

# Package: pipeflow (via r-universe)

December 6, 2024

**Title** Lightweight, General-Purpose Data Analysis Pipelines

**Version** 0.2.1

**Description** A lightweight yet powerful framework for building robust data analysis pipelines. With 'pipeflow', you initialize a pipeline with your dataset and construct your workflow step by step by seamlessly adding R functions. Modify, remove, or insert steps at any stage while 'pipeflow' ensures the integrity and correctness of your pipeline. Designed to help you focus on the 'what' rather than the 'how', this package simplifies the implementation of complex workflows, making even large-scale data analysis projects manageable, adaptable, and reusable with ease.

**License** GPL-3

**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

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.3.2

**NeedsCompilation** no

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

**Repository** <https://rpahl.r-universe.dev>

**RemoteUrl** <https://github.com/rpahl/pipeflow>

**RemoteRef** HEAD

**RemoteSha** 425e3df89798e4e4e1920a633ade26f09358227e

## Contents

Pipeline . . . . .	2
pipelineAliases . . . . .	32
set_log_layout . . . . .	34

<b>Index</b>	<b>36</b>
--------------	-----------

---

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 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$lock_step()`
- `Pipeline$print()`
- `Pipeline$pop_step()`
- `Pipeline$pop_steps_after()`
- `Pipeline$pop_steps_from()`
- `Pipeline$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 lambda/anonymous functions can be used.

**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

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")
```

```

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

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. The `append` takes care of name clashes and dependencies, which will be changed after the `append`.

*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.

`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)

p3 <- Pipeline$new("pipe3")
p3$add("step1", \(z = 1) z)
p1$append(p2)$append(p3)

```

**Method** `append_to_step_names()`: Append string to all step names. Also takes care of updating 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("new", 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 if not set explicitly corresponds 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()

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 the 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$discard_steps("mult")
p

# Re-add steps
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p
# Discard step 'add1' doesn't work as 'add2' and 'mult3' depend on it
try(p$discard_steps("add1"))
p$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)
if (require("visNetwork", quietly = TRUE)) {
  do.call(visNetwork, args = p$get_graph())
}

```

**Method** `get_out()`: Get output of given step after pipeline run.

*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()`: Get all unbound (i.e. not referring to other steps) function parameters defined in the pipeline.

*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. If step not found, an error is given.

*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[["out"]]
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()`: Determine whether 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. If it does not have output, only the parameters are locked. Locking a step is useful if the step happens to share parameter names with other steps but should not be affected when parameters are set commonly for the entire pipeline (see function `set_params` below).

*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** `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 a subset of columns is printed.

*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** `pop_step()`: Remove 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()`: Remove 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()`: Remove 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** `remove_step()`: Remove certain step from the pipeline. If step does not exist, an error is given. If other steps depend on the step to be removed, an error is given, unless `recursive = 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()`: Replace 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$run()$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` data.frame 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()`: Split-copy pipeline by list of 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$set_data_split(
  dataList,
  toStep = utils::tail(self$get_step_names(), 1),
  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)
p3 <- p$set_data_split(dataList)
p3
p3$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)
p3 <- p$set_data_split(dataList, groupBySplit = FALSE)
p3
p3$run()$collect_out() |> str()
```

**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.

*Usage:*

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

*Arguments:*

`params` list of parameters to be set

`warnUndefined` logical whether to give a warning if a parameter 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 = 1) x + y)
p$add("add2", \(x = ~data, y = 1) x + y)
p$add("mult", \(x = 1, z = 1) 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)) # warning: trying to set undefined
)
```

**Method** `set_params_at_step()`: Set unbound parameter values at given pipeline step.

*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 = 1, z = 2) x + y)
p$add("add2", \(x = ~data, y = 1, z = 2) x + y)
p$set_params_at_step("add1", list(y = 3, z = 3))
p$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)
pips <- p$set_data_split(dataList)$split()

```

**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

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")

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

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)

```

```

p3 <- Pipeline$new("pipe3")
p3$add("step1", \(z = 1) z)
p1$append(p2)$append(p3)

## -----
## 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("new", 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()

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$discard_steps("mult")
p

```

```

# Re-add steps
p$add("mult3", \(x = ~add1) x * 3)
p$add("mult4", \(x = ~add2) x * 4)
p
# Discard step 'add1' doesn't work as 'add2' and 'mult3' depend on it
try(p$discard_steps("add1"))
p$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)
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[["out"]]
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$print`
## -----

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

## -----
## 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$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$run()$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)
p3 <- p$set_data_split(dataList)
p3
p3$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)
p3 <- p$set_data_split(dataList, groupBySplit = FALSE)
p3
p3$run()$collect_out() |> str()

## -----
## 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 = 1) x + y)
p$add("add2", \(x = ~data, y = 1) x + y)
p$add("mult", \(x = 1, z = 1) 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)) # warning: trying to set undefined
)

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

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

```

```

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)
pips <- p$set_data_split(dataList)$split()

## -----
## 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()

```

---

pipelineAliases

*Pipeline alias functions*

---

## Description

Alias functions, one for each member function of a Pipeline object.

## Usage

```
pipe_add(pip, ...)
```

```
pipe_append(pip, ...)
```

```
pipe_append_to_step_names(pip, ...)
```

```
pipe_clone(pip, ...)  
pipe_collect_out(pip, ...)  
pipe_discard_steps(pip, ...)  
pipe_get_data(pip, ...)  
pipe_get_depends(pip, ...)  
pipe_get_depends_down(pip, ...)  
pipe_get_depends_up(pip, ...)  
pipe_get_graph(pip, ...)  
pipe_get_out(pip, ...)  
pipe_get_params(pip, ...)  
pipe_get_params_at_step(pip, ...)  
pipe_get_params_unique(pip, ...)  
pipe_get_params_unique_json(pip, ...)  
pipe_get_step(pip, ...)  
pipe_get_step_names(pip, ...)  
pipe_get_step_number(pip, ...)  
pipe_has_step(pip, ...)  
pipe_insert_after(pip, ...)  
pipe_insert_before(pip, ...)  
pipe_length(pip, ...)  
pipe_lock_step(pip, ...)  
pipe_new(...)  
pipe_print(pip, ...)  
pipe_pop_step(pip, ...)
```

```
pipe_pop_steps_after(pip, ...)  
pipe_pop_steps_from(pip, ...)  
pipe_remove_step(pip, ...)  
pipe_rename_step(pip, ...)  
pipe_replace_step(pip, ...)  
pipe_reset(pip, ...)  
pipe_run(pip, ...)  
pipe_run_step(pip, ...)  
pipe_set_data(pip, ...)  
pipe_set_data_split(pip, ...)  
pipe_set_keep_out(pip, ...)  
pipe_set_params(pip, ...)  
pipe_set_params_at_step(pip, ...)  
pipe_split(pip, ...)  
pipe_unlock_step(pip, ...)
```

**Arguments**

pip	A pipeline object
...	Arguments passed to the respective pipeline method

**Value**

The result of the respective pipeline method

---

set_log_layout	<i>Set pipeflow log layout</i>
----------------	--------------------------------

---

**Description**

Set pipeflow log layout

**Usage**

```
set_log_layout(layout)
```

**Arguments**

layout	Layout name
--------	-------------

**Value**

invisibly returns 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()`, 7

`pipe_add` (pipelineAliases), 32

`pipe_append` (pipelineAliases), 32

`pipe_append_to_step_names`  
(pipelineAliases), 32

`pipe_clone` (pipelineAliases), 32

`pipe_collect_out` (pipelineAliases), 32

`pipe_discard_steps` (pipelineAliases), 32

`pipe_get_data` (pipelineAliases), 32

`pipe_get_depends` (pipelineAliases), 32

`pipe_get_depends_down`  
(pipelineAliases), 32

`pipe_get_depends_up` (pipelineAliases),  
32

`pipe_get_graph` (pipelineAliases), 32

`pipe_get_out` (pipelineAliases), 32

`pipe_get_params` (pipelineAliases), 32

`pipe_get_params_at_step`  
(pipelineAliases), 32

`pipe_get_params_unique`  
(pipelineAliases), 32

`pipe_get_params_unique_json`  
(pipelineAliases), 32

`pipe_get_step` (pipelineAliases), 32

`pipe_get_step_names` (pipelineAliases),  
32

`pipe_get_step_number` (pipelineAliases),  
32

`pipe_has_step` (pipelineAliases), 32

`pipe_insert_after` (pipelineAliases), 32

`pipe_insert_before` (pipelineAliases), 32

`pipe_length` (pipelineAliases), 32

`pipe_lock_step` (pipelineAliases), 32

`pipe_new` (pipelineAliases), 32

`pipe_pop_step` (pipelineAliases), 32

`pipe_pop_steps_after` (pipelineAliases),  
32

`pipe_pop_steps_from` (pipelineAliases),  
32

`pipe_print` (pipelineAliases), 32

`pipe_remove_step` (pipelineAliases), 32

`pipe_rename_step` (pipelineAliases), 32

`pipe_replace_step` (pipelineAliases), 32

`pipe_reset` (pipelineAliases), 32

`pipe_run` (pipelineAliases), 32

`pipe_run_step` (pipelineAliases), 32

`pipe_set_data` (pipelineAliases), 32

`pipe_set_data_split` (pipelineAliases),  
32

`pipe_set_keep_out` (pipelineAliases), 32

`pipe_set_params` (pipelineAliases), 32

`pipe_set_params_at_step`  
(pipelineAliases), 32

`pipe_split` (pipelineAliases), 32

`pipe_unlock_step` (pipelineAliases), 32

Pipeline, 2

pipelineAliases, 32

`set_log_layout`, 34

`set_log_layout()`, 3