a": 10, "b": 20}.

```
jq ?.a = .b?
```

```
{"a": {"b": 10}, "b": 20}
```

```
=> {"a":20,"b":20}
```

```
jq ?.a |= .b?
```

```
{"a": {"b": 10}, "b": 20}
```

```
=> {"a":10,"b":20}
```

```
jq ?(.a, .b) = range(3)?
```

```
null
```

```
=> {"a":0,"b":0}, {"a":1,"b":1}, {"a":2,"b":2}
```

```
jq ?(.a, .b) |= range(3)?
```

```
null
```

```
=> {"a":0,"b":0}
```

Complex assignments

Lots more things are allowed on the left-hand side of a jq assignment than in most lan?

guages. We've already seen simple field accesses on the left hand side, and it's no surprise that array accesses work just as well:

```
.posts[0].title = "JQ Manual"
```

What may come as a surprise is that the expression on the left may produce multiple results, referring to different points in the input document:

```
.posts[].comments |= . + ["this is great"]
```

That example appends the string "this is great" to the "comments" array of each post in the input (where the input is an object with a field "posts" which is an array of posts).

When jq encounters an assignment like `a = b?`, it records the "path" taken to select a part of the input document while executing a. This path is then used to find which part of the input to change while executing the assignment. Any filter may be used on the left-hand side of an equals - whichever paths it selects from the input will be where the assignment is performed.

This is a very powerful operation. Suppose we wanted to add a comment to blog posts, using the same "blog" input above. This time, we only want to comment on the posts written by "stedolan". We can find those posts using the "select" function described earlier:

```
.posts[] | select(.author == "stedolan")
```

The paths provided by this operation point to each of the posts that "stedolan" wrote, and we can comment on each of them in the same way that we did before:

```
(.posts[] | select(.author == "stedolan") | .comments) |=  
. + ["terrible."]
```

COMMENTS

You can write comments in your jq filters using #.

A # character (not part of a string) starts a comment. All characters from # to the end of the line are ignored.

If the end of the line is preceded by an odd number of backslash characters, the following line is also considered part of the comment and is ignored.

For example, the following code outputs [1,3,4,7]

```
[  
  1,  
  # foo \  
  2,  
  # bar \\  
]
```



```

3,
4, # baz \\
5, \
6,
7
# comment \
  comment \
    comment
]

```

Backslash continuing the comment on the next line can be useful when writing the "shebang" for a jq script:

```

#!/bin/sh --
# total - Output the sum of the given arguments (or stdin)
# usage: total [numbers...]
# \
exec jq --args -MRnf -- "$0" "$@"
$ARGS.positional |
reduce (
  if . == []
  then inputs
  else .[]
end |
. as $dot |
try tonumber catch false |
if not or isnan then
  @json "total: Invalid number \($dot).\n" | halt_error(1)
end
) as $n (0; . + $n)

```

The `exec` line is considered a comment by jq, so it is ignored. But it is not ignored by sh, since in sh a backslash at the end of the line does not continue the comment. With this trick, when the script is invoked as `total 1 2, /bin/sh -- /path/to/total 1 2` will be run, and sh will then run `exec jq --args -MRnf -- /path/to/total 1 2` replacing itself with a jq interpreter invoked with the specified options (-M, -R, -n, --args), that evaluates the cur?

rent file (\$0), with the arguments (\$@) that were passed to sh.

MODULES

jq has a library/module system. Modules are files whose names end in .jq.

Modules imported by a program are searched for in a default search path (see below). The import and include directives allow the importer to alter this path.

Paths in the search path are subject to various substitutions.

For paths starting with ~/, the user's home directory is substituted for ~.

For paths starting with \$ORIGIN/, the directory where the jq executable is located is substituted for \$ORIGIN.

For paths starting with ./ or paths that are ., the path of the including file is substituted for .. For top-level programs given on the command-line, the current directory is used.

Import directives can optionally specify a search path to which the default is appended.

The default search path is the search path given to the -L command-line option, else ["~/jq", "\$ORIGIN/../lib/jq", "\$ORIGIN/../lib"].

Null and empty string path elements terminate search path processing.

A dependency with relative path foo/bar would be searched for in foo/bar.jq and foo/bar/bar.jq in the given search path. This is intended to allow modules to be placed in a directory along with, for example, version control files, README files, and so on, but also to allow for single-file modules.

Consecutive components with the same name are not allowed to avoid ambiguities (e.g., foo/foo).

For example, with -L\$HOME/.jq a module foo can be found in \$HOME/.jq/foo.jq and \$HOME/.jq/foo/foo.jq.

If .jq exists in the user's home directory, and is a file (not a directory), it is automatically sourced into the main program.

```
import RelativePathString as NAME [<metadata>];
```

Imports a module found at the given path relative to a directory in a search path. A .jq suffix will be added to the relative path string. The module's symbols are prefixed with NAME::.

The optional metadata must be a constant jq expression. It should be an object with keys like homepage and so on. At this time jq only uses the search key/value of the metadata. The metadata is also made available to users via the modulemeta builtin.

The search key in the metadata, if present, should have a string or array value (array of strings); this is the search path to be prefixed to the top-level search path.

`include RelativePathString [<metadata>];`

Imports a module found at the given path relative to a directory in a search path as if it were included in place. A .jq suffix will be added to the relative path string. The module's symbols are imported into the caller's namespace as if the module's content had been included directly.

The optional metadata must be a constant jq expression. It should be an object with keys like homepage and so on. At this time jq only uses the search key/value of the metadata. The metadata is also made available to users via the modulemeta builtin.

`import RelativePathString as $NAME [<metadata>];`

Imports a JSON file found at the given path relative to a directory in a search path. A .json suffix will be added to the relative path string. The file's data will be available as \$NAME::NAME.

The optional metadata must be a constant jq expression. It should be an object with keys like homepage and so on. At this time jq only uses the search key/value of the metadata. The metadata is also made available to users via the modulemeta builtin.

The search key in the metadata, if present, should have a string or array value (array of strings); this is the search path to be prefixed to the top-level search path.

`module <metadata>;`

This directive is entirely optional. It's not required for proper operation. It serves only the purpose of providing metadata that can be read with the modulemeta builtin.

The metadata must be a constant jq expression. It should be an object with keys like homepage. At this time jq doesn't use this metadata, but it is made available to users via the modulemeta builtin.

`modulemeta`

Takes a module name as input and outputs the module's metadata as an object, with the module's imports (including metadata) as an array value for the deps key and the module's defined functions as an array value for the defs key.

Programs can use this to query a module's metadata, which they could then use to, for example, search for, download, and install missing dependencies.

COLORS

To configure alternative colors just set the JQ_COLORS environment variable to colon-delim?

ited list of partial terminal escape sequences like "1;31", in this order:

- ? color for null
- ? color for false
- ? color for true
- ? color for numbers
- ? color for strings
- ? color for arrays
- ? color for objects
- ? color for object keys

The default color scheme is the same as setting JQ_COL?

ORS="0;90:0;39:0;39:0;39:0;32:1;39:1;39:1;34".

This is not a manual for VT100/ANSI escapes. However, each of these color specifications should consist of two numbers separated by a semi-colon, where the first number is one of these:

- ? 1 (bright)
- ? 2 (dim)
- ? 4 (underscore)
- ? 5 (blink)
- ? 7 (reverse)
- ? 8 (hidden)

and the second is one of these:

- ? 30 (black)
- ? 31 (red)
- ? 32 (green)
- ? 33 (yellow)
- ? 34 (blue)
- ? 35 (magenta)
- ? 36 (cyan)
- ? 37 (white)

BUGS

Presumably. Report them or discuss them at:

<https://github.com/jqlang/jq/issues>

Stephen Dolan <mu@netsoc.tcd.ie>

May 2025

JQ(1)