Package: funprog (via r-universe)

September 7, 2024

Type Package

Title Functional Programming

Version 0.3.0

Description High-order functions for data manipulation : sort or group data, given one or more auxiliary functions. Functions are inspired by other pure functional programming languages ('Haskell' mainly). The package also provides built-in function operators for creating compact anonymous functions, as well as the possibility to use the 'purrr' package syntax.

License GPL-2

URL https://py_b.gitlab.io/funprog, https://gitlab.com/py_b/funprog

BugReports https://gitlab.com/py_b/funprog/-/issues

Suggests purrr (>= 0.2.3), testthat

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

Roxygen list(markdown = TRUE)

Repository https://py-b.r-universe.dev

RemoteUrl https://gitlab.com/py_b/funprog

RemoteRef HEAD

RemoteSha f4205c8e5a674ed8362e4b7cdd8dda8521414d7e

Contents

`funprog-package` *Implementation of pure functional programming languages functions*

Description

The **funprog** package implements in R some functions existing in other pure functional programming languages.

Main functions

The package provides high-order functions, for example :

- `group_if`, inspired by Haskell's `groupBy`
- `sort_by`, inspired by Haskell's `sortBy`

Helper functions

Helper functions can be used in conjunction with the main functions :

- `%on%` combines two functions into one and serves to create a predicate function to `group_if`
- `descending` is used to reverse the output of a sorting function used with `sort_by`

`purrr` syntax

If the purrr package is installed, you can use its special syntax to create very compact anonymous functions, for example `~ abs(.x - .y) > 1` instead of `function(x, y) abs(x - y) > 1`.

`descending` *Reverse a sorting function*

Description

Transform a function (typically used in `sort_by`), so that its ouput can be sorted in descending order.

Usage

```
descending(f)
```

Arguments

`f`	a function to modify.

Value

A function returning a numeric vector which, if passed to `order`, will be used to sort some data.

Examples

```
desc_abs <- descending(abs)

x <- -2:1
order(abs(x))
order(desc_abs(x))
```

`group_if` *Group vector values*

Description

Split a vector or a list into groups, given a predicate function.

Usage

```
group_if(x, predicate, na.rm = FALSE)

group_eq(x, na.rm = FALSE)
```

Arguments

`x`	a vector or a list to split into groups.
`predicate`	a binary function returning a boolean value.
`na.rm`	if x is atomic, delete missing values before grouping.

Details

`predicate` will be applied to 2 adjacent elements. If it evaluates to `TRUE`, those elements belong to the same group, otherwise they belong to different groups.

Grouping on equality is the most natural approach, therefore `group_eq` is a convenient shortcut defined as

- `group_if(x, predicate = `==`)` for an atomic vector;
- `group_if(x, predicate = identical)` for a list.

`group_if` (resp. `group_eq`) is inspired by `groupBy` (resp. `group`) in Haskell. *Note that* `group_if` *behaves a little differently : while in Haskell, the comparison is made with the first element in the group, in this R-version the comparison is made with the adjacent element.*

The operator %on% may be helpful to create a predicate with readable syntax.

Value

A list where each element is a group (flattening this list should give back the same values in the same order). Element names are kept.

Examples

```
x1 <- c(3, 4, 2, 2, 1, 1, 1, 3)
group_eq(x1)
group_if(x1, `<=`)
group_if(x1, function(x, y) abs(x - y) > 1)

x2 <- c(3, 4, 2, -2, -1, 1, 1, 3)
group_if(x2, `==` %on% abs)

x3 <- list(1:3, 1:3, 3:5, 1, 2)
group_if(x3, `==` %on% length)
```

`iterate` *Apply a function repeatedly*

Description

Apply a function to a value, then reapply the same function to the result and so on... until a condition on the result is met (or a certain number of iterations reached).

Usage

```
iterate(x, f, stop_fun = NULL, stop_n = Inf, accumulate = FALSE)
```

Arguments

`x`	initial value.
`f`	the function to apply.
`stop_fun`	a predicate (function) evaluated on the current result, which will stop the process if its result is `TRUE`. If not provided, the process will stop after `stop_n` iteration (see below).
`stop_n`	maximal number of times the function will be applied (mandatory if `stop_fun` is not defined).
`accumulate`	by default, the function returns only the last element. To get the list of all intermediate results, turn this parameter to `TRUE`.

Details

As it is a very generic function (`x` can be any type of object) and the number of computations cannot be known in advance, `iterate` can be quite inefficient (particularly if you use `accumulate = TRUE`).

Value

The last result, or the list of all results if `accumulate = TRUE`.

Examples

```
# https://en.wikipedia.org/wiki/Collatz_conjecture
syracuse <- function(x) if (x %% 2) 3 * x + 1 else x / 2
iterate(
  10,
  syracuse,
  stop_fun = function(n) n == 1,
  accumulate = TRUE
)

# https://en.wikipedia.org/wiki/H%C3%A9non_map
henon_attractor <-
  iterate(
    c(-1, 0.1),
    function(x) c(1 - 1.4 * x[1]^2 + x[2], 0.3 * x[1]),
    stop_n = 5000,
    accumulate = TRUE
  )
plot(
  sapply(henon_attractor, function(.) .[1]),
  sapply(henon_attractor, function(.) .[2]),
  pch = "."
)
```

`partition` *Partition a vector in two*

Description

Split a vector or a list in 2 groups, given a predicate function.

Usage

```
partition(x, predicate)
```

Arguments

`x`	vector or list to partition.
`predicate`	a function returning a boolean value, to apply to each element of x.

Value

A list of two elements. The first element contains elements of x satisfying the predicate, the second the rest of x. Missing values will be discarded.

Examples

```
partition(c(2, 1, 3, 4, 1, 5), function(x) x < 3)
partition(list(1:3, NA, c(1, NA, 3)), anyNA)
```

sort_by *Sort with auxiliary function*

Description

Sort a vector or a list, given one or more auxiliary functions.

Usage

```
sort_by(x, ..., method = c("auto", "shell", "radix"))
```

Arguments

x	vector or list to sort.
...	one or several functions to apply to x. Use descending for reversed order.
method	the method for ties (see order).

Details

The output of the first function will be used as first key for sorting, the output of the second function as second key, and so on... Therefore, these outputs should be sortable (i.e. atomic vectors).

sort_by is inspired by sortBy in Haskell.

Value

A vector or list containing rearranged elements of x.

See Also

order which is used for rearranging elements.

Examples

```
sort_by(-3:2, abs)
sort_by(-3:2, abs, function(x) -x)
sort_by(list(5:7, 0, 1:4), length)
sort_by(list(1:2, 3:4, 5), length, descending(sum))
```

unique_by *Unique with auxiliary function*

Description

Remove duplicate elements, given a transformation.

Usage

```
unique_by(x, f, first = TRUE)
```

Arguments

x	a vector or a list.
f	a function to apply to each element of x. This function must produce comparable results.
first	if several elements are identical after being transformed by f, keep the first. Otherwise, keep the last.

Value

An object of the same type as x. Only elements that are unique after being transformed by f are kept.

Examples

```
unique_by(-3:2, abs)
unique_by(-3:2, abs, first = FALSE)
unique_by(c(1, 2, 4, 5, 6), function(x) x %% 3)
unique_by(list(1:2, 2:3, 2:4), length)
```

%on% *Transform a binary function with a unary function*

Description

Execute the binary function f on the results of applying unary function g to two arguments x and y.

Usage

```
f %on% g
```

Arguments

f	a binary function.
g	a unary function.

Details

Formally, `%on%` is defined this way : `function(f, g) function(x, y) f(g(x), g(y))`.

f can be a function taking two arguments but also a variadic function (i.e. whose first argument is ...), which will be fed with exactly two arguments.

A typical usage of this function is in combination with function like `group_if`.

Value

A binary function. This function transforms 2 inputs (with g) and combines the outputs (with f).

Examples

```
h <- max %on% abs
h(-2, 1)
```

Index