Package ‘rrapply’

March 7, 2022

Type Package

Title Revisiting Base Rapply

Version 1.2.4

Date 2022-03-06

Description The minimal 'rrapply'-package contains a single function rrapply(), providing an extended implementation of 'R'-base rapply() by allowing to recursively apply a function to elements of a nested list based on a general condition function and including the possibility to prune or aggregate nested list elements from the result. In addition, special arguments can be supplied to access the name, location, parents and siblings in the nested list of the element under evaluation. The rrapply() function builds upon rapply()'s native 'C' implementation and requires no other package dependencies.

BugReports `https://github.com/JorisChau/rrapply/issues`

URL `https://jorischau.github.io/rrapply/`,
`https://github.com/JorisChau/rrapply`

Depends R (>= 3.5)

Encoding UTF-8

License GPL-3

LazyData true

RoxygenNote 7.1.2

NeedsCompilation yes

Author Joris Chau [aut, cre]

Maintainer Joris Chau <`joris.chau@openanalytics.eu`>

Repository CRAN

Date/Publication 2022-03-07 10:40:02 UTC

R topics documented:

`pokedex` *Pokedex of Pokemon GO*

Description

A nested list containing property values of the original 151 Pokemon present in Pokemon GO. The data is available in JSON format from GitHub (credits to Gianluca Bonifazi).

Usage

```
pokedex
```

Format

A nested list containing 151 sublists with up to 17 list elements:

id integer, Identification number

num character, Pokemon number in official Pokedex

name character, Pokemon name

img character, URL to png image of the Pokemon

type character, Pokemon type

height character, Pokemon height

weight character, Pokemon weight

candy character, type of candy used to evolve Pokemon or given when transfered

candy_count integer, amount of candies required to evolve

egg character, travel distance to hatch the egg

spawn_change numeric, spawn change percentage

avg_spawns integer, number of spawns per 10.000 spawns

spawn_time character, local time at which spawns are most active

multipliers numeric, multiplier of Combat Power (CP) after evolution

weakness character, types of Pokemon this Pokemon is weak to

next_evolution list, numbers (`num`) and names (`name`) with successive evolutions

prev_evolution list, numbers (`num`) and names (`name`) with previous evolutions

Source

PokemonGO-Pokedex

`renewable_energy_by_country` *UNSD renewable energy share by country in 2016*

Description

A nested list containing renewable energy shares as a percentage in the total energy consumption per country in 2016. The dataset is publicly available at the United Nations Open SDG Data Hub.

Usage

```
renewable_energy_by_country
```

Format

The 249 countries and areas are structured as a nested list based on their geographical location according to the United Nations M49 Standard (UNSD-M49). The numeric values listed for each country or area are percentages, if no data is available the value is `NA`. Each list element contains an `"M49-code"` attribute with the UN Standard Country or Area Codes for Statistical Use (Series M, No. 49).

Source

UNSD_SDG07

`rrapply` *Reimplementation of base-R's rapply*

Description

`rrapply` is a reimplemented and extended version of `rapply` to recursively apply a function `f` to a set of elements of a list and deciding *how* the result is structured.

Usage

```
rrapply(
  object,
  condition,
  f,
  classes = "ANY",
  deflt = NULL,
 how = c("replace", "list", "unlist", "prune", "flatten", "melt", "bind", "recurse",
    "unmelt"),
  ...
)
```

Arguments

`object`	a `list`, `expression` vector, or `call` object, i.e., “list-like”.
`condition`	a condition `function` of one “principal” argument and optional special arguments `.xname`, `.xpos`, `.xparents` and/or `.xsiblings` (see ‘Details’), passing further arguments via `...`.
`f`	a `function` of one “principal” argument and optional special arguments `.xname`, `.xpos`, `.xparents` and/or `.xsiblings` (see ‘Details’), passing further arguments via `...`.
`classes`	character vector of `class` names, or `"ANY"` to match the class of any terminal node. Include `"list"` or `"data.frame"` to match the class of non-terminal nodes as well.
`deflt`	the default result (only used if `how = "list"` or `how = "unlist"`).
`how`	character string partially matching the nine possibilities given: see ‘Details’.
`...`	additional arguments passed to the call to `f` and `condition`.

Value

If `how = "unlist"`, a vector as in `rapply`. If `how = "list"`, `how = "replace"` or `how = "recurse"`, “list-like” of similar structure as `object` as in `rapply`. If `how = "prune"`, a pruned “list-like” object of similar structure as `object` with pruned list elements based on `classes` and `condition`. If `how = "flatten"`, a flattened pruned vector or list with pruned elements based on `classes` and `condition`. The result of `how = "flatten"` is coerced to vector similar to `how = "unlist"`. If `how = "melt"`, a melted data.frame containing the node paths and values of the pruned list elements based on `classes` and `condition`. If `how = "unmelt"`, a nested list with list names and values defined in the data.frame `object`.

How to structure result

In addition to `rapply`’s modes to set `how` equal to `"replace"`, `"list"` or `"unlist"`, five choices `"prune"`, `"flatten"`, `"melt"`, `"bind"`, `"unmelt"` and `"recurse"` are available. `how = "prune"` filters all list elements not subject to application of `f` from the list `object`. The original list structure is retained, similar to the non-pruned options `how = "replace"` or `how = "list"`. `how = "flatten"` is an efficient way to return a flattened unnested version of the pruned list. `how = "flatten"` uses similar coercion rules as `how = "unlist"`. `how = "melt"` returns a melted data.frame of the pruned list, each row contains the path of a single terminal node in the pruned list at depth layers `L1`, `L2`, and so on. The column `"value"` contains the possible coerced values at the terminal nodes and is equivalent to the result of `how = "flatten"`. If no list names are present, the node names in the data.frame default to the indices of the list elements `"1"`, `"2"`, etc. `how = "bind"` is used to unnest a nested list containing repeated sublists into a wide data.frame. Each repeated sublist is expanded as a single row in the data.frame and identical names between sublists are aligned as individual columns. Repeated sublists are recognized as several sublists on the same nested level that are either unnamed or have the same name for each sublist. `how = "unmelt"` is a special case that reconstructs a nested list from a melted data.frame. For this reason, `how = "unmelt"` only applies to data.frames in the same format as returned by `how = "melt"`. Internally, `how = "unmelt"` first reconstructs a nested list from the melted data.frame and second uses the same framework as `how = "replace"`. `how = "recurse"` is a specialized option that is only useful in combination with e.g.

classes = "list" to recurse further into updated “list-like” elements. This is explained in more detail below.

Condition function

Both rapply and rrapply allow to apply f to list elements of certain classes via the classes argument. rrapply generalizes this concept via an additional condition argument, which accepts any function to use as a condition or predicate to select list elements to which f is applied. Conceptually, the f function is applied to all list elements for which the condition function exactly evaluates to TRUE similar to isTRUE. If the condition function is missing, f is applied to all list elements. Since the condition function generalizes the classes argument, it is allowed to use the deflt argument together with how = "list" or how = "unlist" to set a default value to all list elements for which the condition does not evaluate to TRUE.

Correct use of ...

The principal argument of the f and condition functions evaluates to the content of the list element. Any further arguments to f and condition (besides the special arguments .xname and .xpos discussed below) supplied via the dots ... argument need to be defined as function arguments in *both* the f and condition function (if existing), even if they are not used in the function itself. See also the ‘Examples’ section.

Special arguments .xname, .xpos, .xparents and .xsiblings

The f and condition functions accept four special arguments .xname, .xpos, .xparents and .xsiblings in addition to the first principal argument. The .xname argument evaluates to the name of the list element. The .xpos argument evaluates to the position of the element in the nested list structured as an integer vector. That is, if x = list(list("y","z")), then an .xpos location of c(1,2) corresponds to the list element x[[c(1,2)]]. The .xparents argument evaluates to a vector of all parent node names in the path to the list element. The .xsiblings argument evaluates to the complete (sub)list that includes the list element as a direct child. The names .xname, .xpos, .xparents or .xsiblings need to be explicitly included as function arguments in f and condition (in addition to the principal argument). See the package vignette for example uses of these special variables.

Avoid recursing into list nodes

By default, rrapply recurses into any “list-like” element. If classes = "list", this behavior is overridden and the f function is also applied to any list element of object that satisfies condition. For expression objects, use classes = "language", classes = "expression" or classes = "pairlist" to avoid recursing into branches of the abstract syntax tree of object. If the condition or classes arguments are not satisfied for a “list-like” element, rrapply will recurse further into the sublist, apply the f function to the nodes that satisfy condition and classes, and so on. Note that this behavior can only be triggered using the classes argument and not the condition argument.

Recursive list node updating

If classes = "list" and how = "recurse", rrapply applies the f function to any list element of object that satisfies condition similar to the previous section using how = "replace", but recurses further into the *updated* list-like element after application of the f function. The primary use of how

= "recurse" in combination with classes = "list" is to recursively update for instance the names of all nodes in a nested list. Additional examples are found in the package vignette.

Avoid recursing into data.frames

If classes = "ANY" (default), rrapply recurses into all "list-like" objects equivalent to rapply. Since data.frames are "list-like" objects, the f function will descend into the individual columns of a data.frame. To avoid this behavior, set classes = "data.frame", in which case the f and condition functions are applied directly to the data.frame and not its columns. Note that this behavior can only be triggered using the classes argument and not the condition argument.

List attributes

In rapply intermediate list attributes (not located at terminal nodes) are kept when how = "replace", but are dropped when how = "list". To avoid unexpected behavior, rrapply always preserves intermediate list attributes when using how = "replace", how = "list" or how = "prune". If how = "flatten" or how = "unlist" intermediate list attributes cannot be preserved as the result is no longer a nested list.

Expressions

Call objects and expression vectors are also accepted as object argument, which are treated as nested lists based on their internal abstract syntax trees. As such, all functionality that applies to nested lists extends directly to call objects and expression vectors. If object is a call object or expression vector, how = "replace" always maintains the type of object, whereas how = "list" returns the result structured as a nested list. how = "prune", how = "flatten" and how = "melt" return the pruned abstract syntax tree as: a nested list, a flattened list and a melted data.frame respectively. This is identical to application of rrapply to the abstract syntax tree formatted as a nested list.

Note

rrapply allows the f function argument to be missing, in which case no function is applied to the list elements.

how = "unmelt" requires as input a data.frame as returned by how = "melt" with character columns to name the nested list components and a final list- or vector-column containing the values of the nested list elements.

See Also

rapply

Examples

```
# Example data

## Nested list of renewable energy (%) of total energy consumption per country in 2016
data("renewable_energy_by_country")

## Subset values for countries and areas in Oceania
```

```
renewable_oceania <- renewable_energy_by_country[["World"]]["Oceania"]

## Nested list of Pokemon properties in Pokemon GO
data("pokedex")

# List pruning and unnesting

## Drop all logical NA's while preserving list structure
na_drop_oceania <- rrapply(
  renewable_oceania,
  f = function(x) x,
  classes = "numeric",
  how = "prune"
)
str(na_drop_oceania, list.len = 3, give.attr = FALSE)

## Drop all logical NA's and return unnested list
na_drop_oceania2 <- rrapply(
  renewable_oceania,
  f = function(x) x,
  classes = "numeric",
  how = "flatten"
)

head(na_drop_oceania2, n = 10)

## Drop all logical NA's and return melted data.frame
na_drop_oceania3 <- rrapply(
  renewable_oceania,
  f = identity,
  classes = "numeric",
  how = "melt"
)

head(na_drop_oceania3)

## Reconstruct nested list from melted data.frame
na_drop_oceania4 <- rrapply(
  na_drop_oceania3,
  how = "unmelt"
)

str(na_drop_oceania4, list.len = 3, give.attr = FALSE)

## Convert nested list to wide data.frame
pokedex_wide <- rrapply(pokedex, how = "bind")

head(pokedex_wide)

# Condition function

## Drop all NA elements using condition function
na_drop_oceania3 <- rrapply(
  renewable_oceania,
  condition = Negate(is.na),
  f = function(x) x,
  how = "prune"
)
str(na_drop_oceania3, list.len = 3, give.attr = FALSE)

## Replace NA elements by a new value via the ... argument
## NB: the 'newvalue' argument should be present as function
## argument in both 'f' and 'condition', even if unused.
na_zero_oceania <- rrapply(
  renewable_oceania,
  condition = function(x, newvalue) is.na(x),
  f = function(x, newvalue) newvalue,
  newvalue = 0,
  how = "replace"
)
str(na_zero_oceania, list.len = 3, give.attr = FALSE)

## Filter all countries with values above 85%
renewable_energy_above_85 <- rrapply(
  renewable_energy_by_country,
  condition = function(x) x > 85,
  how = "prune"
)
str(renewable_energy_above_85, give.attr = FALSE)

# Special arguments .xname, .xpos, .xparents and .xsiblings

## Apply a function using the name of the node
renewable_oceania_text <- rrapply(
  renewable_oceania,
  f = function(x, .xname) sprintf("Renewable energy in %s: %.2f%%", .xname, x),
  condition = Negate(is.na),
  how = "flatten"
)
head(renewable_oceania_text, n = 10)

## Extract values based on country names
renewable_benelux <- rrapply(
  renewable_energy_by_country,
  condition = function(x, .xname) .xname %in% c("Belgium", "Netherlands", "Luxembourg"),
  how = "prune"
)
str(renewable_benelux, give.attr = FALSE)

## Filter European countries with value above 50%
renewable_europe_above_50 <- rrapply(
  renewable_energy_by_country,
  condition = function(x, .xpos) identical(.xpos[c(1, 2)], c(1L, 5L)) & x > 50,
  how = "prune"
)
str(renewable_europe_above_50, give.attr = FALSE)
```

```
## Filter European countries with value above 50%
renewable_europe_above_50 <- rrapply(
  renewable_energy_by_country,
  condition = function(x, .xparents) "Europe" %in% .xparents & x > 50,
  how = "prune"
)
str(renewable_europe_above_50, give.attr = FALSE)

## Return position of Sweden in list
(xpos_sweden <- rrapply(
  renewable_energy_by_country,
  condition = function(x, .xname) identical(.xname, "Sweden"),
  f = function(x, .xpos) .xpos,
  how = "flatten"
))
renewable_energy_by_country[[xpos_sweden$Sweden]]

## Return siblings of Sweden in list
siblings_sweden <- rrapply(
  renewable_energy_by_country,
  condition = function(x, .xsiblings) "Sweden" %in% names(.xsiblings),
  how = "flatten"
)
head(siblings_sweden, n = 10)

## Return wide Pokedex data.frame without evolutions
pokedex_small <- rrapply(
   pokedex,
   condition = function(x, .xparents) !any(grepl("evolution", .xparents)),
   how = "bind"
)

head(pokedex_small)

# Modifying list elements

## Calculate mean value of Europe
rrapply(
  renewable_energy_by_country,
  classes = "list",
  condition = function(x, .xname) .xname == "Europe",
  f = function(x) mean(unlist(x), na.rm = TRUE),
  how = "flatten"
)

## Calculate mean value for each continent
renewable_continent_summary <- rrapply(
  renewable_energy_by_country,
  classes = "list",
  condition = function(x, .xpos) length(.xpos) == 2,
  f = function(x) mean(unlist(x), na.rm = TRUE)
)
```

```
## Antarctica's value is missing
str(renewable_continent_summary, give.attr = FALSE)

# List node updating

## replace country names by M-49 codes
renewable_M49 <- rrapply(
  list(renewable_energy_by_country),
  classes = "list",
  f = function(x) {
    names(x) <- vapply(x, attr, character(1L), which = "M49-code")
    return(x)
  },
  how = "recurse"
)

str(renewable_M49[[1]], max.level = 3, list.len = 3, give.attr = FALSE)

# List attributes

## how = "list" preserves all list attributes
na_drop_oceania_attr <- rrapply(
  renewable_oceania,
  f = function(x) replace(x, is.na(x), 0),
  how = "list"
)
str(na_drop_oceania_attr, max.level = 2)

## how = "prune" also preserves list attributes
na_drop_oceania_attr2 <- rrapply(
  renewable_oceania,
  condition = Negate(is.na),
  how = "prune"
)
str(na_drop_oceania_attr2, max.level = 2)

# Expressions

## Replace logicals by integers
call_old <- quote(y <- x <- 1 + TRUE)
call_new <- rrapply(call_old,
  classes = "logical",
  f = as.numeric,
  how = "replace"
)
str(call_new)

## Update and decompose call object
call_ast <- rrapply(call_old,
  f = function(x) ifelse(is.logical(x), as.numeric(x), x),
  how = "list"
)
```

```
str(call_ast)

## Prune and decompose expression
expr <- expression(y <- x <- 1, f(g(2 * pi)))
is_new_name <- function(x) !exists(as.character(x), envir = baseenv())
expr_prune <- rrapply(expr,
  classes = "name",
  condition = is_new_name,
  how = "prune"
)
str(expr_prune)

## Prune and flatten expression
expr_flatten <- rrapply(expr,
  classes = "name",
  condition = is_new_name,
  how = "flatten"
)
str(expr_flatten)

## Prune and melt expression
expr_melt <- rrapply(expr,
  classes = "name",
  condition = is_new_name,
  f = as.character,
  how = "melt"
)
expr_melt

## Avoid recursing into call objects
rrapply(
  expr,
  classes = "language",
  condition = function(x) !any(sapply(x, is.call)),
  how = "flatten"
)
```

Index