Package 'rixpress'

May 19, 2026

Title Build Reproducible Analytical Pipelines with 'Nix'

Version 0.12.3

Description Streamlines the creation of reproducible analytical pipelines using 'default.nix' expressions generated via the 'rix' package for reproducibility. Define derivations in 'R', 'Python' or 'Julia', chain them into a composition of pure functions and build the resulting pipeline using 'Nix' as the underlying end-to-end build tool. Functions to plot the pipeline as a directed acyclic graph are included, as well as functions to load and inspect intermediary results for interactive analysis. User experience heavily inspired by the 'targets' package.

License GPL (>= 3)

Encoding UTF-8

URL https://github.com/ropensci/rixpress/, https://docs.ropensci.org/rixpress/

BugReports https://github.com/ropensci/rixpress/issues/

Depends R (>= 4.1.0)

Imports cli, igraph, jsonlite, processx

RoxygenNote 7.3.3

SystemRequirements Nix

Language en-GB

Suggests chronicler, dplyr, ggdag, ggplot2, knitr, maybe, mockery, reticulate, rix, rmarkdown, testthat (>= 3.0.0), usethis, visNetwork

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Bruno Rodrigues [aut, cre] (ORCID: <https://orcid.org/0000-0002-3211-3689>), William Michael Landau [rev] (William reviewed the package (v. 0.2.0) for rOpenSci, see <https://github.com/ropensci/software-review/issues/706>),

Anthony Martinez [rev] (ORCID: <https://orcid.org/0000-0002-4295-0261>,
Anthony reviewed the package (v. 0.2.0) for rOpenSci, see
<https://github.com/ropensci/software-review/issues/625>)

Maintainer Bruno Rodrigues <bruno@brodrigues.co>

Repository CRAN

Date/Publication 2026-05-19 07:30:07 UTC

Contents

add_import *Add an Import Statement to Python Files in the _rixpress Folder Matching a Nix Environment Name*

Description

This function appends a specified import statement to the end of each Python file within the `_rixpress` folder and its subdirectories, but only for files whose base name matches the provided Nix environment.

Usage

```
add_import(import_statement, nix_env, project_path = ".")
```

Arguments

`import_statement`	A character string representing the import statement to be added. For example, `"import numpy as np"`.
`nix_env`	A character string naming the Nix environment file (e.g. `"default.nix"` or `"py-env.nix"` or similar).
`project_path`	Path to root of project, typically ".".

Value

No return value; the function performs in-place modifications of the files.

See Also

Other python import: `adjust_import()`

Examples

```
## Not run:
add_import("import numpy as np", "default.nix")
add_import("import numpy as np", "default.nix", project_path = "path/to/project")

## End(Not run)
```

adjust_import *Adjust Python Import Statements*

Description

When calling rxp_populate(), a file containing Python import statements is automatically generated inside the _rixpress folder. For example, if the numpy package is needed, the file will include a line like "import numpy". However, Python programmers often write "import numpy as np" instead.

Usage

```
adjust_import(old_import, new_import, project_path = ".")
```

Arguments

old_import	A character string representing the import statement to be replaced. For example, "import pillow".
new_import	A character string representing the new import statement to replace with. For example, "from PIL import Image".
project_path	Path to root of project, typically ".".

Details

In some cases, the correct import statement is entirely different. For example, for the pillow package, the generated file will contain "import pillow", which is incorrect—Python code should import from the PIL namespace instead, e.g., "from PIL import Image".

Because these adjustments cannot be automated reliably, the adjust_import() function allows you to search and replace import statements programmatically. It reads each file in the _rixpress folder, performs the replacement, and writes the modified content back to the file.

Value

No return value; the function performs in-place modifications of the files.

See Also

Other python import: add_import()

Examples

```
## Not run:
adjust_import("import pillow", "from PIL import Image")
adjust_import("import pillow", "from PIL import Image", project_path = "path/to/project")

## End(Not run)
```

print.rxp_derivation *Print Method for Derivation Objects*

Description

Print Method for Derivation Objects

Usage

```
## S3 method for class 'rxp_derivation'
print(x, ...)
```

Arguments

x	An object of class "rxp_derivation"
...	Additional arguments passed to print methods

Value

Nothing, prints a summary of the derivation object to the console.

See Also

Other utilities: rxp_check_chronicles(), rxp_copy(), rxp_gc(), rxp_init(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_read(), rxp_trace()

Examples

```
## Not run:
# d0 is a previously defined derivation
  print(d0)

## End(Not run)
```

print.rxp_pipeline *Print Method for rxp_pipeline Objects*

Description

Print Method for rxp_pipeline Objects

Usage

```
## S3 method for class 'rxp_pipeline'
print(x, ...)
```

Arguments

x	An object of class "rxp_pipeline"
...	Additional arguments passed to print methods

Value

Nothing, prints a summary of the pipeline object to the console.

---

rxp_check_chronicles *Check Pipeline Outputs for Chronicle Status*

Description

Scans all derivation outputs for chronicle objects and reports their status: success (Just, no warnings), warning (Just with warnings), or nothing (failed computation). Only active when chronicler is installed.

Usage

```
rxp_check_chronicles(project_path = ".", which_log = NULL)
```

Arguments

project_path	Character, defaults to ".". Path to the root directory of the project.
which_log	Character, defaults to NULL. If NULL the most recent build log is used. If a string is provided, it's used as a regular expression to match against available log files.

Details

This function is useful when using the {chronicler} package in your rixpress pipeline. Because chronicler catches errors and warnings, returning Nothing values instead of failing, Nix builds will always succeed. This function helps identify derivations that contain failed computations.

The function displays one of three symbols for each chronicle:

- checkmark Success: Just value, no warnings or errors
- warning sign Warning: Just value, but warnings were captured
- X mark Nothing: Failed computation, errors captured

Value

A data frame with columns: derivation, chronicle_state, num_operations, num_failed, failed_functions, messages. Returns NULL invisibly if chronicler is not installed or no chronicle objects are found.

See Also

Other utilities: print.rxp_derivation(), rxp_copy(), rxp_gc(), rxp_init(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_read(), rxp_trace()

Examples

```
## Not run:
  # After building a pipeline with chronicler functions
  rxp_check_chronicles()

  # Check a specific build log
  rxp_check_chronicles(which_log = "20250131")

## End(Not run)
```

---

rxp_copy *Copy Derivations From the Nix Store to Current Working Directory*

Description

When Nix builds a derivation, its output is saved in the Nix store located under `/nix/store/`. Even though you can import the derivations into the current R session using `rxp_read()` or `rxp_load()`, it can be useful to copy the outputs to the current working directory. This is especially useful for Quarto documents, where there can be more than one input, as is the case for `html` output.

Usage

```
rxp_copy(derivation_name = NULL, dir_mode = "0755", file_mode = "0644")
```

Arguments

`derivation_name`	The name of the derivation to copy. If empty, then all the derivations are copied.
`dir_mode`	Character, default "0755". POSIX permission mode to apply to directories under the copied output (including the top-level output directory).
`file_mode`	Character, default "0644". POSIX permission mode to apply to files under the copied output.

Value

Nothing, the contents of the Nix store are copied to the current working directory.

See Also

Other utilities: print.rxp_derivation(), rxp_check_chronicles(), rxp_gc(), rxp_init(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_read(), rxp_trace()

Examples

```
## Not run:
  # Copy all derivations to the current working directory
  rxp_copy()

  # Copy a specific derivation
  rxp_copy("mtcars")

  # Copy with custom permissions (e.g., make scripts executable)
  rxp_copy("my_deriv", dir_mode = "0755", file_mode = "0644")

  # Copy a Quarto document output with multiple files
  rxp_copy("my_quarto_doc")

## End(Not run)
```

`rxp_dag_for_ci` *Export DAG of Pipeline and Prepare It for Rendering on CI*

Description

This function generates a DOT file representation of the pipeline DAG, suitable for visualization, potentially on CI platforms. It is called by `rxp_ga()`.

Usage

```
rxp_dag_for_ci(
  nodes_and_edges = get_nodes_edges(),
  output_file = "_rixpress/dag.dot"
)
```

Arguments

`nodes_and_edges`	List, output of `get_nodes_edges()`. Defaults to calling `get_nodes_edges()`.
`output_file`	Character, the path where the DOT file should be saved. Defaults to `"_rixpress/dag.dot"`. The directory will be created if it doesn't exist.

Value

Nothing, writes the DOT file to the specified `output_file`.

See Also

Other ci utilities: `rxp_ga()`, `rxp_write_dag()`

Examples

```
## Not run:
  # Generate the default _rixpress/dag.dot
  rxp_dag_for_ci()


## End(Not run)
```

`rxp_export_artifacts` *Export Nix Store Paths to an Archive*

Description

Creates a single archive file containing the specified Nix store paths and their dependencies. This archive can be transferred to another machine and imported into its Nix store.

Usage

```
rxp_export_artifacts(
  archive_file = "_rixpress/pipeline_outputs.nar",
  which_log = NULL,
  project_path = "."
)
```

Arguments

`archive_file`	Character, path to the archive, defaults to "_rixpress/pipeline-outputs.nar"
`which_log`	Character or NULL, regex pattern to match a specific log file. If NULL (default), the most recent log file will be used.
`project_path`	Character, defaults to ".". Path to the root directory of the project.

Value

Nothing, creates an archive file at the specified location.

See Also

Other archive caching functions: `rxp_import_artifacts()`

Examples

```
## Not run:
  # Export the most recent build to the default location
  rxp_export_artifacts()

  # Export a specific build to a custom location
  rxp_export_artifacts(
    archive_file = "my_archive.nar",
```

```
    which_log = "20250510"
  )

## End(Not run)
```

`rxp_ga` *Run a Pipeline on GitHub Actions*

Description

Run a Pipeline on GitHub Actions

Usage

```
rxp_ga()
```

Details

This function puts a `.yaml` file inside the `.github/workflows/` folder on the root of your project. This workflow file expects both scripts generated by `rxp_init()`, `gen-env.R` and `gen-pipeline.R` to be present. If that's not the case, edit the `.yaml` file accordingly. Build artifacts are archived and restored automatically between runs. Make sure to give read and write permissions to the GitHub Actions bot.

Value

Nothing, copies file to a directory.

See Also

Other ci utilities: `rxp_dag_for_ci()`, `rxp_write_dag()`

Examples

```
## Not run:
  rxp_ga()

## End(Not run)
```

rxp_gc *Garbage Collect Rixpress Build Artifacts and Logs*

Description

This function performs garbage collection on Nix store paths and build log files generated by rixpress. It can operate in two modes: full garbage collection (when `keep_since = NULL`) or targeted deletion based on log file age.

Usage

```
rxp_gc(
  keep_since = NULL,
  project_path = ".",
  dry_run = FALSE,
  timeout_sec = 300,
  verbose = FALSE,
  ask = TRUE
)
```

Arguments

`keep_since`	Date or character string (YYYY-MM-DD format). If provided, only build logs older than this date will be targeted for deletion, along with their associated Nix store paths. If `NULL`, performs a full Nix garbage collection. Default is `NULL`.
`project_path`	Character string specifying the path to the project directory containing the `_rixpress` folder with build logs. Default is `"."` (current directory).
`dry_run`	Logical. If `TRUE`, shows what would be deleted without actually performing any deletions. Useful for previewing the cleanup operation. Default is `FALSE`.
`timeout_sec`	Numeric. Timeout in seconds for individual Nix commands. Also used for concurrency lock expiration. Default is 300 seconds.
`verbose`	Logical. If `TRUE`, provides detailed output including full paths, command outputs, and diagnostic information about references preventing deletion. Default is `FALSE`.
`ask`	Logical. If `TRUE`, ask for user confirmation before performing deleting artifacts. Default is `TRUE`.

Details

The function operates in two modes:

Full Garbage Collection Mode (`keep_since = NULL`):

- Runs `nix-store --gc` to delete all unreferenced store paths
- Does not delete any build log files
- Suitable for complete cleanup of unused Nix store paths

Targeted Deletion Mode (`keep_since` specified):

- Identifies build logs older than the specified date
- Extracts store paths from old logs using `rxp_inspect()`
- Protects recent store paths by creating temporary GC roots
- Attempts to delete old store paths individually using `nix-store --delete`
- Deletes the corresponding build log `.json` files from `_rixpress/`
- Handles referenced paths gracefully (paths that cannot be deleted due to dependencies)

Concurrency Safety: The function uses a lock file mechanism to prevent multiple instances from running simultaneously, which could interfere with each other's GC root management.

Reference Handling: Some store paths may not be deletable because they are still referenced by:

- User or system profile generations
- Active Nix shell environments
- Result symlinks in project directories
- Other store paths that depend on them

These paths are reported but not considered errors.

Value

Invisibly returns a list with cleanup summary information:

- `kept`: Vector of build log filenames that were kept
- `deleted`: Vector of build log filenames targeted for deletion
- `protected`: Number of store paths protected via GC roots (date-based mode)
- `deleted_count`: Number of store paths successfully deleted
- `failed_count`: Number of store paths that failed to delete
- `referenced_count`: Number of store paths skipped due to references
- `log_files_deleted`: Number of build log files successfully deleted
- `log_files_failed`: Number of build log files that failed to delete
- `dry_run_details`: List of detailed information when dry_run = TRUE

See Also

`rxp_list_logs`, `rxp_inspect`

Other utilities: `print.rxp_derivation()`, `rxp_check_chronicles()`, `rxp_copy()`, `rxp_init()`, `rxp_inspect()`, `rxp_list_logs()`, `rxp_load()`, `rxp_read()`, `rxp_trace()`

Examples

```
## Not run:
# Preview what would be deleted (dry run)
rxp_gc(keep_since = "2025-08-01", dry_run = TRUE, verbose = TRUE)

# Delete artifacts from builds older than August 1st, 2025
rxp_gc(keep_since = "2025-08-01")

# Full garbage collection of all unreferenced store paths
rxp_gc()

# Clean up artifacts older than 30 days ago
rxp_gc(keep_since = Sys.Date() - 30)

## End(Not run)
```

---

rxp_ggdag *Create a Directed Acyclic Graph (DAG) Representing the Pipeline Using* {ggplot2}

Description

Uses {ggdag} to generate the plot. {ggdag} is a soft dependency of {rixpress} so you need to install it to use this function. When derivations are organized into pipelines using `rxp_pipeline()`, nodes use a dual-encoding approach: the interior fill shows the derivation type (R, Python, etc.) while a thick border shows the pipeline group colour.

Usage

```
rxp_ggdag(
  nodes_and_edges = get_nodes_edges(),
  color_by = c("pipeline", "type"),
  colour_by = NULL
)
```

Arguments

`nodes_and_edges`	List, output of `get_nodes_edges()`.
`color_by`	Character, either "pipeline" (default) or "type". When "pipeline", nodes show type as fill colour and pipeline as border. When "type", nodes are coloured entirely by derivation type (rxp_r, rxp_py, etc.).
`colour_by`	Character, alias for `color_by`.

Value

A {ggplot2} object.

See Also

Other visualisation functions: `rxp_visnetwork()`

Examples

```
## Not run:
  rxp_ggdag()  # Dual encoding: fill = type, border = pipeline
  rxp_ggdag(colour_by = "type")  # Color entirely by derivation type

## End(Not run)
```

---

rxp_import_artifacts *Import Nix Store Paths from an Archive*

Description

Imports the store paths contained in an archive file into the local Nix store. Useful for transferring built outputs between machines.

Usage

```
rxp_import_artifacts(archive_file = "_rixpress/pipeline_outputs.nar")
```

Arguments

`archive_file`	Character, path to the archive, defaults to "_rixpress/pipeline-outputs.nar"

Value

Nothing, imports the archive contents into the local Nix store.

See Also

Other archive caching functions: `rxp_export_artifacts()`

Examples

```
## Not run:
  # Import from the default archive location
  rxp_import_artifacts()

  # Import from a custom archive file
  rxp_import_artifacts("path/to/my_archive.nar")

## End(Not run)
```

rxp_init *Initialize Rixpress Project*

Description

Generates gen-env.R and gen-pipeline.R scripts in the specified project directory, after asking the user for confirmation. If the user declines, no changes are made.

Usage

```
rxp_init(project_path = ".", skip_prompt = FALSE)
```

Arguments

project_path	Character string specifying the project's path.
skip_prompt	Logical. If TRUE, skips all confirmation prompts and proceeds with initialization, useful on continuous integration. Defaults to FALSE.

Details

Creates (overwriting if they already exist):

- gen-env.R: Script to define an execution environment with {rix}.
- gen-pipeline.R: Defines a data pipeline with {rixpress}.

Value

Logical. Returns TRUE if initialization was successful, FALSE if the operation was cancelled by the user.

See Also

Other utilities: print.rxp_derivation(), rxp_check_chronicles(), rxp_copy(), rxp_gc(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_read(), rxp_trace()

Examples

```
# Default usage (will prompt before any action)
## Not run:
  rxp_init()

## End(Not run)
```

`rxp_inspect` *Inspect the Build Result of a Pipeline*

Description

Returns a data frame with four columns: - derivation: the name of the derivation - build_success: whether the build was successful or not - path: the path of this derivation in the Nix store - output: the output, if this derivation was built successfully. Empty outputs mean that this derivation was not built successfully. Several outputs for a single derivation are possible. In the `derivation` column you will find an object called `all-derivations`. This object is generated automatically for internal purposes, and you can safely ignore it.

Usage

```
rxp_inspect(project_path = ".", which_log = NULL)
```

Arguments

`project_path`	Character, defaults to ".". Path to the root directory of the project.
`which_log`	Character, defaults to NULL. If NULL the most recent build log is used. If a string is provided, it's used as a regular expression to match against available log files.

Value

A data frame with derivation names, if their build was successful, their paths in the /nix/store, and their build outputs.

See Also

Other utilities: `print.rxp_derivation()`, `rxp_check_chronicles()`, `rxp_copy()`, `rxp_gc()`, `rxp_init()`, `rxp_list_logs()`, `rxp_load()`, `rxp_read()`, `rxp_trace()`

Examples

```
## Not run:
  # Inspect the most recent build
  build_results <- rxp_inspect()

  # Inspect a specific build log
  build_results <- rxp_inspect(which_log = "20250510")

  # Check which derivations failed
  failed <- subset(build_results, !build_success)

## End(Not run)
```

rxp_jl *Create a Nix Expression Running a Julia Function*

Description

Create a Nix Expression Running a Julia Function

Usage

```
rxp_jl(
  name,
  expr,
  additional_files = "",
  user_functions = "",
  nix_env = "default.nix",
  encoder = NULL,
  decoder = NULL,
  env_var = NULL,
  noop_build = FALSE
)
```

Arguments

name	Symbol, name of the derivation.
expr	Character, Julia code to generate the expression. Ideally it should be a call to a pure function. Multi-line expressions are not supported.
additional_files	Character vector, additional files to include during the build process. For example, if a function expects a certain file to be available, this is where you should include it.
user_functions	Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.
nix_env	Character, path to the Nix environment file, default is "default.nix".
encoder	Character, defaults to NULL. The name of the Julia function used to serialize the object. It must accept two arguments: the object to serialize (first), and the target file path (second). If NULL, the default behaviour uses the built-in `Serialization.serialize` API. Define any custom serializer in `functions.jl`. See `vignette("encoding-decoding")` for more details.
decoder	Character or named vector/list, defaults to NULL. Can be: • A single string for the Julia function to unserialize all upstream objects • A named vector/list where names are upstream dependency names and values are their specific unserialize functions If NULL, the default is `Serialization.deserialize`. See `vignette("encoding-decoding")` for more details.

env_var	Character vector, defaults to NULL. A named vector of environment variables to set before running the Julia script, e.g., c("JULIA_DEPOT_PATH" = "/path/to/depot"). Each entry will be added as an export statement in the build phase.
noop_build	Logical, defaults to FALSE. If TRUE, the derivation produces a no-op build (a stub output with no actual build steps). Any downstream derivations depending on a no-op build will themselves also become no-op builds.

Details

At a basic level, rxp_jl(filtered_data, "filter(df, :col .> 10)") is equivalent to filtered_data = filter(df, :co in Julia. rxp_jl() generates the required Nix boilerplate to output a so-called "derivation" in Nix jargon. A Nix derivation is a recipe that defines how to create an output (in this case filtered_data) including its dependencies, build steps, and output paths.

Value

An object of class derivation which inherits from lists.

See Also

Other derivations: rxp_jl_file(), rxp_py(), rxp_py_file(), rxp_qmd(), rxp_r(), rxp_r_file(), rxp_rmd()

Examples

```
## Not run:
# Basic usage, no custom serializer
rxp_jl(
  name = filtered_df,
  expr = "filter(df, :col .> 10)"
)

# Skip building this derivation
rxp_jl(
  name = model_result,
  expr = "train_model(data)",
  noop_build = TRUE
)

# Custom serialization: assume `save_my_obj(obj, path)` is defined in functions.jl
rxp_jl(
  name = model_output,
  expr = "train_model(data)",
  encoder = "save_my_obj",
  user_functions = "functions.jl"
)

## End(Not run)
```

`rxp_jl_file` *Creates a Nix Expression That Reads In a File (or Folder of Data) Using Julia*

Description

Creates a Nix Expression That Reads In a File (or Folder of Data) Using Julia

Usage

```
rxp_jl_file(...)
```

Arguments

`...` Arguments passed on to `rxp_file`

`name` Symbol, the name of the derivation.

`path` Character, the file path to include (e.g., "data/mtcars.shp") or a folder path (e.g., "data"). See details.

`read_function` Function, an R function to read the data, taking one argument (the path). This can be a user-defined function that is made available using `user_functions`. See details.

`user_functions` Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.

`nix_env` Character, path to the Nix environment file, default is "default.nix".

`env_var` List, defaults to NULL. A named list of environment variables to set before running the R script, e.g., c(VAR = "hello"). Each entry will be added as an export statement in the build phase.

`encoder` Function/character, defaults to NULL. A language-specific serializer to write the loaded object to disk.

- R: function/symbol/character (e.g., `qs::qsave`) taking `(object, path)`. Defaults to `saveRDS`.
- Python: character name of a function taking `(object, path)`. Defaults to using `pickle.dump`.
- Julia: character name of a function taking `(object, path)`. Defaults to using `Serialization.serialize`.

Details

The basic usage is to provide a path to a file, and the function to read it. For example: `rxp_r_file(mtcars, path = "data/mtcars.csv", read_function = read.csv)`. It is also possible instead to point to a folder that contains many files that should all be read at once, for example: `rxp_r_file(many_csvs, path = "data", read_`
See the `vignette("importing-data")` vignette for more detailed examples.

Value

An object of class `rxp_derivation`.

See Also

Other derivations: `rxp_jl()`, `rxp_py()`, `rxp_py_file()`, `rxp_qmd()`, `rxp_r()`, `rxp_r_file()`, `rxp_rmd()`

`rxp_list_logs` *List All Available Build Logs*

Description

Returns a data frame with information about all build logs in the project's _rixpress directory.

Usage

```
rxp_list_logs(project_path = ".")
```

Arguments

`project_path`	Character, defaults to ".". Path to the root directory of the project.

Value

A data frame with log filenames, modification times, and file sizes.

See Also

Other utilities: `print.rxp_derivation()`, `rxp_check_chronicles()`, `rxp_copy()`, `rxp_gc()`, `rxp_init()`, `rxp_inspect()`, `rxp_load()`, `rxp_read()`, `rxp_trace()`

Examples

```
## Not run:
  # List all build logs in the current project
  logs <- rxp_list_logs()

  # List logs from a specific project directory
  logs <- rxp_list_logs("path/to/project")

## End(Not run)
```

rxp_load *Load Output of a Derivation*

Description

Loads the output of derivations in the parent frame of the current session, returns a path if reading directly is not possible.

Usage

```
rxp_load(derivation_name, which_log = NULL, project_path = ".")
```

Arguments

derivation_name	Character, the name of the derivation.
which_log	Character, defaults to NULL. If NULL the most recent build log is used. If a string is provided, it's used as a regular expression to match against available log files.
project_path	Character, defaults to ".". Path to the root directory of the project.

Details

When `derivation_name` points to a single R object, it gets loaded in the current session using `assign(..., envir = parent.frame())`, which corresponds to the global environment in a regular interactive session. If you're trying to load a Python object and `{reticulate}` is available, `reticulate::py_load_object()` is used and then the object gets loaded into the global environment. In case the derivation is pointing to several outputs (which can happen when building a Quarto document for example) or loading fails, the path to the object is returned instead.

Value

Nothing, this function has the side effect of loading objects into the parent frame.

See Also

Other utilities: `print.rxp_derivation()`, `rxp_check_chronicles()`, `rxp_copy()`, `rxp_gc()`, `rxp_init()`, `rxp_inspect()`, `rxp_list_logs()`, `rxp_read()`, `rxp_trace()`

Examples

```
## Not run:
  # Load an R object
  rxp_load("mtcars")

  # Load a Python object
  rxp_load("my_python_model")
```

```
  # Load from a specific build log
  rxp_load("mtcars", which_log = "2025-05-10")

## End(Not run)
```

`rxp_make` *Build Pipeline Using Nix*

Description

Runs `nix-build` with a quiet flag, outputting to `_rixpress/result`.

Usage

```
rxp_make(verbose = 0L, max_jobs = 1, cores = 1)
```

Arguments

`verbose`	Integer, defaults to 0L. Verbosity level: 0 = show progress indicators only, 1+ = show nix output with increasing verbosity. 0: "Progress only", 1: "Informational", 2: "Talkative", 3: "Chatty", 4: "Debug", 5: "Vomit". Values higher than 5 are capped to 5. Each level adds one –verbose flag to nix-store command.
`max_jobs`	Integer, number of derivations to be built in parallel.
`cores`	Integer, number of cores a derivation can use during build.

Details

When the `{chronicler}` package is available, `rxp_make()` automatically calls `rxp_check_chronicles()` after a successful build to check for `Nothing` values in chronicle objects. This helps detect silent failures in pipelines that use chronicler's `record()` decorated functions. See `vignette("chronicler")` for more details.

Value

A character vector of paths to the built outputs.

See Also

Other pipeline functions: `rxp_pipeline()`, `rxp_populate()`

Examples

```
## Not run:
  # Build the pipeline with progress indicators (default)
  rxp_make()

  # Build with verbose output and parallel execution
  rxp_make(verbose = 2, max_jobs = 4, cores = 2)
```

```
  # Maximum verbosity
  rxp_make(verbose = 3)

## End(Not run)
```

`rxp_pipeline` *Create a Named Pipeline of Derivations*

Description

Groups multiple derivations into a named pipeline for organizational purposes. This allows you to structure large projects into logical sub-pipelines (e.g., "ETL", "Model", "Report") that are visually distinguished in DAG visualizations.

Usage

```
rxp_pipeline(name, path, color = NULL, ...)
```

Arguments

`name`	Character, the name of the pipeline (e.g., "ETL", "Model").
`path`	Character path to an R script returning a list of derivations, OR a list of derivation objects created by `rxp_r()`, `rxp_py()`, etc.
`color`	Character, optional. A CSS color name (e.g., "darkorange") or hex code (e.g., "#FF5733") to use for this pipeline's nodes in DAG visualizations. If NULL, a default color will be assigned.
`...`	Additional arguments (currently unused, reserved for future use).

Details

The `rxp_pipeline()` function is used to organize derivations into logical groups. When passed to `rxp_populate()`, the derivations are flattened but retain their group and color metadata, which is then used in DAG visualizations (`rxp_visnetwork()` and `rxp_ggdag()`) to distinguish different parts of your workflow.

This pattern enables a "Master Script" workflow where you can define sub-pipelines in separate R scripts that each return a list of derivations. You then pass the paths to these scripts to `rxp_pipeline()`:

Value

An object of class `rxp_pipeline` containing the derivations with pipeline metadata attached.

See Also

Other pipeline functions: `rxp_make()`, `rxp_populate()`

Examples

```
## Not run:
  # Define derivations in separate scripts
  # pipelines/01_etl.R returns: list(rxp_r(...), rxp_r(...))
  # pipelines/02_model.R returns: list(rxp_r(...), rxp_r(...))

  # Master script (run.R):

  # Create named pipelines with colors by pointing to the files
  pipe_etl <- rxp_pipeline("ETL", "pipelines/01_etl.R", color = "darkorange")
  pipe_model <- rxp_pipeline("Model", "pipelines/02_model.R", color = "dodgerblue")

  # Build the combined pipeline
  rxp_populate(list(pipe_etl, pipe_model))
  rxp_make()

  # Visualize - ETL nodes will be orange, Model nodes will be blue
  rxp_visnetwork()

## End(Not run)
```

---

rxp_populate *Generate Nix Pipeline Code*

Description

Generate Nix Pipeline Code

Usage

```
rxp_populate(derivs, project_path = ".", build = FALSE, py_imports = NULL, ...)
```

Arguments

derivs	A list of derivation objects, where each object is a list of five elements: • *name*, name of the derivation, • *snippet*, the nix code snippet to build this derivation, • *type*, can be R, Python or Quarto, • *additional_files*, character vector of paths to files to make available to build sandbox, • *nix_env*, path to Nix environment to build this derivation. A single deriv is the output of `rxp_r()`, `rxp_qmd()` or `rxp_py()` function.
project_path	Path to root of project, defaults to ".".
build	Logical, defaults to FALSE. Should the pipeline get built right after being generated? When FALSE, use `rxp_make()` to build the pipeline at a later stage.

`py_imports` Named character vector of Python import rewrites. Names are the base modules that rixpress auto-imports as "import name", and values are the desired import lines. For example: c(numpy = "import numpy as np", xgboost = "from xgboost import XGBClassifier"). Each entry is applied by replacing "import name" with the provided string across generated _rixpress Python library files.

`...` Further arguments passed down to methods. Use `max-jobs` and `cores` to set parallelism during build. See the documentation of `rxp_make()` for more details.

Details

This function generates a `pipeline.nix` file based on a list of derivation objects. Each derivation defines a build step, and `rxp_populate()` chains these steps and handles the serialization and conversion of Python objects into R objects (or vice-versa). Derivations are created with `rxp_r()`, `rxp_py()` and so on. By default, the pipeline is also immediately built after being generated, but the build process can be postponed by setting `build` to FALSE. In this case, the pipeline can then be built using `rxp_make()` at a later stage. The generated `pipeline.nix` expression includes:

- the required imports of environments, typically `default.nix` files generated by the `rix` package;
- correct handling of interdependencies of the different derivations;
- serialization and deserialization of both R and Python objects, and conversion between them when objects are passed from one language to another;
- correct loading of R and Python packages, or extra functions needed to build specific targets

The `_rixpress` folder contains:

- R, Python or Julia scripts to load the required packages that need to be available to the pipeline.
- a JSON file with the DAG of the pipeline, used for visualisation, and to allow `rxp_populate()` to generate the right dependencies between derivations.
- `.rds` files with build logs, required for `rxp_inspect()` and `rxp_gc()`. See `vignette("debugging")` for more details.

Inline Python import adjustments In some cases, due to the automatic handling of Python packages, users might want to change import statements. By default if, say, `pandas` is needed to build a derivation, it will be imported with `import pandas`. However, Python programmers typically use `import pandas as pd`. You can either:

- use `py_imports` to rewrite these automatically during population, or
- use `adjust_import()` and `add_import()` for advanced/manual control. See `vignette("polyglot")` for more details.

Value

Nothing, writes a file called `pipeline.nix` with the Nix code to build the pipeline, as well as folder called _rixpress with required internal files.

See Also

Other pipeline functions: `rxp_make()`, `rxp_pipeline()`

Examples

```
## Not run:
# Create derivation objects
d1 <- rxp_r(mtcars_am, filter(mtcars, am == 1))
d2 <- rxp_r(mtcars_head, head(mtcars_am))
list_derivs <- list(d1, d2)

# Generate and build in one go
rxp_populate(derivs = list_derivs, project_path = ".", build = TRUE)

# Or only populate, with inline Python import adjustments
rxp_populate(
  derivs = list_derivs,
  project_path = ".",
  build = FALSE,
  py_imports = c(pandas = "import pandas as pd")
)
# Then later:
rxp_make()

## End(Not run)
```

---

rxp_py *Create a Nix Expression Running a Python Function*

Description

Create a Nix Expression Running a Python Function

Usage

```
rxp_py(
  name,
  expr,
  additional_files = "",
  user_functions = "",
  nix_env = "default.nix",
  encoder = NULL,
  decoder = NULL,
  env_var = NULL,
  noop_build = FALSE
)
```

Arguments

name	Symbol, name of the derivation.
expr	Character, Python code to generate the expression. Ideally it should be a call to a pure function. Multi-line expressions are not supported.

additional_files
: Character vector, additional files to include during the build process. For example, if a function expects a certain file to be available, this is where you should include it.

user_functions
: Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.

nix_env
: Character, path to the Nix environment file, default is "default.nix".

encoder
: Character, defaults to NULL. The name of the Python function used to serialize the object. It must accept two arguments: the object to serialize (first), and the target file path (second). If NULL, the default behaviour uses `pickle.dump`. Define this function in `functions.py`. See `vignette("encoding-decoding")` for more details.

decoder
: Character or named vector/list, defaults to NULL. Can be:

  - A single string for the Python function to unserialize all upstream objects
  - A named vector/list where names are upstream dependency names and values are their specific unserialize functions If NULL, the default uses `pickle.load`. See `vignette("encoding-decoding")` for more details.

env_var
: Character vector, defaults to NULL. A named vector of environment variables before running the Python script, e.g., c(PYTHONPATH = "/path/to/modules"). Each entry will be added as an export statement in the build phase.

noop_build
: Logical, defaults to FALSE. If TRUE, the derivation produces a no-op build (a stub output with no actual build steps). Any downstream derivations depending on a no-op build will themselves also become no-op builds.

Details

At a basic level, `rxp_py(mtcars_am, "mtcars.filter(polars.col('am') == 1).to_pandas()")` is equivalent to `mtcars_am = mtcars.filter(polars.col('am') == 1).to_pandas()`. `rxp_py()` generates the required Nix boilerplate to output a so-called "derivation" in Nix jargon. A Nix derivation is a recipe that defines how to create an output (in this case `mtcars_am`) including its dependencies, build steps, and output paths.

Value

An object of class derivation which inherits from lists.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py_file()`, `rxp_qmd()`, `rxp_r()`, `rxp_r_file()`, `rxp_rmd()`

Examples

```
## Not run:
  rxp_py(
```

```
  mtcars_pl_am,
  expr = "mtcars_pl.filter(polars.col('am') == 1).to_pandas()"
)

# Skip building this derivation
rxp_py(
  data_prep,
  expr = "preprocess_data(raw_data)",
  noop_build = TRUE
)

# Custom serialization
rxp_py(
  mtcars_pl_am,
  expr = "mtcars_pl.filter(polars.col('am') == 1).to_pandas()",
  user_functions = "functions.py",
  encoder = "serialize_model",
  additional_files = "some_required_file.bin")

## End(Not run)
```

`rxp_py2r` *Transfer Python Object into an R Session*

Description

Transfer Python Object into an R Session

Usage

```
rxp_py2r(name, expr, nix_env = "default.nix", project_path = ".")
```

Arguments

`name`	Symbol, name of the derivation.
`expr`	Symbol, Python object to be loaded into R.
`nix_env`	Character, path to the Nix environment file, default is "default.nix".
`project_path`	Character, path to the project root, default is ".".

Details

`rxp_py2r(my_obj, my_python_object)` loads a serialized Python object and saves it as an RDS file using `reticulate::py_load_object()`.

Value

An object of class `rxp_derivation`.

See Also

Other interop functions: `rxp_r2py()`

`rxp_py_file` *Creates a Nix Expression That Reads In a File (or Folder of Data) Using Python*

Description

Creates a Nix Expression That Reads In a File (or Folder of Data) Using Python

Usage

```
rxp_py_file(...)
```

Arguments

`...` Arguments passed on to `rxp_file`

`name` Symbol, the name of the derivation.

`path` Character, the file path to include (e.g., "data/mtcars.shp") or a folder path (e.g., "data"). See details.

`read_function` Function, an R function to read the data, taking one argument (the path). This can be a user-defined function that is made available using `user_functions`. See details.

`user_functions` Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.

`nix_env` Character, path to the Nix environment file, default is "default.nix".

`env_var` List, defaults to NULL. A named list of environment variables to set before running the R script, e.g., c(VAR = "hello"). Each entry will be added as an export statement in the build phase.

`encoder` Function/character, defaults to NULL. A language-specific serializer to write the loaded object to disk.

- R: function/symbol/character (e.g., `qs::qsave`) taking `(object, path)`. Defaults to `saveRDS`.
- Python: character name of a function taking `(object, path)`. Defaults to using `pickle.dump`.
- Julia: character name of a function taking `(object, path)`. Defaults to using `Serialization.serialize`.

Details

The basic usage is to provide a path to a file, and the function to read it. For example: `rxp_r_file(mtcars, path = "data/mtcars.csv", read_function = read.csv)`. It is also possible instead to point to a folder that contains many files that should all be read at once, for example: `rxp_r_file(many_csvs, path = "data", read_`
See the `vignette("importing-data")` vignette for more detailed examples.

Value

An object of class `rxp_derivation`.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py()`, `rxp_qmd()`, `rxp_r()`, `rxp_r_file()`, `rxp_rmd()`

`rxp_qmd` *Render a Quarto Document as a Nix Derivation*

Description

Render a Quarto Document as a Nix Derivation

Usage

```
rxp_qmd(
  name,
  qmd_file,
  additional_files = "",
  nix_env = "default.nix",
  args = "",
  env_var = NULL,
  noop_build = FALSE
)
```

Arguments

`name`	Symbol, derivation name.
`qmd_file`	Character, path to .qmd file.
`additional_files`	Character vector, additional files to include, for example a folder containing images to include in the Quarto document.
`nix_env`	Character, path to the Nix environment file, default is "default.nix".
`args`	A character of additional arguments to be passed directly to the `quarto` command.
`env_var`	List, defaults to NULL. A named list of environment variables to set before running the Quarto render command, e.g., c(QUARTO_PROFILE = "production"). Each entry will be added as an export statement in the build phase.
`noop_build`	Logical, defaults to FALSE. If TRUE, the derivation produces a no-op build (a stub output with no actual build steps). Any downstream derivations depending on a no-op build will themselves also become no-op builds.

Details

To include built derivations in the document, `rxp_read("derivation_name")` should be put in the .qmd file.

Value

An object of class derivation which inherits from lists.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py()`, `rxp_py_file()`, `rxp_r()`, `rxp_r_file()`, `rxp_rmd()`

Examples

```
## Not run:
  # Compile a .qmd file to a pdf using typst
  # `images` is a folder containing images to include in the Quarto doc
  rxp_qmd(
    name = report,
    qmd_file = "report.qmd",
    additional_files = "images",
    args = "--to typst"
  )

  # Skip building this derivation
  rxp_qmd(
    name = draft_report,
    qmd_file = "draft.qmd",
    noop_build = TRUE
  )

## End(Not run)
```

---

rxp_r *Create a Nix Expression Running an R Function*

Description

Create a Nix Expression Running an R Function

Usage

```
rxp_r(
  name,
  expr,
  additional_files = "",
  user_functions = "",
```

```
  nix_env = "default.nix",
  encoder = NULL,
  decoder = NULL,
  env_var = NULL,
  noop_build = FALSE
)
```

Arguments

name	Symbol, name of the derivation.
expr	R code to generate the expression. Ideally it should be a call to a pure function, or a piped expression. Multi-line expressions are not supported.
additional_files	Character vector, additional files to include during the build process. For example, if a function expects a certain file to be available, this is where you should include it.
user_functions	Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.
nix_env	Character, path to the Nix environment file, default is "default.nix".
encoder	Function or character defaults to NULL. A function used to encode (serialize) objects for transfer between derivations. It must accept two arguments: the object to encode (first), and the target file path (second). If your function has a different signature, wrap it to match this interface. By default, `saveRDS()` is used, but this may yield unexpected results, especially for complex objects like machine learning models. For instance, for `{keras}` models, use `keras::save_model_hdf5()` to capture the full model (architecture, weights, training config, optimiser state, etc.). See `vignette("encoding-decoding")` for more details.
decoder	Function, character, or named vector/list, defaults to NULL. Can be: • A single function/string to decode (unserialize) all upstream objects (e.g., `readRDS`) • A named vector/list where names are upstream dependency names and values are their specific decoding functions (e.g., `c(mtcars_tail = "qs::qread", mtcars_head = "read.csv")`) By default, `readRDS()` is used. See `vignette("encoding-decoding` for more details.
env_var	Character vector, defaults to NULL. A named vector of environment variables to set before running the R script, e.g., `c("CMDSTAN" = "${defaultPkgs.cmdstan}/opt/cmdstan)"`. Each entry will be added as an export statement in the build phase.
noop_build	Logical, defaults to FALSE. If TRUE, the derivation produces a no-op build (a stub output with no actual build steps). Any downstream derivations depending on a no-op build will themselves also become no-op builds.

Details

At a basic level, `rxp_r(mtcars_am, filter(mtcars, am == 1))` is equivalent to `mtcars_am <- filter(mtcars, am == 1)`. `rxp_r()` generates the required Nix boilerplate to output a so-called

"derivation" in Nix jargon. A Nix derivation is a recipe that defines how to create an output (in this case `mtcars_am`) including its dependencies, build steps, and output paths.

Value

An object of class derivation which inherits from lists.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py()`, `rxp_py_file()`, `rxp_qmd()`, `rxp_r_file()`, `rxp_rmd()`

Examples

```
## Not run:
  # Basic usage
  rxp_r(name = filtered_mtcars, expr = filter(mtcars, am == 1))

  # Skip building this derivation
  rxp_r(
    name = turtles,
    expr = occurrence(species, geometry = atlantic),
    noop_build = TRUE
  )

  # Serialize object using qs
  rxp_r(
   name = filtered_mtcars,
   expr = filter(mtcars, am == 1),
   encoder = qs::qsave
  )
  # Unerialize using qs::qread in the next derivation
  rxp_r(
   name = mtcars_mpg,
   expr = select(filtered_mtcars, mpg),
   decoder = qs::qread
  )

## End(Not run)
```

---

rxp_r2py *Transfer R Object into a Python Session*

Description

Transfer R Object into a Python Session

Usage

```
rxp_r2py(name, expr, nix_env = "default.nix", project_path = ".")
```

Arguments

name	Symbol, name of the derivation.
expr	Symbol, R object to be saved into a Python pickle.
nix_env	Character, path to the Nix environment file, default is "default.nix".
project_path	Character, path to the project root, default is ".".

Details

`rxp_r2py(my_obj, my_r_object)` saves an R object to a Python pickle using `reticulate::py_save_object()`.

Value

An object of class `rxp_derivation`.

See Also

Other interop functions: `rxp_py2r()`

---

rxp_read *Read Output of a Derivation*

Description

Reads the output of derivations in the current session, returns a path if reading directly is not possible.

Usage

```
rxp_read(derivation_name, which_log = NULL, project_path = ".")
```

Arguments

derivation_name	Character, the name of the derivation.
which_log	Character, defaults to NULL. If NULL the most recent build log is used. If a string is provided, it's used as a regular expression to match against available log files.
project_path	Character, defaults to ".". Path to the root directory of the project.

Details

When `derivation_name` points to a single R object, it gets read in the current session using `readRDS()`. If it's a Python object and `{reticulate}` is available, `reticulate::py_load_object()` is used. In case the derivation is pointing to several outputs (which can happen when building a Quarto document for example) or neither `readRDS()` nor `reticulate::py_load_object()` successfully read the object, the path to the object is returned instead.

Value

The derivation's output.

See Also

Other utilities: print.rxp_derivation(), rxp_check_chronicles(), rxp_copy(), rxp_gc(), rxp_init(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_trace()

Examples

```
## Not run:
  mtcars <- rxp_read("mtcars")

  # Read from a specific build log
  mtcars <- rxp_read("mtcars", which_log = "2025-05-10")

## End(Not run)
```

---

rxp_rmd *Render an R Markdown Document as a Nix Derivation*

Description

Render an R Markdown Document as a Nix Derivation

Usage

```
rxp_rmd(
  name,
  rmd_file,
  additional_files = "",
  nix_env = "default.nix",
  params = NULL,
  env_var = NULL,
  noop_build = FALSE
)
```

Arguments

name	Symbol, derivation name.
rmd_file	Character, path to .Rmd file.
additional_files	Character vector, additional files to include, for example a folder containing the pictures to include in the R Markdown document.
nix_env	Character, path to the Nix environment file, default is "default.nix".
params	List, parameters to pass to the R Markdown document. Default is NULL.

env_var	List, defaults to NULL. A named list of environment variables to set before running the R Markdown render command, e.g., c(RSTUDIO_PANDOC = "/path/to/pandoc"). Each entry will be added as an export statement in the build phase.
noop_build	Logical, defaults to FALSE. If TRUE, the derivation produces a no-op build (a stub output with no actual build steps). Any downstream derivations depending on a no-op build will themselves also become no-op builds.

Details

To include objects built in the pipeline, `rxp_read("derivation_name")` should be put in the .Rmd file.

Value

An object of class derivation which inherits from lists.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py()`, `rxp_py_file()`, `rxp_qmd()`, `rxp_r()`, `rxp_r_file()`

Examples

```
## Not run:
  # Compile a .Rmd file to a pdf
  # `images` is a folder containing images to include in the R Markdown doc
  rxp_rmd(
    name = report,
    rmd_file = "report.Rmd",
    additional_files = "images"
  )

  # Skip building this derivation
  rxp_rmd(
    name = draft_report,
    rmd_file = "draft.Rmd",
    noop_build = TRUE
  )

## End(Not run)
```

---

rxp_r_file *Creates a Nix Expression That Reads In a File (or Folder of Data) Using R*

Description

Creates a Nix Expression That Reads In a File (or Folder of Data) Using R

Usage

```
rxp_r_file(...)
```

Arguments

...	Arguments passed on to rxp_file
	name Symbol, the name of the derivation.
	path Character, the file path to include (e.g., "data/mtcars.shp") or a folder path (e.g., "data"). See details.
	read_function Function, an R function to read the data, taking one argument (the path). This can be a user-defined function that is made available using user_functions. See details.
	user_functions Character vector, user-defined functions to include. This should be a script (or scripts) containing user-defined functions to include during the build process for this derivation. It is recommended to use one script per function, and only include the required script(s) in the derivation.
	nix_env Character, path to the Nix environment file, default is "default.nix".
	env_var List, defaults to NULL. A named list of environment variables to set before running the R script, e.g., c(VAR = "hello"). Each entry will be added as an export statement in the build phase.
	encoder Function/character, defaults to NULL. A language-specific serializer to write the loaded object to disk.
	• R: function/symbol/character (e.g., qs::qsave) taking (object, path). Defaults to saveRDS.
	• Python: character name of a function taking (object, path). Defaults to using pickle.dump.
	• Julia: character name of a function taking (object, path). Defaults to using Serialization.serialize.

Details

The basic usage is to provide a path to a file, and the function to read it. For example: `rxp_r_file(mtcars, path = "data/mtcars.csv", read_function = read.csv)`. It is also possible instead to point to a folder that contains many files that should all be read at once, for example: `rxp_r_file(many_csvs, path = "data", read_`
See the `vignette("importing-data")` vignette for more detailed examples.

Value

An object of class `rxp_derivation`.

See Also

Other derivations: `rxp_jl()`, `rxp_jl_file()`, `rxp_py()`, `rxp_py_file()`, `rxp_qmd()`, `rxp_r()`, `rxp_rmd()`

rxp_trace *Trace Lineage of Derivations*

Description

Trace Lineage of Derivations

Usage

```
rxp_trace(
  name = NULL,
  dag_file = file.path("_rixpress", "dag.json"),
  transitive = TRUE,
  include_self = FALSE
)
```

Arguments

name	Character, defaults to NULL. Name of the derivation to inspect. If NULL, the function prints the whole pipeline (inverted global view).
dag_file	Character, defaults to "_rixpress/dag.json". Path to dag.json.
transitive	Logical, defaults to TRUE. If TRUE, show transitive closure and mark transitive-only nodes with "*". If FALSE, show immediate neighbours only.
include_self	Logical, defaults to FALSE. If TRUE, include name itself in the results.

Value

Invisibly, a named list mapping each inspected derivation name to a list with elements: - dependencies - reverse_dependencies The function also prints a tree representation to the console.

See Also

Other utilities: print.rxp_derivation(), rxp_check_chronicles(), rxp_copy(), rxp_gc(), rxp_init(), rxp_inspect(), rxp_list_logs(), rxp_load(), rxp_read()

rxp_visnetwork *Create a Directed Acyclic Graph (DAG) Representing the Pipeline Using* {visNetwork}

Description

Uses {visNetwork} to generate the plot. {visNetwork} is a soft dependency of {rixpress} so you need to install it to use this function. When derivations are organized into pipelines using rxp_pipeline(), nodes use a dual-encoding approach: the interior fill shows the derivation type (R, Python, etc.) while the border shows the pipeline group colour.

Usage

```
rxp_visnetwork(
  nodes_and_edges = get_nodes_edges(),
  color_by = c("pipeline", "type"),
  colour_by = NULL
)
```

Arguments

nodes_and_edges	List, output of `get_nodes_edges()`.
color_by	Character, either "pipeline" (default) or "type". When "pipeline", nodes show type as fill colour and pipeline as border. When "type", nodes are colored by their derivation type (rxp_r, rxp_py, etc.).
colour_by	Character, alias for `color_by`.

Value

Nothing, this function opens a new tab in your browser with the DAG generated using `{visNetwork}`.

See Also

Other visualisation functions: `rxp_ggdag()`

Examples

```
## Not run:
  rxp_visnetwork()
  rxp_visnetwork(colour_by = "type")  # Color by derivation type instead

## End(Not run)
```

---

rxp_write_dag *Generate a DAG From a List of Derivations*

Description

Creates a JSON representation of a directed acyclic graph (DAG) based on dependencies between derivations. Is automatically called by `rxp_populate()`.

Usage

```
rxp_write_dag(rxp_list, output_file = "_rixpress/dag.json")
```

Arguments

rxp_list	A list of derivations.
output_file	Path to the output JSON file. Defaults to "_rixpress/dag.json".

Value

Nothing, writes a JSON file representing the DAG.

See Also

Other ci utilities: `rxp_dag_for_ci()`, `rxp_ga()`

Examples

```
## Not run:
  rxp_write_dag(rxp_list)

## End(Not run)
```

Index