

Package: pipeR (via r-universe)

September 4, 2024

Type Package

Title Multi-Paradigm Pipeline Implementation

Version 0.6.1.3

Author Kun Ren <ken@renkun.me>

Maintainer Kun Ren <ken@renkun.me>

Description Provides various styles of function chaining methods: Pipe operator, Pipe object, and pipeline function, each representing a distinct pipeline model yet sharing almost a common set of features: A value can be piped to the first unnamed argument of a function and to dot symbol in an enclosed expression. The syntax is designed to make the pipeline more readable and friendly to a wide range of operations.

Depends R (>= 2.15)

Date 2016-04-04

Suggests testthat

License MIT + file LICENSE

URL <https://renkun.me/pipeR>, <https://github.com/renkun-ken/pipeR>,
<https://renkun.me/pipeR-tutorial>

BugReports <https://github.com/renkun-ken/pipeR/issues>

Roxygen list(wrap = FALSE)

ByteCompile TRUE

RoxygenNote 5.0.1

Repository <https://renkun-ken.r-universe.dev>

RemoteUrl <https://github.com/renkun-ken/piper>

RemoteRef HEAD

RemoteSha b62a360a84d2cd1a2c0c51bc1f55ee03b912a363

Contents

pipeR-package	2
Pipe	3
pipeline	5
%>>%	7
Index	12

pipeR-package	<i>The pipeR package</i>
---------------	--------------------------

Description

pipeR implements various function chaining methods: %>>% operator, Pipe object, and pipeline function. Each represents a distinct pipeline model yet shares a common set of features designed to build easy-to-read/write/maintain pipelines. To learn more, please visit [pipeR Tutorial](#).

Details

pipeR package defines a set of syntax tailored for unified, intuitive piping experience. The package is designed to help organize code as a streamline that is consistent with logic and intuition.

The following example shows how traditional code can be written in different function chaining styles.

Examples

```
# Traditional code:
plot(density(sample(mtcars$mpg, size = 10000, replace = TRUE),
  kernel = "gaussian"), col = "red", main="density of mpg (bootstrap)")

# Operator-based pipeline using %>>%:
mtcars$mpg %>>%
  sample(size = 10000, replace = TRUE) %>>%
  density(kernel = "gaussian") %>>%
  plot(col = "red", main = "density of mpg (bootstrap)")

# Object-based pipeline using Pipe():
Pipe(mtcars$mpg)$
  sample(size = 10000, replace = TRUE)$
  density(kernel = "gaussian")$
  plot(col = "red", main = "density of mpg (bootstrap)")

# Argument-based pipeline using pipeline():
pipeline(mtcars$mpg,
  sample(size = 10000, replace = TRUE),
  density(kernel = "gaussian"),
  plot(col = "red", main = "density of mpg (bootstrap)"))

# Expression-based pipeline using pipeline():
```

```

pipeline({
  mtcars$mpg
  sample(size = 10000, replace = TRUE)
  density(kernel = "gaussian")
  plot(col = "red", main = "density of mpg (bootstrap)")
})

```

Pipe	<i>Create a Pipe object that stores a value and allows command chaining with \$.</i>
------	--

Description

Create a Pipe object that stores a value and allows command chaining with \$.

Usage

```
Pipe(value = NULL)
```

Arguments

value value to pipe (default is NULL)

Details

Pipe() function creates a Pipe object that provides object-like command chaining mechanism, which avoids using external operator and can be cleaner than operator-based pipeline.

Pipe() creates a Pipe object that allows using \$ to perform first-argument piping, call .() to evaluate an expression with . or symbol defined by lambda expression, for side effect, or simply extract an element from the stored value. \$value or [] ends a pipeline and extracts its final value.

The functionality of Pipe object fully covers that of the pipe operator %>>% and provides more features. For example, Pipe object supports directly subsetting \$value by [...], extracting element by [[...]], and assigning value by \$item <-, [...] <-, and [[...] <-.

A typical usage of Pipe object is to start with Pipe() and end with \$value or [].

print() and str() are implemented for Pipe object. Use header = FALSE to suppress Pipe header message in printed results. Use options(Pipe.header = FALSE) to suppress it globally.

If the Pipe object is used in more than one pipelines, a recommended usage is to name the object specially so that it is easy to distinguish the Pipe object from the value it stores. For example, it can start with p.

Value

Pipe object

Examples

```

## Not run:
# Pipe as first-argument using $
Pipe(rnorm(100))$mean()
Pipe(rnorm(100))$plot(col="red")

# Extract the value from the Pipe object using []
Pipe(rnorm(100))$c(4,5) []

# Pipe to an expression with . or symbol defined in
# lambda expression to represent the object
Pipe(rnorm(100))$.(1 + .) []
Pipe(rnorm(100))$.(x ~ 1 + x) []

# Pipe for side effect
Pipe(rnorm(100))$
  .(~ cat("number:",length(.),"\\n"))$
  summary()

Pipe(rnorm(100))$
  .(~ x ~ cat("number:",length(x),"\\n"))$
  summary()

# Assignment
Pipe(rnorm(100))$
  .(~ x)$
  mean()

Pipe(rnorm(100))$
  .(~ x <- length(.))$
  mean()

Pipe(rnorm(100))%
  .(x <- c(min(.),max(.)))$
  mean()

# Extract element with \code{.(name)}
Pipe(mtcars)$lm(formula = mpg ~ cyl + wt)$.(coefficients)

# Command chaining
Pipe(rnorm(100,mean=10))$
  log()$
  diff()$
  plot(col="red")

Pipe(rnorm(100))$
  density(kernel = "rect")$
  plot(col = "blue")

# Store an continue piping
pipe1 <- Pipe(rnorm(100,mean=10))$log()$diff()
pipe1$plot(col="red")

```

```
# Subsetting, extracting, and assigning

p <- Pipe(list(a=1,b=2))
p["a"]
p[["a"]]
p$a <- 2
p["b"] <- NULL
p[["a"]] <- 3
p[length(.)] # . = p$value

# Data manipulation with dplyr
library(dplyr)
Pipe(mtcars)$
  select(mpg,cyl,disp,hp)$
  filter(mpg <= median(mpg))$
  mutate(rmpg = mpg / max(mpg))$
  group_by(cyl)$
  do(data.frame(mean=mean(. $rmpg),median=median(. $rmpg))) []

# Graphics with ggvis
library(ggvis)
Pipe(mtcars)$
  ggvis(~ mpg, ~ wt)$
  layer_points()

# Data manipulation with rlist
library(rlist)
Pipe(list(1,2,3))$
  list.map(. + 1)$
  list.filter(. <= 5)$
  list.sort(.) []

# Lazy evaluation
p1 <- Pipe(mtcars)$
  ggvis(~ mpg, ~ wt)
p1$layer_points()
p1$layer_bars()

# Stored Pipe
f1 <- Pipe(rnorm(100))$plot
f1(col="red")
f1(col="green")

## End(Not run)
```

Description

Evaluate an expression pipeline enclosed by `{}` or a sequence of expressions as as pipeline. This functions works to chain expressions without using `%>%` operator but produce the same result.

Usage

```
pipeline(...)
```

Arguments

... Pipeline expressions. Supply multiple pipeline expressions as arguments or only an enclosed expression within `{}` as the first argument.

Details

When `pipeline(...)` is called with multiple arguments, the arguments will be regarded as pipeline expressions.

When `pipeline(...)` is called with a single argument, the argument is expected to be a block expression enclosed by `{}` in which each expression will be regarded as a pipeline expression.

The pipeline expressions will be chained sequentially by `%>%` and be evaluated to produce the same results as if using the pipe operator.

Examples

```
pipeline(1:10, sin, sum)

pipeline(1:10, plot(col = "red", type = "l"))

pipeline(mtcars,
  lm(formula = mpg ~ cyl + wt),
  summary,
  coef)

pipeline({
  mtcars
  lm(formula = mpg ~ cyl + wt)
  summary
  coef
})

pipeline({
  mtcars
  "Sample data" ? head(., 3)
  lm(formula = mpg ~ cyl + wt)
  ~ lmodel
  summary
  ? .$r.squared
  coef
})
```

```

pipeline({
  mtcars
  "estimating a linear model ..."
  lm(formula = mpg ~ cyl + wt)
  "summarizing the model ..."
  summary
})

```

%>>%

*Pipe an object forward***Description**

The %>>% operator pipes the object on the left-hand side to the right-hand side according to the syntax.

Usage

```
x %>>% expr
```

Arguments

x	object
expr	expression

Details

Pipe operator %>>% determines the piping mechanism by the syntax of the expression on the right-hand side.

%>>% supports the following syntaxes:

1. Pipe to first unnamed argument:

Whenever a function name or call with or without parameters follows the operator, the left-hand side value will be piped to the right-hand side function as the first unnamed argument.

```
x %>>% f evaluated as f(x)
```

```
x %>>% f(...) evaluated as f(x, ...)
```

```
x %>>% package::name(...) evaluated as package::name(x, ...)
```

2. Pipe to . in enclosed expression:

Whenever an expression following the operator is enclosed by {}, the expression will be evaluated with . representing the left-hand side value. It is the same with expression enclosed with () unless it contains a lambda expression or assignment expression.

```
x %>>% { expr } evaluated as { expr } given . = x
```

```
x %>>% ( expr ) evaluated as expr given . = x
```

3. Pipe by lambda expression:

A lambda expression is a formula whose left-hand side is a symbol used to represent the value being piped and right-hand side is an expression to be evaluated with the symbol.

`x %>>% (p ~ expr)` as `expr` given `p = x`

4. Pipe for side-effect:

If one only cares about the side effect (e.g. printing intermediate results, plotting graphics, assigning value to symbol) of an expression rather than its returned value, write a lambda expression that starts with `~` indicating side effect (or branching, in the sense of pipeline building).

`x %>>% (~ f(.))` evaluated as `{f(x); x}`.

`x %>>% (~ p ~ f(p))` evaluated as `{f(x); x}`

5. Pipe for assignment

Equal operator (`=`) and assignment operators (`<-` and `->`) perform assignment. This is particularly useful when one needs to save an intermediate value in the middle of a pipeline without breaking it.

Assignment as side-effect

In general, `x %>>% (~ y = ...)` is evaluated as `y <- x %>>% (...)` and returns `x`.

`x %>>% (~ y)` evaluated as `y <- x` and returns `x`, where `y` must be a symbol.

`x %>>% (~ y = f(.))` evaluated as `y <- f(x)` and returns `x`.

`x %>>% (~ y = p ~ f(p))` evaluated as `y <- f(x)` and returns `x`.

Piping with assignment

In general, `x %>>% (y = ...)` is evaluated as `y <- x %>>% (...)`.

`x %>>% (y = f(.))` evaluated as `y <- f(x)` and returns `f(x)`.

`x %>>% (y = p ~ f(p))` evaluated as `y <- f(x)` and returns `f(x)`.

The equal sign above can be interchangeably used as the assignment operator `<-`. Note that the global assignment operator `<<-` and `->>` in a pipeline also performs global assignment that is subject to side-effect outside the calling environment.

In some cases, users might need to create a group of symbols. The following code calls `assign` to dynamically determine the symbol name when its value is evaluated.

```
for (i in 1:5) rnorm(i) %>>% (assign(paste0("rnorm", i), .))
```

To avoid exporting a symbol to the calling environment, use a symbol name starting with `.` like

6. Pipe for element extraction:

If a symbol is enclosed within `()`, it tells the operator to extract element from the left-hand side value. It works with vector, list, environment and all other objects for which `[[]]` is defined. Moreover, it also works with S4 object.

`x %>>% (name)` as `x[["name"]]` when `x` is list, environment, data.frame, etc; and `x@name` when `x` is S4 object.

7. Pipe to string:

If an object is piped to a single character value, then the string will be `cat()` and starts a new paragraph. This is useful for indicating the executing process of a pipeline.

`x %>>% "print something" %>>% f(y)` will `cat("print something")` and then evaluate `f(x,y)`.

8. Pipe for questioning:

If a lambda expression start with `?`, the expression will be a side effect printing the syntax and the value of the expression. This is a light-weight version of side-effect piping and can be useful for simple inspection and debugging for pipeline operations.

`x %>>% (? expr)` will print the value of `expr` and return `x`, just like a question.

`x %>>% ("title" ? expr)` will print "title" as the question, the value of `expr`, and return `x`.

Examples

```
## Not run:
# Pipe as first-argument to a function name
rnorm(100) %>>% plot

# Pipe as first-argument to a function call
rnorm(100) %>>% plot()
rnorm(100) %>>% plot(col="red")
rnorm(100) %>>% plot(col="red",main=length(.))

# Pipe as first-argument to a function call in namespace
# (in this case, parentheses are required)
rnorm(100) %>>% base::mean()

# Pipe to . in an expression enclosed by braces
# representing the piped object
rnorm(100) %>>% { plot(.,col="red",main=length(.)) }

# Pipe to . in an expression enclosed by parentheses
# representing the piped object
rnorm(100) %>>% (plot(.,col="red",main=length(.)))

# Pipe to an expression enclosed by parentheses with
# lambda expression in the form of x ~ f(x).
rnorm(100) %>>% (x ~ plot(x,col="red",main=length(x)))

# Pipe to an expression for side effect and return
# the input value
rnorm(100) %>>%
  (~ cat("Number of points:",length(.))) %>>%
  summary

rnorm(100) %>>%
  (~ x ~ cat("Number of points:",length(x))) %>>%
  summary

# Assign the input value to a symbol in calling environment
# as side-effect
mtcars %>>%
  subset(mpg <= mean(mpg)) %>>%
  (~ sub_mtcars) %>>%
  summary

# Assign to a symbol the value calculated by lambda expression
# as side effect
```

```

mtcars %>>%
  (~ summary_mtcars = summary(.)) %>>%
  (~ lm_mtcars = df ~ lm(mpg ~ ., data = df)) %>>%
  subset(mpg <= mean(mpg)) %>>%
  summary

# Modifying values in calling environment
"col_" %>>%
  paste0(colnames(mtcars)) %>>%
  {names(mtcars) <- .}

rnorm(100) %>>% {
  num_mean <- mean(.)
  num_sd <- sd(.)
  num_var <- var(.)
}

rnorm(100) %>>% {
  .mean <- mean(.)
  .sd <- sd(.)
  ci <- .mean + c(-1,1) * 1.96 * .sd
}

for(i in 1:10) rnorm(i) %>>% (assign(paste0("var", i), .))

# Pipe for element extraction
mtcars %>>% (mpg)

# Pipe for questioning and printing
rnorm(100) %>>%
  (? summary(.)) %>>%
  plot(col="red")

mtcars %>>%
  "data prepared" %>>%
  lm(formula = mpg ~ wt + cyl) %>>%
  summary %>>%
  coef

mtcars %>>%
  ("Sample data" ? head(., 3)) %>>%
  lm(formula = mpg ~ wt + cyl) %>>%
  summary %>>%
  coef

# Pipe to an anonymous function
rnorm(100) %>>% (function(x) mean(x))()
rnorm(100) %>>% {function(x) mean(x)}()

# Pipe to an enclosed function to make a closure
z <- rnorm(100) %>>% (function(x) mean(x+.))
z(1) # 3

```

%>>%

11

```
z <- rnorm(100) %>>% {function(x) mean(x+.)}
z(1) # 3

# Data manipulation with dplyr
library(dplyr)
iris %>>%
  mutate(Sepal.Size=Sepal.Length*Sepal.Width,
         Petal.Size=Petal.Length*Petal.Width) %>>%
  select(Sepal.Size,Petal.Size,Species) %>>%
  group_by(Species) %>>%
  do( arrange(.,desc(Sepal.Size+Petal.Size)) %>>% head(3))

# Data manipulation with rlist
library(rlist)
list(1,2,3) %>>%
  list.map(. + 1) %>>%
  list.filter(. <= 5) %>>%
  list.sort(.)

## End(Not run)
```

Index

`%>>%`, [7](#)

Pipe, [3](#)

pipeline, [5](#)

pipeR-package, [2](#)