

Package ‘jq’

August 29, 2016

Title Client for 'jq', a JSON Processor

Description Client for 'jq', a JSON processor (<<http://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 0.2.4

Depends R (>= 3.1.2)

License MIT + file LICENSE

LazyData true

VignetteBuilder knitr

URL <https://github.com/ropensci/jqr>

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

LinkingTo BH, Rcpp

Imports Rcpp, lazyeval, magrittr

Suggests jsonlite, testthat, knitr, covr

RoxygenNote 5.0.1

NeedsCompilation yes

Author Rich FitzJohn [aut],
Jeroen Ooms [aut],
Scott Chamberlain [aut, cre],
Stefan Milton Bache [aut],
Stephen Dolan [aut, cph] (jq library),
jq contributors [ctb] (jq library; authors listed in inst/AUTHORS.jq),
Free Software Foundation [ctb, cph] (parser code in src/parser.c),
David M Gay [ctb, cph] (code in src/jv_dtoa.c)

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2016-07-29 06:41:07

R topics documented:

at	2
combine	3
dot	4
funs	5
githubcommits	5
index	6
jq	7
jqr	8
jq_flags	9
keys	9
logicaltests	10
manip	11
maths	14
paths	16
peek	17
rangej	17
recurse	18
select	18
sortj	19
string	20
types	20
vars	21
Index	23

at	<i>Format strings and escaping</i>
----	------------------------------------

Description

Format strings and escaping

Usage

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

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

Arguments

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

 combine

Combine json pieces

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

 dot

dot and related functions

Description

dot and related functions

Usage

```
dot(.data)

dot_(.data, dots = ".")

dotstr(.data, ...)

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

Arguments

<code>.data</code>	input. This can be JSON input, or an object of class <code>jqr</code> that has JSON and query params combined, which is passed from function to function when using the <code>jqr</code> DSL.
<code>dots</code>	dots
<code>...</code>	Comma separated list of unquoted variable names
<code>.dots</code>	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
fxn	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(.[0])')
"[1,2]" %>% funs('f(a;b;c;d;e;f): [a+1,b,c,d,e,f]', 'f(.[0];.[1];.[0];.[0];.[0])')
"[1,2,3,4]" %>% funs('fac: if . == 1 then 1 else . * (. - 1 | fac) end', '.[.] | fac')
```

githubcommits

*GitHub Commits Data***Description**

GitHub Commits Data

Format

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

index	<i>index and related functions</i>
-------	------------------------------------

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 <code>jqr</code> that has JSON and query params combined, which is passed from function to function when using the <code>jqr</code> DSL.
...	Comma separated list of unquoted variable names
.dots	Used to work around non-standard evaluation
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
str %>% indexif
str %>% indexif(name)
'{"name": "JSON", "good": true}' %>% indexif(name)
'{"name": "JSON", "good": true}' %>% indexif(good)
'{"name": "JSON", "good": true}' %>% indexif(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}]' %>% dotindex(good)
'[{"name":"JSON", "good":{"foo":5}}, {"name":"XML", "good":{"foo":6}}]' %>% dotindex(good)
'[{"name":"JSON", "good":{"foo":5}}, {"name":"XML", "good":{"foo":6}}]' %>% dotindex(good.foo)

```

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. jq doesn't do pretty printing.

Usage

```

jq(x, ...)

## S3 method for class 'jqr'
jq(x, ...)

## S3 method for class 'character'
jq(x, ..., flags = jq_flags())

```

Arguments

x	json object or character string with json data.
...	character specification of jq query. Each element in code... will be combined with " ", which is convenient for long queries.
flags	See jq_flags

See Also

[peek](#)

Examples

```
'{"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')
```

jqr

jqr: An R client for the C library jq

Description

jqr: 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

jqr is currently using jq v1.4

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

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

```
jq_flags(pretty = FALSE, ascii = FALSE, color = FALSE, sorted = FALSE)
```

```
flags(.data, pretty = FALSE, ascii = FALSE, color = FALSE,
      sorted = FALSE)
```

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
.data	A jq object.

Examples

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

keys

*Operations on keys, or by keys***Description**

keys takes no input, and retrieves keys. del deletes provided keys. haskey checks if a json string has a key or keys.

Usage

```
keys(.data)

del(.data, ...)

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

haskey(.data, ...)

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

Arguments

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

Examples

```
# 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)
```

logicaltests

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

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

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

manip

Manipulation operations

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)  
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

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

See Also[add](#)**Examples**

```

# join
str <- '["a","b,c,d","e"]'
jq(str, 'join(", ")')
str %>% join
str %>% join(`;`)
str %>% join(`yep`)

# split
jq('a, b,c,d, e', 'split(", ")')

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

# rtrimstr
jq('["fo", "foo", "barfoo", "foobar", "foob"]', '[".|"]|rtrimstr("foo")')
'["fo", "foo", "barfoo", "foobar", "foob"]' %>% index() %>% rtrimstr(foo)

# startswith
str <- '["fo", "foo", "barfoo", "foobar", "barfoob"]'
jq(str, '[".|"]|startswith("foo")')
str %>% index %>% startswith(foo)

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

# get index (location) of a character
## input has to be quoted
str <- '"a,b, cd, efg, hijk"'
str %>% index_loc(", ")
str %>% index_loc(",")
str %>% index_loc("j")
str %>% rindex_loc(", ")
str %>% indices(", ")

# tojson, fromjson, toString, tonumber
'[1, "foo", ["foo"]]' %>% index %>% toString
'[1, "1"]' %>% index %>% tonumber
'[1, "foo", ["foo"]]' %>% index %>% tojson
'[1, "foo", ["foo"]]' %>% index %>% tojson %>% fromjson

# contains
'"foobar"' %>% contains("bar")

```

```
'["foobar", "foobaz", "blarp"]' %>% contains(`["baz", "bar"]`)
'["foobar", "foobaz", "blarp"]' %>% contains(`["bazzzzz", "bar"]`)
str <- '{"foo": 12, "bar": [1,2,{"barp":12, "blip":13}]}'
str %>% contains(`{foo: 12, bar: [{barp: 12}]}`)
str %>% contains(`{foo: 12, bar: [{barp: 15}]}`)

# unique
'[1,2,5,3,5,3,1,3]' %>% uniquej
str <- '{"foo": 1, "bar": 2}, {"foo": 1, "bar": 3}, {"foo": 4, "bar": 5}'
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")
```

maths

Math operations

Description

Math operations

Usage

`do(.data, ...)`

`do_(.data, ..., .dots)`

`lengthj(.data)`

`sqrtj(.data)`

`floorj(.data)`

`minj(.data, ...)`

`minj_(.data, ..., .dots)`

`maxj(.data, ...)`

`maxj_(.data, ..., .dots)`

`ad(.data)`

`map(.data, ...)`

```
map_(.data, ..., .dots)
```

Arguments

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

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)

# 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)
'[5,4,2,7]' %>% index() %>% do(. != 4)

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

# sqrt
'9' %>% sqrtj

# floor
'3.14159' %>% 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)

# find maximum
```

```

'[5,4,2,7]' %>% maxj
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% maxj
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% maxj(foo)
'[{"foo":1, "bar":14}, {"foo":2, "bar":3}]' %>% maxj(bar)

# increment values
## requires special % operators, they get escaped internally
'{"foo": 1}' %>% do(.foo %+=% 1)
'{"foo": 1}' %>% do(.foo %-= % 1)
'{"foo": 1}' %>% do(.foo %*=% 4)
'{"foo": 1}' %>% do(.foo %/= % 10)
'{"foo": 1}' %>% do(.foo %/=% 10)
### fix me - %= doesn't work
# '{"foo": 1}' %>% do(.foo %%= % 10)

# add
'["a","b","c"]' %>% ad
'[1, 2, 3]' %>% ad
'[]' %>% ad

# map
## as far as I know, this only works with numbers, thus it's
## in the maths section
'[1, 2, 3]' %>% map(+1)
'[1, 2, 3]' %>% map(/1)
'[1, 2, 3]' %>% map(*4)

```

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	<i>Peek at a query</i>
------	------------------------

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	<i>Produce range of numbers</i>
--------	---------------------------------

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

```
rangej(2:4)
2:4 %>% rangej
2:1000 %>% rangej
# errors, correctly though
# 2 %>% rangej
```

recurse	<i>Search through a recursive structure - extract data from all levels</i>
---------	----------------------------------------------------------------------------

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 <code>jqr</code> that has JSON and query params combined, which is passed from function to function when using the <code>jqr</code> 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[]) %>% select(name)
x %>% recurse(.children[]) %>% select(name) %>% string
```

select	<i>Select variables</i>
--------	-------------------------

Description

Select variables

Usage

```
select(.data, ...)
```

```
select_(.data, ..., .dots)
```

Arguments

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

Examples

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

# using json dataset, just first element
x <- githubcommits %>% index(0)
x %>%
  select(message = .commit.message, name = .commit.committer.name)
x %>% select(sha = .commit.tree.sha, author = .author.login)

# using json dataset, all elements
x <- index(githubcommits)
x %>% select(message = .commit.message, name = .commit.committer.name)
x %>% select(sha = .sha, name = .commit.committer.name)
```

 sortj

Sort and related

Description

Sort and related

Usage`sortj(.data, ...)``sortj_(.data, ..., .dots)``reverse(.data)`**Arguments**

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

Examples

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

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

string	<i>Give back a character string</i>
--------	-------------------------------------

Description

Give back a character string

Usage

```
string(.data)
```

Arguments

.data (list) input, using higher level interface

See Also

[peek](#)

Examples

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

types	<i>Types and related functions</i>
-------	------------------------------------

Description

Types and related functions

Usage

```
types(.data)

type(.data, ...)

type_(.data, ..., .dots)
```

Arguments

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

Examples

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

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

vars

Variables

Description

Variables

Usage

```
vars(.data, ...)
```

```
vars_(.data, ..., .dots)
```

Arguments

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

Examples

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

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

Index

*Topic **datasets**

githubcommits, 5

ad (maths), 14

add, 13

allj (logicaltests), 10

anyj (logicaltests), 10

at, 2

at_ (at), 2

combine, 3

contains (manip), 11

contains_ (manip), 11

del (keys), 9

del_ (keys), 9

do (maths), 14

do_ (maths), 14

dot, 4

dot_ (dot), 4

dotindex (index), 6

dotindex_ (index), 6

dotstr (dot), 4

dotstr_ (dot), 4

endswith (manip), 11

endswith_ (manip), 11

flags (jq_flags), 9

floorj (maths), 14

fromjson (manip), 11

funs, 5

githubcommits, 5

group (manip), 11

group_ (manip), 11

haskey (keys), 9

haskey_ (keys), 9

index, 6

index_ (index), 6

index_loc (manip), 11

index_loc_ (manip), 11

indexif (index), 6

indexif_ (index), 6

indices (manip), 11

indices_ (manip), 11

join (manip), 11

join_ (manip), 11

jq, 7, 8, 17

jq_flags, 7, 9

jq_r, 8

jq_r-package (jq_r), 8

keys, 9

lengthj (maths), 14

logicaltests, 10

ltrimstr (manip), 11

ltrimstr_ (manip), 11

manip, 11

map (maths), 14

map_ (maths), 14

maths, 14

maxj (maths), 14

maxj_ (maths), 14

minj (maths), 14

minj_ (maths), 14

paths, 16

peek, 7, 17, 20

rangej, 17

recurse, 18

recurse_ (recurse), 18

reverse (sortj), 19

rindex_loc (manip), 11

rindex_loc_ (manip), 11

rtrimstr (manip), 11

rtrimstr_(manip), 11

select, 18
select_(select), 18
sortj, 19
sortj_(sortj), 19
splitj (manip), 11
splitj_(manip), 11
sqrtj (maths), 14
startswith (manip), 11
startswith_(manip), 11
string, 20

tojson (manip), 11
tonumber (manip), 11
tostring (manip), 11
type (types), 20
type_ (types), 20
types, 20

uniquej (manip), 11
uniquej_(manip), 11

vars, 21
vars_ (vars), 21