JQ

L'Operatore Pipe |

Concetto Fondamentale

L'operatore **pipe** (`|`) è il concetto centrale di jq e rappresenta il **flusso di dati** attraverso una sequenza di trasformazioni.

Funzionamento

```
input | filter1 | filter2 | filter3 | output
```

Il pipe prende l'**output** del filtro a sinistra e lo passa come **input** al filtro a destra.

Esempi Pratici

```
# Esempio 1: Chain semplice
echo '{"name": "Mario", "age": 30}' | jq '.name | length'
```

Flusso:

1. Input: `{"name": "Mario", "age": 30}`
2. `.name` estrae: `"Mario"`
3. `| length` calcola: `5`
4. Output: `5`

```
# Esempio 2: Chain complessa
echo '[1,2,3,4,5]' | jq '.[] | . * 2 | select(. > 5)'
```

Flusso:

1. Input: `[1,2,3,4,5]`
2. `.[]` produce: `1`, `2`, `3`, `4`, `5` (valori separati)
3. `. * 2` moltiplica: `2`, `4`, `6`, `8`, `10`
4. `select(. > 5)` filtra: `6`, `8`, `10`

Differenza con Altri Operatori

```
# Pipe |  -> passa l'output come input
.name | length

# Comma , -> produce output multipli indipendenti
.name, .age

# Semicolon ; -> separa espressioni indipendenti
.name; .age
```

VOCABOLARIO JQ

1. OPERATORI DI BASE

`.` (Identity / Dot)

Descrizione: Rappresenta l'input corrente, invariato.

```
# Restituisce l'intero input
echo '{"x": 1}' | jq '.'
# Output: {"x": 1}

# Usato per riferirsi al valore corrente
echo '[1,2,3]' | jq '.[] | . * 2'
# Output: 2, 4, 6
```

`.field` (Field Access)

Descrizione: Accede a un campo di un oggetto.

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

# Accesso annidato
echo '{"person": {"name": "Mario"}}' | jq '.person.name'
# Output: "Mario"
```

`.field?` (Optional Field Access)

Descrizione: Accesso sicuro che non genera errore se il campo non esiste.

```
echo '{"name": "Mario"}' | jq '.age?'
# Output: null (invece di errore)

echo '{"name": "Mario", "age": 30}' | jq '.age?'
# Output: 30
```

`.[index]` (Array Index)

Descrizione: Accede a un elemento di un array per indice (0-based).

```
echo '[10, 20, 30]' | jq '.[1]'
# Output: 20

# Indice negativo (dall'ultimo)
echo '[10, 20, 30]' | jq '.[-1]'
# Output: 30
```

`.[start:end]` (Array Slice)

Descrizione: Estrae una porzione di array.

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

# Dall'inizio al 3° elemento
echo '[0,1,2,3,4,5]' | jq '.[:3]'
# Output: [0, 1, 2]

# Dal 3° elemento alla fine
echo '[0,1,2,3,4,5]' | jq '.[3:]'
# Output: [3, 4, 5]
```

`.[]` (Array Iterator)

Descrizione: Itera su tutti gli elementi di un array, producendo valori separati.

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

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

---

2. OPERATORI DI COSTRUZIONE

`[]` (Array Construction)

Descrizione: Crea un array raccogliendo gli output.

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

# Raccoglie risultati multipli
echo '{"items":[1,2,3]}' | jq '[.items[] | . * 2]'
# Output: [2, 4, 6]
```

`{}` (Object Construction)

Descrizione: Crea un oggetto JSON.

```
# Campi espliciti
echo '{"x":1, "y":2}' | jq '{a: .x, b: .y}'
# Output: {"a": 1, "b": 2}

# Shorthand (chiave = nome campo)
echo '{"name":"Mario", "age":30}' | jq '{name, age}'
# Output: {"name": "Mario", "age": 30}

# Chiavi calcolate
echo '{"key":"name", "value":"Mario"}' | jq '{(.key): .value}'
# Output: {"name": "Mario"}
```

---

3. OPERATORI LOGICI E DI COMPARAZIONE

`==` (Equals)

```
echo '{"x": 5}' | jq '.x == 5'
# Output: true
```

`!=` (Not Equals)

```
echo '{"x": 5}' | jq '.x != 10'
# Output: true
```

<, <=, >, >= (Comparison)

```
echo '{"x": 5}' | jq '.x > 3'
# Output: true
```

and, or, not (Boolean Logic)

```
echo '{"x": 5, "y": 10}' | jq '.x > 3 and .y < 20'
# Output: true

echo '{"x": 5}' | jq '.x > 10 or .x < 3'
# Output: false

echo '{"active": true}' | jq '.active | not'
# Output: false
```

---

4. OPERATORI ARITMETICI

+ (Addition / Concatenation)

```
# Numeri
echo '{"x": 5, "y": 3}' | jq '.x + .y'
# Output: 8

# Stringhe
echo '{"first": "Hello", "last": "World"}' | jq '.first + " " + .last'
# Output: "Hello World"

# Array
echo '{"a": [1,2], "b": [3,4]}' | jq '.a + .b'
# Output: [1, 2, 3, 4]

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

- (Subtraction / Array Difference)

```
# Numeri
echo '{"x": 10, "y": 3}' | jq '.x - .y'
# Output: 7

# Array (rimuove elementi)
echo '{"a": [1,2,3,4], "b": [2,4]}' | jq '.a - .b'
# Output: [1, 3]
```

* (Multiplication / String Repeat / Object Merge)

```
# Numeri
echo '{"x": 5, "y": 3}' | jq '.x * .y'
# Output: 15

# Stringhe (ripetizione)
echo '{"str": "ab"}' | jq '.str * 3'
# Output: "ababab"

# Oggetti (merge ricorsivo)
echo '{"a":{"x":1}, "b":{"y":2}}' | jq '.a * .b'
# Output: {"x": 1, "y": 2}
```

/ (Division)

```
echo '{"x": 10, "y": 2}' | jq '.x / .y'
# Output: 5
```

% (Modulo)

```
echo '{"x": 10, "y": 3}' | jq '.x % .y'
# Output: 1
```

---

5. FUNZIONI DI SELEZIONE E FILTRO

`select(condition)`

Descrizione: Passa attraverso solo i valori che soddisfano la condizione.

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

# Con oggetti
echo '[{"age":25},{"age":35}]' | jq '.[] | select(.age > 30)'
# Output: {"age": 35}
```

`empty`

Descrizione: Non produce output (utile per filtrare via elementi).

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

`error(message)`

Descrizione: Genera un errore con messaggio.

```
echo '{"x": -5}' | jq 'if .x < 0 then error("Valore negativo") else .x end'
# Output: errore con messaggio
```

---

6. FUNZIONI DI ARRAY

`length`

Descrizione: Restituisce lunghezza di array, oggetto o stringa.

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

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

echo '"Hello"' | jq 'length'
# Output: 5
```

`add`

Descrizione: Somma elementi di un array o merge oggetti.

```
# Somma numeri
echo '[1,2,3,4]' | jq 'add'
# Output: 10

# Concatena stringhe
echo '["Hello", " ", "World"]' | jq 'add'
# Output: "Hello World"

# Merge oggetti
echo '[{"a":1}, {"b":2}, {"c":3}]' | jq 'add'
# Output: {"a": 1, "b": 2, "c": 3}
```

`reverse`

Descrizione: Inverte un array.

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

`sort`

Descrizione: Ordina un array.

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

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

`sort_by(expression)`

Descrizione: Ordina array di oggetti per un campo.

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

# Ordine inverso
echo '[{"name":"Mario","age":30}, {"name":"Luigi","age":25}]' | jq 'sort_by(.age) | reverse'
```

`group_by(expression)`

Descrizione: Raggruppa elementi con stesso valore di espressione.

```
echo '[{"type":"A","val":1}, {"type":"B","val":2}, {"type":"A","val":3}]' | jq 'group_by(.type)'
# Output: [[{"type":"A","val":1}, {"type":"A","val":3}], [{"type":"B","val":2}]]
```

`unique`

Descrizione: Rimuove duplicati (array deve essere ordinato o si può usare `unique_by`).

```
echo '[1,2,2,3,3,3,4]' | jq 'unique'
# Output: [1, 2, 3, 4]
```

`unique_by(expression)`

Descrizione: Rimuove duplicati basandosi su un'espressione.

```
echo '[{"name":"Mario","age":30}, {"name":"Luigi","age":30}]' | jq 'unique_by(.age)'
# Output: [{"name":"Mario","age":30}]
```

`max`, `min`

Descrizione: Valore massimo/minimo in un array.

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

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

`max_by(expression)`, `min_by(expression)`

Descrizione: Elemento con valore massimo/minimo per un campo.

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

`flatten`

Descrizione: Appiattisce array annidati.

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

# Con profondità
echo '[[[1,2]], [[3,4]]]' | jq 'flatten(1)'
# Output: [[1, 2], [3, 4]]
```

`map(expression)`

Descrizione: Applica espressione a ogni elemento di un array.

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

echo '[{"x":1}, {"x":2}]' | jq 'map(.x)'
# Output: [1, 2]
```

`map_values(expression)`

Descrizione: Applica espressione a ogni valore di un oggetto.

```
echo '{"a":1, "b":2, "c":3}' | jq 'map_values(. * 2)'
# Output: {"a": 2, "b": 4, "c": 6}
```

---

7. FUNZIONI DI OGGETTI

keys

Descrizione: Restituisce array di chiavi (ordinate).

```
echo '{"z":1, "a":2, "m":3}' | jq 'keys'
# Output: ["a", "m", "z"]
```

keys_unsorted

Descrizione: Restituisce chiavi senza ordinamento.

```
echo '{"z":1, "a":2, "m":3}' | jq 'keys_unsorted'
# Output: ["z", "a", "m"]
```

values

Descrizione: Restituisce tutti i valori di un oggetto.

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

has(key)

Descrizione: Verifica se un oggetto ha una chiave.

```
echo '{"name":"Mario", "age":30}' | jq 'has("name")'
# Output: true

echo '{"name":"Mario", "age":30}' | jq 'has("address")'
# Output: false
```

in(object)

Descrizione: Verifica se una chiave esiste in un oggetto.

```
echo '"name"' | jq '. as $key | {"name":"Mario"} | has($key)'
# Output: true
```

del(path)

Descrizione: Elimina un campo o elemento.

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

# Elementi array
echo '[1,2,3,4]' | jq 'del(.[1])'
# Output: [1, 3, 4]
```

`with_entries(expression)`

Descrizione: Trasforma oggetto lavorando su coppie chiave-valore.

```
echo '{"a":1, "b":2}' | jq 'with_entries(.value = .value * 2)'
# Output: {"a": 2, "b": 4}

# Ogni entry è {key: "...", value: ...}
echo '{"a":1, "b":2}' | jq 'with_entries(.key = .key + "_new")'
# Output: {"a_new": 1, "b_new": 2}
```

`to_entries`

Descrizione: Converte oggetto in array di `{key, value}`.

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

`from_entries`

Descrizione: Converte array di `{key, value}` in oggetto.

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

# Alternativa con "name" invece di "key"
echo '[{"name":"a","value":1}]' | jq 'from_entries'
# Output: {"a": 1}
```

---

8. FUNZIONI DI STRINGHE

`tostring`

Descrizione: Converte valore in stringa.

```
echo '123' | jq 'tostring'
# Output: "123"

echo 'true' | jq 'tostring'
# Output: "true"
```

tonumber

Descrizione: Converte stringa in numero.

```
echo '"123"' | jq 'tonumber'
# Output: 123

echo '"123.45"' | jq 'tonumber'
# Output: 123.45
```

split(separator)

Descrizione: Divide stringa in array.

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

join(separator)

Descrizione: Unisce array in stringa.

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

startswith(string), endswith(string)

Descrizione: Verifica inizio/fine stringa.

```
echo '"Hello World"' | jq 'startswith("Hello")'
# Output: true

echo '"Hello World"' | jq 'endswith("World")'
# Output: true
```

contains(string)

Descrizione: Verifica se contiene sottostringa.

```
echo '"Hello World"' | jq 'contains("Wo")'
# Output: true
```

`test(regex)`, `test(regex; flags)`

Descrizione: Test con espressione regolare.

```
echo '"Hello123"' | jq 'test("[0-9]+")'
# Output: true

# Case insensitive
echo '"Hello"' | jq 'test("hello"; "i")'
# Output: true
```

`match(regex)`, `match(regex; flags)`

Descrizione: Restituisce match di regex.

```
echo '"Hello 123 World"' | jq 'match("[0-9]+")'
# Output: {"offset": 6, "length": 3, "string": "123", "captures": []}
```

`capture(regex)`, `capture(regex; flags)`

Descrizione: Cattura gruppi da regex in oggetto.

```
echo '"John Doe"' | jq 'capture("(?<first>\\w+) (?<last>\\w+)")'
# Output: {"first": "John", "last": "Doe"}
```

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

Descrizione: Sostituzione globale con regex.

```
echo '"Hello World"' | jq 'gsub("o"; "0")'
# Output: "Hell0 W0rld"

echo '"Hello World"' | jq 'gsub("world"; "Universe"; "i")'
# Output: "Hello Universe"
```

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

Descrizione: Sostituzione singola (prima occorrenza).

```
echo '"Hello World"' | jq 'sub("o"; "0")'
# Output: "Hell0 World"
```

`ascii_downcase`, `ascii_upcase`

Descrizione: Conversione maiuscole/minuscole.

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

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

`ltrimstr(string)`, `rtrimstr(string)`

Descrizione: Rimuove prefisso/suffisso.

```
echo '"Hello World"' | jq 'ltrimstr("Hello ")'
# Output: "World"

echo '"Hello World"' | jq 'rtrimstr(" World")'
# Output: "Hello"
```

---

9. FUNZIONI DI TIPO E CONVERSIONE

`type`

Descrizione: Restituisce il tipo di un valore.

```
echo '123' | jq 'type'
# Output: "number"

echo '"text"' | jq 'type'
# Output: "string"

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

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

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

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

`arrays`, `objects`, `iterables`, `booleans`, `numbers`, `strings`, `nulls`, `values`, `scalars`

Descrizione: Filtri che passano solo valori del tipo specificato.

```
echo '[1, "text", null, true, []]' | jq '.[] | numbers'
# Output: 1

echo '[1, "text", null, true, []]' | jq '.[] | strings'
```

```
# Output: "text"

# values = tutto tranne null
echo '[1, null, "text"]' | jq '.[] | values'
# Output: 1
#         "text"

# scalars = non array/object
echo '[1, [2], {"a":3}]' | jq '.[] | scalars'
# Output: 1
```

---

10. CONDIZIONALI E CONTROLLO

`if-then-else-end`

Descrizione: Condizionale standard.

```
echo '{"x": 10}' | jq 'if .x > 5 then "grande" else "piccolo" end'
# Output: "grande"

# Senza else
echo '{"x": 10}' | jq 'if .x > 5 then "grande" end'
# Output: "grande"

# elif
echo '{"x": 5}' | jq 'if .x > 10 then "grande" elif .x > 5 then "medio" else "piccolo" end'
# Output: "piccolo"
```

`//` (Alternative Operator)

Descrizione: Fornisce valore alternativo se il primo è `null` o `false`.

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

echo '{"x": 10}' | jq '.x // "default"'
# Output: 10

echo '{}' | jq '.x // "default"'
# Output: "default"
```

`try-catch`

Descrizione: Gestione errori.

```
echo '"not a number"' | jq 'try tonumber catch "errore conversione"'
# Output: "errore conversione"

echo '"123"' | jq 'try tonumber catch "errore conversione"'
# Output: 123
```

---

11. VARIABILI E BINDING

`as $variable`

Descrizione: Assegna un valore a una variabile.

```
echo '{"x": 5, "y": 10}' | jq '.x as $saved | .y = .y + $saved'
# Output: {"x": 5, "y": 15}

# Con array
echo '[1,2,3]' | jq '.[] as $item | $item * 2'
# Output: 2
#         4
#         6
```

`$__loc__`

Descrizione: Variabile speciale con info sulla posizione nel codice.

---

12. FUNZIONI AVANZATE

`any` / `any(condition)` / `any(generator; condition)`

Descrizione: Verifica se almeno un elemento soddisfa la condizione.

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

echo '[1,2,3]' | jq 'any(. > 10)'
# Output: false

# Con generatore
echo '{"items":[1,2,3]}' | jq 'any(.items[]; . > 2)'
# Output: true
```

`all` / `all(condition)` / `all(generator; condition)`

Descrizione: Verifica se tutti gli elementi soddisfano la condizione.

```
echo '[2,4,6,8]' | jq 'all(. % 2 == 0)'
# Output: true

echo '[2,3,4]' | jq 'all(. % 2 == 0)'
# Output: false
```

`range(n)` / `range(from; to)` / `range(from; to; step)`

Descrizione: Genera sequenza di numeri.

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

echo 'null' | jq '[range(2; 7)]'
# Output: [2, 3, 4, 5, 6]

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

`until(condition; update)`

Descrizione: Ripete update finché condition è vera.

```
echo '1' | jq 'until(. > 5; . + 1)'
# Output: 6
```

`while(condition; update)`

Descrizione: Ripete update mentre condition è vera.

```
echo '10' | jq 'while(. > 0; . - 1)'
# Output: 0
```

`limit(n; expression)`

Descrizione: Limita il numero di output.

```
echo 'null' | jq '[limit(3; range(10))]'
# Output: [0, 1, 2]
```

`first` / `first(expression)` / `last` / `last(expression)`

Descrizione: Primo/ultimo elemento.

```
echo '[1,2,3,4,5]' | jq 'first'
# Output: 1

echo '[1,2,3,4,5]' | jq 'last'
# Output: 5

# Con espressione
echo '[1,2,3,4,5]' | jq 'first(.[] | select(. > 2))'
# Output: 3
```

`nth(n)` / `nth(n; expression)`

Descrizione: N-esimo elemento (0-based).

```
echo '[10,20,30,40]' | jq 'nth(2)'
# Output: 30

# Negativi dall'ultimo
echo '[10,20,30,40]' | jq 'nth(-1)'
# Output: 40
```

`indices(value)`

Descrizione: Restituisce indici di tutte le occorrenze.

```
echo '[1,2,3,2,1]' | jq 'indices(2)'
# Output: [1, 3]

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

`index(value)` / `rindex(value)`

Descrizione: Primo/ultimo indice di un valore.

```
echo '[1,2,3,2,1]' | jq 'index(2)'
# Output: 1

echo '[1,2,3,2,1]' | jq 'rindex(2)'
# Output: 3
```

`inside(container)`

Descrizione: Verifica se un valore è contenuto in un altro.

```
echo '[2,0]' | jq 'inside([2,1,0])'
# Output: true

echo '"bar"' | jq 'inside("foobar")'
# Output: true
```

---

13. FUNZIONI DI RIDUZIONE

`reduce expression as $var (init; update)`

Descrizione: Riduzione/accumulo con stato.

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

# Concatenazione
echo '["a","b","c"]' | jq 'reduce .[] as $item (""; . + $item)'
# Output: "abc"

# Esempio complesso: conteggio per tipo
echo '[1,"a",2,"b",3]' | jq 'reduce .[] as $item ({}; .[$item | type] += 1)'
# Output: {"number": 3, "string": 2}
```

`foreach expression as $var (init; update; extract)`

Descrizione: Iterazione con stato ed estrazione.

```
echo '[1,2,3]' | jq 'foreach .[] as $item (0; . + $item; .)'
# Output: 1
#         3
#         6
```

---

14. FUNZIONI DI PATH E RECURSIONE

`path(expression)`

Descrizione: Restituisce il percorso di un valore nella struttura.

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

`paths` / `paths(filter)`

Descrizione: Tutti i percorsi nella struttura.

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

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

`getpath(path)`

Descrizione: Ottiene valore da un percorso.

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

`setpath(path; value)`

Descrizione: Imposta valore in un percorso.

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

# Crea struttura se non esiste
echo '{}' | jq 'setpath(["a","b","c"]; 1)'
# Output: {"a":{"b":{"c":1}}}
```

`delpaths(paths)`

Descrizione: Elimina multipli percorsi.

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

`recurse` / `recurse(f)` / `recurse(f; condition)`

Descrizione: Recursione su una struttura.

```
# Tutti i valori ricorsivamente
echo '{"a":1, "b":{"c":2, "d":{"e":3}}}' | jq '.. | scalars'
# Output: 1
#         2
#         3

# Con funzione custom
echo '{"a":{"b":{"c":1}}}' | jq 'recurse(.a?, .b?, .c?)'
```

`..` (Recursive Descent)

Descrizione: Shorthand per `recurse`, visita tutti i valori ricorsivamente.

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

# Trova tutti i "name" annidati
echo '{"person":{"name":"Mario", "child":{"name":"Luigi"}}}' | jq '.. | .name? // empty'
# Output: "Mario"
#         "Luigi"
```

`walk(f)`

Descrizione: Applica funzione ricorsivamente, bottom-up.

```
# Moltiplica tutti i numeri per 2
echo '{"a":1, "b":{"c":2}}' | jq 'walk(if type == "number" then . * 2 else . end)'
# Output: {"a":2, "b":{"c":4}}
```

---

15. FUNZIONI DI I/O E FORMATO

`@base64` / `@base64d`

Descrizione: Encoding/decoding Base64.

```
echo '"Hello World"' | jq '@base64'
# Output: "SGVsbG8gV29ybGQ="

echo '"SGVsbG8gV29ybGQ="' | jq '@base64d'
# Output: "Hello World"
```

`@uri` / `@csv` / `@tsv` / `@html` / `@json` / `@text`

Descrizione: Formattatori per vari output.

```
# URI encoding
echo '"hello world"' | jq '@uri'
# Output: "hello%20world"

# CSV (richiede array)
echo '["a","b","c"]' | jq '@csv'
# Output: "\"a\",\"b\",\"c\""

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

# HTML escaping
echo '"<div>Test</div>"' | jq '@html'
# Output: "&lt;div&gt;Test&lt;/div&gt;"
```

`@sh`

Descrizione: Escaping per shell.

```
echo '"rm -rf /"' | jq '@sh'
# Output: "'rm -rf /'"
```

`format(fmt)`

Descrizione: Formatting con formato specificato.

```
echo '"text"' | jq 'format("html")'
# Output: "text" (escaped per HTML)
```

---

16. FUNZIONI MATEMATICHE

`floor`, `ceil`, `round`

Descrizione: Arrotondamenti.

```
echo '3.7' | jq 'floor'
# Output: 3

echo '3.2' | jq 'ceil'
# Output: 4

echo '3.5' | jq 'round'
# Output: 4
```

`sqrt`

Descrizione: Radice quadrata.

```
echo '16' | jq 'sqrt'
# Output: 4
```

`pow(exp)`

Descrizione: Elevamento a potenza.

```
echo '2' | jq 'pow(3)'
# Output: 8
```

`log`, `log10`, `log2`, `exp`, `exp10`, `exp2`

Descrizione: Funzioni logaritmiche ed esponenziali.

```
echo '10' | jq 'log10'
# Output: 1

echo '8' | jq 'log2'
# Output: 3
```

`fabs`

Descrizione: Valore assoluto.

```
echo '-5.5' | jq 'fabs'
# Output: 5.5
```

`sin`, `cos`, `tan`, `asin`, `acos`, `atan`

Descrizione: Funzioni trigonometriche.

```
echo '0' | jq 'sin'
# Output: 0

echo '3.14159' | jq 'cos'
# Output: -0.9999999999999997 (circa -1)
```

17. FUNZIONI DI DATA E ORA

`now`

Descrizione: Timestamp Unix corrente.

```
echo 'null' | jq 'now'
# Output: 1702468234.123456 (esempio)
```

`fromdateiso8601` / `todateiso8601`

Descrizione: Conversione date ISO8601.

```
echo '"2024-01-15T10:30:00Z"' | jq 'fromdateiso8601'
# Output: 1705316400

echo '1705316400' | jq 'todateiso8601'
# Output: "2024-01-15T10:30:00Z"
```

`fromdate` / `todate`

Descrizione: Conversione date formato standard.

```
echo '"2024-01-15T10:30:00Z"' | jq 'fromdate'
# Output: 1705316400
```

`strftime(fmt)` / `strptime(fmt)`

Descrizione: Formattazione date personalizzata.

```
echo '1705316400' | jq 'strftime("%Y-%m-%d")'
# Output: "2024-01-15"

echo '"2024-01-15"' | jq 'strptime("%Y-%m-%d") | mktime'
# Output: 1705276800
```

`gmtime` / `mktime`

Descrizione: Conversione timestamp ↔ broken-down time.

```
echo '1705316400' | jq 'gmtime'
# Output: [2024,0,15,10,30,0,1,14]
```

---

18. FUNZIONI SPECIALI

`env`

Descrizione: Accesso alle variabili d'ambiente.

```
jq -n 'env.HOME'
# Output: "/home/username"

jq -n 'env.PATH'
# Output: "/usr/local/bin:/usr/bin:/bin"
```

`$ENV`

Descrizione: Oggetto con tutte le variabili d'ambiente.

```
jq -n '$ENV.USER'
# Output: "username"
```

`input` / `inputs`

Descrizione: Legge input aggiuntivi (per processing multi-file).

```
# Combina due file JSON
jq -n '[inputs]' file1.json file2.json
```

`debug` / `debug(prefix)`

Descrizione: Output di debug su stderr.

```
echo '{"x": 5}' | jq '.x | debug | . * 2'
# stderr: ["DEBUG:",5]
# stdout: 10
```

`sql` / `@sql`

Descrizione: Formattazione SQL (non standard, dipende dalla versione).

---

19. OPERATORI SPECIALI

, (Comma)

Descrizione: Produce output multipli indipendenti.

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

; (Semicolon)

Descrizione: Separa espressioni indipendenti (come newline).

```
echo '{"x":1}' | jq '.x = 2; .y = 3'
# Output: {"x": 2, "y": 3}
```

? (Try Operator)

Descrizione: Supprime errori (come try-catch leggero).

```
# Senza ?
echo '{"a":1}' | jq '.b.c'
# Errore: Cannot index null

# Con ?
echo '{"a":1}' | jq '.b.c?'
# Output: null

# Molto utile con array
echo '[{"name":"Mario"}, {}]' | jq '.[].name?'
# Output: "Mario"
#         null
```

---

20. FUNZIONI CUSTOM

`def name: body;` / `def name(params): body;`

Descrizione: Definizione di funzioni custom.

```
# Funzione semplice
echo '5' | jq 'def double: . * 2; double'
# Output: 10

# Con parametri
echo '5' | jq 'def multiply(n): . * n; multiply(3)'
# Output: 15

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

FLAGS COMUNI DELLA COMMAND LINE

-c / --compact-output

Output compatto (una riga).

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

-r / --raw-output

Output raw (senza quote per stringhe).

```
echo '["a", "b", "c"]' | jq -r '.[]'
# Output: a
#         b
#         c
```

-R / --raw-input

Tratta input come stringhe raw (non JSON).

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

-s / --slurp

Legge tutto l'input in un array.

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

-n / --null-input

Non legge input (utile per generare dati).

```
jq -n '{x: 1, y: 2}'
# Output: {"x": 1, "y": 2}
```

-e / --exit-status

Exit code basato su output (0 se true/valori, 1 se false/null, 4 se nessun output).

```
echo '{"x": 5}' | jq -e '.x > 3'
echo $?  # 0 (success)
```

-f / --from-file

Legge programma jq da file.

```
# query.jq contiene: .[] | select(.age > 30)
jq -f query.jq data.json
```

-S / --sort-keys

Ordina chiavi di oggetti in output.

```
echo '{"z":1, "a":2, "m":3}' | jq -S '.'
# Output: {"a":2, "m":3, "z":1}
```

`-C` / `-M` (Color / Monochrome)

Output colorato o monocromatico.

```
jq -C '.' file.json  # colorato
jq -M '.' file.json  # senza colori
```

`--arg name value`

Passa variabile stringa al programma.

```
jq --arg myvar "Hello" '{message: $myvar}' <<< 'null'
# Output: {"message": "Hello"}
```

`--argjson name json`

Passa variabile JSON al programma.

```
jq --argjson myvar '{"x": 10}' '{data: $myvar}' <<< 'null'
# Output: {"data": {"x": 10}}
```

`--slurpfile name file`

Carica JSON da file in variabile.

```
jq --slurpfile data data.json '{imported: $data}' <<< 'null'
```

ESEMPI DI PATTERN COMUNI

1. Filtrare e trasformare

```
echo '[{"name":"Mario","age":30}, {"name":"Luigi","age":25}]' | \
  jq '[.[] | select(.age > 26) | {name, older: true}]'
```

2. Aggregazioni complesse

```
echo '[{"type":"A","val":10}, {"type":"B","val":20}, {"type":"A","val":5}]' | \
  jq 'group_by(.type) | map({type: .[0].type, total: map(.val) | add})'
```

3. Ristrutturazione profonda

```
echo '{"users": [{"id":1,"name":"Mario"}, {"id":2,"name":"Luigi"}]}' | \
  jq '{users: [.users[] | {userId: .id, userName: .name}]}'
```

4. Validazione dati

```
echo '[{"x":5}, {"x":-3}, {"x":10}]' | \
  jq 'map(if .x < 0 then error("Valore negativo trovato") else . end)'
```

5. Merge condizionale

```
echo '{"a":1, "b":2}' | \
  jq '. + if .a > 0 then {"status": "positive"} else {"status": "non-positive"} end'
```

ERRORI COMUNI E SOLUZIONI

1. Iterare vs Costruire Array

```
# ❌ Errato: produce valori separati
.items[]

# ✅ Corretto: produce un array
[.items[]]
```

2. Accesso a campi che potrebbero non esistere

```
# ❌ Errato: genera errore se "b" non esiste
.a.b.c

# ✅ Corretto: usa ?
.a.b?.c?

# ✅ Alternativa: usa // per default
.a.b.c // "default"
```

3. Modificare oggetti

```
# ❌ Errato: non modifica, solo legge
.x = 10

# ✅ Corretto: per modificare serve assegnamento
{x: 10, y: .y}

# ✅ O update in place
. + {x: 10}
```

4. Concatenare stringhe con null

```
# ❌ Errato: errore se un campo è null
.firstName + " " + .lastName

# ✅ Corretto: gestisci null
(.firstName // "") + " " + (.lastName // "")

# ✅ Alternativa
[.firstName, .lastName] | map(. // "") | join(" ")
```
