Package 'geojson'

June 23, 2020

Type Package

Title Classes for 'GeoJSON'

Description Classes for 'GeoJSON' to make working with 'GeoJSON' easier. Includes S3 classes for 'GeoJSON' classes with brief summary output, and a few methods such as extracting and adding bounding boxes, properties, and coordinate reference systems; working with newline delimited 'GeoJSON'; linting through the 'geojsonlint' package; and serializing to/from 'Geobuf' binary 'GeoJSON' format.

Version 0.3.4

License MIT + file LICENSE

URL https://docs.ropensci.org/geojson, https://github.com/ropensci/geojson

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

LazyData true

VignetteBuilder knitr

Encoding UTF-8

Imports methods, sp, jsonlite (>= 1.6), protolite (>= 1.8), jqr (>= 1.1.0), magrittr, lazyeval

Suggests geojsonlint (>= 0.2.0), tibble, testthat, knitr, rmarkdown, sf, stringi

RoxygenNote 7.1.0

X-schema.org-applicationCategory Geospatial

X-schema.org-keywords geojson, geospatial, conversion, data, input-output, bbox, polygon, geobuf

X-schema.org-isPartOf https://ropensci.org

NeedsCompilation no

Author Scott Chamberlain [aut, cre] (<https://orcid.org/0000-0003-1444-9135>), Jeroen Ooms [aut]

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2020-06-23 20:00:03 UTC

R topics documented:

`geojson-package` *geojson*

Description

Classes for GeoJSON to make working with GeoJSON easier

Package API

GeoJSON objects:

- `feature` - Feature
- `featurecollection` - FeatureCollection
- `geometrycollection` - GeometryCollection
- `linestring` - LineString

- multilinestring - MultiLineString
- multipoint - MultiPoint
- multipolygon - MultiPolygon
- point - Point
- polygon - Polygon

The above are assigned two classes. All of them are class **geojson**, but also have a class name that is **geo** plus the name of the geometry, e.g., **geopolygon** for polygon.

GeoJSON properties:

- properties_add, properties_get - Add or get properties
- crs_add, crs_get - Add or get CRS
- bbox_add, bbox_get - Add or get bounding box

GeoJSON operations:

- geo_bbox - calculate a bounding box for any GeoJSON object
- geo_pretty - pretty print any GeoJSON object
- geo_type - get the object type for any GeoJSON object
- geo_write - easily write any GeoJSON to a file
- More complete GeoJSON operations are provdied in the package **geoops**

GeoJSON/Geobuf serialization:

- from_geobuf - Geobuf to GeoJSON
- to_geobuf - GeoJSON to Geobuf
- Check out https://github.com/mapbox/geobuf for inormation on the Geobuf format

Coordinate Reference System

According to RFC 7946 (https://tools.ietf.org/html/rfc7946#page-12) the CRS for all GeoJSON objects must be WGS-84, equivalent to `urn:ogc:def:crs:OGC::CRS84`. And lat/long must be in decimal degrees.

Given the above, but considering that GeoJSON blobs exist that have CRS attributes in them, we provide CRS helpers in this package. But moving forward these are not likely to be used much.

Coordinate precision

According to RFC 7946 (https://tools.ietf.org/html/rfc7946#section-11.2) consider that 6 decimal places amoutns to ~10 centimeters, a precision well within that of current GPS sytems. Further, A GeoJSON text containing many detailed Polygons can be inflated almost by a factor of two by increasing coordinate precision from 6 to 15 decimal places - so consider whether it is worth it to have more decimal places.

Author(s)

Scott Chamberlain, Jeroen Ooms

as.geojson *Geojson class*

Description

Geojson class

Usage

```
as.geojson(x)

## S4 method for signature 'json'
as.geojson(x)

## S4 method for signature 'geojson'
as.geojson(x)

## S4 method for signature 'character'
as.geojson(x)

## S4 method for signature 'SpatialPointsDataFrame'
as.geojson(x)

## S4 method for signature 'SpatialPoints'
as.geojson(x)

## S4 method for signature 'SpatialLinesDataFrame'
as.geojson(x)

## S4 method for signature 'SpatialLines'
as.geojson(x)

## S4 method for signature 'SpatialPolygonsDataFrame'
as.geojson(x)

## S4 method for signature 'SpatialPolygons'
as.geojson(x)
```

Arguments

x	input, an object of class character, json, SpatialPoints, SpatialPointsDataFrame, SpatialLines, SpatialLinesDataFrame, SpatialPolygons, or SpatialPolygonsDataFrame

Details

The `print.geojson` method prints the geojson geometry type, the bounding box, number of features (if applicable), and the geometries and their lengths

Value

an object of class geojson/json

Examples

```
# character
as.geojson(geojson_data$featurecollection_point)
as.geojson(geojson_data$polygons_average)
as.geojson(geojson_data$polygons_aggregate)
as.geojson(geojson_data$points_count)

# sp classes

## SpatialPoints
library(sp)
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPoints(cbind(x,y))
as.geojson(s)

## SpatialPointsDataFrame
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
as.geojson(s)

## SpatialLines
L1 <- Line(cbind(c(1,2,3), c(3,2,2)))
L2 <- Line(cbind(c(1.05,2.05,3.05), c(3.05,2.05,2.05)))
L3 <- Line(cbind(c(1,2,3),c(1,1.5,1)))
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
as.geojson(sl1)

## SpatialLinesDataFrame
sl12 <- SpatialLines(list(Ls1, Ls2))
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
as.geojson(sldf)

## SpatialPolygons
poly1 <- Polygons(list(Polygon(cbind(c(-100,-90,-85,-100),
   c(40,50,45,40)))), "1")
poly2 <- Polygons(list(Polygon(cbind(c(-90,-80,-75,-90),
   c(30,40,35,30)))), "2")
sp_poly <- SpatialPolygons(list(poly1, poly2), 1:2)
as.geojson(sp_poly)

## SpatialPolygonsDataFrame
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
as.geojson(sp_polydf)
```

```
## sf objects
if (requireNamespace('sf')) {
  nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
  as.geojson(nc)
}
```

---

bbox *Add or get bounding box*

Description

Add or get bounding box

Usage

```
bbox_add(x, bbox = NULL)

bbox_get(x)
```

Arguments

x	An object of class `geojson`
bbox	(numeric) a vector or list of length 4 for a 2D bounding box or length 6 for a 3D bounding box. If `NULL`, the bounding box is calculated for you

Details

Note that `bbox_get` outputs the bbox if it exists, but does not calculate it from the geojson. See `geo_bbox` to calculate a bounding box. Bounding boxes can be 2D or 3D.

Value

- bbox_add: an object of class jqson/character from **jqr**
- bbox_get: a bounding box, of the form `[west,south,east,north]` for 2D or of the form `[west,south,min-altitude,east,north,max-altitude]` for 3D

References

https://tools.ietf.org/html/rfc7946#section-5

Examples

```
# make a polygon
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
  ]
}'
(y <- polygon(x))

# add bbox - without an input, we figure out the 2D bbox for you
y %>% feature() %>% bbox_add()
## 2D bbox
y %>% feature() %>% bbox_add(c(100.0, -10.0, 105.0, 10.0))
## 3D bbox
y %>% feature() %>% bbox_add(c(100.0, -10.0, 3, 105.0, 10.0, 17))

# get bounding box
z <- y %>% feature() %>% bbox_add()
bbox_get(z)

## returns NULL if no bounding box
bbox_get(x)
```

`crs` *Add or get CRS*

Description

Add or get CRS

Usage

```
crs_add(x, crs)

crs_get(x)
```

Arguments

`x`	An object of class `geojson`
`crs`	(character) a CRS string. required.

Details

According to RFC 7946 (https://tools.ietf.org/html/rfc7946#page-12) the CRS for all GeoJSON objects must be WGS-84, equivalent to `urn:ogc:def:crs:OGC::CRS84`. And lat/long must be in decimal degrees.

Given the above, but considering that GeoJSON blobs exist that have CRS attributes in them, we provide CRS helpers here. But moving forward these are not likely to be used much.

References

https://github.com/OSGeo/proj.4, http://geojson.org/geojson-spec.html#coordinate-reference-system-ob

Examples

```
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
  ]
}'

# add crs
crs <- '{"type": "name",
 "properties": {
     "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
}}'
x %>% feature() %>% crs_add(crs)

# get crs
z <- x %>% feature() %>% crs_add(crs)
crs_get(z)
```

---

feature *feature class*

Description

feature class

Usage

```
feature(x)
```

Arguments

x	input

Details

Feature objects:

- A feature object must have a member with the name "geometry". The value of the geometry member is a geometry object as defined above or a JSON null value.
- A feature object must have a member with the name "properties". The value of the properties member is an object (any JSON object or a JSON null value).
- If a feature has a commonly used identifier, that identifier should be included as a member of the feature object with the name "id".

Examples

```
# point -> feature
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
point(x) %>% feature()

# multipoint -> feature
x <- '{"type": "MultiPoint", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
multipoint(x) %>% feature()

# linestring -> feature
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
linestring(x) %>% feature()

# multilinestring -> feature
x <- '{ "type": "MultiLineString",
 "coordinates": [ [ [100.0, 0.0], [101.0, 1.0] ], [ [102.0, 2.0], [103.0, 3.0] ] ] }'
multilinestring(x) %>% feature()

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(multilinestring(x)))
```

---

featurecollection *featurecollection class*

Description

featurecollection class

Usage

```
featurecollection(x)
```

Arguments

x	input

Examples

```
file <- system.file("examples", 'featurecollection1.geojson',
  package = "geojson")
file <- system.file("examples", 'featurecollection2.geojson',
  package = "geojson")
str <- paste0(readLines(file), collapse = " ")
(y <- featurecollection(str))
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
```

```
unlink(f)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))

# features to featurecollection
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
point(x) %>% feature() %>% featurecollection()

## all points
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
y <- '{ "type": "Point", "coordinates": [100.0, 50.0] }'
featls <- lapply(list(x, y), function(z) feature(point(z)))
featurecollection(featls)
```

---

geobuf *Geobuf serialization*

Description

Geobuf serialization

Usage

```
from_geobuf(x, pretty = FALSE)

to_geobuf(x, file = NULL, decimals = 6)
```

Arguments

x	(character) a file or raw object for `from_geobuf`, and json string for `to_geobuf`
pretty	(logical) pretty print JSON. Default: `FALSE`
file	(character) file to write protobuf to. if NULL, geobuf raw binary returned
decimals	(integer) how many decimals (digits behind the dot) to store for numbers

Details

`from_geobuf` uses `protolite::geobuf2json()`, while `to_geobuf` uses `protolite::json2geobuf()`

Note that **protolite** expects either a **Feature**, **FeatureCollection**, or **Geometry** class geojson object, Thus, for `to_geobuf` we check the geojson class, and convert to a **Feature** if the class is something other than the acceptable set.

Value

for `from_geobuf` JSON as a character string, and for `to_geobuf` raw or file written to disk

References

Geobuf is a compact binary encoding for geographic data using protocol buffers https://github.com/mapbox/geobuf

Examples

```
file <- system.file("examples/test.pb", package = "geojson")
(json <- from_geobuf(file))
from_geobuf(file, pretty = TRUE)
pb <- to_geobuf(json)
f <- tempfile(fileext = ".pb")
to_geobuf(json, f)
from_geobuf(f)

object.size(json)
object.size(pb)
file.info(file)$size
file.info(f)$size

file <- system.file("examples/featurecollection1.geojson",
  package = "geojson")
json <- paste0(readLines(file), collapse = "")
to_geobuf(json)

# other geojson class objects
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
  ]
}'
(y <- polygon(x))
to_geobuf(y)

x <- '{"type": "MultiPoint", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
(y <- multipoint(x))
to_geobuf(y)
```

---

geojson_data *Data for use in examples*

Description

Data for use in examples

Format

A list of character strings of points or polygons in FeatureCollection or Feature Geojson formats.

Details

The data objects included in the list, accessible by name

- featurecollection_point - FeatureCollection with a single point
- filter_features - FeatureCollection of points
- points_average - FeatureCollection of points
- polygons_average - FeatureCollection of polygons
- points_count - FeatureCollection of points
- polygons_count - FeatureCollection of polygons
- points_within - FeatureCollection of points
- polygons_within - FeatureCollection of polygons
- poly - Feaure of a single 1 degree by 1 degree polygon
- multipoly - FeatureCollection of two 1 degree by 1 degree polygons
- polygons_aggregate - FeatureCollection of Polygons from turf.js examples
- points_aggregate - FeatureCollection of Points from turf.js examples

---

geometrycollection *geometrycollection class*

Description

geometrycollection class

Usage

```
geometrycollection(x)
```

Arguments

x	input

Examples

```
x <- '{
 "type": "GeometryCollection",
 "geometries": [
   {
     "type": "Point",
     "coordinates": [100.0, 0.0]
   },
   {
     "type": "LineString",
     "coordinates": [ [101.0, 0.0], [102.0, 1.0] ]
   }
  ]
```

```
}'
(y <- geometrycollection(x))
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

# bigger geometrycollection
file <- system.file("examples", "geometrycollection1.geojson", package = "geojson")
(y <- geometrycollection(paste0(readLines(file), collapse="")))
geo_type(y)
geo_pretty(y)
```

---

geo_bbox *Calculate a bounding box*

Description

Calculate a bounding box

Usage

```
geo_bbox(x)
```

Arguments

x	an object of class geojson

Details

Supports inputs of type: character, point, multipoint, linestring, multilinestring, polygon, multipoygon, feature, and featurecollection

On character inputs, we lint the input to make sure it's proper JSON and GeoJSON, then caculate the bounding box

Value

a vector of four doubles: min lon, min lat, max lon, max lat

Examples

```
# point
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
(y <- point(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()

# multipoint
```

```
x <- '{"type": "MultiPoint", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
(y <- multipoint(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()

# linestring
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
(y <- linestring(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()
file <- system.file("examples", 'linestring_one.geojson',
  package = "geojson")
con <- file(file)
str <- paste0(readLines(con), collapse = " ")
(y <- linestring(str))
geo_bbox(y)
y %>% feature() %>% geo_bbox()
close(con)

## Not run:
# multilinestring
x <- '{ "type": "MultiLineString",
 "coordinates": [ [ [100.0, 0.0], [101.0, 1.0] ], [ [102.0, 2.0],
 [103.0, 3.0] ] ] }'
(y <- multilinestring(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()

# polygon
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
  ]
}'
(y <- polygon(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()

# multipolygon
x <- '{ "type": "MultiPolygon",
"coordinates": [
  [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
  [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
  [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
  ]
}'
(y <- multipolygon(x))
geo_bbox(y)
y %>% feature() %>% geo_bbox()

# featurecollection
file <- system.file("examples", 'featurecollection2.geojson',
  package = "geojson")
```

```
str <- paste0(readLines(file), collapse = " ")
x <- featurecollection(str)
geo_bbox(x)

# character
file <- system.file("examples", 'featurecollection2.geojson',
  package = "geojson")
str <- paste0(readLines(file), collapse = " ")
geo_bbox(str)

# json
library('jsonlite')
geo_bbox(toJSON(fromJSON(str), auto_unbox = TRUE))

## End(Not run)
```

`geo_pretty` *Pretty print geojson*

Description

Pretty print geojson

Usage

```
geo_pretty(x)
```

Arguments

`x`	input, an object of class `geojson`

Details

Wrapper around `prettify`

Examples

```
geo_pretty(point('{ "type": "Point", "coordinates": [100.0, 0.0] }'))

x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [100.0, 1.0], [101.0, 1.0], [101.0, 0.0], [100.0, 0.0] ]
  ]
}'
poly <- polygon(x)
geo_pretty(poly)
```

geo_type *Get geometry type*

Description

Get geometry type

Usage

```
geo_type(x)
```

Arguments

x	input, an object of class geojson

Examples

```
geo_type(point('{ "type": "Point", "coordinates": [100.0, 0.0] }'))

x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [100.0, 1.0], [101.0, 1.0], [101.0, 0.0], [100.0, 0.0] ]
  ]
}'
poly <- polygon(x)

geo_type(poly)
```

geo_write *Write geojson to disk*

Description

Write geojson to disk

Usage

```
geo_write(x, file)
```

Arguments

x	input, an object of class geojson
file	(character) a file path, or connection

Details

Wrapper around `jsonlite::toJSON()` and `cat`

Examples

```
file <- tempfile(fileext = ".geojson")
geo_write(
  point('{ "type": "Point", "coordinates": [100.0, 0.0] }'),
  file
)
readLines(file)
unlink(file)
```

`linestring` *linestring class*

Description

linestring class

Usage

```
linestring(x)
```

Arguments

x	input

Examples

```
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
(y <- linestring(x))
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))
```

linting_opts *GeoJSON Linting*

Description

GeoJSON Linting

Usage

```
linting_opts(
  lint = FALSE,
  method = "hint",
  error = FALSE,
  suppress_pkgcheck_warnings = FALSE
)
```

Arguments

lint	(logical) lint geojson or not. Default: FALSE
method	(character) method to use: • hint - uses geojsonlint::geojson_hint() • lint - uses geojsonlint::geojson_lint() • validate - uses geojsonlint::geojson_validate()
error	(logical) Throw an error on parse failure? If TRUE, then function returns TRUE on success, and stop with the error message on error. Default: FALSE
suppress_pkgcheck_warnings	(logical) Suppress warning when geojsonlint is not installed? Default: FALSE

Details

if you have **geojsonlint** installed, we can lint your GeoJSON inputs for you. If not, we skip that step.

Note that even if you aren't linting your geojson with **geojsonlint**, we still do some minimal checks.

Examples

```
linting_opts(lint = TRUE)

linting_opts(lint = TRUE, method = "hint")
linting_opts(lint = TRUE, method = "hint", error = TRUE)
linting_opts(lint = TRUE, method = "lint")
linting_opts(lint = TRUE, method = "lint", error = TRUE)
linting_opts(lint = TRUE, method = "validate")
linting_opts(lint = TRUE, method = "validate", error = TRUE)
```

multilinestring *multilinestring class*

Description

multilinestring class

Usage

```
multilinestring(x)
```

Arguments

x	input

Examples

```
x <- '{ "type": "MultiLineString",
 "coordinates": [ [ [100.0, 0.0], [101.0, 1.0] ], [ [102.0, 2.0], [103.0, 3.0] ] ] }'
(y <- multilinestring(x))
y[1]
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

file <- system.file("examples", 'multilinestring_one.geojson',
  package = "geojson")
con <- file(file)
str <- paste0(readLines(con), collapse = " ")
(y <- multilinestring(str))
y[1]
geo_type(y)
geo_pretty(y)
close(con)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))
```

multipoint *multipoint class*

Description

multipoint class

Usage

```
multipoint(x)
```

Arguments

x	input

Examples

```
x <- '{"type": "MultiPoint", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ] }'
(y <- multipoint(x))
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))

# as.geojson coercion
as.geojson(x)
```

---

multipolygon *multipolygon class*

Description

multipolygon class

Usage

```
multipolygon(x)
```

Arguments

x	input

Examples

```
x <- '{ "type": "MultiPolygon",
"coordinates": [
  [[[102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0]]],
  [[[100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0]],
  [[100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2]]]
  ]
}'
(y <- multipolygon(x))
```

```
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))
```

---

ndgeo *Read and write newline-delimited GeoJSON (GeoJSON text sequences)*

Description

There are various flavors of newline-delimited GeoJSON, all of which we aim to handle here. See Details for more.

Usage

```
ndgeo_write(x, file, sep = "\n")

## Default S3 method:
ndgeo_write(x, file, sep = "\n")

## S3 method for class 'geofeaturecollection'
ndgeo_write(x, file, sep = "\n")

## S3 method for class 'geofeature'
ndgeo_write(x, file, sep = "\n")

ndgeo_read(txt, pagesize = 500, verbose = TRUE)
```

Arguments

x	input, an object of class geojson
file	(character) a file. not a connection. required.
sep	(character) a character separator to use in writeLines()
txt	text, a file, or a url. required.
pagesize	(integer) number of lines to read/write from/to the connection per iteration
verbose	(logical) print messages. default: TRUE

Details

- `ndgeo_write`: writes **geojson** package types as newline-delimited GeoJSON to a file
- `ndgeo_read`: reads newline-delimited GeoJSON from a string, file, or URL into the appropriate geojson type

As an alternative to `ndgeo_read`, you can simply use `jsonlite::stream_in()` to convert newline-delimited GeoJSON to a data.frame

Value

a `geojson` class object

Note

IMPORTANT: `ngeo_read` for now only handles lines of geojson in your file that are either features or geometry objects (e.g., point, multipoint, polygon, multipolygon, linestring, multilinestring)

References

Newline-delimited JSON has a few flavors. The only difference between ndjson http://ndjson.org/ and JSON Lines http://jsonlines.org/ I can tell is that the former requires UTF-8 encoding, while the latter does not.

GeoJSON text sequences has a specification found at https://tools.ietf.org/html/rfc8142. The spec states that:

- a GeoJSON text sequence is any number of GeoJSON RFC7946 texts
- each line encoded in UTF-8 RFC3629
- each line preceded by one ASCII RFC20 record separator (RS; "0x1e") character
- each line followed by a line feed (LF)
- each JSON text MUST contain a single GeoJSON object as defined in RFC7946

See also the GeoJSON specification https://tools.ietf.org/html/rfc7946

Examples

```
# featurecollection
## write
file <- system.file("examples", 'featurecollection2.geojson',
  package = "geojson")
str <- paste0(readLines(file), collapse = " ")
(x <- featurecollection(str))
outfile <- tempfile(fileext = ".geojson")
ndgeo_write(x, outfile)
readLines(outfile)
jsonlite::stream_in(file(outfile))
## read
ndgeo_read(outfile)
unlink(outfile)

# read from an existing file
```

```
## GeoJSON objects all of same type: Feature
file <- system.file("examples", 'ndgeojson1.json', package = "geojson")
ndgeo_read(file)
## GeoJSON objects all of same type: Point
file <- system.file("examples", 'ndgeojson2.json', package = "geojson")
ndgeo_read(file)
## GeoJSON objects of mixed type: Point, and Feature
file <- system.file("examples", 'ndgeojson3.json', package = "geojson")
ndgeo_read(file)

## Not run:
# read from a file
url <- "https://raw.githubusercontent.com/ropensci/geojson/master/inst/examples/ndgeojson1.json"
f <- tempfile(fileext = ".geojsonl")
download.file(url, f)
x <- ndgeo_read(f)
x
unlink(f)

# read from a URL
url <- "https://raw.githubusercontent.com/ropensci/geojson/master/inst/examples/ndgeojson1.json"
x <- ndgeo_read(url)
x

# geojson text sequences from file
file <- system.file("examples", 'featurecollection2.geojson',
  package = "geojson")
str <- paste0(readLines(file), collapse = " ")
x <- featurecollection(str)
outfile <- tempfile(fileext = ".geojson")
ndgeo_write(x, outfile, sep = "\u001e\n")
con <- file(outfile)
readLines(con)
close(con)
ndgeo_read(outfile)
unlink(outfile)

## End(Not run)
```

`point` *point class*

Description

point class

Usage

```
point(x)
```

Arguments

x	input

Examples

```
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
(y <- point(x))
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))

# as.geojson coercion
as.geojson(x)
```

---

polygon *polygon class*

Description

polygon class

Usage

```
polygon(x)
```

Arguments

x	input

Examples

```
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [100.0, 1.0], [101.0, 1.0], [101.0, 0.0], [100.0, 0.0] ]
  ]
}'
(y <- polygon(x))
y[1]
geo_type(y)
geo_pretty(y)
geo_write(y, f <- tempfile(fileext = ".geojson"))
jsonlite::fromJSON(f, FALSE)
unlink(f)
```

```
x <- '{ "type": "Polygon",
"coordinates": [
  [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],
  [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ]
  ]
}'
(y <- polygon(x))
y[1]
geo_type(y)
geo_pretty(y)

# add to a data.frame
library('tibble')
tibble(a = 1:5, b = list(y))
```

`properties` *Add or get properties*

Description

Add or get properties

Usage

```
properties_add(x, ..., .list = NULL)

properties_get(x, property)
```

Arguments

`x`	An object of class `geojson`
`...`	Properties to be added, supports NSE as well as SE
`.list`	a named list of properties to add. must be named
`property`	(character) property name

References

http://geojson.org/geojson-spec.html

Examples

```
# add properties
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
(y <- linestring(x))
y %>% feature() %>% properties_add(population = 1000)

## add with a named list already created
```

```
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
(y <- linestring(x))
props <- list(population = 1000, temperature = 89, size = 5)
y %>% feature() %>% properties_add(.list = props)

## combination of NSE and .list
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
(y <- linestring(x))
props <- list(population = 1000, temperature = 89, size = 5)
y %>% feature() %>% properties_add(stuff = 4, .list = props)

# features to featurecollection
x <- '{ "type": "Point", "coordinates": [100.0, 0.0] }'
point(x) %>%
  feature() %>%
  featurecollection() %>%
  properties_add(population = 10)

# get property
x <- '{ "type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
(y <- linestring(x))
x <- y %>% feature() %>% properties_add(population = 1000)
properties_get(x, property = 'population')
```

`to_geojson` *Convert GeoJSON character string to approriate GeoJSON class*

Description

Automatically detects and adds the class

Usage

```
to_geojson(x)
```

Arguments

x	GeoJSON character string

Examples

```
mp <- '{"type":"MultiPoint","coordinates":[[100,0],[101,1]]}'
to_geojson(mp)

ft <- '{"type":"Feature","properties":{"a":"b"},
"geometry":{"type": "MultiPoint","coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}}'
to_geojson(mp)

fc <- '{"type":"FeatureCollection","features":[{"type":"Feature","properties":{"a":"b"},
"geometry":{"type": "MultiPoint","coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}}]}'
to_geojson(fc)
```

Index