Package 'geojsonio'

September 1, 2017

Title Convert Data from and to 'GeoJSON' or 'TopoJSON'

Description Convert data to 'GeoJSON' or 'TopoJSON' from various R classes, including vectors, lists, data frames, shape files, and spatial classes. 'geojsonio' does not aim to replace packages like 'sp', 'rgdal', 'rgeos', but rather aims to be a high level client to simplify conversions of data from and to 'GeoJSON' and 'TopoJSON'.

Version 0.4.2

License MIT + file LICENSE

URL https://github.com/ropensci/geojsonio

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

LazyData true

VignetteBuilder knitr

Depends R (>= 2.10)

Imports methods, sp, rgdal (>= 1.1-1), rgeos, httr (>= 1.1.0), maptools, jsonlite (>= 0.9.21), magrittr, readr (>= 0.2.2), V8, geojson

Suggests gistr, testthat, knitr, leaflet, sf

Enhances RColorBrewer

RoxygenNote 6.0.1

NeedsCompilation no

Author Scott Chamberlain [aut, cre], Andy Teucher [aut]

Maintainer Scott Chamberlain <myrmecocystus@gmail.com>

Repository CRAN

Date/Publication 2017-09-01 20:22:42 UTC

R topics documented:

`as.json` *Convert inputs to JSON*

Description

Convert inputs to JSON

Usage

```
as.json(x, ...)
```

Arguments

`x`	Input
`...`	Further args passed on to `toJSON`

Examples

```
## Not run:
(res <- geojson_list(us_cities[1:2,], lat='lat', lon='long'))
as.json(res)
as.json(res, pretty = TRUE)

vec <- c(-99.74,32.45)
as.json(geojson_list(vec))
as.json(geojson_list(vec), pretty = TRUE)

## End(Not run)
```

---

as.location *Convert a path or URL to a location object.*

Description

Convert a path or URL to a location object.

Usage

```
as.location(x, ...)
```

Arguments

x	Input.
...	Ignored.

Examples

```
## Not run:
# A file
file <- system.file("examples", "zillow_or.geojson", package = "geojsonio")
as.location(file)

# A URL
url <- "https://raw.githubusercontent.com/glynnbird/usstatesgeojson/master/california.geojson"
as.location(url)

## End(Not run)
```

bounds *Get bounds for a list or geo_list*

Description

Get bounds for a list or geo_list

Usage

```
bounds(x, ...)
```

Arguments

x	An object of class list or geo_list
...	Ignored

Value

A vector of the form min longitude, min latitude, max longitude, max latitude

Examples

```
# numeric
vec <- c(-99.74,32.45)
x <- geojson_list(vec)
bounds(x)

# list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
x <- geojson_list(mylist)
bounds(x)

# data.frame
x <- geojson_list(states[1:20,])
bounds(x)
```

canada_cities *This is the same data set from the maps library, named differently*

Description

This database is of Canadian cities of population greater than about 1,000. Also included are province capitals of any population size.

Format

A list with 6 components, namely "name", "country.etc", "pop", "lat", "long", and "capital", containing the city name, the province abbreviation, approximate population (as at January 2006), latitude, longitude and capital status indication (0 for non-capital, 1 for capital, 2 for provincial

---

`centroid` *Get centroid for a geo_list*

Description

Get centroid for a geo_list

Usage

```
centroid(x, ...)
```

Arguments

`x`	An object of class geo_list
`...`	Ignored

Value

A vector of the form longitude, latitude

Examples

```
# numeric
vec <- c(-99.74,32.45)
x <- geojson_list(vec)
centroid(x)

# list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
x <- geojson_list(mylist)
centroid(x)

# data.frame
x <- geojson_list(states[1:20,])
centroid(x)
```

`file_to_geojson` *Convert spatial data files to GeoJSON from various formats.*

Description

You can use a web interface called Ogre, or do conversions locally using the rgdal package.

Usage

```
file_to_geojson(input, method = "web", output = ".", parse = FALSE,
  encoding = "CP1250", verbose = FALSE, ...)
```

Arguments

`input`	The file being uploaded, path to the file on your machine.
`method`	(character) One of "web" (default) or "local". Matches on partial strings. This parameter determines how the data is read. "web" means we use the Ogre web service, and "local" means we use **rgdal**. See Details fore more.
`output`	Destination for output geojson file. Defaults to current working directory, and gives a random alphanumeric file name
`parse`	(logical) To parse geojson to data.frame like structures if possible. Default: `FALSE`
`encoding`	(character) The encoding passed to `readOGR`. Default: CP1250
`verbose`	(logical) Printing of `readOGR` progress. Default: `FALSE`
`...`	Additional parameters passed to `readOGR`

Method parameter

The web option uses the Ogre web API. Ogre currently has an output size limit of 15MB. See here http://ogre.adc4gis.com/ for info on the Ogre web API. The local option uses the function `writeOGR` from the package rgdal.

Ogre

Note that for Shapefiles, GML, MapInfo, and VRT, you need to send zip files to Ogre. For other file types (.bna, .csv, .dgn, .dxf, .gxt, .txt, .json, .geojson, .rss, .georss, .xml, .gmt, .kml, .kmz) you send the actual file with that file extension.

Linting GeoJSON

If you're having trouble rendering GeoJSON files, ensure you have a valid GeoJSON file by running it through the package **geojsonlint**, which has a variety of different GeoJSON linters.

Examples

```
## Not run:
file <- system.file("examples", "norway_maple.kml", package = "geojsonio")

# KML type file - using the web method
file_to_geojson(input=file, method='web', output='kml_web')
## read into memory
file_to_geojson(input=file, method='web', output = ":memory:")
file_to_geojson(input=file, method='local', output = ":memory:")

# KML type file - using the local method
file_to_geojson(input=file, method='local', output='kml_local')

# Shp type file - using the web method - input is a zipped shp bundle
file <- system.file("examples", "bison.zip", package = "geojsonio")
file_to_geojson(file, method='web', output='shp_web')

# Shp type file - using the local method - input is the actual .shp file
file <- system.file("examples", "bison.zip", package = "geojsonio")
dir <- tempdir()
unzip(file, exdir = dir)
list.files(dir)
shpfile <- file.path(dir, "bison-Bison_bison-20130704-120856.shp")
file_to_geojson(shpfile, method='local', output='shp_local')

# Neighborhoods in the US
## beware, this is a long running example
# url <- 'http://www.nws.noaa.gov/geodata/catalog/national/data/ci08au12.zip'
# out <- file_to_geojson(input=url, method='web', output='cities')

# geojson with .json extension
x <- gsub("\n", "", paste0('https://gist.githubusercontent.com/hunterowens/
25ea24e198c80c9fbcc7/raw/7fd3efda9009f902b5a991a506cea52db19ba143/
wards2014.json', collapse = ""))
res <- file_to_geojson(x)
jsonlite::fromJSON(res)
res <- file_to_geojson(x, method = "local")
jsonlite::fromJSON(res)

## End(Not run)
```

---

geo2topo *GeoJSON to TopoJSON and back*

Description

GeoJSON to TopoJSON and back

Usage

```
geo2topo(x)

topo2geo(x, ...)
```

Arguments

x	GeoJSON or TopoJSON as a character string, json, a file path, or url
...	for topo2geo args passed on to readOGR

Value

An object of class json, of either GeoJSON or TopoJSON

See Also

topojson_write, topojson_read

Examples

```
# geojson to topojson
x <- '{"type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
z <- geo2topo(x)
jsonlite::prettify(z)
## Not run:
library(leaflet)
leaflet() %>%
  addProviderTiles(provider = "Stamen.Terrain") %>%
  addTopoJSON(z)

## End(Not run)

# topojson to geojson
w <- topo2geo(z)
jsonlite::prettify(w)

## larger examples
file <- system.file("examples", "us_states.topojson", package = "geojsonio")
topo2geo(file)
```

---

geojson-add *Add together geo_list or json objects*

Description

Add together geo_list or json objects

Usage

```
## S3 method for class 'geo_list'
x1 + x2

## S3 method for class 'json'
x1 + x2
```

Arguments

x1	An object of class geo_list or json
x2	A component to add to x1, of class geo_list or json

Details

If the first object is an object of class geo_list, you can add another object of class geo_list or of class json, and will result in a geo_list object.

If the first object is an object of class json, you can add another object of class json or of class geo_list, and will result in a json object.

See Also

geojson_list, geojson_json

Examples

```
## Not run:
# geo_list + geo_list
## Note: geo_list is the output type from geojson_list, it's just a list with
## a class attached so we know it's geojson :)
vec <- c(-99.74,32.45)
a <- geojson_list(vec)
vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0), c(100.0,0.0))
b <- geojson_list(vecs, geometry="polygon")
a + b

# json + json
c <- geojson_json(c(-99.74,32.45))
vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0), c(100.0,0.0))
d <- geojson_json(vecs, geometry="polygon")
c + d
(c + d) %>% pretty

## End(Not run)
```

geojsonio **I/O for GeoJSON**

Description

Convert various data formats to/from GeoJSON or TopoJSON. This package focuses mostly on converting lists, data.frame's, numeric, SpatialPolygons, SpatialPolygonsDataFrame, and more to GeoJSON with the help of `rgdal` and friends. You can currently read TopoJSON - writing TopoJSON will come in a future version of this package.

Package organization

The core functions in this package are organized first around what you're working with or want to get, GeoJSON or TopoJSON, then convert to or read from various formats:

- `geojson_list` - convert to GeoJSON as R list format
- `geojson_json` - convert to GeoJSON as json
- `geojson_sp` - convert to a spatial object from `geojson_list` or `geojson_json`
- `geojson_read` / `topojson_read` - read a GeoJSON/TopoJSON file from file path or URL
- `geojson_write` / `topojson_write` - write a GeoJSON file locally (TopoJSON coming later)

Other interesting functions:

- `map_gist` - Create a GitHub gist (renders as an interactive map)
- `map_leaf` - Create a local interactive map using the `leaflet` package
- `lint` - Checks validity of GeoJSON using the Javascript library geojsonhint. See also `geojsonio-deprecated`
- `validate` - Checks validity of GeoJSON using the web service at http://geojsonlint.com/. See also `geojsonio-deprecated`
- `geo2topo` - Convert GeoJSON to TopoJSON
- `topo2geo` - Convert TopoJSON to GeoJSON

All of the above functions have methods for various classes, including `numeric` vectors, `data.frame`, `list`, `SpatialPolygons`, `SpatialLines`, `SpatialPoints`, and many more - which will try to do the right thing based on the data you give as input.

Author(s)

Scott Chamberlain <myrmecocystus@gmail.com>

Andy Teucher <andy.teucher@gmail.com>

geojsonio-deprecated *Deprecated functions in geojsonio*

Description

- lint: In the next version this function will be removed, defunct. See `geojsonlint::geojson_hint`
- validate: In the next version this function will be removed, defunct. See `geojsonlint::geojson_lint`

geojson_json *Convert many input types with spatial data to geojson specified as a json string*

Description

Convert many input types with spatial data to geojson specified as a json string

Usage

```
geojson_json(input, lat = NULL, lon = NULL, group = NULL,
  geometry = "point", type = "FeatureCollection", convert_wgs84 = FALSE,
  crs = NULL, ...)
```

Arguments

input	Input list, data.frame, spatial class, or sf class. Inputs can also be dplyr `tbl_df` class since it inherits from `data.frame`.
lat	(character) Latitude name. The default is `NULL`, and we attempt to guess.
lon	(character) Longitude name. The default is `NULL`, and we attempt to guess.
group	(character) A grouping variable to perform grouping for polygons - doesn't apply for points
geometry	(character) One of point (Default) or polygon.
type	(character)The type of collection. One of FeatureCollection (default) or GeometryCollection.
convert_wgs84	Should the input be converted to the standard coordinate reference system defined for GeoJSON (geographic coordinate reference system, using the WGS84 datum, with longitude and latitude units of decimal degrees; EPSG: 4326). Default is `FALSE` though this may change in a future package version. This will only work for `sf` or `Spatial` objects with a CRS already defined. If one is not defined but you know what it is, you may define it in the `crs` argument below.
crs	The CRS of the input if it is not already defined. This can be an epsg code as a four or five digit integer or a valid proj4 string. This argument will be ignored if `convert_wgs84` is `FALSE` or the object already has a CRS.
...	Further args passed on to internal functions. For Spatial* classes, it is passed through to `writeOGR`. For sf classes, data.frames, lists, numerics, and geo_lists, it is passed through to `toJSON`.

Details

This function creates a geojson structure as a json character string; it does not write a file using rgdal - see geojson_write for that.

Note that all sp class objects will output as FeatureCollection objects, while other classes (numeric, list, data.frame) can be output as FeatureCollection or GeometryCollection objects. We're working on allowing GeometryCollection option for sp class objects.

Also note that with sp classes we do make a round-trip, using writeOGR to write GeoJSON to disk, then read it back in. This is fast and we don't have to think about it too much, but this disk round-trip is not ideal.

For sf classes (sf, sfc, sfg), the following conversions are made:

- sfg: the appropriate geometry Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, G
- sfc: GeometryCollection, unless the sfc is length 1, then the geometry as above
- sf: FeatureCollection

Value

An object of class geo_json (and json)

Examples

```
## Not run:
# From a numeric vector of length 2, making a point type
geojson_json(c(-99.74,32.45), pretty=TRUE)
geojson_json(c(-99.74,32.45), type = "GeometryCollection", pretty=TRUE)

## polygon type
### this requires numeric class input, so inputting a list will dispatch on the list method
poly <- c(c(-114.345703125,39.436192999314095),
          c(-114.345703125,43.45291889355468),
          c(-106.61132812499999,43.45291889355468),
          c(-106.61132812499999,39.436192999314095),
          c(-114.345703125,39.436192999314095))
geojson_json(poly, geometry = "polygon", pretty=TRUE)

# Lists
## From a list of numeric vectors to a polygon
vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0), c(100.0,0.0))
geojson_json(vecs, geometry="polygon", pretty=TRUE)

## from a named list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
geojson_json(mylist, lat='latitude', lon='longitude')

# From a data.frame to points
geojson_json(us_cities[1:2,], lat='lat', lon='long', pretty=TRUE)
geojson_json(us_cities[1:2,], lat='lat', lon='long',
   type="GeometryCollection", pretty=TRUE)
```

```
# from data.frame to polygons
head(states)
## make list for input to e.g., rMaps
geojson_json(states[1:351, ], lat='lat', lon='long', geometry="polygon", group='group')

# from a geo_list
a <- geojson_list(us_cities[1:2,], lat='lat', lon='long')
geojson_json(a)

# sp classes

## From SpatialPolygons class
library('sp')
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)
geojson_json(sp_poly)
geojson_json(sp_poly, pretty=TRUE)

## Another SpatialPolygons
library("sp")
library("rgeos")
pt <- SpatialPoints(coordinates(list(x = 0, y = 0)), CRS("+proj=longlat +datum=WGS84"))
## transfrom to web mercator becuase geos needs project coords
crs <- gsub("\n", "", paste0("+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0
   +y_0=0 +k=1.0 +units=m +nadgrids=@null +wktext +no_defs", collapse = ""))
pt <- spTransform(pt, CRS(crs))
## buffer
pt <- gBuffer(pt, width = 100)
pt <- spTransform(pt, CRS("+proj=longlat +datum=WGS84"))
geojson_json(pt)

## data.frame to geojson
geojson_write(us_cities[1:2,], lat='lat', lon='long') %>% as.json

# From SpatialPoints class
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPoints(cbind(x,y))
geojson_json(s)

## From SpatialPointsDataFrame class
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
geojson_json(s)

## From SpatialLines class
library("sp")
c1 <- cbind(c(1,2,3), c(3,2,2))
c2 <- cbind(c1[,1]+.05,c1[,2]+.05)
c3 <- cbind(c(1,2,3),c(1,1.5,1))
```

```
L1 <- Line(c1)
L2 <- Line(c2)
L3 <- Line(c3)
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
sl12 <- SpatialLines(list(Ls1, Ls2))
geojson_json(sl1)
geojson_json(sl12)

## From SpatialLinesDataFrame class
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
geojson_json(sldf)
geojson_json(sldf, pretty=TRUE)

## From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
geojson_json(y)

## From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
geojson_json(sgdf)

# From SpatialRings
library("rgeos")
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
geojson_json(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
geojson_json(r1r2df)

# From SpatialPixels
library("sp")
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
geojson_json(pixels)

# From SpatialPixelsDataFrame
library("sp")
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
geojson_json(pixelsdf)
```

```
# From SpatialCollections
library("sp")
library("rgeos")
pts <- SpatialPoints(cbind(c(1,2,3,4,5), c(3,2,5,1,4)))
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")
poly <- SpatialPolygons(list(poly1, poly2), 1:2)
dat <- SpatialCollections(pts, polygons = poly)
geojson_json(dat)

# From sf classes:
if (require(sf)) {
## sfg (a single simple features geometry)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  poly <- rbind(c(1,1), c(1,2), c(2,2), c(1,1))
  poly_sfg <-st_polygon(list(p1))
  geojson_json(poly_sfg)

## sfc (a collection of geometries)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  p2 <- rbind(c(5,5), c(5,6), c(4,5), c(5,5))
  poly_sfc <- st_sfc(st_polygon(list(p1)), st_polygon(list(p2)))
  geojson_json(poly_sfc)

## sf (collection of geometries with attributes)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  p2 <- rbind(c(5,5), c(5,6), c(4,5), c(5,5))
  poly_sfc <- st_sfc(st_polygon(list(p1)), st_polygon(list(p2)))
  poly_sf <- st_sf(foo = c("a", "b"), bar = 1:2, poly_sfc)
  geojson_json(poly_sf)
}

## Pretty print a json string
geojson_json(c(-99.74,32.45))
geojson_json(c(-99.74,32.45)) %>% pretty

## End(Not run)
```

`geojson_list` *Convert many input types with spatial data to geojson specified as a list*

Description

Convert many input types with spatial data to geojson specified as a list

Usage

```
geojson_list(input, lat = NULL, lon = NULL, group = NULL,
```

```
geometry = "point", type = "FeatureCollection", convert_wgs84 = FALSE,
crs = NULL, ...)
```

Arguments

input	Input list, data.frame, spatial class, or sf class. Inputs can also be dplyr tbl_df class since it inherits from data.frame.
lat	(character) Latitude name. The default is NULL, and we attempt to guess.
lon	(character) Longitude name. The default is NULL, and we attempt to guess.
group	(character) A grouping variable to perform grouping for polygons - doesn't apply for points
geometry	(character) One of point (Default) or polygon.
type	(character) The type of collection. One of FeatureCollection (default) or GeometryCollection.
convert_wgs84	Should the input be converted to the standard coordinate reference system defined for GeoJSON (geographic coordinate reference system, using the WGS84 datum, with longitude and latitude units of decimal degrees; EPSG: 4326). Default is FALSE though this may change in a future package version. This will only work for sf or Spatial objects with a CRS already defined. If one is not defined but you know what it is, you may define it in the crs argument below.
crs	The CRS of the input if it is not already defined. This can be an epsg code as a four or five digit integer or a valid proj4 string. This argument will be ignored if convert_wgs84 is FALSE or the object already has a CRS.
...	Ignored

Details

This function creates a geojson structure as an R list; it does not write a file using rgdal - see geojson_write for that.

Note that all sp class objects will output as FeatureCollection objects, while other classes (numeric, list, data.frame) can be output as FeatureCollection or GeometryCollection objects. We're working on allowing GeometryCollection option for sp class objects.

Also note that with sp classes we do make a round-trip, using writeOGR to write GeoJSON to disk, then read it back in. This is fast and we don't have to think about it too much, but this disk round-trip is not ideal.

For sf classes (sf, sfc, sfg), the following conversions are made:

- sfg: the appropriate geometry Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, G
- sfc: GeometryCollection, unless the sfc is length 1, then the geometry as above
- sf: FeatureCollection

For list and data.frame objects, you don't have to pass in lat and lon parameters if they are named appropriately (e.g., lat/latitude, lon/long/longitude), as they will be auto-detected. If they can not be found, the function will stop and warn you to specify the parameters specifically.

Examples

```
## Not run:
# From a numeric vector of length 2 to a point
vec <- c(-99.74,32.45)
geojson_list(vec)

# Lists
## From a list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
geojson_list(mylist)

## From a list of numeric vectors to a polygon
vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0), c(100.0,0.0))
geojson_list(vecs, geometry="polygon")

# from data.frame to points
(res <- geojson_list(us_cities[1:2,], lat='lat', lon='long'))
as.json(res)
## guess lat/long columns
geojson_list(us_cities[1:2,])
geojson_list(states[1:3,])
geojson_list(states[1:351,], geometry="polygon", group='group')
geojson_list(canada_cities[1:30,])
## a data.frame with columsn not named appropriately, but you can specify them
# dat <- data.frame(a = c(31, 41), b = c(-120, -110))
# geojson_list(dat)
# geojson_list(dat, lat="a", lon="b")

# from data.frame to polygons
head(states)
geojson_list(states[1:351, ], lat='lat', lon='long', geometry="polygon", group='group')

# From SpatialPolygons class
library('sp')
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)
geojson_list(sp_poly)

# From SpatialPolygonsDataFrame class
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
geojson_list(input = sp_polydf)

# From SpatialPoints class
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPoints(cbind(x,y))
geojson_list(s)
```

```
# From SpatialPointsDataFrame class
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
geojson_list(s)

# From SpatialLines class
library("sp")
c1 <- cbind(c(1,2,3), c(3,2,2))
c2 <- cbind(c1[,1]+.05,c1[,2]+.05)
c3 <- cbind(c(1,2,3),c(1,1.5,1))
L1 <- Line(c1)
L2 <- Line(c2)
L3 <- Line(c3)
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
sl12 <- SpatialLines(list(Ls1, Ls2))
geojson_list(sl1)
geojson_list(sl12)
as.json(geojson_list(sl12))
as.json(geojson_list(sl12), pretty=TRUE)

# From SpatialLinesDataFrame class
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
geojson_list(sldf)
as.json(geojson_list(sldf))
as.json(geojson_list(sldf), pretty=TRUE)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
geojson_list(y)

# From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
geojson_list(sgdf)

# From SpatialRings
library("rgeos")
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
geojson_list(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
geojson_list(r1r2df)
```

```
# From SpatialPixels
library("sp")
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
geojson_list(pixels)

# From SpatialPixelsDataFrame
library("sp")
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
geojson_list(pixelsdf)

# From SpatialCollections
library("sp")
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")
poly <- SpatialPolygons(list(poly1, poly2), 1:2)
coordinates(us_cities) <- ~long+lat
dat <- SpatialCollections(points = us_cities, polygons = poly)
out <- geojson_list(dat)
out$SpatialPoints
out$SpatialPolygons

## End(Not run)

# From sf classes:
if (require(sf)) {
## sfg (a single simple features geometry)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  poly <- rbind(c(1,1), c(1,2), c(2,2), c(1,1))
  poly_sfg <-st_polygon(list(p1))
  geojson_list(poly_sfg)

## sfc (a collection of geometries)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  p2 <- rbind(c(5,5), c(5,6), c(4,5), c(5,5))
  poly_sfc <- st_sfc(st_polygon(list(p1)), st_polygon(list(p2)))
  geojson_list(poly_sfc)

## sf (collection of geometries with attributes)
  p1 <- rbind(c(0,0), c(1,0), c(3,2), c(2,4), c(1,4), c(0,0))
  p2 <- rbind(c(5,5), c(5,6), c(4,5), c(5,5))
  poly_sfc <- st_sfc(st_polygon(list(p1)), st_polygon(list(p2)))
  poly_sf <- st_sf(foo = c("a", "b"), bar = 1:2, poly_sfc)
  geojson_list(poly_sf)
}
```

`geojson_read` *Read geojson or other formats from a local file or a URL*

Description

Read geojson or other formats from a local file or a URL

Usage

```
geojson_read(x, method = "web", parse = FALSE, what = "list", ...)
```

Arguments

x	(character) Path to a local file or a URL.
method	(character) One of "web" (default) or "local". Matches on partial strings. This parameter determines how the data is read. "web" means we use the Ogre web service, and "local" means we use **rgdal**. See Details fore more.
parse	(logical) To parse geojson to data.frame like structures if possible. Default: FALSE
what	(character) What to return. One of "list" or "sp" (for Spatial class). Default: "list". If "sp" chosen, forced to method="local".
...	Additional parameters passed to readOGR

Details

Uses file_to_geojson internally to give back geojson, and other helper functions when returning spatial classes.

This function supports various geospatial file formats from a URL, as well as local kml, shp, and geojson file formats.

Value

various, depending on what's chosen in what parameter

Method parameter

The web option uses the Ogre web API. Ogre currently has an output size limit of 15MB. See here http://ogre.adc4gis.com/ for info on the Ogre web API. The local option uses the function writeOGR from the package rgdal.

Ogre

Note that for Shapefiles, GML, MapInfo, and VRT, you need to send zip files to Ogre. For other file types (.bna, .csv, .dgn, .dxf, .gxt, .txt, .json, .geojson, .rss, .georss, .xml, .gmt, .kml, .kmz) you send the actual file with that file extension.

Linting GeoJSON

If you're having trouble rendering GeoJSON files, ensure you have a valid GeoJSON file by running it through the package **geojsonlint**, which has a variety of different GeoJSON linters.

See Also

topojson_read, geojson_write

Examples

```
## Not run:
# From a file
file <- system.file("examples", "california.geojson", package = "geojsonio")
(out <- geojson_read(file))

# From a URL
url <- "https://raw.githubusercontent.com/glynnbird/usstatesgeojson/master/california.geojson"
geojson_read(url, method = "local")

# Use as.location first if you want
geojson_read(as.location(file))

# use jsonlite to parse to data.frame structures where possible
geojson_read(url, method = "local", parse = TRUE)

# output a SpatialClass object
## read kml
file <- system.file("examples", "norway_maple.kml", package = "geojsonio")
geojson_read(as.location(file), what = "sp")
## read geojson
file <- system.file("examples", "california.geojson", package = "geojsonio")
geojson_read(as.location(file), what = "sp")
## read geojson from a url
url <- "https://raw.githubusercontent.com/glynnbird/usstatesgeojson/master/california.geojson"
geojson_read(url, what = "sp")
## read from a shape file
file <- system.file("examples", "bison.zip", package = "geojsonio")
dir <- tempdir()
unzip(file, exdir = dir)
shpfile <- list.files(dir, pattern = ".shp", full.names = TRUE)
geojson_read(shpfile, what = "sp")

x <- "https://raw.githubusercontent.com/johan/world.geo.json/master/countries.geo.json"
geojson_read(x, method = "local", what = "sp")
geojson_read(x, method = "local", what = "list")

utils::download.file(x, destfile = basename(x))
geojson_read(basename(x), method = "local", what = "sp")

# doesn't work right now
## file <- system.file("examples", "feature_collection.geojson",
##   package = "geojsonio")
## geojson_read(file, what = "sp")

## End(Not run)
```

`geojson_sp` *Convert output of geojson_list or geojson_json to spatial classes*

Description

Convert output of geojson_list or geojson_json to spatial classes

Usage

```
geojson_sp(x, disambiguateFIDs = TRUE, ...)
```

Arguments

`x`	Object of class `geo_list` or `geo_json`
`disambiguateFIDs`	(logical) default `FALSE`, if `TRUE`, and FID values are not unique, they will be set to unique values 1:N for N features; problem observed in GML files
`...`	Further args passed on to `readOGR`

Details

The spatial class object returned will depend on the input GeoJSON. Sometimes you will get back a `SpatialPoints` class, and sometimes a `SpatialPolygonsDataFrame` class, etc., depending on what the structure of the GeoJSON.

The reading and writing of the CRS to/from geojson is inconsistent. You can directly set the CRS by passing a valid PROJ4 string to the `p4s` argument in `readOGR`.

Value

A spatial class object, see Details.

Examples

```
## Not run:
library(sp)

# geo_list ------------------
## From a numeric vector of length 2 to a point
vec <- c(-99.74,32.45)
geojson_list(vec) %>% geojson_sp

## Lists
## From a list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
geojson_list(mylist) %>% geojson_sp
geojson_list(mylist) %>% geojson_sp %>% plot
```

```
## From a list of numeric vectors to a polygon
vecs <- list(c(100.0,0.0), c(101.0,0.0), c(101.0,1.0), c(100.0,1.0), c(100.0,0.0))
geojson_list(vecs, geometry="polygon") %>% geojson_sp
geojson_list(vecs, geometry="polygon") %>% geojson_sp %>% plot

# geo_json ------------------
## from point
geojson_json(c(-99.74,32.45)) %>% geojson_sp
geojson_json(c(-99.74,32.45)) %>% geojson_sp %>% plot

# from featurecollectino of points
geojson_json(us_cities[1:2,], lat='lat', lon='long') %>% geojson_sp
geojson_json(us_cities[1:2,], lat='lat', lon='long') %>% geojson_sp %>% plot

# Set the CRS via the p4s argument
geojson_json(us_cities[1:2,], lat='lat', lon='long') %>% geojson_sp(p4s = "+init=epsg:4326")

# json ----------------------
x <- geojson_json(us_cities[1:2,], lat='lat', lon='long')
geojson_sp(x)

## End(Not run)
```

`geojson_style` *Style a data.frame or list prior to converting to geojson*

Description

This helps you add styling following the Simplestyle Spec. See Details

Usage

```
geojson_style(input, var = NULL, var_col = NULL, var_sym = NULL,
  var_size = NULL, var_stroke = NULL, var_stroke_width = NULL,
  var_stroke_opacity = NULL, var_fill = NULL, var_fill_opacity = NULL,
  color = NULL, symbol = NULL, size = NULL, stroke = NULL,
  stroke_width = NULL, stroke_opacity = NULL, fill = NULL,
  fill_opacity = NULL)
```

Arguments

`input`	A data.frame or a list
`var`	(character) A single variable to map colors, symbols, and/or sizes to.
`var_col`	(character) A single variable to map colors to.
`var_sym`	(character) A single variable to map symbols to.
`var_size`	(character) A single variable to map size to.
`var_stroke`	(character) A single variable to map stroke to.

var_stroke_width
: (character) A single variable to map stroke width to.

var_stroke_opacity
: (character) A single variable to map stroke opacity to.

var_fill
: (character) A single variable to map fill to.

var_fill_opacity
: (character) A single variable to map fill opacity to.

color
: (character) Valid RGB hex color. Assigned to the variable `marker-color`

symbol
: (character) An icon ID from the Maki project http://www.mapbox.com/maki/ or a single alphanumeric character (a-z or 0-9). Assigned to the variable `marker-symbol`

size
: (character) One of 'small', 'medium', or 'large'. Assigned to the variable `marker-size`

stroke
: (character) Color of a polygon edge or line (RGB). Assigned to the variable `stroke`

stroke_width
: (numeric) Width of a polygon edge or line (number > 0). Assigned to the variable `stroke-width`

stroke_opacity
: (numeric) Opacity of a polygon edge or line (0.0 - 1.0). Assigned to the variable `stroke-opacity`

fill
: (character) The color of the interior of a polygon (GRB). Assigned to the variable `fill`

fill_opacity
: (character) The opacity of the interior of a polygon (0.0-1.0). Assigned to the variable `fill-opacity`

Details

The parameters color, symbol, size, stroke, stroke_width, stroke_opacity, fill, and fill_opacity expect a vector of size 1 (recycled), or exact length of vector being applied to in your input data.

This function helps add styling data to a list or data.frame following the Simplestyle Spec (https://github.com/mapbox/simplestyle-spec/tree/master/1.1.0), used by MapBox and GitHub Gists (that renders geoJSON/topoJSON as interactive maps).

There are a few other style variables, but deal with polygons

GitHub has a nice help article on geoJSON files https://help.github.com/articles/mapping-geojson-files-on-git

Please do get in touch if you think anything should change in this function.

Examples

```
## Not run:
## from data.frames - point data
library("RColorBrewer")
smalluscities <-
   subset(us_cities, country.etc == 'OR' | country.etc == 'NY' | country.etc == 'CA')

### Just color
geojson_style(smalluscities, var = 'country.etc',
   color=brewer.pal(length(unique(smalluscities$country.etc)), "Blues"))
### Just size
```

```
geojson_style(smalluscities, var = 'country.etc', size=c('small','medium','large'))
### Color and size
geojson_style(smalluscities, var = 'country.etc',
   color=brewer.pal(length(unique(smalluscities$country.etc)), "Blues"),
   size=c('small','medium','large'))

## from lists - point data
mylist <- list(list(latitude=30, longitude=120, state="US"),
               list(latitude=32, longitude=130, state="OR"),
               list(latitude=38, longitude=125, state="NY"),
               list(latitude=40, longitude=128, state="VT"))
# just color
geojson_style(mylist, var = 'state',
   color=brewer.pal(length(unique(sapply(mylist, '[[', 'state'))), "Blues"))
# color and size
geojson_style(mylist, var = 'state',
   color=brewer.pal(length(unique(sapply(mylist, '[[', 'state'))), "Blues"),
   size=c('small','medium','large','large'))
# color, size, and symbol
geojson_style(mylist, var = 'state',
   color=brewer.pal(length(unique(sapply(mylist, '[[', 'state'))), "Blues"),
   size=c('small','medium','large','large'),
   symbol="zoo")
# stroke, fill
geojson_style(mylist, var = 'state',
   stroke=brewer.pal(length(unique(sapply(mylist, '[[', 'state'))), "Blues"),
   fill=brewer.pal(length(unique(sapply(mylist, '[[', 'state'))), "Greens"))

# from data.frame - polygon data
smallstates <- states[states$group %in% 1:3, ]
head(smallstates)
geojson_style(smallstates, var = 'group',
   stroke = brewer.pal(length(unique(smallstates$group)), "Blues"),
   stroke_width = c(1, 2, 3),
   fill = brewer.pal(length(unique(smallstates$group)), "Greens"))

## End(Not run)
```

`geojson_write` *Convert many input types with spatial data to a geojson file*

Description

Convert many input types with spatial data to a geojson file

Usage

```
geojson_write(input, lat = NULL, lon = NULL, geometry = "point",
  group = NULL, file = "myfile.geojson", overwrite = TRUE,
  precision = NULL, convert_wgs84 = FALSE, crs = NULL, ...)
```

Arguments

input	Input list, data.frame, spatial class, or sf class. Inputs can also be dplyr tbl_df class since it inherits from data.frame.
lat	(character) Latitude name. The default is NULL, and we attempt to guess.
lon	(character) Longitude name. The default is NULL, and we attempt to guess.
geometry	(character) One of point (Default) or polygon.
group	(character) A grouping variable to perform grouping for polygons - doesn't apply for points
file	(character) A path and file name (e.g., myfile), with the .geojson file extension. Default writes to current working directory.
overwrite	(logical) Overwrite the file given in file with input. Default: TRUE. If this param is FALSE and the file already exists, we stop with error message.
precision	desired number of decimal places for the coordinates in the geojson file. Using fewer decimal places can decrease file sizes (at the cost of precision).
convert_wgs84	Should the input be converted to the standard coordinate reference system defined for GeoJSON (geographic coordinate reference system, using the WGS84 datum, with longitude and latitude units of decimal degrees; EPSG: 4326). Default is FALSE though this may change in a future package version. This will only work for sf or Spatial objects with a CRS already defined. If one is not defined but you know what it is, you may define it in the crs argument below.
crs	The CRS of the input if it is not already defined. This can be an epsg code as a four or five digit integer or a valid proj4 string. This argument will be ignored if convert_wgs84 is FALSE or the object already has a CRS.
...	Further args passed on to internal functions. For Spatial* classes, data.frames, regular lists, and numerics, it is passed through to writeOGR. For sf classes, geo_lists and json classes, it is passed through to toJSON.

Value

A geojson_write class, with two elements:

- path: path to the file with the GeoJSON
- type: type of object the GeoJSON came from, e.g., SpatialPoints

See Also

geojson_list, geojson_json, topojson_write

Examples

```
## Not run:
# From a data.frame
## to points
geojson_write(us_cities[1:2,], lat='lat', lon='long')

## to polygons
```

```
head(states)
geojson_write(input=states, lat='lat', lon='long',
  geometry='polygon', group="group")

## partial states dataset to points (defaults to points)
geojson_write(input=states, lat='lat', lon='long')

## Lists
### list of numeric pairs
poly <- list(c(-114.345703125,39.436192999314095),
          c(-114.345703125,43.45291889355468),
          c(-106.61132812499999,43.45291889355468),
          c(-106.61132812499999,39.436192999314095),
          c(-114.345703125,39.436192999314095))
geojson_write(poly, geometry = "polygon")

### named list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
geojson_write(mylist)

# From a numeric vector of length 2
## Expected order is lon, lat
vec <- c(-99.74, 32.45)
geojson_write(vec)

## polygon from a series of numeric pairs
### this requires numeric class input, so inputting a list will
### dispatch on the list method
poly <- c(c(-114.345703125,39.436192999314095),
          c(-114.345703125,43.45291889355468),
          c(-106.61132812499999,43.45291889355468),
          c(-106.61132812499999,39.436192999314095),
          c(-114.345703125,39.436192999314095))
geojson_write(poly, geometry = "polygon")

# Write output of geojson_list to file
res <- geojson_list(us_cities[1:2,], lat='lat', lon='long')
class(res)
geojson_write(res)

# Write output of geojson_json to file
res <- geojson_json(us_cities[1:2,], lat='lat', lon='long')
class(res)
geojson_write(res)

# From SpatialPolygons class
library('sp')
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)
```

```
geojson_write(sp_poly)

# From SpatialPolygonsDataFrame class
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
geojson_write(input = sp_polydf)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
geojson_write(y)

# From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
geojson_write(sgdf)

# From SpatialRings
library(rgeos)
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
geojson_write(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
geojson_write(r1r2df)

# From SpatialPixels
library("sp")
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
geojson_write(pixels)

# From SpatialPixelsDataFrame
library("sp")
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
geojson_write(pixelsdf)

# From SpatialCollections
library("sp")
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")
poly <- SpatialPolygons(list(poly1, poly2), 1:2)
coordinates(us_cities) <- ~long+lat
dat <- SpatialCollections(points = us_cities, polygons = poly)
geojson_write(dat)

## End(Not run)
```

```
# From sf classes:
if (require(sf)) {
  file <- system.file("examples", "feature_collection.geojson", package = "geojsonio")
  sf_fc <- st_read(file, quiet = TRUE)
  geojson_write(sf_fc)
}
```

`lint` *Lint geojson*

Description

Lint geojson

Usage

```
lint(x, ...)
```

Arguments

`x`	Input, a geojson character string or list
`...`	Further args passed on to helper functions.

Details

This function is Deprecated - and will be removed in the next version of this package. See `geojsonio-deprecated` for more information

Examples

```
## Not run:
lint('{"type": "FooBar"}')
lint('{ "type": "FeatureCollection" }')
lint('{"type":"Point","geometry":{"type":"Point","coordinates":[-80,40]},"properties":{}}')

# From a list turned into geo_list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
x <- geojson_list(mylist)
class(x)
lint(x)

# A file
file <- system.file("examples", "zillow_or.geojson", package = "geojsonio")
lint(as.location(file))

# A URL
url <- "https://raw.githubusercontent.com/glynnbird/usstatesgeojson/master/california.geojson"
lint(as.location(url))
```

```
# from json (jsonlite class)
x <- jsonlite::minify('{ "type": "FeatureCollection" }')
class(x)
lint(x)

# From SpatialPoints class
library("sp")
a <- c(1,2,3,4,5)
b <- c(3,2,5,1,4)
(x <- SpatialPoints(cbind(a,b)))
class(x)
lint(x)

# From a data.frame
## need to specify what columns are lat and long with a data.frame
lint(us_cities[1:2,], lat='lat', lon='long')

# From numeric
vec <- c(32.45,-99.74)
lint(vec)

# From a list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
lint(mylist)

## End(Not run)
```

`map_gist` *Publish an interactive map as a GitHub gist*

Description

There are two ways to authorize to work with your GitHub account:

- PAT - Generate a personal access token (PAT) at https://help.github.com/articles/creating-an-access-token-for-command-line-use and record it in the GITHUB_PAT envar in your `.Renviron` file.
- Interactive - Interactively login into your GitHub account and authorise with OAuth.

Using the PAT method is recommended.

Using the gist_auth() function you can authenticate separately first, or if you're not authenticated, this function will run internally with each function call. If you have a PAT, that will be used, if not, OAuth will be used.

Usage

```
map_gist(input, lat = "lat", lon = "long", geometry = "point",
  group = NULL, type = "FeatureCollection", file = "myfile.geojson",
  description = "", public = TRUE, browse = TRUE, ...)
```

Arguments

input	Input object
lat	Name of latitude variable
lon	Name of longitude variable
geometry	(character) Are polygons in the object
group	(character) A grouping variable to perform grouping for polygons - doesn't apply for points
type	(character) One of FeatureCollection or GeometryCollection
file	File name to use to put up as the gist file
description	Description for the GitHub gist, or leave to default (=no description)
public	(logical) Want gist to be public or not? Default: TRUE
browse	If TRUE (default) the map opens in your default browser.
...	Further arguments passed on to POST

Examples

```
## Not run:
# From file
file <- "myfile.geojson"
geojson_write(us_cities[1:20, ], lat='lat', lon='long', file = file)
map_gist(file=as.location(file))

# From SpatialPoints class
library("sp")
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPoints(cbind(x,y))
map_gist(s)

# from SpatialPointsDataFrame class
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
map_gist(s)

# from SpatialPolygons class
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)
map_gist(sp_poly)

# From SpatialPolygonsDataFrame class
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
map_gist(sp_poly)

# From SpatialLines class
```

```
c1 <- cbind(c(1,2,3), c(3,2,2))
c2 <- cbind(c1[,1]+.05,c1[,2]+.05)
c3 <- cbind(c(1,2,3),c(1,1.5,1))
L1 <- Line(c1)
L2 <- Line(c2)
L3 <- Line(c3)
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
sl12 <- SpatialLines(list(Ls1, Ls2))
map_gist(sl1)

# From SpatialLinesDataFrame class
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
map_gist(sldf)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
map_gist(y)

# From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
map_gist(sgdf)

# from data.frame
## to points
map_gist(us_cities)

## to polygons
head(states)
map_gist(states[1:351, ], lat='lat', lon='long', geometry="polygon", group='group')

## From a list
mylist <- list(list(lat=30, long=120, marker="red"),
               list(lat=30, long=130, marker="blue"))
map_gist(mylist, lat="lat", lon="long")

# From a numeric vector
## of length 2 to a point
vec <- c(-99.74,32.45)
map_gist(vec)

## this requires numeric class input, so inputting a list will dispatch on the list method
poly <- c(c(-114.345703125,39.436192999314095),
          c(-114.345703125,43.45291889355468),
          c(-106.61132812499999,43.45291889355468),
          c(-106.61132812499999,39.436192999314095),
```

```
          c(-114.345703125,39.436192999314095))
map_gist(poly, geometry = "polygon")

# From a json object
(x <- geojson_json(c(-99.74,32.45)))
map_gist(x)
## another example
map_gist(geojson_json(us_cities[1:10,], lat='lat', lon='long'))

# From a geo_list object
(res <- geojson_list(us_cities[1:2,], lat='lat', lon='long'))
map_gist(res)

# From SpatialPixels
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
map_gist(pixels)

# From SpatialPixelsDataFrame
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
map_gist(pixelsdf)

# From SpatialRings
library("rgeos")
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
map_gist(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
map_gist(r1r2df)

## End(Not run)
```

`map_leaf` *Make an interactive map locally*

Description

Make an interactive map locally

Usage

```
map_leaf(input, lat = NULL, lon = NULL, basemap = "Stamen.Toner", ...)
```

Arguments

input	Input object
lat	Name of latitude variable
lon	Name of longitude variable
basemap	Basemap to use. See addProviderTiles. Default: Stamen.Toner
...	Further arguments passed on to addPolygons, addMarkers, addGeoJSON, or addPolylines

Examples

```
## Not run:
# We'll need leaflet below
library("leaflet")

# From file
file <- "myfile.geojson"
geojson_write(us_cities[1:20, ], lat='lat', lon='long', file = file)
map_leaf(as.location(file))

# From SpatialPoints class
library("sp")
x <- c(1,2,3,4,20)
y <- c(3,2,5,3,4)
s <- SpatialPoints(cbind(x,y))
map_leaf(s)

# from SpatialPointsDataFrame class
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
map_leaf(s)

# from SpatialPolygons class
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)
map_leaf(sp_poly)

# From SpatialPolygonsDataFrame class
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
map_leaf(sp_poly)

# From SpatialLines class
c1 <- cbind(c(1,2,3), c(3,2,2))
c2 <- cbind(c1[,1]+.05,c1[,2]+.05)
c3 <- cbind(c(1,2,3),c(1,1.5,1))
L1 <- Line(c1)
L2 <- Line(c2)
```

```
L3 <- Line(c3)
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
sl12 <- SpatialLines(list(Ls1, Ls2))
map_leaf(sl1)
map_leaf(sl12)

# From SpatialLinesDataFrame class
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
map_leaf(sldf)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
map_leaf(y)

# From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
map_leaf(sgdf)

# from data.frame
map_leaf(us_cities)

## another example
head(states)
map_leaf(states[1:351, ])

## From a named list
mylist <- list(list(lat=30, long=120, marker="red"),
               list(lat=30, long=130, marker="blue"))
map_leaf(mylist, lat="lat", lon="long")

## From an unnamed list
poly <- list(c(-114.345703125,39.436192999314095),
             c(-114.345703125,43.45291889355468),
             c(-106.61132812499999,43.45291889355468),
             c(-106.61132812499999,39.436192999314095),
             c(-114.345703125,39.436192999314095))
map_leaf(poly)
## NOTE: Polygons from lists aren't supported yet

# From a json object
map_leaf(geojson_json(c(-99.74, 32.45)))
map_leaf(geojson_json(c(-119, 45)))
map_leaf(geojson_json(c(-99.74, 32.45)))
## another example
map_leaf(geojson_json(us_cities[1:10,], lat='lat', lon='long'))
```

```
# From a geo_list object
(res <- geojson_list(us_cities[1:2,], lat='lat', lon='long'))
map_leaf(res)

# From SpatialPixels
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
map_leaf(pixels)

# From SpatialPixelsDataFrame
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
map_leaf(pixelsdf)

# From SpatialRings
library("rgeos")
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
map_leaf(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
map_leaf(r1r2df)

# basemap toggling ------------------------
map_leaf(us_cities, basemap = "Acetate.terrain")
map_leaf(us_cities, basemap = "CartoDB.Positron")
map_leaf(us_cities, basemap = "OpenTopoMap")

# leaflet options -------------------------
map_leaf(us_cities) %>%
  addPopups(-122.327298, 47.597131, "foo bar", options = popupOptions(closeButton = FALSE))

####### not working yet
# From a numeric vector
## of length 2 to a point
## vec <- c(-99.74,32.45)
## map_leaf(vec)

## End(Not run)
```

`pretty` *Convert json input to pretty printed output*

Description

Convert json input to pretty printed output

Usage

```
pretty(x, indent = 4)
```

Arguments

x	Input, character string
indent	(integer) Number of spaces to indent

Details

Only works with json class input. This is a simple wrapper around prettify, so you can easily use that yourself.

---

projections *topojson projections and extensions*

Description

topojson projections and extensions

Usage

```
projections(proj, rotate = NULL, center = NULL, translate = NULL,
  scale = NULL, clipAngle = NULL, precision = NULL, parallels = NULL,
  clipExtent = NULL, invert = NULL)
```

Arguments

proj	Map projection name. One of albers, albersUsa, azimuthalEqualArea, azimuthalEquidistant, conicEqualArea, conicConformal, conicEquidistant, equirectangular, gnomonic, mercator, orthographic, stereographic, or transverseMercator.
rotate	If rotation is specified, sets the projection's three-axis rotation to the specified angles yaw, pitch and roll (or equivalently longitude, latitude and roll) in degrees and returns the projection. If rotation is not specified, returns the current rotation which defaults [0, 0, 0]. If the specified rotation has only two values, rather than three, the roll is assumed to be 0.
center	If center is specified, sets the projection's center to the specified location, a two-element array of longitude and latitude in degrees and returns the projection. If center is not specified, returns the current center which defaults to (0,0)
translate	If point is specified, sets the projection's translation offset to the specified two-element array [x, y] and returns the projection. If point is not specified, returns the current translation offset which defaults to [480, 250]. The translation offset determines the pixel coordinates of the projection's center. The default translation offset places (0,0) at the center of a 960x500 area.

`scale` If scale is specified, sets the projection's scale factor to the specified value and returns the projection. If scale is not specified, returns the current scale factor which defaults to 150. The scale factor corresponds linearly to the distance between projected points. However, scale factors are not consistent across projections.

`clipAngle` If angle is specified, sets the projection's clipping circle radius to the specified angle in degrees and returns the projection. If angle is null, switches to antimeridian cutting rather than small-circle clipping. If angle is not specified, returns the current clip angle which defaults to null. Small-circle clipping is independent of viewport clipping via clipExtent.

`precision` If precision is specified, sets the threshold for the projection's adaptive resampling to the specified value in pixels and returns the projection. This value corresponds to the Douglas-Peucker distance. If precision is not specified, returns the projection's current resampling precision which defaults to Math.SQRT(1/2).

`parallels` Depends on the projection used! See https://github.com/mbostock/d3/wiki/Geo-Projections#standard-projections for help

`clipExtent` If extent is specified, sets the projection's viewport clip extent to the specified bounds in pixels and returns the projection. The extent bounds are specified as an array [[x0, y0], [x1, y1]], where x0 is the left-side of the viewport, y0 is the top, x1 is the right and y1 is the bottom. If extent is null, no viewport clipping is performed. If extent is not specified, returns the current viewport clip extent which defaults to null. Viewport clipping is independent of small-circle clipping via clipAngle.

`invert` Projects backward from Cartesian coordinates (in pixels) to spherical coordinates (in degrees). Returns an array [longitude, latitude] given the input array [x, y].

Examples

```
projections(proj="albers")
projections(proj="albers", rotate='[98 + 00 / 60, -35 - 00 / 60]', scale=5700)
projections(proj="albers", scale=5700)
projections(proj="albers", translate='[55 * width / 100, 52 * height / 100]')
projections(proj="albers", clipAngle=90)
projections(proj="albers", precision=0.1)
projections(proj="albers", parallels='[30, 62]')
projections(proj="albers", clipExtent='[[105 - 87, 40], [105 + 87 + 1e-6, 82 + 1e-6]]')
projections(proj="albers", invert=60)
projections("orthographic")

## Not run:
projections("alber")

## End(Not run)
```

states *This is the same data set from the ggplot2 library*

Description

This is a data.frame with "long", "lat", "group", "order", "region", and "subregion" columns specifying polygons for each US state.

topojson_read *Read topojson from a local file or a URL*

Description

Read topojson from a local file or a URL

Usage

```
topojson_read(x, ...)
```

Arguments

x	Path to a local file or a URL.
...	Further args passed on to readOGR

Details

Returns a Spatial class (e.g., SpatialPolygonsDataFrame), but you can easily and quickly get this to geojson, see examples.

Note that this does not give you Topojson, but gives you a sp style spatial class - which you can use then to turn it into geojson as a list or json.

Value

A Spatial Class, varies depending on input

See Also

geojson_read, topojson_write

Examples

```
## Not run:
# From a file
file <- system.file("examples", "us_states.topojson", package = "geojsonio")
topojson_read(file)

# From a URL
url <- "https://raw.githubusercontent.com/shawnbot/d3-cartogram/master/data/us-states.topojson"
topojson_read(url)

# Use as.location first if you want
topojson_read(as.location(file))

# quickly convert to geojson as a list
file <- system.file("examples", "us_states.topojson", package = "geojsonio")
tmp <- topojson_read(file)
geojson_list(tmp)
geojson_json(tmp)

## End(Not run)
```

---

topojson_write *Write topojson from various inputs*

Description

Write topojson from various inputs

Usage

```
topojson_write(input, lat = NULL, lon = NULL, geometry = "point",
  group = NULL, file = "myfile.topojson", overwrite = TRUE,
  precision = NULL, convert_wgs84 = FALSE, crs = NULL, ...)
```

Arguments

input	Input list, data.frame, spatial class, or sf class. Inputs can also be dplyr tbl_df class since it inherits from data.frame.
lat	(character) Latitude name. The default is NULL, and we attempt to guess.
lon	(character) Longitude name. The default is NULL, and we attempt to guess.
geometry	(character) One of point (Default) or polygon.
group	(character) A grouping variable to perform grouping for polygons - doesn't apply for points
file	(character) A path and file name (e.g., myfile), with the .geojson file extension. Default writes to current working directory.
overwrite	(logical) Overwrite the file given in file with input. Default: TRUE. If this param is FALSE and the file already exists, we stop with error message.

precision	desired number of decimal places for the coordinates in the geojson file. Using fewer decimal places can decrease file sizes (at the cost of precision).
convert_wgs84	Should the input be converted to the standard coordinate reference system defined for GeoJSON (geographic coordinate reference system, using the WGS84 datum, with longitude and latitude units of decimal degrees; EPSG: 4326). Default is FALSE though this may change in a future package version. This will only work for sf or Spatial objects with a CRS already defined. If one is not defined but you know what it is, you may define it in the crs argument below.
crs	The CRS of the input if it is not already defined. This can be an epsg code as a four or five digit integer or a valid proj4 string. This argument will be ignored if convert_wgs84 is FALSE or the object already has a CRS.
...	Further args passed on to internal functions. For Spatial* classes, data.frames, regular lists, and numerics, it is passed through to writeOGR. For sf classes, geo_lists and json classes, it is passed through to toJSON.

Details

Under the hood we simply wrap geojson_write, then take the GeoJSON output of that operation, then convert to TopoJSON with geo2topo, then write to disk.

Unfortunately, this process requires a number of round trips to disk, so speed ups will hopefully come soon.

Value

A topojson_write class, with two elements:

- path: path to the file with the TopoJSON
- type: type of object the TopoJSON came from, e.g., SpatialPoints

See Also

geojson_write, topojson_read

Examples

```
# From a data.frame
## to points
topojson_write(us_cities[1:2,], lat='lat', lon='long')

## to polygons
head(states)
topojson_write(input=states, lat='lat', lon='long',
  geometry='polygon', group="group")

## partial states dataset to points (defaults to points)
topojson_write(input=states, lat='lat', lon='long')

## Lists
### list of numeric pairs
```

```
poly <- list(c(-114.345703125,39.436192999314095),
          c(-114.345703125,43.45291889355468),
          c(-106.61132812499999,43.45291889355468),
          c(-106.61132812499999,39.436192999314095),
          c(-114.345703125,39.436192999314095))
topojson_write(poly, geometry = "polygon")

### named list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
topojson_write(mylist)

# From a numeric vector of length 2
## Expected order is lon, lat
vec <- c(-99.74, 32.45)
topojson_write(vec)

# from TopoJSON as JSON
x <- system.file("examples/point.json", package = "geojsonio")
tj <- structure(paste0(readLines(x), collapse = ""), class = "json")
topojson_write(tj, file = "my.topojson")

# convert GeoJSON to TopoJSON, then write
x <- '{"type": "LineString", "coordinates": [ [100.0, 0.0], [101.0, 1.0] ]}'
topojson_write(geo2topo(x), file = "out.topojson")

# SpatialPoints class
library(sp)
x <- c(1,2,3,4,5)
y <- c(3,2,5,1,4)
s <- SpatialPoints(cbind(x,y))
res <- topojson_write(s, file = "out.topojson")
readLines("out.topojson")

# SpatialPointsDataFrame class
s <- SpatialPointsDataFrame(cbind(x,y), mtcars[1:5,])
topojson_write(s, file = "out.topojson")
readLines("out.topojson")

# SpatialLines class
c1 <- cbind(c(1,2,3), c(3,2,2))
c2 <- cbind(c1[,1]+.05,c1[,2]+.05)
c3 <- cbind(c(1,2,3),c(1,1.5,1))
L1 <- Line(c1)
L2 <- Line(c2)
L3 <- Line(c3)
Ls1 <- Lines(list(L1), ID = "a")
Ls2 <- Lines(list(L2, L3), ID = "b")
sl1 <- SpatialLines(list(Ls1))
sl12 <- SpatialLines(list(Ls1, Ls2))
topojson_write(sl1, file = "out.topojson")
readLines("out.topojson")
```

```
# SpatialLinesDataFrame class
dat <- data.frame(X = c("Blue", "Green"),
                  Y = c("Train", "Plane"),
                  Z = c("Road", "River"), row.names = c("a", "b"))
sldf <- SpatialLinesDataFrame(sl12, dat)
topojson_write(sldf, file = "out.topojson")
readLines("out.topojson")

# SpatialPolygons class
library('sp')
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)
res <- topojson_write(sp_poly, file = "out.topojson")
readLines(res$path)

# From SpatialPolygonsDataFrame class
sp_polydf <- as(sp_poly, "SpatialPolygonsDataFrame")
res <- topojson_write(sp_polydf, file = "out.topojson")
readLines(res$path)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
topojson_write(y)

# From SpatialGrid
x <- GridTopology(c(0,0), c(1,1), c(5,5))
y <- SpatialGrid(x)
res <- topojson_write(y)
readLines(res$path)

# From SpatialGridDataFrame
sgdim <- c(3,4)
sg <- SpatialGrid(GridTopology(rep(0,2), rep(10,2), sgdim))
sgdf <- SpatialGridDataFrame(sg, data.frame(val = 1:12))
topojson_write(sgdf)

# From SpatialPixels
library("sp")
pixels <- suppressWarnings(SpatialPixels(SpatialPoints(us_cities[c("long", "lat")])))
summary(pixels)
topojson_write(pixels)

# From SpatialPixelsDataFrame
library("sp")
pixelsdf <- suppressWarnings(
 SpatialPixelsDataFrame(points = canada_cities[c("long", "lat")], data = canada_cities)
)
topojson_write(pixelsdf)
```

```
# From SpatialRings
library(rgeos)
r1 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="1")
r2 <- Ring(cbind(x=c(1,1,2,2,1), y=c(1,2,2,1,1)), ID="2")
r1r2 <- SpatialRings(list(r1, r2))
class(r1r2)
topojson_write(r1r2)

# From SpatialRingsDataFrame
dat <- data.frame(id = c(1,2), value = 3:4)
r1r2df <- SpatialRingsDataFrame(r1r2, data = dat)
geojson_write(r1r2df)

# From SpatialCollections
library("sp")
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")
poly <- SpatialPolygons(list(poly1, poly2), 1:2)
coordinates(us_cities) <- ~long+lat
dat <- SpatialCollections(points = us_cities, polygons = poly)
topojson_write(dat)

# From sf classes:
if (require(sf)) {
  file <- system.file("examples", "feature_collection.geojson", package = "geojsonio")
  sf_fc <- st_read(file, quiet = TRUE)
  topojson_write(sf_fc)
}
```

`us_cities` *This is the same data set from the maps library, named differently*

Description

This database is of us cities of population greater than about 40,000. Also included are state capitals of any population size.

Format

A list with 6 components, namely "name", "country.etc", "pop", "lat", "long", and "capital", containing the city name, the state abbreviation, approximate population (as at January 2006), latitude, longitude and capital status indication (0 for non-capital, 1 for capital, 2 for state capital.

validate *Validate a geoJSON file, json object, list, or Spatial class.*

Description

Validate a geoJSON file, json object, list, or Spatial class.

Usage

```
validate(x, ...)
```

Arguments

x	Input list, data.frame, or spatial class. Inputs can also be dplyr `tbl_df` class since it inherits from `data.frame`.
...	Further args passed on to helper functions.

Details

Uses the web service at http://geojsonlint.com/

This function is Deprecated - and will be removed in the next version of this package. See geojsonio-deprecated for more information

Examples

```
## Not run:
# From a json character string
validate(x = '{"type": "Point", "coordinates": [-100, 80]}') # good
validate(x = '{"type": "Rhombus", "coordinates": [[1, 2], [3, 4], [5, 6]]}') # bad

# A file
file <- system.file("examples", "zillow_or.geojson", package = "geojsonio")
validate(x = as.location(file))

# A URL
url <- "https://raw.githubusercontent.com/glynnbird/usstatesgeojson/master/california.geojson"
validate(as.location(url))

# From output of geojson_list
(x <- geojson_list(us_cities[1:2,], lat='lat', lon='long'))
validate(x)

# From output of geojson_json
(x <- geojson_json(us_cities[1:2,], lat='lat', lon='long'))
validate(x)

# From a list turned into geo_list
mylist <- list(list(latitude=30, longitude=120, marker="red"),
               list(latitude=30, longitude=130, marker="blue"))
```

```
x <- geojson_list(mylist)
class(x)
validate(x)

# From SpatialPoints class
library("sp")
a <- c(1,2,3,4,5)
b <- c(3,2,5,1,4)
(x <- SpatialPoints(cbind(a,b)))
class(x)
validate(x)

## End(Not run)
```

Index