

Package ‘pipeflow’

July 26, 2025

Title Lightweight, General-Purpose Data Analysis Pipelines

Version 0.2.3

Description A lightweight yet powerful framework for building robust data analysis pipelines. With 'pipeflow', you initialize a pipeline with your dataset and construct workflows step by step by adding R functions. You can modify, remove, or insert steps and parameters at any stage, while 'pipeflow' ensures the pipeline's integrity. Overall, this package offers a beginner-friendly framework that simplifies and streamlines the development of data analysis pipelines by making them modular, intuitive, and adaptable.

Depends R (>= 4.2.0)

Imports data.table, jsonlite, lgr, methods, R6, stats, utils

Suggests ggplot2, knitr, mockery, rmarkdown, testthat, visNetwork

URL <https://rpahl.github.io/pipeflow/>,
<https://github.com/rpahl/pipeflow>

BugReports <https://github.com/rpahl/pipeflow/issues>

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

NeedsCompilation no

License GPL-3

Maintainer Roman Pahl <roman.pahl@gmail.com>

Author Roman Pahl [aut, cre]

Repository CRAN

Date/Publication 2025-07-26 18:40:02 UTC

Contents

Pipeline	2
pipe_add	35
pipe_append	36
pipe_append_to_step_names	37
pipe_clone	38
pipe_collect_out	39
pipe_discard_steps	40
pipe_get_data	41
pipe_get_depends	41
pipe_get_graph	43
pipe_get_out	43
pipe_get_params	44
pipe_get_step	45
pipe_insert_after	47
pipe_length	48
pipe_lock_step	48
pipe_new	49
pipe_pop_step	50
pipe_print	51
pipe_remove_step	52
pipe_rename_step	53
pipe_replace_step	54
pipe_reset	55
pipe_run	56
pipe_run_step	57
pipe_set_data	58
pipe_set_data_split	59
pipe_set_keep_out	60
pipe_set_params	61
pipe_set_params_at_step	62
pipe_split	63
set_log_layout	63
Index	65

Pipeline	<i>Pipeline Class</i>
----------	-----------------------

Description

This class implements an analysis pipeline. A pipeline consists of a sequence of analysis steps, which can be added one by one. Each added step may or may not depend on one or more previous steps. The pipeline keeps track of the dependencies among these steps and will ensure that all dependencies are met on creation of the pipeline, that is, before the the pipeline is run. Once the pipeline is run, the output is stored in the pipeline along with each step and can be accessed later. Different pipelines can be bound together while preserving all dependencies within each pipeline.

Public fields

name string name of the pipeline

pipeline data.table the pipeline where each row represents one step.

Methods**Public methods:**

- Pipeline\$new()
- Pipeline\$add()
- Pipeline\$append()
- Pipeline\$append_to_step_names()
- Pipeline\$collect_out()
- Pipeline\$discard_steps()
- Pipeline\$get_data()
- Pipeline\$get_depends()
- Pipeline\$get_depends_down()
- Pipeline\$get_depends_up()
- Pipeline\$get_graph()
- Pipeline\$get_out()
- Pipeline\$get_params()
- Pipeline\$get_params_at_step()
- Pipeline\$get_params_unique()
- Pipeline\$get_params_unique_json()
- Pipeline\$get_step()
- Pipeline\$get_step_names()
- Pipeline\$get_step_number()
- Pipeline\$has_step()
- Pipeline\$insert_after()
- Pipeline\$insert_before()
- Pipeline\$length()
- Pipeline\$lock_step()
- Pipeline\$pop_step()
- Pipeline\$pop_steps_after()
- Pipeline\$pop_steps_from()
- Pipeline\$print()
- Pipeline\$remove_step()
- Pipeline\$rename_step()
- Pipeline\$replace_step()
- Pipeline\$reset()
- Pipeline\$run()
- Pipeline\$run_step()
- Pipeline\$set_data()

- `Pipeline$set_data_split()`
- `Pipeline$set_keep_out()`
- `Pipeline$set_params()`
- `Pipeline$set_params_at_step()`
- `Pipeline$split()`
- `Pipeline$unlock_step()`
- `Pipeline$clone()`

Method `new()`: constructor

Usage:

```
Pipeline$new(name, data = NULL, logger = NULL)
```

Arguments:

`name` the name of the Pipeline

`data` optional data used at the start of the pipeline. The data also can be set later using the `set_data` function.

`logger` custom logger to be used for logging. If no logger is provided, the default logger is used, which should be sufficient for most use cases. If you do want to use your own custom log function, you need to provide a function that obeys the following form:

```
function(level, msg, ...) { your custom logging code here }
```

The `level` argument is a string and will be one of `info`, `warn`, or `error`. The `msg` argument is a string containing the message to be logged. The `...` argument is a list of named parameters, which can be used to add additional information to the log message. Currently, this is only used to add the context in case of a step giving a warning or error.

Note that with the default logger, the log layout can be altered any time via `set_log_layout()`.

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("myPipe", data = data.frame(x = 1:8))
p

# Passing custom logger
my_logger <- function(level, msg, ...) {
  cat(level, msg, "\n")
}
p <- Pipeline$new("myPipe", logger = my_logger)
```

Method `add()`: Add pipeline step

Usage:

```
Pipeline$add(
  step,
  fun,
  params = list(),
  description = "",
  group = step,
  keepOut = FALSE
)
```

Arguments:

step string the name of the step. Each step name must be unique.

fun function or name of the function to be applied at the step. Both existing and anonymous/lambda functions can be used. All function parameters must have default values. If a parameter is missing a default value in the function signature, alternatively, it can be set via the **params** argument (see Examples section with `mean()` function).

params list list of parameters to set or overwrite parameters of the passed function.

description string optional description of the step

group string output collected after pipeline execution (see function `collect_out`) is grouped by the defined group names. By default, this is the name of the step, which comes in handy when the pipeline is copy-appended multiple times to keep the results of the same function/step grouped at one place.

keepOut logical if FALSE (default) the output of the step is not collected when calling `collect_out` after the pipeline run. This option is used to only keep the results that matter and skip intermediate results that are not needed. See also function `collect_out` for more details.

Returns: returns the Pipeline object invisibly

Examples:

```
# Add steps with lambda functions
p <- Pipeline$new("myPipe", data = 1)
p$add("s1", \(x = ~data) 2*x) # use input data
p$add("s2", \(x = ~data, y = ~s1) x * y)
try(p$add("s2", \(z = 3) 3)) # error: step 's2' exists already
try(p$add("s3", \(z = ~foo) 3)) # dependency 'foo' not found
p

# Add step with existing function
p <- Pipeline$new("myPipe", data = c(1, 2, NA, 3, 4))
p$add("calc_mean", mean, params = list(x = ~data, na.rm = TRUE))
p$run()$get_out("calc_mean")

# Step description
p <- Pipeline$new("myPipe", data = 1:10)
p$add("s1", \(x = ~data) 2*x, description = "multiply by 2")
print(p)
print(p, verbose = TRUE) # print all columns

# Group output
p <- Pipeline$new("myPipe", data = data.frame(x = 1:5, y = 1:5))
p$add("prep_x", \(data = ~data) data$x, group = "prep")
p$add("prep_y", \(data = ~data) (data$y)^2, group = "prep")
p$add("sum", \(x = ~prep_x, y = ~prep_y) x + y)
p$run()$collect_out(all = TRUE)
```

Method `append()`: Append another pipeline When appending, `pipeflow` takes care of potential name clashes with respect to step names and dependencies, that is, if needed, it will automatically adapt step names and dependencies to make sure they are unique in the merged pipeline.

Usage:

```
Pipeline$append(p, outAsIn = FALSE, tryAutofixNames = TRUE, sep = ".")
```

Arguments:

p Pipeline object to be appended.

outAsIn logical if TRUE, output of first pipeline is used as input for the second pipeline.

tryAutofixNames logical if TRUE, name clashes are tried to be automatically resolved by appending the 2nd pipeline's name. Only set to FALSE, if you know what you are doing.

sep string separator used when auto-resolving step names

Returns: returns new combined Pipeline.

Examples:

```
# Append pipeline
```

```
p1 <- Pipeline$new("pipe1")
```

```
p1$add("step1", \(x = 1) x)
```

```
p2 <- Pipeline$new("pipe2")
```

```
p2$add("step2", \(y = 1) y)
```

```
p1$append(p2)
```

```
# Append pipeline with potential name clashes
```

```
p3 <- Pipeline$new("pipe3")
```

```
p3$add("step1", \(z = 1) z)
```

```
p1$append(p2)$append(p3)
```

```
# Use output of first pipeline as input for second pipeline
```

```
p1 <- Pipeline$new("pipe1", data = 8)
```

```
p2 <- Pipeline$new("pipe2")
```

```
p1$add("square", \(x = ~data) x^2)
```

```
p2$add("log2", \(x = ~data) log2(x))
```

```
p12 <- p1$append(p2, outAsIn = TRUE)
```

```
p12$run()$get_out("log2")
```

```
p12
```

```
# Custom name separator
```

```
p1$append(p2, sep = "___")
```

Method `append_to_step_names()`: Appends string to all step names and takes care of updating step dependencies accordingly.

Usage:

```
Pipeline$append_to_step_names(postfix, sep = ".")
```

Arguments:

postfix string to be appended to each step name.

sep string separator between step name and postfix.

Returns: returns the Pipeline object invisibly

Examples:

```

p <- Pipeline$new("pipe")
p$add("step1", \(x = 1) x)
p$add("step2", \(y = 1) y)
p$append_to_step_names("new")
p
p$append_to_step_names("foo", sep = "__")
p

```

Method `collect_out()`: Collect output after pipeline run, by default, from all steps for which `keepOut` was set to `TRUE`. The output is grouped by the group names (see `group` parameter in function `add`), which by default are set identical to the step names.

Usage:

```
Pipeline$collect_out(groupBy = "group", all = FALSE)
```

Arguments:

`groupBy` string column of pipeline by which to group the output.

`all` logical if `TRUE` all output is collected regardless of the `keepOut` flag. This can be useful for debugging.

Returns: list containing the output, named after the groups, which, by default, are the steps.

Examples:

```

p <- Pipeline$new("pipe", data = 1:2)
p$add("step1", \(x = ~data) x + 2)
p$add("step2", \(x = ~step1) x + 2, keepOut = TRUE)
p$run()
p$collect_out()
p$collect_out(all = TRUE) |> str()

# Grouped output
p <- Pipeline$new("pipe", data = 1:2)
p$add("step1", \(x = ~data) x + 2, group = "add")
p$add("step2", \(x = ~step1, y = 2) x + y, group = "add")
p$add("step3", \(x = ~data) x * 3, group = "mult")
p$add("step4", \(x = ~data, y = 2) x * y, group = "mult")
p
p$run()
p$collect_out(all = TRUE) |> str()

# Grouped by state
p$set_params(list(y = 5))
p
p$collect_out(groupBy = "state", all = TRUE) |> str()

```

Method `discard_steps()`: Discard all steps that match a given pattern.

Usage:

```
Pipeline$discard_steps(pattern, recursive = FALSE, fixed = TRUE, ...)
```

Arguments:

pattern string containing a regular expression (or character string for fixed = TRUE) to be matched.

recursive logical if TRUE the step is removed together with all its downstream dependencies. fixed logical If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.

... further arguments passed to `grep()`.

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~add1) x + 2)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p

p$discard_steps("mult")
p

# Re-add steps
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p
# Discarding 'add1' does not work ...
try(p$discard_steps("add1"))

# ... unless we enforce to remove its downstream dependencies as well
p$discard_steps("add1", recursive = TRUE) # this works
p

# Trying to discard non-existent steps is just ignored
p$discard_steps("non-existent")
```

Method `get_data()`: Get data

Usage:

```
Pipeline$get_data()
```

Returns: the output defined in the data step, which by default is the first step of the pipeline

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$get_data()
p$set_data(3:4)
p$get_data()
```

Method `get_depends()`: Get all dependencies defined in the pipeline

Usage:

```
Pipeline$get_depends()
```

Returns: named list of dependencies for each step

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$get_depends()
```

Method `get_depends_down()`: Get all downstream dependencies of given step, by default descending recursively.

Usage:

```
Pipeline$get_depends_down(step, recursive = TRUE)
```

Arguments:

`step` string name of step

`recursive` logical if TRUE, dependencies of dependencies are also returned.

Returns: list of downstream dependencies

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p$get_depends_down("add1")
p$get_depends_down("add1", recursive = FALSE)
```

Method `get_depends_up()`: Get all upstream dependencies of given step, by default descending recursively.

Usage:

```
Pipeline$get_depends_up(step, recursive = TRUE)
```

Arguments:

`step` string name of step

`recursive` logical if TRUE, dependencies of dependencies are also returned.

Returns: list of upstream dependencies

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p$get_depends_up("mult4")
p$get_depends_up("mult4", recursive = FALSE)
```

Method `get_graph()`: Visualize the pipeline as a graph.

Usage:

```
Pipeline$get_graph(groups = NULL)
```

Arguments:

`groups` character if not NULL, only steps belonging to the given groups are considered.

Returns: two data frames, one for nodes and one for edges ready to be used with the `visNetwork` package.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p$add("mult1", \(x = ~add1, y = ~add2) x * y)
graph <- pipe_get_graph(p)

graph

if (require("visNetwork", quietly = TRUE)) {
  do.call(visNetwork, args = p$get_graph())
}
```

Method `get_out()`: Get output of given step

Usage:

```
Pipeline$get_out(step)
```

Arguments:

`step` string name of step

Returns: the output at the given step.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$run()
p$get_out("add1")
p$get_out("add2")
```

Method `get_params()`: Set unbound function parameters defined in the pipeline where 'unbound' means parameters that are not linked to other steps. Trying # to set parameters that don't exist in the pipeline is ignored, by default, with a warning.

Usage:

```
Pipeline$get_params(ignoreHidden = TRUE)
```

Arguments:

`ignoreHidden` logical if TRUE, hidden parameters (i.e. all names starting with a dot) are ignored and thus not returned.

Returns: list of parameters, sorted and named by step. Steps with no parameters are filtered out.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
```

```
p$add("add3", \( ) 1 + 2)
p$get_params() |> str()
p$get_params(ignoreHidden = FALSE) |> str()
```

Method `get_params_at_step()`: Get all unbound (i.e. not referring to other steps) at given step name.

Usage:

```
Pipeline$get_params_at_step(step, ignoreHidden = TRUE)
```

Arguments:

`step` string name of step

`ignoreHidden` logical if TRUE, hidden parameters (i.e. all names starting with a dot) are ignored and thus not returned.

Returns: list of parameters defined at given step.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("add3", \( ) 1 + 2)
p$get_params_at_step("add2")
p$get_params_at_step("add2", ignoreHidden = FALSE)
p$get_params_at_step("add3")
```

Method `get_params_unique()`: Get all unbound (i.e. not referring to other steps) parameters defined in the pipeline, but only list each parameter once. The values of the parameters, will be the values of the first step where the parameter was defined. This is particularly useful after the parameters were set using the `set_params` function, which will set the same value for all steps.

Usage:

```
Pipeline$get_params_unique(ignoreHidden = TRUE)
```

Arguments:

`ignoreHidden` logical if TRUE, hidden parameters (i.e. all names starting with a dot) are ignored and thus not returned.

Returns: list of unique parameters

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("mult1", \(x = 1, y = 2, .z = 3, b = ~add2) x * y * b)
p$get_params_unique()
p$get_params_unique(ignoreHidden = FALSE)
```

Method `get_params_unique_json()`: Get all unique function parameters in json format.

Usage:

```
Pipeline$get_params_unique_json(ignoreHidden = TRUE)
```

Arguments:

ignoreHidden logical if TRUE, hidden parameters (i.e. all names starting with a dot) are ignored and thus not returned.

Returns: list flat unnamed json list of unique function parameters

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("mult1", \(x = 1, y = 2, .z = 3, b = ~add2) x * y * b)
p$get_params_unique_json()
p$get_params_unique_json(ignoreHidden = FALSE)
```

Method get_step(): Get step of pipeline

Usage:

```
Pipeline$get_step(step)
```

Arguments:

step string name of step

Returns: data.table row containing the step.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, z = ~add1) x + y + z)
p$run()
add1 <- p$get_step("add1")
print(add1)
add1[["params"]]
add1[["fun"]]
try()
try(p$get_step("foo")) # error: step 'foo' does not exist
```

Method get_step_names(): Get step names of pipeline

Usage:

```
Pipeline$get_step_names()
```

Returns: character vector of step names

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$get_step_names()
```

Method get_step_number(): Get step number

Usage:

```
Pipeline$get_step_number(step)
```

Arguments:

step string name of step

Returns: the step number in the pipeline

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$get_step_number("f2")
```

Method `has_step()`: Check if pipeline has given step

Usage:

```
Pipeline$has_step(step)
```

Arguments:

`step` string name of step

Returns: logical whether step exists

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$has_step("f2")
p$has_step("foo")
```

Method `insert_after()`: Insert step after a certain step

Usage:

```
Pipeline$insert_after(afterStep, step, ...)
```

Arguments:

`afterStep` string name of step after which to insert

`step` string name of step to insert

`...` further arguments passed to add method of the pipeline

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = 1) x)
p$add("f2", \(x = ~f1) x)
p$insert_after("f1", "f3", \(x = ~f1) x)
p
```

Method `insert_before()`: Insert step before a certain step

Usage:

```
Pipeline$insert_before(beforeStep, step, ...)
```

Arguments:

`beforeStep` string name of step before which to insert

`step` string name of step to insert

`...` further arguments passed to add method of the pipeline

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = 1) x)
p$add("f2", \(x = ~f1) x)
p$insert_before("f2", "f3", \(x = ~f1) x)
p
```

Method `length()`: Length of the pipeline aka number of pipeline steps.

Usage:

```
Pipeline$length()
```

Returns: numeric length of pipeline.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$length()
```

Method `lock_step()`: Locking a step means that both its parameters and its output (given it has output) are locked such that neither setting new pipeline parameters nor future pipeline runs can change the current parameter and output content.

Usage:

```
Pipeline$lock_step(step)
```

Arguments:

`step` string name of step

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = 1, data = ~data) x + data)
p$add("add2", \(x = 1, data = ~data) x + data)
p$run()
p$get_out("add1")
p$get_out("add2")
p$lock_step("add1")

p$set_data(3)
p$set_params(list(x = 3))
p$run()
p$get_out("add1")
p$get_out("add2")
```

Method `pop_step()`: Drop last step from the pipeline.

Usage:

```
Pipeline$pop_step()
```

Returns: string the name of the step that was removed

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p
p$pop_step() # "f2"
p
```

Method `pop_steps_after()`: Drop all steps after the given step.

Usage:

```
Pipeline$pop_steps_after(step)
```

Arguments:

`step` string name of step

Returns: character vector of steps that were removed.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$add("f3", \(z = 1) z)
p$pop_steps_after("f1") # "f2", "f3"
p
```

Method `pop_steps_from()`: Drop all steps from and including the given step.

Usage:

```
Pipeline$pop_steps_from(step)
```

Arguments:

`step` string name of step

Returns: character vector of steps that were removed.

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$add("f3", \(z = 1) z)
p$pop_steps_from("f2") # "f2", "f3"
p
```

Method `print()`: Print the pipeline as a table.

Usage:

```
Pipeline$print(verbose = FALSE)
```

Arguments:

`verbose` logical if TRUE, print all columns of the pipeline, otherwise only the most relevant columns are displayed.

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$print()
```

Method `remove_step()`: Remove certain step from the pipeline. If other steps depend on the step to be removed, an error is given and the removal is blocked, unless `recursive` was set to `TRUE`.

Usage:

```
Pipeline$remove_step(step, recursive = FALSE)
```

Arguments:

`step` string the name of the step to be removed.

`recursive` logical if `TRUE` the step is removed together with all its downstream dependencies.

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p$add("mult1", \(x = 1, y = ~add2) x * y)
p$remove_step("mult1")
p
try(p$remove_step("add1")) # fails because "add2" depends on "add1"
p$remove_step("add1", recursive = TRUE) # removes "add1" and "add2"
p
```

Method `rename_step()`: Safely rename a step in the pipeline. If new step name would result in a name clash, an error is given.

Usage:

```
Pipeline$rename_step(from, to)
```

Arguments:

`from` string the name of the step to be renamed.

`to` string the new name of the step.

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p
try(p$rename_step("add1", "add2")) # fails because "add2" exists
p$rename_step("add1", "first_add") # Ok
p
```

Method `replace_step()`: Replaces an existing pipeline step.

Usage:

```
Pipeline$replace_step(
  step,
  fun,
  params = list(),
  description = "",
  group = step,
  keepOut = FALSE
)
```

Arguments:

`step` string the name of the step to be replaced. Step must exist.

`fun` string or function operation to be applied at the step. Both existing and lambda/anonymous functions can be used.

`params` list list of parameters to overwrite default parameters of existing functions.

`description` string optional description of the step

`group` string grouping information (by default the same as the name of the step. Any output collected later (see function `collect_out` by default is put together by these group names. This, for example, comes in handy when the pipeline is copy-appended multiple times to keep the results of the same function/step at one place.

`keepOut` logical if FALSE the output of the function will be cleaned at the end of the whole pipeline execution. This option is used to only keep the results that matter.

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("add2", \(x = ~data, y = 2) x + y)
p$add("mult", \(x = 1, y = 2) x * y, keepOut = TRUE)
p$run()$collect_out()
p$replace_step("mult", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
p$run()$collect_out()
try(p$replace_step("foo", \(x = 1) x)) # step 'foo' does not exist
```

Method `reset()`: Resets the pipeline to the state before it was run. This means that all output is removed and the state of all steps is reset to 'New'.

Usage:

```
Pipeline$reset()
```

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$run()
p
p$reset()
p
```

Method run(): Run all new and/or outdated pipeline steps.

Usage:

```
Pipeline$run(
  force = FALSE,
  recursive = TRUE,
  cleanUnkept = FALSE,
  progress = NULL,
  showLog = TRUE
)
```

Arguments:

force logical if TRUE all steps are run regardless of whether they are outdated or not.

recursive logical if TRUE and a step returns a new pipeline, the run of the current pipeline is aborted and the new pipeline is run recursively.

cleanUnkept logical if TRUE all output that was not marked to be kept is removed after the pipeline run. This option can be useful if temporary results require a lot of memory.

progress function this parameter can be used to provide a custom progress function of the form `function(value, detail)`, which will show the progress of the pipeline run for each step, where `value` is the current step number and `detail` is the name of the step.

showLog logical should the steps be logged during the pipeline run?

Returns: returns the Pipeline object invisibly

Examples:

```
# Simple pipeline
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("add2", \(x = ~add1, z = 2) x + z)
p$add("final", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
p$run()$collect_out()
p$set_params(list(z = 4)) # outdates steps add2 and final
p
p$run()$collect_out()
p$run(cleanUnkept = TRUE) # clean up temporary results
p

# Recursive pipeline
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("new_pipe", \(x = ~add1) {
  pp <- Pipeline$new("new_pipe", data = x)
  pp$add("add1", \(x = ~data) x + 1)
  pp$add("add2", \(x = ~add1) x + 2, keepOut = TRUE)
}
)
p$run(recursive = TRUE)$collect_out()

# Run pipeline with progress bar
p <- Pipeline$new("pipe", data = 1)
```

```

p$add("first step", \() Sys.sleep(1))
p$add("second step", \() Sys.sleep(1))
p$add("last step", \() Sys.sleep(1))
pb <- txtProgressBar(min = 1, max = p$length(), style = 3)
fprogress <- function(value, detail) {
  setTxtProgressBar(pb, value)
}
p$run(progress = fprogress, showLog = FALSE)

```

Method `run_step()`: Run given pipeline step possibly together with upstream and downstream dependencies.

Usage:

```

Pipeline$run_step(
  step,
  upstream = TRUE,
  downstream = FALSE,
  cleanUnkept = FALSE
)

```

Arguments:

`step` string name of step

`upstream` logical if TRUE, run all dependent upstream steps first.

`downstream` logical if TRUE, run all dependent downstream afterwards.

`cleanUnkept` logical if TRUE all output that was not marked to be kept is removed after the pipeline run. This option can be useful if temporary results require a lot of memory.

Returns: returns the Pipeline object invisibly

Examples:

```

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \ (x = ~data, y = 1) x + y)
p$add("add2", \ (x = ~add1, z = 2) x + z)
p$add("mult", \ (x = ~add1, y = ~add2) x * y)
p$run_step("add2")
p$run_step("add2", downstream = TRUE)
p$run_step("mult", upstream = TRUE)

```

Method `set_data()`: Set data in first step of pipeline.

Usage:

```

Pipeline$set_data(data)

```

Arguments:

`data` initial data set

Returns: returns the Pipeline object invisibly

Examples:

```

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \ (x = ~data, y = 1) x + y, keepOut = TRUE)
p$run()$collect_out()
p$set_data(3)
p$run()$collect_out()

```

Method `set_data_split()`: This function can be used to apply the pipeline repeatedly to various data sets. For this, the pipeline split-copies itself by the list of given data sets. Each sub-pipeline will have one of the data sets set as input data. The step names of the sub-pipelines will be the original step names plus the name of the data set.

Usage:

```
Pipeline$set_data_split(
  dataList,
  toStep = character(),
  groupBySplit = TRUE,
  sep = "."
)
```

Arguments:

`dataList` list of data sets

`toStep` string step name marking optional subset of the pipeline, at which the data split should be applied to.

`groupBySplit` logical whether to set step groups according to data split.

`sep` string separator to be used between step name and data set name when creating the new step names.

Returns: new combined Pipeline with each sub-pipeline having set one of the data sets.

Examples:

```
# Split by three data sets
dataList <- list(a = 1, b = 2, c = 3)
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
p$set_data_split(dataList)
p
p$run()$collect_out() |> str()

# Don't group output by split
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
p$set_data_split(dataList, groupBySplit = FALSE)
p
p$run()$collect_out() |> str()

# Split up to certain step
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1)
p$add("mult", \(x = ~data, y = ~add1) x * y)
p$add("average_result", \(x = ~mult) mean(unlist(x)), keepOut = TRUE)
p
p$get_depends()[["average_result"]]

p$set_data_split(dataList, toStep = "mult")
```

```
p
p$get_depends()[["average_result"]]

p$run()$collect_out()
```

Method `set_keep_out()`: Change the `keepOut` flag at a given pipeline step, which determines whether the output of that step is collected when calling `collect_out()` after the pipeline was run.

Usage:

```
Pipeline$set_keep_out(step, keepOut = TRUE)
```

Arguments:

`step` string name of step

`keepOut` logical whether to keep output of step

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y, keepOut = TRUE)
p$add("add2", \(x = ~data, y = 2) x + y)
p$add("mult", \(x = ~add1, y = ~add2) x * y)
p$run()$collect_out()
p$set_keep_out("add1", keepOut = FALSE)
p$set_keep_out("mult", keepOut = TRUE)
p$collect_out()
```

Method `set_params()`: Set parameters in the pipeline. If a parameter occurs in several steps, the parameter is set commonly in all steps. Trying to set parameters that don't exist in the pipeline is ignored, by default, with a warning.

Usage:

```
Pipeline$set_params(params, warnUndefined = TRUE)
```

Arguments:

`params` list of parameters to be set

`warnUndefined` logical whether to give a warning when trying to set a parameter that is not defined in the pipeline.

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 2) x + y)
p$add("add2", \(x = ~data, y = 3) x + y)
p$add("mult", \(x = 4, z = 5) x * z)
p$get_params()
p$set_params(list(x = 3, y = 3))
p$get_params()
p$set_params(list(x = 5, z = 3))
p$get_params()
```

```
suppressWarnings(
  p$set_params(list(foo = 3)) # gives warning as 'foo' is undefined
)
p$set_params(list(foo = 3), warnUndefined = FALSE)
```

Method `set_params_at_step()`: Set unbound function parameters defined at given pipeline step where 'unbound' means parameters that are not linked to other steps.

Usage:

```
Pipeline$set_params_at_step(step, params)
```

Arguments:

`step` string the name of the step

`params` list of parameters to be set

Returns: returns the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 2, z = 3) x + y)
p$set_params_at_step("add1", list(y = 5, z = 6))
p$get_params()
try(p$set_params_at_step("add1", list(foo = 3))) # foo not defined
```

Method `split()`: Splits pipeline into its independent parts.

Usage:

```
Pipeline$split()
```

Returns: list of Pipeline objects

Examples:

```
# Example for two independent calculation paths
p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = ~data) x)
p$add("f2", \(x = 1) x)
p$add("f3", \(x = ~f1) x)
p$add("f4", \(x = ~f2) x)
p$split()

# Example of split by three data sets
dataList <- list(a = 1, b = 2, c = 3)
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
pipes <- p$set_data_split(dataList)$split()
pipes
```

Method `unlock_step()`: Unlock previously locked step. If step was not locked, the command is ignored.

Usage:

```
Pipeline$unlock_step(step)
```

Arguments:

step string name of step

Returns: the Pipeline object invisibly

Examples:

```
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = 1, data = ~data) x + data)
p$add("add2", \(x = 1, data = ~data) x + data)
p$lock_step("add1")
p$set_params(list(x = 3))
p$get_params()
p$unlock_step("add1")
p$set_params(list(x = 3))
p$get_params()
```

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
Pipeline$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Roman Pahl

Examples

```
## -----
## Method `Pipeline$new`
## -----

p <- Pipeline$new("myPipe", data = data.frame(x = 1:8))
p

# Passing custom logger
my_logger <- function(level, msg, ...) {
  cat(level, msg, "\n")
}
p <- Pipeline$new("myPipe", logger = my_logger)

## -----
## Method `Pipeline$add`
## -----

# Add steps with lambda functions
p <- Pipeline$new("myPipe", data = 1)
p$add("s1", \(x = ~data) 2*x) # use input data
p$add("s2", \(x = ~data, y = ~s1) x * y)
try(p$add("s2", \(z = 3) 3)) # error: step 's2' exists already
try(p$add("s3", \(z = ~foo) 3)) # dependency 'foo' not found
```

```

p

# Add step with existing function
p <- Pipeline$new("myPipe", data = c(1, 2, NA, 3, 4))
p$add("calc_mean", mean, params = list(x = ~data, na.rm = TRUE))
p$run()$get_out("calc_mean")

# Step description
p <- Pipeline$new("myPipe", data = 1:10)
p$add("s1", \(x = ~data) 2*x, description = "multiply by 2")
print(p)
print(p, verbose = TRUE) # print all columns

# Group output
p <- Pipeline$new("myPipe", data = data.frame(x = 1:5, y = 1:5))
p$add("prep_x", \(data = ~data) data$x, group = "prep")
p$add("prep_y", \(data = ~data) (data$y)^2, group = "prep")
p$add("sum", \(x = ~prep_x, y = ~prep_y) x + y)
p$run()$collect_out(all = TRUE)

## -----
## Method `Pipeline$append`
## -----

# Append pipeline
p1 <- Pipeline$new("pipe1")
p1$add("step1", \(x = 1) x)
p2 <- Pipeline$new("pipe2")
p2$add("step2", \(y = 1) y)
p1$append(p2)

# Append pipeline with potential name clashes
p3 <- Pipeline$new("pipe3")
p3$add("step1", \(z = 1) z)
p1$append(p2)$append(p3)

# Use output of first pipeline as input for second pipeline
p1 <- Pipeline$new("pipe1", data = 8)
p2 <- Pipeline$new("pipe2")
p1$add("square", \(x = ~data) x^2)
p2$add("log2", \(x = ~data) log2(x))

p12 <- p1$append(p2, outAsIn = TRUE)
p12$run()$get_out("log2")
p12

# Custom name separator
p1$append(p2, sep = "___")

## -----
## Method `Pipeline$append_to_step_names`
## -----

```

```

p <- Pipeline$new("pipe")
p$add("step1", \(x = 1) x)
p$add("step2", \(y = 1) y)
p$append_to_step_names("new")
p
p$append_to_step_names("foo", sep = "__")
p

## -----
## Method `Pipeline$collect_out`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("step1", \(x = ~data) x + 2)
p$add("step2", \(x = ~step1) x + 2, keepOut = TRUE)
p$run()
p$collect_out()
p$collect_out(all = TRUE) |> str()

# Grouped output
p <- Pipeline$new("pipe", data = 1:2)
p$add("step1", \(x = ~data) x + 2, group = "add")
p$add("step2", \(x = ~step1, y = 2) x + y, group = "add")
p$add("step3", \(x = ~data) x * 3, group = "mult")
p$add("step4", \(x = ~data, y = 2) x * y, group = "mult")
p
p$run()
p$collect_out(all = TRUE) |> str()

# Grouped by state
p$set_params(list(y = 5))
p
p$collect_out(groupBy = "state", all = TRUE) |> str()

## -----
## Method `Pipeline$discard_steps`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~add1) x + 2)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p

p$discard_steps("mult")
p

# Re-add steps
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p
# Discarding 'add1' does not work ...

```

```

try(p$discard_steps("add1"))

# ... unless we enforce to remove its downstream dependencies as well
p$discard_steps("add1", recursive = TRUE) # this works
p

# Trying to discard non-existent steps is just ignored
p$discard_steps("non-existent")

## -----
## Method `Pipeline$get_data`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$get_data()
p$set_data(3:4)
p$get_data()

## -----
## Method `Pipeline$get_depends`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$get_depends()

## -----
## Method `Pipeline$get_depends_down`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p$get_depends_down("add1")
p$get_depends_down("add1", recursive = FALSE)

## -----
## Method `Pipeline$get_depends_up`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p$get_depends_up("mult4")
p$get_depends_up("mult4", recursive = FALSE)

## -----
## Method `Pipeline$get_graph`

```

```

## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p$add("mult1", \(x = ~add1, y = ~add2) x * y)
graph <- pipe_get_graph(p)
graph

if (require("visNetwork", quietly = TRUE)) {
  do.call(visNetwork, args = p$get_graph())
}

## -----
## Method `Pipeline$get_out`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(x = ~data) x + 1)
p$add("add2", \(x = ~data, y = ~add1) x + y)
p$run()
p$get_out("add1")
p$get_out("add2")

## -----
## Method `Pipeline$get_params`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("add3", \( ) 1 + 2)
p$get_params() |> str()
p$get_params(ignoreHidden = FALSE) |> str()

## -----
## Method `Pipeline$get_params_at_step`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("add3", \( ) 1 + 2)
p$get_params_at_step("add2")
p$get_params_at_step("add2", ignoreHidden = FALSE)
p$get_params_at_step("add3")

## -----
## Method `Pipeline$get_params_unique`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)

```

```

p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("mult1", \(x = 1, y = 2, .z = 3, b = ~add2) x * y * b)
p$get_params_unique()
p$get_params_unique(ignoreHidden = FALSE)

## -----
## Method `Pipeline$get_params_unique_json`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, .z = 3) x + y + .z)
p$add("mult1", \(x = 1, y = 2, .z = 3, b = ~add2) x * y * b)
p$get_params_unique_json()
p$get_params_unique_json(ignoreHidden = FALSE)

## -----
## Method `Pipeline$get_step`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = 2, z = ~add1) x + y + z)
p$run()
add1 <- p$get_step("add1")
print(add1)
add1[["params"]]
add1[["fun"]]
try()
try(p$get_step("foo")) # error: step 'foo' does not exist

## -----
## Method `Pipeline$get_step_names`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$get_step_names()

## -----
## Method `Pipeline$get_step_number`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$get_step_number("f2")

## -----
## Method `Pipeline$has_step`
## -----

```

```

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$has_step("f2")
p$has_step("foo")

## -----
## Method `Pipeline$insert_after`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = 1) x)
p$add("f2", \(x = ~f1) x)
p$insert_after("f1", "f3", \(x = ~f1) x)
p

## -----
## Method `Pipeline$insert_before`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = 1) x)
p$add("f2", \(x = ~f1) x)
p$insert_before("f2", "f3", \(x = ~f1) x)
p

## -----
## Method `Pipeline$length`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$length()

## -----
## Method `Pipeline$lock_step`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = 1, data = ~data) x + data)
p$add("add2", \(x = 1, data = ~data) x + data)
p$run()
p$get_out("add1")
p$get_out("add2")
p$lock_step("add1")

p$set_data(3)
p$set_params(list(x = 3))
p$run()
p$get_out("add1")
p$get_out("add2")

```

```
## -----
## Method `Pipeline$pop_step`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p
p$pop_step() # "f2"
p

## -----
## Method `Pipeline$pop_steps_after`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$add("f3", \(z = 1) z)
p$pop_steps_after("f1") # "f2", "f3"
p

## -----
## Method `Pipeline$pop_steps_from`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$add("f3", \(z = 1) z)
p$pop_steps_from("f2") # "f2", "f3"
p

## -----
## Method `Pipeline$print`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$print()

## -----
## Method `Pipeline$remove_step`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p$add("mult1", \(x = 1, y = ~add2) x * y)
p$remove_step("mult1")
p
try(p$remove_step("add1")) # fails because "add2" depends on "add1"
```

```

p$remove_step("add1", recursive = TRUE) # removes "add1" and "add2"
p

## -----
## Method `Pipeline$rename_step`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$add("add2", \(x = 1, y = ~add1) x + y)
p
try(p$rename_step("add1", "add2")) # fails because "add2" exists
p$rename_step("add1", "first_add") # Ok
p

## -----
## Method `Pipeline$replace_step`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("add2", \(x = ~data, y = 2) x + y)
p$add("mult", \(x = 1, y = 2) x * y, keepOut = TRUE)
p$run()$collect_out()
p$replace_step("mult", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
p$run()$collect_out()
try(p$replace_step("foo", \(x = 1) x)) # step 'foo' does not exist

## -----
## Method `Pipeline$reset`
## -----

p <- Pipeline$new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
p$run()
p
p$reset()
p

## -----
## Method `Pipeline$run`
## -----

# Simple pipeline
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("add2", \(x = ~add1, z = 2) x + z)
p$add("final", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
p$run()$collect_out()
p$set_params(list(z = 4)) # outdates steps add2 and final
p
p$run()$collect_out()

```

```

p$run(cleanUnkept = TRUE) # clean up temporary results
p

# Recursive pipeline
p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("new_pipe", \(x = ~add1) {
  pp <- Pipeline$new("new_pipe", data = x)
  pp$add("add1", \(x = ~data) x + 1)
  pp$add("add2", \(x = ~add1) x + 2, keepOut = TRUE)
})
p$run(recursive = TRUE)$collect_out()

# Run pipeline with progress bar
p <- Pipeline$new("pipe", data = 1)
p$add("first step", \() Sys.sleep(1))
p$add("second step", \() Sys.sleep(1))
p$add("last step", \() Sys.sleep(1))
pb <- txtProgressBar(min = 1, max = p$length(), style = 3)
fprogress <- function(value, detail) {
  setTxtProgressBar(pb, value)
}
p$run(progress = fprogress, showLog = FALSE)

## -----
## Method `Pipeline$run_step`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y)
p$add("add2", \(x = ~add1, z = 2) x + z)
p$add("mult", \(x = ~add1, y = ~add2) x * y)
p$run_step("add2")
p$run_step("add2", downstream = TRUE)
p$run_step("mult", upstream = TRUE)

## -----
## Method `Pipeline$set_data`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y, keepOut = TRUE)
p$run()$collect_out()
p$set_data(3)
p$run()$collect_out()

## -----
## Method `Pipeline$set_data_split`
## -----

# Split by three data sets
dataList <- list(a = 1, b = 2, c = 3)

```

```

p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
p$set_data_split(dataList)
p
p$run()$collect_out() |> str()

# Don't group output by split
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
p$set_data_split(dataList, groupBySplit = FALSE)
p
p$run()$collect_out() |> str()

# Split up to certain step
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1)
p$add("mult", \(x = ~data, y = ~add1) x * y)
p$add("average_result", \(x = ~mult) mean(unlist(x)), keepOut = TRUE)
p
p$get_depends()[["average_result"]]

p$set_data_split(dataList, toStep = "mult")
p
p$get_depends()[["average_result"]]

p$run()$collect_out()

## -----
## Method `Pipeline$set_keep_out`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 1) x + y, keepOut = TRUE)
p$add("add2", \(x = ~data, y = 2) x + y)
p$add("mult", \(x = ~add1, y = ~add2) x * y)
p$run()$collect_out()
p$set_keep_out("add1", keepOut = FALSE)
p$set_keep_out("mult", keepOut = TRUE)
p$collect_out()

## -----
## Method `Pipeline$set_params`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 2) x + y)
p$add("add2", \(x = ~data, y = 3) x + y)
p$add("mult", \(x = 4, z = 5) x * z)
p$get_params()
p$set_params(list(x = 3, y = 3))
p$get_params()

```

```

p$set_params(list(x = 5, z = 3))
p$get_params()
suppressWarnings(
  p$set_params(list(foo = 3)) # gives warning as 'foo' is undefined
)
p$set_params(list(foo = 3), warnUndefined = FALSE)

## -----
## Method `Pipeline$set_params_at_step`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = ~data, y = 2, z = 3) x + y)
p$set_params_at_step("add1", list(y = 5, z = 6))
p$get_params()
try(p$set_params_at_step("add1", list(foo = 3))) # foo not defined

## -----
## Method `Pipeline$split`
## -----

# Example for two independent calculation paths
p <- Pipeline$new("pipe", data = 1)
p$add("f1", \(x = ~data) x)
p$add("f2", \(x = 1) x)
p$add("f3", \(x = ~f1) x)
p$add("f4", \(x = ~f2) x)
p$split()

# Example of split by three data sets
dataList <- list(a = 1, b = 2, c = 3)
p <- Pipeline$new("pipe")
p$add("add1", \(x = ~data) x + 1, keepOut = TRUE)
p$add("mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
pipes <- p$set_data_split(dataList)$split()
pipes

## -----
## Method `Pipeline$unlock_step`
## -----

p <- Pipeline$new("pipe", data = 1)
p$add("add1", \(x = 1, data = ~data) x + data)
p$add("add2", \(x = 1, data = ~data) x + data)
p$lock_step("add1")
p$set_params(list(x = 3))
p$get_params()
p$unlock_step("add1")
p$set_params(list(x = 3))
p$get_params()

```

pipe_add	<i>Add pipeline step</i>
----------	--------------------------

Description

A pipeline consists of a series of steps, which usually are added one by one. Each step is made up of a function computing something once the pipeline is run. This function can be an existing R function (e.g. `mean()`) or an anonymous/lambda function specifically defined for the pipeline. One useful feature is that function parameters can refer to results of earlier pipeline steps using the syntax `x = ~earlier_step_name` - see the Examples for more details.

Usage

```
pipe_add(
  pip,
  step,
  fun,
  params = list(),
  description = "",
  group = step,
  keepOut = FALSE
)
```

Arguments

pip	Pipeline object
step	string the name of the step. Each step name must be unique.
fun	function or name of the function to be applied at the step. Both existing and anonymous/lambda functions can be used. All function parameters must have default values. If a parameter is missing a default value in the function signature, alternatively, it can be set via the params argument (see Examples section with <code>mean()</code> function).
params	list list of parameters to set or overwrite parameters of the passed function.
description	string optional description of the step
group	string output collected after pipeline execution (see <code>pipe_collect_out()</code> is grouped by the defined group names. By default, this is the name of the step, which comes in handy when the pipeline is copy-appended multiple times to keep the results of the same function/step grouped at one place.
keepOut	logical if FALSE (default) the output of the step is not collected when calling <code>pipe_collect_out()</code> after the pipeline run. This option is used to only keep the results that matter and skip intermediate results that are not needed. See also function <code>pipe_collect_out()</code> for more details.

Value

returns the Pipeline object invisibly

Examples

```
# Add steps with lambda functions
p <- pipe_new("myPipe", data = 1)
pipe_add(p, "s1", \(x = ~data) 2*x) # use input data
pipe_add(p, "s2", \(x = ~data, y = ~s1) x * y)
try(pipe_add(p, "s2", \(z = 3) 3)) # error: step 's2' exists already
try(pipe_add(p, "s3", \(z = ~foo) 3)) # dependency 'foo' not found
p

# Add step with existing function
p <- pipe_new("myPipe", data = c(1, 2, NA, 3, 4))
try(pipe_add(p, "calc_mean", mean)) # default value for x is missing
pipe_add(p, "calc_mean", mean, params = list(x = ~data, na.rm = TRUE))
p |> pipe_run() |> pipe_get_out("calc_mean")

# Step description
p <- pipe_new("myPipe", data = 1:10)
pipe_add(p, "s1", \(x = ~data) 2*x, description = "multiply by 2")
print(p, verbose = TRUE) # print all columns including description

# Group output
p <- pipe_new("myPipe", data = data.frame(x = 1:2, y = 3:4))
pipe_add(p, "prep_x", \(data = ~data) data$x, group = "prep")
pipe_add(p, "prep_y", \(data = ~data) (data$y)^2, group = "prep")
pipe_add(p, "sum", \(x = ~prep_x, y = ~prep_y) x + y)
p |> pipe_run() |> pipe_collect_out(all = TRUE)
```

pipe_append

Append two pipelines

Description

When appending, pipeflow takes care of potential name clashes with respect to step names and dependencies, that is, if needed, it will automatically adapt step names and dependencies to make sure they are unique in the merged pipeline.

Usage

```
pipe_append(pip, p, outAsIn = FALSE, tryAutofixNames = TRUE, sep = ".")
```

Arguments

pip	Pipeline object to be appended to.
p	Pipeline object to be appended.
outAsIn	logical if TRUE, output of first pipeline is used as input for the second pipeline.

`tryAutofixNames`
 logical if TRUE, name clashes are tried to be automatically resolved by appending the 2nd pipeline's name. Only set to FALSE, if you know what you are doing.

`sep`
 string separator used when auto-resolving step names

Value

returns new combined Pipeline object.

Examples

```
# Append pipeline
p1 <- pipe_new("pipe1")
pipe_add(p1, "step1", \(x = 1) x)
p2 <- pipe_new("pipe2")
pipe_add(p2, "step2", \(y = 1) y)
p1 |> pipe_append(p2)

# Append pipeline with potential name clashes
p3 <- pipe_new("pipe3")
pipe_add(p3, "step1", \(z = 1) z)
p1 |> pipe_append(p2) |> pipe_append(p3)

# Use output of first pipeline as input for second pipeline
p1 <- pipe_new("pipe1", data = 8)
p2 <- pipe_new("pipe2")
pipe_add(p1, "square", \(x = ~data) x^2)
pipe_add(p2, "log2", \(x = ~data) log2(x))

p12 <- p1 |> pipe_append(p2, outAsIn = TRUE)
p12 |> pipe_run() |> pipe_get_out("log2")
p12

# Custom name separator for adapted step names
p1 |> pipe_append(p2, sep = "___")
```

pipe_append_to_step_names

Append string to all step names

Description

Appends string to all step names and takes care of updating step dependencies accordingly.

Usage

```
pipe_append_to_step_names(pip, postfix, sep = ".")
```

Arguments

pip	Pipeline object
postfix	string to be appended to each step name.
sep	string separator between step name and postfix.

Value

returns the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe")
pipe_add(p, "step1", \(x = 1) x)
pipe_add(p, "step2", \(y = 1) y)
pipe_append_to_step_names(p, "new")
p
pipe_append_to_step_names(p, "foo", sep = "__")
p
```

pipe_clone	<i>Clone pipeline</i>
------------	-----------------------

Description

Creates a copy of a pipeline object.

Usage

```
pipe_clone(pip, deep = FALSE)
```

Arguments

pip	Pipeline object
deep	logical whether to perform a deep copy

Value

returns the copied Pipeline object

Examples

```
p1 <- pipe_new("pipe")
pipe_add(p1, "step1", \(x = 1) x)
p2 <- pipe_clone(p1)
pipe_add(p2, "step2", \(y = 1) y)
p1
p2
```

pipe_collect_out	<i>Collect output from entire pipeline</i>
------------------	--

Description

Collects output after pipeline run, by default, from all steps for which keepOut was set to TRUE when steps were added (see `pipe_add()`). The output is grouped by the group names (see group parameter in `pipe_add()`), which by default are set identical to the step names.

Usage

```
pipe_collect_out(pip, groupBy = "group", all = FALSE)
```

Arguments

pip	Pipeline object
groupBy	string column of pipeline by which to group the output.
all	logical if TRUE all output is collected regardless of the keepOut flag. This can be useful for debugging.

Value

list containing the output, named after the groups, which, by default, are the steps.

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "step1", \(x = ~data) x + 2)
pipe_add(p, "step2", \(x = ~step1) x + 2, keepOut = TRUE)
pipe_run(p)
pipe_collect_out(p)
pipe_collect_out(p, all = TRUE) |> str()

# Grouped output
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "step1", \(x = ~data) x + 2, group = "add")
pipe_add(p, "step2", \(x = ~step1, y = 2) x + y, group = "add")
pipe_add(p, "step3", \(x = ~data) x * 3, group = "mult")
pipe_add(p, "step4", \(x = ~data, y = 2) x * y, group = "mult")
p

pipe_run(p)
pipe_collect_out(p, all = TRUE) |> str()

# Grouped by state
pipe_set_params(p, list(y = 5))
p

pipe_collect_out(p, groupBy = "state", all = TRUE) |> str()
```

pipe_discard_steps	<i>Discard steps from the pipeline</i>
--------------------	--

Description

Discard all steps that match a given pattern.

Usage

```
pipe_discard_steps(pip, pattern, recursive = FALSE, fixed = TRUE, ...)
```

Arguments

pip	Pipeline object
pattern	string containing a regular expression (or character string for fixed = TRUE) to be matched.
recursive	logical if TRUE the step is removed together with all its downstream dependencies.
fixed	logical If TRUE, pattern is a string to be matched as is. Overrides all conflicting arguments.
...	further arguments passed to grep() .

Value

the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "add2", \(x = ~add1) x + 2)
pipe_add(p, "mult3", \(x = ~add1) x * 3)
pipe_add(p, "mult4", \(x = ~add2) x * 4)
p

pipe_discard_steps(p, "mult")
p

# Re-add steps
pipe_add(p, "mult3", \(x = ~add1) x * 3)
pipe_add(p, "mult4", \(x = ~add2) x * 4)
p

# Discarding 'add1' does not work ...
try(pipe_discard_steps(p, "add1"))

# ... unless we enforce to remove its downstream dependencies as well
pipe_discard_steps(p, "add1", recursive = TRUE)
```

```
p  
  
# Trying to discard non-existent steps is just ignored  
pipe_discard_steps(p, "non-existent")
```

pipe_get_data	<i>Get data</i>
---------------	-----------------

Description

Get the data set for the pipeline

Usage

```
pipe_get_data(pip)
```

Arguments

pip	Pipeline object
-----	-----------------

Value

the output defined in the data step, which by default is the first step of the pipeline

Examples

```
p <- pipe_new("pipe", data = 1:2)  
pipe_get_data(p)  
pipe_set_data(p, 3:4)  
pipe_get_data(p)
```

pipe_get_depends	<i>Get step dependencies</i>
------------------	------------------------------

Description

Get step dependencies

Usage

```
pipe_get_depends(pip)  
  
pipe_get_depends_down(pip, step, recursive = TRUE)  
  
pipe_get_depends_up(pip, step, recursive = TRUE)
```

Arguments

<code>pip</code>	Pipeline object
<code>step</code>	string name of step
<code>recursive</code>	logical if TRUE, dependencies of dependencies are also returned.

Value

- `pipe_get_depends`: named list of dependencies for each step
- `pipe_get_depends_down`: list of downstream dependencies
- `pipe_get_depends_up`: list of downstream dependencies

Methods

- `pipe_get_depends`: get all dependencies for all steps defined in the pipeline
- `pipe_get_depends_down`: get all downstream dependencies of a given step, by default descending recursively.
- `pipe_get_depends_up`: get all upstream dependencies of a given step, by default descending recursively.

Examples

```
# pipe_get_depends
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "add2", \(x = ~data, y = ~add1) x + y)
pipe_get_depends(p)

# pipe_get_depends_down
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "add2", \(x = ~data, y = ~add1) x + y)
pipe_add(p, "mult3", \(x = ~add1) x * 3)
pipe_add(p, "mult4", \(x = ~add2) x * 4)
pipe_get_depends_down(p, "add1")
pipe_get_depends_down(p, "add1", recursive = FALSE)

# pipe_get_depends_up
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "add2", \(x = ~data, y = ~add1) x + y)
pipe_add(p, "mult3", \(x = ~add1) x * 3)
pipe_add(p, "mult4", \(x = ~add2) x * 4)
pipe_get_depends_up(p, "mult4")
pipe_get_depends_up(p, "mult4", recursive = FALSE)
```

pipe_get_graph	<i>Pipeline graph</i>
----------------	-----------------------

Description

Get the pipeline as a graph with nodes and edges.

Usage

```
pipe_get_graph(pip, groups = NULL)
```

Arguments

pip	Pipeline object
groups	character if not NULL, only steps belonging to the given groups are considered.

Value

list with two data frames, one for nodes and one for edges ready to be used with the [visNetwork::visNetwork\(\)](#) function of the [visNetwork](#) package.

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = ~add1) x + y)
pipe_add(p, "mult1", \(x = ~add1, y = ~add2) x * y)
graph <- pipe_get_graph(p)
graph

if (require("visNetwork", quietly = TRUE)) {
  do.call(visNetwork, args = graph)
}
```

pipe_get_out	<i>Get output of given step</i>
--------------	---------------------------------

Description

Get output of given step

Usage

```
pipe_get_out(pip, step)
```

Arguments

pip	Pipeline object
step	string name of step

Value

the output at the given step.

See Also

[pipe_collect_out\(\)](#) to collect output of multiple steps.

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "add2", \(x = ~data, y = ~add1) x + y)
pipe_run(p)
pipe_get_out(p, "add1")
pipe_get_out(p, "add2")
```

pipe_get_params	<i>Get pipeline parameters</i>
-----------------	--------------------------------

Description

Retrieves unbound function parameters defined in the pipeline where 'unbound' means parameters that are not linked to other steps.

Usage

```
pipe_get_params(pip, ignoreHidden = TRUE)

pipe_get_params_at_step(pip, step, ignoreHidden = TRUE)

pipe_get_params_unique(pip, ignoreHidden = TRUE)

pipe_get_params_unique_json(pip, ignoreHidden = TRUE)
```

Arguments

pip	Pipeline object
ignoreHidden	logical if TRUE, hidden parameters (i.e. all parameter names starting with a dot) are ignored and thus not returned.
step	string name of step

Value

- pipe_get_params: list of parameters, sorted and named by step - steps with no parameters are filtered out
- pipe_get_params_at_step: list of parameters at given step
- pipe_get_params_unique: list of parameters where each parameter is only listed once. The values of the parameters will be the values of the first step where the parameters were defined, respectively.
- get_params_unique_json: flat unnamed json list of unique parameters

Examples

```
# pipe_get_params
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = 2, .z = 3) x + y + .z)
pipe_add(p, "add3", \( ) 1 + 2)
pipe_get_params(p, ) |> str()
pipe_get_params(p, ignoreHidden = FALSE) |> str()

# pipe_get_params_at_step
pipe_get_params_at_step(p, "add2")
pipe_get_params_at_step(p, "add2", ignoreHidden = FALSE)
pipe_get_params_at_step(p, "add3")

# pipe_get_params_unique
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = 2, .z = 3) x + y + .z)
pipe_add(p, "mult1", \(x = 4, y = 5, .z = 6, b = ~add2) x * y * b)
pipe_get_params_unique(p)
pipe_get_params_unique(p, ignoreHidden = FALSE)

# get_params_unique_json
pipe_get_params_unique_json(p)
pipe_get_params_unique_json(p, ignoreHidden = FALSE)
```

pipe_get_step	<i>Get step information</i>
---------------	-----------------------------

Description

Get step information

Usage

```
pipe_get_step(pip, step)
```

```
pipe_get_step_names(pip)
```

```
pipe_get_step_number(pip, step)
```

```
pipe_has_step(pip, step)
```

Arguments

pip	Pipeline object
step	string name of step

Value

- pipe_get_step: data.table row containing the step
- pipe_get_step_names: character vector of step names
- pipe_get_step_number: the step number in the pipeline
- pipe_get_step_number: whether step exists

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = 2, z = ~add1) x + y + z)
pipe_run(p)

# pipe_get_step_names
pipe_get_step_names(p)

# get_step_number
pipe_get_step_number(p, "add1")
pipe_get_step_number(p, "add2")

# pipe_has_step
pipe_has_step(p, "add1")
pipe_has_step(p, "foo")

# pipe_get_step
add1 <- pipe_get_step(p, "add1")
add1

add1[["params"]]

add1[["fun"]]

try(p$get_step("foo")) # error: step 'foo' does not exist
```

pipe_insert_after	<i>Insert step</i>
-------------------	--------------------

Description

Insert step

Usage

```
pipe_insert_after(pip, afterStep, step, ...)
```

```
pipe_insert_before(pip, beforeStep, step, ...)
```

Arguments

pip	Pipeline object
afterStep	string name of step after which to insert
step	string name of step to insert
...	further arguments passed to pipe_add()
beforeStep	string name of step before which to insert

Value

returns the Pipeline object invisibly

Methods

- `pipe_insert_after`: insert step after a certain step of the pipeline
- `pipe_insert_before`: insert step before a certain step of the pipeline

Examples

```
# pipe_insert_after
p <- pipe_new("pipe", data = 1)
pipe_add(p, "f1", \(x = 1) x)
pipe_add(p, "f2", \(x = ~f1) x)
pipe_insert_after(p, "f1", step = "after_f1", \(x = ~f1) x)
p

# insert_before
pipe_insert_before(p, "f2", step = "before_f2", \(x = ~f1) 2 * x)
p
```

pipe_length	<i>Length of the pipeline</i>
-------------	-------------------------------

Description

Length of the pipeline

Usage

```
pipe_length(pip)
```

Arguments

pip	Pipeline object
-----	-----------------

Value

numeric length of pipeline, that is, the total number of steps

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "f1", \(x = 1) x)
pipe_add(p, "f2", \(y = 1) y)
p
pipe_length(p)
```

pipe_lock_step	<i>Lock steps</i>
----------------	-------------------

Description

Locking a step means that both its parameters and its output (given it has output) are locked such that neither setting new pipeline parameters nor future pipeline runs can change the current parameter and output content. To unlock a locked step, use [pipe_unlock_step\(\)](#).

Usage

```
pipe_lock_step(pip, step)

pipe_unlock_step(pip, step)
```

Arguments

pip	Pipeline object
step	string name of step to lock or unlock

Value

the Pipeline object invisibly

Examples

```
# pipe_lock_step
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = 1, data = ~data) x + data)
pipe_add(p, "add2", \(x = 1, data = ~data) x + data)
pipe_run(p)
pipe_get_out(p, "add1")
pipe_get_out(p, "add2")
pipe_lock_step(p, "add1")

pipe_set_data(p, 3)
pipe_set_params(p, list(x = 3))
pipe_run(p)
pipe_get_out(p, "add1")
pipe_get_out(p, "add2")

# pipe_unlock_step
pipe_unlock_step(p, "add1")
pipe_set_params(p, list(x = 3))
pipe_run(p)
pipe_get_out(p, "add1")
```

pipe_new

Create new pipeline

Description

A new pipeline is always initialized with one 'data' step, which basically is a function returning the data.

Usage

```
pipe_new(name, data = NULL, logger = NULL)
```

Arguments

name	the name of the Pipeline
data	optional data used at the start of the pipeline. The data also can be set later using the <code>pipe_set_data()</code> function.
logger	custom logger to be used for logging. If no logger is provided, the default logger is used, which should be sufficient for most use cases. If you do want to use your own custom log function, you need to provide a function that obeys the following form: function(level, msg, ...) { your custom logging code here }

The `level` argument is a string and will be one of `info`, `warn`, or `error`. The `msg` argument is a string containing the message to be logged. The `...` argument is a list of named parameters, which can be used to add additional information to the log message. Currently, this is only used to add the context in case of a step giving a warning or error.

Note that with the default logger, the log layout can be altered any time via `set_log_layout()`.

Value

returns the Pipeline object invisibly

Examples

```
data <- data.frame(x = 1:2, y = 3:4)
p <- pipe_new("myPipe", data = data)
p |> pipe_run() |> pipe_get_out("data")

# Setting data later
p <- pipe_new("myPipe")
pipe_get_data(p)

p <- pipe_set_data(p, data)
pipe_get_data(p)
p |> pipe_run() |> pipe_get_out("data")

# Initialize with custom logger
my_logger <- function(level, msg, ...) {
  cat(level, msg, "\n")
}
p <- pipe_new("myPipe", data = data, logger = my_logger)
p |> pipe_run() |> pipe_get_out("data")
```

pipe_pop_step

Pop steps from the pipeline

Description

Use this function to drop steps from the end of the pipeline.

Usage

```
pipe_pop_step(pip)

pipe_pop_steps_after(pip, step)

pipe_pop_steps_from(pip, step)
```

Arguments

pip	Pipeline object
step	string name of step

Value

string the name of the step that was removed

Methods

- pipe_pop_step: drop last step from the pipeline
- pipe_pop_steps_after: drop all steps after given steps
- pipe_pop_steps_from: drop all steps from and including given steps

Examples

```
# pipe_pop_step
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "f1", \(x = 1) x)
pipe_add(p, "f2", \(y = 1) y)
p
pipe_pop_step(p)
p

# pipe_pop_steps_after
pipe_add(p, "f2", \(y = 1) y)
pipe_add(p, "f3", \(z = 1) z)
p
pipe_pop_steps_after(p, "f1")
p

# pipe_pop_steps_from
pipe_add(p, "f2", \(y = 1) y)
pipe_add(p, "f3", \(z = 1) z)
p
pipe_pop_steps_from(p, "f1")
p
```

pipe_print	<i>Print the pipeline as a table</i>
------------	--------------------------------------

Description

Print the pipeline as a table

Usage

```
pipe_print(pip, verbose = FALSE)
```

Arguments

pip	Pipeline object
verbose	logical if TRUE, print all columns of the pipeline, otherwise only the most relevant columns are displayed.

Value

the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1:2)
p$add("f1", \(x = 1) x)
p$add("f2", \(y = 1) y)
pipe_print(p)
pipe_print(p, verbose = TRUE)

# Also works with standard print function
print(p)
print(p, verbose = TRUE)
```

pipe_remove_step	<i>Remove certain step from the pipeline.</i>
------------------	---

Description

Can be used to remove any given step. If other steps depend on the step to be removed, an error is given and the removal is blocked, unless recursive was set to TRUE.

Usage

```
pipe_remove_step(pip, step, recursive = FALSE)
```

Arguments

pip	Pipeline object
step	string the name of the step to be removed
recursive	logical if TRUE the step is removed together with all its downstream dependencies.

Value

the Pipeline object invisibly

Examples

```

p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = ~add1) x + y)
pipe_add(p, "mult1", \(x = 1, y = ~add2) x * y)
p

pipe_remove_step(p, "mult1")
p

try(pipe_remove_step(p, "add1"))
pipe_remove_step(p, "add1", recursive = TRUE)
p

```

pipe_rename_step	<i>Rename step</i>
------------------	--------------------

Description

Safely rename a step in the pipeline. If new step name would result in a name clash, an error is given.

Usage

```
pipe_rename_step(pip, from, to)
```

Arguments

pip	Pipeline object
from	string the name of the step to be renamed.
to	string the new name of the step.

Value

the Pipeline object invisibly

Examples

```

p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "add1", \(data = ~data, x = 1) x + data)
pipe_add(p, "add2", \(x = 1, y = ~add1) x + y)
p

try(pipe_rename_step(p, from = "add1", to = "add2"))

pipe_rename_step(p, from = "add1", to = "first_add")
p

```

pipe_replace_step	<i>Replace pipeline step</i>
-------------------	------------------------------

Description

Replaces an existing pipeline step.

Usage

```
pipe_replace_step(
  pip,
  step,
  fun,
  params = list(),
  description = "",
  group = step,
  keepOut = FALSE
)
```

Arguments

pip	Pipeline object
step	string the name of the step. Each step name must be unique.
fun	function or name of the function to be applied at the step. Both existing and anonymous/lambda functions can be used. All function parameters must have default values. If a parameter is missing a default value in the function signature, alternatively, it can be set via the params argument (see Examples section with mean() function).
params	list list of parameters to set or overwrite parameters of the passed function.
description	string optional description of the step
group	string output collected after pipeline execution (see function pipe_collect_out()) is grouped by the defined group names. By default, this is the name of the step, which comes in handy when the pipeline is copy-appended multiple times to keep the results of the same function/step grouped at one place.
keepOut	logical if FALSE (default) the output of the step is not collected when calling pipe_collect_out() after the pipeline run. This option is used to only keep the results that matter and skip intermediate results that are not needed. See also function pipe_collect_out() for more details.

Value

returns the Pipeline object invisibly

See Also

[pipe_add\(\)](#)

Examples

```
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y)
pipe_add(p, "add2", \(x = ~data, y = 2) x + y)
pipe_add(p, "mult", \(x = 1, y = 2) x * y, keepOut = TRUE)
pipe_run(p) |> pipe_collect_out()
pipe_replace_step(p, "mult", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
pipe_run(p) |> pipe_collect_out()
try(pipe_replace_step(p, "foo", \(x = 1) x)) # step 'foo' does not exist
```

pipe_reset

*Reset pipeline***Description**

Resets the pipeline to the state before it was run. This means that all output is removed and the state of all steps is reset to 'New'.

Usage

```
pipe_reset(pip)
```

Arguments

pip Pipeline object

Value

returns the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1:2)
pipe_add(p, "f1", \(x = 1) x)
pipe_add(p, "f2", \(y = 1) y)
pipe_run(p, )
p

pipe_reset(p)
p
```

pipe_run

*Run pipeline***Description**

Runs all new and/or outdated pipeline steps.

Usage

```
pipe_run(
  pip,
  force = FALSE,
  recursive = TRUE,
  cleanUnkept = FALSE,
  progress = NULL,
  showLog = TRUE
)
```

Arguments

pip	Pipeline object
force	logical if TRUE all steps are run regardless of whether they are outdated or not.
recursive	logical if TRUE and a step returns a new pipeline, the run of the current pipeline is aborted and the new pipeline is run recursively.
cleanUnkept	logical if TRUE all output that was not marked to be kept is removed after the pipeline run. This option can be useful if temporary results require a lot of memory.
progress	function this parameter can be used to provide a custom progress function of the form function(value, detail), which will show the progress of the pipeline run for each step, where value is the current step number and detail is the name of the step.
showLog	logical should the steps be logged during the pipeline run?

Value

returns the Pipeline object invisibly

Examples

```
# Simple pipeline
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y)
pipe_add(p, "add2", \(x = ~add1, z = 2) x + z)
pipe_add(p, "final", \(x = ~add1, y = ~add2) x * y, keepOut = TRUE)
p |> pipe_run() |> pipe_collect_out()
pipe_set_params(p, list(z = 4)) # outdates steps add2 and final
```

```

p

p |> pipe_run() |> pipe_collect_out()

pipe_run(p, cleanUnkept = TRUE)
p

# Recursive pipeline (for advanced users)
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y)
pipe_add(p, "new_pipe", \(x = ~add1) {
  p2 <- pipe_new("new_pipe", data = x)
  pipe_add(p2, "add1", \(x = ~data) x + 1)
  pipe_add(p2, "add2", \(x = ~add1) x + 2, keepOut = TRUE)
}
)
p |> pipe_run() |> pipe_collect_out()

# Run pipeline with progress bar
p <- pipe_new("pipe", data = 1)
pipe_add(p, "first step", \() Sys.sleep(0.5))
pipe_add(p, "second step", \() Sys.sleep(0.5))
pipe_add(p, "last step", \() Sys.sleep(0.5))
pb <- txtProgressBar(min = 1, max = pipe_length(p), style = 3)
fprogress <- function(value, detail) {
  setTxtProgressBar(pb, value)
}
pipe_run(p, progress = fprogress, showLog = FALSE)

```

pipe_run_step

Run specific step

Description

Run given pipeline step possibly together with upstream and/or downstream dependencies.

Usage

```

pipe_run_step(
  pip,
  step,
  upstream = TRUE,
  downstream = FALSE,
  cleanUnkept = FALSE
)

```

Arguments

pip Pipeline object

step	string name of step
upstream	logical if TRUE, run all dependent upstream steps first.
downstream	logical if TRUE, run all dependent downstream afterwards.
cleanUnkept	logical if TRUE all output that was not marked to be kept is removed after the pipeline run. This option can be useful if temporary results require a lot of memory.

Value

returns the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y)
pipe_add(p, "add2", \(x = ~add1, z = 2) x + z)
pipe_add(p, "mult", \(x = ~add1, y = ~add2) x * y)
pipe_run_step(p, "add2")

pipe_run_step(p, "add2", downstream = TRUE)

pipe_run_step(p, "mult", upstream = TRUE)
```

pipe_set_data	<i>Set data</i>
---------------	-----------------

Description

Set data in first step of pipeline.

Usage

```
pipe_set_data(pip, data)
```

Arguments

pip	Pipeline object
data	initial data set.

Value

returns the Pipeline object invisibly

Examples

```

p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y, keepOut = TRUE)
p |> pipe_run() |> pipe_collect_out()

pipe_set_data(p, 3)
p |> pipe_run() |> pipe_collect_out()

```

pipe_set_data_split	<i>Split-multiply pipeline by list of data sets</i>
---------------------	---

Description

This function can be used to apply the pipeline repeatedly to various data sets. For this, the pipeline split-copies itself by the list of given data sets. Each sub-pipeline will have one of the data sets set as input data. The step names of the sub-pipelines will be the original step names plus the name of the data set.

Usage

```

pipe_set_data_split(
  pip,
  dataList,
  toStep = character(),
  groupBySplit = TRUE,
  sep = "."
)

```

Arguments

pip	Pipeline object
dataList	list of data sets
toStep	string step name marking optional subset of the pipeline, to which the data split should be applied to.
groupBySplit	logical whether to set step groups according to data split.
sep	string separator to be used between step name and data set name when creating the new step names.

Value

new combined Pipeline with each sub-pipeline having set one of the data sets.

Examples

```
# Split by three data sets
dataList <- list(a = 1, b = 2, c = 3)
p <- pipe_new("pipe")
pipe_add(p, "add1", \(x = ~data) x + 1, keepOut = TRUE)
pipe_add(p, "mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
pipe_set_data_split(p, dataList)
p

p |> pipe_run() |> pipe_collect_out() |> str()

# Don't group output by split
p <- pipe_new("pipe")
pipe_add(p, "add1", \(x = ~data) x + 1, keepOut = TRUE)
pipe_add(p, "mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
pipe_set_data_split(p, dataList, groupBySplit = FALSE)
p

p |> pipe_run() |> pipe_collect_out() |> str()

# Split up to certain step
p <- pipe_new("pipe")
pipe_add(p, "add1", \(x = ~data) x + 1)
pipe_add(p, "mult", \(x = ~data, y = ~add1) x * y)
pipe_add(p, "average_result", \(x = ~mult) mean(unlist(x)), keepOut = TRUE)
p
pipe_get_depends(p)[["average_result"]]

pipe_set_data_split(p, dataList, toStep = "mult")
p
pipe_get_depends(p)[["average_result"]]

p |> pipe_run() |> pipe_collect_out() |> str()
```

pipe_set_keep_out	<i>Change output flag</i>
-------------------	---------------------------

Description

Change the keepOut flag at a given pipeline step, which determines whether the output of that step is collected when calling `pipe_collect_out()` after the pipeline was run. See column `keepOut` when printing a pipeline to view the status.

Usage

```
pipe_set_keep_out(pip, step, keepOut = TRUE)
```

Arguments

pip	Pipeline object
step	string name of step
keepOut	logical whether to keep output of step

Value

the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 1) x + y, keepOut = TRUE)
pipe_add(p, "add2", \(x = ~data, y = 2) x + y)
pipe_add(p, "mult", \(x = ~add1, y = ~add2) x * y)
p |> pipe_run() |> pipe_collect_out()

pipe_set_keep_out(p, "add1", keepOut = FALSE)
pipe_set_keep_out(p, "mult", keepOut = TRUE)
p |> pipe_run() |> pipe_collect_out()
```

pipe_set_params	<i>Set pipeline parameters</i>
-----------------	--------------------------------

Description

Set unbound function parameters defined in the pipeline where 'unbound' means parameters that are not linked to other steps. Trying to set parameters that don't exist in the pipeline is ignored, by default, with a warning.

Usage

```
pipe_set_params(pip, params, warnUndefined = TRUE)
```

Arguments

pip	Pipeline object
params	list of parameters to be set
warnUndefined	logical whether to give a warning when trying to set a parameter that is not defined in the pipeline.

Value

returns the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 2) x + y)
pipe_add(p, "add2", \(x = ~data, y = 3) x + y)
pipe_add(p, "mult", \(x = 4, z = 5) x * z)
pipe_get_params(p)
pipe_set_params(p, params = list(x = 3, y = 3))
pipe_get_params(p)
pipe_set_params(p, params = list(x = 5, z = 3))
pipe_get_params(p)

suppressWarnings(
  pipe_set_params(p, list(foo = 3)) # gives warning as 'foo' is undefined
)
pipe_set_params(p, list(foo = 3), warnUndefined = FALSE)
```

```
pipe_set_params_at_step
```

Set parameters at step

Description

Set unbound function parameters defined at given pipeline step where 'unbound' means parameters that are not linked to other steps. If one or more parameters don't exist, an error is given.

Usage

```
pipe_set_params_at_step(pip, step, params)
```

Arguments

pip	Pipeline object
step	string the name of the step
params	list of parameters to be set

Value

returns the Pipeline object invisibly

Examples

```
p <- pipe_new("pipe", data = 1)
pipe_add(p, "add1", \(x = ~data, y = 2, z = 3) x + y)
pipe_set_params_at_step(p, step = "add1", params = list(y = 5, z = 6))
pipe_get_params(p)

try(
  pipe_set_params_at_step(p, step = "add1", params = list(foo = 3))
)
```

pipe_split

*Split-up pipeline***Description**

Splits pipeline into its independent parts. This can be useful, for example, to split-up the pipeline in order to run each part in parallel.

Usage

```
pipe_split(pip)
```

Arguments

pip Pipeline object

Value

list of Pipeline objects

Examples

```
# Example for two independent calculation paths
p <- pipe_new("pipe", data = 1)
pipe_add(p, "f1", \(x = ~data) x)
pipe_add(p, "f2", \(x = 1) x)
pipe_add(p, "f3", \(x = ~f1) x)
pipe_add(p, "f4", \(x = ~f2) x)
pipe_split(p)

# Example of split by three data sets
dataList <- list(a = 1, b = 2, c = 3)
p <- pipe_new("pipe")
pipe_add(p, "add1", \(x = ~data) x + 1, keepOut = TRUE)
pipe_add(p, "mult", \(x = ~data, y = ~add1) x * y, keepOut = TRUE)
pipes <- pipe_set_data_split(p, dataList) |> pipe_split()
pipes
```

set_log_layout

*Set pipeflow log layout***Description**

This function provides an easy way to set the basic log layout of the pipeline logging. For a fine-grained control of the logger, which you can retrieve via `lgr::get_logger("pipeflow")`, see e.g. the [logger_config](#) function from the [lgr](#) package.

Usage

```
set_log_layout(layout = c("text", "json"))
```

Arguments

layout Layout name, which at this point can be either 'text' or 'json'.

Value

invisibly returns a Logger object

Examples

```
p <- Pipeline$new("pipe", data = 1:2)
p$add("add1", \(data = ~data, x = 1) x + data)
p$run()

lg <- set_log_layout("json")
print(lg)

p$run()

set_log_layout("text")
p$run()
```

Index

`grep()`, [8](#), [40](#)

`lgr`, [63](#)

`logger_config`, [63](#)

`mean()`, [5](#), [35](#), [54](#)

`pipe_add`, [35](#)

`pipe_add()`, [39](#), [47](#), [54](#)

`pipe_append`, [36](#)

`pipe_append_to_step_names`, [37](#)

`pipe_clone`, [38](#)

`pipe_collect_out`, [39](#)

`pipe_collect_out()`, [35](#), [44](#), [54](#), [60](#)

`pipe_discard_steps`, [40](#)

`pipe_get_data`, [41](#)

`pipe_get_depends`, [41](#)

`pipe_get_depends_down`
 (`pipe_get_depends`), [41](#)

`pipe_get_depends_up` (`pipe_get_depends`),
 [41](#)

`pipe_get_graph`, [43](#)

`pipe_get_out`, [43](#)

`pipe_get_params`, [44](#)

`pipe_get_params_at_step`
 (`pipe_get_params`), [44](#)

`pipe_get_params_unique`
 (`pipe_get_params`), [44](#)

`pipe_get_params_unique_json`
 (`pipe_get_params`), [44](#)

`pipe_get_step`, [45](#)

`pipe_get_step_names` (`pipe_get_step`), [45](#)

`pipe_get_step_number` (`pipe_get_step`), [45](#)

`pipe_has_step` (`pipe_get_step`), [45](#)

`pipe_insert_after`, [47](#)

`pipe_insert_before` (`pipe_insert_after`),
 [47](#)

`pipe_length`, [48](#)

`pipe_lock_step`, [48](#)

`pipe_new`, [49](#)

`pipe_pop_step`, [50](#)

`pipe_pop_steps_after` (`pipe_pop_step`), [50](#)

`pipe_pop_steps_from` (`pipe_pop_step`), [50](#)

`pipe_print`, [51](#)

`pipe_remove_step`, [52](#)

`pipe_rename_step`, [53](#)

`pipe_replace_step`, [54](#)

`pipe_reset`, [55](#)

`pipe_run`, [56](#)

`pipe_run_step`, [57](#)

`pipe_set_data`, [58](#)

`pipe_set_data()`, [49](#)

`pipe_set_data_split`, [59](#)

`pipe_set_keep_out`, [60](#)

`pipe_set_params`, [61](#)

`pipe_set_params_at_step`, [62](#)

`pipe_split`, [63](#)

`pipe_unlock_step` (`pipe_lock_step`), [48](#)

`pipe_unlock_step()`, [48](#)

`Pipeline`, [2](#)

`set_log_layout`, [63](#)

`set_log_layout()`, [4](#), [50](#)

`visNetwork`, [43](#)

`visNetwork::visNetwork()`, [43](#)