Package: SigBridgeRUtils (via r-universe)

May 13, 2026

Title Some Utilities & Base Supports for 'SigBridgeR'

Version 0.2.6

Description Provides fundamental function support for 'SigBridgeR' and its single-cell phenotypic screening algorithm, including optional functions.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Suggests furrr, future, MASS, Matrix, matrixStats, microbenchmark, preprocessCore, Rfast, SeuratObject, sparseMatrixStats, testthat

Imports chk, cli, data.table, processx, purrr, reticulate, rlang

Config/testthat/parallel true

Config/testthat/edition 3

NeedsCompilation no

Author Yuxi Yang [cre, aut] (ORCID: <https://orcid.org/0009-0006-1329-1224>)

Maintainer Yuxi Yang <15364051195@163.com>

Config/pak/sysreqs libpng-dev python3

Repository https://exceret.r-universe.dev

Date/Publication 2026-03-12 19:40:07 UTC

RemoteUrl https://github.com/cran/SigBridgeRUtils

RemoteRef HEAD

RemoteSha 478226b616b5dfc90219c4e5fad2751dcdfca601

Contents

`AddCaller2cli` *A Decorator for Adding Caller Info to CLI Functions*

Description

Wraps CLI functions to automatically prepend the caller's identity (function name or 'global') to the output message.

Usage

```
AddCaller2cli(cli_func)
```

Arguments

`cli_func`	A CLI function (e.g., `cli_alert_info`).

Value

A wrapper function that formats output as "[caller]: message".

AddMisc *Safely Add Miscellaneous Data to Seurat Object*

Description

Adds arbitrary data to the `@misc` slot of a Seurat object with automatic key conflict resolution. If the key already exists, automatically appends a numeric suffix to ensure unique key naming (e.g., "mykey_1", "mykey_2").

Usage

```
AddMisc(
  seurat_obj,
  ..., # key = value
  cover = TRUE # overwrite existing data
)
```

Arguments

`seurat_obj`	A Seurat object to modify
`...`	key-value pairs to add to the `@misc` slot.
`cover`	Logical indicating whether to overwrite existing data. If (default TRUE).

Value

The modified Seurat object with added `@misc` data. The original object structure is preserved with no other modifications.

Key Generation Rules

1. If `key` doesn't exist: uses as-is
2. If `key` exists: appends the next available number (e.g., "key_1", "key_2")
3. If numbered keys exist (e.g., "key_2"): increments the highest number

Examples

```
## Not run:
# Basic usage
seurat_obj <- AddMisc(seurat_obj, "QC_stats" = qc_df)

# Auto-incrementing example
seurat_obj <- AddMisc(seurat_obj, markers = markers1)
seurat_obj <- AddMisc(seurat_obj, markers = markers2, cover=FALSE)
# Stores as "markers" and "markers_1"


## End(Not run)
```

AddTimeStamp2cli *A Decorator for Adding Timestamp to CLI Functions*

Description

A higher-order function that wraps CLI functions to automatically prepend timestamps to their output messages. This creates a modified version of any CLI function that includes timestamp information in its output.

Usage

```
AddTimeStamp2cli(cli_func)
```

Arguments

cli_func	A CLI function from the cli package (e.g., cli_alert_info, cli_warn) that will be wrapped with timestamp functionality.

Details

This function uses force to ensure the CLI function is evaluated at creation time. The timestamp is generated using a TimeStamp() function which should be available in the execution environment and is inserted using cli's glue-like syntax "{ }".

Value

Returns a modified version of the input function that automatically adds a timestamp in the format "[{timestamp}]" to the beginning of all character messages passed to it.

See Also

CreateTimeStampCliEnv for creating a complete environment of timestamped CLI functions.

Other TimeStamp: CreateTimeStampCliEnv(), TimeStamp()

Examples

```
## Not run:
# Create a timestamp-enabled version of cli_alert_info
timestamp_alert <- AddTimeStamp2cli(cli::cli_alert_info)
timestamp_alert("This message will have a timestamp")

## End(Not run)
```

`all_identical` *Check all the Objects Are Identical*

Description

all_identical checks if all provided objects are pairwise identical and returns a symmetric matrix showing the comparison results between all object pairs.

Usage

```
all_identical(..., names = NULL)
```

Arguments

`...`	Objects to compare for identity
`names`	Optional character vector of names for the objects. If not provided, objects will be named as "Obj1", "Obj2", etc.

Details

This function provides a comprehensive way to compare multiple objects at once, returning a matrix that shows all pairwise comparisons. This is particularly useful for testing and validation scenarios where you need to verify that multiple objects are identical.

Value

A symmetric logical matrix where entry [i, j] indicates whether object i is identical to object j. The matrix has dimensions n x n where n is the number of objects, with row and column names from the names parameter.

See Also

`identical()` for pairwise object comparison, `matrix()` for creating matrices

Examples

```
# Compare identical objects
x <- 1:5
y <- 1:5
z <- 1:5

# All objects are identical
result <- all_identical(x, y, z)
print(result)

# Compare different objects with custom names
a <- 1:3
b <- 1:5
```

```
c <- 1:3

result2 <- all_identical(a, b, c, names = c("first", "second", "third"))
print(result2)

# Single object case
single_result <- all_identical(x)
print(single_result)
```

CreateCallerCliEnv *Create Environment with Caller-Aware CLI Functions*

Description

Generates an environment containing CLI functions that automatically report their caller (Global or Function Name).

Usage

```
CreateCallerCliEnv(
  cli_functions = c("cli_alert_info", "cli_alert_success", "cli_alert_warning",
    "cli_alert_danger", "cli_inform")
)
```

Arguments

`cli_functions`	Character vector of function names from package `cli`.

Value

An environment with wrapped functions.

Examples

```
## Not run:
caller_cli <- CreateCallerCliEnv()

# Global context
caller_cli$cli_alert_info("Hello")
# Output: [global]: Hello

# Function context
f <- function(x) { caller_cli$cli_alert_success("Result is {x}") }
f(100)
# Output: [f()]: Result is 100

## End(Not run)
```

`CreateTimeStampCliEnv` *Create Environment with Timestamped CLI Functions*

Description

Generates an environment containing wrapped versions of common CLI functions that automatically include timestamps in their output. This provides a convenient way to use multiple CLI functions with consistent timestamping.

Usage

```
CreateTimeStampCliEnv(
  cli_functions = c("cli_alert_info", "cli_alert_success", "cli_alert_warning",
    "cli_alert_danger")
)
```

Arguments

`cli_functions`	from package `cli` to be wrapped with timestamp functionality.

Details

This function creates a new environment and populates it with timestamped versions of commonly used CLI functions from the `cli` package. Only functions that exist in the loaded `cli` package are added to the environment. The function uses `walk` for side-effect iteration over the function names.

Value

Returns an environment containing timestamp-wrapped versions of:

- `cli_warn`
- `cli_alert_info`
- `cli_alert_success`
- `cli_alert_warning`
- `cli_alert_danger`

Each function in the environment will automatically prepend timestamps to its output messages.

See Also

`AddTimeStamp2cli` for the wrapper function used internally.

Other TimeStamp: `AddTimeStamp2cli()`, `TimeStamp()`

Examples

```
## Not run:
# Create environment with timestamped CLI functions
cli_env <- CreateTimeStampCliEnv()

# Use timestamped functions
cli_env$cli_alert_info("System started")
cli_env$cli_alert_success("Operation completed")

## End(Not run)
```

`FilterArgs4Func` *Keep Wanted Arguments According to A Function from Dots*

Description

FilterArgs4Func filters a list of arguments to include only those that match the formal arguments of a specified function, with optional support for preserving additional arguments via the `keep` parameter. This is useful for preparing argument lists for function calls, especially when dealing with functions that have many optional parameters or when passing arguments through multiple function layers.

Usage

```
FilterArgs4Func(args_list, fun, keep = NULL)
```

Arguments

`args_list`	A named list of arguments to filter
`fun`	The target function whose formal arguments will be used for filtering
`keep`	Character vector of argument names to preserve regardless of whether they appear in `fun`'s formal parameters. Default is `NULL`. Useful for retaining arguments needed by downstream functions or wrappers that consume `...`.

Details

This function is particularly useful in scenarios where you have a large list of parameters and want to pass only the relevant ones to a specific function while preserving certain arguments for downstream processing (e.g., arguments consumed by nested `...` parameters).

The `keep` parameter enables flexible argument forwarding patterns common in wrapper functions and pipeline designs.

Value

A filtered list containing:

- Arguments from `args_list` that match formal parameters of `fun` (excluding the "..." parameter)
- Additional arguments specified in `keep` (if not `NULL`)

See Also

`formals()` for accessing function formal arguments, `do.call()` for executing functions with argument lists, `names()` for working with list names

Examples

```
## Not run:
# Example function with specific parameters
example_function <- function(a, b, c = 10, d = 20) {
  return(a + b + c + d)
}

# Create a list with both relevant and irrelevant arguments
all_args <- list(
  a = 1,
  b = 2,
  c = 3,
  e = 4,  # Not in function formals
  f = 5   # Not in function formals
)

# Basic usage: filter to only include arguments matching function parameters
filtered_args <- FilterArgs4Func(all_args, example_function)
print(filtered_args)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
#>
#> $c
#> [1] 3

# Advanced usage: preserve additional arguments for downstream processing
filtered_with_keep <- FilterArgs4Func(all_args, example_function, keep = c("e", "f"))
print(filtered_with_keep)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
#>
#> $c
#> [1] 3
```

```
#>
#> $e
#> [1] 4
#>
#> $f
#> [1] 5

# Execute with filtered arguments
result <- do.call(example_function, filtered_args)
print(result)
#> [1] 16

## End(Not run)
```

`GetCallerInfo` *Get Caller Name*

Description

Retrieves the name of the function that called the current execution context. If called from the global environment, returns "global".

Usage

```
GetCallerInfo(offset = 2)
```

Arguments

`offset`	Integer. The number of stack frames to go back. Defaults to 2 (skipping the GetCallerInfo frame and the Wrapper frame to find the User's function).

Value

Character string. E.g., "my_function()" or "global".

Examples

```
## Not run:
f <- function() { GetCallerInfo() }
f() # Returns "f()"

## End(Not run)
```

GetPythonPath *Get Python Executable Path*

Description

This function attempts to find the Python executable within a given environment path. It checks different candidate paths based on the operating system.

Usage

```
GetPythonPath(path)
```

Arguments

path	Character string specifying the path to the environment where Python might be located.

Value

Character string with the normalized path to the Python executable if found, otherwise NA_character_.

ginv2 *Generalized Inverse (Moore-Penrose Inverse)*

Description

Compute the Moore-Penrose generalized inverse of a matrix. This is an S3 generic function with methods for base matrices, dense Matrix objects, and sparse Matrix objects.

Default method for base R matrices (from MASS::ginv)

Usage

```
ginv2(X, tol = sqrt(.Machine$double.eps), ...)

## Default S3 method:
ginv2(X, tol = sqrt(.Machine$double.eps), ...)
```

Arguments

X	A numeric or complex matrix
tol	Tolerance for determining rank. Default is sqrt(.Machine$double.eps)
...	Additional arguments passed to methods

Value

The generalized inverse of X

Examples

```
## Not run:
# Base R matrix
m <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)
ginv2(m)

# Dense Matrix
library(Matrix)
dm <- Matrix(m, sparse = FALSE)
ginv2(dm)

# Sparse Matrix
sm <- Matrix(m, sparse = TRUE)
ginv2(sm)

## End(Not run)
```

`ListPyEnv` *List Available Python Environments*

Description

Discovers and lists available Python environments of various types on the system. This generic function provides a unified interface to find Conda environments and virtual environments (venv) through S3 method dispatch.

Default method that lists all Python environments by combining results from Conda and virtual environment discovery methods.

Discovers Conda environments using multiple detection strategies for maximum reliability. First attempts to use system Conda commands, then falls back to reticulate's built-in Conda interface if Conda command is unavailable or fails. Returns empty data frame if Conda is not available or no environments are found.

Discovers virtual environments by searching common venv locations including user directories (`~/.virtualenvs`, `~/.venvs`) and project folders (`./venv`, `./.venv`). Supports custom search paths through the `venv_locations` parameter. Returns empty data frame if no virtual environments are found in the specified locations.

Usage

```
ListPyEnv(
  env_type = c("all", "conda", "venv", "virtualenv"),
  timeout = 30000L,
  venv_locations = c("~/.virtualenvs", "~/.venvs", "./venv", "./.venv"),
  verbose = TRUE,
  ...
)

## Default S3 method:
```

```
ListPyEnv(
  env_type = c("all", "conda", "venv", "virtualenv"),
  timeout = 30000L,
  venv_locations = c("~/.virtualenvs", "~/.venvs", "./venv", "./.venv"),
  verbose = getFuncOption("verbose") %||% TRUE,
  ...
)

## S3 method for class 'conda'
ListPyEnv(
  env_type = c("all", "conda", "venv", "virtualenv"),
  timeout = 30000L,
  venv_locations = c("~/.virtualenvs", "~/.venvs", "./venv", "./.venv"),
  verbose = getFuncOption("verbose") %||% TRUE,
  ...
)

## S3 method for class 'venv'
ListPyEnv(
  env_type = c("all", "conda", "venv", "virtualenv"),
  timeout = 30000L,
  venv_locations = c("~/.virtualenvs", "~/.venvs", "./venv", "./.venv"),
  verbose = getFuncOption("verbose") %||% TRUE,
  ...
)
```

Arguments

`env_type`	Character string specifying the type of environments to list. One of: `"all"`, `"conda"`, `"venv"`. Defaults to `"all"`.
`timeout`	The maximum timeout time when using system commands, only effective when `env_type=conda`.
`venv_locations`	Character vector of directory paths to search for virtual environments. Default includes standard locations and common project directories.
`verbose`	Logical indicating whether to print verbose output.
`...`	For future use.

Details

The function uses S3 method dispatch to handle different environment types:

- `"all"`: Combines results from all environment types using `rbind()`
- `"conda"`: Searches for Conda environments using multiple methods:
  - Primary: `reticulate::conda_list()` for reliable environment detection
  - Fallback: System `conda info --envs` command for broader compatibility
- `"venv"`: Searches common virtual environment locations including user directories and project folders

Each method includes comprehensive error handling and will return empty results with informative warnings if no environments are found or if errors occur during discovery.

Value

A data frame with the following columns:

- `name` - Character vector of environment names
- `python` - Character vector of paths to Python executables
- `type` - Character vector indicating environment type (`"conda"` or `"venv"`)

Returns an empty data frame with these columns if no environments are found.

Examples

```
## Not run:
# List all Python environments
ListPyEnv("all")

# List only Conda environments
ListPyEnv("conda")

# List only virtual environments with custom search paths
ListPyEnv("venv", venv_locations = c("~/my_envs", "./project_env"))

## End(Not run)
```

MatchArg *Argument Matching with Default Fallback*

Description

A robust argument matching function that supports exact matching, partial matching, and provides sensible defaults when no match is found. This is an internal utility function not intended for direct use by package users.

Usage

```
MatchArg(arg, choices, default = choices[1], call = rlang::caller_env(), ...)
```

Arguments

`arg`	The argument to match against choices. Can be `NULL` or a character vector.
`choices`	A character vector of valid choices to match against.
`default`	The default value to return if no match is found and `arg` is `NULL`. Defaults to the first element of `choices`.
`call`	caller env
`...`	No usage

Details

This function provides a more flexible alternative to `base::match.arg()` with the following matching strategy:

1. If `arg` is `NULL`, returns the `default` value
2. Attempts exact matching using `base::match()`
3. Falls back to partial matching using `base::pmatch()`
4. If no match found and `default` is not `NULL`, returns `default`
5. Otherwise, throws an informative error with valid choices

The function uses `rlang::caller_env()` for accurate error reporting in the context where the function was called.

Value

Returns the matched choice from the `choices` vector. If no match is found and `arg` is `NULL`, returns the `default` value. If no match is found and `arg` is not `NULL`, throws an informative error.

See Also

`match` for exact matching \ `pmatch` for partial matching \ `caller_env` for calling environment context

Examples

```
## Not run:
# Internal usage examples
MatchArg("app", c("apple", "banana", "application"))  # Returns "apple"
MatchArg(NULL, c("red", "green", "blue"))             # Returns "red" (default)
MatchArg("gr", c("red", "green", "blue"))             # Returns "green"

# Would error: MatchArg("invalid", c("valid1", "valid2"))

## End(Not run)
```

`MatchFunc2Args` *Match Functions to Argument List*

Description

Identifies functions compatible with a given set of named arguments. A function is considered compatible if:

- It has a `...` parameter (**loose matching**), OR
- All argument names in `args_list` exist in its formal parameters (**strict matching**, default behavior).

Usage

```
MatchFunc2Args(
  args_list,
  ...,
  name_only = FALSE,
  top_one_only = FALSE,
  dots_enabled = FALSE
)
```

Arguments

args_list	Named list of arguments to match. Must have non-empty names when non-empty.
...	Functions to test for compatibility.
name_only	Logical. If TRUE, return character vector of function names/identifiers instead of function objects. Default: FALSE.
top_one_only	Logical. If TRUE, return only the single best-matching function (ranked by number of matched parameters and parameter position). Default: FALSE.
dots_enabled	Logical. If TRUE, enable loose matching: any function with a ... parameter is considered compatible regardless of other parameter names. Default: FALSE (strict matching).

Value

- name_only = FALSE, top_one_only = FALSE: List of compatible function objects
- name_only = TRUE, top_one_only = FALSE: Character vector of function identifiers (named functions retain their symbol name; anonymous functions become "anonymous_<index>")
- top_one_only = TRUE: Single function object or name (depending on name_only)

See Also

FilterArgs4Func() for filtering arguments to a function.(Reverse of this function)

Examples

```
## Not run:
f1 <- function(a, b) a + b
f2 <- function(x, y, ...) x * y
f3 <- function(p, q) p - q

args <- list(a = 1, b = 2)

# Strict matching (default): returns f1 only
MatchFunc2Args(args, f1, f2, f3)

# Loose matching: returns f1 and f2 (both accept 'a' and 'b' via strict match or ...)
MatchFunc2Args(args, f1, f2, f3, dots_enabled = TRUE)
```

```
# Return only function names
MatchFunc2Args(args, f1, f2, name_only = TRUE)
# Returns: c("f1", "f2") when dots_enabled=TRUE

## End(Not run)
```

`matrix-stats` *Matrix Statistics Functions (Not Necessarily)*

Description

A collection of functions for computing matrix statistics with fallback implementations when specialized packages are not available.

- sparseMatrixStats -> sparseMatrix
- matrixStats -> matrix
- Matrix -> denseMatrix
- base R -> matrix

normalize.quantiles performs quantile normalization on a matrix, transforming the distributions of each column to match a common target distribution. Uses preprocessCore::normalize.quantiles if available, otherwise provides a pure R implementation.

Usage

```
rowMeans3(x, na.rm = FALSE, ...)

colMeans3(x, na.rm = FALSE, ...)

rowVars3(x, na.rm = FALSE, ...)

colVars3(x, na.rm = FALSE, ...)

rowSds3(x, na.rm = FALSE, ...)

colSds3(x, na.rm = FALSE, ...)

colQuantiles3(x, probs = seq(0, 1, 0.25), ...)

rowMaxs3(
  x,
  rows = NULL,
  cols = NULL,
  na.rm = FALSE,
  dim. = dim(x),
  ...,
  useNames = TRUE
```

```
)

colSums3(x, na.rm = FALSE, ...)

rowSums3(x, na.rm = FALSE, ...)

rowMedians3(
  x,
  rows = NULL,
  cols = NULL,
  na.rm = FALSE,
  dim. = dim(x),
  ...,
  useNames = TRUE
)

colMedians3(
  x,
  rows = NULL,
  cols = NULL,
  na.rm = FALSE,
  dim. = dim(x),
  ...,
  useNames = TRUE
)

normalize.quantiles(x, copy = TRUE, keep.names = FALSE, ...)
```

Arguments

x	A numeric matrix where columns represent samples and rows represent features.
na.rm	Logical indicating whether to remove missing values
...	Additional arguments (currently not used).
probs	Numeric vector of probabilities with values between 0 and 1.
rows, cols	Indices specifying subset of rows/columns to operate over
dim.	Dimensions of the input matrix
useNames	Logical indicating whether to preserve row names in output
copy	Logical indicating whether to work on a copy of the matrix (TRUE) or modify in-place (FALSE).
keep.names	Logical indicating whether to preserve row and column names.

Value

A numeric vector of length nrow(x) containing row variances.

A numeric vector of length nrow(x) containing row variances.

A numeric vector of length nrow(x) containing row variances.

A numeric vector of length ncol(x) containing column variances.

A numeric vector of length nrow(x) containing row standard deviations.

A numeric vector of length ncol(x) containing column standard deviations.

A matrix of quantiles with length(probs) rows and ncol(x) columns.

A numeric vector of length nrow(x) containing row maximums

A numeric vector of length `ncol(x)` containing column sums.

A numeric vector of length `nrow(x)` containing row sums.

A numeric vector of length `nrow(x)` containing row sums.

A numeric vector of length `nrow(x)` containing row sums.

A numeric matrix of the same dimensions as x with quantile-normalized data.

See Also

`matrixStats::rowVars()` for the underlying implementation

`matrixStats::colSds()` for the underlying implementation

`preprocessCore::normalize.quantiles()` for the underlying implementation

Examples

```
mat <- matrix(rnorm(100), nrow = 10, ncol = 10)

# Compute row variances
row_vars <- rowVars3(mat)

# With missing values
mat[1, 1] <- NA
row_vars_na <- rowVars3(mat, na.rm = TRUE)

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)

# Compute column variances
col_vars <- colVars3(mat)

# With missing values
mat[1, 1] <- NA
col_vars_na <- colVars3(mat, na.rm = TRUE)

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)
row_sds <- rowSds3(mat)

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)
col_sds <- colSds3(mat)

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)

# Compute quartiles for each column
quartiles <- colQuantiles3(mat)
```

```
# Compute specific quantiles
specific_quantiles <- colQuantiles3(mat, probs = c(0.1, 0.5, 0.9))

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)

# Compute row maximums
row_maxs <- rowMaxs3(mat)

# With missing values
mat[1, 1] <- NA
row_maxs_na <- rowMaxs3(mat, na.rm = TRUE)

mat <- matrix(rnorm(100), nrow = 10, ncol = 10)

# Perform quantile normalization
normalized_mat <- normalize.quantiles(mat)

# Preserve original names
rownames(mat) <- paste0("Gene", 1:10)
colnames(mat) <- paste0("Sample", 1:10)
normalized_with_names <- normalize.quantiles(mat, keep.names = TRUE)
```

`NULL_or` *Null Coalescing Operator*

Description

The %||% operator provides a convenient way to handle NULL values by returning a default value when the left-hand side is NULL. This is particularly useful for providing fallback values in function arguments and data processing.

Usage

```
lhs %||% rhs
```

Arguments

`lhs`	Left-hand side value to check for NULL
`rhs`	Right-hand side value to return if lhs is NULL

Details

This operator follows the same semantics as the null coalescing operator found in other programming languages (e.g., ?? in C#, ?: in JavaScript). It provides a concise way to specify default values without verbose if-else statements.

Value

Returns lhs if it is not NULL, otherwise returns rhs.

See Also

`is.null()` for checking NULL values, `ifelse()` for more complex conditional logic

Examples

```
## Not run:
# Basic usage with NULL values
NULL %||% "default value"
# Returns "default value"

"actual value" %||% "default value"
# Returns "actual value"

# Practical use in functions
my_function <- function(x = NULL) {
x <- x %||% "default_parameter"
print(x)
}

my_function() # Prints "default_parameter"
my_function("custom_value") # Prints "custom_value"

# Handling potentially NULL results
result <- tryCatch(
some_operation(),
error = function(e) NULL
)

final_value <- result %||% "fallback"

## End(Not run)
```

---

rowname-utils *Row Name Utilities*

Description

Functions for converting between column-based row names and explicit row name attributes. These utilities provide convenient ways to handle row names in data frames and matrices, serving as alternatives to tibble's row name handling functions.

Col2Rownames converts a specified column to row names and removes the original column. This is useful when working with data that has identifier columns that should serve as row names rather than regular data columns.

Rownames2Col converts row names to an explicit column and removes the row names attribute. This is useful when preparing data for functions that don't handle row names well, or when row names need to be included as regular data for analysis or visualization.

Usage

```
Col2Rownames(.data, var = "rowname")

Rownames2Col(.data, var = "rowname")
```

Arguments

.data	A data frame or matrix-like object with row names
var	Character string specifying the name for the new column that will contain the row names. Defaults to "rowname".

Value

The input object with row names set from the specified column and that column removed from the data.

The input object with row names converted to a new column and the row names attribute set to NULL.

Examples

```
# Create sample data with an ID column
df <- data.frame(
gene_id = paste0("GENE", 1:5),
expression = rnorm(5),
p_value = runif(5)
)

# Convert gene_id column to row names
df_with_rownames <- Col2Rownames(df, var = "gene_id")
print(rownames(df_with_rownames))

# Create sample data with row names
df <- data.frame(
expression = rnorm(5),
p_value = runif(5)
)
rownames(df) <- paste0("GENE", 1:5)

# Convert row names to explicit column
df_with_col <- Rownames2Col(df, var = "gene_id")
print(df_with_col$gene_id)
```

SetupPyEnv *Create or Use Python Environment with Required Packages*

Description

Sets up a Python environment with specified packages. This function can create new environments or reuse existing ones, supporting both Conda and venv environment types. It ensures all required dependencies are properly installed and verified.

Default method for unsupported environment types. Throws an informative error with supported environment types.

Usage

```
SetupPyEnv(env_type = c("conda", "venv"), ...)

## Default S3 method:
SetupPyEnv(env_type = c("conda", "venv"), ...)
```

Arguments

`env_type`	Character string specifying the type of Python environment to create or use. One of: `"conda"`, `"venv"`.
`...`	Additional parameters passed to specific environment methods.

Details

This function provides a comprehensive solution for Python environment management in R projects, particularly for machine learning workflows requiring TensorFlow. Key features include:

- **Environment Creation**: Automatically creates new environments or reuses existing ones with the same name
- **Package Management**: Installs specified Python packages with version pinning support
- **Verification**: Validates environment setup and package installations
- **Flexible Methods**: Supports different backend methods for environment creation (reticulate vs system calls)

The function uses S3 method dispatch to handle different environment types, allowing for extensible support of additional environment managers in the future.

Value

A data frame containing verification results for the environment setup, including installation status of all required packages. Invisibly returns the verification results.

See Also

`reticulate::conda_create()`, `reticulate::virtualenv_create()` for underlying environment creation functions.

Examples

```
## Not run:
# Setup a Conda environment with default parameters
SetupPyEnv("conda")

# Setup a venv environment
SetupPyEnv("venv")

## End(Not run)
```

SetupPyEnv.conda *Setup Conda Python Environment*

Description

Creates and configures a Conda environment specifically designed for screening workflows. This function provides multiple methods for environment creation and package installation, including support for environment files, with comprehensive verification and error handling.

Usage

```
## S3 method for class 'conda'
SetupPyEnv(
  env_type = "conda",
  env_name = "r-reticulate-degas",
  method = c("reticulate", "system", "environment"),
  env_file = NULL,
  python_version = "3.9.15",
  packages = c(tensorflow = "2.4.1", protobuf = "3.20.3"),
  recreate = FALSE,
  use_conda_forge = TRUE,
  ...
)
```

Arguments

env_type	Character string specifying the environment type. For this method, must be "conda".
env_name	Character string specifying the Conda environment name. Default: "r-reticulate-degas".
method	Character string specifying the method for environment creation and package installation. One of: "reticulate" (uses reticulate package), "system" (uses system conda commands), or "environment" (uses YAML environment file). Default: "reticulate".
env_file	Character string specifying the path to a Conda environment YAML file. Used when method = "environment". Default: NULL

`python_version` Character string specifying the Python version to install. Default: "3.9.15".

`packages` Named character vector of Python packages to install. Package names as names, versions as values. Use "any" for version to install latest available. Default includes tensorflow, protobuf, and numpy.

`recreate` Logical indicating whether to force recreation of the environment if it already exists. Default: FALSE.

`use_conda_forge` Logical indicating whether to use the conda-forge channel for package installation. Default: TRUE.

`...` Additional arguments. Currently supports:

- `verbose`: Logical indicating whether to print progress messages. Defaults to `TRUE`.
- `timeout`: Numeric specifying the timeout in seconds for package installation. Defaults to `180L`.

Value

Invisibly returns NULL.

Note

The function requires Conda to be installed and accessible on the system PATH or through reticulate. For method = "environment", the specified YAML file must exist and be properly formatted. The function includes extensive error handling but may fail if Conda is not properly configured.

See Also

`reticulate::conda_create()`, `reticulate::py_install()` for the underlying functions used in reticulate method.

Examples

```
## Not run:
# Setup using reticulate method (default)
SetupPyEnv.conda(
  env_name = "my-degas-env",
  python_version = "3.9.15"
)

# Setup using environment file
SetupPyEnv.conda(
  method = "environment",
  env_file = "path/to/environment.yml"
)

# Setup with custom packages
SetupPyEnv.conda(
  packages = c(
    "tensorflow" = "2.4.1",
```

```
    "scikit-learn" = "1.0.2",
    "pandas" = "any"
  )
)

## End(Not run)
```

SetupPyEnv.venv *Setup Virtual Environment (venv)*

Description

Creates and configures a Python virtual environment (venv) specifically designed for screening workflows. This function provides a lightweight, isolated Python environment alternative to Conda environments with similar package management capabilities.

Usage

```
## S3 method for class 'venv'
SetupPyEnv(
  env_type = "venv",
  env_name = "r-reticulate-degas",
  python_version = "3.9.15",
  packages = c(tensorflow = "2.4.1", protobuf = "3.20.3"),
  python_path = NULL,
  recreate = FALSE,
  ...
)
```

Arguments

env_type	Character string specifying the environment type. For this method, must be "venv".
env_name	Character string specifying the virtual environment name. Default: "r-reticulate-degas".
python_version	Character string specifying the Python version to use. Default: "3.9.15".
packages	Named character vector of Python packages to install. Package names as names, versions as values. Use "any" for version to install latest available. Default includes tensorflow, protobuf, and numpy.
python_path	Character string specifying the path to a specific Python executable. If NULL, uses the system default or installs the specified version. Default: NULL.
recreate	Logical indicating whether to force recreation of the virtual environment if it already exists. Default: FALSE.
...	Additional arguments. Currently supports: • verbose: Logical indicating whether to print progress messages. Defaults to TRUE.

Value

Invisibly returns NULL.

Note

Virtual environments require a base Python installation. If the specified Python version is not available, the function will attempt to install it using reticulate. Virtual environments are generally faster to create than Conda environments but may have more limited package availability compared to Conda-forge.

See Also

`reticulate::virtualenv_create()`, `reticulate::virtualenv_remove()`, `reticulate::use_virtualenv()` for the underlying virtual environment management functions.

Examples

```
## Not run:
# Setup virtual environment with default parameters
SetupPyEnv.venv()

# Setup with custom Python version and packages
SetupPyEnv.venv(
  env_name = "my-degas-venv",
  python_version = "3.8.12",
  packages = c(
    "tensorflow" = "2.4.1",
    "scikit-learn" = "1.0.2",
    "pandas" = "any"
  )
)

# Force recreate existing environment
SetupPyEnv.venv(
  env_name = "existing-env",
  recreate = TRUE
)

## End(Not run)
```

`SigBridgeR_Function_Setting`

Configuration Functions for SigBridgeR Package

Description

These functions provide a centralized configuration system for the SigBridgeR package, allowing users to set and retrieve package-specific options with automatic naming conventions.

setFunOption sets one or more configuration options for the SigBridgeR package. Options are automatically prefixed with "SigBridgeR." if not already present, ensuring proper namespace isolation.

getFuncOption retrieves configuration options for the SigBridgeR package. The function automatically handles the "SigBridgeR." prefix, allowing users to reference options with or without the explicit prefix.

checkFuncOption validates configuration options for the SigBridgeR package. This function ensures that all options meet type and value requirements before they are set in the global options.

Usage

```
setFuncOption(...)

getFuncOption(option = NULL, default = NULL)

checkFuncOption(option, value, call = rlang::caller_env())
```

Arguments

`...`	Named arguments representing option-value pairs. Options can be provided with or without the "SigBridgeR." prefix. If the prefix is missing, it will be automatically added.
`option`	Character string specifying the option name to check
`default`	The default value to return if the option is not set. Defaults to NULL.
`value`	The proposed value to assign to the option
`call`	The execution environment of a currently running function

Details

This function performs the following validations:

verbose Must be single logical values (TRUE/FALSE)

timeout, seed Must be single integer values

The function is automatically called by setFuncOption to ensure all configuration options are valid before they are set.

Value

Invisibly returns NULL. The function is called for its side effects of setting package options.

The value of the specified option if it exists, otherwise the provided default value.

No return value. The function throws an error if the value doesn't meet the required specifications for the given option.

Available Opions

- verbose: A logical value indicating whether to print verbose messages, defaults to TRUE
- timeout: An integer specifying the timeout in seconds for parallel processing, defaults to ‘180L“
- seed: An integer specifying the random seed for reproducible results, defaults to ‘123L“

Examples

```
# Set options with automatic prefixing
setFuncOption(verbose = TRUE)

# Retrieve options with automatic prefixing
getFuncOption("verbose")
```

TimeStamp *Generate Timestamp String*

Description

Creates a formatted character string representing the current system time. The format is "YYYY/mm/DD HH:MM:SS" (year/month/day hour:minute:second).

Usage

```
TimeStamp()
```

Value

Character string with current time in "YYYY/mm/DD HH:MM:SS" format. If system time is unavailable, returns a fixed timestamp.

See Also

Other TimeStamp: AddTimeStamp2cli(), CreateTimeStampCliEnv()

Examples

```
## Not run:
# Current time as formatted string
TimeStamp()
# Returns something like: "2025/06/15 16:04:00"

## End(Not run)
```

Index