# Package 'jsontools'

October 13, 2022

**Title** Working with JSON Vectors

**Version** 0.1.0

**Description** A toolbox for working with JSON vectors similar to the functions 'Postgres' provides to work with JSON columns. It supports in parsing and formatting JSON, extracting data from JSON, and modifying JSON data.

**License** GPL-3

**Depends** R (>= 3.2.0)

**Imports** bit64, DBI, glue, jsonlite, magrittr, methods, pillar, purrr, rlang, RSQLite, tibble, tidyselect, vctrs, withr

**Suggests** curl, dplyr, knitr, repurrrsive, rmarkdown, testthat (>= 2.1.0)

**VignetteBuilder** knitr

**Config/testthat/edition** 3

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.1.1

**NeedsCompilation** no

**Author** Maximilian Girlich [aut, cre]

**Maintainer** Maximilian Girlich <maximilian.girlich@outlook.com>

**Repository** CRAN

**Date/Publication** 2021-03-22 09:40:09 UTC

## R topics documented:

---

`format_json` *Convert* R *objects to JSON*

---

## Description

`format_json` is only a wrapper around the great `jsonlite::toJSON()`. The differences are

- expose argument `json_verbatim`, `rownames`, and `always_decimal`.
- use `json_verbatim = TRUE` by default so that JSON isn't escaped again.
- return a `json2` object.

`format_json_list()` converts each element of a list to JSON.

To make sure that a length one vector is not turned into an array use `json_u()` or `jsonlite::unbox()`.

## Usage

```
format_json(
  x,
  null = c("list", "null"),
  na = c("null", "string"),
  auto_unbox = FALSE,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "integer"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  digits = 4,
  json_verbatim = TRUE,
  force = FALSE,
  pretty = FALSE,
  rownames = FALSE,
  always_decimal = FALSE,
  ...
)

format_json_list(
  x,
  null = c("list", "null"),
  na = c("null", "string"),
  auto_unbox = FALSE,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "integer"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  digits = 4,
  json_verbatim = TRUE,
  force = FALSE,
  pretty = FALSE,
  rownames = FALSE,
  always_decimal = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | the object to be encoded |
| `null`, `na`, `auto_unbox`, `dataframe`, `matrix`, `Date` | passed on to `jsonlite::toJSON`. |

| | |
|---|---|
| POSIXt, factor, complex, raw, digits, force, pretty, ... | passed on to jsonlite::toJSON. |
| json_verbatim | Leave JSON as it is and do not encode it again? |
| rownames | For data.frames add a field _row with the row name? |
| always_decimal | Use real number notation in whole number doubles? |

## Value

A json2 vector.

## See Also

write_json(), format_json_rowwise()

## Examples

```
# null
x_null <- list(a = NULL, b = 1)
format_json(x_null)
format_json(x_null, null = "null")

# na
x_na <- list(a = NA, b = 1)
format_json(x_na)
format_json(x_na, na = "string")

# auto_unbox
x_autounbox <- list(1, 1:3)
format_json(x_autounbox)
format_json(x_autounbox, auto_unbox = TRUE)

# dataframe conversion
x_df <- iris[1:2, ]
format_json(x_df, pretty = TRUE)
format_json(x_df, dataframe = "columns", pretty = TRUE)
format_json(x_df, dataframe = "values", pretty = TRUE)

# json_verbatim
x_json <- json2('["a","b"]')
format_json(x_json)
format_json(x_json, json_verbatim = FALSE)

# Decimal vs significant digits
x <- 10 + pi
format_json(x)
format_json(x, digits = NA)
format_json(x, digits = 2)
format_json(x, digits = I(2))

# Force decimal representation
format_json(12)
format_json(12, always_decimal = TRUE)
```

`format_json_rowwise` *Convert each tbl row to JSON*

## Description

Convert each row of a data frame to a JSON object (this basically produces ndjson).

## Usage

```
format_json_rowwise(
  df,
  null = c("list", "null"),
  na = c("null", "string"),
  auto_unbox = TRUE,
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "integer"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo"),
  digits = 4,
  json_verbatim = TRUE,
  force = FALSE,
  rownames = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| `df` | A dataframe. |
| `null, na, POSIXt, factor, complex, matrix, Date` | passed on to `jsonlite::stream_out`. |
| `auto_unbox, raw, digits, force, ...` | passed on to `jsonlite::stream_out`. |
| `json_verbatim` | Leave JSON as it is and do not encode it again? |
| `rownames` | For data.frames add a field `_row` with the row name? |

## Details

Under the hood `jsonlite::stream_out` is used for the conversion.

## Value

A `json2` vector of JSON objects.

## See Also

write_json(), format_json()

## Examples

```
format_json_rowwise(mtcars[1:3, ])

# use a dataframe column for nested objects
df <- data.frame(x = 1:2)
df$y <- data.frame(z = c("a", "b"))
format_json_rowwise(df)

if (require("dplyr", quietly = TRUE, warn.conflicts = FALSE)) {
  tibble <- tibble::tibble
  # often useful in mutate/transmute
  mtcars %>%
    transmute(json = format_json_rowwise(tibble(mpg, cyl, extra = tibble(disp, hp))))
}
```

---

# got_chars_json *Game of Thrones characters*

---

## Description

A JSON with some basic information about some of the Game of Thrones characters.

## Usage

```
got_chars_json
```

## Format

A JSON array of objects. Each object describes a character and has the following keys:

- url
- id
- name
- alive
- titles
- aliases
- allegiances

## Source

https://anapioficeandfire.com

## Examples

```
got_chars_json
```

## json2 *Construct a json2 object*

### Description

`new_json2()` is a fast, low-level constructor that takes a character vector. `json2()` checks the input for validity. `as_json2()` and `is_json2()` are simple forwarders to `vctrs::vec_cast()` and `vctrs::vec_is()` respectively.

### Usage

```
json2(x = character())

new_json2(x = character())

as_json2(x)

is_json2(x)
```

### Arguments

| | |
|---|---|
| x | A character vector. |

### Value

A `json2` vector.

### Examples

```
json2()
json2('{"abc": 1}')

new_json2()
new_json2('{"abc": 1}')
new_json2(c('{"abc": 1}', '{"def": 2}', "[1, 2, 3]", NA))
```

## json_array_agg *Aggregate into a JSON array*

### Description

Aggregate into a JSON array

### Usage

```
json_array_agg(x)
```

### Arguments

| | |
|---|---|
| x | Vector to collapse into JSON array. |

### Value

A json2 vector of length one with the elements of x collapsed to a JSON array.

### Examples

```
json_array_agg(1:3)
json_array_agg(json2(c('{"a": 1}', '{"b": 2}')))

# can be quite useful in combination with `dplyr::group_by()`
if (require("dplyr", quietly = TRUE, warn.conflicts = FALSE)) {
  tibble::tibble(
    group = c(1, 1, 2, 2),
    json = c(1:4)
  ) %>%
    dplyr::group_by(group) %>%
    dplyr::summarise(json = json_array_agg(json))
}
```

---

## json_array_length *Get array length of JSON arrays*

---

### Description

Get array length of JSON arrays

### Usage

```
json_array_length(x, wrap_scalars = FALSE)
```

### Arguments

| | |
|---|---|
| x | A JSON vector. |
| wrap_scalars | Consider scalars as length one array? |

### Value

An integer vector of array lengths.

### Examples

```
json_array_length(c(NA, "[1, 2, 3]", "[1, 2]"))

# scalars produce an error unless `wrap_scalars` is `TRUE`
json_array_length(1, wrap_scalars = TRUE)
```

## json_array_types *Get the type of array elements*

### Description

This is barely useful on its own but can be of some use in combination with `json_array_length()`.

### Usage

```
json_array_types(x)
```

### Arguments

| | |
|---|---|
| x | A JSON vector. |

### Value

A character vector of JSON types.

### See Also

`json_type()`

### Examples

```
json_array_types(c("[1, true]", '["a", [1]]'))
```

## json_delete *Remove JSON element*

### Description

Remote the elements at the specified paths from a JSON vector.

### Usage

```
json_delete(x, ...)
```

### Arguments

| | |
|---|---|
| x | A JSON vector. |
| ... | Paths to delete. |

### Value

A `json2` vector similar to `x` with the specified paths removed from it.

## Examples

```
x <- c('{"a": 11, "b": {"x": 12}}', NA)

json_delete(x, "$.a")
# remove from multiple paths at once
json_delete(x, "$.a", "$.b")
# remove at a nested path
json_delete(x, "$.b.x")

# non-existing elements are just ignored
json_delete(x, "$.abc")
```

---

`json_extract` *Extract an element from JSON*

---

## Description

Extract an element at a given path.

## Usage

```
json_extract(
  x,
  path,
  ptype = NULL,
  default = NULL,
  na = NA,
  wrap_scalars = FALSE,
  bigint_as_char = bigint_default()
)
```

## Arguments

| | |
|---|---|
| `x` | A JSON vector. |
| `path` | Path to element. This must be a valid JSONpath expression. For example `"$.a.b[0]` extracts the 1 in `{"a": {"b": [1, 2]}}`. |
| `ptype` | Output type. If `NULL`, the default, the output type is determined by computing the common type across all elements. Use `new_json_array()` resp. `new_json_object()` if you know every element is an array resp. object. Mind that the return type will only be `json2`. |
| `default` | Default value if path doesn't exist or element at path is empty. |
| `na` | Default value if element of `x` is `NA`. |
| `wrap_scalars` | Should scalar values be wrapped? Note that scalars are only wrapped if either<br>• `ptype` is `new_json_array()` or `json2` vector.<br>• `ptype` is `NULL` and the elements are a mix of scalar values and arrays. |
| `bigint_as_char` | Convert big integers to character? The option `jsontools.bigint_as_char` is used as default. |

## Value

A vector with class given by ptype and length equal to x. Mind that for new_json_array() and new_json_object() the return type will only be json2.

## Examples

```
x1 <- '{"a": 1, "b": 2}'
json_extract(x1, "$.a")
json_extract('{"a": {"b": 1}}', "$.a")

# `NA` values stay `NA` ...
json_extract(c(NA_character_, x1), "$.a")
# ... but can return the value of `na` instead.
json_extract(c(NA_character_, x1), "$.a", na = 3)

# missing paths error by default ...
try(json_extract(x1, "$.c"))
# ... but can be replaced by the value of `default` instead.
json_extract(x1, "$.c", default = "not there")

# make sure to error if you don't get back an array
json_extract('{"a": [1]}', "$.a", ptype = new_json_array())
try(json_extract('{"a": {"b": 1}}', "$.a", ptype = new_json_array()))
```

---

# json_flatten *Flatten a JSON array*

---

## Description

Flatten a JSON array

## Usage

```
json_flatten(
  x,
  ptype = NULL,
  allow_scalars = FALSE,
  wrap_scalars = FALSE,
  bigint_as_char = bigint_default()
)
```

## Arguments

| | |
|---|---|
| x | A JSON vector. |
| ptype | Output type. If NULL, the default, the output type is determined by computing the common type across all elements. Use new_json_array() resp. new_json_object() if you know every element is an array resp. object. Mind that the return type will only be json2. |

| | |
|---|---|
| allow_scalars | Do not error for scalar elements? |
| wrap_scalars | Should scalar values be wrapped? Note that scalars are only wrapped if either<br>• ptype is new_json_array() or json2 vector.<br>• ptype is NULL and the elements are a mix of scalar values and arrays. |
| bigint_as_char | Convert big integers to character? The option jsontools.bigint_as_char is used as default. |

## Value

A flattened version of x with class given by ptype and length equal to the sum of the json_array_length() of the components of x.

## See Also

json_unnest_longer(), json_unnest_wider()

## Examples

```
json_flatten(c("[1, 2]", "[3]"))

# names are kep
json_flatten(c(x = "[1, 2]", y = "[3]"))

# scalar elements produce an error ...
try(json_flatten(c(x = "[1, 2]", y = "3")))
# ... but can be explicitly allowed with `allow_scalars`
json_flatten(c(x = "[1, 2]", y = "3"), allow_scalars = TRUE)
```

---

# json_hoist *Rectangle a JSON vector*

---

## Description

Rectangle a JSON vector

## Usage

```
json_hoist(
  .data,
  .col,
  ...,
  .remove = TRUE,
  .ptype = list(),
  .wrap_scalars = list(),
  .default = list(),
  .na = list(),
  bigint_as_char = bigint_default()
)
```

### Arguments

| | |
|---|---|
| `.data` | A data frame. |
| `.col` | JSON-column. |
| `...` | Elements of `.col` to turn into columns in the form `col_name = "JSON path"`. |
| `.remove` | If `TRUE`, the default, will remove the column `.col`. |
| `.ptype, .wrap_scalars, .default, .na, bigint_as_char` | Optionally, a named list of parameters passed to `json_extract()`. |

### Value

A data frame, or subclass of data frame with new columns as specified in `...`.

### Examples

```
df <- tibble::tibble(
  id = 1:5,
  json = json_flatten(got_chars_json)
)
json_hoist(df, json, url = "$.url", name = "$.name")

# the names can also be generated automatically
json_hoist(df, json, "$.url", "$.name")
```

---

## `json_is_valid` *Assert vector is valid JSON.*

---

### Description

Uses `jsonlite::validate()` under the hood.

### Usage

```
json_is_valid(x)

json_assert_valid(x, x_arg = "")
```

### Arguments

| | |
|---|---|
| `x` | A character vector. |
| `x_arg` | Argument name for `x`. Used in error message to inform the user about the location of the error. |

### Value

`json_is_valid()` returns a vector of `TRUE` and `FALSE`. `json_assert_valid()` either throws an error with information on the invalid elements or returns `x` invisibly

## Examples

```
json_is_valid("[1, 2]")
json_is_valid("[1, 2")

json_assert_valid("[1, 2]")
## Not run:
json_assert_valid("[1, 2")

## End(Not run)
```

---

`json_keys` *Get keys of JSON object resp. array*

---

## Description

Get keys of JSON object resp. array

## Usage

```
json_keys(x)
```

## Arguments

| | |
|---|---|
| x | A JSON vector |

## Value

A list of keys.

## Examples

```
json_keys(c(
  '{"a": 1, "b": 2}',
  '{"x": 1, "y": 2}',
  "[1, 2]"
))
```

## json_merge *Merge two JSON objects*

### Description

By merging two objects you can add, modify, or remove elements of an object. Arrays cannot be modified but only replaced as a whole. It is mostly a small wrapper around the SQLite function `json_patch()`.

### Usage

```
json_merge(x, y)
```

### Arguments

| | |
|---|---|
| x | A JSON vector to update. |
| y | A JSON vector with updated values. |

### Value

A json2 vector.

### Examples

```
# update element with key "a"
json_merge('{"a": 1, "c": 3}', '{"a": 11}')

# you can also add elements
json_merge('{"a": 1, "c": 3}', '{"b": 2}')

# remove elements with `null`
json_merge('{"a": 1, "c": 3}', '{"c": null}')
```

## json_mutate *Update values*

### Description

Update values

### Usage

```
json_mutate(x, ...)
```

### Arguments

| | |
|---|---|
| x | A JSON vector. |
| ... | Name-value pairs. The name is the JSON path (without leading "$"). |

### Value

A json2 vector similar to x with the components modified as specified in ....

### Examples

```
x_na <- c('{"a": 11, "b": {"x": 12}}', NA, '{"a": 21, "b": {"x": 22}}')
# update with different values
json_mutate(x_na, .a = 1:3)

# NA is translated to null
json_mutate(x_na, .a = 1:3, .b.x = NA)

# create new keys
json_mutate(x_na, .c = 0, .d.x = c("a", "b", "c"))
```

---

## json_path_exists *Does the path exist?*

---

### Description

Does the path exist?

### Usage

```
json_path_exists(x, path)
```

### Arguments

| | |
|---|---|
| x | A JSON vector. |
| path | Path to element. This must be a valid JSONpath expression. For example "$.a.b[0] extracts the 1 in {"a": {"b": [1, 2]}}. |

### Value

A logical vector.

## Examples

```
json_path_exists(
  c(
    '{"a": 1}',
    '{"b": 2}',
    "[1, 2]",
    NA_character_
  ),
  "$.a"
)
```

---

`json_prettify` *Prettify/Minify a JSON vector*

---

## Description

A wrapper around `jsonlite::prettify()` resp. `jsonlite::minify()`.

## Usage

```
json_prettify(x, indent = 3)

json_minify(x)
```

## Arguments

| | |
|---|---|
| `x` | A JSON vector. |
| `indent` | number of spaces to indent. |

## Value

A `json2` vector.

## Examples

```
x <- c("[1,2,   3]", '{"a": 1, "b": 2}')
json_prettify(x)
json_minify(x)
```

## json_type *Query the JSON type*

### Description

The JSON types are

- null
- true, false
- integer
- real
- array
- object

### Usage

```
json_type(x, path = NULL)
```

### Arguments

| | |
|---|---|
| x | A JSON vector |
| path | Path to element. This must be a valid JSONpath expression. For example `"$.a.b[0]` extracts the 1 in `{"a": {"b": [1, 2]}}`. |

### Value

A character vector of JSON types

### Examples

```
json_type(c(NA, "1", "null", "[1,2]", '{"a": 1}'))
```

## json_u *Unbox a vector or data frame*

### Description

Mark a vector of length one to not be wrapped in an array when formatted as `JSON`. This is only a tiny wrapper around `jsonlite::unbox()` to avoid conflict with `rlang::unbox()`.

### Usage

```
json_u(x)
```

### Arguments

| | |
|---|---|
| x | atomic vector of length 1, or data frame with 1 row. |

### Value

A singleton version of x.

### Examples

```
format_json(list(foo = 123))
format_json(list(foo = json_u(123)))
```

---

## json_unnest_longer *Unnest a JSON array column*

---

### Description

Unnest a column of JSON arrays in a data frame producing a longer data frame.

### Usage

```
json_unnest_longer(
  data,
  col,
  values_to = NULL,
  row_numbers_to = NULL,
  indices_to = NULL,
  ptype = NULL,
  wrap_scalars = FALSE,
  bigint_as_char = bigint_default()
)
```

### Arguments

| | |
|---|---|
| data | A data frame. |
| col | JSON-column of arrays to extract components from. |
| values_to | Name of column to store vector values. Defaults to col. |
| row_numbers_to | Name of column to store the row number before unnesting. |
| indices_to | Name of column to store the array index of each element; note that this starts with 0. |
| ptype | Output type. If NULL, the default, the output type is determined by computing the common type across all elements. Use new_json_array() resp. new_json_object() if you know every element is an array resp. object. Mind that the return type will only be json2. |
| wrap_scalars | Should scalar values be wrapped? Note that scalars are only wrapped if either |

- ptype is new_json_array() or json2 vector.
- ptype is NULL and the elements are a mix of scalar values and arrays.

bigint_as_char Convert big integers to character? The option jsontools.bigint_as_char is used as default.

## Value

A data frame, or subclass of data frame.

## See Also

json_unnest_wider()

## Examples

```
df <- tibble::tibble(
  x = c("a", "b"),
  json = c("[1, 2]", "[3, 4, 5]")
)
df

df %>%
  json_unnest_longer(
    "json",
    row_numbers_to = "id",
    indices_to = "index"
  )
```

---

# json_unnest_wider *Unnest a JSON object into columns*

---

## Description

Unnest a column of JSON objects in a data frame producing a wider data frame.

## Usage

```
json_unnest_wider(
  data,
  col,
  ptype = list(),
  names_sort = FALSE,
  names_sep = NULL,
  names_repair = "check_unique",
  wrap_scalars = FALSE,
  bigint_as_char = bigint_default()
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| col | JSON-column of arrays to extract components from. |
| ptype | Output type. If NULL, the default, the output type is determined by computing the common type across all elements. Use new_json_array() resp. new_json_object() if you know every element is an array resp. object. Mind that the return type will only be json2. |
| names_sort | Should the extracted columns be sorted by name? If FALSE, the default, the columns are sorted by appearance. |
| names_sep | If NULL, the default, the keys of the objects in col are used as the column names. If a character it is used to join col and the object keys. |
| names_repair | What happens if the output has invalid column names? |
| wrap_scalars | A named list of TRUE or FALSE specifying for each field whether to wrap scalar values in a JSON array. Unspecified fields are not wrapped. This can also be a single value of TRUE or FALSE that is then used for every field. Note that scalars are only wrapped if either<br>• ptype is new_json_array() or json2 vector.<br>• ptype is NULL and the elements are a mix of scalar values and arrays. |
| bigint_as_char | Convert big integers to character? The option jsontools.bigint_as_char is used as default. |

## Value

A data frame, or subclass of data frame of the same length as data.

## See Also

json_unnest_longer()

## Examples

```
# turn all components of item into columns with json_unnest_wider()
tibble::tibble(
  id = 1:2,
  x = c(
    '{"name": "Peter", "age": 19}',
    '{"age": 37}'
  )
) %>%
  json_unnest_wider(x)

# sort names and specify proto types
tibble::tibble(
  id = 1:2,
  x = c(
    '{"name": "Peter", "age": 19, "purchase_ids": [1, 2]}',
    '{"age": 37, "purchase_ids": []}'
```

```
  )
) %>%
  json_unnest_wider(
    x,
    ptype = list(
      age = integer(),
      name = character(),
      purchase_id = new_json_array()
    ),
    names_sort = TRUE
  )
```

---

`json_wrap_scalars` *Wrap scalars in a JSON array*

---

## Description

Wrap scalars in a JSON array

## Usage

```
json_wrap_scalars(x)
```

## Arguments

| | |
|---|---|
| `x` | A character or numeric vector. |

## Value

A `json2` vector similar to `x` with the JSON scalars wrapped as JSON array.

## Examples

```
json_wrap_scalars(c('["a", "b"]', "c", "d"))
json_wrap_scalars(c(1, 2))
```

---

`new_json_object` *Prototype helpers*

---

## Description

Create a JSON object/array prototype to pass to the `.ptype` argument in some of the functions.

## Usage

```
new_json_object()

new_json_array()
```

### Value

An empty `json2_object` resp. `json2_array`.

---

`object` *Creating JSON objects*

---

### Description

Creating JSON objects

### Create a JSON object from `R` values

Given some `R` values you can easily create an object with them by putting them into a named list and applying `format_json()`:

```
id <- 1
x <- 1:3
y <- c("a", "b")

list(id = json_u(id), x = x, y = y) %>%
  format_json()

# or in some cases you might be interested in using `dataframe = "columns"`
tibble::tibble(
  x = 1:3,
  y = c("a", "b", "c")
) %>%
  format_json(dataframe = "columns")
```

To create multiple objects at once (basically a vectorised version) put the values in a data frame and apply `format_json_rowwise()`:

```
df <- tibble::tibble(
  id = 1,
  x = list(1:2, 3:4, 5),
  y = c("a", "b", "c")
)
format_json_rowwise(df)
```

parse_json *Convert JSON to an* R *object*

## Description

A wrapper around the great `jsonlite::parse_json`. The differences are:

- expose argument `bigint_as_char` with default `TRUE`.
- control how to handle `NA` and `NULL`.
- `simplifyDataFrame`, `simplifyMatrix`, and `flatten` default to `FALSE` as they are not very stable in many real world APIs. Use the tibblify package for a more robust conversion to a dataframe.
- don't collapse strings but error instead if they have more than one element.

## Usage

```
parse_json(
  x,
  .na = json_na_error(),
  .null = NULL,
  simplifyVector = TRUE,
  simplifyDataFrame = FALSE,
  simplifyMatrix = FALSE,
  flatten = FALSE,
  bigint_as_char = bigint_default(),
  ...
)
```

## Arguments

| | |
|---|---|
| `x` | a scalar JSON character |
| `.na` | Value to return if `x` is `NA`. By default an error of class `jsontools_error_na_json` is thrown. |
| `.null` | Return the prototype of `.null` if `x` is `NULL` or a zero length character |
| `simplifyVector, simplifyDataFrame, simplifyMatrix, flatten, ...` | passed on to `jsonlite::parse_json`. |
| `bigint_as_char` | Parse big integers as character? The option `jsontools.bigint_as_char` is used as default. |

## Details

To parse a vector of JSON use `parse_json_vector`.

## Value

A R object. The type depends on the input but is usually a list or a data frame.

## Examples

```
# Parse escaped unicode
parse_json('{"city" : "Z\\u00FCrich"}')

# big integers
big_num <- "9007199254740993"
as.character(parse_json(big_num, bigint_as_char = FALSE))
as.character(parse_json(big_num, bigint_as_char = TRUE))

# NA error by default
try(parse_json(NA))
# ... but one can specify a default value
parse_json(NA, .na = data.frame(a = 1, b = 2))

# input of size 0
parse_json(NULL)
parse_json(character(), .null = data.frame(a = 1, b = 2))
```

---

# parse_json_vector *Parse a vector of JSON into a list*

---

## Description

Parse a vector of JSON into a list

## Usage

```
parse_json_vector(
  x,
  .na = json_na_error(),
  .null = NULL,
  simplifyVector = TRUE,
  simplifyDataFrame = FALSE,
  simplifyMatrix = FALSE,
  flatten = FALSE,
  bigint_as_char = bigint_default(),
  ...
)
```

## Arguments

| | |
|---|---|
| x | a scalar JSON character |
| .na | Value to return if x is NA. By default an error of class jsontools_error_na_json is thrown. |
| .null | Return the prototype of .null if x is NULL or a zero length character |
| simplifyVector | passed on to jsonlite::parse_json. |

| | |
|---|---|
| simplifyDataFrame | passed on to jsonlite::parse_json. |
| simplifyMatrix | passed on to jsonlite::parse_json. |
| flatten | passed on to jsonlite::parse_json. |
| bigint_as_char | Parse big integers as character? The option jsontools.bigint_as_char is used as default. |
| ... | passed on to jsonlite::parse_json. |

## Value

A list of the same length as x.

## Examples

```
parse_json_vector(x = c('"a"', '"b"'))
parse_json_vector(x = c('"a"', '["b", "c"]'))
parse_json_vector(x = c('"a"', NA), .na = 1)
```

---

# read_json *Read JSON from disk or url*

---

## Description

Read JSON from disk or url

## Usage

```
read_json(path, ...)
```

## Arguments

| | |
|---|---|
| path | Path or connection to read from |
| ... | arguments passed on to parse_json |

## Value

A R object. The type depends on the input but is usually a list or a data frame.

vec_ptype2.json2 *Coercion*

## Description

Double dispatch methods to support `vctrs::vec_ptype2()`.

## Usage

```
## S3 method for class 'json2'
vec_ptype2(x, y, ..., x_arg = "", y_arg = "")

## S3 method for class 'json'
vec_ptype2(x, y, ..., x_arg = "", y_arg = "")

## S3 method for class 'jqson'
vec_ptype2(x, y, ..., x_arg = "", y_arg = "")

## S3 method for class 'pq_jsonb'
vec_ptype2(x, y, ..., x_arg = "", y_arg = "")

vec_ptype2.pq_json(x, y, ..., x_arg = "", y_arg = "")
```

## Arguments

| | |
|---|---|
| x | Vector types. |
| y | Vector types. |
| ... | These dots are for future extensions and must be empty. |
| x_arg | Argument names for x and y. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`). |
| y_arg | Argument names for x and y. These are used in error messages to inform the user about the locations of incompatible types (see `stop_incompatible_type()`). |

## Value

A length-zero R object.

write_json *Write JSON*

## Description

Write JSON

## Usage

```
write_json(x, path, ...)
```

## Arguments

| | |
|---|---|
| x | An object to write to disk. |
| path | Path or connection to write to. |
| ... | arguments passed on to format_json |

## Value

Returns the input x invisibly.

## See Also

format_json(), format_json_rowwise()

# Index