

# {rapply}: Revisiting R-base rapply()



The minimal rapply-package contains a single function `rapply()`, providing an extended implementation of R-base's `rapply()` function. `rapply()` recursively applies a function `f` to elements of a nested list and controls how to structure the returned result.

## Function signature

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

- **object** a "list-like" object;
- **condition** a condition function for application of `f`;
- **f** a function to recursively apply to each list element;
- **classes** classes to which `f` is applied, can include "list" or "data.frame";
- **deflt** a default return value;
- **how** how to structure the result;
- **options** additional options for `how`;
- **...** additional arguments for `f` and condition functions;

## Example data

```
library(rapply)
# data: renewable energy per country 2016
# as % of total energy consumption
data("renewable_energy_by_country")
# data: pokemon properties in pokemon GO
data("pokedex")
```

## How to structure the result

### • how = "replace"

replaces elements `x` satisfying `condition` and classes by `f(x)` and maintains list structure:

```
# replace all missing values by 0
rapply(
  renewable_energy_by_country,
  condition = \(x) is.na(x),
  f = \(x) 0,
  how = "replace"
)
```

### • how = "list"

replaces elements `x` satisfying `condition` and classes by `f(x)` and others by `deflt` maintaining list structure:

```
# replace all missing values by 0
rapply(
  renewable_energy_by_country,
  condition = \(x) !is.na(x),
  deflt = 0,
  how = "list"
)
```

### • how = "unlist"

similar to `how = "list"` unlisting the returned result:

```
# replace missing values by 0 and unlist
rapply(
  renewable_energy_by_country,
  classes = "numeric",
  deflt = 0,
  how = "unlist"
)
```

### • how = "prune"

similar to `how = "replace"` pruning all elements not subject to `f`:

```
# prune all missing values and maintain
# list structure
rapply(
  renewable_energy_by_country,
  condition = \(x) !is.na(x),
  how = "prune"
)
```

### • how = "flatten"

similar to `how = "prune"` returning a flattened unnested pruned list. Coercion is the same as `how = "unlist"` (using the default options):

```
# prune all missing values and return
# flattened list
rapply(
  renewable_energy_by_country,
  condition = \(x) !is.na(x),
  how = "flatten"
)
```

### • how = "melt"

similar to `how = "prune"` returning a melted data.frame of the pruned list with columns `L1`, `L2`, ..., `value`. Each row contains the path and value of an element `x`:

```
# prune all missing values and melt list
l <- rapply(
  renewable_energy_by_country,
  condition = \(x) !is.na(x),
  how = "melt"
)
```

### • how = "unmelt"

reconstructs a nested list from a melted data.frame as returned by `how = "melt"`:

```
# unmeld data.frame back to nested list
rapply(l, how = "unmelt")
```

### • how = "bind"

similar to `how = "prune"` unnesting repeated list elements into a wide data.frame. Each repeated element expands to a single row with columns aligned by names:

```
# unnest repeated list to wide data.frame
rapply(pokedex, how = "bind")
```

```
# unnest to wide data.frame and include
# parent node names as columns L1, L2, ...
rapply(
  pokedex,
  how = "bind",
  options = list(namecols = T)
)
```

### • how = "recurse"

similar to `how = "replace"` but recurses further into modified elements satisfying `condition` and classes after application of `f`:

```
# recursively remove all list attributes
rapply(
  renewable_energy_by_country,
  f = \(x) c(x),
  classes = c("list", "ANY"),
  how = "recurse"
)
```

### • how = "names"

similar to `how = "recurse"` replacing the name of element `x` by `f(x)` instead of its content using `classes = c("list", "ANY")` by default:

```
## recursively capitalize all names in list
rapply(
  renewable_energy_by_country,
  f = \(x, .xname) toupper(.xname),
  how = "names"
)
```

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

The `f` and `condition` functions accept four special arguments in addition to the principal argument:

• **.xname** evaluates to the name of the current list element:

```
# filter list elements by name
rapply(
  renewable_energy_by_country,
  condition = \(x, .xname) .xname == "Belgium",
  how = "prune"
)
```

• **.xpos** evaluates to the position of the element in the nested list as an integer vector:

```
# return position of element in list
rapply(
  renewable_energy_by_country,
  condition = \(x, .xname) .xname == "Belgium",
  f = \(x, .xpos) .xpos,
  how = "flatten"
)
```

• **.xparents** evaluates to the vector of parent names of the current element. **.xsiblings** evaluates to the parent list containing the current element and its direct siblings:

```
# filter list elements by parent names
rapply(
  renewable_energy_by_country,
  condition = \(x, .xparents)
    "Europe" %in% .xparents,
  how = "melt"
)
```