Package 'mintyr'

June 20, 2026

Title High-Performance Phenotypic Data Pipelines for Breeding

Version 0.1.3

Description A streamlined toolkit specifically designed for genomic selection and quantitative genetics in animal breeding. It provides high-performance data manipulation backed by 'data.table', focusing on multi-breed and multi-trait nested grouping operations. Features include zero-copy data importing, automated cross-validation splitting, and robust tools to generate and batch-export formatted phenotypic files required by various breeding software (e.g., 'ASReml-R', 'HIBLUP', 'DMU'), heavily optimizing iterative variance component analysis and large-scale evaluation pipelines.

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 data.table, parallel, readxl, rsample, stats, utils, writexl

Suggests knitr, rmarkdown

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 2026-06-20 07:00:02 UTC

Contents

`c2p_nest` *Column to Pair Nested Transformation*

Description

Generates combinations of specified columns and creates a nested data structure based on these pairs. Each nested subset renames the combined columns to `value1`, `value2`, ... (up to `pairs_n`) to support uniform iterative analyses such as genetic correlation estimation.

Usage

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

Arguments

`data`	A data.frame or data.table to be transformed.
`cols2bind`	A character vector of column names or a numeric vector of column indices to be combined into pairs. Must not overlap with `by`.
`by`	A character vector of column names or a numeric vector of column indices to group by. Default is `NULL`.
`pairs_n`	A positive integer >= 2 indicating the size of each column combination (e.g., 2 for pairwise). Default is 2.
`sep`	A single character string used as a separator when constructing the `pairs` identifier column. Default is `"-"`.
`nest_type`	A character string specifying the class of each nested object: `"dt"` (data.table, default) or `"df"` (data.frame).

Details

The columns specified in `cols2bind` are renamed to `value1`, `value2`, ... within each nested subset. The original column names are preserved in the `pairs` column (e.g., `"Sepal.Length-Sepal.Width"`), ensuring full traceability for downstream iterative analyses such as genetic correlation estimation.

Columns that belong to neither `cols2bind` nor `by` (referred to internally as "extra columns") are retained inside the nested subsets so that covariates or ID fields remain accessible. Grouping columns (`by`) are *not* duplicated inside the nested data because they are already present as outer key columns in the returned table.

When the number of requested combinations exceeds 500 a message is emitted; above 5000 a warning is raised, as memory usage grows linearly with the combination count.

Value

A `data.table` with columns:

pairs Character. The column-combination identifier, e.g. `"Sepal.Length-Sepal.Width"`.

... Any `by` grouping columns, one per variable.

data List-column. Each cell holds a data.table (or data.frame when `nest_type = "df"`) containing `value1`, `value2`, ..., plus any extra columns that were neither in `cols2bind` nor `by`.

See Also

`combn` for the underlying combination generator.

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

# 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
```

`export_list` *Export a List of Data Frames with Hierarchical Directory Management*

Description

Exports every element of a named (or unnamed) `list` of `data.frame` / `data.table` objects to `txt` or `csv` files. Element names may contain forward-slashes (`/`) to encode arbitrary subdirectory depth, e.g. `"group_a/subject_01/results"` writes `<export_path>/group_a/subject_01/results.txt`. Unnamed elements are automatically labelled `split_<i>`.

Usage

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

Arguments

`split_dt`	A non-empty `list` whose elements are `data.frame`, `data.table`, or any object coercible via `data.table::as.data.table()`.
`export_path`	Single character string - the root export directory. Created recursively if absent. Defaults to `tempdir()`.
`file_type`	`"txt"` (tab-separated, default) or `"csv"` (comma-separated). Case-insensitive.

Details

Performance design:

- All element names are resolved and path components split in a single vectorised pass *before* the write loop, so no string work occurs inside the hot path.
- Unique subdirectories are collected and created in one batch (k `dir.create()` syscalls, where $k \leq n$).
- The field separator is resolved once at function entry.
- `as.data.table()` on an existing `data.table` is a reference-pass (no copy).

Error handling: Individual element failures emit a `warning` and are skipped; the remaining elements continue to be processed.

Value

An invisible named `character` vector of the absolute file paths written, with length equal to the number of successfully exported elements. The total count is accessible via `length()` on the return value.

Dependencies

Requires the **data.table** package.

See Also

`fwrite`

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 Directory Organization*

Description

Exports list-columns containing `data.frame` or `data.table` objects from a `data.frame`/`data.table` to `txt` or `csv` files, automatically constructing a hierarchical directory structure from non-nested columns. Exportable nested columns (those holding `data.frame`/`data.table` elements) are distinguished from non-exportable custom-object columns (e.g. `rsplit` from the rsample package); only the former are written to disk 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 at least one nested list-column. Must have one or more rows.
`group_cols`	Optional character vector of column names used to build the hierarchical output directory structure. When `NULL` (default), all non-nested columns are used automatically.
`nest_cols`	Optional character vector of nested column names to export. When `NULL` (default), all columns whose elements are `data.frame`/`data.table` objects are exported automatically; custom-object list-columns are reported and skipped. Specifying a non-`data.frame` column triggers a warning and that column is skipped.
`export_path`	Single character string specifying the root export directory. Defaults to `tempdir()`. Created recursively if it does not exist.
`file_type`	Either `"txt"` (tab-separated, default) or `"csv"` (comma-separated). Case-insensitive.

Details

Nested column classification (mutually exclusive):

- *Exportable* — every element inherits from `data.frame` or `data.table`.
- *Non-exportable* — empty lists or elements of any other class (e.g. `rsplit`, `vfold_split`). Reported to the console; never written.

Directory layout: `export_path / <group1_value> / <group2_value> / <nest_col_name>.<file_type>`

Performance notes:

- Row data is accessed via `.subset2()` (zero-copy column access) rather than `nest_dt[i]`, eliminating per-row `data.table` allocation in the hot loop.
- All `n` output directory paths are pre-computed in a single vectorised `do.call(file.path, ...)` call before the loop; only the `k` unique paths are then passed to `dir.create()`, replacing `n` syscalls with `k` (`k <= n`; often `k << n` when many rows share the same group).
- The field separator and output filenames are computed once before the loop.
- `seq_len()` is used instead of `1:n` to avoid the `1:0` edge-case bug.
- All list-column introspection uses `vapply` with explicit `FUN.VALUE` to guarantee return types and prevent silent coercion.

Value

An invisible `integer` giving the total number of files successfully written. Returns `0L` when no exportable columns are found or all nested data are empty/`NULL`.

Dependencies

Requires the **data.table** package for data manipulation and file I/O (`fwrite`).

See Also

`fwrite`

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
```

`export_xlsx` *Export Data to XLSX Files*

Description

The natural complement to `import_xlsx()`. Takes a combined data object (the kind produced by `import_xlsx()` with `rbind = TRUE`) and writes it back to disk. The output **destination** is decided by a single `path` argument, and **worksheet splitting follows the data automatically** — there are no separate "modes" to choose:

- `path` **is a directory** (no `.xlsx` extension) — write *one file per* `file_col` *value* into that directory, with one worksheet per `sheet_col` value inside each file. This is the transparent round-trip of `import_xlsx()`.
- `path` **is a file** (ends in `.xlsx`) — write *everything into a single workbook*. Worksheets are named `"<file_col>_<sheet_col>"` (or just `"<sheet_col>"` / `"<file_col>"` when only one tracking column is present), and plain data without tracking columns becomes a single sheet.

Columns injected by `import_xlsx()` (`file_col`, `sheet_col`) are stripped from the output by default so the exported sheets are identical to the originals. Plain `data.frames` without any tracking columns (e.g. `mtcars`) are supported and are written as a single sheet — but only to a file `path`, since there is nothing to split files by.

Usage

```
export_xlsx(
  data,
  path,
  file_col = "excel_name",
```

```
  sheet_col = "sheet_name",
  sheet_name = "Sheet1",
  drop_cols = TRUE,
  overwrite = TRUE,
  verbose = FALSE
)
```

Arguments

data	A `data.frame`, `data.table`, or `tibble` to export. Tracking columns (`file_col` / `sheet_col`) are optional.
path	`character(1)`. The output destination. If it ends in `.xlsx` (case-insensitive) it is treated as a **single workbook**; otherwise it is treated as an **output directory** and one file is written per `file_col` value. Parent directories are created recursively as needed.
file_col	`character(1)`. Name of the column identifying the source file. Default `"excel_name"`.
sheet_col	`character(1)`. Name of the column identifying the source sheet. Default `"sheet_name"`.
sheet_name	`character(1)`. Worksheet tab name used when `data` has neither tracking column (single-sheet fallback). Default `"Sheet1"`.
drop_cols	`logical(1)`. When `TRUE` (default) the tracking columns are removed from every exported sheet so the round-trip is transparent.
overwrite	`logical(1)`. Allow overwriting existing files. Default `TRUE`.
verbose	`logical(1)`. Print a message for every sheet/file written. Default `FALSE`.

Details

Why writexl?: **writexl** writes `.xlsx` via a minimal C library with no Java or Perl dependency. It is faster than **openxlsx** for plain export and produces smaller files, at the cost of no cell formatting, formulas, or styles. For those, use **openxlsx / openxlsx2**.

Sheet-name sanitisation: Excel sheet names are limited to 31 characters and may not contain `[ ] * ? / \ :`. Both constraints are enforced automatically.

Directory vs. file dispatch: The single `path` argument is classified purely by its extension: a trailing `.xlsx` means "one workbook", anything else means "a directory of files". If you want a directory whose name happens to end in `.xlsx`, append a trailing slash, or pass an explicit file name.

Value

Invisibly, a named `character` vector of written file paths: named by `file_col` value in directory mode, or by `path` in single-workbook mode.

Examples

```
# Example: Excel file export demonstrations
# Example 1: Export a plain data.frame to a single workbook
out_file <- file.path(tempdir(), "test.xlsx")
export_xlsx(
  mtcars,                           # Data to export (no tracking columns)
  path       = out_file,            # Ends in .xlsx -> one workbook
  sheet_name = "test"               # Worksheet tab name for the single sheet
)
# Clean up the generated file
file.remove(out_file)

# Example 2: Split into one file per group
out_files <- export_xlsx(
  iris,                             # Data to export
  path      = tempdir(),            # A directory -> one file per file_col value
  file_col  = "Species",            # Column whose values name the output files
  drop_cols = FALSE                 # Keep the Species column in each output file
)
# Clean up the generated files (export_xlsx returns the written paths)
file.remove(out_files)
```

`format_digits` *Format Numeric Columns to Fixed-Decimal Character Strings*

Description

Format Numeric Columns to Fixed-Decimal Character Strings

Usage

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

Arguments

`data`	A data.frame or data.table. The input dataset.
`cols`	A character or integer vector specifying columns to format. If NULL (default), all numeric columns are formatted.
`digits`	A non-negative integer specifying decimal places. Defaults to 2.
`percentage`	Logical. If TRUE, values are multiplied by 100 and a "%" sign is appended. Defaults to FALSE.
`nan_as_na`	Logical. If TRUE, NaN is treated identically to NA and coerced to NA_character_. If FALSE (default), NaN is preserved as the string "NaN".

Details

The function processes columns in the following order:

1. Validates all input parameters with informative error messages.
2. Copies the input only once: data.table inputs are deep-copied via `copy()`; data.frame inputs are copied implicitly by `as.data.table()`, avoiding a redundant second copy.
3. Resolves `cols` to a character vector of valid numeric column names, warning and skipping any non-numeric columns specified.
4. Applies a vectorised formatting function via `lapply(.SD, fn)` and `:=`, so all target columns are dispatched in a single data.table call rather than a column-by-column loop.

NA and NaN handling:

- `NA_real_` is always returned as `NA_character_`.
- `NaN` is returned as `"NaN"` by default. Set `nan_as_na = TRUE` to coerce it to `NA_character_` instead.

Rounding uses explicit `round()` before `sprintf()` to guarantee consistent results across platforms (Windows, Linux, macOS), where the underlying C library's rounding behaviour may otherwise differ.

Value

A data.table with the specified numeric columns formatted as character strings. The original object is never modified.

Note

- `data` must be a data.frame or data.table.
- Integer column indices in `cols` are converted to column names internally; duplicates are silently removed.
- `digits` accepts numeric values such as `2.0` and coerces them to integer; non-integer-valued numbers raise an error.
- The function depends only on `data.table` and base R.

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_path_info` *Extract Path Segments or Filenames from File Paths*

Description

`get_path_info` is a merged, upgraded replacement for `get_path_segment` and `get_filename`. It operates in two modes:

- **Mode A (when `n` is specified)**: Extract a specific path segment by position, supporting forward indexing, reverse indexing, and range extraction.
- **Mode B (when `n = NULL`)**: Extract the filename, with optional removal of the file extension and/or the directory prefix.

Usage

```
get_path_info(paths, n = NULL, rm_extension = TRUE, rm_path = TRUE)
```

Arguments

`paths`	A `character` vector of file system paths. Supports mixed separators (`/` and `\\`) and Windows drive letters (e.g. `C:`).
`n`	A `numeric` segment index. Defaults to `NULL` (enters filename mode). • Positive integer: forward index from the path start; `1` = first segment. • Negative integer: reverse index from the path end; `-1` = last segment (i.e. the filename segment). • Length-2 vector: extract a contiguous range, e.g. `c(2, 4)` or `c(-3, -1)`. • `0` is not allowed.
`rm_extension`	A `logical(1)` flag controlling extension removal. Defaults to `TRUE`. • In Mode B (`n = NULL`): always applied. • In Mode A: **only applied when** `n == -1` (explicitly targeting the filename segment). Has no effect for intermediate directory segments (e.g. `n = 2`).
`rm_path`	A `logical(1)` flag controlling whether the directory prefix is stripped, keeping only the filename. Defaults to `TRUE`. Only applies in Mode B (`n = NULL`); ignored when `n` is specified.

Details

Path normalisation (internal, fully vectorised):

1. All backslashes and consecutive slashes are collapsed to a single `/`.
2. Windows drive letter prefixes (`C:`, `D:`, etc.) are stripped.
3. Leading and trailing `/` characters are removed.
4. Paths that are empty after the above steps (e.g. original inputs `"C:/"`, `"/"`, `""`) are coerced to `NA_character_`.

Extension-stripping behaviour (internal `.strip_ext` helper):

Input	Output	Notes
`"report.txt"`	`"report"`	Standard file — last extension removed
`"data.tar.gz"`	`"data.tar"`	Compound extension — only last level removed
`".bashrc"`	`".bashrc"`	Pure dot-file (no second dot) — unchanged
`".report.xlsx"`	`".report"`	Dot-file with extension — extension removed
`"no_ext"`	`"no_ext"`	No extension — returned as-is
`"file."`	`"file."`	Trailing isolated dot — returned as-is

NA safety: `strsplit(NA_character_, ...)` returns `list(NA)` with `length` 1, not `character(0)`. Consequently, every `vapply` callback guards against NA paths with an explicit `anyNA(x)` check rather than `length(x) == 0`.

Value

A `character` vector of the same length as `paths`:

- Returns the extracted segment string when the segment exists.
- Returns `NA_character_` when the segment index exceeds the path depth, the input element is `NA`, or the path reduces to empty after normalisation (e.g. `"C:/"`, `"/"`).

See Also

`base::basename()`, `tools::file_path_sans_ext()`

Examples

```
paths <- c("C:/Users/foo/Documents/report.xlsx",
           "/home/user/.bashrc",
           "relative/path/to/data.csv",
           ".hidden.tar.gz",
           NA_character_)

# Mode B: filename only, extension stripped (default)
get_path_info(paths)

# Mode B: filename only, extension preserved
get_path_info(paths, rm_extension = FALSE)
```

```
# Mode B: full normalised path, extension stripped
get_path_info(paths, rm_path = FALSE)

# Mode A: extract the 2nd path segment
get_path_info(paths, n = 2)

# Mode A: extract the last segment with extension stripped (n = -1 linkage)
get_path_info(paths, n = -1, rm_extension = TRUE)

# Mode A: range extraction
get_path_info(paths, n = c(2, 3))
```

`import_csv` *Flexible CSV/TXT File Import via data.table*

Description

Reads one or more CSV/TXT files using `fread` as the backend. Supports flexible combination strategies and source-file tracking. All return values are `data.table` objects.

Usage

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

Arguments

`file`	A non-empty `character` vector of file paths to CSV/TXT files. All paths must point to existing, accessible files.
`rbind`	A `logical` scalar controlling the combination strategy: • `TRUE` (default): Combine all files into a single `data.table`. • `FALSE`: Return a named `list` of individual `data.table` objects.
`rbind_label`	A `character` scalar or `NULL` specifying the source-tracking column name (default: `"_file"`). Set to `NULL` to suppress the source column. Only applies when `rbind = TRUE`.
`full_path`	A `logical` scalar controlling path representation in labels: • `FALSE` (default): Use only the filename (via `basename()`). • `TRUE`: Use the full file path.
`keep_ext`	A `logical` scalar controlling whether the file extension is retained in labels:

- FALSE (default): Strip the file extension (e.g., "data").
- TRUE: Retain the file extension (e.g., "data.csv").

... Additional arguments passed directly to fread (e.g., select, drop, na.strings, skip, nThread).

Details

Label generation is controlled by the combination of full_path and keep_ext:

full_path = FALSE, keep_ext = FALSE	Filename without extension: "data"
full_path = FALSE, keep_ext = TRUE	Filename with extension: "data.csv"
full_path = TRUE, keep_ext = FALSE	Full path without extension: "/path/to/data"
full_path = TRUE, keep_ext = TRUE	Full path with extension: "/path/to/data.csv"

When rbind = TRUE and rbind_label is not NULL, rbindlist is called with idcol = rbind_label, which generates the source column directly during the merge step without any intermediate copies.

Value

- rbind = TRUE: A single data.table containing all imported rows. If rbind_label is not NULL, the first column contains the source file label for each row.
- rbind = FALSE: A named list of data.table objects. List names are derived from file paths according to full_path and keep_ext settings.

Note

- All specified files must exist and be readable at call time.
- rbind = TRUE assumes compatible column structures across files; mismatched columns are automatically aligned via fill = TRUE.
- Logical parameters (rbind, full_path, keep_ext) reject NA values explicitly.

See Also

fread, rbindlist

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
  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
)
```

`import_xlsx` *Import Data from XLSX Files*

Description

A high-performance function for importing data from one or multiple Excel files into `data.table` format, with fine-grained control over source tracking columns, sheet selection, row skipping, and optional parallel reading across (file, sheet) pairs.

Performance characteristics:

- `excel_sheets()` called exactly once per file (cached).
- The real cost of an Excel import is `read_excel()` parsing, which is single-threaded C++ and *not* affected by the `data.table` thread pool. The flat (file x sheet) task list is therefore read in parallel **across processes** when `workers > 1`: a fork pool on Unix/macOS, a PSOCK cluster on Windows. The cluster / fork pool is always torn down on exit.
- `setDT()` converts tibbles in-place — zero vector copies.
- Tracking columns injected via `:=` on small per-sheet tables *before* the single final `rbindlist`.
- The final `rbindlist` uses the ambient `data.table` thread pool; tune it globally with `setDTthreads` if needed. Workers are pinned to a single thread during the read phase so parallel processes do not oversubscribe cores.

Usage

```
import_xlsx(
  file,
  rbind = TRUE,
  sheet = NULL,
  skip = 0L,
  show_excel_name = TRUE,
  show_sheet_name = TRUE,
  workers = 1L,
  verbose = FALSE,
  ...
)
```

Arguments

`file`	Non-empty `character` vector of paths to existing `.xlsx` / `.xls` files.
`rbind`	`logical(1)`. `TRUE` (default) binds all sheets into a single `data.table`. `FALSE` returns a flat named list keyed as `"<filename>_<sheetname>"`.

sheet — Positive integer vector or NULL (default). NULL imports every sheet. Indices must be valid across all supplied files.

skip — Non-negative integer(1). Number of rows to skip before reading the header. Forwarded directly to read_excel. Default 0L.

show_excel_name — logical(1). When TRUE (default) and rbind = TRUE, prepends an excel_name column recording the source filename (extension stripped). Silently ignored when rbind = FALSE (provenance is already encoded in list-element names).

show_sheet_name — logical(1). When TRUE (default) and rbind = TRUE, prepends a sheet_name column recording the source sheet. Silently ignored when rbind = FALSE.

workers — integer(1). Number of parallel processes used to read the (file x sheet) tasks. Default 1L (serial). Values > 1 open a fork pool on Unix/macOS or a PSOCK cluster on Windows; the pool is capped at the number of tasks and always shut down on exit. Parallel reading pays off when files are large or numerous; for many tiny sheets the process / serialization overhead can dominate, so leave this at 1L unless the import is heavy.

verbose — logical(1). When TRUE, print one message() per sheet (source file, sheet name, and row x col dimensions, with empty sheets flagged) followed by a summary line. Default FALSE. Output is emitted from the master process, so it surfaces identically in serial and parallel modes.

... — Additional arguments forwarded to read_excel (e.g. col_types, na, trim_ws). **Do not** pass path, sheet, or skip here; use the dedicated parameters above.

Value

rbind = TRUE A data.table. Tracking columns excel_name and/or sheet_name are prepended when their respective show_* flags are TRUE.

rbind = FALSE A named list of data.tables, each element named "<filename>_<sheetname>". The list carries a "source_files" attribute with the original file paths.

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

nest_cv applies rsample::vfold_cv to each nested data frame within a data.table, returning an expanded result table containing the corresponding training and validation splits for each row.

Usage

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

Arguments

nest_dt	A data.frame or data.table containing at least one nested data.frame/data.table column.
v	Number of folds. Must be an integer >= 2. Default is 10.
repeats	Number of repeats. Must be an integer >= 1. Default is 1.
strata	A single character string specifying the stratification column name. Set to NULL for no stratification. Default is NULL.

breaks	Number of bins for stratifying a numeric variable. Only used when strata is non-NULL. Default is 4.
pool	Proportion threshold for pooling small strata. Only used when strata is non-NULL. Default is 0.1.
...	Additional arguments passed to rsample::vfold_cv.

Details

The function performs the following steps:

1. Validates that nest_dt is a non-empty data.frame or data.table with at least one nested column whose elements all inherit from data.frame.
2. Selects the target nested column: prefers a column named "data"; otherwise falls back to the first detected nested column.
3. When strata is specified, verifies that the column exists in every nested data frame before calling rsample::vfold_cv.
4. Iterates over each row, applies vfold_cv via do.call, expands the resulting splits into a data.table, and broadcasts the row's non-nested metadata columns across all CV rows.
5. Combines all per-row results with rbindlist in a single pass.

Value

A data.table with the following columns:

- All non-nested columns from nest_dt (broadcast across CV rows).
- splits — cross-validation split objects from rsample::vfold_cv.
- id (and id2 for repeated CV) — fold identifiers.
- train — list column of training data frames for each split.
- validate — list column of validation data frames for each split.

Note

- nest_dt must contain at least one nested column of data.frames or data.tables.
- as.data.table() is used instead of data.table::copy(): if the input is already a data.table, no copy is made.
- strata must be a column name present in **all** nested data frames.
- breaks and pool are forwarded to rsample::vfold_cv only when strata is non-NULL, avoiding invalid argument errors.
- The per-row loop with rbindlist corrects a silent bug in naive chained [approaches where all rows incorrectly shared the first row's CV splits.

See Also

- rsample::vfold_cv() — underlying cross-validation function
- rsample::training() — extract training set from a split
- rsample::testing() — extract test/validation set from a split

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. It smartly iterates through variables to perfectly preserve non-target contextual variables while utilizing native `dcast` for extreme performance.

Usage

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

Arguments

`data`	Input `data frame` or `data table`.
`rows2bind`	A character column name or numeric index to be used as row values.
`by`	A character vector or numeric vector of column indices to transform.
`nest_type`	Output nesting format (`"dt"` or `"df"`). Default `"dt"`.

Value

A nested `data.table` containing `name` and `data` columns, with all contextual features preserved inside the nested structures.

Examples

```
# Example: Row-to-pairs nesting with column names
r2p_nest(
  mtcars,
  rows2bind = "cyl",
  by = c("hp", "drat", "wt")
)
# 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 *Apply Cross-Validation to a List of Datasets*

Description

split_cv applies rsample::vfold_cv to each dataset in a named or unnamed list, returning a list of data.table objects that each contain the CV split objects alongside the corresponding training and validation sets.

Usage

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

Arguments

split_dt	A list whose every element is a data.frame or data.table. Must be non-empty.
v	Number of folds. Must be a single integer >= 2. Default is 10.
repeats	Number of repeats. Must be a single integer >= 1. Default is 1.
strata	A single character string naming the stratification column. The column must exist in **every** dataset. Set to NULL for no stratification. Default is NULL.
breaks	Number of bins when stratifying a numeric variable. Used only when strata is non-NULL. Default is 4.
pool	Proportion threshold for pooling small strata. Used only when strata is non-NULL. Default is 0.1.
...	Additional arguments forwarded to rsample::vfold_cv.

Details

For each dataset in split_dt the function:

1. Validates inputs once before entering the processing loop.
2. Builds a vfold_cv argument list, appending stratification parameters only when strata is non-NULL to avoid passing unsupported arguments to rsample.
3. Converts the rsample tibble to a data.table in a single as.data.table() call, preserving all fold-identifier columns (id, id2) without hard-coding on the value of repeats.
4. Appends train and validate list-columns by reference via :=.

Value

A list of data.table objects (one per input dataset), each containing:

- splits — rsample split objects.
- id — fold identifier (always present).
- id2 — repeat identifier (present only when repeats > 1).
- train — list-column of training data frames.
- validate — list-column of validation data frames.

The output list preserves the names of split_dt.

Note

- When strata is specified, it must exist in **all** datasets; a missing column raises an error rather than silently falling back to unstratified CV.
- breaks and pool are forwarded to rsample::vfold_cv only when strata is non-NULL, preventing invalid-argument errors.
- as.data.table() on an already-data.table input is a no-op (no copy is made).

See Also

- rsample::vfold_cv() — underlying cross-validation function
- rsample::training() — extract training set from a split
- rsample::testing() — extract validation set from a split
- nest_cv() — nested data.table variant of this utility

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 or Bottom Percentage of Data*

Description

Selects the top (largest) or bottom (smallest) percentage of data based on specified traits. Positive percentages extract the largest values; negative percentages extract the smallest values.

Usage

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

Arguments

data	A data.frame or data.table.
perc	A numeric vector strictly between -1 and 1 (excluding 0). Positive values (e.g., 0.05) select the top X% of largest values. Negative values (e.g., -0.1) select the bottom X% of smallest values.
trait	A character vector of column names to analyse.
by	A character vector of column names to group by. Default is NULL.
keep_data	Logical. If TRUE, returns a named list where each element contains both stat (summary statistics) and data (the subset rows). If FALSE (default), returns a single combined data.frame of statistics for all perc values.

Value

- keep_data = FALSE: a data.frame with one row per by / trait / perc combination, containing columns n, min, max, mean, median, sd, se, cv, selection.
- keep_data = TRUE: a named list (one element per perc value) where each element is a list with $stat and $data.

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
```

---

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

Description

w2l_nest reshapes a wide-format data.frame or data.table into long format, then nests the result by name (the pivoted column identifier) and any optional grouping variables supplied via by. Each row of the returned table contains a nested data.table or data.frame in the data list-column.

Usage

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

Arguments

data	data.frame or data.table. Wide-format input dataset. Converted in-place to data.table via setDT() if necessary (no copy).
cols2l	numeric or character. Columns to pivot from wide to long, specified as integer indices or column names. Default NULL: when NULL, by must be provided and the function performs a pure nest operation (no melting).
by	numeric or character. Additional grouping variables for hierarchical nesting, specified as integer indices or column names. Default NULL.
nest_type	character. Class of each nested subset: "dt" for data.table (default) or "df" for data.frame.

Details

Column resolution: both cols2l and by accept either integer column positions or character column names. Out-of-bounds indices and unknown names are caught early with informative error messages.

Overlap guard: if any column appears in both cols2l and by, the function stops with an error before attempting to melt, preventing silent structural corruption.

Factor-free melting: melt() is called with variable.factor = FALSE so the name column is always character, avoiding unexpected factor-level ordering in downstream grouping operations.

Memory efficiency:

- setDT() converts data.frame inputs by reference — no full copy.
- .SDcols restricts .SD to non-key columns, eliminating redundant storage of grouping keys inside each nested object.
- For nest_type = "df", setattr(copy(.SD), "class", "data.frame") modifies the class attribute on a shallow copy rather than performing a deep column-by-column duplication as as.data.frame() would.

Value

A data.table with one row per combination of name (and by levels, if provided). The data list-column holds the corresponding nested data.table or data.frame for each group. Grouping key columns are never duplicated inside the nested objects.

Note

- Passing an empty table (0 rows) triggers a warning() and returns an empty nested data.table immediately.
- cols2l and by must not overlap; overlapping columns will raise an error.
- nest_type values other than "dt" or "df" raise an error.

See Also

tidytable::nest_by() for a tidyverse-style equivalent.

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 a Named List*

Description

w2l_split reshapes a wide-format data.frame or data.table into long format, then splits the result into a named list keyed by the pivoted column identifier (variable) and any optional grouping variables supplied via by. List element names are derived directly from the grouping key combinations produced by split(), guaranteeing name-to-content alignment.

Usage

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

Arguments

data	data.frame or data.table. Wide-format input dataset. Converted in-place to data.table via setDT() if necessary (no copy).

cols2l	numeric or character. Columns to pivot from wide to long, specified as integer indices or column names. Default NULL: when NULL, by must be provided and the function splits the data as-is without melting.
by	numeric or character. Additional grouping variables used as secondary split keys, specified as integer indices or column names. Default NULL.
split_type	character. Class of each list element: "dt" for data.table (default) or "df" for data.frame.
sep	character. Separator used when concatenating multiple grouping key values into a single list-element name. Default "_".

Details

Name safety: list names are produced by data.table::split() itself using its by argument, not reconstructed from raw row order. This eliminates the name-to-content misalignment that arises when unique() on the original data and split()'s internal sort order diverge.

Column resolution: both cols2l and by accept integer column positions or character column names. Out-of-bounds indices and unknown names are caught early with informative error messages.

Overlap guard: columns appearing in both cols2l and by raise an error before melting to prevent id.vars / measure.vars conflicts.

Factor-free melting: melt() is called with variable.factor = FALSE so the variable column is always character, keeping split() sort order consistent with lexicographic expectations.

Memory efficiency:

- setDT() converts data.frame inputs by reference — no full copy.
- For split_type = "df", setattr(copy(x), "class", "data.frame") modifies the class on a shallow copy, avoiding the deep column-by-column duplication that as.data.frame() triggers.

Value

A named list of data.table or data.frame objects (controlled by split_type). Names reflect the key combination of variable (and by levels if provided), joined by sep.

- If by is NULL, the list is keyed by the pivoted column names only.
- If by is specified, the list is keyed by variable and all by level combinations.

Note

- An empty input table (0 rows) triggers a warning() and returns an empty list immediately.
- cols2l and by must not overlap; shared columns raise an error.
- split_type values other than "dt" or "df" raise an error.

See Also

tidytable::group_split() for a tidyverse-style equivalent.

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