Package ‘jqr’

May 6, 2021

Title  Client for ‘jq’, a ‘JSON’ Processor

Description  Client for ‘jq’, a ‘JSON’ processor (<https://stedolan.github.io/jq/>), written in C. ‘jq’ allows the following with ‘JSON’ data: index into, parse, do calculations, cut up and filter, change key names and values, perform conditionals and comparisons, and more.

Version  1.2.1

Depends  R (>= 3.1.2)

License  MIT + file LICENSE

Encoding  UTF-8

Language  en-US

LazyData  true

URL  https://docs.ropensci.org/jqr/ (docs),
     https://github.com/ropensci/jqr (devel)

BugReports  https://github.com/ropensci/jqr/issues

SystemRequirements  libjq: jq-devel (rpm) or libjq-dev (deb)

Imports  magrittr, lazyeval

Suggests  jsonlite, testthat

RoxygenNote  7.1.1

NeedsCompilation  yes

Author  Rich FitzJohn [aut],
        Jeroen Ooms [aut],
        Scott Chamberlain [aut, cre],
        Stefan Milton Bache [aut]

Maintainer  Scott Chamberlain <myrmecocystus@gmail.com>

Repository  CRAN

Date/Publication  2021-05-06 14:30:02 UTC
### R topics documented:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>2</td>
</tr>
<tr>
<td>build</td>
<td>3</td>
</tr>
<tr>
<td>combine</td>
<td>4</td>
</tr>
<tr>
<td>commits</td>
<td>5</td>
</tr>
<tr>
<td>dot</td>
<td>5</td>
</tr>
<tr>
<td>funs</td>
<td>6</td>
</tr>
<tr>
<td>index</td>
<td>6</td>
</tr>
<tr>
<td>jq</td>
<td>8</td>
</tr>
<tr>
<td>jqr</td>
<td>9</td>
</tr>
<tr>
<td>jqr_new</td>
<td>10</td>
</tr>
<tr>
<td>jq_flags</td>
<td>11</td>
</tr>
<tr>
<td>keys</td>
<td>12</td>
</tr>
<tr>
<td>logicaltests</td>
<td>13</td>
</tr>
<tr>
<td>manip</td>
<td>14</td>
</tr>
<tr>
<td>maths</td>
<td>17</td>
</tr>
<tr>
<td>paths</td>
<td>19</td>
</tr>
<tr>
<td>peek</td>
<td>20</td>
</tr>
<tr>
<td>rangej</td>
<td>20</td>
</tr>
<tr>
<td>recurse</td>
<td>21</td>
</tr>
<tr>
<td>select</td>
<td>22</td>
</tr>
<tr>
<td>sortj</td>
<td>23</td>
</tr>
<tr>
<td>string</td>
<td>24</td>
</tr>
<tr>
<td>types</td>
<td>24</td>
</tr>
<tr>
<td>vars</td>
<td>25</td>
</tr>
</tbody>
</table>

### Index

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>26</td>
</tr>
</tbody>
</table>

---

## at

**Description**

Format strings and escaping

**Usage**

```r
at(.data, ...)
```

```r
at_.(data, ..., .dots)
```

**Arguments**

- `.data` - input. This can be JSON input, or an object of class `jqr` that has JSON and query params combined, which is passed from function to function when using the `jqr` DSL.
- `...` - Comma separated list of unquoted variable names
Build arrays and objects

```
build
  .dots Used to work around non-standard evaluation
dots dots

Examples
  x <- '="user":"stedolan","titles:["JQ Primer", "More JQ"]"'
  x %>% at(base64) %>% peek
  x %>% at(base64)
  x %>% index() %>% at(base64)

  y <- '["fo", "foo", "barfoo", "foobar", "barfoob"]'
  y %>% index() %>% at(base64)

  # prepare for shell use
  y %>% index() %>% at(sh)

  # rendered as csv with double quotes
  z <- '[1, 2, 3, "a"]'
  z %>% at(csv)

  # rendered as csv with double quotes
  z %>% index()
  z %>% index() %>% at(text)

  # % encode for URI's
  #### DOESNT WORK --------------------------
  # html escape
  #### DOESNT WORK --------------------------
  # serialize to json
  #### DOESNT WORK --------------------------
```

Description

Build arrays and objects

Usage

```
build_array(.data, ...)

build_array_(.data, ..., .dots)

build_object(.data, ...)

build_object_(.data, ..., .dots)
```
Arguments

.data input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.

... Comma separated list of unquoted variable names

.dots Used to work around non-standard evaluation

dots dots

Examples

## BUILD ARRAYS

```r
x <- '{"user":"stedolan", "projects": ["jq", "wikiflow"]}'
jq(x, ".[.user, .projects[]]")
x %>% build_array(.user, .projects[])
```

```
jq('[1, 2, 3]', '[ .[] | . * 2]')
'[[1, 2, 3]] %>% build_array([[] | . * 2]
```

## BUILD OBJECTS

```
'("foo": 5, "bar": 7)' %>% build_object(a = .foo) %>% peek
'("foo": 5, "bar": 7)' %>% build_object(a = .foo)
```

# using json dataset, just first element

```r
x <- commits %>% index(0)
x %>%
  build_object(message = .commit.message, name = .commit.committer.name)
x %>% build_object(sha = .commit.tree.sha, author = .author.login)
```

# using json dataset, all elements

```r
x <- index(commits)
x %>% build_object(message = .commit.message, name = .commit.committer.name)
x %>% build_object(sha = .sha, name = .commit.committer.name)
```

# many JSON inputs

```
'("foo": 5, "bar": 7) {"foo": 50, "bar": 7} {"foo": 500, "bar": 7}' %>%
  build_object(hello = .foo)
```

<table>
<thead>
<tr>
<th></th>
<th>Combine json pieces</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine</td>
<td>Combine json pieces</td>
</tr>
</tbody>
</table>

Description

Combine json pieces

Usage

`combine(x)`
Arguments

x  Input, of class json

Examples

x <- '{"foo": 5, "bar": 7}%' %>% select(a = .foo)
  combine(x)

(x <- commits %>% index() %>%
  select(sha = .sha, name = .commit.committer.name))
  combine(x)

commitst  GitHub Commits Data

Description

GitHub Commits Data

Format

A character string of json github commits data for the jq repo.

dot  dot and related functions

Description

dot and related functions

Usage

dot(.data)

dot_(.data, dots = ".")

dotstr(.data, ...)

dotstr_(.data, ..., .dots)

Arguments

.data  input. This can be JSON input, or an object of class jqr that has JSON and
       query params combined, which is passed from function to function when using
       the jqr DSL.

dots  dots

...  Comma separated list of unquoted variable names

.dots  Used to work around non-standard evaluation
Examples

str <- '[["name":"JSON", "good":true],{"name":"XML", "good":false}]'

str %>% dot
str %>% index %>% dotstr(name)
'("foo": 5, "bar": 8)' %>% dot
'("foo": 5, "bar": 8)' %>% dotstr(foo)
'("foo": {"bar": 8})' %>% dotstr(foo.bar)

funs

Define and use functions

Description
Define and use functions

Usage
funs(.data, fxn, action)

Arguments
.data input
.fxnn A function definition, without def (added internally)
.action What to do with the function on the data

Examples

jq("[1,2,10,20]", 'def increment: . + 1; map(increment)')
"[1,2,10,20]" %>% funs('increment: . + 1', 'map(increment)')
"[1,2,10,20]" %>% funs('increment: . / 100', 'map(increment)')
"[1,2,10,20]" %>% funs('increment: . / 100', 'map(increment)')
'[[1,2],[10,20]]' %>% funs('addvalue(f): f as $x | map(. + $x)', 'addvalue(.[])')
"[1,2]" %>% funs('f(a;b;c;d;e:f): [a+1,b,c,d,e,f]', 'f([0];[1];[0];[0];[0];[0])')
"[1,2,3,4]" %>% funs('fac: if . == 1 then 1 else . * ( . - 1 | fac) end', '[[.]] | fac')

index

Description

index and related functions
Usage

index(.data, ...)

index_(.data, ..., .dots)

indexif(.data, ...)

indexif_(.data, ..., .dots)

dotindex(.data, ...)

dotindex_(.data, ..., .dots)

Arguments

.data input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.

... Comma separated list of unquoted variable names

dots Used to work around non-standard evaluation

dots dots dots

Details

- index/index_- queries like: .[], .[0], .[1:5], .["foo"]
- indexif/indexif_- queries like: .["foo"]?
- dotindex/dotindex_- queries like: .[].foo, .[].foo.bar

Examples

str <- '[["name":"JSON", "good":true}, {"name":"XML", "good":false}]
str %>% index
  \"\"name\"":\"JSON\", \"good\":true\} \% index(name)
  \"\"name\"":\"JSON\", \"good\":true\} \% index(\"good\")
  \"\"name\"":\"JSON\", \"good\":true\} \% index(\"that\")
  \"\"a\": 1, \"b\": 1\} \% index
  [\] \% index
  [[\"\"name\"":\"JSON\", \"good\":true}, {\"name\":\"XML\", \"good\":false}] \% index(0)
  \"\"a\",\"b\",\"c\",\"d\","e\"] \% index(2)
  \"\"a\",\"b\",\"c\",\"d\",\"e\"] \% index(2:4)
  \"\"a\",\"b\",\"c\",\"d\",\"e\"] \% index(2:5)
  \"\"a\",\"b\",\"c\",\"d\",\"e\"] \% index(3)
  \"\"a\",\"b\",\"c\",\"d\",\"e\"] \% index(-2)

str % index % select(bad = .name)

  [[\"\"name\"":\"JSON\", \"good\":true}, {\"name\":\"XML\", \"good\":false}] \% dotindex(name)
  [[\"\"name\"":\"JSON\", \"good\":true}, {\"name\":\"XML\", \"good\":false}] \%
jq

Execute a query with jq

Description

jq is meant to work with the high level interface in this package. jq also provides access to the low level interface in which you can use jq query strings just as you would on the command line. Output gets class of json, and pretty prints to the console for easier viewing. jqr doesn’t do pretty printing.

Usage

```r
jq(x, ...) 
## S3 method for class 'jqr'
jq(x, ...) 
## S3 method for class 'character'
jq(x, ..., flags = jq_flags()) 
## S3 method for class 'json'
jq(x, ..., flags = jq_flags()) 
## S3 method for class 'connection'
jq(x, ..., flags = jq_flags(), out = NULL)
```

Arguments

- `x`: json object or character string with json data. this can be one or more valid json objects
- `...`: character specification of jq query. Each element in code... will be combined with " | ", which is convenient for long queries.
- `flags`: See jq_flags
- `out`: a filename, callback function, connection object to stream output. Set to ‘NULL’ to buffer all output and return a character vector.

See Also

- peek
Examples

```r
'{"a": 7}' %>% do(.a + 1)
'[8,3,null,6]' %>% sortj

x <- '[{"message": "hello", "name": "jenn"},
{"message": "world", "name": "beth"}]
jq(index(x))

jq('{"a": 7, "b": 4}', 'keys')
jq('[8,3,null,6]', 'sort')

# many json inputs
jq(c('[123, 456]', '[77, 88, 99]', '[41]'), '.[]')
# Stream from connection
tmp <- tempfile()
writeLines(c('[123, 456]', '[77, 88, 99]', '[41]'), tmp)
jq(file(tmp), '.[]')

## Not run:
# from a url
x <- 'http://jeroen.github.io/data/diamonds.json'
jq(url(x), '.[]')

# from a file
file <- file.path(tempdir(), "diamonds.nd.json")
download.file(x, destfile = file)
jq(file(file), ".carat")
jq(file(file), "select(.carat > 1.5)")
jq(file(file), "select(.carat > 4 and .cut == "Fair")")

## End(Not run)
```

Description

An R client for the C library jq

Low-level

Low level interface, in which you can execute ‘jq’ code just as you would on the command line. Available via `jq`

High-level DSL

High-level, uses a suite of functions to construct queries. Queries are constructed, then executed internally with `jq`
Pipes
The high level DSL supports piping, though you don’t have to use pipes.

NSE and SE
Most DSL functions have NSE (non-standard evaluation) and SE (standard evaluation) versions, which make jqr easy to use for interactive use as well as programming.

jq version
We link to jq through the installed version on your system, so the version can vary. Run jq --version to get your jq version

indexing
note that jq indexing starts at 0, whereas R indexing starts at 1. So when you want the first thing in an array using jq, for example, you want 0, not 1

output data format
Note that with both the low level interface and the high level DSL, we print the output to look like a valid JSON object to make it easier to look at. However, it’s important to know that the output is really just a simple character string or vector of strings - it’s just the print function that pretty prints it and makes it look like a single JSON object. What jq is giving you often is a stream of valid JSON objects, each one of which is valid, but altogether are not valid. However, a trick you can do is to wrap your jq program in brackets like [.[]] instead of .[] to give a single JSON object

Related to above, you can use the function provided string with the high level DSL to get back a character string instead of pretty printed version

---

**jqr_new**

**JQ Streaming API**

Description
Low level JQ API. First create a program using a 'query' and 'flags' and then feed pieces of data.

Usage

```
jqr_new(query, flags = jq_flags())
```

```
jqr_feed(jqr_program, json, unlist = TRUE, finalize = FALSE)
```
Arguments

- **query**: string with a valid jq program
- **flags**: See ❄_jq_flags_❄
- **jq_program**: object returned by \([\text{jqr}\_\text{new}]\)
- **json**: character vector with json data. If the JSON object is incomplete, you must set `finalize` to `FALSE` otherwise you get an error.
- **unlist**: if `TRUE` returns a single character vector with all output for each string in `json` input
- **finalize**: completes the parsing and verifies that the JSON string is valid. Set this to `TRUE` when feeding the final piece of data.

Examples

```r
program <- jqr_new(".[]")
jqr_feed(program, c("[123, 456]", "[77, 88, 99]"))
jqr_feed(program, c("[41, 234]"))
jqr_feed(program, "", finalize = TRUE)
```

jq_flags

Flags for use with jq

Description

The `flags` function is provided for the high-level DSL approach, whereas the `jq_flags` function is used to provide the low-level `jq` with the appropriate flags.

Usage

```r
jq_flags(
  pretty = FALSE,
  ascii = FALSE,
  color = FALSE,
  sorted = FALSE,
  stream = FALSE,
  seq = FALSE
)
```

```r
flags(
  .data,
  pretty = FALSE,
  ascii = FALSE,
  color = FALSE,
  sorted = FALSE,
  stream = FALSE,
  seq = FALSE
)
```
keys

Arguments

pretty  Pretty print the json (different to jsonlite’s pretty printing).
ascii   Force jq to produce pure ASCII output with non-ASCII characters replaced by equivalent escape sequences.
color   Add ANSI escape sequences for coloured output
sorted  Output fields of each object with keys in sorted order
stream  Parse the input in streaming fashion, outputing arrays of path and leaf values like jq --stream command line.
seq     Use the application/json-seq MIME type scheme for separating JSON like the jq --seq command line.
.data   A jqr object.

Examples

```javascript
'("a": 7, "z":0, "b": 4)'
jq("{"a": 7, "z":0, "b": 4}"), "." flags = jq_flags(sorted = TRUE)
```

Description

keys takes no input, and retrieves keys. del deletes provided keys. haskey checks if a json string has a key, or the input array has an element at the given index.

Usage

keys(.data)

del(.data, ...)

del_(.data, ..., .dots)

haskey(.data, ...)

haskey_(.data, ..., .dots)

Arguments

.data  input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.
...    Comma separated list of unquoted variable names
.dots  Used to work around non-standard evaluation
dots   dots
Examples

```r
# get keys
str <- '("foo": 5, "bar": 7)'
jq(str, "keys")
str %>% keys()

# delete by key name
jq(str, "del(.bar)")
str %>% del(bar)

# check for key existence
str3 <- '[[0,1], ["a","b","c"]]
jq(str3, "map(has(2))")
str3 %>% haskey(2)
jq(str3, "map(has(1,2))")
str3 %>% haskey(1,2)
```

```r
## many JSON inputs
'("foo": 5, "bar": 7) "hello": 5, "world": 7)' %>% keys
'("foo": 5, "bar": 7) "hello": 5, "bar": 7)' %>% del(bar)
```

---

### Logical tests

#### Description

Logical tests

#### Usage

- `allj(.data)`

- `anyj(.data)`

#### Arguments

- `.data` - input. This can be JSON input, or an object of class `jqr` that has JSON and query params combined, which is passed from function to function when using the `jqr` DSL.

#### Examples

```r
# any
'[true, false]' %>% anyj
'[false, false]' %>% anyj
'[]' %>% anyj

# all
'[true, false]' %>% allj
```
## many JSON inputs

`'[true, false] [true, true] [false, false]' %>% anyj
'[/true, false] [true, true] [false, false]' %>% allj

### Description

Manipulation operations

### Usage

*join*(*.data, ...)*

*join_*(.*data, ..., .dots)*

*splitj*(*.data, ...)*

*splitj_*(.*data, ..., .dots)*

*ltrimstr*(*.data, ...)*

*ltrimstr_*(.*data, ..., .dots)*

*rtrimstr*(*.data, ...)*

*rtrimstr_*(.*data, ..., .dots)*

*startswith*(*.data, ...)*

*startswith_*(.*data, ..., .dots)*

*endswith*(*.data, ...)*

*endswith_*(.*data, ..., .dots)*

*index_loc*(*.data, ...)*

*index_loc_*(.*data, ..., .dots)*

*rindex_loc*(*.data, ...)*

*rindex_loc_*(.*data, ..., .dots)*

rindex_loc_(*.data, ..., .dots)
indices(.data, ...)  
indices_(.data, ..., .dots)  
tojson(.data)  
fromjson(.data)  
tostring(.data)  
tonumber(.data)  
contains(.data, ...)  
contains_(.data, ..., .dots)  
uniquej(.data, ...)  
uniquej_(.data, ..., .dots)  
group(.data, ...)  
group_(.data, ..., .dots)

Arguments

.data  
input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.

...  
Comma separated list of unquoted variable names

dots  
Used to work around non-standard evaluation

dots

See Also

add

Examples

# join
str <- ['"a","b,c,d","e"']
jq(str, 'join(" ", )')
str %>% join
str %>% join(';')
str %>% join('yep')
## many JSON inputs
['"a","b,c,d","e"'] ['"a","f,e,f"] %>% join('
---
')
# split
jq("'a, b, c, d, e'", 'split(" ")')

# ltrimstr
jq(['"fo", "foo", "barfoo", "foobar", "afoo"'], ['.'[.]ltrimstr("foo")'])

# rtrimstr
jq(['"fo", "foo", "barfoo", "foob"'], ['.'[.]rtrimstr("foo")'])

# startswith
str <- ['"fo", "foo", "barfoo", "foob"']
jq(str, ['.'[.]startswith("foo")'])

# endswith
jq(str, ['.'[.]endswith("foo")'])
str %>% index %>% endswith(foo)
str %>% index %>% endswith("foo")
str %>% index %>% endswith("bar")

# unique
'[, 1, 2, 5, 3, 5, 3, 1, 3]' %>% uniquej
str <- ['["foo": 1, "bar": 2, {"foo": 1, "bar": 3, {"foo": 4, "bar": 5}]}']
```r
str %>% uniquej(foo)
str %>% uniquej("foo")
'[,"chunky", "bacon", "kitten", "cicada", "asparagus"]' %>% uniquej(length)

# group
x <- 
'[,"foo":1, "bar":10], 
[,"foo":3, "bar":100], 
[,"foo":1, "bar":1]'

x %>% group(foo)
x %>% group_("foo")
```
Examples

## do math
jq('"a": 7', '.a + 1')
# adding null gives back same result
jq('"a": 7', '.a + null')
jq('"a": 7', '.a += 1')
'("a": 7) %>% do(.a + 1)
# '{"a": 7}' %>% do(.a += 1) # this doesn't work quite yet
'("a": [1,2], "b": [3,4])' %>% do(.a + .b)
'("a": [1,2], "b": [3,4])' %>% do(.a -= .b)
'("a": 3)' %>% do(4 - .a)
'"xml", "yaml", "json"' %>% do(\'. - ["xml", "yaml"]\')
'5' %>% do(10 / \.* 3)
## many JSON inputs
'{"a": [1,2], "b": [3,4]} {'a": [1,5], "b": [3,10]}' %>% do(.a + .b)

## comparisons
'[5,4,2,7]' %>% index() %>% do(. < 4)
'[5,4,2,7]' %>% index() %>% do(. > 4)
'[5,4,2,7]' %>% index() %>% do(. <= 4)
'[5,4,2,7]' %>% index() %>% do(. >= 4)
'[5,4,2,7]' %>% index() %>% do(. != 4)
## many JSON inputs
'[5,4,2,7] [4,3,200,0.1]' %>% index() %>% do(. < 4)

## length
[1,2], "string", {"a":2}, null] %>% index %>% lengthj

## sqrt
'9' %>% sqrtj
## many JSON inputs
'9 4 5' %>% sqrtj

## floor
'3.14159' %>% floorj
## many JSON inputs
'3.14159 30.14 45.9' %>% floorj

## find minimum
'[5,4,2,7]' %>% minj
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% minj
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% minj(foo)
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% minj(bar)
## many JSON inputs
'[{"foo":1}, {"foo":14}] [{"foo":2}, {"foo":3}]' %>% minj(foo)

## find maximum
paths

Outputs paths to all the elements in its input

Description

Outputs paths to all the elements in its input

Usage

paths(.data)

Arguments

.data input
Examples

```
'[[1,[[{{"a":2}}]]]]' %>% paths
'[[{"name":"JSON","good":true}, {"name":"XML","good":false}]]' %>% paths
```

peek  
** Peek at a query 

Description

Prints the query resulting from jq all in one character string just as you would execute it on the command line. Output gets class of json, and pretty prints to the console for easier viewing.

Usage

```
peek(.data)
```

Arguments

- **.data**  
  (list) input, using higher level interface

See Also

jq.

Examples

```
'{{"a": 7}}' %>% do(.a + 1) %>% peek
'[[8,3,null,6]]' %>% sortj %>% peek
```

rangej  
** Produce range of numbers 

Description

Produce range of numbers

Usage

```
rangej(x, array = FALSE)
```

Arguments

- **x**  
  Input, single number or number range.
- **array**  
  (logical) Create array. Default: FALSE
Examples

2:4 %>% rangej
2:1000 %>% rangej
1 %>% rangej
4 %>% rangej

recurse
Search through a recursive structure - extract data from all levels

Description
Search through a recursive structure - extract data from all levels

Usage
recurse(.data, ...)
recurse_(.data, ..., .dots)

Arguments
.data input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.
... Comma separated list of unquoted variable names
.dots Used to work around non-standard evaluation
dots dots

Examples
x <- '{"name": "/", "children": [
  {"name": "/bin", "children": [
    {"name": "/bin/ls", "children": []},
    {"name": "/bin/sh", "children": []}],
  {"name": "/home", "children": [
    {"name": "/home/stephen", "children": [
      {"name": "/home/stephen/jq", "children": []}]]}]})'
x %>% recurse(.children[]) %>% build_object(name)
x %>% recurse(.children[]) %>% build_object(name) %>% string
select  Select - filtering

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

Usage
```r
select(.data, ...)
```
```r
select_(.data, ..., .dots)
```

Arguments
- `.data` input. This can be JSON input, or an object of class `jqr` that has JSON and query params combined, which is passed from function to function when using the `jqr` DSL.
- `...` Comma separated list of unquoted variable names
- `.dots` Used to work around non-standard evaluation
- `dots` dots

Note
this function has changed what it does dramatically. we were using this function for object construction, which is now done with `build_object`

Examples
```r
jq('[1,5,3,0,7]', 'map(select(. > = 2))')
```
```r
jq('[1,5,3,0,7]' %>% map(select(. > = 2))

'("foo": 4, "bar": 7)' %>% select(.foo == 4)
'("foo": 5, "bar": 7) {"foo": 4, "bar": 7}' %>% select(.foo == 4)
'[['"foo": 5, "bar": 7], {"foo": 4, "bar": 7}]]' %>% index() %>%
  select(.foo == 4)
'("foo": 4, "bar": 7) {"foo": 5, "bar": 7} {"foo": 8, "bar": 7}' %>%
  select(.foo < 6)

x <- '{"foo": 4, "bar": 2} {"foo": 5, "bar": 4} {"foo": 8, "bar": 12}'
x <- jq(x, 'select((.foo < 6) and (.bar > 3))')
x <- jq(x, 'select((.foo < 6) or (.bar > 3))')
x <- x %>% select((.foo < 6) && (.bar > 3))
x <- x %>% select((.foo < 6) || (.bar > 3))

x <- '[["foo": 5, "bar": 7], {"foo": 4, "bar": 7}, {"foo": 4, "bar": 9}]'
```
sortj

\[
\text{jq(x, \{[] | select(.foo == 4) | (user: .bar)\}')}
\]
\[
x \%% \text{index()} \%% \text{select(.foo == 4)} \%% \text{build_object(user = .bar)}
\]

---

sortj  

Sort and related

Description

Sort and related

Usage

\[
\text{sortj(.data, ...)}
\]
\[
\text{sortj(.data, ..., .dots)}
\]
\[
\text{reverse(.data)}
\]

Arguments

.data  
  input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.

...  
  Comma separated list of unquoted variable names

dots  
  Used to work around non-standard evaluation

dots

Examples

# sort
\['[8,3,null,6]' \%% sortj
\['[\{\text{"foo":4, \"bar":10}, \{\text{"foo":3, \"bar":100}, \{\text{"foo":2, \"bar":1}\}'] \%% sortj(\text{foo})

# reverse order
\['[1,2,3,4]' \%% reverse

# many JSON inputs
\['[\{\text{"foo":7}, \{\text{"foo":4}\}], [\{\text{"foo":300}, \{\text{"foo":1}\}], [\{\text{"foo":2}, \{\text{"foo":1}\}'] \%% sortj(\text{foo})

\['[1,2,3,4] [10,20,30,40] [100,200,300,4000]' \%% reverse
string

Give back a character string

Description

Give back a character string

Usage

\texttt{string(.data)}

Arguments

\begin{itemize}
\item \texttt{.data} \hspace{1cm} (list) input, using higher level interface
\end{itemize}

See Also

\texttt{peek}

Examples

\begin{verbatim}
'("a": 7)' %>% do(.a + 1) %>% string
'[8,3,null,6]' %>% sortj %>% string
\end{verbatim}

types

Types and related functions

Description

Types and related functions

Usage

\texttt{types(.data)}

\texttt{type(.data, ...)}

\texttt{type_(.data, ..., .dots)}

Arguments

\begin{itemize}
\item \texttt{.data} \hspace{1cm} input. This can be JSON input, or an object of class \texttt{jqr} that has JSON and query params combined, which is passed from function to function when using the \texttt{jqr} DSL.
\item \texttt{...} \hspace{1cm} Comma separated list of unquoted variable names
\item \texttt{.dots} \hspace{1cm} Used to work around non-standard evaluation
\item \texttt{dots} \hspace{1cm} dots
\end{itemize}
Examples

# get type information for each element
jq('[0, false, [], {}, null, "hello"], map(type)
'[[0, false, [], {}, null, "hello"]]
'[[0, false, [], {}, null, "hello", true, [1,2,3]]]

# select elements by type
jq('[0, false, [], {}, null, "hello"], [.[] | numbers,booleans')
'[[0, false, [], {}, null, "hello"]]

vars

Variables

Description

Variables

Usage

vars(.data, ...)
vars_(.data, ..., .dots)

Arguments

.data    input. This can be JSON input, or an object of class jqr that has JSON and query params combined, which is passed from function to function when using the jqr DSL.
...      Comma separated list of unquoted variable names
.dots    Used to work around non-standard evaluation
dots

Examples

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

x %>% dotstr(posts[])%>% string
x %>% dotstr(posts[]) %>% string
x %>% vars(realnames = names) %>% dotstr(posts[]) %>%
    build_object(title, author = "$names[.author]")
Index

* datasets
  commits, 5
  ad (maths), 17
  add, 15
  allj (logicaltests), 13
  anyj (logicaltests), 13
  at, 2
  at_ (at), 2
  build, 3
  build_array (build), 3
  build_array_ (build), 3
  build_object, 22
  build_object_ (build), 3
  combine, 4
  commits, 5
  contains (manip), 14
  contains_ (manip), 14
  del (keys), 12
  del_ (keys), 12
  do (maths), 17
  do_ (maths), 17
  dot, 5
  dot_ (dot), 5
  dotindex (index), 6
  dotindex_ (index), 6
  dotstr (dot), 5
  dotstr_ (dot), 5
  endswith (manip), 14
  endswith_ (manip), 14
  flags (jq_flags), 11
  floorj (maths), 17
  fromjson (manip), 14
  funs, 6
  group (manip), 14
  group_ (manip), 14
  haskey (keys), 12
  haskey_ (keys), 12
  index, 6
  index_ (index), 6
  index_loc (manip), 14
  index_loc_ (manip), 14
  indexif (index), 6
  indexif_ (index), 6
  indices (manip), 14
  indices_ (manip), 14
  join (manip), 14
  join_ (manip), 14
  jq, 8, 9, 20
  jq_flags, 8, 11, 11
  jqr, 9
  jqr-package (jqr), 9
  jqr_feed (jqr_new), 10
  jqr_new, 10
  keys, 12
  lengthj (maths), 17
  logicaltests, 13
  ltrimstr (manip), 14
  ltrimstr_ (manip), 14
  manip, 14
  map (maths), 17
  map_ (maths), 17
  maths, 17
  maxj (maths), 17
  maxj_ (maths), 17
  minj (maths), 17
  minj_ (maths), 17
  paths, 19

26
peek, 8, 20, 24
rangej, 20
recurr, 21
recurr_(recurr), 21
reverse (sortj), 23
rindex_loc (manip), 14
rindex_loc_ (manip), 14
rtrimstr (manip), 14
rtrimstr_ (manip), 14
select, 22
select_ (select), 22
sortj, 23
sortj_ (sortj), 23
splitj (manip), 14
splitj_ (manip), 14
sqrtj (maths), 17
startswith (manip), 14
startswith_ (manip), 14
string, 10, 24
tojson (manip), 14
tonumber (manip), 14
tostring (manip), 14
type (types), 24
type_ (types), 24
types, 24
uniquej (manip), 14
uniquej_ (manip), 14
vars, 25
vars_ (vars), 25