Package 'ripe'

December 6, 2019

Title Rerun 'Magrittr' Pipelines

Version 0.1.0

Description Functions in a 'magrittr' pipeline are evaluated sequentially making it impossible to rerun them directly from the end of the pipeline, while preserving stochastic characteristics embedded within the pipeline. This package allows the user to rerun pipelines using natural 'magrittr' syntax.

Depends R (>= 3.5.0)

Imports magrittr

License MIT + file LICENSE

URL https://github.com/yonicd/ripe

BugReports https://github.com/yonicd/ripe/issues

Encoding UTF-8

LazyData true

RoxygenNote 7.0.1

Suggests testthat, knitr, rmarkdown, purrr, parallel, dplyr, covr

VignetteBuilder knitr

NeedsCompilation no

Author Jonathan Sidi [aut, cre]

Maintainer Jonathan Sidi <yonicd@gmail.com>

Repository CRAN

Date/Publication 2019-12-06 10:10:02 UTC

R topics documented:

lazy 2
ripe 2
%>% 3

Index **6**

lazy *Convert magrittr chain lazy function*

Description

Input a chain in magrittr syntax and make it a lazyeval function.

Usage

```
lazy(x)
```

Arguments

x	magrittr chain of functions

Value

function

See Also

Other ripe: `ripe()`

Examples

```
f <- stats::runif(20)%>%sample(10)%>%lazy()

f()
```

ripe *Replicate magrittr chain*

Description

Lazily replicate magrittr chain

Usage

```
ripe(x, f, ...)
```

Arguments

x	chain
f	replication function
...	arguments to pass to f

Value

object that f outputs

See Also

rerun

Other ripe: lazy()

Examples

```
stats::runif(20)%>%
 sample(4)%>%
 ripe(replicate,n=4,simplify=FALSE)

stats::runif(20)%>%
 sample(4)%>%
 ripe(lapply,X=1:4)
```

%>% *magrittr forward-pipe operator*

Description

Pipe an object forward into a function or call expression.

Details

Using %>% with unary function calls

When functions require only one argument, `x %>% f` is equivalent to `f(x)` (not exactly equivalent; see technical note below.)

Placing lhs as the first argument in rhs call

The default behavior of %>% when multiple arguments are required in the rhs call, is to place lhs as the first argument, i.e. `x %>% f(y)` is equivalent to `f(x,y)`.

Placing lhs elsewhere in rhs call

Often you will want lhs to the rhs call at another position than the first. For this purpose you can use the dot (.) as placeholder. For example, `y %>% f(x,.)` is equivalent to `f(x,y)` and `z %>% f(x,y,arg = .)` is equivalent to `f(x,y,arg = z)`.

Using the dot for secondary purposes

Often, some attribute or property of lhs is desired in the rhs call in addition to the value of lhs itself, e.g. the number of rows or columns. It is perfectly valid to use the dot placeholder several times in the rhs call, but by design the behavior is slightly different when using it inside nested function calls. In particular, if the placeholder is only used in a nested function call, lhs will also be placed as the first argument! The reason for this is that in most use-cases this produces the most readable code. For example, `iris %>% subset(1:nrow(.) %% 2 == 0)` is equivalent to `iris %>%`

`subset(.,1:nrow(.) %% 2 == 0)` but slightly more compact. It is possible to overrule this behavior by enclosing the `rhs` in braces. For example, `1:10 %>% {c(min(.),max(.))}` is equivalent to `c(min(1:10),max(1:10))`.

Using %>% with call- or function-producing `rhs`
It is possible to force evaluation of `rhs` before the piping of `lhs` takes place. This is useful when `rhs` produces the relevant call or function. To evaluate `rhs` first, enclose it in parentheses, i.e. `a %>% (function(x) x^2)`, and `1:10 %>% (call("sum"))`. Another example where this is relevant is for reference class methods which are accessed using the `$` operator, where one would do `x %>% (rc$f)`, and not `x %>% rc$f`.

Using lambda expressions with `%>%`
Each `rhs` is essentially a one-expression body of a unary function. Therefore defining lambdas in magrittr is very natural, and as the definitions of regular functions: if more than a single expression is needed one encloses the body in a pair of braces, `{ rhs }`. However, note that within braces there are no "first-argument rule": it will be exactly like writing a unary function where the argument name is "`.`" (the dot).

Using the dot-place holder as `lhs`
When the dot is used as `lhs`, the result will be a functional sequence, i.e. a function which applies the entire chain of right-hand sides in turn to its input. See the examples.

Technical notes

The magrittr pipe operators use non-standard evaluation. They capture their inputs and examines them to figure out how to proceed. First a function is produced from all of the individual right-hand side expressions, and then the result is obtained by applying this function to the left-hand side. For most purposes, one can disregard the subtle aspects of magrittr's evaluation, but some functions may capture their calling environment, and thus using the operators will not be exactly equivalent to the "standard call" without pipe-operators.

Another note is that special attention is advised when using non-magrittr operators in a pipe-chain (`+`, `-`, `$`, etc.), as operator precedence will impact how the chain is evaluated. In general it is advised to use the aliases provided by magrittr.

See Also

%<>%, %T>%, %$%

Examples

```
# Basic use:
iris %>% head

# Use with lhs as first argument
iris %>% head(10)

# Using the dot place-holder
"Ceci n'est pas une pipe" %>% gsub("une", "un", .)
```

```
# When dot is nested, lhs is still placed first:
sample(1:10) %>% paste0(LETTERS[.])

# This can be avoided:
rnorm(100) %>% {c(min(.), mean(.), max(.))} %>% floor

# Lambda expressions:
iris %>%
{
  size <- sample(1:10, size = 1)
  rbind(head(., size), tail(., size))
}

# renaming in lambdas:
iris %>%
{
  my_data <- .
  size <- sample(1:10, size = 1)
  rbind(head(my_data, size), tail(my_data, size))
}

# Building unary functions with %>%
trig_fest <- . %>% tan %>% cos %>% sin

1:10 %>% trig_fest
trig_fest(1:10)
```

Index

%<>%, *4*
%>%, 3
%T>%, *4*
%$%, *4*

lazy, 2, *3*

rerun, *3*
ripe, *2*, 2