Package 'ravepipeline'

May 30, 2026

Type Package

Title Reproducible Pipeline Infrastructure for Neuroscience

Version 0.1.0

Language en-US

Description Defines the underlying pipeline structure for reproducible neuroscience, adopted by 'RAVE' (reproducible analysis and visualization of intracranial electroencephalography); provides high-level class definition to build, compile, set, execute, and share analysis pipelines. Both R and 'Python' are supported, with 'Markdown' and 'shiny' dashboard templates for extending and building customized pipelines. See the full documentations at <https://rave.wiki>; to cite us, check out our paper by Magnotti, Wang, and Beauchamp (2020, <doi:10.1016/j.neuroimage.2020.117341>), or run citation(``ravepipeline'') for details.

Copyright Trustees of University of Pennsylvania owns the copyright of the package unless otherwise stated. Zhengjia Wang owns the copyright of all the low-level functions included in 'R/common.R', 'R/fastmap2', 'R/fastqueue2.R', 'R/filesys.R', 'R/fst.R', 'R/json.R', 'R/os_info.R', 'R/parallel.R', 'R/progress.R', 'R/simplelocker.R', 'R/yaml.R', and all the template files under 'inst/rave-pipelines' and 'inst/rave-modules', these files are licensed under 'MIT'.

License MIT + file LICENSE

Encoding UTF-8

URL https://dipterix.org/ravepipeline/, https://rave.wiki, https://github.com/dipterix/ravepipeline

BugReports https://github.com/dipterix/ravepipeline/issues

Imports stats, tools, utils, base64enc, callr, cli, digest, fastmap, future, fst (>= 0.9.8), glue, jsonlite, knitr, promises, R6, remotes, rlang, targets, uuid, yaml, logger

Suggests ellmer (>= 0.4.0), dipsaus, filearray, future.apply, globals, ieegio, pkgsearch, rpymat, rmarkdown, rstudioapi, shidashi, threeBrain, testthat (>= 3.0.0), visNetwork, later, shiny, mirai

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Zhengjia Wang [aut, cre, cph], John Magnotti [ctb, res], Xiang Zhang [ctb, res], Michael Beauchamp [ctb, res], Trustees of University of Pennsylvania [cph] (Copyright Holder)

Maintainer Zhengjia Wang <dipterix.wang@gmail.com>

Repository CRAN

Date/Publication 2026-05-30 05:10:03 UTC

Contents

`base64-utils` *Convert from and to 'base64' string*

Description

Encode or decode 'base64' raw or url-safe string

Usage

```
base64_urlencode(x)

base64_encode(x)

base64_urldecode(x)

base64_decode(x)

base64_plot(
  expr,
  width = 480,
  height = 480,
  ...,
  quoted = FALSE,
  envir = parent.frame()
)
```

Arguments

`x`	for encoders, this is an R raw or character vectors; for decoders this is 'base64' encoded strings
`expr`	expression for plot, will saved to a png and converted to 'base64' string
`width, height`	image size in pixels
`...`	passed to png
`quoted, envir`	non-standard evaluation settings

Value

`base64_encode`, `base64_plot` returns 'base64' string in raw format; `base64_urlencode` returns 'base64' string url-safe format; `base64_urldecode` returns the original string; `base64_decode` returns original raw vectors.

Examples

```
# ---- For direct base64URI -----------------------------------


file_raw <- as.raw(1:255)

# raw base64
base64_raw <- base64_encode(file_raw)
base64_raw

as.integer(base64_decode(base64_raw))

# ---- For URL-save base64 ------------------------------------
# Can be used in URL
base64_url <- base64_urlencode(
  paste(c(letters, LETTERS, 0:9),
  collapse = ""))
base64_url

base64_urldecode(base64_url)

# ---- Convert R plots to base64 ------------------------------
img <- base64_plot({
  plot(1:10)
}, width = 320, height = 320)

# summary
print(img)

# get base64 content
img_base64 <- format(img, type = "content")

# save to png
tmppng <- tempfile(fileext = ".png")
writeBin(base64_decode(img_base64), con = tmppng)

# cleanup
unlink(tmppng)

# Format as svg
format(img, type = "html_svg")
```

`dir_create2` *Force creating directory with checks*

Description

Force creating directory with checks

Usage

```
dir_create2(x, showWarnings = FALSE, recursive = TRUE, check = TRUE, ...)
```

Arguments

x	path to create
showWarnings, recursive, ...	passed to dir.create
check	whether to check the directory after creation

Value

Normalized path

Examples

```
path <- file.path(tempfile(), 'a', 'b', 'c')

# The following are equivalent
dir.create(path, showWarnings = FALSE, recursive = TRUE)

dir_create2(path)
```

---

html-embed *Embed binary data or JSON strings into HTML files*

Description

`html_embed_write` encodes JSON strings, plain-text strings, and binary files as base64 `<script>` tags and injects them into an HTML file.

`html_embed_read` reads `<script>` tags back out of a saved HTML file and reconstructs the original data.

Usage

```
html_embed_read(path, name = NULL, parse_json = TRUE, update = FALSE)

html_embed_write(
  html_path,
  json_string = list(),
  text_string = list(),
  binary_paths = list(),
  missing_action = c("error", "warning", "ignore")
)
```

Arguments

path	character path to an HTML file, or a manifest object returned by a previous call to `html_embed_read(path, name = NULL)`.
name	character; the `data-for` name of the entry to decode. When `NULL` (default) the function returns a manifest object that lists all embedded entries without decoding them.
parse_json	logical; when `TRUE` (default) JSON entries are parsed with `jsonlite::fromJSON` before being returned.
update	logical; when `FALSE` (default) already-decoded entries cached in the manifest are returned as-is without re-reading the file.
html_path	character; path to the HTML file to write. If the file does not exist, behavior is controlled by `missing_action`.
json_string	named list of character strings; each element is a UTF-8 JSON string. The list name becomes the `data-for` attribute.
text_string	named list of character strings; each element is an arbitrary UTF-8 plain-text string. The list name becomes the `data-for` attribute.
binary_paths	named list of character strings; each element is an absolute path to a binary file to embed. The list name becomes the `data-for` attribute.
missing_action	character; what to do when `html_path` does not exist. `"error"` (default) stops with an error; `"warning"` emits a warning and creates the file; `"ignore"` creates the file silently.

Details

`html_embed_write` streams data after `</body>` (or before `</html>`, or appends when neither tag is found). Large inputs are split into $\approx 48\,\mathrm{KB}$ chunks; each chunk gets its own `<script>` tag with a sequential `data-partition` index.

`html_embed_read`: when `name` is `NULL` it returns a manifest object that lists all embedded entries; subsequent calls with a specific `name` use seek positions stored in the manifest to retrieve only the requested partitions. Files written by compatible tools (e.g. **threeBrain**) are handled transparently.

The per-entry `<script>` tag format:

```
<script type='text/plain;charset=UTF-8'
        data-for='<name>'
        data-partition='<N>'
        data-type='application/json|text/plain|application/octet-stream'
        data-size='<total bytes>'
        data-start='<byte offset>'
        data-partition-size='<this chunk bytes>'>
BASE64 (72-character wrapped lines)
</script>
```

Value

`html_embed_write`: `html_path`, invisibly.

`html_embed_read`: when `name` is `NULL`, a manifest object of class `ravepipeline_html_embed_manifest` listing all embedded entries. When `name` is specified the manifest is returned with the requested entry decoded and cached; access it via `manifest$content[[name]]`: a character string for JSON/text data, or a raw vector for binary data.

Examples

```
html_file <- tempfile(fileext = ".html")
writeLines(
  c("<html>", "<head></head>", "<body></body>", "</html>"),
  html_file
)

# ---- Write: embed JSON and binary data into an HTML file --------
tmp <- tempfile(fileext = ".bin")
writeBin(as.raw(0:255), tmp)

html_embed_write(
  html_file,
  json_string  = list(meta = '{"version":1}'),
  text_string  = list(note = "hello world"),
  binary_paths = list(data = tmp)
)

# ---- Read: list all embedded entries ----------------------------
manifest <- html_embed_read(html_file)
print(manifest)

# ---- Read: decode a specific entry ------------------------------
manifest <- html_embed_read(html_file, name = "meta")
manifest$content[["meta"]]   # character (JSON string or parsed object)

manifest <- html_embed_read(manifest, name = "data")
manifest$content[["data"]]   # raw vector

unlink(c(tmp, html_file))
```

---

install_modules *Install 'RAVE' modules*

Description

Low-level function exported for down-stream 'RAVE' packages.

Usage

```
install_modules(modules, dependencies = FALSE)
```

Arguments

modules	a vector of characters, repository names; default is to automatically determined from a public registry
dependencies	whether to update dependent packages; default is false

Value

nothing

---

logger *Logger system used by 'RAVE'*

Description

Keep track of messages printed by modules or functions

Usage

```
logger(
  ...,
  level = c("info", "success", "warning", "error", "fatal", "debug", "trace"),
  calc_delta = "auto",
  .envir = parent.frame(),
  .sep = "",
  use_glue = FALSE,
  reset_timer = FALSE
)

set_logger_path(root_path, max_bytes, max_files)

logger_threshold(
  level = c("info", "success", "warning", "error", "fatal", "debug", "trace"),
  module_id,
  type = c("console", "file", "both")
)

logger_error_condition(cond, level = "error")
```

Arguments

..., .envir, .sep	passed to glue, if use_glue is true
level	the level of message, choices are 'info' (default), 'success', 'warning', 'error', 'fatal', 'debug', 'trace'

calc_delta	whether to calculate time difference between current message and previous message; default is 'auto', which prints time difference when level is 'debug'. This behavior can be changed by altering calc_delta by a logical TRUE to enable or FALSE to disable.
use_glue	whether to use glue to combine ...; default is false
reset_timer	whether to reset timer used by calc_delta
root_path	root directory if you want log messages to be saved to hard disks; if root_path is NULL, "", or nullfile, then logger path will be unset.
max_bytes	maximum file size for each logger partitions
max_files	maximum number of partition files to hold the log; old files will be deleted.
module_id	'RAVE' module identification string, or name-space; default is 'base'
type	which type of logging should be set; default is 'console', if file log is enabled through set_logger_path, type could be 'file' or 'both'. Default log level is 'info' on console and 'debug' on file.
cond	condition to log

Value

The message without time-stamps

Examples

```
logger("This is a message")

a <- 1
logger("A message with glue: a={a}")

logger("A message without glue: a={a}", use_glue = FALSE)


logger("Message A", calc_delta = TRUE, reset_timer = TRUE)
logger("Seconds before logging another message", calc_delta = TRUE)


# by default, debug and trace messages won't be displayed
logger('debug message', level = 'debug')

# adjust logger level, make sure `module_id` is a valid RAVE module ID
logger_threshold('debug', module_id = NULL)

# Debug message will display
logger('debug message', level = 'debug')

# Trace message will not display as it's lower than debug level
logger('trace message', level = 'trace')
```

`module_add` *Add new 'RAVE' (2.0) module to current project*

Description

Creates a 'RAVE' pipeline with additional dashboard module from template.

Usage

```
module_add(
  module_id,
  module_label,
  path = ".",
  type = c("default", "bare", "scheduler", "python"),
  ...,
  pipeline_name = module_id,
  overwrite = FALSE
)
```

Arguments

`module_id`	module ID to create, must be unique; users cannot install two modules with identical module ID. We recommend that a module ID follows snake format, starting with lab name, for example, `'beauchamplab_imaging_preprocess'`, `'karaslab_freez'`, or `'upenn_ese25_fooof'`.
`module_label`	a friendly label to display in the dashboard
`path`	project root path; default is current directory
`type`	template to choose, options are `'default'` and `'bare'`
`...`	additional configurations to the module such as `'order'`, `'group'`, `'badge'`
`pipeline_name`	the pipeline name to create along with the module; default is identical to `module_id` (strongly recommended); leave it default unless you know what you are doing.
`overwrite`	whether to overwrite existing module if module with same ID exists; default is false

Value

Nothing.

Examples

```
# For demonstrating this example only
project_root <- tempfile()
dir.create(project_root, showWarnings = FALSE, recursive = TRUE)
```

```
# Add a module
module_id <- "mylab_my_first_module"
module_add(
  module_id = module_id,
  module_label = "My Pipeline",
  path = project_root
)


# show the structure
cat(
  list.files(
    project_root,
    recursive = TRUE,
    full.names = FALSE,
    include.dirs = TRUE
  ),
  sep = "\n"
)

unlink(project_root, recursive = TRUE)
```

---

module_registry *'RAVE' module registry*

Description

Create, view, or reserve the module registry

Usage

```
module_registry(
  title,
  repo,
  modules,
  authors,
  url = sprintf("https://github.com/%s", repo)
)

module_registry2(repo, description)

get_modules_registries(update = NA)

get_module_description(path)

add_module_registry(title, repo, modules, authors, url, dry_run = FALSE)
```

Arguments

title	title of the registry, usually identical to the description title in 'DESCRIPTION' or RAVE-CONFIG file
repo	'Github' repository
modules	characters of module ID, must only contain letters, digits, underscore, dash; must not be duplicated with existing registered modules
authors	a list of module authors; there must be one and only one author with 'cre' role (see person). This author will be considered maintainer, who will be in charge if editing the registry
url	the web address of the repository
update	whether to force updating the registry
path, description	path to 'DESCRIPTION' or RAVE-CONFIG file
dry_run	whether to generate and preview message content instead of opening an email link

Details

A 'RAVE' registry contains the following data entries: repository title, name, 'URL', authors, and a list of module IDs. 'RAVE' requires that each module must use a unique module ID. It will cause an issue if two modules share the same ID. Therefore 'RAVE' maintains a public registry list such that the module maintainers can register their own module ID and prevent other people from using it.

To register your own module ID, please use add_module_registry to validate and send an email to the 'RAVE' development team.

Value

a registry object, or a list of registries

Examples

```
library(ravepipeline)

# create your own registry
module_registry(
  repo = "rave-ieeg/rave-pipelines",
  title = "A Collection of 'RAVE' Builtin Pipelines",
  authors = list(
    list("Zhengjia", "Wang", role = c("cre", "aut"),
         email = "dipterix@rave.wiki")
  ),
  modules = "brain_viewer"
)


## Not run:
```

```
# This example will need access to Github and will open an email link

# get current registries
get_modules_registries(FALSE)

# If your repository is on Github and RAVE-CONFIG file exists
module_registry2("rave-ieeg/rave-pipelines")

# send a request to add your registry
registry <- module_registry2("rave-ieeg/rave-pipelines")
add_module_registry(registry)


## End(Not run)
```

`pipeline` *Creates 'RAVE' pipeline instance*

Description

Set pipeline inputs, execute, and read pipeline outputs

Usage

```
pipeline(
  pipeline_name,
  settings_file = "settings.yaml",
  paths = pipeline_root(),
  temporary = FALSE
)

pipeline_from_path(path, settings_file = "settings.yaml")
```

Arguments

`pipeline_name`	the name of the pipeline, usually title field in the `'DESCRIPTION'` file, or the pipeline folder name (if description file is missing)
`settings_file`	the name of the settings file, usually stores user inputs
`paths`	the paths to search for the pipeline, usually the parent directory of the pipeline; default is `pipeline_root`, which only search for pipelines that are installed or in current working directory.
`temporary`	see `pipeline_root`
`path`	the pipeline folder

Value

A PipelineTools instance

Examples

```
library(ravepipeline)

if(interactive()) {

  # ------------ Set up a bare minimal example pipeline ---------------
  root_path <- tempdir()
  pipeline_root_folder <- file.path(root_path, "modules")

  # create pipeline folder
  pipeline_path <- pipeline_create_template(
    root_path = pipeline_root_folder, pipeline_name = "raveio_demo",
    overwrite = TRUE, activate = FALSE, template_type = "rmd-bare")

  # Set initial user inputs
  yaml::write_yaml(
    x = list(
      n = 100,
      pch = 16,
      col = "steelblue"
    ),
    file = file.path(pipeline_path, "settings.yaml")
  )

  # build the pipeline for the first time
  # this is a one-time setup
  pipeline_build(pipeline_path)

  # Temporarily redirect the pipeline project root
  # to `root_path`
  old_opt <- options("raveio.pipeline.project_root" = root_path)
  # Make sure the options are reset
  on.exit({ options(old_opt) })

  # Compile the pipeline document
  pipeline_render(
    module_id = "raveio_demo",
    project_path = root_path
  )

  ## Not run:

    # Open web browser to see compiled report
    utils::browseURL(file.path(pipeline_path, "main.html"))


## End(Not run)
```

```
  # ---------------------- Example starts ------------------------

  # Load pipeline
  pipeline <- pipeline(
    pipeline_name = "raveio_demo",
    paths = pipeline_root_folder,
    temporary = TRUE
  )

  # Check which pipeline targets to run
  pipeline$target_table

  # Run to `plot_data`, RAVE pipeline will automatically
  # calculate which up-stream targets need to be updated
  # and evaluate these targets
  pipeline$run("plot_data")

  # Customize settings
  pipeline$set_settings(pch = 2)

  # Run again with the new inputs, since input_data does not change,
  # the pipeline will skip that target automatically
  pipeline$run("plot_data")

  # Read intermediate data
  head(pipeline$read("input_data"))

  # or use `[]` to get results
  pipeline[c("n", "pch", "col")]
  pipeline[-c("input_data")]

  # Check evaluating status
  pipeline$progress("details")

  # result summary & cache table
  pipeline$result_table

  # visualize the target dependency graph
  pipeline$visualize(glimpse = TRUE)

  # ---------------------- Clean up ------------------------
  unlink(pipeline_path, recursive = TRUE)
}
```

---

pipeline-knitr-markdown

Configure 'rmarkdown' files to build 'RAVE' pipelines

Description

Allows building 'RAVE' pipelines from `'rmarkdown'` files. Please use it in `'rmarkdown'` scripts only. Use `pipeline_create_template` to create an example.

Usage

```
configure_knitr(languages = c("R", "python"))

pipeline_setup_rmd(
  module_id,
  env = parent.frame(),
  collapse = TRUE,
  comment = "#>",
  languages = c("R", "python"),
  project_path = getOption("raveio.pipeline.project_root", default =
    rs_active_project(child_ok = TRUE, shiny_ok = TRUE))
)

pipeline_render(
  module_id,
  ...,
  env = new.env(parent = parent.frame()),
  entry_file = "main.Rmd",
  project_path = getOption("raveio.pipeline.project_root", default =
    rs_active_project(child_ok = TRUE, shiny_ok = TRUE))
)
```

Arguments

`languages`	one or more programming languages to support; options are `'R'` and `'python'`
`module_id`	the module ID, usually the name of direct parent folder containing the pipeline file
`env`	environment to set up the pipeline translator
`collapse, comment`	passed to `set` method of `opts_chunk`
`project_path`	the project path containing all the pipeline folders, usually the active project folder
`...`	passed to internal function calls
`entry_file`	the file to compile; default is `"main.Rmd"`

Value

A function that is supposed to be called later that builds the pipeline scripts

Examples

```
configure_knitr("R")

## Not run:

# Requires to configure Python
configure_knitr("python")

# This function must be called in an Rmd file setup block
# for example, see
# https://rave.wiki/posts/customize_modules/python_module_01.html

pipeline_setup_rmd("my_module_id")


## End(Not run)
```

`PipelineCollections` *Connect and schedule pipelines*

Description

Experimental, subject to change in the future.

Public fields

`verbose` whether to verbose the build

Active bindings

`root_path` path to the directory that contains pipelines and scheduler

`collection_path` path to the pipeline collections

`pipeline_ids` pipeline ID codes

Methods

Public methods:

- `PipelineCollection$new()`
- `PipelineCollection$add_pipeline()`
- `PipelineCollection$build_pipelines()`
- `PipelineCollection$run()`
- `PipelineCollection$get_scheduler()`

`PipelineCollection$new()`: Constructor

Usage:

```
PipelineCollection$new(root_path = NULL, overwrite = FALSE)
```

Arguments:

`root_path` where to store the pipelines and intermediate results

`overwrite` whether to overwrite if `root_path` exists

`PipelineCollection$add_pipeline()`: Add pipeline into the collection

Usage:

```
PipelineCollection$add_pipeline(
  x,
  names = NULL,
  deps = NULL,
  pre_hook = NULL,
  post_hook = NULL,
  cue = c("always", "thorough", "never"),
  search_paths = pipeline_root(),
  standalone = TRUE,
  hook_envir = parent.frame()
)
```

Arguments:

`x` a pipeline name (can be found via `pipeline_list`), or a `PipelineTools`

`names` pipeline targets to execute

`deps` pipeline IDs to depend on; see 'Values' below

`pre_hook` function to run before the pipeline; the function needs two arguments: input map (can be edit in-place), and path to a directory that allows to store temporary files

`post_hook` function to run after the pipeline; the function needs two arguments: pipeline object, and path to a directory that allows to store intermediate results

`cue` whether to always run dependence

`search_paths` where to search for pipeline if `x` is a character; ignored when `x` is a pipeline object

`standalone` whether the pipeline should be standalone, set to `TRUE` if the same pipeline added multiple times should run independently; default is true

`hook_envir` where to look for global environments if `pre_hook` or `post_hook` contains global variables; default is the calling environment

Returns: A list containing

`id` the pipeline ID that can be used by `deps`

`pipeline` forked pipeline instance

`target_names` copy of `names`

`depend_on` copy of `deps`

`cue` copy of `cue`

`standalone` copy of `standalone`

`PipelineCollection$build_pipelines()`: Build pipelines and visualize

Usage:

```
PipelineCollection$build_pipelines(visualize = TRUE)
```

Arguments:

`visualize` whether to visualize the pipeline; default is true

`PipelineCollection$run()`: Run the collection of pipelines

Usage:

```
PipelineCollection$run(
  error = c("error", "warning", "ignore"),
  .scheduler = c("none", "future", "clustermq"),
  .type = c("callr", "smart", "vanilla"),
  .as_promise = FALSE,
  .async = FALSE,
  rebuild = NA,
  ...
)
```

Arguments:

`error` what to do when error occurs; default is `'error'` throwing errors; other choices are `'warning'` and `'ignore'`

`.scheduler`, `.type`, `.as_promise`, `.async`, `...` passed to `pipeline_run`

`rebuild` whether to re-build the pipeline; default is `NA` (if the pipeline has been built before, then do not rebuild)

`PipelineCollection$get_scheduler()`: Get `scheduler` object

Usage:

```
PipelineCollection$get_scheduler()
```

`PipelineResult` *Class definition for 'RAVE' pipeline results*

Description

Class definition for 'RAVE' pipeline results

Public fields

`progressor` progress bar object, usually generated a progress instance

`promise` a `promise` instance that monitors the pipeline progress

`verbose` whether to print warning messages

`names` names of the pipeline to build

`async_callback` function callback to call in each check loop; only used when the pipeline is running in `async=TRUE` mode

`check_interval` used when `async=TRUE` in `pipeline_run`, interval in seconds to check the progress

Active bindings

`variables` target variables of the pipeline

`variable_descriptions` readable descriptions of the target variables

`valid` logical true or false whether the result instance hasn't been invalidated

`status` result status, possible status are `'initialize'`, `'running'`, `'finished'`, `'canceled'`, and `'errored'`. Note that `'finished'` only means the pipeline process has been finished.

`process` (read-only) process object if the pipeline is running in `'async'` mode, or `NULL`; see `r_bg`.

Methods

Public methods:

- `PipelineResult$validate()`
- `PipelineResult$invalidate()`
- `PipelineResult$get_progress()`
- `PipelineResult$new()`
- `PipelineResult$run()`
- `PipelineResult$await()`
- `PipelineResult$print()`
- `PipelineResult$get_values()`
- `PipelineResult$clone()`

`PipelineResult$validate()`: check if result is valid, raises errors when invalidated

Usage:

```
PipelineResult$validate()
```

`PipelineResult$invalidate()`: invalidate the pipeline result

Usage:

```
PipelineResult$invalidate()
```

`PipelineResult$get_progress()`: get pipeline progress

Usage:

```
PipelineResult$get_progress()
```

`PipelineResult$new()`: constructor (internal)

Usage:

```
PipelineResult$new(path = character(0L), verbose = FALSE)
```

Arguments:

`path` pipeline path

`verbose` whether to print warnings

`PipelineResult$run()`: run pipeline (internal)

Usage:

```
PipelineResult$run(
  expr,
  env = parent.frame(),
  quoted = FALSE,
  async = FALSE,
  process = NULL
)
```

Arguments:

`expr` expression to evaluate

`env` environment of `expr`

`quoted` whether `expr` has been quoted

`async` whether the process runs in other sessions

`process` the process object inherits `process`, will be inferred from `expr` if `process=NULL`, and will raise errors if cannot be found

`PipelineResult$await()`: wait until some targets get finished

Usage:

```
PipelineResult$await(names = NULL, timeout = Inf)
```

Arguments:

`names` target names to wait, default is `NULL`, i.e. to wait for all targets that have been scheduled

`timeout` maximum waiting time in seconds

Returns: `TRUE` if the target is finished, or `FALSE` if timeout is reached

`PipelineResult$print()`: print method

Usage:

```
PipelineResult$print()
```

`PipelineResult$get_values()`: get results

Usage:

```
PipelineResult$get_values(names = NULL, ...)
```

Arguments:

`names` the target names to read

`...` passed to `pipeline_read`

`PipelineResult$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

PipelineTools *Class definition for 'RAVE' pipelines*

Description

Class definition for 'RAVE' pipelines

Super class

RAVESerializable -> PipelineTools

Active bindings

`description` pipeline description

`settings_path` absolute path to the settings file

`extdata_path` absolute path to the user-defined pipeline data folder

`preference_path` directory to the pipeline preference folder

`target_table` table of target names and their descriptions

`result_table` summary of the results, including signatures of data and commands

`pipeline_path` the absolute path of the pipeline

`pipeline_name` the code name of the pipeline

`available_reports` available reports and their configurations

`task` shiny task object, see method `'run_ask_task'`

Methods

Public methods:

- PipelineTools$@marshal()
- PipelineTools$@unmarshal()
- PipelineTools$new()
- PipelineTools$set_settings()
- PipelineTools$get_settings()
- PipelineTools$import_settings()
- PipelineTools$read()
- PipelineTools$run()
- PipelineTools$run_as_task()
- PipelineTools$eval()
- PipelineTools$shared_env()
- PipelineTools$python_module()
- PipelineTools$progress()
- PipelineTools$attach()
- PipelineTools$visualize()

- PipelineTools$target_ancestors()
- PipelineTools$fork()
- PipelineTools$fork_to_subject()
- PipelineTools$with_activated()
- PipelineTools$clean()
- PipelineTools$save_data()
- PipelineTools$load_data()
- PipelineTools$set_preferences()
- PipelineTools$get_preferences()
- PipelineTools$has_preferences()
- PipelineTools$source_document()
- PipelineTools$generate_report()
- PipelineTools$clone()

`PipelineTools$@marshal()`: Create an atomic list that can be serialized

Usage:

```
PipelineTools$@marshal(...)
```

Arguments:

`...` ignored

`PipelineTools$@unmarshal()`: Restore an object from an atomic list

Usage:

```
PipelineTools$@unmarshal(object, ...)
```

Arguments:

`object` a list from `'@marshal'`

`...` ignored

`PipelineTools$new()`: construction function

Usage:

```
PipelineTools$new(
  pipeline_name,
  settings_file = "settings.yaml",
  paths = pipeline_root(),
  temporary = FALSE
)
```

Arguments:

`pipeline_name` name of the pipeline, usually in the pipeline `'DESCRIPTION'` file, or pipeline folder name

`settings_file` the file name of the settings file, where the user inputs are stored

`paths` the paths to find the pipeline, usually the parent folder of the pipeline; default is `pipeline_root()`

`temporary` whether not to save `paths` to current pipeline root registry. Set this to `TRUE` when importing pipelines from subject pipeline folders

`PipelineTools$set_settings()`: set inputs

Usage:

```
PipelineTools$set_settings(..., .list = NULL)
```

Arguments:

`..., .list` named list of inputs; all inputs should be named, otherwise errors will be raised

`PipelineTools$get_settings()`: get current inputs

Usage:

```
PipelineTools$get_settings(key, default = NULL, constraint)
```

Arguments:

`key` the input name; default is missing, i.e., to get all the settings

`default` default value if not found

`constraint` the constraint of the results; if input value is not from `constraint`, then only the first element of `constraint` will be returned.

Returns: The value of the inputs, or a list if `key` is missing

`PipelineTools$import_settings()`: import input settings from file: this can be a `'settings.yaml'` from an exported pipeline, or a report `HTML`

Usage:

```
PipelineTools$import_settings(
  path,
  format = c("auto", "yaml", "html"),
  src_pipeline = NULL,
  settings_names = NULL,
  dry_run = TRUE
)
```

Arguments:

`path` path to the file containing settings information

`format` format of the file; default is to derive from the file extension

`src_pipeline` pipeline or pipeline name from which the settings file was generated. For `HTML` reports, this is automatically derived; default is current pipeline if the information cannot be obtained

`settings_names` names of the input settings to import; default is `NULL` to import all. This option helps avoid changing the underlying data, such as project and subject that has been loaded, and only adjust analysis parameters.

`dry_run` whether to set current pipeline immediately; default is `FALSE`.

Returns: Imported settings as a list

`PipelineTools$read()`: read intermediate variables

Usage:

```
PipelineTools$read(var_names, ifnotfound = NULL, ..., simplify = TRUE)
```

Arguments:

`var_names` the target names, can be obtained via `x$target_table` member; default is missing, i.e., to read all the intermediate variables

`ifnotfound` variable default value if not found

`simplify, ...` other parameters passing to `pipeline_read`

Returns: The values of the targets

`PipelineTools$run()`: run the pipeline

Usage:

```
PipelineTools$run(
  names = NULL,
  async = FALSE,
  as_promise = async,
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  return_values = TRUE,
  debug = FALSE,
  ...
)
```

Arguments:

`names` pipeline variable names to calculate; default is to calculate all the targets

`async` whether to run asynchronous in another process

`as_promise` whether to return a `PipelineResult` instance

`scheduler, type, envir, callr_function, return_values, debug, ...` passed to `pipeline_run` if `as_promise` is true, otherwise these arguments will be passed to `pipeline_run_bare`

Returns: A `PipelineResult` instance if `as_promise` or `async` is true; otherwise a list of values for input `names`

`PipelineTools$run_as_task()`: Run pipeline as `shiny` extended task, requires package **shiny**

Usage:

```
PipelineTools$run_as_task(
  names = NULL,
  with_progress = TRUE,
  check_internals = 0.5,
  ...
)
```

Arguments:

`names` target names to build, see method `'run'`

`with_progress` whether to show progress; default is true

`check_internals,` progress update frequency in seconds; default is 0.5 seconds

`...` arguments passed to `rave_progress`

Returns: `shiny` extended task; see `ExtendedTask`

Examples:

```
# pipeline <- ... (initialize pipeline somewhere)

# runs within shiny
server <- function(input, output, session) {

  pipeline$run_as_task()

  shiny::observe({
    shiny::showNotification(pipeline$task$status())
  })

}
```

`PipelineTools$eval()`: run the pipeline in order; unlike `$run()`, this method does not use the `targets` infrastructure, hence the pipeline results will not be stored, and the order of `names` will be respected.

Usage:

```
PipelineTools$eval(
  names,
  env = parent.frame(),
  shortcut = FALSE,
  clean = TRUE,
  ...
)
```

Arguments:

`names` pipeline variable names to calculate; must be specified

`env` environment to evaluate and store the results

`shortcut` logical or characters; default is `FALSE`, meaning `names` and all the dependencies (if missing from `env`) will be evaluated; set to `TRUE` if only `names` are to be evaluated. When `shortcut` is a character vector, it should be a list of targets (including their ancestors) whose values can be assumed to be up-to-date, and the evaluation of those targets can be skipped.

`clean` whether to evaluate without polluting `env`

`...` passed to `pipeline_eval`

`PipelineTools$shared_env()`: run the pipeline shared library in scripts starting with path `R/shared`

Usage:

```
PipelineTools$shared_env(callr_function = callr::r)
```

Arguments:

`callr_function` either `callr::r` or `NULL`; when `callr::r`, the environment will be loaded in isolated R session and serialized back to the main session to avoid contaminating the main session environment; when `NULL`, the code will be sourced directly in current environment.

Returns: An environment of shared variables

`PipelineTools$python_module()`: get 'Python' module embedded in the pipeline

Usage:

```
PipelineTools$python_module(
  type = c("info", "module", "shared", "exist"),
  must_work = TRUE
)
```

Arguments:

`type` return type, choices are `'info'` (get basic information such as module path, default), `'module'` (load module and return it), `'shared'` (load a shared sub-module from the module, which is shared also in report script), and `'exist'` (returns true or false on whether the module exists or not)

`must_work` whether the module needs to be existed or not. If `TRUE`, the raise errors when the module does not exist; default is `TRUE`, ignored when `type` is `'exist'`.

Returns: See `type`

`PipelineTools$progress()`: get progress of the pipeline

Usage:

```
PipelineTools$progress(method = c("summary", "details"))
```

Arguments:

`method` either `'summary'` or `'details'`

Returns: A table of the progress

`PipelineTools$attach()`: attach pipeline tool to environment (internally used)

Usage:

```
PipelineTools$attach(env)
```

Arguments:

`env` an environment

`PipelineTools$visualize()`: visualize pipeline target dependency graph

Usage:

```
PipelineTools$visualize(
  glimpse = FALSE,
  aspect_ratio = 2,
  node_size = 30,
  label_size = 40,
  ...
)
```

Arguments:

`glimpse` whether to glimpse the graph network or render the state

`aspect_ratio` controls node spacing

`node_size`, `label_size` size of nodes and node labels

`...` passed to `pipeline_visualize`

Returns: a list where the names are target names and values are the corresponding dependence

`PipelineTools$target_ancestors()`: a helper function to get target ancestors

Usage:

```
PipelineTools$target_ancestors(names, skip_names = NULL)
```

Arguments:

`names` targets whose ancestor targets need to be queried

`skip_names` targets that are assumed to be up-to-date, hence will be excluded, notice this exclusion is recursive, that means not only `skip_names` are excluded, but also their ancestors will be excluded from the result.

Returns: ancestor target names (including `names`)

`PipelineTools$fork()`: fork (copy) the current pipeline to a new directory

Usage:

```
PipelineTools$fork(path, policy = "default")
```

Arguments:

`path` path to the new pipeline, a folder will be created there

`policy` fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying `main.html` and `shared` folder

Returns: A new pipeline object based on the path given

`PipelineTools$fork_to_subject()`: fork (copy) the current pipeline to a 'RAVE' subject

Usage:

```
PipelineTools$fork_to_subject(
  subject,
  label = "NA",
  policy = "default",
  delete_old = FALSE,
  sanitize = TRUE
)
```

Arguments:

`subject` subject ID or instance in which pipeline will be saved

`label` pipeline label describing the pipeline

`policy` fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying `main.html` and `shared` folder

`delete_old` whether to delete old pipelines with the same label default is false

`sanitize` whether to sanitize the registry at save. This will remove missing folders and import manually copied pipelines to the registry (only for the pipelines with the same name)

Returns: A new pipeline object based on the path given

`PipelineTools$with_activated()`: run code with pipeline activated, some environment variables and function behaviors might change under such condition (for example, `targets` package functions)

Usage:

```
PipelineTools$with_activated(expr, quoted = FALSE, env = parent.frame())
```

Arguments:

`expr` expression to evaluate

`quoted` whether `expr` is quoted; default is false

`env` environment to run `expr`

`PipelineTools$clean()`: clean all or part of the data store

Usage:

```
PipelineTools$clean(
  destroy = c("all", "cloud", "local", "meta", "process", "preferences", "progress",
    "objects", "scratch", "workspaces"),
  ask = FALSE
)
```

Arguments:

`destroy`, `ask` see `tar_destroy`

`PipelineTools$save_data()`: save data to pipeline data folder

Usage:

```
PipelineTools$save_data(
  data,
  name,
  format = c("json", "yaml", "csv", "fst", "rds"),
  overwrite = FALSE,
  ...
)
```

Arguments:

`data` R object

`name` the name of the data to save, must start with letters

`format` serialize format, choices are `'json'`, `'yaml'`, `'csv'`, `'fst'`, `'rds'`; default is `'json'`. To save arbitrary objects such as functions or environments, use `'rds'`

`overwrite` whether to overwrite existing files; default is no

`...` passed to saver functions

Returns: the saved file path

`PipelineTools$load_data()`: load data from pipeline data folder

Usage:

```
PipelineTools$load_data(
  name,
  error_if_missing = TRUE,
  default_if_missing = NULL,
  format = c("auto", "json", "yaml", "csv", "fst", "rds"),
  ...
)
```

Arguments:

`name` the name of the data

`error_if_missing` whether to raise errors if the name is missing

`default_if_missing` default values to return if the name is missing

`format` the format of the data, default is automatically obtained from the file extension

`...` passed to loader functions

Returns: the data if file is found or a default value

`PipelineTools$set_preferences()`: set persistent preferences from the pipeline. The preferences should not affect how pipeline is working, hence usually stores minor variables such as graphic options. Changing preferences will not invalidate pipeline cache.

Usage:

```
PipelineTools$set_preferences(..., .list = NULL)
```

Arguments:

`..., .list` key-value pairs of initial preference values. The keys must start with 'global' or the module ID, followed by dot and preference type and names. For example `'global.graphics.continuous_palette'` for setting palette colors for continuous heat-map; "global" means the settings should be applied to all 'RAVE' modules. The module-level preference, `'power_explorer.export.default_format'` sets the default format for power-explorer export dialogue.

`name` preference name, must contain only letters, digits, underscore, and hyphen, will be coerced to lower case (case-insensitive)

Returns: A list of key-value pairs

`PipelineTools$get_preferences()`: get persistent preferences from the pipeline.

Usage:

```
PipelineTools$get_preferences(
  keys,
  simplify = TRUE,
  ifnotfound = NULL,
  validator = NULL,
  modes = NULL,
  ...
)
```

Arguments:

`keys` characters to get the preferences

`simplify` whether to simplify the results when length of key is 1; default is true; set to false to always return a list of preferences

`ifnotfound` default value when the key is missing

`validator` `NULL` or function to validate the values; see 'Examples'

`modes` length of zero (no type-constraint), character vector with length of one or `length(keys)` specifying the type of preference values; see `pipeline_get_preferences`

`...` passed to `validator` if `validator` is a function

Returns: A list of the preferences. If `simplify` is true and length if keys is 1, then returns the value of that preference

Examples:

```
library(ravepipeline)
if(interactive() && length(pipeline_list()) > 0) {
  pipeline <- pipeline("power_explorer")

  # set dummy preference
  pipeline$set_preferences("global.example.dummy_preference" = 1:3)

  # get preference
  pipeline$get_preferences("global.example.dummy_preference")

  # get preference with validator to ensure the value length to be 1
  pipeline$get_preferences(
    "global.example.dummy_preference",
    validator = function(value) {
      stopifnot(length(value) == 1)
    },
    ifnotfound = 100
  )

  pipeline$has_preferences("global.example.dummy_preference")
}
```

`PipelineTools$has_preferences()`: whether pipeline has preference keys

Usage:

```
PipelineTools$has_preferences(keys, ...)
```

Arguments:

`keys` characters name of the preferences

`...` passed to internal methods

Returns: logical whether the keys exist

`PipelineTools$source_document()`: obtain the source document

Usage:

```
PipelineTools$source_document()
```

Returns: characters if the source document (`main.Rmd`) is found, otherwise `NULL`

`PipelineTools$generate_report()`: generate pipeline

Usage:

```
PipelineTools$generate_report(
  name,
  subject = NULL,
  output_dir = NULL,
  output_format = "auto",
  clean = FALSE,
  callback = NULL,
  ...
)
```

Arguments:

`name` report name, see field `'available_reports'`

`subject` subject helps determine the `output_dir` and working directories

`output_dir` parent folder where output will be stored

`output_format` output format

`clean` whether to clean the output; default is false

`callback` callback function (if not `NULL`) to run once the report is created; typically used for actions such as zipping the report directory, sending out report via emails. The function must only take one argument, which is the directory where the report resides. The callback function will be evaluated in a separate session so please make sure the function itself is self-contained.

`...` passed to `'rmarkdown'` render function

Returns: A job identification number, see `resolve_job` for querying job details

`PipelineTools$clone()`**:** The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`pipeline`

Examples

```
## ------------------------------------------------
## Method `PipelineTools$run_as_task()`
## ------------------------------------------------



# pipeline <- ... (initialize pipeline somewhere)

# runs within shiny
server <- function(input, output, session) {

  pipeline$run_as_task()

  shiny::observe({
    shiny::showNotification(pipeline$task$status())
  })

}

## ------------------------------------------------
## Method `PipelineTools$get_preferences()`
## ------------------------------------------------
```

```
library(ravepipeline)
if(interactive() && length(pipeline_list()) > 0) {
  pipeline <- pipeline("power_explorer")

  # set dummy preference
  pipeline$set_preferences("global.example.dummy_preference" = 1:3)

  # get preference
  pipeline$get_preferences("global.example.dummy_preference")

  # get preference with validator to ensure the value length to be 1
  pipeline$get_preferences(
    "global.example.dummy_preference",
    validator = function(value) {
      stopifnot(length(value) == 1)
    },
    ifnotfound = 100
  )

  pipeline$has_preferences("global.example.dummy_preference")
}
```

`pipeline_collection` *Combine and execute pipelines*

Description

Experimental, subject to change in the future.

Usage

```
pipeline_collection(root_path = NULL, overwrite = FALSE)
```

Arguments

`root_path`	directory to store pipelines and results
`overwrite`	whether to overwrite if `root_path` exists; default is false, and raises an error when `root_path` exists

Value

A `PipelineCollections` instance

`pipeline_install` *Install 'RAVE' pipelines*

Description

Install 'RAVE' pipelines

Usage

```
pipeline_install_local(
  src,
  to = c("default", "custom", "workdir", "tempdir"),
  upgrade = FALSE,
  force = FALSE,
  set_default = NA,
  ...
)

pipeline_install_github(
  repo,
  to = c("default", "custom", "workdir", "tempdir"),
  upgrade = FALSE,
  force = FALSE,
  set_default = NA,
  ...
)
```

Arguments

`src`	pipeline directory
`to`	installation path; choices are `'default'`, `'custom'`, `'workdir'`, and `'tempdir'`. Please specify pipeline root path via `pipeline_root` when `'custom'` is used.
`upgrade`	whether to upgrade the dependence; default is `FALSE` for stability, however, it is highly recommended to upgrade your dependencies
`force`	whether to force installing the pipelines
`set_default`	whether to set current pipeline module folder as the default, will be automatically set when the pipeline is from the official 'Github' repository.
`...`	other parameters not used
`repo`	'Github' repository in user-repository combination, for example, `'rave-ieeg/rave-pipeline'`

Value

nothing

Examples

```
## Not run:


pipeline_install_github("rave-ieeg/pipelines")


# or download github.com/rave-ieeg/pipelines repository, extract
# to a folder, and call
pipeline_install_local("path/to/pipeline/folder")


## End(Not run)
```

`pipeline_plot_data` *Create plot data from within pipeline make-file*

Description

Tags an R object so that calling `plot` on it outside the pipeline can still dispatch the correct `S3` method, even though that method is only defined inside the pipeline's shared R scripts.

Usage

```
pipeline_plot_data(
  x,
  name = substitute(x),
  strip_oldclasses = TRUE,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  pipeline_name = NULL
)
```

Arguments

`x`	R object to be used as plot data.
`name`	`S3` class name for which `plot.<name>` is implemented in the pipeline's `R/shared*.R` files. Must contain only ASCII letters, digits, dots, or underscores. Defaults to the unevaluated expression passed as `x`.
`strip_oldclasses`	if `TRUE` (default) and `x` already carries a `"ravepipeline_plot_data"` class from a previous call, the stale plot classes are stripped before re-tagging. Set to `FALSE` to preserve the full original class vector.
`pipe_dir`	path to the active pipeline directory. Do not set this when calling from within a pipeline make-file; the default reads the `RAVE_PIPELINE` environment variable which is set automatically during `pipeline_run`.
`pipeline_name`	character string overriding the pipeline name stored in the returned object. When `NULL` (default) the name is inferred from `pipe_dir`.

Value

Object `x` with the class vector `c(name, "ravepipeline_plot_data", <original classes>)` and two extra attributes: `pipeline_name` and `pipeline_plot_class`.

How plotting dispatch works

A RAVE pipeline keeps its plot helpers in files whose names start with `shared` inside the pipeline's `R/` folder (e.g. `R/shared-plots.R`). Those files are sourced automatically every time the pipeline runs, but they are *not* available in an ordinary interactive R session.

`pipeline_plot_data` bridges the two contexts by:

1. Inserting `name` and the sentinel class `"ravepipeline_plot_data"` to the class vector of `x`.
2. Attaching the pipeline name as an attribute so the object can be re-associated with its pipeline later.

When `plot()` is subsequently called:

Inside the pipeline (during `pipeline_run`) The environment variable `RAVE_PIPELINE_ACTIVE` is `"true"`, the shared scripts have already been sourced, and `plot.<name>` is in scope. `plot.ravepipeline_plot_data` simply calls `NextMethod()` so dispatch falls through to `plot.<name>`.

Outside the pipeline (interactive session, report, Shiny app) `plot.ravepipeline_plot_data` locates the pipeline by `pipeline_name`, calls `$shared_env()` to source all `R/shared*.R` files in an isolated environment, and then evaluates `plot(x)` inside that environment, where `plot.<name>` is now available.

Implementing a pipeline plot method

Step 1: define the `S3` method in any file whose name starts with `shared` inside the pipeline's `R/` directory (e.g. `R/shared-plots.R`). The function receives the original object `x` with its user-defined class prepended, so standard R dispatch applies:

```
# R/shared-plots.R  (inside the pipeline source tree)
plot.my_pipeline_result <- function(x, ...) {
  graphics::plot(
    x$time, x$signal,
    type = "l",
    xlab = "Time (s)",
    ylab = "Amplitude",
    main = x$title
  )
}
```

Step 2: wrap the target inside `main.Rmd` (or any pipeline make-file) by calling `pipeline_plot_data` with the same `name` you used for the `S3` method:

```
# main.Rmd  (pipeline make-file target block)
result_plot <- {
  ravepipeline::pipeline_plot_data(
```

```
    list(time = seq(0, 1, by = 0.01),
         signal = sin(2 * pi * 10 * seq(0, 1, by = 0.01)),
         title  = "10 Hz sine wave"),
    name = "my_pipeline_result"
  )
}
```

Step 3: call `plot()` anywhere:

```
# Interactive session or report
p <- pipeline("my_pipeline")
result <- p$read("result_plot")
plot(result)   # sources R/shared-plots.R automatically, then calls
               # plot.my_pipeline_result(result)
```

Examples

```
# 1.  R/shared-plots.R  -- define the S3 method
plot.toy_example <- function(x, ...) {
  graphics::plot(x$data,
                 xlab = "Index", ylab = "Value",
                 main = x$title %||% "")
}

# 2.  main.Rmd target block -- wrap the data
plot_data <- ravepipeline::pipeline_plot_data(
  list(data = 1:10, title = "Toy example"),
  name = "toy_example",
  pipeline_name = "toy_pipeline"
)

# 3.  Interactive session -- just call plot()
plot(plot_data)  # dispatches to plot.toy_example via shared_env
```

`pipeline_settings_get_set` *Get or change pipeline input parameter settings*

Description

Get or change pipeline input parameter settings

Usage

```
pipeline_settings_set(
  ...,
  pipeline_path = Sys.getenv("RAVE_PIPELINE", "."),
  pipeline_settings_path = file.path(pipeline_path, "settings.yaml")
)

pipeline_settings_get(
  key,
  default = NULL,
  constraint = NULL,
  pipeline_path = Sys.getenv("RAVE_PIPELINE", "."),
  pipeline_settings_path = file.path(pipeline_path, "settings.yaml")
)
```

Arguments

`pipeline_path`	the root directory of the pipeline
`pipeline_settings_path`	the settings file of the pipeline, must be a 'yaml' file; default is `'settings.yaml'` in the current pipeline
`key, ...`	the character key(s) to get or set
`default`	the default value is key is missing
`constraint`	the constraint of the resulting value; if not `NULL`, then result must be within the `constraint` values, otherwise the first element of `constraint` will be returned. This is useful to make sure the results stay within given options

Value

`pipeline_settings_set` returns a list of all the settings. `pipeline_settings_get` returns the value of given key.

Examples

```
root_path <- tempfile()
pipeline_root_folder <- file.path(root_path, "modules")

# create pipeline folder
pipeline_path <- pipeline_create_template(
  root_path = pipeline_root_folder, pipeline_name = "raveio_demo",
  overwrite = TRUE, activate = FALSE, template_type = "rmd-bare")

# Set initial user inputs
yaml::write_yaml(
  x = list(
    n = 100,
    pch = 16,
```

```
    col = "steelblue"
  ),
  file = file.path(pipeline_path, "settings.yaml")
)

# build the pipeline for the first time
# this is a one-time setup
pipeline_build(pipeline_path)

# get pipeline settings
pipeline_settings_get(
  key = "n",
  pipeline_path = pipeline_path
)

# get variable with default if missing
pipeline_settings_get(
  key = "missing_variable",
  default = "missing",
  pipeline_path = pipeline_path
)

pipeline_settings_set(
  missing_variable = "A",
  pipeline_path = pipeline_path
)

pipeline_settings_get(
  key = "missing_variable",
  default = "missing",
  pipeline_path = pipeline_path
)


unlink(root_path, recursive = TRUE)
```

`pipeline_translate_settings` *Translate pipeline settings between pipelines*

Description

Translate pipeline settings between pipelines using export and/or import wizard functions defined in each pipeline's `R/import-export-wizard.R` file. `pipeline_export_wizard` and `pipeline_import_wizard` register those wizard functions.

Usage

```
pipeline_translate_settings(src_pipeline, dst_pipeline, settings = NULL)

pipeline_export_wizard(fun, pipeline_name, env = parent.frame())

pipeline_import_wizard(fun, pipeline_name, env = parent.frame())
```

Arguments

src_pipeline	character name or a PipelineTools instance of the source pipeline whose settings are being translated.
dst_pipeline	character name or a PipelineTools instance of the destination pipeline to translate the settings into.
settings	named list of settings to translate. If NULL (default), the current settings of src_pipeline are read automatically.
fun	a function with signature function(settings) that performs the settings translation and returns the modified settings list.
pipeline_name	character; the pipeline name this wizard handles, with context-dependent meaning: • In pipeline_export_wizard: the **destination** pipeline name: the wizard is invoked when exporting the current pipeline's settings *to* pipeline_name (corresponds to dst_pipeline_name in pipeline_translate_settings). • In pipeline_import_wizard: the **source** pipeline name: the wizard is invoked when importing settings *from* pipeline_name into the current pipeline (corresponds to src_pipeline_name in pipeline_translate_settings).
env	environment in which to register the wizard. Defaults to the calling frame, i.e. the sourced import-export-wizard.R environment.

Details

Translation proceeds in up to two passes:

1. **Export pass**: the source pipeline may declare an export wizard keyed by dst_pipeline_name; if present it converts the settings into the destination format.
2. **Import pass**: the destination pipeline may declare an import wizard keyed by the (possibly already-converted) source pipeline name; if present it applies an additional filter. This also handles the case where the destination pipeline defines a self-filter applied after the export pass.

At least one wizard must exist; otherwise an error is raised.

Value

pipeline_translate_settings A named list of translated settings compatible with dst_pipeline_name.

pipeline_export_wizard, pipeline_import_wizard fun, invisibly. Called for the side effect of registering the wizard in env.

Examples

```
## Not run:

# Translate settings from "pipelineA" to "pipelineB"
new_settings <- pipeline_translate_settings(
  src_pipeline = "pipelineA",
  dst_pipeline = "pipelineB"
)

# To achieve this, you would define export and/or import wizards in the
# respective pipelines.

# Option 1: Inside the source pipeline (pipelineA):
# file `R/import-export-wizard.R`, define an export wizard for pipelineB:

pipeline_export_wizard(
  pipeline_name = "pipelineB",
  fun = function(settings) {
    # settings is the current settings list of pipelineA
    settings$frequency_range <- settings$freq_range
    settings$freq_range <- NULL
    settings
  }
)

# Option 2: Inside the destination pipeline (pipelineB):
# file `R/import-export-wizard.R`, define an import wizard for pipelineA:

pipeline_import_wizard(
  pipeline_name = "pipelineA",
  fun = function(settings) {
    # settings is the current settings list of pipelineA
    settings$frequency_range <- settings$freq_range
    settings$freq_range <- NULL
    settings
  }
)


## End(Not run)
```

`rave-pipeline` *Low-level 'RAVE' pipeline functions*

Description

Utility functions for 'RAVE' pipelines, currently designed for internal development use. The infrastructure will be deployed to 'RAVE' in the future to facilitate the "self-expanding" aim. Please check the official 'RAVE' website.

Usage

```
pipeline_root(root_path, temporary = FALSE)

pipeline_list(root_path = pipeline_root())

pipeline_find(name, root_path = pipeline_root())

pipeline_attach(name, root_path = pipeline_root())

pipeline_run(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  names = NULL,
  async = FALSE,
  check_interval = 0.5,
  progress_quiet = !async,
  progress_max = NA,
  progress_title = "Running pipeline",
  return_values = TRUE,
  debug = FALSE,
  ...
)

pipeline_clean(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
 destroy = c("all", "cloud", "local", "meta", "process", "preferences", "progress",
    "objects", "scratch", "workspaces"),
  ask = FALSE
)

pipeline_run_bare(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  scheduler = c("none", "future", "clustermq"),
  type = c("smart", "callr", "vanilla"),
  envir = new.env(parent = globalenv()),
  callr_function = NULL,
  names = NULL,
  return_values = TRUE,
  debug = FALSE,
  ...
)

load_targets(..., env = NULL)

pipeline_target_names(pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))
```

```
pipeline_debug(
  quick = TRUE,
  env = parent.frame(),
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  skip_names
)

pipeline_dep_targets(
  names,
  skip_names = NULL,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", ".")
)

pipeline_eval(
  names,
  env = new.env(parent = parent.frame()),
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  settings_path = file.path(pipe_dir, "settings.yaml"),
  shortcut = FALSE
)

pipeline_visualize(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  glimpse = FALSE,
  targets_only = TRUE,
  shortcut = FALSE,
  zoom_speed = 0.1,
  ...
)

pipeline_progress(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  method = c("summary", "details", "custom"),
  func = targets::tar_progress_summary
)

pipeline_fork(
  src = Sys.getenv("RAVE_PIPELINE", "."),
  dest = tempfile(pattern = "rave_pipeline_"),
  policy = "default",
  activate = FALSE,
  ...
)

pipeline_build(pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))

pipeline_read(
```

```
  var_names,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  branches = NULL,
  ifnotfound = NULL,
  dependencies = c("none", "ancestors_only", "all"),
  simplify = TRUE,
  ...
)

pipeline_vartable(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  targets_only = TRUE,
  complete_only = FALSE,
  ...
)

pipeline_hasname(var_names, pipe_dir = Sys.getenv("RAVE_PIPELINE", "."))

pipeline_watch(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  targets_only = TRUE,
  ...
)

pipeline_create_template(
  root_path,
  pipeline_name,
  overwrite = FALSE,
  activate = TRUE,
  template_type = c("rmd", "r", "rmd-bare", "rmd-scheduler", "rmd-python")
)

pipeline_create_subject_pipeline(
  subject,
  pipeline_name,
  overwrite = FALSE,
  activate = TRUE,
  template_type = c("rmd", "r", "rmd-python")
)

pipeline_description(file)

pipeline_load_extdata(
  name,
  format = c("auto", "json", "yaml", "csv", "fst", "rds"),
  error_if_missing = TRUE,
  default_if_missing = NULL,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
```

```
  ...
)

pipeline_save_extdata(
  data,
  name,
  format = c("json", "yaml", "csv", "fst", "rds"),
  overwrite = FALSE,
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  ...
)

pipeline_shared(
  pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  callr_function = callr::r
)
```

Arguments

`root_path`	the root directory for pipeline templates
`temporary`	whether not to save `paths` to current pipeline root registry. Set this to `TRUE` when importing pipelines from subject pipeline folders
`name`, `pipeline_name`	the pipeline name to create; usually also the folder
`pipe_dir`	where the pipeline directory is; can be set via system environment `Sys.setenv("RAVE_PIPELINE"=...)`
`scheduler`	how to schedule the target jobs: default is `'none'`, which is sequential. If you have multiple heavy-weighted jobs that can be scheduled at the same time, you can choose `'future'` or `'clustermq'`
`type`	how the pipeline should be executed; current choices are `"smart"` to enable 'future' package if possible, `'callr'` to use `r`, or `'vanilla'` to run everything sequentially in the main session.
`callr_function`	function that will be passed to `tar_make`; will be forced to be `NULL` if `type='vanilla'`, or `r` if `type='callr'`
`names`	the names of pipeline targets that are to be executed; default is `NULL`, which runs all targets; use `pipeline_target_names` to check all your available target names.
`async`	whether to run pipeline without blocking the main session
`check_interval`	when running in background (non-blocking mode), how often to check the pipeline
`progress_title`, `progress_max`, `progress_quiet`	control the progress
`return_values`	whether to return pipeline target values; default is true; only works in `pipeline_run_bare` and will be ignored by `pipeline_run`
`debug`	whether to debug the process; default is false
`...`	other parameters, targets, etc.
`destroy`	what part of data repository needs to be cleaned

ask	whether to ask
env, envir	environment to execute the pipeline
quick	whether to skip finished targets to save time
skip_names	hint of target names to fast skip provided they are up-to-date; only used when quick=TRUE. If missing, then skip_names will be automatically determined
settings_path	path to settings file name within subject's pipeline path
shortcut	whether to display shortcut targets
glimpse	whether to hide network status when visualizing the pipelines
targets_only	whether to return the variable table for targets only; default is true
zoom_speed	zoom speed when visualizing the pipeline dependence
method	how the progress should be presented; choices are "summary", "details", "custom". If custom method is chosen, then func will be called
func	function to call when reading customized pipeline progress; default is tar_progress_summary
src, dest	pipeline folder to copy the pipeline script from and to
policy	fork policy defined by module author, see text file 'fork-policy' under the pipeline directory; if missing, then default to avoid copying main.html and shared folder
activate	whether to activate the new pipeline folder from dest; default is false
var_names	variable name to fetch or to check
branches	branch to read from; see tar_read
ifnotfound	default values to return if variable is not found
dependencies	whether to load dependent targets, choices are 'none' (default, only load targets specified by names), 'ancestors_only' (load all but the ancestors targets), and 'all' (both targets and ancestors)
simplify	whether to simplify the output
complete_only	whether only to show completed and up-to-date target variables; default is false
overwrite	whether to overwrite existing pipeline; default is false so users can double-check; if true, then existing pipeline, including the data will be erased
template_type	which template type to create; choices are 'r' or 'rmd'
subject	character indicating valid 'RAVE' subject ID, or a RAVESubject instance
file	path to the 'DESCRIPTION' file under the pipeline folder, or pipeline collection folder that contains the pipeline information, structures, dependencies, etc.
format	format of the extended data, default is 'json', other choices are 'yaml', 'fst', 'csv', 'rds'
error_if_missing, default_if_missing	what to do if the extended data is not found
data	extended data to be saved

Value

`pipeline_root` the root directories of the pipelines

`pipeline_list` the available pipeline names under `pipeline_root`

`pipeline_find` the path to the pipeline

`pipeline_run` a `PipelineResult` instance

`load_targets` a list of targets to build

`pipeline_target_names` a vector of characters indicating the pipeline target names

`pipeline_visualize` a widget visualizing the target dependence structure

`pipeline_progress` a table of building progress

`pipeline_fork` a normalized path of the forked pipeline directory

`pipeline_read` the value of corresponding `var_names`, or a named list if `var_names` has more than one element

`pipeline_vartable` a table of summaries of the variables; can raise errors if pipeline has never been executed

`pipeline_hasname` logical, whether the pipeline has variable built

`pipeline_watch` a basic shiny application to monitor the progress

`pipeline_description` the list of descriptions of the pipeline or pipeline collection

---

rave-pipeline-jobs *Run a function (job) in another session*

Description

Run a function (job) in another session

Usage

```
start_job(
  fun,
  fun_args = list(),
  packages = NULL,
  workdir = NULL,
  method = c("callr", "rs_job", "mirai"),
  name = NULL,
  ensure_init = TRUE,
  digest_key = NULL,
  envvars = NULL,
  log_path = NULL
)

check_job(job_id)
```

```
resolve_job(
  job_id,
  timeout = Inf,
  auto_remove = TRUE,
  must_init = TRUE,
  unresolved = c("warning", "error", "silent"),
  log_maxline = getOption("ravepipeline.log_maxline", 0L)
)

remove_job(job_id)
```

Arguments

fun	function to evaluate
fun_args	list of function arguments
packages	list of packages to load
workdir	working directory; default is temporary path
method	job type; choices are 'rs_job' (only used in 'RStudio' environment), 'mirai' (when package 'mirai' is installed), and 'callr' (default).
name	name of the job
ensure_init	whether to make sure the job has been started; default is true
digest_key	a string that will affect how job ID is generated; used internally
envvars	additional environment variables to set; must be a named list of environment variables
log_path	path to a log file for capturing both standard output and messages (stderr) from the job; default is NULL (no logging). Relative paths are resolved against workdir. The file is created at job preparation time; if creation fails or the path is a directory, logging is silently skipped.
job_id	job identification number
timeout	timeout in seconds before the resolve ends; jobs that are still running are subject to unresolved policy
auto_remove	whether to automatically remove the job if resolved; default it true
must_init	whether the resolve should error out if the job is not initialized: typically meaning the either the resolving occurs too soon (only when ensure_init=FALSE) or the job files are corrupted; default is true
unresolved	what to do if the job is still running after timing-out; default is 'warning' and return NULL, other choices are 'error' or 'silent'
log_maxline	maximum number of log lines to read from the tail of the log file when resolving a job; default is getOption("ravepipeline.log_maxline", 0L). The log lines are attached to the result as attribute "rave_logs" if non-empty.

Value

For start_job, a string of job identification number; check_job returns the job status; resolve_job returns the function result.

Examples

```
## Not run:

# Basic use
job_id <- start_job(function() {
  Sys.sleep(1)
  Sys.getpid()
})

check_job(job_id)

result <- resolve_job(job_id)


# As promise
library(promises)
as.promise(
  start_job(function() {
    Sys.sleep(1)
    Sys.getpid()
  })
) %...>%
  print()


## End(Not run)
```

---

rave-pipeline-preferences

Pipeline preference management (low-level)

Description

Get, set, and check persistent preference values for 'RAVE' pipelines and modules. Preferences are stored in a global on-disk store that survives across R sessions.

Usage

```
pipeline_set_preferences(
  ...,
  .list = NULL,
  .pipe_dir = Sys.getenv("RAVE_PIPELINE", "."),
  .preference_instance = NULL
)

pipeline_get_preferences(
  keys,
```

```
  simplify = TRUE,
  ifnotfound = NULL,
  validator = NULL,
  modes = NULL,
  ...,
  .preference_instance = NULL
)

pipeline_has_preferences(keys, ..., .preference_instance = NULL)
```

Arguments

`..., .list`	for `pipeline_set_preferences`: named values to store, where each name is a preference key; for `pipeline_get_preferences`: additional arguments forwarded to `validator`
`.pipe_dir`	the active pipeline directory used to determine the allowed key prefix; defaults to the `RAVE_PIPELINE` environment variable or the current working directory
`.preference_instance`	pipeline preference instance: this is automatically filled when calling from `pipeline$get_preferences(` When `NULL`, the shared on-disk preference store is used automatically
`keys`	one or more preference key strings following the `[prefix].[type].[key]` naming convention
`simplify`	if `TRUE` (default) and exactly one key is requested, return the value directly instead of a length-one named list
`ifnotfound`	value returned when a requested key is absent or fails validation; default is `NULL`
`validator`	`NULL` or a single-argument function that validates each retrieved value; any extra arguments in `...` are forwarded to it. If the function signals an error, `ifnotfound` is returned for that key instead
`modes`	`NULL`, or a character vector of expected R `mode` strings (e.g. `"numeric"`, `"character"`) recycled to match the length of `keys`. A stored value whose mode does not match is treated as missing and replaced by `ifnotfound`

Details

Preference keys must follow a three-part dot-separated naming convention `[prefix].[type].[key]`:

`prefix` Either `"global"` (shared across all modules) or a specific module ID such as `"power_explorer"`. When calling `pipeline_set_preferences` from within a pipeline, the allowed prefixes are automatically restricted to `"global"` and the current pipeline name.

`type` A category string such as `"graphics"` or `"export"`.

`key` The individual preference item, e.g. `"use_ggplot"` or `"default_format"`.

Valid examples: `"global.graphics.use_ggplot"`, `"power_explorer.export.default_format"`.

Setting a preference value to `NULL` removes the key from the store.

Value

`pipeline_set_preferences` Invisibly returns the named list of values that were passed in.

`pipeline_get_preferences` The preference value(s): a single value when `simplify = TRUE` and one key is requested, otherwise a named list with one element per key.

`pipeline_has_preferences` A logical vector the same length as `keys` indicating which keys currently exist in the preference store.

Examples

```
## Not run:
# Set preferences (keys use [prefix].[type].[key] convention)
pipeline_set_preferences(
  "global.graphics.use_ggplot" = TRUE,
  "global.graphics.cex" = 1.2
)

# Check whether keys exist
pipeline_has_preferences(
  c("global.graphics.use_ggplot", "global.graphics.cex")
)

# Retrieve a single preference (returns the value directly)
pipeline_get_preferences("global.graphics.cex")

# Retrieve multiple preferences as a named list
pipeline_get_preferences(
  keys = c("global.graphics.use_ggplot", "global.graphics.cex"),
  simplify = FALSE
)

# Return a default when the key is absent
pipeline_get_preferences("global.graphics.missing_key", ifnotfound = FALSE)

# Validate the stored mode; fall back to default on mismatch
pipeline_get_preferences(
  "global.graphics.cex",
  modes = "numeric",
  ifnotfound = 1.0
)

# Remove a preference by setting it to NULL
pipeline_set_preferences("global.graphics.cex" = NULL)

## End(Not run)
```

`rave-serialize-refhook`

Serialization reference hook generic functions

Description

Serialization reference hook generic functions

Usage

```
rave_serialize_refhook(object)

rave_serialize_impl(object)

## Default S3 method:
rave_serialize_impl(object)

## S3 method for class 'RAVESerializable'
rave_serialize_impl(object)

## S3 method for class '`rave-brain`'
rave_serialize_impl(object)

rave_unserialize_refhook(x)

rave_unserialize_impl(x)

## Default S3 method:
rave_unserialize_impl(x)

## S3 method for class 'rave_serialized'
rave_unserialize_impl(x)

## S3 method for class 'rave_serialized_r6'
rave_unserialize_impl(x)

## S3 method for class '`rave_serialized_rave-brain`'
rave_unserialize_impl(x)
```

Arguments

`object`	Object to serialize (environment or external pointers)
`x`	raw or string objects that will be passed to `unserialize` function before further reconstruction

Value

`rave_serialize_refhook` returns either serialized objects in string (raw vector converted to char via `rawToChar`), or `NULL` indicating the object undergoing default serialization; `rave_unserialize_refhook` returns the reconstructed object.

Examples

```
# This example requires additional `filearray` package
```

```
# If you are an RAVE user (installed RAVE via rave.wiki)
# then this package was installed

x0 <- array(rnorm(240000), c(200, 300, 4))
x1 <- filearray::as_filearray(x0)
x2 <- RAVEFileArray$new(x1, temporary = TRUE)

r0 <- serialize(x0, NULL, refhook = rave_serialize_refhook)
r1 <- serialize(x1, NULL, refhook = rave_serialize_refhook)
r2 <- serialize(x2, NULL, refhook = rave_serialize_refhook)

# Compare the serialization sizes
c(length(r0), length(r1), length(r2))

y0 <- unserialize(r0, refhook = rave_unserialize_refhook)
y1 <- unserialize(r1, refhook = rave_unserialize_refhook)
y2 <- unserialize(r2, refhook = rave_unserialize_refhook)

all(y0 == x0)
all(y1[] == x0)
all(y2[] == x0)

## Not run:

# 3D Brain, this example needs RAVE installation, not included in
# this package, needs extra installations available at rave.wiki

# 4 MB
brain <- ravecore::rave_brain("demo/DemoSubject")

# 52 KB
rbrain <- serialize(brain, NULL, refhook = rave_serialize_refhook)

brain2 <- unserialize(rbrain, refhook = rave_unserialize_refhook)

brain2$plot()


## End(Not run)
```

`rave-snippet` *'RAVE' code snippets*

Description

Run snippet code

Usage

```
update_local_snippet(force = TRUE)

install_snippet(path)

list_snippets()

load_snippet(topic, local = TRUE)
```

Arguments

force	whether to force updating the snippets; default is true
path	for installing code snippets locally only; can be an R script, a zip file, or a directory
topic	snippet topic
local	whether to use local snippets first before requesting online repository

Value

`load_snippet` returns snippet as a function, others return nothing

Examples

```
# This example script requires running in an interactive session

if(interactive()) {

# ---- Example 1: Install built-in pipeline snippets ------------
update_local_snippet(force = TRUE)

# ---- Example 2: Install customized pipeline snippets ----------
snippets <- file.path(
  "https://github.com/rave-ieeg/rave-gists",
  "archive/refs/heads/main.zip",
  fsep = "/"
)
tempf <- tempfile(fileext = ".zip")
utils::download.file(url = snippets, destfile = tempf)

install_snippet(tempf)

}

# ---- List snippets --------------------------------------------

# list all topics
list_snippets()


# ---- Run snippets as functions --------------------------------
```

```
topic <- "image-burn-contacts-to-t1"

# check whether this example can run
# This snippet requires installing package `raveio`
# which is currently not on CRAN (soon it will)

condition_met <- topic %in% list_snippets() &&
  (system.file(package = "raveio") != "")

if( interactive() && condition_met ) {

  snippet <- load_snippet(topic)

  # Read snippet documentation
  print(snippet)


  results <- snippet(
    subject_code = "DemoSubject",
    project_name = "demo",
    save_path = NA,
    blank_underlay = FALSE
  )

  plot(results)
}
```

`RAVEFileArray` *'R6' wrapper for* `'FileArray'`

Description

Wrapper for better serialization (check 'See also')

Super class

`RAVESerializable` -> `RAVEFileArray`

Public fields

`temporary` whether this file array is to be upon garbage collection; default is false. The file array will be deleted if the temporary flag is set to true and the array mode is `'readwrite'`

Active bindings

`valid` whether the array is valid and ready to read

`@impl` the underlying array object

Methods

Public methods:

- RAVEFileArray$@marshal()
- RAVEFileArray$@unmarshal()
- RAVEFileArray$new()
- RAVEFileArray$clone()

`RAVEFileArray$@marshal()`: Serialization helper, convert the object to a descriptive list

Usage:

```
RAVEFileArray$@marshal(...)
```

Arguments:

`...` ignored

`RAVEFileArray$@unmarshal()`: Serialization helper, convert the object from a descriptive list

Usage:

```
RAVEFileArray$@unmarshal(object, ...)
```

Arguments:

`object` serialized list

`...` ignored

`RAVEFileArray$new()`: Constructor

Usage:

```
RAVEFileArray$new(x, temporary = FALSE)
```

Arguments:

`x` file array or can be converted to `as_filearray`

`temporary` whether this file array is to be deleted once the object is out-of-scope; default is false

`RAVEFileArray$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`RAVESerializable` `rave-serialize-refhook`

raveio-option *Set/Get 'RAVE' option*

Description

Persist settings on local configuration file

Usage

```
raveio_setopt(key, value, .save = TRUE)

raveio_resetopt(all = FALSE)

raveio_getopt(key, default = NA, temp = TRUE)

raveio_confpath(cfile = "settings.yaml")
```

Arguments

key	character, option name
value	character or logical of length 1, option value
.save	whether to save to local drive, internally used to temporary change option. Not recommended to use it directly.
all	whether to reset all non-default keys
default	is key not found, return default value
temp	when saving, whether the key-value pair should be considered temporary, a temporary settings will be ignored when saving; when getting options, setting temp to false will reveal the actual settings.
cfile	file name in configuration path

Details

raveio_setopt stores key-value pair in local path. The values are persistent and shared across multiple sessions. There are some read-only keys such as "session_string". Trying to set those keys will result in error.

The following keys are reserved by 'RAVE':

data_dir Directory path, where processed data are stored; default is at home directory, folder ~/rave_data/data_dir

raw_data_dir Directory path, where raw data files are stored, mainly the original signal files and imaging files; default is at home directory, folder ~/rave_data/raw_dir

max_worker Maximum number of CPU cores to use; default is one less than the total number of CPU cores

mni_template_root Directory path, where 'MNI' templates are stored

`raveio_getopt` returns value corresponding to the keys. If key is missing, the whole option will be returned.

If set `all=TRUE`, `raveio_resetopt` resets all keys including non-standard ones. However `"session_string"` will never reset.

Value

`raveio_setopt` returns modified `value`; `raveio_resetopt` returns current settings as a list; `raveio_confpath` returns absolute path for the settings file; `raveio_getopt` returns the settings value to the given key, or `default` if not found.

Side-Effects

The following options will alter other packages and might cause changes in behaviors:

`'disable_fork_clusters'` This option will change the `options` `'dipsaus.no.fork'` and `'dipsaus.cluster.backup'`, which handles the parallel computing

`'threeBrain_template_subject'` This option will set and persist option `'threeBrain.template_subject'`, which changes the default group-level template brain.

See Also

`R_user_dir`

Examples

```
# get one RAVE option
ncore <- raveio_getopt("max_worker")
print(ncore)

# get all options
raveio_getopt()

# set option
raveio_setopt("disable_fork_clusters", FALSE)
```

---

ravepipeline-constants

Constant variables used in 'RAVE' pipeline

Description

Regular expression `PIPELINE_FORK_PATTERN` defines the file matching rules when forking a pipeline; see `pipeline_fork` for details.

Usage

```
PIPELINE_FORK_PATTERN
```

ravepipeline_finalize_installation
Download 'RAVE' built-in pipelines and code snippets

Description

The official built-in pipeline repository is located at https://github.com/rave-ieeg/rave-pipelines; The code snippet repository is located at https://github.com/rave-ieeg/rave-gists.

Usage

```
ravepipeline_finalize_installation(
  upgrade = c("ask", "always", "never", "config-only", "data-only"),
  async = FALSE,
  ...
)
```

Arguments

upgrade	rules to upgrade dependencies; default is to ask if needed
async	whether to run in the background; ignore for now
...	ignored; reserved for external calls.

Value

A list built-in pipelines will be installed, the function itself returns nothing.

Examples

```
## Not run:

# This function requires connection to the Github, and must run
# under interactive session since an user prompt will be displayed

ravepipeline_finalize_installation()


## End(Not run)
```

`RAVESerializable` *Abstract class for 'RAVE' serialization*

Description

For package inheritance only; do not instantiate the class directly.

Methods

Public methods:

- `RAVESerializable$new()`
- `RAVESerializable$@marshal()`
- `RAVESerializable$@unmarshal()`
- `RAVESerializable$@compare()`
- `RAVESerializable$clone()`

`RAVESerializable$new()`: Abstract constructor

Usage:

```
RAVESerializable$new(...)
```

Arguments:

`...` ignored

`RAVESerializable$@marshal()`: Create an atomic list that can be serialized

Usage:

```
RAVESerializable$@marshal(...)
```

Arguments:

`...` ignored

`RAVESerializable$@unmarshal()`: Restore an object from an atomic list

Usage:

```
RAVESerializable$@unmarshal(object, ...)
```

Arguments:

`object` a list from `'@marshal'`

`...` ignored

`RAVESerializable$@compare()`: How two object can be compared to each other

Usage:

```
RAVESerializable$@compare(other)
```

Arguments:

`other` another object to compare with self

`RAVESerializable$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

RAVEFileArray rave-serialize-refhook

`rave_progress` *'RAVE' progress*

Description

Automatically displays 'shiny' progress when shiny is present, or text messages to track progress

Usage

```
rave_progress(
  title,
  max = 1,
  ...,
  quiet = FALSE,
  session = get_shiny_session(),
  shiny_auto_close = FALSE,
  log = NULL
)
```

Arguments

`title`	progress title
`max`	maximum steps
`...`	passed to shiny progress
`quiet`	whether to suppress the progress
`session`	shiny session
`shiny_auto_close`	whether to automatically close the progress bar when the parent function is closed
`log`	alternative log function if not default (`message`)

Value

A list of functions to control the progress bar

Examples

```
# Naive example
progress <- rave_progress(title = "progress", max = 10)
progress$inc("job 1")
progress$inc("job 2")
progress$close()
```

```
# Within function
slow_sum <- function(n = 11) {
  p <- rave_progress(title = "progress", max = n,
                     shiny_auto_close = TRUE)
  s <- 0
  for( i in seq(1, n) ) {
    Sys.sleep(0.1)
    p$inc(sprintf("adding %d", i))
    s <- s + i
  }
  invisible(s)
}

slow_sum()
```

`read-write-yaml` *Read write* `'YAML'` *format*

Description

supports reading data into a map object, and write the map to files with names sorted for consistency

Usage

```
load_yaml(file, ..., map = NULL)

save_yaml(x, file, ..., sorted = FALSE)
```

Arguments

`file`	file to read from or write to
`...`	passed to `as.list`
`map`	a `fastmap` object; can be generated by `fastmap` or `dipsaus` package; default is to create a new map internally
`x`	list or map to write
`sorted`	whether to sort the list by name; default is false

Value

A map object

Examples

```
tfile <- tempfile(fileext = ".yml")

save_yaml(list(b = 2, a = 1), tfile, sorted = TRUE)

cat(readLines(tfile), sep = "\n")

load_yaml(tfile)

unlink(tfile)
```

`with_rave_parallel` *Internal parallel functions*

Description

Experimental parallel functions, intended for internal use now. The goal is to allow 'RAVE' functions to gain the potential benefit from parallel computing, but allow users to control whether to do it.

Usage

```
with_rave_parallel(expr, .workers = 0)

lapply_jobs(
  x,
  fun,
  ...,
  .globals = list(),
  .workers = 0,
  .always = FALSE,
  callback = NULL
)
```

Arguments

`expr`	expression to evaluate with parallel workers
`.workers`	number of workers: note the actual numbers may differ, depending on the options and number of input elements
`x`	a list, vector, array of R objects
`fun`	function to apply to each element of `x`
`...`	additional arguments to be passed to `fun`
`.globals`	global variables to be serialized

`.always`	whether always use workers, only considered when number of workers is one; default is false, then run jobs in the main process when only one worker is required
`callback`	callback function, input is each element of `x` and should return a string, for progress bar
`workers`	number of workers

Details

By default, `lapply_jobs` is almost identical to `lapply`. It only runs in parallel when running inside of `with_rave_parallel`.

The hard max-limit number of workers are determined by the 'RAVE' option `raveio_getopt('max_worker')`. Users can lower this number for memory-intensive tasks manually, via argument `.workers`. The actual number of workers might be less than the requested ones, this is often a result of sort input `x`. If the number of input iterations has fewer than the max worker size, then the number of workers automatically shrinks to the length of input list. All workers will be a child process running separate from the main session, except for when only one worker is needed and `.always=FALSE`: the only task will be executed in the main session.

Each worker session will run a completely isolated new process. There is a ramp-up serialization that is needed for global objects (objects that are defined elsewhere or outside of the function). Please make sure the global objects are specified explicitly in `.globals`, a named list. Unlike `future` package, users must specify the global objects.

The global objects might be large to serialize. Please optimize the code to avoid serializing big objects, especially environments or functions. All objects inheriting `RAVESerializable` class with `@marshal` and `@unmarshal` methods implemented correctly will be serialized with reference hook `rave_serialize_refhook`, making them extremely efficient.

Examples

```
# Run without `with_rave_parallel`
res <- lapply_jobs(1:5, function(x, ...) {
  c(child = Sys.getpid(), ...)
}, main = Sys.getpid())

simplify2array(res)

# Comparison
f <- function(n = 5, workers = 0) {
  system.time({
    ravepipeline::lapply_jobs(seq_len(n), function(x, ...) {
      Sys.sleep(1)
      c(child = Sys.getpid(), ...)
    }, main = Sys.getpid(), .workers = workers, callback = I)
  })
}

## Not run:
```

```
# Without parallel
f()
#>    user  system elapsed
#>   0.022   0.019   5.010


# with parallel
with_rave_parallel({
  f()
})
#>    user  system elapsed
#>   0.729   0.190   1.460


## End(Not run)
```

Index