Package 'mintyr'

May 8, 2026

Title Streamlined Data Processing Tools for Genomic Selection

Version 0.1.2

Description A toolkit for genomic selection in animal breeding with emphasis on multi-breed and multi-trait nested grouping operations. Streamlines iterative analysis workflows when working with 'ASReml-R' package. Includes utility functions for phenotypic data processing commonly used by animal breeders.

License MIT + file LICENSE

URL https://tony2015116.github.io/mintyr/, https://github.com/tony2015116/mintyr

BugReports https://github.com/tony2015116/mintyr/issues

Depends R (>= 4.1.0)

Imports arrow, data.table, dplyr, purrr, readxl, rlang, rsample, rstatix, stats, tibble, utils

Suggests knitr, rmarkdown, testthat, tidyr, tools

VignetteBuilder knitr

Config/fusen/version 0.6.0

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Guo Meng [aut, cre], Guo Meng [cph]

Maintainer Guo Meng <tony2015116@163.com>

Repository CRAN

Date/Publication 2025-10-25 07:00:02 UTC

Contents

`c2p_nest` *Column to Pair Nested Transformation*

Description

A sophisticated data transformation tool for generating column pair combinations and creating nested data structures with advanced configuration options.

Usage

```
c2p_nest(data, cols2bind, by = NULL, pairs_n = 2, sep = "-", nest_type = "dt")
```

Arguments

`data`	Input `data frame` or `data table` • Must contain valid columns for transformation • Supports multiple data types
`cols2bind`	Column specification for pair generation • Can be a `character` vector of column names • Can be a `numeric` vector of column indices • Must reference existing columns in the dataset
`by`	Optional grouping specification • Can be a `character` vector of column names • Can be a `numeric` vector of column indices

- Enables hierarchical nested transformations
- Supports multi-level aggregation
- Default is NULL

`pairs_n` `numeric` indicating combination size

- Minimum value: 2
- Maximum value: Length of `cols2bind`
- Controls column pair complexity
- Default is 2

`sep` `character` separator for pair naming

- Used in generating combination identifiers
- Must be a single character
- Default is "-"

`nest_type` Output nesting format

- `"dt"`: Returns nested `data table` (default)
- `"df"`: Returns nested `data frame`

Details

Advanced Transformation Mechanism:

1. Input validation and preprocessing
2. Dynamic column combination generation
3. Flexible pair transformation
4. Nested data structure creation

Transformation Process:

- Validate input parameters and column specifications
- Convert numeric indices to column names if necessary
- Generate column combinations
- Create subset data tables
- Merge and nest transformed data

Column Specification:

- Supports both column names and numeric indices
- Numeric indices must be within valid range (1 to ncol)
- Column names must exist in the dataset
- Flexible specification for both cols2bind and by parameters

Value

`data table` containing nested transformation results

- Includes `pairs` column identifying column combinations
- Contains `data` column storing nested data structures
- Supports optional grouping variables

Note

Key Operation Constraints:

- Requires non-empty input data
- Column specifications must be valid (either names or indices)
- Supports flexible combination strategies
- Computational complexity increases with combination size

See Also

- `utils::combn()` Combination generation

Examples

```
# Example data preparation: Define column names for combination
col_names <- c("Sepal.Length", "Sepal.Width", "Petal.Length")

# Example 1: Basic column-to-pairs nesting with custom separator
c2p_nest(
  iris,                   # Input iris dataset
  cols2bind = col_names,  # Columns to be combined as pairs
  pairs_n = 2,            # Create pairs of 2 columns
  sep = "&"               # Custom separator for pair names
)
# Returns a nested data.table where:
# - pairs: combined column names (e.g., "Sepal.Length&Sepal.Width")
# - data: list column containing data.tables with value1, value2 columns

# Example 2: Column-to-pairs nesting with numeric indices and grouping
c2p_nest(
  iris,                   # Input iris dataset
  cols2bind = 1:3,        # First 3 columns to be combined
  pairs_n = 2,            # Create pairs of 2 columns
  by = 5                  # Group by 5th column (Species)
)
# Returns a nested data.table where:
# - pairs: combined column names
# - Species: grouping variable
# - data: list column containing data.tables grouped by Species
```

`convert_nest` *Convert Nested Columns Between* `data.frame` *and* `data.table`

Description

The `convert_nest` function transforms a `data.frame` or `data.table` by converting nested columns to either `data.frame` or `data.table` format while preserving the original data structure. Nested columns are automatically detected based on list column identification.

Usage

```
convert_nest(data, to = c("df", "dt"))
```

Arguments

`data`	A `data.frame` or `data.table` containing nested columns
`to`	A `character` string specifying the target format. Options are `"df"` (data frame) or `"dt"` (data table). Defaults to `"df"`.

Details

Advanced Nested Column Conversion Features:

- Intelligent automatic detection of all nested (list) columns
- Comprehensive conversion of entire data structure
- Non-destructive transformation with data copying
- Seamless handling of mixed nested structures

Automatic Detection and Validation:

- Automatically identifies all list columns in the dataset
- Issues warning if no nested columns are detected
- Returns original data unchanged when no list columns exist
- Ensures data integrity through comprehensive checks

Conversion Strategies:

1. Nested column identification based on `is.list()` detection
2. Preservation of original data integrity through copying
3. Flexible handling of mixed data structures
4. Consistent type conversion across all nested elements

Nested Column Handling:

- Automatically processes all `list` columns
- Handles `data.table`, `data.frame`, and generic `list` inputs
- Maintains original column structure and order
- Prevents in-place modification of source data

Value

A transformed `data.frame` or `data.table` with all nested columns converted to the specified format. If no nested columns are found, returns the original data with a warning.

Note

Conversion Characteristics:

- Non-destructive transformation of all nested columns
- Automatic detection eliminates need for manual column specification
- Supports flexible input and output formats
- Minimal performance overhead

Warning Conditions:

- Issues warning if no list columns are found in the input data
- Returns original data unchanged when no conversion is needed
- Provides clear messages for troubleshooting

Examples

```
# Example 1: Create nested data structures
# Create single nested column
df_nest1 <- iris |>
  dplyr::group_nest(Species)     # Group and nest by Species

# Create multiple nested columns
df_nest2 <- iris |>
  dplyr::group_nest(Species) |>  # Group and nest by Species
  dplyr::mutate(
    data2 = purrr::map(          # Create second nested column
      data,
      dplyr::mutate,
      c = 2
    )
  )

# Example 2: Convert nested structures
# Convert data frame to data table
convert_nest(
  df_nest1,                      # Input nested data frame
  to = "dt"                      # Convert to data.table
)

# Example 3: Convert data table to data frame
dt_nest <- mintyr::w2l_nest(
  data = iris,                   # Input dataset
  cols2l = 1:2                   # Columns to nest
)
convert_nest(
  dt_nest,                       # Input nested data table
  to = "df"                      # Convert to data frame
)
```

`export_list` *Export List with Advanced Directory Management*

Description

The `export_list` function exports a list of `data.frame`, `data.table`, or compatible data structures with sophisticated directory handling, flexible naming, and multiple file format support.

Usage

```
export_list(split_dt, export_path = tempdir(), file_type = "txt")
```

Arguments

`split_dt`	A `list` of `data.frame`, `data.table`, or compatible data structures to be exported.
`export_path`	Base directory path for file export. Defaults to a temporary directory created by `tempdir()`.
`file_type`	File export format, either `"txt"` (tab-separated) or `"csv"`. Defaults to `"txt"`.

Details

Comprehensive List Export Features:

- Advanced nested directory structure support based on list element names
- Intelligent handling of unnamed list elements
- Automatic conversion to `data.table` for consistent export
- Hierarchical directory creation with nested path names
- Multi-format file export with intelligent separator selection
- Robust error handling and input validation

File Export Capabilities:

- Supports `"txt"` (tab-separated) and `"csv"` formats
- Intelligent file naming based on list element names
- Handles complex nested directory structures
- Efficient file writing using `data.table::fwrite()`

Value

An `integer` representing the total number of files exported successfully.

Note

Key Capabilities:

- Flexible list naming and directory management
- Comprehensive support for `data.frame` and `data.table` inputs
- Intelligent default naming for unnamed elements
- High-performance file writing mechanism

Examples

```
# Example: Export split data to files

# Step 1: Create split data structure
dt_split <- w2l_split(
  data = iris,              # Input iris dataset
  cols2l = 1:2,             # Columns to be split
  by = "Species"            # Grouping variable
)

# Step 2: Export split data to files
export_list(
  split_dt = dt_split       # Input list of data.tables
)
# Returns the number of files created
# Files are saved in tempdir() with .txt extension

# Check exported files
list.files(
  path = tempdir(),         # Default export directory
  pattern = "txt",          # File type pattern to search
  recursive = TRUE          # Search in subdirectories
)

# Clean up exported files
files <- list.files(
  path = tempdir(),         # Default export directory
  pattern = "txt",          # File type pattern to search
  recursive = TRUE,         # Search in subdirectories
  full.names = TRUE         # Return full file paths
)
file.remove(files)          # Remove all exported files
```

`export_nest` *Export Nested Data Structures with Hierarchical Organization*

Description

Intelligently exports nested data from `data.frame` or `data.table` objects with sophisticated grouping capabilities and flexible handling of multiple nested column types. This function distinguishes between exportable data.frame/data.table columns and non-exportable custom object list columns (such as rsample cross-validation splits), processing only the appropriate types by default.

Usage

```
export_nest(
  nest_dt,
  group_cols = NULL,
  nest_cols = NULL,
  export_path = tempdir(),
  file_type = "txt"
)
```

Arguments

`nest_dt`	A `data.frame` or `data.table` containing one or more nested columns. Exportable nested columns contain `data.frame` or `data.table` objects. Non-exportable columns contain custom objects such as `rsplit` from the rsample package or other list-based structures. The input cannot be empty.
`group_cols`	Optional character vector specifying column names to use for hierarchical grouping. These columns determine the directory structure for exported files. If `NULL` (default), automatically uses all non-nested columns as grouping variables. Missing or invalid columns are handled gracefully with informative warnings.
`nest_cols`	Optional character vector specifying which nested columns to export. If `NULL` (default), automatically processes only columns containing `data.frame` or `data.table` objects. Custom object list columns (e.g., `rsplit`, `vfold_split` from rsample) are identified and reported but NOT exported. Specifying non-data.frame columns in `nest_cols` triggers a warning and those columns are skipped.
`export_path`	Character string specifying the base directory for file export. Defaults to `tempdir()`. The function creates this directory recursively if it does not exist.
`file_type`	Character string indicating export format: `"txt"` for tab-separated values or `"csv"` for comma-separated values. Defaults to `"txt"`. Case-insensitive.

Details

Nested Column Type Detection: The function automatically detects and categorizes nested columns into two types:

- **Exportable columns (Data.frame/data.table)**: Columns containing `data.frame` or `data.table` objects. These are the only columns exported to files by default.
- **Non-exportable columns (Custom objects)**: Columns containing other list-type objects such as `rsplit` (rsample cross-validation splits), `vfold_split`, empty lists, or other custom S3/S4 objects. These columns are identified and reported but cannot be exported as txt/csv files.

Grouping Strategy:

- When `group_cols = NULL`, all non-nested columns automatically become grouping variables.
- Grouping columns create a hierarchical directory structure where each unique combination of group values generates a separate subdirectory.
- Files are organized as: `export_path/group1_value/group2_value/nest_col.ext`
- If no valid group columns exist, files export to the root `export_path`.

File Organization:

- One file is generated per exportable nested column per row (e.g., row 1 with 2 data.frame columns generates 2 files).
- Only data.frame/data.table nested columns are written; custom object columns are skipped.
- Filenames follow the pattern: `{nested_column_name}.{file_type}` (e.g., `data.txt`, `results.csv`).
- Files are written using `data.table::fwrite()` for efficient I/O.
- Empty or `NULL` nested data are silently skipped without interrupting the export process.

Error Handling:

- Parameter validation occurs early, with informative error messages for invalid inputs.
- Missing group columns trigger warnings but do not halt execution.
- Custom object columns are identified and reported when `nest_cols = NULL`, allowing users to be aware of non-exportable data.
- Invalid or non-data.frame nested columns in `nest_cols` are skipped with warnings.
- Individual row export failures generate warnings but continue processing remaining rows.

Data.table Requirement: The data.table package is required. The function automatically checks for its availability and converts input data to `data.table` format if necessary.

Value

An invisible `integer` representing the total number of files successfully exported. Returns `0` if no exportable data.frame/data.table columns are found or if all nested data are empty/NULL.

Dependencies

Requires the `data.table` package for efficient data manipulation and I/O operations.

Limitations

Custom object columns (e.g., `rsplit` from rsample, cross-validation folds) cannot be exported as txt/csv files because they are not standard data structures. These columns are identified automatically and reported to the console. If you need to export rsample split information, consider extracting the indices or data using rsample utility functions first.

Use Cases

- Exporting structured data from tidymodels workflows that also contain cross-validation splits
- Batch exporting multiple nested data.frame columns with automatic hierarchical organization
- Creating organized file hierarchies based on grouping variables (e.g., by experiment, participant, or time period)
- Integration with reproducible research workflows

Note

- The function does not modify the input `nest_dt`; it is non-destructive.
- Empty input data.frames trigger an error; use `if (nrow(nest_dt) > 0)` to validate input first.
- Custom object columns detected when `nest_cols = NULL` are reported as informational messages; no error occurs.
- Attempting to export custom object columns via `nest_cols` will skip them with a warning.
- All messages and warnings are printed to console; capture output programmatically if needed via `capture.output()` or similar functions.
- File paths are constructed using `file.path()`, ensuring cross-platform compatibility.

See Also

`fwrite` for details on file writing,

Examples

```
# Example 1: Basic nested data export workflow
# Step 1: Create nested data structure
dt_nest <- w2l_nest(
  data = iris,              # Input iris dataset
  cols2l = 1:2,             # Columns to be nested
  by = "Species"            # Grouping variable
)

# Step 2: Export nested data to files
export_nest(
  nest_dt = dt_nest,        # Input nested data.table
  nest_cols = "data",       # Column containing nested data
  group_cols = c("name", "Species")  # Columns to create directory structure
)
# Returns the number of files created
# Creates directory structure: tempdir()/name/Species/data.txt

# Check exported files
list.files(
  path = tempdir(),         # Default export directory
  pattern = "txt",          # File type pattern to search
  recursive = TRUE          # Search in subdirectories
)
# Returns list of created files and their paths

# Clean up exported files
files <- list.files(
  path = tempdir(),         # Default export directory
  pattern = "txt",          # File type pattern to search
  recursive = TRUE,         # Search in subdirectories
  full.names = TRUE         # Return full file paths
)
file.remove(files)          # Remove all exported files
```

format_digits *Format Numeric Columns with Specified Digits*

Description

The format_digits function formats numeric columns in a data frame or data table by rounding numbers to a specified number of decimal places and converting them to character strings. It can optionally format the numbers as percentages.

Usage

```
format_digits(data, cols = NULL, digits = 2, percentage = FALSE)
```

Arguments

data	A data.frame or data.table. The input data containing numeric columns to format.
cols	An optional numeric or character vector specifying the columns to format. If NULL (default), all numeric columns are formatted.
digits	A non-negative integer specifying the number of decimal places to use. Defaults to 2.
percentage	A logical value indicating whether to format the numbers as percentages. If TRUE, the numbers are multiplied by 100 and a percent sign (%) is appended. Defaults to FALSE.

Details

The function performs the following steps:

1. Validates the input parameters, ensuring that data is a data.frame or data.table, cols (if provided) are valid column names or indices, and digits is a non-negative integer.
2. Converts data to a data.table if it is not already one.
3. Creates a formatting function based on the digits and percentage parameters:
   - If percentage = FALSE, numbers are rounded to digits decimal places.
   - If percentage = TRUE, numbers are multiplied by 100, rounded to digits decimal places, and a percent sign (%) is appended.
4. Applies the formatting function to the specified columns:
   - If cols is NULL, the function formats all numeric columns in data.
   - If cols is specified, only those columns are formatted.
5. Returns a new data.table with the formatted columns.

Value

A data.table with the specified numeric columns formatted as character strings with the specified number of decimal places. If percentage = TRUE, the numbers are shown as percentages.

Note

- The input `data` must be a `data.frame` or `data.table`.
- If `cols` is specified, it must be a vector of valid column names or indices present in `data`.
- The `digits` parameter must be a single non-negative integer.
- The original `data` is not modified; a modified copy is returned.

Examples

```
# Example: Number formatting demonstrations

# Setup test data
dt <- data.table::data.table(
  a = c(0.1234, 0.5678),      # Numeric column 1
  b = c(0.2345, 0.6789),      # Numeric column 2
  c = c("text1", "text2")     # Text column
)

# Example 1: Format all numeric columns
format_digits(
  dt,                         # Input data table
  digits = 2                  # Round to 2 decimal places
)

# Example 2: Format specific column as percentage
format_digits(
  dt,                         # Input data table
  cols = c("a"),              # Only format column 'a'
  digits = 2,                 # Round to 2 decimal places
  percentage = TRUE           # Convert to percentage
)
```

---

get_filename *Extract Filenames from File Paths*

Description

The `get_filename` function extracts filenames from file paths with options to remove file extensions and/or directory paths.

Usage

```
get_filename(paths, rm_extension = TRUE, rm_path = TRUE)
```

Arguments

paths	A character vector containing file system paths. Must be valid and accessible path strings.
rm_extension	A logical flag controlling file extension removal: • TRUE: Strips file extensions from filenames • FALSE: Preserves complete filename with extension Default is TRUE.
rm_path	A logical flag managing directory path handling: • TRUE: Extracts only the filename, discarding directory information • FALSE: Retains complete path information Default is TRUE.

Details

The function performs the following operations:

- Validates input paths
- Handles empty input vectors
- Optionally removes directory paths using basename
- Optionally removes file extensions using regex substitution

Value

A character vector of processed filenames with applied transformations.

Note

- If both rm_extension and rm_path are FALSE, a warning is issued and the original paths are returned
- Supports multiple file paths in the input vector

See Also

- base::basename() for basic filename extraction

Examples

```
# Example: File path processing demonstrations

# Setup test files
xlsx_files <- mintyr_example(
  mintyr_examples("xlsx_test")    # Get example Excel files
)

# Example 1: Extract filenames without extensions
get_filename(
  xlsx_files,                     # Input file paths
  rm_extension = TRUE,            # Remove file extensions
  rm_path = TRUE                  # Remove directory paths
)
```

```
# Example 2: Keep file extensions
get_filename(
  xlsx_files,                    # Input file paths
  rm_extension = FALSE,          # Keep file extensions
  rm_path = TRUE                 # Remove directory paths
)

# Example 3: Keep full paths without extensions
get_filename(
  xlsx_files,                    # Input file paths
  rm_extension = TRUE,           # Remove file extensions
  rm_path = FALSE                # Keep directory paths
)
```

`get_path_segment` *Extract Specific Segments from File Paths*

Description

The `get_path_segment` function extracts specific segments from file paths provided as character strings. Segments can be extracted from either the beginning or the end of the path, depending on the value of `n`.

Usage

```
get_path_segment(paths, n = 1)
```

Arguments

`paths` A 'character vector' containing file system paths

- Must be non-empty
- Path segments separated by forward slash `'/'`
- Supports absolute and relative paths
- Handles cross-platform path representations
- Supports paths with mixed separators (`'\\'` and `'/'`)

`n` Numeric index for segment selection

- Positive values: Select from path start
- Negative values: Select from path end
- Supports single index or range extraction
- Cannot be `0`
- Default is `1` (first segment)

Details

Sophisticated Path Segment Extraction Mechanism:

1. Comprehensive input validation
2. Path normalization and preprocessing
3. Robust cross-platform path segmentation
4. Flexible indexing with forward and backward navigation
5. Intelligent segment retrieval
6. Graceful handling of edge cases

Indexing Behavior:

- Positive `n`: Forward indexing from path start - `n = 1`: First segment - `n = 2`: Second segment
- Negative `n`: Reverse indexing from path end - `n = -1`: Last segment - `n = -2`: Second-to-last segment
- Range extraction: Supports `c(start, end)` index specification

Path Parsing Characteristics:

- Standardizes path separators to `'/'`
- Removes drive letters (e.g., `'C:'`)
- Ignores consecutive `'/'` delimiters
- Removes leading and trailing separators
- Returns `NA_character_` for non-existent segments
- Supports complex path structures

Value

'character vector' with extracted path segments

- Matching segments for valid indices
- `NA_character_` for segments beyond path length

Note

Critical Operational Constraints:

- Requires non-empty 'paths' input
- `n` must be non-zero numeric value
- Supports cross-platform path representations
- Minimal computational overhead
- Preserves path segment order

See Also

- `tools::file_path_sans_ext()` File extension manipulation

Examples

```
# Example: Path segment extraction demonstrations

# Setup test paths
paths <- c(
  "C:/home/user/documents",   # Windows style path
  "/var/log/system",          # Unix system path
  "/usr/local/bin"            # Unix binary path
)

# Example 1: Extract first segment
get_path_segment(
  paths,                      # Input paths
  1                           # Get first segment
)
# Returns: c("home", "var", "usr")

# Example 2: Extract second-to-last segment
get_path_segment(
  paths,                      # Input paths
  -2                          # Get second-to-last segment
)
# Returns: c("user", "log", "local")

# Example 3: Extract from first to last segment
get_path_segment(
  paths,                      # Input paths
  c(1,-1)                     # Range from first to last
)
# Returns full paths without drive letters

# Example 4: Extract first three segments
get_path_segment(
  paths,                      # Input paths
  c(1,3)                      # Range from first to third
)
# Returns: c("home/user/documents", "var/log/system", "usr/local/bin")

# Example 5: Extract last two segments (reverse order)
get_path_segment(
  paths,                      # Input paths
  c(-1,-2)                    # Range from last to second-to-last
)
# Returns: c("documents/user", "system/log", "bin/local")

# Example 6: Extract first two segments
get_path_segment(
  paths,                      # Input paths
  c(1,2)                      # Range from first to second
)
# Returns: c("home/user", "var/log", "usr/local")
```

import_csv *Flexible* CSV/TXT *File Import with Multiple Backend Support*

Description

A comprehensive CSV or TXT file import function offering advanced reading capabilities through data.table and arrow packages with intelligent data combination strategies.

Usage

```
import_csv(
  file,
  package = "data.table",
  rbind = TRUE,
  rbind_label = "_file",
  full_path = FALSE,
  keep_ext = FALSE,
  ...
)
```

Arguments

file	A character vector of file paths to CSV files. Must point to existing and accessible files.
package	A character string specifying the backend package: • "data.table": Uses data.table::fread() (default) • "arrow": Uses arrow::read_csv_arrow() Determines the underlying reading mechanism.
rbind	A logical value controlling data combination strategy: • TRUE: Combines all files into a single data object (default) • FALSE: Returns a list of individual data objects
rbind_label	A character string or NULL for source file tracking: • character: Specifies the column name for file source labeling (default: "_file") • NULL: Disables source file tracking
full_path	A logical value controlling path display in file labels: • TRUE: Uses full file path • FALSE: Uses only filename (default)
keep_ext	A logical value controlling file extension in labels: • TRUE: Retains file extension (e.g., .csv) • FALSE: Removes file extension (default)
...	Additional arguments passed to backend-specific reading functions (e.g., col_types, na.strings, skip).

Details

The function provides a unified interface for reading CSV files using either data.table or arrow package. When reading multiple files, it can either combine them into a single data object or return them as a list. File source tracking is supported through the `rbind_label` parameter.

File labeling behavior is controlled by `full_path` and `keep_ext` parameters:

- `full_path = FALSE, keep_ext = FALSE`: Filename without extension (e.g., `"data"`)
- `full_path = FALSE, keep_ext = TRUE`: Filename with extension (e.g., `"data.csv"`)
- `full_path = TRUE, keep_ext = FALSE`: Full path without extension (e.g., `"/path/to/data"`)
- `full_path = TRUE, keep_ext = TRUE`: Full path with extension (e.g., `"/path/to/data.csv"`)

Value

Depends on the `rbind` parameter:

- If `rbind = TRUE`: A single data object (from chosen package) containing all imported data, with source file information in `rbind_label` column
- If `rbind = FALSE`: A named list of data objects with names derived from input file paths based on `full_path` and `keep_ext` settings

Note

Critical Import Considerations:

- Requires all specified files to be accessible `CSV/TXT` files
- Supports flexible backend selection via `package` parameter
- `rbind = TRUE` assumes compatible data structures across files
- Missing columns are automatically aligned when combining data
- File labeling is customizable through `full_path` and `keep_ext` parameters

See Also

- `data.table::fread()` for `data.table` backend
- `arrow::read_csv_arrow()` for `arrow` backend
- `data.table::rbindlist()` for data combination

Examples

```
# Example: CSV file import demonstrations

# Setup test files
csv_files <- mintyr_example(
  mintyr_examples("csv_test")     # Get example CSV files
)

# Example 1: Import and combine CSV files using data.table
import_csv(
  csv_files,                      # Input CSV file paths
```

```
  package = "data.table",         # Use data.table for reading
  rbind = TRUE,                   # Combine all files into one data.table
  rbind_label = "_file",          # Column name for file source
  keep_ext = TRUE,                # Include .csv extension in _file column
  full_path = TRUE                # Show complete file paths in _file column
)

# Example 2: Import files separately using arrow
import_csv(
  csv_files,                      # Input CSV file paths
  package = "arrow",              # Use arrow for reading
  rbind = FALSE                   # Keep files as separate data.tables
)
```

`import_xlsx` *Import Data from* `XLSX` *Files with Advanced Handling*

Description

A robust and flexible function for importing data from one or multiple `XLSX` files, offering comprehensive options for sheet selection, data combination, and source tracking.

Usage

```
import_xlsx(file, rbind = TRUE, sheet = NULL, ...)
```

Arguments

`file`	A `character` vector of file paths to `Excel` files. Must point to existing `.xlsx` or `.xls` files.
`rbind`	A `logical` value controlling data combination strategy: • `TRUE`: Combines all data into a single `data.table` • `FALSE`: Returns a list of `data.tables` Default is `TRUE`.
`sheet`	A `numeric` vector or `NULL` specifying sheet import strategy: • `NULL` (default): Imports all sheets • `numeric`: Imports only specified sheet indices
`...`	Additional arguments passed to `readxl::read_excel()`, such as `col_types`, `skip`, or `na`.

Details

The function provides a comprehensive solution for importing Excel data with the following features:

- Supports multiple files and sheets
- Automatic source tracking for files and sheets
- Flexible combining options
- Handles missing columns across sheets when combining
- Preserves original data types through readxl

Value

Depends on the rbind parameter:

- If rbind = TRUE: A single data.table with additional tracking columns: - excel_name: Source file name (without extension) - sheet_name: Source sheet name
- If rbind = FALSE: A named list of data.tables with format "filename_sheetname"

Note

Critical Import Considerations:

- Requires all specified files to be accessible Excel files
- Sheet indices must be valid across input files
- rbind = TRUE assumes compatible data structures
- Missing columns are automatically filled with NA
- File extensions are automatically removed in tracking columns

See Also

- readxl::read_excel() for underlying Excel reading
- data.table::rbindlist() for data combination

Examples

```
# Example: Excel file import demonstrations

# Setup test files
xlsx_files <- mintyr_example(
  mintyr_examples("xlsx_test")    # Get example Excel files
)

# Example 1: Import and combine all sheets from all files
import_xlsx(
  xlsx_files,                     # Input Excel file paths
  rbind = TRUE                    # Combine all sheets into one data.table
)

# Example 2: Import specific sheets separately
import_xlsx(
  xlsx_files,                     # Input Excel file paths
  rbind = FALSE,                  # Keep sheets as separate data.tables
  sheet = 2                       # Only import first sheet
)
```

mintyr_example *Get path to mintyr examples*

Description

`mintyr` comes bundled with a number of sample files in its `inst/extdata` directory. Use `mintyr_example()` to retrieve the full file path to a specific example file.

Usage

```
mintyr_example(path = NULL)
```

Arguments

`path`	Name of the example file to locate. If NULL or missing, returns the directory path containing the examples.

Value

Character string containing the full path to the requested example file.

See Also

`mintyr_examples()` to list all available example files

Examples

```
# Get path to an example file
mintyr_example("csv_test1.csv")
```

mintyr_examples *List all available example files in mintyr package*

Description

`mintyr` comes bundled with a number of sample files in its `inst/extdata` directory. This function lists all available example files, optionally filtered by a pattern.

Usage

```
mintyr_examples(pattern = NULL)
```

Arguments

`pattern`	A regular expression to filter filenames. If `NULL` (default), all available files are returned.

Value

A character vector containing the names of example files. If no files match the pattern or if the example directory is empty, returns a zero-length character vector.

See Also

`mintyr_example()` to get the full path of a specific example file

Examples

```
# List all example files
mintyr_examples()
```

`nest_cv` *Apply Cross-Validation to Nested Data*

Description

The `nest_cv` function applies cross-validation splits to nested data frames or data tables within a data table. It uses the `rsample` package's `vfold_cv` function to create cross-validation splits for predictive modeling and analysis on nested datasets.

Usage

```
nest_cv(
  nest_dt,
  v = 10,
  repeats = 1,
  strata = NULL,
  breaks = 4,
  pool = 0.1,
  ...
)
```

Arguments

`nest_dt`	A `data.frame` or `data.table` containing at least one nested `data.frame` or `data.table` column. • Supports multi-level nested structures • Requires at least one nested data column
`v`	The number of partitions of the data set.
`repeats`	The number of times to repeat the V-fold partitioning.
`strata`	A variable in `data` (single character or name) used to conduct stratified sampling. When not `NULL`, each resample is created within the stratification variable. Numeric `strata` are binned into quartiles.

breaks	A single number giving the number of bins desired to stratify a numeric stratification variable.
pool	A proportion of data used to determine if a particular group is too small and should be pooled into another group. We do not recommend decreasing this argument below its default of 0.1 because of the dangers of stratifying groups that are too small.
...	These dots are for future extensions and must be empty.

Details

The function performs the following steps:

1. Checks if the input nest_dt is non-empty and contains at least one nested column of data.frames or data.tables.
2. Identifies the nested columns and non-nested columns within nest_dt.
3. Applies rsample::vfold_cv to each nested data frame in the specified nested column(s), creating the cross-validation splits.
4. Expands the cross-validation splits and associates them with the non-nested columns.
5. Extracts the training and validation data for each split and adds them to the output data table.

If the strata parameter is provided, stratified sampling is performed during the cross-validation. Additional arguments can be passed to rsample::vfold_cv via ....

Value

A data.table containing the cross-validation splits for each nested dataset. It includes:

- Original non-nested columns from nest_dt.
- splits: The cross-validation split objects returned by rsample::vfold_cv.
- train: The training data for each split.
- validate: The validation data for each split.

Note

- The nest_dt must contain at least one nested column of data.frames or data.tables.
- The function converts nest_dt to a data.table internally to ensure efficient data manipulation.
- The strata parameter should be a column name present in the nested data frames.
- If strata is specified, ensure that the specified column exists in all nested data frames.
- The breaks and pool parameters are used when strata is a numeric variable and control how stratification is handled.
- Additional arguments passed through ... are forwarded to rsample::vfold_cv.

See Also

- rsample::vfold_cv() Underlying cross-validation function
- rsample::training() Extract training set
- rsample::testing() Extract test set

Examples

```
# Example: Cross-validation for nested data.table demonstrations

# Setup test data
dt_nest <- w2l_nest(
  data = iris,                  # Input dataset
  cols2l = 1:2                  # Nest first 2 columns
)

# Example 1: Basic 2-fold cross-validation
nest_cv(
  nest_dt = dt_nest,            # Input nested data.table
  v = 2                         # Number of folds (2-fold CV)
)

# Example 2: Repeated 2-fold cross-validation
nest_cv(
  nest_dt = dt_nest,            # Input nested data.table
  v = 2,                        # Number of folds (2-fold CV)
  repeats = 2                   # Number of repetitions
)
```

---

r2p_nest *Row to Pair Nested Transformation*

Description

A sophisticated data transformation tool for performing row pair conversion and creating nested data structures with advanced configuration options.

Usage

```
r2p_nest(data, rows2bind, by, nest_type = "dt")
```

Arguments

data	Input data frame or data table • Must contain valid columns for transformation • Supports multiple data types
rows2bind	Row binding specification • Can be a character column name

- Can be a `numeric` column index
- Must be a single column identifier

`by` Grouping specification for nested pairing

- Can be a `character` vector of column names
- Can be a `numeric` vector of column indices
- Must specify at least one column
- Supports multi-column transformation

`nest_type` Output nesting format

- `"dt"`: Returns nested `data table` (default)
- `"df"`: Returns nested `data frame`

Details

Advanced Transformation Mechanism:

1. Input validation and preprocessing
2. Dynamic column identification
3. Flexible row pairing across specified columns
4. Nested data structure generation

Transformation Process:

- Validate input parameters and column specifications
- Convert numeric indices to column names if necessary
- Reshape data from wide to long format
- Perform column-wise nested transformation
- Generate final nested structure

Column Specification:

- Supports both column names and numeric indices
- Numeric indices must be within valid range (1 to ncol)
- Column names must exist in the dataset
- Flexible specification for both rows2bind and by parameters

Value

`data table` containing nested transformation results

- Includes `name` column identifying source columns
- Contains `data` column storing nested data structures

Note

Key Operation Constraints:

- Requires non-empty input data
- Column specifications must be valid (either names or indices)
- By parameter must specify at least one column
- Low computational overhead

See Also

- `data.table::melt()` Long format conversion
- `data.table::dcast()` Wide format conversion
- `base::rbind()` Row binding utility
- `c2p_nest()` Column to pair nested transformation

Examples

```
# Example 1: Row-to-pairs nesting with column names
r2p_nest(
  mtcars,                    # Input mtcars dataset
  rows2bind = "cyl",         # Column to be used as row values
  by = c("hp", "drat", "wt")  # Columns to be transformed into pairs
)
# Returns a nested data.table where:
# - name: variable names (hp, drat, wt)
# - data: list column containing data.tables with rows grouped by cyl values

# Example 2: Row-to-pairs nesting with numeric indices
r2p_nest(
  mtcars,                    # Input mtcars dataset
  rows2bind = 2,             # Use 2nd column (cyl) as row values
  by = 4:6                   # Use columns 4-6 (hp, drat, wt) for pairs
)
# Returns a nested data.table where:
# - name: variable names from columns 4-6
# - data: list column containing data.tables with rows grouped by cyl values
```

---

split_cv *Cross-Validation Split Generator*

Description

A robust cross-validation splitting utility for multiple datasets with advanced stratification and configuration options.

Usage

```
split_cv(
  split_dt,
  v = 10,
  repeats = 1,
  strata = NULL,
  breaks = 4,
  pool = 0.1,
  ...
)
```

Arguments

split_dt	list of input datasets • Must contain data.frame or data.table elements • Supports multiple dataset processing • Cannot be empty
v	The number of partitions of the data set.
repeats	The number of times to repeat the V-fold partitioning.
strata	A variable in data (single character or name) used to conduct stratified sampling. When not NULL, each resample is created within the stratification variable. Numeric strata are binned into quartiles.
breaks	A single number giving the number of bins desired to stratify a numeric stratification variable.
pool	A proportion of data used to determine if a particular group is too small and should be pooled into another group. We do not recommend decreasing this argument below its default of 0.1 because of the dangers of stratifying groups that are too small.
...	These dots are for future extensions and must be empty.

Details

Advanced Cross-Validation Mechanism:

1. Input dataset validation
2. Stratified or unstratified sampling
3. Flexible fold generation
4. Train-validate set creation

Sampling Strategies:

- Supports multiple dataset processing
- Handles stratified and unstratified sampling
- Generates reproducible cross-validation splits

Value

`list` of `data.table` objects containing:

- `splits`: Cross-validation split objects
- `train`: Training dataset subsets
- `validate`: Validation dataset subsets

Note

Important Constraints:

- Requires non-empty input datasets
- All datasets must be `data.frame` or `data.table`
- Strata column must exist if specified
- Computational resources impact large dataset processing

See Also

- `rsample::vfold_cv()` Core cross-validation function

Examples

```
# Prepare example data: Convert first 3 columns of iris dataset to long format and split
dt_split <- w2l_split(data = iris, cols2l = 1:3)
# dt_split is now a list containing 3 data tables for Sepal.Length, Sepal.Width, and Petal.Length

# Example 1: Single cross-validation (no repeats)
split_cv(
  split_dt = dt_split,  # Input list of split data
  v = 3,                # Set 3-fold cross-validation
  repeats = 1           # Perform cross-validation once (no repeats)
)
# Returns a list where each element contains:
# - splits: rsample split objects
# - id: fold numbers (Fold1, Fold2, Fold3)
# - train: training set data
# - validate: validation set data

# Example 2: Repeated cross-validation
split_cv(
  split_dt = dt_split,  # Input list of split data
  v = 3,                # Set 3-fold cross-validation
  repeats = 2           # Perform cross-validation twice
)
# Returns a list where each element contains:
# - splits: rsample split objects
# - id: repeat numbers (Repeat1, Repeat2)
# - id2: fold numbers (Fold1, Fold2, Fold3)
# - train: training set data
# - validate: validation set data
```

top_perc *Select Top Percentage of Data and Statistical Summarization*

Description

The top_perc function selects the top percentage of data based on a specified trait and computes summary statistics. It allows for grouping by additional columns and offers flexibility in the type of statistics calculated. The function can also retain the selected data if needed.

Usage

```
top_perc(data, perc, trait, by = NULL, type = "mean_sd", keep_data = FALSE)
```

Arguments

data	A data.frame containing the source dataset for analysis • Supports various data frame-like structures • Automatically converts non-data frame inputs
perc	Numeric vector of percentages for data selection • Range: -1 to 1 • Positive values: Select top percentiles • Negative values: Select bottom percentiles • Multiple percentiles supported
trait	Character string specifying the 'selection column' • Must be a valid column name in the input data • Used as the basis for top/bottom percentage selection
by	Optional character vector for 'grouping columns' • Default is NULL • Enables stratified analysis • Allows granular percentage selection within groups
type	Statistical summary type • Default: "mean_sd" • Controls the type of summary statistics computed • Supports various summary methods from rstatix
keep_data	Logical flag for data retention • Default: FALSE • TRUE: Return both summary statistics and selected data • FALSE: Return only summary statistics

Value

A list or data frame:

- If keep_data is FALSE, a data frame with summary statistics.
- If keep_data is TRUE, a list where each element is a list containing summary statistics (stat) and the selected top data (data).

Note

- The perc parameter accepts values between -1 and 1. Positive values select the top percentage, while negative values select the bottom percentage.
- The function performs initial checks to ensure required arguments are provided and valid.
- Grouping by additional columns (by) is optional and allows for more granular analysis.
- The type parameter specifies the type of summary statistics to compute, with "mean_sd" as the default.
- If keep_data is set to TRUE, the function will return both the summary statistics and the selected top data for each percentage.

See Also

- rstatix::get_summary_stats() Statistical summary computation
- dplyr::top_frac() Percentage-based data selection

Examples

```
# Example 1: Basic usage with single trait
# This example selects the top 10% of observations based on Petal.Width
# keep_data=TRUE returns both summary statistics and the filtered data
top_perc(iris,
         perc = 0.1,                # Select top 10%
         trait = c("Petal.Width"),  # Column to analyze
         keep_data = TRUE)          # Return both stats and filtered data

# Example 2: Using grouping with 'by' parameter
# This example performs the same analysis but separately for each Species
# Returns nested list with stats and filtered data for each group
top_perc(iris,
         perc = 0.1,                # Select top 10%
         trait = c("Petal.Width"),  # Column to analyze
         by = "Species")            # Group by Species

# Example 3: Complex example with multiple percentages and grouping variables
# Reshape data from wide to long format for Sepal.Length and Sepal.Width
iris |>
  tidyr::pivot_longer(1:2,
                      names_to = "names",
                      values_to = "values") |>
  mintyr::top_perc(
    perc = c(0.1, -0.2),
```

```
  trait = "values",
  by = c("Species", "names"),
  type = "mean_sd")
```

`w2l_nest` *Reshape Wide Data to Long Format and Nest by Specified Columns*

Description

The `w2l_nest` function reshapes wide-format data into long-format and nests it by specified columns. It handles both `data.frame` and `data.table` objects and provides options for grouping and nesting the data.

Usage

```
w2l_nest(data, cols2l = NULL, by = NULL, nest_type = "dt")
```

Arguments

`data`	`data.frame` or `data.table` • Input dataset in wide format • Automatically converted to `data.table` if necessary
`cols2l`	`numeric` or `character` columns to transform • Specifies columns for wide-to-long conversion • Can be column indices or column names • Default is `NULL`
`by`	`numeric` or `character` grouping variables • Optional columns for additional data stratification • Can be column indices or column names • Used to create hierarchical nested structures • Default is `NULL`
`nest_type`	`character` output data type • Defines nested data object type • Possible values: – `"dt"`: nested `data.table` – `"df"`: nested `data.frame` • Default is `"dt"`

Details

The function melts the specified wide columns into long format and nests the resulting data by the `name` column and any additional grouping variables specified in `by`. The nested data can be in the form of `data.table` or `data.frame` objects, controlled by the `nest_type` parameter.

Both `cols2l` and `by` parameters accept either column indices or column names, providing flexible ways to specify the columns for transformation and grouping.

Value

`data.table` with nested data in long format, grouped by specified columns if provided. Each row contains a nested `data.table` or `data.frame` under the column data, depending on nest_type.

- If `by` is `NULL`, returns a `data.table` nested by `name`.
- If `by` is specified, returns a `data.table` nested by `name` and the grouping variables.

Note

- Both `cols2l` and `by` parameters can be specified using either numeric indices or character column names.
- When using numeric indices, they must be valid column positions in the data (1 to ncol(data)).
- When using character names, all specified columns must exist in the data.
- The function converts `data.frame` to `data.table` if necessary.
- The `nest_type` parameter controls whether nested data are `data.table` (`"dt"`) or `data.frame` (`"df"`) objects.
- If `nest_type` is not `"dt"` or `"df"`, the function will stop with an error.

See Also

Related functions and packages:

- `tidytable::nest_by()` Nest data.tables by group

Examples

```
# Example: Wide to long format nesting demonstrations

# Example 1: Basic nesting by group
w2l_nest(
  data = iris,                    # Input dataset
  by = "Species"                  # Group by Species column
)

# Example 2: Nest specific columns with numeric indices
w2l_nest(
  data = iris,                    # Input dataset
  cols2l = 1:4,                   # Select first 4 columns to nest
  by = "Species"                  # Group by Species column
)

# Example 3: Nest specific columns with column names
w2l_nest(
  data = iris,                    # Input dataset
  cols2l = c("Sepal.Length",      # Select columns by name
             "Sepal.Width",
             "Petal.Length"),
  by = 5                          # Group by column index 5 (Species)
)
# Returns similar structure to Example 2
```

w2l_split *Reshape Wide Data to Long Format and Split into List*

Description

The w2l_split function reshapes wide-format data into long-format and splits it into a list by variable names and optional grouping columns. It handles both data.frame and data.table objects.

Usage

```
w2l_split(data, cols2l = NULL, by = NULL, split_type = "dt", sep = "_")
```

Arguments

data	data.frame or data.table • Input dataset in wide format • Automatically converted to data.table if necessary
cols2l	numeric or character columns to transform • Specifies columns for wide-to-long conversion • Can be column indices or column names • Default is NULL
by	numeric or character grouping variables • Optional columns for data splitting • Can be column indices or column names • Used to create hierarchical split structure • Default is NULL
split_type	character output data type • Defines split data object type • Possible values: – "dt": split data.table objects – "df": split data.frame objects • Default is "dt"
sep	character separator • Used for combining split names • Default is "_"

Details

The function melts the specified wide columns into long format and splits the resulting data into a list based on the variable names and any additional grouping variables specified in by. The split data can be in the form of data.table or data.frame objects, controlled by the split_type parameter.

Both cols2l and by parameters accept either column indices or column names, providing flexible ways to specify the columns for transformation and splitting.

Value

A list of `data.table` or `data.frame` objects (depending on `split_type`), split by variable names and optional grouping columns.

- If `by` is `NULL`, returns a list split by variable names only.
- If `by` is specified, returns a list split by both variable names and grouping variables.

Note

- Both `cols2l` and `by` parameters can be specified using either numeric indices or character column names.
- When using numeric indices, they must be valid column positions in the data (1 to ncol(data)).
- When using character names, all specified columns must exist in the data.
- The function converts `data.frame` to `data.table` if necessary.
- The `split_type` parameter controls whether split data are `data.table` (`"dt"`) or `data.frame` (`"df"`) objects.
- If `split_type` is not `"dt"` or `"df"`, the function will stop with an error.

See Also

Related functions and packages:

- `tidytable::group_split()` Split data frame by groups

Examples

```
# Example: Wide to long format splitting demonstrations

# Example 1: Basic splitting by Species
w2l_split(
  data = iris,                    # Input dataset
  by = "Species"                  # Split by Species column
) |>
  lapply(head)                    # Show first 6 rows of each split

# Example 2: Split specific columns using numeric indices
w2l_split(
  data = iris,                    # Input dataset
  cols2l = 1:3,                   # Select first 3 columns to split
  by = 5                          # Split by column index 5 (Species)
) |>
  lapply(head)                    # Show first 6 rows of each split

# Example 3: Split specific columns using column names
list_res <- w2l_split(
  data = iris,                    # Input dataset
  cols2l = c("Sepal.Length",      # Select columns by name
             "Sepal.Width"),
  by = "Species"                  # Split by Species column
```

```
)
lapply(list_res, head)            # Show first 6 rows of each split
# Returns similar structure to Example 2
```

Index