URADICAL

POCKET REFERENCE

Command-Line JSON Processing

Alan Bradley

jq Pocket Reference

Command-Line JSON Processing

Alan Bradley

uRadical

Table of Contents

1. Introduction
2. Command-Line Options
3. Basic Filters
4. Types and Values
5. Operators
6. Conditionals and Comparisons
7. Array Operations
8. Object Operations
9. String Functions
10. Path Expressions
11. Reduce and Recursion
12. Defining Functions
13. Input/Output
14. Common Recipes
15. Quick Reference Tables

Chapter 1: Introduction

What is jq?

jq is a lightweight, command-line JSON processor. It allows you to slice, filter, transform, and manipulate structured JSON data with a concise, expressive syntax. Think of it as `sed` or `awk` for JSON.

jq is:

- Fast - Written in portable C with no runtime dependencies
- Expressive - A complete functional language for JSON transformation
- Unix-friendly - Reads from stdin, writes to stdout, composes with pipes

When to Use jq

jq excels when you need to:

- Extract specific fields from API responses
- Transform JSON data between formats
- Filter arrays based on conditions
- Aggregate or summarise data
- Pretty-print JSON for debugging
- Process JSON logs or configuration files
- Script interactions with JSON-outputting CLI tools

Installation

macOS:

```
brew install jq
```

Debian / Ubuntu:

```
sudo apt install jq
```

Fedora:

```
sudo dnf install jq
```

Arch Linux:

```
sudo pacman -S jq
```

Windows (Chocolatey):

```
choco install jq
```

Windows (Scoop):

```
scoop install jq
```

From source:

```
git clone https://github.com/jqlang/jq.git
cd jq
autoreconf -i
./configure
make
sudo make install
```

Verifying Installation

```
jq --version
# jq-1.7.1
```

Basic Usage

jq takes a filter as its first argument and applies it to JSON input:

```
echo '{"name": "Alice", "age": 30}' | jq '.name'
# "Alice"
```

Read from a file:

```
jq '.name' data.json
```

Read from a URL (with curl):

```
curl -s https://api.example.com/data | jq '.results'
```

The Identity Filter

The simplest jq filter is `.` (dot), which outputs the input unchanged but pretty-printed:

```
echo '{"a":1,"b":2}' | jq '.'
# {
#   "a": 1,
#   "b": 2
# }
```

This is often used simply to format minified JSON for readability.

Getting Help

Built-in help:

```
jq --help
```

Online manual:

```
https://jqlang.github.io/jq/manual/
```

Interactive playground:

```
https://jqplay.org
```

Chapter 2: Command-Line Options

Synopsis

```
jq [options] <filter> [files...]
jq [options] --from-file <file> [files...]
```

If no files are specified, jq reads from standard input.

Output Options

-r, --raw-output

Output raw strings without JSON quotes. Essential when piping jq output to other commands.

```
echo '{"name": "Alice"}' | jq '.name'
# "Alice"

echo '{"name": "Alice"}' | jq -r '.name'
# Alice
```

-j, --join-output

Like `-r` but without a trailing newline. Useful when constructing strings from multiple values.

```
echo '["a","b","c"]' | jq -j '.[]'
# abc
```

-c, --compact-output

Output each JSON object on a single line without pretty-printing. Useful for line-oriented processing or reducing file size.

```
echo '{"a": 1, "b": 2}' | jq -c '.'
# {"a":1,"b":2}
```

-C, --color-output

Force coloured output even when piping to another command.

```
jq -C '.' data.json | less -R
```

-M, --monochrome-output

Disable coloured output.

```
jq -M '.' data.json > output.json
```

-S, --sort-keys

Sort object keys alphabetically in output.

```
echo '{"b": 2, "a": 1}' | jq -S '.'
# {
#   "a": 1,
#   "b": 2
# }
```

-a, --ascii-output

Escape non-ASCII characters in output.

```
echo '{"name": "□□□"}' | jq -a '.'
# {
#   "name": "\u65e5\u672c\u8a9e"
# }
```

--tab

Use tabs instead of two spaces for indentation.

```
echo '{"a": {"b": 1}}' | jq --tab '.'
```

--indent n

Set indentation level (0-7). Default is 2.

```
echo '{"a": 1}' | jq --indent 4 '.'
```

Input Options

-n, --null-input

Don't read input. Start with `null`. Useful for generating JSON from scratch.

```
jq -n '{"created": "today", "items": [1,2,3]}'
# {
#   "created": "today",
#   "items": [1, 2, 3]
# }
```

-R, --raw-input

Read input as raw strings, one per line, rather than JSON.

```
echo -e "line1\nline2" | jq -R '.'
# "line1"
# "line2"
```

-s, --slurp

Read entire input into a single array. Without this, jq processes each JSON object independently.

```
echo -e '{"a":1}\n{"a":2}' | jq -s '.'
# [
#   {"a": 1},
#   {"a": 2}
# ]
```

Combine with `-R` to slurp raw lines into an array:

```
echo -e "line1\nline2" | jq -Rs 'split("\n") | map(select(. != ""))'
# ["line1", "line2"]
```

--slurpfile name file

Read JSON file into a variable as an array.

```
echo '{"id": 1}' | jq --slurpfile users users.json '.id as $id | $users[] | select(.id
```

--rawfile name file

Read file contents as a raw string into a variable.

```
jq -n --rawfile template email.txt '{body: $template}'
```

--jsonargs

Treat remaining arguments as JSON values, accessible via `$ARGS.positional`.

```
jq -n --jsonargs '$ARGS.positional' 1 '"hello"' '{"a":1}'
# [1, "hello", {"a": 1}]
```

--args

Treat remaining arguments as strings, accessible via `$ARGS.positional`.

```
jq -n --args '$ARGS.positional' foo bar baz
# ["foo", "bar", "baz"]
```

Variable Options

--arg name value

Pass a string value as a variable.

```
jq -n --arg name "Alice" '{"greeting": "Hello, \($name)"}'
# {"greeting": "Hello, Alice"}
```

--argjson name value

Pass a JSON value as a variable.

```
jq -n --argjson count 42 '{"count": $count}'
# {"count": 42}

jq -n --argjson enabled true '{"enabled": $enabled}'
# {"enabled": true}
```

Execution Options

-e, --exit-status

Set exit status based on output:

- 0 if last output is neither `false` nor `null`
- 1 if last output is `false` or `null`
- 5 if no valid outputs

Useful in shell scripts:

```
if echo '{"active": true}' | jq -e '.active' > /dev/null; then
```

```
    echo "Is active"
fi
```

-f, --from-file file

Read filter from a file instead of command line. Useful for complex filters.

```
jq -f transform.jq data.json
```

-L directory

Prepend directory to the search path for modules/includes.

```
jq -L ./lib 'include "helpers"; myfunction'
```

--seq

Parse input as application/json-seq (RFC 7464).

-i, --in-place

Edit file in place. Requires jq 1.7+.

```
jq -i '.version = "2.0"' package.json
```

Environment

$ENV

Access environment variables within jq:

```
export MY_VAR="hello"
echo '{}' | jq '{value: $ENV.MY_VAR}'
```

```
# {"value": "hello"}
```

env

The `env` object contains all environment variables:

```
echo 'null' | jq -n 'env.HOME'
# "/home/username"
```

Exit Codes

Code	Meaning
0	Success (with `-e`: last output was not false/null)
1	With `-e`: last output was false or null
2	Usage error
3	Compile error in filter
4	Runtime error
5	With `-e`: no valid outputs

Chapter 3: Basic Filters

Identity: .

The identity filter outputs its input unchanged. Used for pretty-printing or as a starting point in pipelines.

```
echo '{"a":1,"b":2}' | jq '.'
# {
#   "a": 1,
#   "b": 2
# }
```

Field Access: .field

Extract a field from an object by name.

```
echo '{"name": "Alice", "age": 30}' | jq '.name'
# "Alice"
```

Nested Fields

Chain field access with dots:

```
echo '{"user": {"name": "Alice", "address": {"city": "London"}}}' | jq '.user.address.
# "London"
```

Fields with Special Characters

Use quotes for field names containing spaces, dots, or special characters:

```
echo '{"first name": "Alice", "user.id": 123}' | jq '."first name"'
# "Alice"
```

```
echo '{"first name": "Alice", "user.id": 123}' | jq '.["user.id"]'
# 123
```

Optional Field Access: .field?

The ? suffix suppresses errors when a field doesn't exist or when the input isn't an object:

```
echo '{"a": 1}' | jq '.b?'
# null

echo '"not an object"' | jq '.field?'
# (no output, no error)

echo '"not an object"' | jq '.field'
# error: Cannot index string with string "field"
```

Array Index: .[n]

Access array elements by zero-based index.

```
echo '["a", "b", "c", "d"]' | jq '.[0]'
# "a"

echo '["a", "b", "c", "d"]' | jq '.[2]'
# "c"
```

Negative Indices

Negative indices count from the end:

```
echo '["a", "b", "c", "d"]' | jq '.[-1]'
# "d"

echo '["a", "b", "c", "d"]' | jq '.[-2]'
# "c"
```

Optional Index: .[n]?

Suppress errors for out-of-bounds access:

```
echo '["a", "b"]' | jq '.[10]?'
# null
```

Array Slice: .[start:end]

Extract a subarray. The start index is inclusive, end is exclusive.

```
echo '["a", "b", "c", "d", "e"]' | jq '.[1:3]'
# ["b", "c"]
```

Open-Ended Slices

Omit start or end for open-ended slices:

```
echo '["a", "b", "c", "d", "e"]' | jq '.[2:]'
# ["c", "d", "e"]

echo '["a", "b", "c", "d", "e"]' | jq '.[:3]'
# ["a", "b", "c"]
```

Negative Slice Indices

```
echo '["a", "b", "c", "d", "e"]' | jq '.[-2:]'
# ["d", "e"]

echo '["a", "b", "c", "d", "e"]' | jq '.[:-2]'
# ["a", "b", "c"]
```

Array/Object Iteration: .[]

Iterate over all elements of an array or all values of an object. Produces multiple outputs.

```
echo '["a", "b", "c"]' | jq '.[]'
```

```
# "a"
# "b"
# "c"

echo '{"x": 1, "y": 2}' | jq '.[]'
# 1
# 2
```

Combined with Field Access

```
echo '[{"name": "Alice"}, {"name": "Bob"}]' | jq '.[].name'
# "Alice"
# "Bob"
```

Optional Iteration: .[]?

Suppress errors when input is not iterable:

```
echo '123' | jq '.[]?'
# (no output, no error)
```

Pipes: |

Chain filters together. Output of the left filter becomes input to the right filter.

```
echo '{"user": {"name": "Alice"}}' | jq '.user | .name'
# "Alice"
```

Pipes can be chained indefinitely:

```
echo '{"data": {"users": [{"name": "Alice"}]}}' | jq '.data | .users | .[0] | .name'
# "Alice"
```

Pipe vs Dot Chaining

These are equivalent:

```
jq '.user.name'
jq '.user | .name'
```

Pipes become necessary when using filters that aren't simple field access:

```
echo '[3, 1, 2]' | jq 'sort | .[0]'
# 1
```

Parentheses: ()

Group expressions to control evaluation order.

```
echo '{"a": {"b": 1}}' | jq '(.a.b) + 1'
# 2
```

Parentheses are essential when constructing complex expressions:

```
echo '5' | jq '(. * 2) + 3'
# 13
```

Comma: ,

Run multiple filters on the same input, producing multiple outputs.

```
echo '{"a": 1, "b": 2}' | jq '.a, .b'
# 1
# 2
```

Useful for extracting multiple fields:

```
echo '{"name": "Alice", "age": 30, "city": "London"}' | jq '.name, .age'
# "Alice"
# 30
```

Constructing Arrays: []

Wrap filter output in an array:

```
echo '{"a": 1, "b": 2}' | jq '[.a, .b]'
# [1, 2]
```

Collect iterated values into an array:

```
echo '[{"n": 1}, {"n": 2}, {"n": 3}]' | jq '[.[].n]'
# [1, 2, 3]
```

Constructing Objects: {}

Build new objects with explicit keys:

```
echo '{"name": "Alice", "years": 30}' | jq '{username: .name, age: .years}'
# {"username": "Alice", "age": 30}
```

Shorthand Syntax

When key and filter have the same name, use shorthand:

```
echo '{"name": "Alice", "age": 30}' | jq '{name, age}'
# {"name": "Alice", "age": 30}
```

Dynamic Keys

Use parentheses for computed key names:

```
echo '{"key": "foo", "value": 42}' | jq '{(.key): .value}'
# {"foo": 42}
```

Combining Multiple Inputs

When iterating, object construction applies to each element:

```
echo '[{"first": "Alice", "last": "Smith"}, {"first": "Bob", "last": "Jones"}]' | jq '
# {"name": "Alice"}
# {"name": "Bob"}
```

Wrap in array to collect:

```
echo '[{"first": "Alice", "last": "Smith"}, {"first": "Bob", "last": "Jones"}]' | jq '
# [{"name": "Alice"}, {"name": "Bob"}]
```

Recursive Descent: ..

Recursively descend into all values, producing every value in the structure.

```
echo '{"a": {"b": {"c": 1}}}' | jq '.. | numbers'
# 1
```

Find all values at any depth:

```
echo '{"a": {"id": 1}, "b": {"id": 2, "c": {"id": 3}}}' | jq '.. | .id? // empty'
# 1
# 2
# 3
```

Chapter 4: Types and Values

JSON Types

jq works with the standard JSON types:

Type	Examples
null	`null`
boolean	`true`, `false`
number	`42`, `3.14`, `-17`, `2.5e10`
string	`"hello"`, `"line1\nline2"`
array	`[]`, `[1, 2, 3]`, `["a", {"b": 1}]`
object	`{}`, `{"key": "value"}`

Literal Values

Use literal values directly in filters:

```
jq -n 'null'
# null

jq -n 'true'
# true

jq -n '42'
# 42

jq -n '"hello"'
# "hello"

jq -n '[1, 2, 3]'
# [1, 2, 3]

jq -n '{"name": "Alice"}'
# {"name": "Alice"}
```

Type Checking: type

Returns the type of a value as a string:

```
echo 'null' | jq 'type'
# "null"

echo 'true' | jq 'type'
# "boolean"

echo '42' | jq 'type'
# "number"

echo '"hello"' | jq 'type'
# "string"

echo '[1,2,3]' | jq 'type'
# "array"

echo '{"a":1}' | jq 'type'
# "object"
```

Type Filters

These filters select values of a specific type, passing them through or producing no output:

strings

```
echo '["a", 1, "b", true]' | jq '.[] | strings'
# "a"
# "b"
```

numbers

```
echo '["a", 1, "b", 2.5]' | jq '.[] | numbers'
# 1
# 2.5
```

booleans

```
echo '[true, 1, false, "yes"]' | jq '.[] | booleans'
# true
# false
```

nulls

```
echo '[null, 1, null, "a"]' | jq '.[] | nulls'
# null
# null
```

arrays

```
echo '[[1,2], "a", [3], {}]' | jq '.[] | arrays'
# [1, 2]
# [3]
```

objects

```
echo '[{"a":1}, [1,2], {"b":2}]' | jq '.[] | objects'
# {"a": 1}
# {"b": 2}
```

iterables

Selects arrays and objects:

```
echo '[{"a":1}, [1,2], "text", 42]' | jq '.[] | iterables'
# {"a": 1}
# [1, 2]
```

values

Selects everything except null:

```
echo '[1, null, "a", null]' | jq '.[] | values'
# 1
```

```
# "a"
```

scalars

Selects non-iterables (strings, numbers, booleans, null):

```
echo '[1, [2], "a", {"b":3}]' | jq '.[] | scalars'
# 1
# "a"
```

Type Predicates

These return true or false based on type:

isnormal

Returns true for normal (finite, non-NaN) numbers:

```
echo '42' | jq 'isnormal'
# true
```

isinfinite

```
echo '1e1000' | jq 'isinfinite'
# true
```

isnan

```
jq -n 'nan | isnan'
# true
```

isfinite

```
echo '42' | jq 'isfinite'
```

```
# true
```

Type Conversion

tostring

Convert to string:

```
echo '42' | jq 'tostring'
# "42"

echo 'true' | jq 'tostring'
# "true"

echo '[1,2]' | jq 'tostring'
# "[1,2]"
```

tonumber

Convert string to number:

```
echo '"42"' | jq 'tonumber'
# 42

echo '"3.14"' | jq 'tonumber'
# 3.14
```

String Interpolation

Embed values in strings with `\()`:

```
echo '{"name": "Alice", "age": 30}' | jq '"Name: \(.name), Age: \(.age)"'
# "Name: Alice, Age: 30"
```

Works with any expression:

```
echo '5' | jq '"Square: \(. * .)"'
# "Square: 25"
```

Null Handling

null

The literal null value:

```
jq -n 'null'
# null
```

empty

Produces no output at all (not even null):

```
jq -n 'empty'
# (no output)

echo '[1,2,3]' | jq '.[] | if . == 2 then empty else . end'
# 1
# 3
```

Alternative Operator: //

Return right side if left side is false or null:

```
echo '{"a": null}' | jq '.a // "default"'
# "default"

echo '{"a": 1}' | jq '.a // "default"'
# 1

echo '{}' | jq '.missing // "default"'
# "default"
```

Chain for multiple fallbacks:

```
echo '{}' | jq '.a // .b // .c // "none"'
# "none"
```

not

Logical negation:

```
echo 'true' | jq 'not'
# false

echo 'null' | jq 'not'
# true
```

Special Number Values

nan

Not a number:

```
jq -n 'nan'
# null
```

infinite

Positive infinity:

```
jq -n 'infinite'
# 1.7976931348623157e+308
```

Length: length

Returns the length of values:

Type	Returns
string	number of Unicode codepoints
array	number of elements
object	number of key-value pairs

null	zero
number	absolute value

```
echo '"hello"' | jq 'length'
# 5

echo '[1, 2, 3]' | jq 'length'
# 3

echo '{"a": 1, "b": 2}' | jq 'length'
# 2

echo 'null' | jq 'length'
# 0

echo '-5' | jq 'length'
# 5
```

utf8bytelength

For strings, returns byte length instead of character count:

```
echo '"□□□"' | jq 'utf8bytelength'
# 9

echo '"□□□"' | jq 'length'
# 3
```

Keys and Values

keys

Get object keys as a sorted array:

```
echo '{"b": 2, "a": 1, "c": 3}' | jq 'keys'
# ["a", "b", "c"]
```

For arrays, returns indices:

```
echo '["x", "y", "z"]' | jq 'keys'
# [0, 1, 2]
```

keys_unsorted

Get object keys in original order:

```
echo '{"b": 2, "a": 1, "c": 3}' | jq 'keys_unsorted'
# ["b", "a", "c"]
```

values

Get all values (equivalent to `.[]` wrapped in array):

```
echo '{"a": 1, "b": 2}' | jq '[.[] | values]'
# [1, 2]
```

has(key)

Check if object has a key or array has an index:

```
echo '{"a": 1}' | jq 'has("a")'
# true

echo '{"a": 1}' | jq 'has("b")'
# false

echo '[1, 2, 3]' | jq 'has(2)'
# true

echo '[1, 2, 3]' | jq 'has(5)'
# false
```

in(object)

Check if key exists in object (reversed operands from `has`):

```
echo '"a"' | jq 'in({"a": 1})'
# true
```

contains and inside

Test if a value contains another:

```
echo '{"a": 1, "b": 2}' | jq 'contains({"a": 1})'
# true

echo '[1, 2, 3]' | jq 'contains([2, 3])'
# true

echo '{"a": 1}' | jq 'inside({"a": 1, "b": 2})'
# true
```

Chapter 5: Operators

Arithmetic Operators

Addition: +

```
echo '5' | jq '. + 3'
# 8
```

Addition also concatenates strings:

```
echo '["hello", "world"]' | jq '.[0] + " " + .[1]'
# "hello world"
```

And merges arrays:

```
echo '[[1,2], [3,4]]' | jq '.[0] + .[1]'
# [1, 2, 3, 4]
```

And merges objects (right side wins on conflicts):

```
echo '[{"a":1}, {"b":2, "a":99}]' | jq '.[0] + .[1]'
# {"a": 99, "b": 2}
```

Subtraction: -

```
echo '10' | jq '. - 3'
# 7
```

For arrays, removes elements:

```
echo '[[1,2,3,4], [2,4]]' | jq '.[0] - .[1]'
# [1, 3]
```

Multiplication: *

```
echo '6' | jq '. * 7'
# 42
```

String repetition:

```
echo '"ab"' | jq '. * 3'
# "ababab"
```

Deep object merge:

```
echo '[{"a": {"x":1}}, {"a": {"y":2}}]' | jq '.[0] * .[1]'
# {"a": {"x": 1, "y": 2}}
```

Division: /

```
echo '20' | jq '. / 4'
# 5
```

String split:

```
echo '"a,b,c"' | jq '. / ","'
# ["a", "b", "c"]
```

Modulo: %

```
echo '17' | jq '. % 5'
# 2
```

Negation: -

```
echo '5' | jq '-.'
# -5
```

Comparison Operators

All comparisons return `true` or `false`.

Equality: ==

```
echo '5' | jq '. == 5'
# true

echo '{"a":1}' | jq '. == {"a":1}'
# true
```

Inequality: !=

```
echo '5' | jq '. != 3'
# true
```

Less Than: <

```
echo '5' | jq '. < 10'
# true
```

Less Than or Equal: <=

```
echo '5' | jq '. <= 5'
# true
```

Greater Than: >

```
echo '5' | jq '. > 3'
# true
```

Greater Than or Equal: >=

```
echo '5' | jq '. >= 5'
# true
```

Comparison Order

Values of different types are ordered:

```
null < false < true < numbers < strings < arrays < objects
```

```
echo '[true, null, 1, "a", [], {}]' | jq 'sort'
# [null, true, 1, "a", [], {}]
```

Logical Operators

and

```
echo '{"a": true, "b": false}' | jq '.a and .b'
# false

echo '{"a": true, "b": true}' | jq '.a and .b'
# true
```

or

```
echo '{"a": true, "b": false}' | jq '.a or .b'
# true

echo '{"a": false, "b": false}' | jq '.a or .b'
# false
```

not

```
echo 'true' | jq 'not'
# false

echo 'false' | jq 'not'
# true
```

```
echo 'null' | jq 'not'
# true
```

Truthiness

In jq, `false` and `null` are falsy. Everything else is truthy, including:

- `0`
- `""`
- `[]`
- `{}`

```
echo '0' | jq 'if . then "truthy" else "falsy" end'
# "truthy"

echo '""' | jq 'if . then "truthy" else "falsy" end'
# "truthy"

echo 'null' | jq 'if . then "truthy" else "falsy" end'
# "falsy"
```

Alternative Operator: //

Returns right side if left side is `false` or `null`:

```
echo 'null' | jq '. // "default"'
# "default"

echo 'false' | jq '. // "default"'
# "default"

echo '0' | jq '. // "default"'
# 0

echo '""' | jq '. // "default"'
# ""
```

Chaining:

```
echo '{}' | jq '.a // .b // .c // "none"'
# "none"
```

Try-Catch: try-catch / ?

try-catch

Catch errors and provide fallback:

```
echo '"not a number"' | jq 'try tonumber catch "invalid"'
# "invalid"

echo '"42"' | jq 'try tonumber catch "invalid"'
# 42
```

Error Suppression: ?

The `?` suffix suppresses errors, producing no output instead:

```
echo '"text"' | jq '.foo?'
# null

echo '"text"' | jq '.foo.bar?'
# (no output)
```

Equivalent to `try EXPR catch empty`:

```
echo '"text"' | jq 'try .foo.bar catch empty'
# (no output)
```

Update Operators

Update: |=

Apply a filter to update a value:

```
echo '{"a": 5}' | jq '.a |= . + 1'
# {"a": 6}
```

Update nested values:

```
echo '{"user": {"score": 10}}' | jq '.user.score |= . * 2'
# {"user": {"score": 20}}
```

Update all array elements:

```
echo '[1, 2, 3]' | jq '.[] |= . * 10'
# [10, 20, 30]
```

Arithmetic Update: +=, -=, *=, /=, %=

```
echo '{"count": 5}' | jq '.count += 1'
# {"count": 6}

echo '{"count": 5}' | jq '.count -= 2'
# {"count": 3}

echo '{"count": 5}' | jq '.count *= 3'
# {"count": 15}

echo '{"count": 20}' | jq '.count /= 4'
# {"count": 5}

echo '{"count": 17}' | jq '.count %= 5'
# {"count": 2}
```

Alternative Update: //=

Set value only if currently null or false:

```
echo '{"a": null}' | jq '.a //= "default"'
# {"a": "default"}

echo '{"a": 5}' | jq '.a //= "default"'
# {"a": 5}
```

Assignment: =

Plain assignment (rarely needed, usually |= is better):

```
echo '{"a": 1}' | jq '.b = 2'
# {"a": 1, "b": 2}
```

String Operators

Concatenation: +

```
echo '["hello", "world"]' | jq '.[0] + " " + .[1]'
# "hello world"
```

Repetition: *

```
echo '"abc"' | jq '. * 3'
# "abcabcabc"
```

Split: /

```
echo '"a:b:c"' | jq '. / ":"'
# ["a", "b", "c"]
```

Pipe Expressions

Simple Pipe: |

```
echo '{"a": {"b": 1}}' | jq '.a | .b'
# 1
```

Comma for Multiple Outputs

```
echo '{"a": 1, "b": 2}' | jq '.a, .b'
# 1
# 2
```

Operator Precedence

From highest to lowest:

1. `.field`, `.[index]`, `.[]`
2. `-` (unary negation)
3. `*`, `/`, `%`
4. `+`, `-`
5. `==`, `!=`, `<`, `<=`, `>`, `>=`
6. `not`
7. `and`
8. `or`
9. `|=`, `+=`, `-=`, etc.
10. `//`
11. `,`
12. `|`

Use parentheses to override:

```
echo '2' | jq '(. + 3) * 4'
# 20

echo '2' | jq '. + 3 * 4'
# 14
```

Chapter 6: Conditionals and Comparisons

If-Then-Else

Basic conditional:

```
echo '5' | jq 'if . > 3 then "big" else "small" end'
# "big"
```

The `else` clause is required:

```
echo '5' | jq 'if . > 10 then "big" else . end'
# 5
```

Elif

Chain conditions with `elif`:

```
echo '5' | jq 'if . < 0 then "negative" elif . == 0 then "zero" else "positive" end'
# "positive"
```

Multiple elif:

```
echo '75' | jq '
  if . >= 90 then "A"
  elif . >= 80 then "B"
  elif . >= 70 then "C"
  elif . >= 60 then "D"
  else "F"
  end
'
# "C"
```

Nested Conditionals

```
echo '{"type": "user", "active": true}' | jq '
  if .type == "user" then
    if .active then "active user" else "inactive user" end
  else
    "not a user"
  end
'
# "active user"
```

Conditionals with Multiple Outputs

If the condition produces multiple outputs, the conditional is evaluated for each:

```
echo '[1, 5, 3, 8, 2]' | jq '.[] | if . > 4 then "big" else "small" end'
# "small"
# "big"
# "small"
# "big"
# "small"
```

Select

Filter values based on a condition. Values that pass the condition are output; others produce nothing.

```
echo '[1, 2, 3, 4, 5]' | jq '.[] | select(. > 3)'
# 4
# 5
```

Select with Objects

```
echo '[{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]' | jq '.[] | select(.
# {"name": "Alice", "age": 30}
```

Select with Multiple Conditions

```
echo '[{"name": "Alice", "age": 30, "active": true}, {"name": "Bob", "age": 25, "activ
# {"name": "Alice", "age": 30, "active": true}
```

Select vs If-Then-Else

`select` filters out non-matching values:

```
echo '[1, 2, 3]' | jq '.[] | select(. > 1)'
# 2
# 3
```

`if-then-else` transforms all values:

```
echo '[1, 2, 3]' | jq '.[] | if . > 1 then "big" else "small" end'
# "small"
# "big"
# "big"
```

Keeping Array Structure

Wrap in `map` to maintain array structure:

```
echo '[1, 2, 3, 4, 5]' | jq 'map(select(. > 3))'
# [4, 5]
```

Empty

Produces no output. Useful for filtering or conditional suppression.

```
jq -n 'empty'
# (no output)
```

Using Empty to Filter

```
echo '[1, 2, 3, 4, 5]' | jq '.[] | if . > 3 then . else empty end'
# 4
# 5
```

This is equivalent to `select(. > 3)`.

Empty in Arrays

```
echo 'null' | jq '[1, 2, empty, 3]'
# [1, 2, 3]
```

Error Handling

error

Raise an error with a message:

```
echo '{"value": -5}' | jq 'if .value < 0 then error("negative value") else .value end'
# error: negative value
```

error with no message

```
echo 'null' | jq 'error'
# error: null
```

try-catch

Handle errors gracefully:

```
echo '["1", "two", "3"]' | jq '.[] | try tonumber catch "not a number"'
# 1
# "not a number"
# 3
```

Catch with error message:

```
echo '"text"' | jq 'try (.foo.bar) catch "error: \(.)"'
# "error: Cannot index string with string \"foo\""
```

try without catch

Equivalent to `try EXPR catch empty`:

```
echo '["1", "two", "3"]' | jq '[.[] | try tonumber]'
# [1, 3]
```

Comparison Functions

min and max

```
echo '[3, 1, 4, 1, 5, 9]' | jq 'min'
# 1

echo '[3, 1, 4, 1, 5, 9]' | jq 'max'
# 9
```

min_by and max_by

Find minimum/maximum by a derived value:

```
echo '[{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]' | jq 'min_by(.age)'
# {"name": "Bob", "age": 25}

echo '[{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]' | jq 'max_by(.age)'
# {"name": "Alice", "age": 30}
```

Equality Testing

Plain Equality: ==

Deep equality comparison:

```
echo '[{"a": [1, 2]}, {"a": [1, 2]}]' | jq '.[0] == .[1]'
# true
```

Test for Null

```
echo '{"a": null, "b": 1}' | jq '.a == null'
# true

echo '{"a": null, "b": 1}' | jq '.c == null'
# true
```

Use `has` to distinguish missing from null:

```
echo '{"a": null}' | jq 'has("a"), has("b")'
# true
# false
```

Containment

contains

Test if left contains right:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'contains({"a": 1})'
# true

echo '{"a": 1, "b": 2}' | jq 'contains({"a": 1, "d": 4})'
# false

echo '[1, 2, 3, 4]' | jq 'contains([2, 4])'
# true

echo '"foobar"' | jq 'contains("bar")'
# true
```

inside

Reversed containment (is left inside right?):

```
echo '{"a": 1}' | jq 'inside({"a": 1, "b": 2})'
# true

echo '[1, 2]' | jq 'inside([1, 2, 3])'
# true
```

Any and All

any

True if any element is truthy:

```
echo '[false, false, true]' | jq 'any'
# true

echo '[false, false, false]' | jq 'any'
# false
```

With a condition:

```
echo '[1, 2, 3, 4, 5]' | jq 'any(. > 4)'
# true
```

With generator and condition:

```
echo '[[1,2], [3,4], [5,6]]' | jq 'any(.[]; . > 5)'
# true
```

all

True if all elements are truthy:

```
echo '[true, true, true]' | jq 'all'
# true

echo '[true, false, true]' | jq 'all'
# false
```

With a condition:

```
echo '[2, 4, 6, 8]' | jq 'all(. > 0)'
# true

echo '[2, 4, 6, 8]' | jq 'all(. < 5)'
```

```
# false
```

Limit

Limit the number of outputs:

```
echo 'null' | jq '[limit(3; range(100))]'
# [0, 1, 2]
```

first and last

```
echo '[5, 2, 8, 1, 9]' | jq 'first'
# 5

echo '[5, 2, 8, 1, 9]' | jq 'last'
# 9
```

First/last with generator:

```
echo 'null' | jq 'first(range(10))'
# 0

echo 'null' | jq 'last(range(10))'
# 9
```

nth

Get the nth output (zero-indexed):

```
echo 'null' | jq 'nth(5; range(10))'
# 5
```

Range

Generate a sequence of numbers:

```
echo 'null' | jq '[range(5)]'
# [0, 1, 2, 3, 4]

echo 'null' | jq '[range(2; 5)]'
# [2, 3, 4]

echo 'null' | jq '[range(0; 10; 2)]'
# [0, 2, 4, 6, 8]
```

until and while

Iterate until/while condition:

```
echo '1' | jq 'until(. > 100; . * 2)'
# 128

echo '1' | jq '[while(. < 100; . * 2)]'
# [1, 2, 4, 8, 16, 32, 64]
```

repeat

Repeat a value indefinitely (use with `limit`):

```
echo '"x"' | jq '[limit(5; repeat(.))]'
# ["x", "x", "x", "x", "x"]
```

Chapter 7: Array Operations

Creating Arrays

Literal Arrays

```
jq -n '[1, 2, 3]'
# [1, 2, 3]
```

Collecting Results: []

Wrap any expression in `[]` to collect outputs into an array:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq '[.a, .b, .c]'
# [1, 2, 3]

echo '[1, 2, 3]' | jq '[.[] | . * 2]'
# [2, 4, 6]
```

range

Generate sequences:

```
jq -n '[range(5)]'
# [0, 1, 2, 3, 4]

jq -n '[range(1; 6)]'
# [1, 2, 3, 4, 5]

jq -n '[range(0; 10; 2)]'
# [0, 2, 4, 6, 8]
```

Accessing Elements

By Index

```
echo '["a", "b", "c", "d"]' | jq '.[0]'
# "a"

echo '["a", "b", "c", "d"]' | jq '.[2]'
# "c"

echo '["a", "b", "c", "d"]' | jq '.[-1]'
# "d"
```

first and last

```
echo '["a", "b", "c"]' | jq 'first'
# "a"

echo '["a", "b", "c"]' | jq 'last'
# "c"
```

nth

```
echo '["a", "b", "c", "d"]' | jq 'nth(2)'
# "c"
```

Slicing

```
echo '["a", "b", "c", "d", "e"]' | jq '.[1:4]'
# ["b", "c", "d"]

echo '["a", "b", "c", "d", "e"]' | jq '.[2:]'
# ["c", "d", "e"]

echo '["a", "b", "c", "d", "e"]' | jq '.[:3]'
# ["a", "b", "c"]

echo '["a", "b", "c", "d", "e"]' | jq '.[-2:]'
# ["d", "e"]
```

Iteration

.[]

Iterate over all elements:

```
echo '[1, 2, 3]' | jq '.[]'
# 1
# 2
# 3
```

Iterate with Index

Use `to_entries` or `range`:

```
echo '["a", "b", "c"]' | jq 'to_entries[] | "\(.key): \(.value)"'
# "0: a"
# "1: b"
# "2: c"
```

Or with `range` and `length`:

```
echo '["a", "b", "c"]' | jq 'range(length) as $i | "\($i): \(.[$i])"'
# "0: a"
# "1: b"
# "2: c"
```

Length and Counting

length

```
echo '[1, 2, 3, 4, 5]' | jq 'length'
# 5

echo '[]' | jq 'length'
# 0
```

Counting with Conditions

```
echo '[1, 2, 3, 4, 5]' | jq '[.[] | select(. > 2)] | length'
```

```
# 3

echo '[1, 2, 3, 4, 5]' | jq 'map(select(. > 2)) | length'
# 3
```

Map

Apply a filter to each element:

```
echo '[1, 2, 3]' | jq 'map(. * 2)'
# [2, 4, 6]

echo '[{"name": "Alice"}, {"name": "Bob"}]' | jq 'map(.name)'
# ["Alice", "Bob"]
```

map_values

For objects, apply filter to values while keeping keys:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'map_values(. * 10)'
# {"a": 10, "b": 20, "c": 30}
```

Also works on arrays:

```
echo '[1, 2, 3]' | jq 'map_values(. + 100)'
# [101, 102, 103]
```

Select and Filter

select

Keep elements matching a condition:

```
echo '[1, 2, 3, 4, 5]' | jq 'map(select(. > 2))'
# [3, 4, 5]

echo '[{"age": 25}, {"age": 35}, {"age": 20}]' | jq 'map(select(.age >= 25))'
# [{"age": 25}, {"age": 35}]
```

arrays

Keep only array elements:

```
echo '[[1], "a", [2,3], {"x":1}]' | jq '.[] | arrays'
# [1]
# [2, 3]
```

Sorting

sort

Sort in ascending order:

```
echo '[3, 1, 4, 1, 5, 9, 2, 6]' | jq 'sort'
# [1, 1, 2, 3, 4, 5, 6, 9]

echo '["banana", "apple", "cherry"]' | jq 'sort'
# ["apple", "banana", "cherry"]
```

sort_by

Sort by a derived value:

```
echo '[{"name": "Bob", "age": 30}, {"name": "Alice", "age": 25}]' | jq 'sort_by(.name)
# [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}]

echo '[{"name": "Bob", "age": 30}, {"name": "Alice", "age": 25}]' | jq 'sort_by(.age)'
# [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}]
```

Descending Sort

```
echo '[3, 1, 4, 1, 5]' | jq 'sort | reverse'
# [5, 4, 3, 1, 1]

echo '[{"n": 3}, {"n": 1}, {"n": 2}]' | jq 'sort_by(.n) | reverse'
# [{"n": 3}, {"n": 2}, {"n": 1}]
```

Or negate numeric values:

```
echo '[{"n": 3}, {"n": 1}, {"n": 2}]' | jq 'sort_by(-.n)'
# [{"n": 3}, {"n": 2}, {"n": 1}]
```

reverse

```
echo '[1, 2, 3, 4, 5]' | jq 'reverse'
# [5, 4, 3, 2, 1]
```

Unique

unique

Remove duplicates (sorts the array):

```
echo '[3, 1, 2, 1, 3, 2]' | jq 'unique'
# [1, 2, 3]
```

unique_by

Remove duplicates by a key:

```
echo '[{"id": 1, "v": "a"}, {"id": 2, "v": "b"}, {"id": 1, "v": "c"}]' | jq 'unique_by
# [{"id": 1, "v": "a"}, {"id": 2, "v": "b"}]
```

Grouping

group_by

Group elements by a key:

```
echo '[{"type": "a", "v": 1}, {"type": "b", "v": 2}, {"type": "a", "v": 3}]' | jq 'gro
# [[{"type": "a", "v": 1}, {"type": "a", "v": 3}], [{"type": "b", "v": 2}]]
```

Common pattern - group and summarise:

```
echo '[{"type": "a", "v": 1}, {"type": "b", "v": 2}, {"type": "a", "v": 3}]' | jq '
  group_by(.type) | map({type: .[0].type, count: length, sum: map(.v) | add})
'
# [{"type": "a", "count": 2, "sum": 4}, {"type": "b", "count": 1, "sum": 2}]
```

Flattening

flatten

Flatten nested arrays:

```
echo '[[1, 2], [3, [4, 5]]]' | jq 'flatten'
# [1, 2, 3, 4, 5]
```

Flatten to a specific depth:

```
echo '[[1, 2], [3, [4, 5]]]' | jq 'flatten(1)'
# [1, 2, 3, [4, 5]]
```

add

Concatenate arrays (or sum numbers):

```
echo '[[1, 2], [3, 4], [5]]' | jq 'add'
# [1, 2, 3, 4, 5]

echo '[1, 2, 3, 4, 5]' | jq 'add'
# 15
```

Min and Max

min, max

```
echo '[3, 1, 4, 1, 5, 9]' | jq 'min'
# 1

echo '[3, 1, 4, 1, 5, 9]' | jq 'max'
# 9
```

min_by, max_by

```
echo '[{"name": "Alice", "score": 85}, {"name": "Bob", "score": 92}]' | jq 'min_by(.sc
# {"name": "Alice", "score": 85}

echo '[{"name": "Alice", "score": 85}, {"name": "Bob", "score": 92}]' | jq 'max_by(.sc
# {"name": "Bob", "score": 92}
```

Adding and Removing Elements

+ (concatenation)

```
echo '[[1, 2], [3, 4]]' | jq '.[0] + .[1]'
# [1, 2, 3, 4]
```

- (subtraction)

Remove elements:

```
echo '[[1, 2, 3, 4, 5], [2, 4]]' | jq '.[0] - .[1]'
# [1, 3, 5]
```

+= (append)

```
echo '[1, 2, 3]' | jq '. += [4, 5]'
# [1, 2, 3, 4, 5]
```

Prepend

```
echo '[3, 4, 5]' | jq '[1, 2] + .'
# [1, 2, 3, 4, 5]
```

Insert at Index

```
echo '[1, 2, 5, 6]' | jq '.[:2] + [3, 4] + .[2:]'
# [1, 2, 3, 4, 5, 6]
```

del

Delete by index:

```
echo '[1, 2, 3, 4, 5]' | jq 'del(.[2])'
# [1, 2, 4, 5]
```

Delete multiple:

```
echo '[1, 2, 3, 4, 5]' | jq 'del(.[1], .[3])'
# [1, 3, 5]
```

Membership

index and rindex

Find position of element:

```
echo '["a", "b", "c", "b", "d"]' | jq 'index("b")'
# 1

echo '["a", "b", "c", "b", "d"]' | jq 'rindex("b")'
# 3
```

Returns null if not found:

```
echo '["a", "b", "c"]' | jq 'index("x")'
# null
```

indices

Find all positions:

```
echo '["a", "b", "c", "b", "d", "b"]' | jq 'indices("b")'
# [1, 3, 5]
```

contains

```
echo '[1, 2, 3, 4, 5]' | jq 'contains([2, 4])'
# true

echo '[1, 2, 3]' | jq 'contains([4])'
# false
```

inside

```
echo '[2, 4]' | jq 'inside([1, 2, 3, 4, 5])'
# true
```

any and all

```
echo '[1, 2, 3, 4, 5]' | jq 'any(. > 4)'
# true

echo '[1, 2, 3, 4, 5]' | jq 'all(. > 0)'
# true

echo '[1, 2, 3, 4, 5]' | jq 'all(. > 2)'
# false
```

Combining Arrays

transpose

Transpose array of arrays:

```
echo '[[1, 2, 3], ["a", "b", "c"]]' | jq 'transpose'
# [[1, "a"], [2, "b"], [3, "c"]]
```

zip (using transpose)

```
echo '[["a", "b", "c"], [1, 2, 3]]' | jq 'transpose | map({key: .[0], value: .[1]})'
# [{"key": "a", "value": 1}, {"key": "b", "value": 2}, {"key": "c", "value": 3}]
```

Aggregation

add

Sum numbers or concatenate strings/arrays:

```
echo '[1, 2, 3, 4, 5]' | jq 'add'
# 15

echo '["a", "b", "c"]' | jq 'add'
# "abc"
```

Handling Empty Arrays

```
echo '[]' | jq 'add'
# null

echo '[]' | jq 'add // 0'
# 0
```

Average

```
echo '[1, 2, 3, 4, 5]' | jq 'add / length'
# 3
```

Product

```
echo '[1, 2, 3, 4, 5]' | jq 'reduce .[] as $x (1; . * $x)'
# 120
```

Chapter 8: Object Operations

Creating Objects

Literal Objects

```
jq -n '{"name": "Alice", "age": 30}'
# {"name": "Alice", "age": 30}
```

From Existing Values

```
echo '{"first": "Alice", "last": "Smith", "years": 30}' | jq '{name: .first, age: .yea
# {"name": "Alice", "age": 30}
```

Shorthand Syntax

When key name matches the filter:

```
echo '{"name": "Alice", "age": 30, "city": "London"}' | jq '{name, age}'
# {"name": "Alice", "age": 30}
```

Dynamic Keys

Use parentheses for computed key names:

```
echo '{"key": "username", "value": "alice"}' | jq '{(.key): .value}'
# {"username": "alice"}
```

```
echo '[{"k": "a", "v": 1}, {"k": "b", "v": 2}]' | jq 'map({(.k): .v}) | add'
# {"a": 1, "b": 2}
```

Accessing Values

Field Access

```
echo '{"name": "Alice", "age": 30}' | jq '.name'
# "Alice"
```

Nested Access

```
echo '{"user": {"profile": {"name": "Alice"}}}' | jq '.user.profile.name'
# "Alice"
```

Optional Access

```
echo '{"a": 1}' | jq '.b?'
# null

echo '{"a": 1}' | jq '.b.c?'
# (no output)
```

Bracket Notation

Required for special characters or dynamic access:

```
echo '{"my-key": 1, "other.key": 2}' | jq '.["my-key"]'
# 1

echo '{"a": 1, "b": 2}' | jq '"a" as $k | .[$k]'
# 1
```

Keys and Values

keys

Get sorted keys:

```
echo '{"c": 3, "a": 1, "b": 2}' | jq 'keys'
```

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

keys_unsorted

Get keys in original order:

```
echo '{"c": 3, "a": 1, "b": 2}' | jq 'keys_unsorted'
# ["c", "a", "b"]
```

values

Get all values:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq '[.values]'
# error - use .[] instead

echo '{"a": 1, "b": 2, "c": 3}' | jq '[.[]]'
# [1, 2, 3]
```

Iterate Over Keys and Values

```
echo '{"a": 1, "b": 2}' | jq 'keys[] as $k | "\($k): \(.[$k])"'
# "a: 1"
# "b: 2"
```

has and in

has(key)

Check if object has a key:

```
echo '{"a": 1, "b": null}' | jq 'has("a")'
# true

echo '{"a": 1, "b": null}' | jq 'has("b")'
# true
```

```
echo '{"a": 1, "b": null}' | jq 'has("c")'
# false
```

in(object)

Check if key exists (reversed operands):

```
echo '"a"' | jq 'in({"a": 1, "b": 2})'
# true

echo '"c"' | jq 'in({"a": 1, "b": 2})'
# false
```

Entry Conversion

to_entries

Convert object to array of key-value pairs:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'to_entries'
# [{"key": "a", "value": 1}, {"key": "b", "value": 2}, {"key": "c", "value": 3}]
```

from_entries

Convert array of key-value pairs back to object:

```
echo '[{"key": "a", "value": 1}, {"key": "b", "value": 2}]' | jq 'from_entries'
# {"a": 1, "b": 2}
```

Also accepts `name`/`value` and `k`/`v`:

```
echo '[{"name": "a", "value": 1}, {"k": "b", "v": 2}]' | jq 'from_entries'
# {"a": 1, "b": 2}
```

with_entries

Transform entries (shorthand for `to_entries | map(...) | from_entries`):

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'with_entries(.value *= 10)'
# {"a": 10, "b": 20, "c": 30}
```

Transform keys:

```
echo '{"name": "Alice", "age": 30}' | jq 'with_entries(.key = "user_" + .key)'
# {"user_name": "Alice", "user_age": 30}
```

Filter entries:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'with_entries(select(.value > 1))'
# {"b": 2, "c": 3}
```

Adding and Updating Fields

+ (merge)

Merge objects (right side wins on conflict):

```
echo '[{"a": 1, "b": 2}, {"b": 99, "c": 3}]' | jq '.[0] + .[1]'
# {"a": 1, "b": 99, "c": 3}
```

+= (merge update)

```
echo '{"a": 1}' | jq '. += {"b": 2, "c": 3}'
# {"a": 1, "b": 2, "c": 3}
```

* (recursive merge)

Deep merge objects:

```
echo '[{"a": {"x": 1}}, {"a": {"y": 2}}]' | jq '.[0] * .[1]'
```

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

Compare with shallow merge:

```
echo '[{"a": {"x": 1}}, {"a": {"y": 2}}]' | jq '.[0] + .[1]'
# {"a": {"y": 2}}
```

Simple Assignment

```
echo '{"a": 1}' | jq '.b = 2'
# {"a": 1, "b": 2}

echo '{"a": 1}' | jq '.a = 99'
# {"a": 99}
```

Update Operator: |=

Apply filter to update value:

```
echo '{"count": 5}' | jq '.count |= . + 1'
# {"count": 6}
```

Arithmetic Updates

```
echo '{"n": 10}' | jq '.n += 5'
# {"n": 15}

echo '{"n": 10}' | jq '.n -= 3'
# {"n": 7}

echo '{"n": 10}' | jq '.n *= 2'
# {"n": 20}
```

Alternative Update: //=

Set only if null or missing:

```
echo '{"a": null}' | jq '.a //= "default"'
# {"a": "default"}
```

```
echo '{"a": 1}' | jq '.a //= "default"'
# {"a": 1}

echo '{}' | jq '.a //= "default"'
# {"a": "default"}
```

Removing Fields

del

Delete a field:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'del(.b)'
# {"a": 1, "c": 3}
```

Delete multiple fields:

```
echo '{"a": 1, "b": 2, "c": 3, "d": 4}' | jq 'del(.b, .d)'
# {"a": 1, "c": 3}
```

Delete nested field:

```
echo '{"user": {"name": "Alice", "password": "secret"}}' | jq 'del(.user.password)'
# {"user": {"name": "Alice"}}
```

Keeping Specific Fields

Use object construction:

```
echo '{"a": 1, "b": 2, "c": 3, "d": 4}' | jq '{a, c}'
# {"a": 1, "c": 3}
```

Or filter with `with_entries`:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'with_entries(select(.key == "a" or .key == "c"))
# {"a": 1, "c": 3}
```

Containment

contains

Check if object contains another:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'contains({"a": 1})'
# true

echo '{"a": 1, "b": 2}' | jq 'contains({"a": 1, "d": 4})'
# false
```

Deep containment:

```
echo '{"a": {"b": {"c": 1}}}' | jq 'contains({"a": {"b": {}}})'
# true
```

inside

Reversed containment:

```
echo '{"a": 1}' | jq 'inside({"a": 1, "b": 2})'
# true
```

Iterating Over Objects

.[]

Iterate over values:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq '.[]'
# 1
# 2
# 3
```

keys[]

Iterate over keys:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'keys[]'
# "a"
# "b"
# "c"
```

to_entries[]

Iterate with both key and value:

```
echo '{"a": 1, "b": 2}' | jq 'to_entries[] | "\(.key)=\(.value)"'
# "a=1"
# "b=2"
```

Transforming Objects

Rename Keys

```
echo '{"old_name": "Alice"}' | jq '{new_name: .old_name}'
# {"new_name": "Alice"}
```

Rename with pattern:

```
echo '{"user_name": "Alice", "user_age": 30}' | jq 'with_entries(.key |= ltrimstr("use
# {"name": "Alice", "age": 30}
```

Filter by Key Pattern

```
echo '{"name": "Alice", "age": 30, "name_full": "Alice Smith"}' | jq 'with_entries(sel
# {"name": "Alice", "name_full": "Alice Smith"}
```

Filter by Value Type

```
echo '{"a": 1, "b": "text", "c": 2, "d": [1,2]}' | jq 'with_entries(select(.value | ty
# {"a": 1, "c": 2}
```

Object Paths

paths

Get all paths in object:

```
echo '{"a": {"b": 1}, "c": 2}' | jq '[paths]'
# [["a"], ["a", "b"], ["c"]]
```

Paths to scalar values only:

```
echo '{"a": {"b": 1}, "c": 2}' | jq '[paths(scalars)]'
# [["a", "b"], ["c"]]
```

getpath

Get value at path:

```
echo '{"a": {"b": {"c": 1}}}' | jq 'getpath(["a", "b", "c"])'
# 1
```

setpath

Set value at path:

```
echo '{"a": 1}' | jq 'setpath(["b", "c"]; 2)'
# {"a": 1, "b": {"c": 2}}
```

delpaths

Delete multiple paths:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'delpaths([["a"], ["c"]])'
# {"b": 2}
```

Object Length

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'length'
# 3

echo '{}' | jq 'length'
# 0
```

Converting Objects

To Array of Values

```
echo '{"a": 1, "b": 2}' | jq '[.[]]'
# [1, 2]
```

To Array of Keys

```
echo '{"a": 1, "b": 2}' | jq 'keys'
# ["a", "b"]
```

To Array of Pairs

```
echo '{"a": 1, "b": 2}' | jq 'to_entries | map([.key, .value])'
# [["a", 1], ["b", 2]]
```

From Array of Pairs

```
echo '[["a", 1], ["b", 2]]' | jq 'map({key: .[0], value: .[1]}) | from_entries'
# {"a": 1, "b": 2}
```

Or using `reduce`:

```
echo '[["a", 1], ["b", 2]]' | jq 'reduce .[] as $p ({}; .[$p[0]] = $p[1])'
# {"a": 1, "b": 2}
```

Chapter 9: String Functions

String Basics

Length

```
echo '"hello"' | jq 'length'
# 5

echo '"□□□"' | jq 'length'
# 3
```

Byte Length

```
echo '"hello"' | jq 'utf8bytelength'
# 5

echo '"□□□"' | jq 'utf8bytelength'
# 9
```

Concatenation

```
echo '["hello", "world"]' | jq '.[0] + " " + .[1]'
# "hello world"
```

Repetition

```
echo '"ab"' | jq '. * 3'
# "ababab"
```

String Interpolation

Embed expressions in strings with `\()`:

```
echo '{"name": "Alice", "age": 30}' | jq '"Name: \(.name), Age: \(.age)"'
# "Name: Alice, Age: 30"

echo '5' | jq '"The square of \(.) is \(. * .)"'
# "The square of 5 is 25"
```

Case Conversion

ascii_downcase

```
echo '"Hello World"' | jq 'ascii_downcase'
# "hello world"
```

ascii_upcase

```
echo '"Hello World"' | jq 'ascii_upcase'
# "HELLO WORLD"
```

Trimming and Padding

ltrimstr

Remove prefix:

```
echo '"hello world"' | jq 'ltrimstr("hello ")'
# "world"

echo '"hello world"' | jq 'ltrimstr("goodbye ")'
# "hello world"
```

rtrimstr

Remove suffix:

```
echo '"hello.txt"' | jq 'rtrimstr(".txt")'
```

```
# "hello"
```

Trim Whitespace

jq doesn't have a built-in trim, but you can use regex:

```
echo '"  hello  "' | jq 'gsub("^\\s+|\\s+$"; "")'
# "hello"
```

Or split and join:

```
echo '"  hello world  "' | jq 'split(" ") | map(select(. != "")) | join(" ")'
# "hello world"
```

Splitting and Joining

split

Split string into array:

```
echo '"a,b,c,d"' | jq 'split(",")'
# ["a", "b", "c", "d"]

echo '"one  two  three"' | jq 'split("  ")'
# ["one", "two", "three"]
```

Split by regex (using `splits`):

```
echo '"a1b2c3d"' | jq '[splits("[0-9]+")]'
# ["a", "b", "c", "d"]
```

join

Join array into string:

```
echo '["a", "b", "c"]' | jq 'join(",")'
```

```
# "a,b,c"

echo '["hello", "world"]' | jq 'join(" ")'
# "hello world"
```

/ (split shorthand)

```
echo '"a/b/c"' | jq '. / "/"'
# ["a", "b", "c"]
```

Substrings

Slicing

```
echo '"hello world"' | jq '.[0:5]'
# "hello"

echo '"hello world"' | jq '.[6:]'
# "world"

echo '"hello world"' | jq '.[-5:]'
# "world"

echo '"hello world"' | jq '.[:-6]'
# "hello"
```

Index Access

```
echo '"hello"' | jq '.[0:1]'
# "h"

echo '"hello"' | jq '.[-1:]'
# "o"
```

Searching

contains

Check if string contains substring:

```
echo '"hello world"' | jq 'contains("world")'
# true

echo '"hello world"' | jq 'contains("foo")'
# false
```

inside

Check if string is inside another:

```
echo '"world"' | jq 'inside("hello world")'
# true
```

startswith

```
echo '"hello world"' | jq 'startswith("hello")'
# true

echo '"hello world"' | jq 'startswith("world")'
# false
```

endswith

```
echo '"hello.txt"' | jq 'endswith(".txt")'
# true

echo '"hello.txt"' | jq 'endswith(".json")'
# false
```

index and rindex

Find position of substring:

```
echo '"hello world"' | jq 'index("o")'
# 4

echo '"hello world"' | jq 'rindex("o")'
```

```
# 7

echo '"hello world"' | jq 'index("x")'
# null
```

indices

Find all positions:

```
echo '"abcabc"' | jq 'indices("bc")'
# [1, 4]
```

Regular Expressions

test

Test if string matches regex:

```
echo '"hello123"' | jq 'test("[0-9]+")'
# true

echo '"hello"' | jq 'test("[0-9]+")'
# false
```

Case insensitive:

```
echo '"Hello"' | jq 'test("hello"; "i")'
# true
```

match

Return match details:

```
echo '"hello123world"' | jq 'match("[0-9]+")'
# {"offset": 5, "length": 3, "string": "123", "captures": []}
```

With captures:

```
echo '"2024-01-15"' | jq 'match("([0-9]{4})-([0-9]{2})-([0-9]{2})")'
# {"offset": 0, "length": 10, "string": "2024-01-15", "captures": [{"offset": 0, "leng
```

Named captures:

```
echo '"2024-01-15"' | jq 'match("(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})
```

capture

Extract named groups as object:

```
echo '"2024-01-15"' | jq 'capture("(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2
# {"year": "2024", "month": "01", "day": "15"}
```

scan

Find all matches:

```
echo '"a1b2c3"' | jq '[scan("[0-9]+")]'
# ["1", "2", "3"]
```

With captures:

```
echo '"key1=val1 key2=val2"' | jq '[scan("([a-z0-9]+)=([a-z0-9]+)")] | map({key: .[0],
# [{"key": "key1", "value": "val1"}, {"key": "key2", "value": "val2"}]
```

splits

Split by regex (produces multiple outputs):

```
echo '"a1b2c3"' | jq '[splits("[0-9]+")]'
# ["a", "b", "c"]
```

Substitution

sub

Replace first match:

```
echo '"hello world"' | jq 'sub("world"; "there")'
# "hello there"
```

With regex:

```
echo '"abc123def"' | jq 'sub("[0-9]+"; "XXX")'
# "abcXXXdef"
```

gsub

Replace all matches:

```
echo '"a1b2c3"' | jq 'gsub("[0-9]"; "X")'
# "aXbXcX"
```

With capture references:

```
echo '"hello world"' | jq 'gsub("(?<word>\\w+)"; "[\(.word)]")'
# "[hello] [world]"
```

Regex Flags

Pass flags as second argument to `test`, `match`, `capture`, `scan`, `sub`, `gsub`:

Flag	Meaning
`i`	Case insensitive
`x`	Extended (ignore whitespace)
`m`	Multiline (^ and $ match line boundaries)
`s`	Single line (. matches newline)
`g`	Global (for sub, same as gsub)

```
echo '"Hello World"' | jq 'gsub("hello"; "hi"; "i")'
# "hi World"
```

Type Conversion

tostring

Convert to string:

```
echo '42' | jq 'tostring'
# "42"

echo 'true' | jq 'tostring'
# "true"

echo '{"a": 1}' | jq 'tostring'
# "{\"a\":1}"
```

tonumber

Convert string to number:

```
echo '"42"' | jq 'tonumber'
# 42

echo '"3.14"' | jq 'tonumber'
# 3.14
```

explode

Convert string to array of codepoints:

```
echo '"abc"' | jq 'explode'
# [97, 98, 99]

echo '"□□"' | jq 'explode'
# [26085, 26412]
```

implode

Convert array of codepoints to string:

```
echo '[97, 98, 99]' | jq 'implode'
# "abc"

echo '[26085, 26412]' | jq 'implode'
# "□□"
```

Formatting

@text

Identity for strings:

```
echo '"hello"' | jq '@text'
# "hello"
```

@json

Convert to JSON string:

```
echo '{"a": 1}' | jq '@json'
# "{\"a\":1}"
```

@html

Escape HTML entities:

```
echo '"<div class=\"test\">"' | jq '@html'
# "&lt;div class=&quot;test&quot;&gt;"
```

@uri

URL encode:

```
echo '"hello world"' | jq '@uri'
# "hello%20world"

echo '"foo=bar&baz=qux"' | jq '@uri'
# "foo%3Dbar%26baz%3Dqux"
```

@csv

Format as CSV row:

```
echo '["a", "b", "c"]' | jq '@csv'
# "\"a\",\"b\",\"c\""

echo '["hello", "world, there", "test"]' | jq '@csv'
# "\"hello\",\"world, there\",\"test\""
```

@tsv

Format as TSV row:

```
echo '["a", "b", "c"]' | jq '@tsv'
# "a\tb\tc"
```

@base64

Encode as base64:

```
echo '"hello world"' | jq '@base64'
# "aGVsbG8gd29ybGQ="
```

@base64d

Decode from base64:

```
echo '"aGVsbG8gd29ybGQ="' | jq '@base64d'
```

```
# "hello world"
```

@sh

Escape for shell:

```
echo '"hello world"' | jq '@sh'
# "'hello world'"

echo '"it'\''s here"' | jq '@sh'
# "'it'\\''s here'"
```

Null Bytes and Special Characters

Handle null bytes

```
echo '"hello\u0000world"' | jq '.'
# "hello\u0000world"
```

Newlines in strings

```
echo '"line1\nline2"' | jq '.'
# "line1\nline2"

echo '"line1\nline2"' | jq -r '.'
# line1
# line2
```

Escape Sequences

Sequence	Character
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash

`\"`	Double quote
`\uXXXX`	Unicode codepoint

```
jq -n '"\t indented \n newline"'
# "\t indented \n newline"

jq -rn '"\t indented \n newline"'
#      indented
#  newline
```

Chapter 10: Path Expressions

What Are Paths?

In jq, a path is an array describing the location of a value within a JSON structure. Each element in the path array is either a string (for object keys) or a number (for array indices).

```
echo '{"user": {"name": "Alice", "scores": [85, 90, 78]}}' | jq 'path(.user.name)'
# ["user", "name"]

echo '{"user": {"name": "Alice", "scores": [85, 90, 78]}}' | jq 'path(.user.scores[1])'
# ["user", "scores", 1]
```

Getting Paths

path(expression)

Get the path to a value:

```
echo '{"a": {"b": 1}}' | jq 'path(.a.b)'
# ["a", "b"]
```

Multiple paths from iteration:

```
echo '{"a": 1, "b": 2}' | jq '[path(.[])]'
# [["a"], ["b"]]
```

paths

Get all paths in a structure:

```
echo '{"a": {"b": 1}, "c": 2}' | jq '[paths]'
# [["a"], ["a", "b"], ["c"]]
```

paths(filter)

Get paths to values matching a filter:

```
echo '{"a": {"b": 1}, "c": "text"}' | jq '[paths(type == "number")]'
# [["a", "b"]]

echo '{"a": [1, 2], "b": {"c": 3}}' | jq '[paths(scalars)]'
# [["a", 0], ["a", 1], ["b", "c"]]
```

Common filters for paths:

```
# Paths to all scalar values
echo '{"a": {"b": 1}, "c": [2, 3]}' | jq '[paths(scalars)]'
# [["a", "b"], ["c", 0], ["c", 1]]

# Paths to all numbers
echo '{"a": "text", "b": 42}' | jq '[paths(numbers)]'
# [["b"]]

# Paths to all strings
echo '{"a": "text", "b": 42}' | jq '[paths(strings)]'
# [["a"]]

# Paths to all arrays
echo '{"a": [1], "b": {"c": [2]}}' | jq '[paths(arrays)]'
# [["a"], ["b", "c"]]
```

leaf_paths

Get paths to all leaf (scalar) values:

```
echo '{"a": {"b": 1, "c": 2}, "d": 3}' | jq '[leaf_paths]'
# [["a", "b"], ["a", "c"], ["d"]]
```

Equivalent to `[paths(scalars)]`.

Getting Values by Path

getpath(path)

Retrieve value at a given path:

```
echo '{"a": {"b": {"c": 42}}}' | jq 'getpath(["a", "b", "c"])'
# 42

echo '{"users": ["Alice", "Bob", "Charlie"]}' | jq 'getpath(["users", 1])'
# "Bob"
```

Non-existent paths return null:

```
echo '{"a": 1}' | jq 'getpath(["x", "y"])'
# null
```

Dynamic Path Access

```
echo '{"data": {"a": {"b": 1}}}' | jq '["data", "a", "b"] as $p | getpath($p)'
# 1
```

Setting Values by Path

setpath(path; value)

Set value at a given path:

```
echo '{"a": 1}' | jq 'setpath(["b"]; 2)'
# {"a": 1, "b": 2}

echo '{"a": 1}' | jq 'setpath(["b", "c"]; 2)'
# {"a": 1, "b": {"c": 2}}
```

Overwrites existing values:

```
echo '{"a": {"b": 1}}' | jq 'setpath(["a", "b"]; 99)'
# {"a": {"b": 99}}
```

Creates intermediate structure:

```
echo '{}' | jq 'setpath(["a", "b", "c"]; 1)'
# {"a": {"b": {"c": 1}}}

echo '{}' | jq 'setpath([0, 1, 2]; "x")'
# [[null, [null, null, "x"]]]
```

Setting Multiple Values

```
echo '{}' | jq 'setpath(["a"]; 1) | setpath(["b"]; 2)'
# {"a": 1, "b": 2}
```

Using reduce:

```
echo 'null' | jq 'reduce range(3) as $i ({}; setpath(["item\($i)"]; $i))'
# {"item0": 0, "item1": 1, "item2": 2}
```

Deleting by Path

delpaths(paths)

Delete multiple paths:

```
echo '{"a": 1, "b": 2, "c": 3}' | jq 'delpaths([["a"], ["c"]])'
# {"b": 2}
```

Delete nested paths:

```
echo '{"user": {"name": "Alice", "password": "secret", "email": "a@b.com"}}' | jq 'del
# {"user": {"name": "Alice", "email": "a@b.com"}}
```

del(path_expression)

More convenient syntax for deletion:

```
echo '{"a": 1, "b": 2}' | jq 'del(.a)'
# {"b": 2}
```

Delete using path variable:

```
echo '{"a": {"b": 1}}' | jq '["a", "b"] as $p | delpaths([$p])'
# {"a": {}}
```

Recursive Descent: ..

The `..` operator recursively descends into all values:

```
echo '{"a": {"b": 1}, "c": [2, 3]}' | jq '[.. | numbers]'
# [1, 2, 3]
```

Find all values at any depth:

```
echo '{"a": {"id": 1}, "b": {"c": {"id": 2}}}' | jq '[.. | .id? // empty]'
# [1, 2]
```

Combining with paths

```
echo '{"a": {"b": 1}, "c": 2}' | jq '[path(..)]'
# [[], ["a"], ["a", "b"], ["c"]]
```

Path Manipulation

Building Paths Dynamically

```
echo '["users", 0, "name"]' | jq '. as $p | {"a": {"users": [{"name": "Alice"}]}} | .a
# "Alice"
```

Path from String

Convert dot notation to path array:

```
echo '"a.b.c"' | jq 'split(".")'
# ["a", "b", "c"]
```

Handle array indices:

```
echo '"users[0].name"' | jq 'gsub("\\[(?<i>[0-9]+)\\]"; ".\(.i)") | split(".") | map(i
# ["users", 0, "name"]
```

Path to String

Convert path array to dot notation:

```
echo '["a", "b", "c"]' | jq 'join(".")'
# "a.b.c"
```

With array index notation:

```
echo '["users", 0, "name"]' | jq 'map(if type == "number" then "[\(.)]" else ".\(.)" e
# "users[0].name"
```

Modifying Values at Paths

Using getpath and setpath

```
echo '{"a": {"b": 5}}' | jq '["a", "b"] as $p | setpath($p; getpath($p) * 2)'
# {"a": {"b": 10}}
```

Using path with update

```
echo '{"counts": {"a": 1, "b": 2}}' | jq '.counts |= with_entries(.value += 10)'
# {"counts": {"a": 11, "b": 12}}
```

Finding Paths by Value

Find path to specific value

```
echo '{"a": {"b": "target"}, "c": "other"}' | jq 'path(.. | select(. == "target"))'
# ["a", "b"]
```

Find all paths to matching values

```
echo '{"a": 1, "b": {"c": 1, "d": 2}}' | jq '[paths(. == 1)]'
# [["a"], ["b", "c"]]
```

Walk Function

walk(filter)

Apply filter to every value recursively (bottom-up):

```
echo '{"a": {"b": 1}, "c": 2}' | jq 'walk(if type == "number" then . * 10 else . end)'
# {"a": {"b": 10}, "c": 20}
```

Transform all string values:

```
echo '{"name": "alice", "data": {"city": "london"}}' | jq 'walk(if type == "string" th
# {"name": "ALICE", "data": {"city": "LONDON"}}
```

Transform all keys:

```
echo '{"user_name": "alice", "user_age": 30}' | jq 'walk(if type == "object" then with
# {"user-name": "alice", "user-age": 30}
```

env and $ENV

Access environment variables:

$ENV

Object containing all environment variables:

```
jq -n '$ENV.HOME'
# "/home/user"

jq -n '$ENV.PATH'
# "/usr/local/bin:/usr/bin:..."
```

env

Same as $ENV:

```
jq -n 'env.USER'
# "username"
```

List all environment variables

```
jq -n '[env | to_entries[] | select(.key | startswith("XDG"))]'
```

Using environment in paths

```
export KEY="user"
echo '{"user": {"name": "Alice"}, "admin": {"name": "Bob"}}' | jq 'getpath([$ENV.KEY,
# "Alice"
```

Practical Examples

Flatten nested structure to paths

```
echo '{"a": {"b": 1, "c": 2}}' | jq '[paths(scalars) as $p | {path: ($p | join(".")),
# [{"path": "a.b", "value": 1}, {"path": "a.c", "value": 2}]
```

Reconstruct from paths

```
echo '[{"path": ["a", "b"], "value": 1}, {"path": ["a", "c"], "value": 2}]' | jq 'redu
# {"a": {"b": 1, "c": 2}}
```

Find and modify deeply nested values

```
echo '{"data": {"users": [{"active": false}, {"active": true}]}}' | jq '(.. | select(t
# {"data": {"users": [{"active": true}, {"active": false}]}}
```

Compare paths between two objects

```
echo '[{"a": 1, "b": 2}, {"a": 1, "c": 3}]' | jq '.[0] as $first | .[1] as $second | {
# {"only_first": [["b"]], "only_second": [["c"]]}
```

Chapter 11: Reduce and Recursion

Reduce

Basic Syntax

```
reduce EXPR as $var (INIT; UPDATE)
```

- `EXPR` - generates values to iterate over
- `$var` - variable bound to each value
- `INIT` - initial accumulator value
- `UPDATE` - expression to update accumulator (`.` refers to current accumulator)

Simple Examples

Sum numbers:

```
echo '[1, 2, 3, 4, 5]' | jq 'reduce .[] as $n (0; . + $n)'
# 15
```

Product:

```
echo '[1, 2, 3, 4, 5]' | jq 'reduce .[] as $n (1; . * $n)'
# 120
```

Count elements:

```
echo '["a", "b", "c"]' | jq 'reduce .[] as $x (0; . + 1)'
# 3
```

Building Arrays

Collect values:

```
echo '[1, 2, 3]' | jq 'reduce .[] as $n ([]; . + [$n * 2])'
# [2, 4, 6]
```

Filter while collecting:

```
echo '[1, 2, 3, 4, 5]' | jq 'reduce .[] as $n ([]; if $n > 2 then . + [$n] else . end)
# [3, 4, 5]
```

Building Objects

Construct object from array:

```
echo '[["a", 1], ["b", 2], ["c", 3]]' | jq 'reduce .[] as $pair ({}; .[$pair[0]] = $pa
# {"a": 1, "b": 2, "c": 3}
```

Count occurrences:

```
echo '["a", "b", "a", "c", "a", "b"]' | jq 'reduce .[] as $x ({}; .[$x] += 1)'
# {"a": 3, "b": 2, "c": 1}
```

Group by key:

```
echo '[{"k": "a", "v": 1}, {"k": "b", "v": 2}, {"k": "a", "v": 3}]' | jq 'reduce .[] a
# {"a": [1, 3], "b": [2]}
```

Reduce with Complex State

Track multiple values:

```
echo '[3, 1, 4, 1, 5, 9, 2, 6]' | jq 'reduce .[] as $n ({min: infinite, max: -infinite
# {"min": 1, "max": 9}
```

Running total:

```
echo '[1, 2, 3, 4, 5]' | jq 'reduce .[] as $n ({sum: 0, values: []}; {sum: (.sum + $n)
```

```
# {"sum": 15, "values": [1, 3, 6, 10, 15]}
```

Reduce vs map

Many `map` operations can be expressed as `reduce`:

```
# Using map
echo '[1, 2, 3]' | jq 'map(. * 2)'
# [2, 4, 6]

# Using reduce
echo '[1, 2, 3]' | jq 'reduce .[] as $n ([]; . + [$n * 2])'
# [2, 4, 6]
```

Use `reduce` when you need:

- Complex accumulator state
- Early termination logic
- Non-array results

foreach

Like reduce but outputs intermediate states:

```
foreach EXPR as $var (INIT; UPDATE)
foreach EXPR as $var (INIT; UPDATE; EXTRACT)
```

```
echo '[1, 2, 3, 4, 5]' | jq '[foreach .[] as $n (0; . + $n)]'
# [1, 3, 6, 10, 15]
```

With extract expression:

```
echo '[1, 2, 3, 4, 5]' | jq '[foreach .[] as $n (0; . + $n; "sum: \(.)")]'
# ["sum: 1", "sum: 3", "sum: 6", "sum: 10", "sum: 15"]
```

Recursion

recurse

Recursively apply a filter:

```
echo '5' | jq '[recurse(if . > 0 then . - 1 else empty end)]'
# [5, 4, 3, 2, 1, 0]
```

Traverse tree structure:

```
echo '{"name": "root", "children": [{"name": "a", "children": []}, {"name": "b", "chil
# ["root", "a", "b", "c"]
```

recurse(f)

Basic form - apply f until it produces no output:

```
echo '1' | jq '[recurse(. * 2; . < 100)]'
# [1, 2, 4, 8, 16, 32, 64]
```

recurse(f; condition)

Stop when condition is false:

```
echo '1' | jq '[recurse(. * 2; . < 50)]'
# [1, 2, 4, 8, 16, 32]
```

recurse_down

Same as `recurse` but guarantees depth-first order:

```
echo '{"a": {"b": {"c": 1}}}' | jq '[recurse_down | scalars]'
# [1]
```

Recursive Descent: ..

The `..` operator is shorthand for `recurse(.[]?)`:

```
echo '{"a": {"b": 1}, "c": [2, 3]}' | jq '[.. | numbers]'
# [1, 2, 3]
```

Equivalent to:

```
echo '{"a": {"b": 1}, "c": [2, 3]}' | jq '[recurse(.[]?) | numbers]'
# [1, 2, 3]
```

Walk

walk(f)

Apply filter to all values bottom-up:

```
echo '{"a": {"b": 1}}' | jq 'walk(if type == "number" then . * 10 else . end)'
# {"a": {"b": 10}}
```

Transform all strings:

```
echo '{"name": "alice", "items": [{"name": "item1"}]}' | jq 'walk(if type == "string"
# {"name": "ALICE", "items": [{"name": "ITEM1"}]}
```

Remove null values:

```
echo '{"a": 1, "b": null, "c": {"d": null, "e": 2}}' | jq 'walk(if type == "object" th
# {"a": 1, "c": {"e": 2}}
```

Order of Operations

`walk` processes bottom-up (children before parents):

```
echo '{"a": {"b": 1}}' | jq 'walk(debug)'
# ["DEBUG:",1]
# ["DEBUG:",{"b":1}]
# ["DEBUG:",{"a":{"b":1}}]
# {"a": {"b": 1}}
```

Iteration Constructs

while

Iterate while condition is true:

```
echo '1' | jq '[while(. < 100; . * 2)]'
# [1, 2, 4, 8, 16, 32, 64]
```

until

Iterate until condition is true:

```
echo '1' | jq 'until(. >= 100; . * 2)'
# 128
```

repeat

Repeat indefinitely (use with `limit`):

```
echo '1' | jq '[limit(5; repeat(. * 2))]'
# [2, 2, 2, 2, 2]
```

With state:

```
jq -n '[limit(5; 1 | recurse(. + 1))]'
# [1, 2, 3, 4, 5]
```

range

Generate number sequences:

```
jq -n '[range(5)]'
# [0, 1, 2, 3, 4]
```

```
jq -n '[range(2; 7)]'
# [2, 3, 4, 5, 6]

jq -n '[range(0; 10; 2)]'
# [0, 2, 4, 6, 8]
```

Limiting Output

limit(n; expr)

Take first n outputs:

```
jq -n '[limit(5; range(1000))]'
# [0, 1, 2, 3, 4]
```

first(expr)

Get first output only:

```
jq -n 'first(range(10))'
# 0
```

last(expr)

Get last output:

```
jq -n 'last(range(10))'
# 9
```

nth(n; expr)

Get nth output (zero-indexed):

```
jq -n 'nth(5; range(10))'
# 5
```

Practical Examples

Factorial

```
echo '5' | jq 'reduce range(1; . + 1) as $n (1; . * $n)'
# 120
```

Fibonacci Sequence

```
jq -n '[limit(10; [0, 1] | recurse([.[1], add]) | .[0])]'
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Flatten Deeply Nested Structure

```
echo '[[1, [2, [3, [4]]]]]' | jq '[recurse(arrays[]) | select(type != "array")]'
# [1, 2, 3, 4]
```

Find All Keys in Nested Object

```
echo '{"a": {"b": {"c": 1}}, "d": 2}' | jq '[.. | objects | keys[]] | unique'
# ["a", "b", "c", "d"]
```

Tree Traversal - Collect Leaf Paths

```
echo '{"a": {"b": 1, "c": 2}, "d": {"e": {"f": 3}}}' | jq '[paths(scalars)]'
# [["a", "b"], ["a", "c"], ["d", "e", "f"]]
```

Sum Values at All Levels

```
echo '{"a": 1, "b": {"c": 2, "d": {"e": 3}}}' | jq '[.. | numbers] | add'
# 6
```

Transform Nested Keys

```
echo '{"user_name": {"first_name": "alice"}}' | jq 'walk(if type == "object" then with
# {"user-name": {"first-name": "alice"}}
```

Running Average

```
echo '[10, 20, 30, 40, 50]' | jq 'reduce .[] as $n ({count: 0, sum: 0, avgs: []}; {cou
# {"count": 5, "sum": 150, "avgs": [10, 15, 20, 25, 30]}
```

State Machine

```
echo '["start", "data", "data", "end", "start", "data", "end"]' | jq 'reduce .[] as $e
  if $event == "start" then .state = "active"
  elif $event == "end" and .state == "active" then .state = "idle" | .sessions += 1
  else .
  end)'
# {"state": "idle", "sessions": 2}
```

Chapter 12: Defining Functions

Basic Function Definition

Syntax

```
def name: body;
def name(args): body;
```

Simple Functions

No arguments:

```
jq -n 'def double: . * 2; 5 | double'
# 10

jq -n 'def greet: "Hello, \(.)!"; "World" | greet'
# "Hello, World!"
```

Functions with Arguments

```
jq -n 'def add(x): . + x; 5 | add(3)'
# 8

jq -n 'def between(min; max): . >= min and . <= max; 5 | between(1; 10)'
# true
```

Multiple Arguments

Arguments are separated by semicolons:

```
jq -n 'def clamp(min; max): if . < min then min elif . > max then max else . end; 15 |
# 10
```

Filter Arguments vs Value Arguments

Filter Arguments (default)

Arguments are filters, evaluated in the caller's context:

```
jq -n 'def apply_twice(f): f | f; 2 | apply_twice(. * 2)'
# 8
```

The argument `f` is the filter `. * 2`, applied twice.

Value Arguments

Use `$name` to capture argument as a value:

```
jq -n 'def addvalue($x): . + $x; 5 | addvalue(3)'
# 8
```

Equivalent to:

```
jq -n 'def addvalue(x): x as $x | . + $x; 5 | addvalue(3)'
# 8
```

Difference Between Filter and Value Arguments

```
# Filter argument - evaluated each time
jq -n 'def call_twice(f): [f, f]; null | call_twice(now)'
# [1704067200.123, 1704067200.124]  (different values)

# Value argument - evaluated once
jq -n 'def call_twice($v): [$v, $v]; null | call_twice(now)'
# [1704067200.123, 1704067200.123]  (same value)
```

Recursive Functions

Functions can call themselves:

```
jq -n 'def factorial: if . <= 1 then 1 else . * ((. - 1) | factorial) end; 5 | factori
# 120
```

```
jq -n 'def fib: if . <= 1 then . else ((. - 1) | fib) + ((. - 2) | fib) end; 10 | fib'
# 55
```

Tail Recursion

More efficient for large inputs:

```
jq -n 'def factorial_tail($acc): if . <= 1 then $acc else (. - 1) | factorial_tail($ac
# 120
```

Multiple Definitions

Define multiple functions:

```
jq -n '
  def double: . * 2;
  def triple: . * 3;
  def process: double | triple;
  5 | process
'
# 30
```

Function Overloading

Functions with different arities are distinct:

```
jq -n '
  def inc: . + 1;
  def inc(x): . + x;
  [5 | inc, 5 | inc(10)]
'
# [6, 15]
```

Local Functions

Define functions inside other functions:

```
jq -n '
  def outer:
    def inner: . * 2;
    inner | inner;
  5 | outer
'
# 20
```

Using Functions with Arrays and Objects

Map-like Functions

```
jq -n '
  def square: . * .;
  [1, 2, 3, 4, 5] | map(square)
'
# [1, 4, 9, 16, 25]
```

Object Transformation

```
jq -n '
  def normalize_keys: with_entries(.key |= ascii_downcase);
  {"Name": "Alice", "AGE": 30} | normalize_keys
'
# {"name": "Alice", "age": 30}
```

Outputting Multiple Values

Functions can produce multiple outputs:

```
jq -n 'def pair: ., . * 2; 5 | pair'
# 5
# 10
```

Collect into array:

```
jq -n 'def pair: ., . * 2; [5 | pair]'
# [5, 10]
```

Error Handling in Functions

Using try-catch

```
jq -n '
  def safe_divide($n):
    if $n == 0 then error("division by zero")
    else . / $n
    end;
  10 | try safe_divide(0) catch "error: \(.)"
'
# "error: division by zero"
```

Returning null on Error

```
jq -n '
  def safe_parse: try tonumber catch null;
  ["42", "abc", "3.14"] | map(safe_parse)
'
# [42, null, 3.14]
```

Function Libraries

Using -f / --from-file

Save functions to a file:

```
# functions.jq
def double: . * 2;
def triple: . * 3;
def square: . * .;
```

Use with jq:

```
echo '5' | jq -f functions.jq 'double | triple'
# 30
```

Combining Library and Filter

```
echo '5' | jq -f functions.jq '. | double'
# 10
```

Using include

In jq files, include other files:

```
# main.jq
include "functions";

. | double | square
```

Using import

Import with namespace:

```
# main.jq
import "functions" as f;

. | f::double | f::square
```

Built-in Function Patterns

Mimicking map

```
jq -n '
  def mymap(f): [.[] | f];
  [1, 2, 3] | mymap(. * 2)
'
# [2, 4, 6]
```

Mimicking select

```
jq -n '
  def myselect(cond): if cond then . else empty end;
  [1, 2, 3, 4, 5] | .[] | myselect(. > 3)
'
# 4
# 5
```

Mimicking reduce

```
jq -n '
  def mysum: reduce .[] as $n (0; . + $n);
  [1, 2, 3, 4, 5] | mysum
'
# 15
```

Practical Function Examples

Safe Object Access

```
jq -n '
  def get($path; $default):
    getpath($path) // $default;
  {"a": {"b": 1}} | get(["a", "c"]; "not found")
'
# "not found"
```

Deep Merge

```
jq -n '
  def deep_merge(a; b):
    if (a | type) == "object" and (b | type) == "object" then
      a + b | with_entries(
        .key as $k |
        .value = deep_merge(a[$k]; b[$k])
      )
    else b // a
    end;
  deep_merge({"a": {"x": 1}}; {"a": {"y": 2}})
'
```

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

Flatten to Dot Notation

```
jq -n '
  def flatten_keys($prefix):
    to_entries | map(
      if .value | type == "object" then
        .value | flatten_keys("\($prefix)\(.key).")
      else
        {"\($prefix)\(.key)": .value}
      end
    ) | add;
  def flatten_keys: flatten_keys("");
  {"a": {"b": 1, "c": {"d": 2}}} | flatten_keys
'
# {"a.b": 1, "a.c.d": 2}
```

Chunk Array

```
jq -n '
  def chunk($n):
    if length == 0 then []
    else [.[:$n]] + (.[n:] | chunk($n))
    end;
  [1,2,3,4,5,6,7,8,9,10] | chunk(3)
'
# [[1,2,3], [4,5,6], [7,8,9], [10]]
```

Unique By

```
jq -n '
  def uniq_by(f):
    reduce .[] as $x ([];
      if any(.[]; (. | f) == ($x | f)) then .
      else . + [$x]
      end
    );
  [{"id": 1, "v": "a"}, {"id": 2, "v": "b"}, {"id": 1, "v": "c"}] | uniq_by(.id)
'
# [{"id": 1, "v": "a"}, {"id": 2, "v": "b"}]
```

Memoization Pattern

```
jq -n '
  def fib_memo:
    def go($memo):
      if . < 2 then [., $memo]
      else
        if $memo[.] then [$memo[.], $memo]
        else
          ((. - 1) | go($memo)) as [$a, $m1] |
          ((. - 2) | go($m1)) as [$b, $m2] |
          ($a + $b) as $result |
          [$result, ($m2 | .[$result] = $result)]
        end
      end;
    (go({}) | .[0]);
  35 | fib_memo
'
# 9227465
```

Tips and Best Practices

Use Descriptive Names

```
def extract_email: .contact.email;
def is_active_user: .status == "active" and .verified;
```

Keep Functions Small

```
def normalize: ascii_downcase | gsub("\\s+"; " ");
def trim: gsub("^\\s+|\\s+$"; "");
def clean_text: trim | normalize;
```

Document Complex Functions

```
# chunk(n) - Split array into chunks of size n
# Input: array
# Output: array of arrays
def chunk($n):
  if length <= $n then [.]
  else [.[:$n]] + (.[n:] | chunk($n))
  end;
```

Chapter 13: Input/Output

Reading Input

Standard Input

By default, jq reads JSON from stdin:

```
echo '{"name": "Alice"}' | jq '.name'
# "Alice"
```

From Files

```
jq '.name' data.json
# "Alice"
```

Multiple files:

```
jq '.name' file1.json file2.json
# "Alice"
# "Bob"
```

input

Read next JSON value from input:

```
echo -e '{"a": 1}\n{"a": 2}' | jq -n 'input'
# {"a": 1}
```

Read two inputs:

```
echo -e '{"a": 1}\n{"a": 2}' | jq -n '[input, input]'
# [{"a": 1}, {"a": 2}]
```

inputs

Read all remaining inputs:

```
echo -e '{"a": 1}\n{"a": 2}\n{"a": 3}' | jq -n '[inputs]'
# [{"a": 1}, {"a": 2}, {"a": 3}]
```

Process each input:

```
echo -e '1\n2\n3' | jq -n '[inputs | . * 2]'
# [2, 4, 6]
```

Slurping: -s / --slurp

Read all inputs into a single array:

```
echo -e '{"a": 1}\n{"a": 2}' | jq -s '.'
# [{"a": 1}, {"a": 2}]
```

Sum values from multiple JSON objects:

```
echo -e '1\n2\n3' | jq -s 'add'
# 6
```

Null Input: -n / --null-input

Don't read input, start with null:

```
jq -n '{"generated": true}'
# {"generated": true}
```

Combine with `inputs` to control reading:

```
echo -e '1\n2\n3' | jq -n 'reduce inputs as $n (0; . + $n)'
# 6
```

Raw Input

-R / --raw-input

Read lines as strings instead of JSON:

```
echo -e "line1\nline2\nline3" | jq -R '.'
# "line1"
# "line2"
# "line3"
```

Combine with Slurp

Read entire file as single string:

```
echo -e "line1\nline2" | jq -Rs '.'
# "line1\nline2\n"
```

Split into array of lines:

```
echo -e "line1\nline2\nline3" | jq -Rs 'split("\n") | map(select(. != ""))'
# ["line1", "line2", "line3"]
```

Processing Text Files

Word count:

```
echo "hello world hello" | jq -Rs 'split("\\s+"; "g") | map(select(. != "")) | length'
# 3
```

Line count:

```
echo -e "line1\nline2\nline3" | jq -Rs 'split("\n") | map(select(. != "")) | length'
# 3
```

Output Formatting

Pretty Print (default)

```
echo '{"a":1,"b":2}' | jq '.'
# {
#   "a": 1,
#   "b": 2
# }
```

Compact: -c / --compact-output

```
echo '{"a": 1, "b": 2}' | jq -c '.'
# {"a":1,"b":2}
```

Raw Output: -r / --raw-output

Output strings without quotes:

```
echo '{"name": "Alice"}' | jq '.name'
# "Alice"

echo '{"name": "Alice"}' | jq -r '.name'
# Alice
```

Join Output: -j / --join-output

Raw output without newlines:

```
echo '["a", "b", "c"]' | jq -j '.[]'
# abc
```

Tab Indentation: --tab

```
echo '{"a": {"b": 1}}' | jq --tab '.'
# {
# □"a": {
# □□"b": 1
# □}
```

```
# }
```

Custom Indentation: --indent n

```
echo '{"a": 1}' | jq --indent 4 '.'
# {
#     "a": 1
# }
```

Sort Keys: -S / --sort-keys

```
echo '{"c": 3, "a": 1, "b": 2}' | jq -S '.'
# {
#   "a": 1,
#   "b": 2,
#   "c": 3
# }
```

Color Output: -C / --color-output

Force colors (even when piping):

```
jq -C '.' data.json | less -R
```

Monochrome: -M / --monochrome-output

Disable colors:

```
jq -M '.' data.json > output.json
```

Format Strings

@text

Identity for strings:

```
echo '"hello"' | jq '@text'
# "hello"
```

@json

Convert to JSON string:

```
echo '{"a": 1}' | jq '@json'
# "{\"a\":1}"
```

Useful for embedding JSON:

```
echo '{"data": [1, 2, 3]}' | jq '{payload: (.data | @json)}'
# {"payload": "[1,2,3]"}
```

@html

Escape HTML entities:

```
echo '"<script>alert(1)</script>"' | jq '@html'
# "&lt;script&gt;alert(1)&lt;/script&gt;"
```

@uri

URL encode:

```
echo '"hello world"' | jq '@uri'
# "hello%20world"

echo '"name=test&value=1"' | jq '@uri'
# "name%3Dtest%26value%3D1"
```

@csv

Format as CSV:

```
echo '["Alice", 30, "London"]' | jq '@csv'
# "\"Alice\",30,\"London\""

echo '[["name", "age"], ["Alice", 30], ["Bob", 25]]' | jq '.[] | @csv'
# "\"name\",\"age\""
# "\"Alice\",30"
# "\"Bob\",25"
```

With -r for usable output:

```
echo '[["name", "age"], ["Alice", 30]]' | jq -r '.[] | @csv'
# "name","age"
# "Alice",30
```

@tsv

Format as TSV:

```
echo '["Alice", 30, "London"]' | jq '@tsv'
# "Alice\t30\tLondon"

echo '[["name", "age"], ["Alice", 30]]' | jq -r '.[] | @tsv'
# name	age
# Alice	30
```

@base64

Encode as base64:

```
echo '"hello world"' | jq '@base64'
# "aGVsbG8gd29ybGQ="
```

@base64d

Decode from base64:

```
echo '"aGVsbG8gd29ybGQ="' | jq '@base64d'
# "hello world"
```

@sh

Escape for shell:

```
echo '"hello world"' | jq '@sh'
# "'hello world'"

echo '["echo", "hello world", "foo'\''bar"]' | jq '@sh'
# "'echo' 'hello world' 'foo'\\''bar'"
```

Use in shell scripts:

```
args=$(echo '["ls", "-la", "/tmp"]' | jq -r '@sh')
eval $args
```

Loading External Data

--slurpfile

Load JSON file into variable as array:

```
# config.json contains: {"debug": true}
echo '{"name": "app"}' | jq --slurpfile cfg config.json '. + $cfg[0]'
# {"name": "app", "debug": true}
```

--jsonargs

Pass JSON arguments:

```
jq -n --jsonargs '{items: $ARGS.positional}' '["a", "b"]' '{"x": 1}'
# {"items": [["a", "b"], {"x": 1}]}
```

--args

Pass string arguments:

```
jq -n --args '{items: $ARGS.positional}' hello world
# {"items": ["hello", "world"]}
```

--rawfile

Load file as raw string:

```
# template.txt contains: Hello, {{name}}!
echo '{"name": "Alice"}' | jq --rawfile tpl template.txt '$tpl | gsub("\\{\\{name\\}\\
```

--arg

Pass single string value:

```
jq -n --arg name "Alice" '{"greeting": "Hello, \($name)"}'
# {"greeting": "Hello, Alice"}
```

--argjson

Pass single JSON value:

```
jq -n --argjson count 42 '{"count": $count}'
# {"count": 42}

jq -n --argjson enabled true '{"enabled": $enabled}'
# {"enabled": true}
```

Streaming

--stream

Parse input as stream of path-value pairs:

```
echo '{"a": {"b": 1}, "c": 2}' | jq --stream '.'
# [["a", "b"], 1]
# [["a", "b"]]
# [["a"], {"b": 1}]
```

```
# [["c"], 2]
# [["c"]]
```

truncate_stream

Remove path prefix:

```
echo '{"a": {"b": 1}}' | jq --stream 'truncate_stream(1)'
# [["b"], 1]
# [["b"]]
```

fromstream

Reconstruct from stream:

```
echo '[[["a"], 1], [["b"], 2]]' | jq 'fromstream(.[])'
# {"a": 1, "b": 2}
```

tostream

Convert to stream:

```
echo '{"a": {"b": 1}}' | jq 'tostream'
# [["a", "b"], 1]
# [["a"]]
```

Processing Large Files

Stream through large file without loading entirely:

```
jq --stream 'select(.[0][0] == "users") | .[1]' huge.json
```

Debugging

debug

Output debug info to stderr:

```
echo '5' | jq '. | debug | . * 2'
# ["DEBUG:", 5]
# 10
```

With label:

```
echo '5' | jq '. as $x | ($x | debug("input")) | . * 2'
# ["DEBUG (input):", 5]
# 10
```

stderr

Write to stderr:

```
echo '{"status": "ok"}' | jq '.status | debug | "processed"'
# ["DEBUG:", "ok"]
# "processed"
```

$__loc__

Get source location (useful in functions):

```
jq -n 'def f: $__loc__; f'
# {"file": "<cmdline>", "line": 1}
```

Exit Status

-e / --exit-status

Set exit status based on output:

```
echo 'null' | jq -e '.'
echo $?
# 1
```

```
echo '{"a": 1}' | jq -e '.a'
echo $?
# 0
```

Exit codes with -e:

Code	Meaning
0	Last output not false/null
1	Last output was false or null
5	No valid outputs

Use in Scripts

```
if echo '{"active": true}' | jq -e '.active' > /dev/null; then
    echo "Is active"
fi
```

Practical Examples

JSON to CSV

```
echo '[{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]' | jq -r '(.[0] | key
# "name","age"
# "Alice",30
# "Bob",25
```

CSV to JSON

```
echo -e "name,age\nAlice,30\nBob,25" | jq -Rs '
  split("\n") | map(select(. != "")) |
  (.[0] | split(",")) as $headers |
  .[1:] | map(split(",") | with_entries(.key = $headers[.key]))
'
# [{"name": "Alice", "age": "30"}, {"name": "Bob", "age": "25"}]
```

Merge Multiple JSON Files

```
jq -s 'add' file1.json file2.json file3.json
```

Process NDJSON (Newline Delimited JSON)

```
echo -e '{"a": 1}\n{"a": 2}\n{"a": 3}' | jq -c 'select(.a > 1)'
# {"a":2}
# {"a":3}
```

Generate Shell Variables

```
eval $(echo '{"name": "Alice", "age": 30}' | jq -r 'to_entries | .[] | "export \(.key)
echo $name $age
# Alice 30
```

Build URL Query String

```
echo '{"name": "Alice Smith", "age": 30}' | jq -r 'to_entries | map("\(.key)=\(.value
# name=Alice%20Smith&age=30
```

Chapter 14: Common Recipes

API Response Processing

Extract Nested Data

```
curl -s https://api.github.com/users/octocat | jq '{name: .name, repos: .public_repos,
```

Handle Paginated Results

```
# Combine multiple pages
curl -s 'https://api.example.com/items?page=1' 'https://api.example.com/items?page=2'
```

Extract from Wrapped Response

```
echo '{"status": "ok", "data": {"users": [{"name": "Alice"}]}}' | jq '.data.users'
# [{"name": "Alice"}]
```

Flatten Nested API Response

```
echo '{"results": [{"user": {"name": "Alice", "email": "a@b.com"}, "score": 95}]}' | j
# {"name": "Alice", "email": "a@b.com", "score": 95}
```

DevOps and Infrastructure

Parse Docker Inspect

```
docker inspect container_name | jq '.[0] | {id: .Id[:12], image: .Config.Image, status
```

Filter Running Containers

```
docker ps --format '{{json .}}' | jq -s 'map(select(.State == "running")) | .[].Names'
```

Parse Kubernetes Output

```
kubectl get pods -o json | jq '.items[] | {name: .metadata.name, status: .status.phase
```

Filter Pods by Status

```
kubectl get pods -o json | jq '.items[] | select(.status.phase != "Running") | .metada
```

AWS EC2 Instances

```
aws ec2 describe-instances | jq '.Reservations[].Instances[] | {id: .InstanceId, type:
```

Filter by Tag

```
aws ec2 describe-instances | jq '.Reservations[].Instances[] | select(.Tags[]? | selec
```

Terraform State

```
terraform show -json | jq '.values.root_module.resources[] | {type: .type, name: .name
```

Log Processing

Parse JSON Logs

```
cat app.log | jq -c 'select(.level == "error")'
```

Extract Timerange

```
cat app.log | jq -c 'select(.timestamp >= "2024-01-01" and .timestamp < "2024-01-02")'
```

Count by Level

```
cat app.log | jq -s 'group_by(.level) | map({level: .[0].level, count: length})'
```

Top Error Messages

```
cat app.log | jq -s '[.[] | select(.level == "error")] | group_by(.message) | map({mes
```

Extract Stack Traces

```
cat app.log | jq -r 'select(.stack != null) | "\(.timestamp) \(.message)\n\(.stack)\n-
```

Data Transformation

Reshape Object

```
echo '{"first_name": "Alice", "last_name": "Smith", "age": 30}' | jq '{name: "\(.first
# {"name": "Alice Smith", "age": 30}
```

Pivot Data

```
echo '[{"date": "2024-01", "type": "a", "value": 10}, {"date": "2024-01", "type": "b",
# [{"date": "2024-01", "a": 10, "b": 20}]
```

Unpivot Data

```
echo '{"date": "2024-01", "a": 10, "b": 20}' | jq '. as $row | keys_unsorted | map(sel
# [{"date": "2024-01", "type": "a", "value": 10}, {"date": "2024-01", "type": "b", "va
```

Normalize Nested Arrays

```
echo '{"user": "Alice", "orders": [{"id": 1}, {"id": 2}]}' | jq '.orders[] | {user: "A
# {"user": "Alice", "order_id": 1}
# {"user": "Alice", "order_id": 2}
```

Denormalize (Join)

```
echo '{"users": [{"id": 1, "name": "Alice"}], "orders": [{"user_id": 1, "item": "Book"
```

Filtering and Searching

Filter by Field Value

```
echo '[{"name": "Alice", "active": true}, {"name": "Bob", "active": false}]' | jq '.[]
# {"name": "Alice", "active": true}
```

Filter by Multiple Conditions

```
echo '[{"name": "Alice", "age": 30, "city": "London"}]' | jq '.[] | select(.age > 25 a
```

Search in Strings

```
echo '[{"name": "Alice"}, {"name": "Bob"}, {"name": "Alison"}]' | jq '.[] | select(.na
# {"name": "Alice"}
# {"name": "Alison"}
```

Case-Insensitive Search

```
echo '[{"name": "Alice"}, {"name": "ALICE"}]' | jq '.[] | select(.name | test("alice";
```

Find in Nested Structure

```
echo '{"a": {"b": {"target": 1}}, "c": {"target": 2}}' | jq '.. | objects | select(has
# 1
# 2
```

Filter by Array Contains

```
echo '[{"name": "Alice", "tags": ["admin", "user"]}, {"name": "Bob", "tags": ["user"]}
# {"name": "Alice", "tags": ["admin", "user"]}
```

Aggregation

Sum

```
echo '[{"amount": 10}, {"amount": 20}, {"amount": 30}]' | jq 'map(.amount) | add'
# 60
```

Average

```
echo '[{"value": 10}, {"value": 20}, {"value": 30}]' | jq 'map(.value) | add / length'
# 20
```

Min / Max

```
echo '[{"score": 85}, {"score": 92}, {"score": 78}]' | jq 'map(.score) | {min: min, ma
# {"min": 78, "max": 92}
```

Count by Category

```
echo '[{"type": "a"}, {"type": "b"}, {"type": "a"}]' | jq 'group_by(.type) | map({type
# [{"type": "a", "count": 2}, {"type": "b", "count": 1}]
```

Sum by Category

```
echo '[{"type": "a", "value": 10}, {"type": "b", "value": 20}, {"type": "a", "value":
# [{"type": "a", "total": 40}, {"type": "b", "total": 20}]
```

Running Total

```
echo '[1, 2, 3, 4, 5]' | jq '[foreach .[] as $n (0; . + $n)]'
# [1, 3, 6, 10, 15]
```

Format Conversion

JSON to CSV

```
echo '[{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}]' | jq -r '(.[0] | key
# "name","age"
# "Alice",30
# "Bob",25
```

JSON to TSV

```
echo '[{"name": "Alice", "age": 30}]' | jq -r '(.[0] | keys_unsorted), (.[] | [.[]]) |
# name□age
# Alice□30
```

CSV to JSON

```
echo -e "name,age\nAlice,30\nBob,25" | jq -Rs 'split("\n") | map(select(. != "")) | (.
```

JSON to YAML-like

```
echo '{"name": "Alice", "age": 30}' | jq -r 'to_entries | map("\(.key): \(.value)") |
# name: Alice
# age: 30
```

Flatten to Key=Value

```
echo '{"db": {"host": "localhost", "port": 5432}}' | jq -r 'paths(scalars) as $p | "\(
# DB_HOST=localhost
# DB_PORT=5432
```

Configuration Management

Merge Configs

```
echo '{"a": 1, "b": 2}' | jq --argjson override '{"b": 99, "c": 3}' '. + $override'
# {"a": 1, "b": 99, "c": 3}
```

Deep Merge Configs

```
echo '{"db": {"host": "localhost"}}' | jq --argjson override '{"db": {"port": 5432}}'
# {"db": {"host": "localhost", "port": 5432}}
```

Environment Variable Substitution

```
export DB_HOST="production.db.com"
echo '{"db": {"host": "localhost"}}' | jq --arg host "$DB_HOST" '.db.host = $host'
# {"db": {"host": "production.db.com"}}
```

Conditional Config

```
echo '{"env": "prod"}' | jq 'if .env == "prod" then . + {debug: false, log_level: "err
```

Extract Specific Keys

```
echo '{"a": 1, "b": 2, "c": 3, "d": 4}' | jq '{a, c}'
# {"a": 1, "c": 3}
```

Remove Sensitive Keys

```
echo '{"user": "alice", "password": "secret", "token": "abc123"}' | jq 'del(.password,
# {"user": "alice"}
```

Package.json Operations

Get Version

```
jq -r '.version' package.json
```

List Dependencies

```
jq -r '.dependencies | keys[]' package.json
```

Check Dependency Version

```
jq -r '.dependencies["lodash"]' package.json
```

Bump Version

```
jq '.version = "2.0.0"' package.json
```

Add Script

```
jq '.scripts.lint = "eslint ."' package.json
```

Merge Dependencies

```
jq -s '.[0].dependencies + .[1].dependencies' package1.json package2.json
```

Diff and Comparison

Compare Two Objects

```
echo '[{"a": 1, "b": 2}, {"a": 1, "b": 3}]' | jq '.[0] as $a | .[1] as $b | {same: ($a
```

Find Added Keys

```
echo '[{"a": 1}, {"a": 1, "b": 2}]' | jq '(.[1] | keys) - (.[0] | keys)'
# ["b"]
```

Find Removed Keys

```
echo '[{"a": 1, "b": 2}, {"a": 1}]' | jq '(.[0] | keys) - (.[1] | keys)'
# ["b"]
```

Deep Diff Paths

```
echo '[{"a": {"b": 1}}, {"a": {"b": 2}}]' | jq '[.[0], .[1]] | [paths(scalars)] as $pa
```

Batch Processing

Process Multiple Files

```
for f in *.json; do
    jq '.processed = true' "$f" > "processed_$f"
done
```

Combine JSON Files

```
jq -s '.' *.json > combined.json
```

Merge Array Files

```
jq -s 'add' file1.json file2.json > merged.json
```

Filter Across Files

```
cat *.json | jq -c 'select(.status == "active")'
```

Shell Integration

Generate Shell Variables

```
eval $(echo '{"host": "localhost", "port": 8080}' | jq -r 'to_entries | .[] | "export
```

Build Command Arguments

```
args=$(echo '["--verbose", "--config", "app.json"]' | jq -r '@sh')
eval "mycommand $args"
```

Create Filenames from JSON

```
echo '[{"id": 1, "name": "report"}, {"id": 2, "name": "summary"}]' | jq -r '.[] | "\(.
# 1_report.txt
# 2_summary.txt
```

Pass JSON to Script

```
echo '{"name": "Alice"}' | jq -r '@json' | xargs -I {} ./process.sh '{}'
```

Error Handling Recipes

Default for Missing Keys

```
echo '{"a": 1}' | jq '.b // "default"'
# "default"
```

Safely Access Nested

```
echo '{"a": 1}' | jq '.b.c.d? // "not found"'
# "not found"
```

Skip Invalid JSON Lines

```
cat mixed.log | while read line; do
    echo "$line" | jq '.' 2>/dev/null || true
done
```

Validate JSON Structure

```
echo '{"name": "Alice"}' | jq 'if has("name") and has("email") then . else error("miss
```

Coalesce Multiple Fields

```
echo '{"nickname": "Al"}' | jq '.name // .nickname // .username // "anonymous"'
# "Al"
```

Chapter 15: Quick Reference Tables

Command-Line Options

Option	Long Form	Description
`-c`	`--compact-output`	Compact output (no pretty-print)
`-r`	`--raw-output`	Output strings without quotes
`-j`	`--join-output`	Raw output without newlines
`-n`	`--null-input`	Don't read input; start with null
`-s`	`--slurp`	Read all inputs into array
`-R`	`--raw-input`	Read lines as strings
`-e`	`--exit-status`	Set exit code based on output
`-S`	`--sort-keys`	Sort object keys
`-C`	`--color-output`	Force colored output
`-M`	`--monochrome-output`	Disable colors
`-f`	`--from-file`	Read filter from file
`-a`	`--ascii-output`	Escape non-ASCII characters
	`--tab`	Use tabs for indentation
	`--indent n`	Set indentation (0-7)
	`--arg name val`	Pass string variable
	`--argjson name val`	Pass JSON variable
	`--slurpfile name file`	Load JSON file as variable
	`--rawfile name file`	Load file as string variable
	`--args`	Remaining args as strings
	`--jsonargs`	Remaining args as JSON

Basic Filters

Filter	Description	Example
`.`	Identity	`jq '.'`
`.field`	Field access	`.name`
`.[n]`	Array index	`.[0]`
`.[-n]`	Negative index	`.[-1]`
`.[m:n]`	Array slice	`.[1:3]`
`.[]`	Iterate all	`.[]`
`.[]?`	Iterate (no error)	`.[]?`
`.field?`	Optional field	`.foo?`
`..`	Recursive descent	`.. \`

Operators

Arithmetic

Operator	Description	Example
`+`	Add / concat	`. + 1`, `"a" + "b"`
`-`	Subtract / diff	`. - 1`, `[1,2] - [2]`
`*`	Multiply / repeat	`. * 2`, `"ab" * 3`
`/`	Divide / split	`. / 2`, `"a,b" / ","`
`%`	Modulo	`. % 3`

Comparison

Operator	Description
`==`	Equal

`!=`	Not equal
`<`	Less than
`<=`	Less or equal
`>`	Greater than
`>=`	Greater or equal

Logical

Operator	Description
`and`	Logical AND
`or`	Logical OR
`not`	Logical NOT

Other

Operator	Description	Example
`\	`	Pipe
`,`	Multiple outputs	`.a, .b`
`//`	Alternative	`.a // "default"`
`?//`	Try-catch	`.a? // "error"`

Update Operators

Operator	Description	Example
`=`	Assign	`.a = 1`
`\	=`	Update
`+=`	Add-assign	`.a += 1`

`-=`	Subtract-assign	`.a -= 1`
`*=`	Multiply-assign	`.a *= 2`
`/=`	Divide-assign	`.a /= 2`
`%=`	Modulo-assign	`.a %= 2`
`//=`	Alternative-assign	`.a //= "default"`

Type Functions

Function	Description
`type`	Get type as string
`isinfinite`	Test for infinity
`isnan`	Test for NaN
`isnormal`	Test for normal number
`isfinite`	Test for finite number
`strings`	Select strings
`numbers`	Select numbers
`booleans`	Select booleans
`nulls`	Select nulls
`arrays`	Select arrays
`objects`	Select objects
`iterables`	Select arrays/objects
`scalars`	Select non-iterables
`values`	Select non-null

Type Conversion

Function	Description	Example

`tostring`	Convert to string	`42 \
`tonumber`	Convert to number	`"42" \
`explode`	String to codepoints	`"ab" \
`implode`	Codepoints to string	`[97,98] \

Array Functions

Function	Description
`length`	Array length
`reverse`	Reverse array
`sort`	Sort array
`sort_by(f)`	Sort by expression
`unique`	Remove duplicates
`unique_by(f)`	Unique by expression
`group_by(f)`	Group by expression
`flatten`	Flatten nested arrays
`flatten(n)`	Flatten n levels
`min`	Minimum value
`max`	Maximum value
`min_by(f)`	Minimum by expression
`max_by(f)`	Maximum by expression
`add`	Sum / concatenate
`first`	First element
`last`	Last element
`nth(n)`	Nth element
`index(x)`	First index of x
`rindex(x)`	Last index of x

`indices(x)`	All indices of x
`contains(x)`	Test containment
`inside(x)`	Reverse containment
`map(f)`	Apply f to each
`map_values(f)`	Apply f to values
`select(f)`	Filter by condition
`empty`	Produce no output
`range(n)`	Generate 0 to n-1
`range(m;n)`	Generate m to n-1
`transpose`	Transpose matrix

Object Functions

Function	Description
`keys`	Sorted keys array
`keys_unsorted`	Keys in order
`has(k)`	Test for key
`in(obj)`	Test key in object
`to_entries`	Object to [{key,value}]
`from_entries`	[{key,value}] to object
`with_entries(f)`	Transform entries
`del(path)`	Delete at path
`getpath(p)`	Get value at path
`setpath(p;v)`	Set value at path
`delpaths(ps)`	Delete multiple paths
`paths`	All paths
`paths(f)`	Paths matching filter

`leaf_paths`	Paths to scalars

String Functions

Function	Description
`length`	Character count
`utf8bytelength`	Byte count
`split(s)`	Split by string
`join(s)`	Join with string
`ltrimstr(s)`	Remove prefix
`rtrimstr(s)`	Remove suffix
`ascii_downcase`	Lowercase
`ascii_upcase`	Uppercase
`startswith(s)`	Test prefix
`endswith(s)`	Test suffix
`index(s)`	First position
`rindex(s)`	Last position
`inside(s)`	Test substring
`contains(s)`	Test contains
`test(re)`	Regex test
`match(re)`	Regex match
`capture(re)`	Named captures
`scan(re)`	All matches
`splits(re)`	Split by regex
`sub(re;s)`	Replace first
`gsub(re;s)`	Replace all

Format Strings

Format	Description	Example
`@text`	Identity	`"hi" \
`@json`	JSON encode	`{a:1} \
`@html`	HTML escape	`"<>" \
`@uri`	URL encode	`"a b" \
`@csv`	CSV format	`["a","b"] \
`@tsv`	TSV format	`["a","b"] \
`@base64`	Base64 encode	`"hi" \
`@base64d`	Base64 decode	`"aGk=" \
`@sh`	Shell escape	`"a b" \

Conditionals

Syntax	Description
`if C then T else F end`	Conditional
`if C then T elif C2 then T2 else F end`	Multiple conditions
`select(C)`	Filter by condition
`empty`	No output
`error(msg)`	Raise error
`try E catch F`	Error handling
`E?`	Suppress errors

Comparison Functions

Function	Description

`min`	Minimum of array
`max`	Maximum of array
`min_by(f)`	Minimum by expression
`max_by(f)`	Maximum by expression
`any`	Any truthy
`any(f)`	Any matching
`all`	All truthy
`all(f)`	All matching
`contains(x)`	Deep contains
`inside(x)`	Reverse contains
`limit(n;f)`	First n outputs
`first(f)`	First output
`last(f)`	Last output
`nth(n;f)`	Nth output

Reduce and Iteration

Syntax	Description
`reduce E as $v (I; U)`	Fold values
`foreach E as $v (I; U)`	Fold with outputs
`while(C; U)`	While loop
`until(C; U)`	Until loop
`repeat(E)`	Repeat forever
`recurse(f)`	Recursive apply
`recurse(f; C)`	Recurse with condition
`walk(f)`	Transform all values

Path Functions

Function	Description
`path(E)`	Get path to value
`paths`	All paths
`paths(f)`	Paths to matching
`leaf_paths`	Paths to leaves
`getpath(p)`	Value at path
`setpath(p;v)`	Set at path
`delpaths(ps)`	Delete paths

I/O Functions

Function	Description
`input`	Read next input
`inputs`	Read all inputs
`debug`	Debug output
`debug(msg)`	Debug with message
`stderr`	Write to stderr
`$ENV`	Environment object
`env`	Environment object
`$__loc__`	Source location

Defining Functions

```
# No arguments
def name: body;

# With arguments (filters)
```

```
def name(a; b): body;

# With value arguments
def name($a; $b): body;

# Recursive
def fact: if . <= 1 then 1 else . * ((. - 1) | fact) end;
```

Exit Codes

Code	Meaning
0	Success
1	With `-e`: output was false/null
2	Usage error
3	Compile error
4	Runtime error
5	With `-e`: no valid outputs

Regex Flags

Flag	Description
`i`	Case insensitive
`x`	Extended (ignore whitespace)
`m`	Multiline (^ $ match lines)
`s`	Single line (. matches newline)
`g`	Global (for sub)

Comparison Order

Values of different types compare as:

```
null < false < true < numbers < strings < arrays < objects
```

Common Patterns

Task	Pattern
Pretty print	`jq '.'`
Compact	`jq -c '.'`
Get field	`jq '.field'`
Raw string	`jq -r '.field'`
Filter array	`jq 'map(select(.x > 1))'`
Transform	`jq 'map(.x * 2)'`
Unique values	`jq 'unique'`
Sort	`jq 'sort_by(.field)'`
Group	`jq 'group_by(.field)'`
Count	`jq 'length'`
Sum	`jq 'map(.n) \`
First N	`jq '.[:5]'`
Last N	`jq '.[-5:]'`
Keys	`jq 'keys'`
Values	`jq '[.[]]'`
Flatten	`jq 'flatten'`
Merge objects	`jq '.[0] + .[1]'`
Deep merge	`jq '.[0] * .[1]'`
Default value	`jq '.x // "default"'`
Delete field	`jq 'del(.field)'`
Rename field	`jq '{new: .old}'`
JSON to CSV	`jq -r '.[] \`