Package: toolero (via r-universe)

June 5, 2026

Title A Toolkit for Research Workflows

Version 0.4.0

Description Provides utility functions to help researchers implement best practices for their coding projects. Includes tools for reading and cleaning data files, initializing R projects with a standard folder structure, creating 'Quarto' documents from reproducible templates with optional sample data and custom styling, detecting the execution context across interactive, 'Quarto', and script-based workflows, splitting data frames into group-level output files, and rendering syntactic tree diagrams as standalone PNG images via 'Typst'.

License MIT + file LICENSE

Depends R (>= 4.2.0)

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

Suggests knitr, pdftools, rmarkdown, spelling, testthat (>= 3.0.0)

Config/testthat/edition 3

Imports cli, fs, glue, janitor, lifecycle, purrr, quarto, readr, renv, rlang, rvest, tibble, tidyr, usethis, withr, xml2, yaml

URL https://github.com/erwinlares/toolero, https://erwinlares.github.io/toolero/

BugReports https://github.com/erwinlares/toolero/issues

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

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

Date/Publication 2026-06-05 19:08:01 UTC

RemoteUrl https://github.com/erwinlares/toolero

RemoteRef HEAD

RemoteSha 3e7fd7b21884354e4395ae288df2552fb519a4a1

Contents

`arborize` *Render a syntactic tree as a PNG image*

Description

Takes a syntactic tree and renders it using Quarto's Typst engine, exporting the result as a PNG image. Supports two rendering backends controlled by `tree_notation`:

Usage

```
arborize(
  tree,
  output = "syntactic-tree.png",
  dpi = 300,
  tree_notation = c("simple", "structured"),
  papersize = "a5",
  margin = "0.5cm",
  provenance = TRUE,
  overwrite = FALSE
)
```

Arguments

`tree`	A character string. For `tree_notation = "simple"`, a bracket notation string e.g. `"[S [NP] [VP]]"`. For `tree_notation = "structured"`, a lingotree `tree()` call string.
`output`	A character string. Path to the output PNG file. Defaults to `"syntactic-tree.png"` in the current working directory.
`dpi`	A numeric value. Resolution of the output PNG in dots per inch. Defaults to `300`. Use `600` for print-quality output.
`tree_notation`	A character string. One of `"simple"` (default) or `"structured"`. Controls which Typst rendering backend is used. See Details.

`papersize`	A character string. Typst paper size for the intermediate PDF. Defaults to `"a5"`. Increase to `"a4"` for very wide trees.
`margin`	A character string. Page margin for the intermediate PDF. Defaults to `"0.5cm"`. Reduce for tighter crops around the tree.
`provenance`	A logical. Whether to write a companion `.yaml` file recording the tree string and all rendering arguments alongside the PNG. Defaults to `TRUE`. The provenance file has the same name as the PNG but with a `.yaml` extension and lives in the same directory. Pass `FALSE` to suppress it.
`overwrite`	A logical. Whether to overwrite existing output files. When `TRUE`, overwrites both the PNG and the provenance file if they exist. Defaults to `FALSE`.

Details

- `"simple"` uses `@preview/syntree` and accepts a bracket notation string, e.g. `"[S [NP [Det the] [N cat]] [VP [V sat]]]"`. This is the most compact input format and suits basic linguistic trees.
- `"structured"` uses `@preview/lingotree` and accepts a nested `tree()` call string. This backend supports per-node styling, movement arrows, and multi-dominant trees.

The function is useful for producing standalone tree figures that can be embedded in any document format – LaTeX, Word, HTML, or presentations – without requiring a full LaTeX installation.

`arborize()` performs the following steps:

1. Validates inputs and resolves the Typst package from `tree_notation`.
2. Builds a minimal `.qmd` document via `.build_arborize_qmd()`.
3. Writes the document and renders it inside a self-cleaning temporary directory managed by `withr::with_tempdir()`.
4. Calls `quarto::quarto_render()` to produce an intermediate PDF via Typst.
5. Converts the PDF to PNG using `pdftools::pdf_convert()`.
6. Reads the PNG bytes into memory before the temporary directory is deleted, then writes them to the specified output path.
7. If `provenance = TRUE`, writes a companion `.yaml` file recording the tree string and all rendering arguments.

On first use, Typst will download the required package from the Typst package registry. This requires an internet connection. Subsequent renders use the locally cached package.

Requires Quarto 1.4 or later with Typst support, and the `pdftools` package for PDF-to-PNG conversion. Install `pdftools` with `install.packages("pdftools")`.

Value

Invisibly returns the path to the output PNG file.

References

syntree Typst package (v0.2.1): https://typst.app/universe/package/syntree

lingotree Typst package (v1.0.0): https://typst.app/universe/package/lingotree

Examples

```
## Not run:
# Simple bracket notation (default) -- also writes tree-1.yaml
arborize("[NP [Det the] [N cat]]", output = "my-trees/tree-1.png")

# Suppress provenance file
arborize("[NP [Det the] [N cat]]", provenance = FALSE)

# Wider tree with print-quality output
arborize(
  paste0(
    "[Aspectual Classes ",
    "[Statives [States]] ",
    "[Dynamic ",
    "[Atelic [Activities]] ",
    "[Telic ",
    "[Instantaneous [Achievements]] ",
    "[Durative [Accomplishments]]]]]"
  ),
  output    = "aspectual-classes.png",
  dpi       = 600,
  papersize = "a4"
)

# Structured notation using lingotree
arborize(
  "tree(
    tag: [VP],
    tree(
      tag: [DP],
      [every],
      [farmer]
    ),
    [smiled]
  )",
  tree_notation = "structured",
  output        = "vp-tree.png"
)

## End(Not run)
```

`check_project` *Check a project for toolero conventions*

Description

`check_project()` audits a project directory and reports whether it follows the structure and conventions that `init_project()` creates. It is useful both for projects initialized with `init_project()` and for existing projects that were created independently.

Usage

```
check_project(path = ".", error = TRUE)
```

Arguments

path	A character string with the path to the project directory. Defaults to "." (the current working directory).
error	Logical. If TRUE (the default), prints a formatted cli report and returns the results invisibly. If FALSE, returns a tibble with columns check, status, and message without printing.

Value

A tibble with columns check, status, and message. Returned invisibly when error = TRUE, visibly when error = FALSE.

Examples

```
# Audit the current working directory

check_project()


# Audit a specific project directory
## Not run:
check_project(path = "path/to/project")

## End(Not run)
```

---

create_qmd *Create a new Quarto document from a template*

Description

Creates a new Quarto document in the specified directory. Optionally copies a sample dataset and a worked analysis example, wires up custom CSS and header styling from a directory of assets, and scaffolds a post-render purl hook for extracting R code.

Usage

```
create_qmd(
  filename = NULL,
  path = ".",
  yaml_data = NULL,
  overwrite = FALSE,
  use_purl = TRUE,
  include_examples = TRUE,
  use_style = FALSE
)
```

Arguments

filename	A string or NULL. Name of the generated .qmd file. Must be supplied explicitly, e.g. "analysis.qmd".
path	A string. Path to the directory where the document will be created. Defaults to "." (the current working directory).
yaml_data	A string or NULL. Path to a YAML file containing metadata to pre-populate the document header. If NULL (the default), the template is copied as-is with placeholder prompts intact.
overwrite	A logical. Whether to overwrite existing files. Defaults to FALSE.
use_purl	Logical. If TRUE (the default), creates a _quarto.yml file with a post-render hook and a purl.R script inside R/ that extracts R code from the rendered document into a .R file. The target document is resolved dynamically by scanning the project root for .qmd files, so the same purl.R works regardless of the document name.
include_examples	Logical. If TRUE (the default), copies a sample dataset (sample.csv) into data-raw/, a placeholder logo (logo.png) into assets/, and uses a template .qmd pre-populated with a worked analysis example. The YAML header includes a params block referencing the sample data. If FALSE, creates a blank .qmd with only the YAML header and no example content, and skips copying the sample dataset and logo.
use_style	Logical or character. Controls whether custom CSS and header assets are wired into the YAML. • FALSE (the default): no custom styling. The YAML format: html: block contains only standard Quarto options. • TRUE: shorthand for "assets/". Scans path/assets/ for .css and .html files and adds them to the YAML. • A directory path (e.g. "my-branding/"): scans the given directory for .css and .html files and adds them to the YAML. If the directory contains exactly one .css file, it is added as css: in the YAML. If exactly one .html file is found, it is added as include-before-body:. If multiple .css or .html files are found, the function errors and asks the user to specify which file to use via yaml_data. If neither is found, a warning is issued.

Details

create_qmd() performs the following steps:

1. Validates that filename is supplied and path exists.
2. If include_examples = TRUE: creates data-raw/ under path and copies sample.csv there. Creates assets/ if needed and copies a placeholder logo.png. Uses the example template for the .qmd.
3. If include_examples = FALSE: uses the skeleton template for the .qmd. No sample data or logo is copied.
4. If use_style is TRUE or a directory path: scans the directory for .css and .html files and injects them into the YAML header.

5. If `yaml_data` is provided, reads the YAML file and substitutes values into the document header. This runs after style injection, so `yaml_data` can override any auto-generated YAML keys.
6. If `use_purl = TRUE`, writes `_quarto.yml` with a post-render hook and copies `purl.R` into `path/R/`.
7. The sample dataset bundled with the template is a subset of the Palmer Penguins dataset. Citation: Horst AM, Hill AP, Gorman KB (2020). palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data. R package version 0.1.0. doi:10.5281/zenodo.3960218

Note: `filename` has no default value and must always be supplied explicitly. Use `tempdir()` for temporary output during testing or exploration.

Value

Invisibly returns `path`.

Examples

```
# Minimal blank document -- no examples, no styling
create_qmd(path = tempdir(), filename = "analysis.qmd",
           include_examples = FALSE)

# Full worked example with sample data and placeholder logo
create_qmd(path = tempdir(), filename = "analysis.qmd",
           overwrite = TRUE)

# Blank document wired to UW branding assets (assumes assets/ exists)
create_qmd(path = tempdir(), filename = "report.qmd",
           include_examples = FALSE, use_style = TRUE,
           overwrite = TRUE)

# Blank document with custom branding from a different directory
create_qmd(path = tempdir(), filename = "report.qmd",
           include_examples = FALSE, use_style = "my-branding/",
           overwrite = TRUE, use_purl = FALSE)

# Pre-populated YAML overrides
yaml_file <- tempfile(fileext = ".yml")
writeLines("author:\n  - name: 'Your Name'", yaml_file)
create_qmd(path = tempdir(), filename = "analysis.qmd",
           yaml_data = yaml_file, overwrite = TRUE)
```

`detect_execution_context` *Detect the current execution context*

Description

Identifies which of three execution environments the code is currently running in: an interactive R session, a `quarto render` call, or a plain `Rscript` invocation. This is useful for writing code that behaves correctly across all three contexts, such as resolving input file paths in a portable way.

Usage

```
detect_execution_context(interactive_fn = interactive)
```

Arguments

`interactive_fn`	A function. Used to detect whether the session is interactive. Defaults to `base::interactive`. Override in tests to simulate different execution environments.

Details

Detection follows a priority order:

1. If `interactive()` is `TRUE`, returns `"interactive"`.
2. If the environment variable `QUARTO_DOCUMENT_PATH` is set and non-empty, returns `"quarto"`.
3. Otherwise, returns `"rscript"`.

Value

A character string, one of `"interactive"`, `"quarto"`, or `"rscript"`.

Examples

```
context <- detect_execution_context()

input_file <- switch(context,
  interactive = "data/sample.csv",
  quarto      = params$input_file,
  rscript     = commandArgs(trailingOnly = TRUE)[1]
)
```

`generate_kb_xml` *Generate a KB-importable XML file from a Quarto document*

Description

Takes a Quarto document and produces an XML file that is directly importable into a UW-Madison Knowledge Base (KB) article. The function re-renders the `.qmd` with `embed-resources: true` so all visual assets are self-contained, extracts the HTML body, and wraps it in the KB XML structure along with metadata drawn from the document's YAML header.

Usage

```
generate_kb_xml(html_path, qmd_path = NULL, output_dir = NULL)
```

Arguments

html_path	A string. Path to the rendered HTML file. Used to infer the output filename and, if qmd_path is NULL, the location of the source .qmd.
qmd_path	A string or NULL. Path to the source .qmd file. If NULL (the default), inferred by replacing the .html extension of html_path with .qmd.
output_dir	A string or NULL. Directory where the .xml file will be written. If NULL (the default), written to the same directory as html_path.

Details

generate_kb_xml() performs the following steps:

1. Validates that html_path exists.
2. Infers qmd_path from html_path if not supplied, then validates it.
3. Extracts title, description, and categories from the .qmd YAML header and maps them to kb_title, kb_summary, and kb_keywords.
4. Re-renders the .qmd in an isolated temporary directory with embed-resources: true so all CSS, images, and JS are self-contained. The data/ and assets/ folders are copied alongside the .qmd to ensure the render succeeds.
5. Extracts the <body> from the embedded HTML.
6. Escapes HTML entities in the body for XML compatibility, as required by the UW-Madison KB import format.
7. Builds the XML structure with kb_title, kb_keywords, kb_summary, and kb_body nodes.
8. Writes the .xml file to output_dir.

Temporary files are managed via withr::local_tempdir() and are automatically cleaned up when the function exits, even on error.

When importing the resulting XML into the KB, check the *Decode HTML entity in body content* option.

Value

Invisibly returns the path to the written .xml file.

Examples

```
# Infer qmd_path automatically, write XML alongside the HTML
# generate_kb_xml(html_path = "docs/analysis.html")

# Supply qmd_path explicitly and write to a specific output directory
# generate_kb_xml(
#   html_path  = "docs/analysis.html",
#   qmd_path   = "analysis.qmd",
```

```
#   output_dir = "exports"
# )
```

`init_project` *Initialize a new R project with a standard folder structure*

Description

`init_project()` creates a new R project at the given path with an opinionated folder structure suited for research workflows. It optionally initializes `renv` for package management and git for version control.

Usage

```
init_project(
  path,
  use_renv = TRUE,
  use_git = TRUE,
  extra_folders = NULL,
  open = FALSE,
  uw_branding = FALSE
)
```

Arguments

`path`	A character string with the path and name of the new project (e.g., `"~/Documents/my-project"`).
`use_renv`	Logical. If `TRUE`, initializes `renv` in the new project. Defaults to `TRUE`.
`use_git`	Logical. If `TRUE`, initializes a git repository in the new project. Defaults to `TRUE`.
`extra_folders`	A character vector of additional folder names to create inside the project. Defaults to `NULL`.
`open`	Logical. If `TRUE`, opens the new project in RStudio after creation. Defaults to `TRUE`.
`uw_branding`	Logical. If `TRUE`, creates an `assets/` folder and populates it with UW-Madison RCI branding files (`styles.css`, `header.html`, `rci-banner.png`). Defaults to `FALSE`.

Value

Called for its side effects. Does not return a value.

Examples

```
## Not run:
init_project(path = file.path(tempdir(), "project1"),
             use_renv = FALSE, use_git = FALSE)

init_project(path = file.path(tempdir(), "project2"),
             uw_branding = TRUE, use_renv = FALSE, use_git = FALSE)

init_project(path = file.path(tempdir(), "project3"),
             extra_folders = c("notebooks"),
             use_renv = FALSE, use_git = FALSE)

## End(Not run)
```

---

qmd_to_r *Extract R code from a Quarto document*

Description

`qmd_to_r()` extracts R code chunks from a `.qmd` file and writes them to a standalone `.R` script using `knitr::purl()`. It works on any `.qmd` file regardless of whether it was created with `create_qmd()`.

Usage

```
qmd_to_r(input, output = NULL, documentation = 1L, quiet = TRUE)
```

Arguments

`input`	A character string with the path to the `.qmd` file.
`output`	A character string with the path to the output `.R` file. If `NULL` (the default), the output file is written to the same directory as `input` with the `.qmd` extension replaced by `.R`.
`documentation`	An integer controlling how much documentation is included in the extracted script. Passed to `knitr::purl()`: `0` strips all documentation; `1` (the default) includes chunk labels as comments; `2` includes full roxygen blocks.
`quiet`	Logical. If `TRUE` (the default), suppresses knitr's own output. toolero provides its own cli feedback instead.

Value

Invisibly returns the path to the output `.R` file.

Examples

```
# Extract R code from a qmd file
qmd <- tempfile(fileext = ".qmd")
writeLines(c(
  "---",
  "title: Analysis",
  "---",
  "",
  "```{r}",
  "x <- 1 + 1",
  "```"
), qmd)

# Default output path: same directory, .R extension
qmd_to_r(input = qmd)

# Explicit output path
out <- tempfile(fileext = ".R")
qmd_to_r(input = qmd, output = out)

# Strip all documentation
qmd_to_r(input = qmd, output = out, documentation = 0L)
```

---

read_clean_csv *Read and clean a CSV file*

Description

`read_clean_csv()` reads a CSV file, standardizes column names, optionally handles missing values, and optionally prints an ingest summary. It combines `readr::read_csv()`, `janitor::clean_names()`, and `tidyr::drop_na()` into a single, reproducibility-friendly step.

Usage

```
read_clean_csv(
  path,
  na = c("", "NA"),
  drop_na = FALSE,
  summary = FALSE,
  verbose = FALSE,
  ...
)
```

Arguments

`path`	A character string with the path to the CSV file.

na	A character vector of strings to treat as missing values. Passed directly to readr::read_csv(). Defaults to c("", "NA"), which matches readr's own default behavior.
drop_na	Logical or character vector. If FALSE (the default), no rows are dropped. If TRUE, drops all rows containing any missing value. If a character vector of column names, drops only rows with missing values in those columns. Always emits a cli message reporting how many rows were dropped and how many remain.
summary	Logical. If TRUE, prints a brief ingest summary after reading and cleaning: row and column counts, number of column names cleaned, and missing value totals. Reflects the final state of the tibble after any drop_na action. Defaults to FALSE.
verbose	Logical. If TRUE, displays column type messages from readr::read_csv(). Defaults to FALSE.
...	Additional arguments passed to readr::read_csv(), such as col_types, skip, or locale.

Value

A tibble with cleaned column names.

Examples

```
sample_path <- system.file("templates", "sample.csv", package = "toolero")

# Basic usage
data <- read_clean_csv(sample_path)

# Explicit missing-value codes
data <- read_clean_csv(sample_path, na = c("", "NA", "N/A", ".", "-999"))

# Drop rows missing in any column
data <- read_clean_csv(sample_path, drop_na = TRUE)

# Drop rows missing in specific columns
data <- read_clean_csv(sample_path, drop_na = c("bill_length_mm", "sex"))

# Print ingest summary
data <- read_clean_csv(sample_path, summary = TRUE)

# Combine arguments
data <- read_clean_csv(
  sample_path,
  na      = c("", "NA", "N/A", "."),
  drop_na = TRUE,
  summary = TRUE
)
```

write_by_group *Split a data frame by a grouping column and write each group to a CSV file*

Description

Splits a data frame by a single grouping column and writes each group to a separate CSV file. Optionally writes a manifest file listing the output files, their group values, and row counts.

Usage

```
write_by_group(data, group_col, output_dir = NULL, manifest = FALSE)
```

Arguments

data	A data frame or tibble to split and save.
group_col	A string. The name of the column to group by.
output_dir	A string or NULL. Path to the directory where output files will be written. Created if it does not exist. If NULL, the user must supply a path explicitly.
manifest	A logical. Whether to write a manifest.csv file to output_dir listing the output files, group values, and row counts. Defaults to FALSE.

Details

Output filenames are derived from the group values of group_col. Values are sanitized before use as filenames: converted to lowercase, spaces and special characters replaced with -, consecutive dashes collapsed, and leading/trailing dashes stripped.

If manifest = TRUE, a manifest.csv is written to output_dir containing three columns: group_value, n_rows, and file_path.

Note: output_dir has no default value. Always supply an explicit path to avoid writing files to unexpected locations. Use tempdir() for temporary output during testing or exploration.

Value

Invisibly returns output_dir.

Examples

```
# Split a small data frame by group and write to a temp directory
data <- data.frame(
  species = c("Adelie", "Adelie", "Gentoo"),
  mass    = c(3750, 3800, 5000)
)
write_by_group(data, group_col = "species", output_dir = tempdir())

# Same but also write a manifest
write_by_group(data, group_col = "species",
```

```
                output_dir = tempdir(), manifest = TRUE)
```

`write_clean_csv` *Write a cleaned data frame to a CSV file*

Description

`write_clean_csv()` writes a data frame to a CSV file using `readr::write_csv()` and emits a cli confirmation message reporting the number of rows and columns written. It is the natural counterpart to `read_clean_csv()`, reinforcing the convention that `data-raw/` holds original inputs and `data/` holds cleaned, analysis-ready outputs.

Usage

```
write_clean_csv(data, path, overwrite = FALSE, ...)
```

Arguments

`data`	A data frame or tibble to write.
`path`	A character string with the path to the output CSV file.
`overwrite`	Logical. If `FALSE` (the default), errors if the file already exists. Set to `TRUE` to overwrite an existing file.
`...`	Additional arguments passed to `readr::write_csv()`, such as `append`, `col_names`, or `quote`.

Details

If column names are not already clean, `write_clean_csv()` applies `janitor::clean_names()` before writing and emits a warning listing the affected columns.

Value

Invisibly returns `path`.

Examples

```
sample_path <- system.file("templates", "sample.csv", package = "toolero")
data <- read_clean_csv(sample_path)

# Write to a temp file
out <- tempfile(fileext = ".csv")
write_clean_csv(data, out)

# Overwrite an existing file
write_clean_csv(data, out, overwrite = TRUE)

# Dirty names are cleaned automatically with a warning
```

```
dirty <- data.frame("First Name" = "Jane", "Last Name" = "Doe",
                    check.names = FALSE)
write_clean_csv(dirty, tempfile(fileext = ".csv"))
```

Index